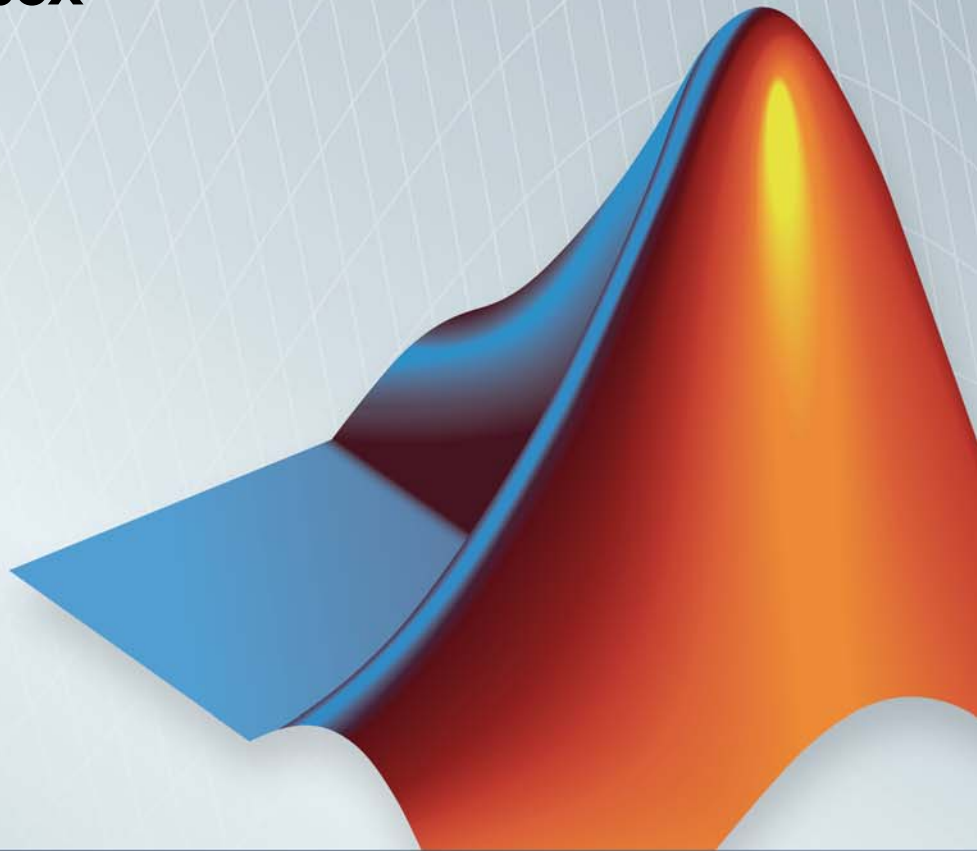


# Statistics Toolbox™

## User's Guide

R2013b



# MATLAB®

## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

For contact information about worldwide offices, see the MathWorks Web site.

*Statistics Toolbox™ User's Guide*

© COPYRIGHT 1993–2013 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

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

## Revision History

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



## Getting Started

### 1

<b>Statistics Toolbox Product Description</b> .....	1-2
Key Features .....	1-2

## Organizing Data

### 2

<b>Create Nominal and Ordinal Arrays</b> .....	2-3
Create Nominal Arrays .....	2-3
Create Ordinal Arrays .....	2-5
<b>Change Category Labels</b> .....	2-8
<b>Reorder Category Levels</b> .....	2-10
Reorder Category Levels in Ordinal Arrays .....	2-10
Reorder Category Levels in Nominal Arrays .....	2-11
<b>Categorize Numeric Data</b> .....	2-15
<b>Merge Category Levels</b> .....	2-18
<b>Add and Drop Category Levels</b> .....	2-20
<b>Plot Data Grouped by Category</b> .....	2-24
<b>Test Differences Between Category Means</b> .....	2-28
<b>Summary Statistics Grouped by Category</b> .....	2-37

<b>Sort Ordinal Arrays</b> .....	<b>2-39</b>
<b>Categorical Arrays</b> .....	<b>2-41</b>
What Are Categorical Arrays? .....	2-41
Categorical Array Conversion .....	2-41
<b>Advantages of Using Categorical Arrays</b> .....	<b>2-43</b>
Manipulate Category Levels .....	2-43
Analysis Using Categorical Arrays .....	2-43
Reduce Memory Requirements .....	2-44
<b>Index and Search Using Categorical Arrays</b> .....	<b>2-46</b>
Index By Category .....	2-46
Common Indexing and Searching .....	2-46
<b>Grouping Variables</b> .....	<b>2-51</b>
What Are Grouping Variables? .....	2-51
Group Definition .....	2-52
Analysis Using Grouping Variables .....	2-52
Missing Group Values .....	2-53
<b>Dummy Indicator Variables</b> .....	<b>2-55</b>
What Are Dummy Variables? .....	2-55
Creating Dummy Variables .....	2-56
<b>Regression with Categorical Covariates</b> .....	<b>2-59</b>
<b>Create a Dataset Array from Workspace Variables</b> ....	<b>2-65</b>
Create a Dataset Array from a Numeric Array .....	2-65
Create Dataset Array from Heterogeneous Workspace Variables .....	2-68
<b>Create a Dataset Array from a File</b> .....	<b>2-71</b>
Create a Dataset Array from a Tab-Delimited Text File ..	2-71
Create a Dataset Array from a Comma-Separated Text File .....	2-74
Create a Dataset Array from an Excel File .....	2-77
<b>Add and Delete Observations</b> .....	<b>2-79</b>

<b>Add and Delete Variables</b> .....	<b>2-83</b>
<b>Access Data in Dataset Array Variables</b> .....	<b>2-87</b>
<b>Select Subsets of Observations</b> .....	<b>2-93</b>
<b>Sort Observations in Dataset Arrays</b> .....	<b>2-97</b>
<b>Merge Dataset Arrays</b> .....	<b>2-101</b>
<b>Stack or Unstack Dataset Arrays</b> .....	<b>2-106</b>
<b>Calculations on Dataset Arrays</b> .....	<b>2-111</b>
<b>Export Dataset Arrays</b> .....	<b>2-115</b>
<b>Clean Messy and Missing Data</b> .....	<b>2-117</b>
<b>Dataset Arrays in the Variables Editor</b> .....	<b>2-122</b>
Open Dataset Arrays in the Variables Editor .....	<b>2-122</b>
Modify Variable and Observation Names .....	<b>2-123</b>
Reorder or Delete Variables .....	<b>2-125</b>
Add New Data .....	<b>2-127</b>
Sort Observations .....	<b>2-129</b>
Select a Subset of Data .....	<b>2-130</b>
Create Plots .....	<b>2-133</b>
<b>Dataset Arrays</b> .....	<b>2-135</b>
What Are Dataset Arrays? .....	<b>2-135</b>
Dataset Array Conversion .....	<b>2-135</b>
Dataset Array Properties .....	<b>2-136</b>
<b>Index and Search Dataset Arrays</b> .....	<b>2-138</b>
Ways To Index and Search .....	<b>2-138</b>
Common Indexing and Searching .....	<b>2-138</b>

## Descriptive Statistics

### 3

<b>Introduction to Descriptive Statistics</b> .....	3-2
<b>Measures of Central Tendency</b> .....	3-3
<b>Measures of Dispersion</b> .....	3-5
<b>Quantiles and Percentiles</b> .....	3-7
<b>Exploratory Analysis of Data</b> .....	3-11
<b>Resampling Statistics</b> .....	3-16
Bootstrap .....	3-16
Jackknife .....	3-19
Parallel Computing Support for Resampling Methods .....	3-20
<b>Data with Missing Values</b> .....	3-21

## Statistical Visualization

### 4

<b>Introduction to Statistical Visualization</b> .....	4-2
<b>Create Scatter Plots Using Grouped Data</b> .....	4-3
<b>Box Plots</b> .....	4-7
<b>Distribution Plots</b> .....	4-9
Normal Probability Plots .....	4-9
Quantile-Quantile Plots .....	4-12
Cumulative Distribution Plots .....	4-15
Other Probability Plots .....	4-16



<b>Working with Probability Distributions</b> .....	5-3
Types of Probability Distributions .....	5-3
How to Work with Probability Distributions .....	5-4
<b>Maximum Likelihood Estimation</b> .....	5-5
<b>Negative Loglikelihood Functions</b> .....	5-8
<b>Random Number Generation</b> .....	5-11
<b>Nonparametric and Empirical Probability Distributions</b> .....	5-14
Overview .....	5-14
Kernel Distribution .....	5-14
Piecewise Linear Distribution .....	5-14
Empirical Cumulative Distribution Function .....	5-15
Pareto Tails .....	5-15
Triangular Distribution .....	5-15
<b>Fit Kernel Distribution Object to Data</b> .....	5-16
<b>Fit Kernel Distribution Using ksdensity</b> .....	5-21
<b>Fit Kernel Distributions to Grouped Data Using ksdensity</b> .....	5-23
<b>Create and Plot Empirical Cumulative Distribution Functions</b> .....	5-26
<b>Fit a Nonparametric Distribution with Pareto Tails</b> ..	5-27
<b>Generate Random Numbers Using the Triangular Distribution</b> .....	5-32
<b>Explore the Probability Distribution Function UI</b> ....	5-37

<b>Model Data Using the Distribution Fitting App</b> .....	<b>5-40</b>
Opening the Distribution Fitting App .....	<b>5-40</b>
Creating and Managing Data Sets .....	<b>5-43</b>
Creating a New Fit .....	<b>5-48</b>
Displaying Results .....	<b>5-53</b>
Managing Fits .....	<b>5-55</b>
Evaluating Fits .....	<b>5-57</b>
Excluding Data .....	<b>5-61</b>
Saving and Loading Sessions .....	<b>5-67</b>
Example: Fitting a Distribution .....	<b>5-68</b>
Generating a File to Fit and Plot Distributions .....	<b>5-75</b>
<b>Custom Distributions Using the Distribution Fitting App</b> .....	<b>5-77</b>
Opening the Distribution Fitting App .....	<b>5-77</b>
Defining Custom Distributions .....	<b>5-78</b>
Importing Custom Distributions .....	<b>5-79</b>
<b>Explore the Random Number Generation UI</b> .....	<b>5-80</b>
<b>Compare Multiple Distribution Fits</b> .....	<b>5-83</b>
<b>Fit Probability Distribution Objects to Grouped Data</b> .....	<b>5-89</b>
<b>Multinomial Probability Distribution Objects</b> .....	<b>5-94</b>
<b>Multinomial Probability Distribution Functions</b> .....	<b>5-98</b>
<b>Generate Random Numbers Using Uniform Distribution Inversion</b> .....	<b>5-101</b>
<b>Use <math>t</math> Location-Scale to Represent Cauchy Distribution</b> .....	<b>5-104</b>
<b>Use Student's <math>t</math> to Generate Standard Cauchy Random Numbers</b> .....	<b>5-108</b>

<b>Probability Distributions Used for Multivariate</b>	
<b>Modeling</b> .....	<b>5-110</b>
Gaussian Mixture Models .....	<b>5-110</b>
Copulas: Generate Correlated Samples .....	<b>5-120</b>

## Random Number Generation

# 6

<b>Generating Random Data</b> .....	<b>6-2</b>
<b>Random Number Generation Functions</b> .....	<b>6-3</b>
<b>Common Generation Methods</b> .....	<b>6-5</b>
Direct Methods .....	<b>6-5</b>
Inversion Methods .....	<b>6-7</b>
Acceptance-Rejection Methods .....	<b>6-9</b>
<b>Representing Sampling Distributions Using Markov</b>	
<b>Chain Samplers</b> .....	<b>6-13</b>
Using the Metropolis-Hastings Algorithm .....	<b>6-13</b>
Using Slice Sampling .....	<b>6-14</b>
<b>Generating Quasi-Random Numbers</b> .....	<b>6-15</b>
Quasi-Random Sequences .....	<b>6-15</b>
Quasi-Random Point Sets .....	<b>6-16</b>
Quasi-Random Streams .....	<b>6-24</b>
<b>Generating Data Using Flexible Families of</b>	
<b>Distributions</b> .....	<b>6-26</b>
Pearson and Johnson Systems .....	<b>6-26</b>
Generating Data Using the Pearson System .....	<b>6-28</b>
Generating Data Using the Johnson System .....	<b>6-30</b>

## Hypothesis Tests

---

### 7

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

## Analysis of Variance

---

### 8

<b>Introduction to Analysis of Variance</b> .....	<b>8-2</b>
<b>ANOVA</b> .....	<b>8-3</b>
One-Way ANOVA .....	<b>8-3</b>
Two-Way ANOVA .....	<b>8-9</b>
N-Way ANOVA .....	<b>8-12</b>
Other ANOVA Models .....	<b>8-26</b>
Analysis of Covariance .....	<b>8-27</b>
Nonparametric Methods .....	<b>8-35</b>
<b>MANOVA</b> .....	<b>8-39</b>
Introduction to MANOVA .....	<b>8-39</b>
ANOVA with Multiple Responses .....	<b>8-39</b>

<b>Parametric Regression Analysis</b> .....	<b>9-3</b>
What Is Parametric Regression? .....	9-3
Choose a Regression Function .....	9-3
Update Legacy Code with New Fitting Methods .....	9-4
<b>What Are Linear Regression Models?</b> .....	<b>9-8</b>
<b>Linear Regression</b> .....	<b>9-12</b>
Prepare Data .....	9-12
Choose a Fitting Method .....	9-14
Choose a Model or Range of Models .....	9-15
Fit Model to Data .....	9-21
Examine Quality and Adjust the Fitted Model .....	9-21
Predict or Simulate Responses to New Data .....	9-40
Share Fitted Models .....	9-43
Linear Regression Workflow .....	9-44
<b>Regression Using Dataset Arrays</b> .....	<b>9-51</b>
<b>Linear Regression with Interaction Effects</b> .....	<b>9-54</b>
<b>Interpret Linear Regression Results</b> .....	<b>9-64</b>
<b>Cook's Distance</b> .....	<b>9-72</b>
Purpose .....	9-72
Definition .....	9-72
How To .....	9-73
Example .....	9-73
<b>Coefficient Standard Errors and Confidence Intervals</b> .....	<b>9-76</b>
Coefficient Covariance and Standard Errors .....	9-76
Compute Coefficient Covariance and Standard Errors ...	9-76
Coefficient Confidence Intervals .....	9-77
Compute Coefficient Confidence Intervals .....	9-78

<b>Coefficient of Determination (R-Squared)</b> .....	<b>9-80</b>
Purpose .....	9-80
Definition .....	9-80
How To .....	9-80
Example .....	9-81
<b>Delete-1 Statistics</b> .....	<b>9-83</b>
Delete-1 Change in Covariance (covratio) .....	9-83
CovRatio Example .....	9-84
Delete-1 Scaled Difference in Coefficient Estimates (Dfbetas) .....	9-86
Dfbetas Example .....	9-87
Delete-1 Scaled Change in Fitted Values (Dffits) .....	9-87
Dffits Example .....	9-88
Delete-1 Variance (S2_i) .....	9-90
Delete-1 Variance Example .....	9-91
<b>Durbin-Watson Test</b> .....	<b>9-93</b>
Purpose .....	9-93
Definition .....	9-93
How To .....	9-93
Example .....	9-93
<b>F-statistic and t-statistic</b> .....	<b>9-95</b>
F-statistic .....	9-95
F-statistic Example .....	9-96
t-statistic .....	9-98
t-statistic Example .....	9-99
<b>Hat Matrix and Leverage</b> .....	<b>9-102</b>
Hat Matrix .....	9-102
Leverage .....	9-103
Determine High Leverage Observations .....	9-104
<b>Residuals</b> .....	<b>9-107</b>
Purpose .....	9-107
Definition .....	9-107
How To .....	9-108
Example .....	9-108
<b>Summary of Output and Diagnostic Statistics</b> .....	<b>9-117</b>

<b>Stepwise Regression</b> .....	<b>9-119</b>
Stepwise Regression to Select Appropriate Models .....	<b>9-119</b>
Compare large and small stepwise models .....	<b>9-119</b>
<b>Robust Regression — Reduce Outlier Effects</b> .....	<b>9-125</b>
What Is Robust Regression? .....	<b>9-125</b>
Robust Regression versus Standard Least-Squares Fit ...	<b>9-125</b>
<b>Ridge Regression</b> .....	<b>9-128</b>
Introduction to Ridge Regression .....	<b>9-128</b>
Ridge Regression .....	<b>9-128</b>
<b>Lasso and Elastic Net</b> .....	<b>9-132</b>
What Are Lasso and Elastic Net? .....	<b>9-132</b>
Lasso Regularization .....	<b>9-132</b>
Lasso and Elastic Net with Cross Validation .....	<b>9-135</b>
Wide Data via Lasso and Parallel Computing .....	<b>9-138</b>
Lasso and Elastic Net Details .....	<b>9-144</b>
References .....	<b>9-145</b>
<b>Partial Least Squares</b> .....	<b>9-147</b>
Introduction to Partial Least Squares .....	<b>9-147</b>
Partial Least Squares .....	<b>9-148</b>
<b>Linear Mixed-Effects Models</b> .....	<b>9-153</b>
<b>Prepare Data for Linear Mixed-Effects Models</b> .....	<b>9-158</b>
Dataset Arrays .....	<b>9-158</b>
Design Matrices .....	<b>9-160</b>
Relation of Matrix Form to Dataset Arrays .....	<b>9-162</b>
<b>Relationship Between Formula and Design Matrices</b> ..	<b>9-164</b>
Formula .....	<b>9-164</b>
Design Matrices for Fixed and Random Effects .....	<b>9-166</b>
Grouping Variables .....	<b>9-169</b>
<b>Estimating Parameters in Linear Mixed-Effects</b>	
<b>Models</b> .....	<b>9-172</b>
Maximum Likelihood (ML) .....	<b>9-173</b>
Restricted Maximum Likelihood (REML) .....	<b>9-174</b>

<b>Multinomial Models for Nominal Responses</b> .....	10-2
<b>Multinomial Models for Ordinal Responses</b> .....	10-5
<b>Hierarchical Multinomial Models</b> .....	10-9
<b>Generalized Linear Models</b> .....	10-12
What Are Generalized Linear Models? .....	10-12
Prepare Data .....	10-13
Choose Generalized Linear Model and Link Function ....	10-15
Choose Fitting Method and Model .....	10-19
Fit Model to Data .....	10-24
Examine Quality and Adjust the Fitted Model .....	10-24
Predict or Simulate Responses to New Data .....	10-37
Share Fitted Models .....	10-40
Generalized Linear Model Workflow .....	10-42
<b>Lasso Regularization of Generalized Linear Models</b> ..	10-48
What is Generalized Linear Model Lasso Regularization? .....	10-48
Regularize Poisson Regression .....	10-48
Regularize Logistic Regression .....	10-52
Regularize Wide Data in Parallel .....	10-59
Generalized Linear Model Lasso and Elastic Net .....	10-65
References .....	10-67

## Nonlinear Regression

<b>Nonlinear Regression</b> .....	11-2
What Are Parametric Nonlinear Regression Models? ....	11-2
Prepare Data .....	11-3
Represent the Nonlinear Model .....	11-4
Choose Initial Vector $\beta_0$ .....	11-7
Fit Nonlinear Model to Data .....	11-7



Examine Quality and Adjust the Fitted Model .....	11-8
Predict or Simulate Responses to New Data .....	11-12
Nonlinear Regression Workflow .....	11-16
<b>Mixed-Effects Models .....</b>	<b>11-23</b>
Introduction to Mixed-Effects Models .....	11-23
Mixed-Effects Model Hierarchy .....	11-24
Specifying Mixed-Effects Models .....	11-25
Specifying Covariate Models .....	11-28
Choosing nlmeFit or nlmeFitsa .....	11-30
Using Output Functions with Mixed-Effects Models .....	11-33
Mixed-Effects Models Using nlmeFit and nlmeFitsa .....	11-38
Examining Residuals for Model Verification .....	11-55
<b>Pitfalls in Fitting Nonlinear Models by Transforming to Linearity .....</b>	<b>11-61</b>

## Survival Analysis

# 12

<b>What Is Survival Analysis? .....</b>	<b>12-2</b>
Introduction .....	12-2
Censoring .....	12-2
Data .....	12-3
Survivor Function .....	12-5
Hazard Function .....	12-7
<b>Kaplan-Meier Method .....</b>	<b>12-11</b>
<b>Hazard and Survivor Functions for Different Groups .....</b>	<b>12-19</b>
<b>Survivor Functions for Two Groups .....</b>	<b>12-26</b>
<b>Cox Proportional Hazards Regression .....</b>	<b>12-31</b>
<b>Cox Proportional Hazards Model for Censored Data ..</b>	<b>12-34</b>

<b>Introduction to Multivariate Methods</b> .....	13-2
<b>Multivariate Linear Regression</b> .....	13-3
Multivariate Linear Regression Model .....	13-3
Solving Multivariate Regression Problems .....	13-4
<b>Estimation of Multivariate Regression Models</b> .....	13-6
Least Squares Estimation .....	13-6
Maximum Likelihood Estimation .....	13-10
Missing Response Data .....	13-12
<b>Set Up Multivariate Regression Problems</b> .....	13-16
Response Matrix .....	13-16
Design Matrices .....	13-21
Common Multivariate Regression Problems .....	13-22
<b>Multivariate General Linear Model</b> .....	13-30
<b>Fixed Effects Panel Model with Concurrent Correlation</b> .....	13-35
<b>Longitudinal Analysis</b> .....	13-43
<b>Multidimensional Scaling</b> .....	13-50
Introduction to Multidimensional Scaling .....	13-50
Classical Multidimensional Scaling .....	13-50
Nonclassical Multidimensional Scaling .....	13-56
Nonmetric Multidimensional Scaling .....	13-58
<b>Procrustes Analysis</b> .....	13-62
Compare Landmark Data .....	13-62
Data Input .....	13-62
Preprocess Data for Accurate Results .....	13-63
Compare Handwritten Shapes .....	13-64
<b>Feature Selection</b> .....	13-71

Introduction to Feature Selection .....	13-71
Sequential Feature Selection .....	13-71
<b>Feature Transformation .....</b>	<b>13-76</b>
Introduction to Feature Transformation .....	13-76
Nonnegative Matrix Factorization .....	13-76
Principal Component Analysis (PCA) .....	13-80
Quality of Life in U.S. Cities .....	13-81
Factor Analysis .....	13-94
<b>Partial Least Squares Regression and Principal Components Regression .....</b>	<b>13-106</b>

## Cluster Analysis

# 14

<b>Introduction to Cluster Analysis .....</b>	<b>14-2</b>
<b>Hierarchical Clustering .....</b>	<b>14-3</b>
Introduction to Hierarchical Clustering .....	14-3
Algorithm Description .....	14-3
Similarity Measures .....	14-4
Linkages .....	14-6
Dendrograms .....	14-8
Verify the Cluster Tree .....	14-10
Create Clusters .....	14-16
<b>k-Means Clustering .....</b>	<b>14-22</b>
Introduction to k-Means Clustering .....	14-22
Create Clusters and Determine Separation .....	14-23
Determine the Correct Number of Clusters .....	14-25
Avoid Local Minima .....	14-28
<b>Gaussian Mixture Models .....</b>	<b>14-30</b>
Introduction to Gaussian Mixture Models .....	14-30
Cluster with Gaussian Mixtures .....	14-30

<b>Parametric Classification</b> .....	<b>15-2</b>
<b>Discriminant Analysis</b> .....	<b>15-3</b>
What Is Discriminant Analysis? .....	15-3
Create Discriminant Analysis Classifiers .....	15-3
Creating a Classifier Using	
ClassificationDiscriminant.fit .....	15-4
How the predict Method Classifies .....	15-6
Create and Visualize Discriminant Analysis Classifier ...	15-9
Improve a Discriminant Analysis Classifier .....	15-15
Regularize a Discriminant Analysis Classifier .....	15-23
Examine the Gaussian Mixture Assumption .....	15-29
Bibliography .....	15-37
 <b>Naive Bayes Classification</b> .....	 <b>15-38</b>
Supported Distributions .....	15-38
 <b>Performance Curves</b> .....	 <b>15-41</b>
Introduction to Performance Curves .....	15-41
What are ROC Curves? .....	15-41
Evaluate Classifier Performance Using perfcurve .....	15-41

**Nonparametric Supervised Learning**

<b>Supervised Learning (Machine Learning) Workflow</b>	
<b>and Algorithms</b> .....	<b>16-2</b>
Steps in Supervised Learning (Machine Learning) .....	16-2
Characteristics of Algorithms .....	16-7
 <b>Classification Using Nearest Neighbors</b> .....	 <b>16-9</b>
Pairwise Distance .....	16-9
k-Nearest Neighbor Search and Radius Search .....	16-12
K-Nearest Neighbor Classification for Supervised	
Learning .....	16-26

Construct a KNN Classifier .....	16-27
Examine the Quality of a KNN Classifier .....	16-27
Predict Classification Based on a KNN Classifier .....	16-28
Modify a KNN Classifier .....	16-29
<b>Classification Trees and Regression Trees .....</b>	<b>16-31</b>
What Are Classification Trees and Regression Trees? ....	16-31
Creating a Classification Tree .....	16-32
Creating a Regression Tree .....	16-32
Viewing a Classification or Regression Tree .....	16-33
How the Fit Methods Create Trees .....	16-36
Predicting Responses With Classification and Regression Trees .....	16-38
Improving Classification Trees and Regression Trees ....	16-39
Alternative: classregtree .....	16-48
<b>Splitting Categorical Predictors .....</b>	<b>16-56</b>
Challenges in Splitting Multilevel Predictors .....	16-56
Pull Left By Purity .....	16-57
Principle Component-Based Partitioning .....	16-57
One Versus All By Class .....	16-58
<b>Ensemble Methods .....</b>	<b>16-59</b>
Framework for Ensemble Learning .....	16-59
Basic Ensemble Examples .....	16-69
Test Ensemble Quality .....	16-71
Classification with Imbalanced Data .....	16-77
Classification: Imbalanced Data or Unequal Misclassification Costs .....	16-83
Classification with Many Categorical Levels .....	16-91
Surrogate Splits .....	16-95
LPBoost and TotalBoost for Small Ensembles .....	16-98
Ensemble Regularization .....	16-103
Tuning RobustBoost .....	16-115
Random Subspace Classification .....	16-119
TreeBagger Examples .....	16-125
Ensemble Algorithms .....	16-146
<b>Support Vector Machines (SVM) .....</b>	<b>16-162</b>
Understanding Support Vector Machines .....	16-162
Using Support Vector Machines .....	16-168
Nonlinear Classifier with Gaussian Kernel .....	16-170

SVM Classification with Cross Validation .....	16-174
<b>Bibliography</b> .....	16-184

## Markov Models

# 17

<b>Introduction to Markov Models</b> .....	17-2
<b>Markov Chains</b> .....	17-3
<b>Hidden Markov Models (HMM)</b> .....	17-5
Introduction to Hidden Markov Models (HMM) .....	17-5
Analyzing Hidden Markov Models .....	17-7

## Design of Experiments

# 18

<b>Design of Experiments</b> .....	18-2
<b>Full Factorial Designs</b> .....	18-3
Multilevel Designs .....	18-3
Two-Level Designs .....	18-4
<b>Fractional Factorial Designs</b> .....	18-5
Introduction to Fractional Factorial Designs .....	18-5
Plackett-Burman Designs .....	18-5
General Fractional Designs .....	18-6
<b>Response Surface Designs</b> .....	18-9
Introduction to Response Surface Designs .....	18-9
Central Composite Designs .....	18-9
Box-Behnken Designs .....	18-13

<b>D-Optimal Designs</b> .....	<b>18-15</b>
Introduction to D-Optimal Designs .....	<b>18-15</b>
Generate D-Optimal Designs .....	<b>18-16</b>
Augment D-Optimal Designs .....	<b>18-19</b>
Specify Fixed Covariate Factors .....	<b>18-20</b>
Specify Categorical Factors .....	<b>18-21</b>
Specify Candidate Sets .....	<b>18-21</b>

## Statistical Process Control

# 19

<b>Introduction to Statistical Process Control</b> .....	<b>19-2</b>
<b>Control Charts</b> .....	<b>19-3</b>
<b>Capability Studies</b> .....	<b>19-7</b>

## Parallel Statistics

# 20

<b>Quick Start Parallel Computing for Statistics</b>	
<b>Toolbox</b> .....	<b>20-2</b>
What Is Parallel Statistics Functionality? .....	<b>20-2</b>
How To Compute in Parallel .....	<b>20-3</b>
Parallel Treebagger .....	<b>20-4</b>
 <b>Concepts of Parallel Computing in Statistics</b>	
<b>Toolbox</b> .....	<b>20-6</b>
Subtleties in Parallel Computing .....	<b>20-6</b>
Vocabulary for Parallel Computation .....	<b>20-6</b>
 <b>When to Run Statistical Functions in Parallel</b> .....	<b>20-7</b>
Why Run in Parallel? .....	<b>20-7</b>
Factors Affecting Speed .....	<b>20-7</b>
Factors Affecting Results .....	<b>20-8</b>

<b>Working with parfor</b> .....	20-9
How Statistical Functions Use parfor .....	20-9
Characteristics of parfor .....	20-10
<b>Reproducibility in Parallel Statistical Computations</b> ..	20-12
Issues and Considerations in Reproducing Parallel Computations .....	20-12
Running Reproducible Parallel Computations .....	20-13
Parallel Statistical Computation Using Random Numbers .....	20-14
<b>Examples of Parallel Statistical Functions</b> .....	20-18
Parallel Jackknife .....	20-18
Parallel Cross Validation .....	20-19
Parallel Bootstrap .....	20-21

## Functions — Alphabetical List

**21** |

### Sample Data Sets

**A** |

### Distribution Reference

**B** |

<b>Bernoulli Distribution</b> .....	B-3
Overview .....	B-3
Parameters .....	B-3
Probability Mass Function .....	B-3
Descriptive Statistics .....	B-3
Relationship to Other Distributions .....	B-4
<b>Beta Distribution</b> .....	B-5



Definition .....	B-5
Background .....	B-5
Parameters .....	B-6
Example .....	B-7
<b>Binomial Distribution</b> .....	<b>B-9</b>
Overview .....	B-9
Parameters .....	B-9
Probability Density Function .....	B-9
Descriptive Statistics .....	B-10
Relationship to Other Distributions .....	B-10
Example .....	B-10
<b>Birnbaum-Saunders Distribution</b> .....	<b>B-13</b>
Definition .....	B-13
Background .....	B-13
Parameters .....	B-14
<b>Burr Type XII Distribution</b> .....	<b>B-15</b>
Definition .....	B-15
Background .....	B-16
Parameters .....	B-17
Fit a Burr Distribution and Draw the cdf .....	B-18
Compare Lognormal and Burr pdfs .....	B-20
Burr pdf for Various Parameters .....	B-22
Survival and Hazard Functions of Burr Distribution .....	B-23
Divergence of Parameter Estimates .....	B-25
<b>Chi-Square Distribution</b> .....	<b>B-28</b>
Definition .....	B-28
Background .....	B-28
Example .....	B-29
<b>Copulas</b> .....	<b>B-30</b>
<b>Custom Distributions</b> .....	<b>B-31</b>
<b>Exponential Distribution</b> .....	<b>B-32</b>
Definition .....	B-32
Background .....	B-32
Parameters .....	B-32

Example .....	B-33
<b>Extreme Value Distribution</b> .....	<b>B-35</b>
Definition .....	B-35
Background .....	B-35
Parameters .....	B-37
Example .....	B-38
<b>F Distribution</b> .....	<b>B-41</b>
Definition .....	B-41
Background .....	B-41
Example .....	B-42
<b>Gamma Distribution</b> .....	<b>B-43</b>
Definition .....	B-43
Background .....	B-43
Parameters .....	B-44
Example .....	B-45
<b>Gaussian Distribution</b> .....	<b>B-46</b>
<b>Gaussian Mixture Distributions</b> .....	<b>B-47</b>
<b>Generalized Extreme Value Distribution</b> .....	<b>B-48</b>
Definition .....	B-48
Background .....	B-48
Parameters .....	B-49
Example .....	B-50
<b>Generalized Pareto Distribution</b> .....	<b>B-53</b>
Definition .....	B-53
Background .....	B-53
Parameters .....	B-54
Example .....	B-55
<b>Geometric Distribution</b> .....	<b>B-57</b>
Overview .....	B-57
Parameters .....	B-57
Probability Mass Function .....	B-57
Descriptive Statistics .....	B-58

Relationship to Other Distributions .....	B-58
Example .....	B-58
<b>Hypergeometric Distribution</b> .....	<b>B-60</b>
Definition .....	B-60
Background .....	B-60
Examples .....	B-61
<b>Inverse Gaussian Distribution</b> .....	<b>B-63</b>
Definition .....	B-63
Background .....	B-63
Parameters .....	B-63
<b>Inverse Wishart Distribution</b> .....	<b>B-64</b>
Definition .....	B-64
Background .....	B-64
Example .....	B-64
See Also .....	B-65
<b>Johnson System</b> .....	<b>B-66</b>
<b>Kernel Distribution</b> .....	<b>B-67</b>
Overview .....	B-67
Kernel Density Estimator .....	B-67
Kernel Smoothing Function .....	B-67
Bandwidth .....	B-73
<b>Logistic Distribution</b> .....	<b>B-76</b>
Overview .....	B-76
Parameters .....	B-76
Probability Density Function .....	B-76
Relationship to Other Distributions .....	B-77
<b>Loglogistic Distribution</b> .....	<b>B-78</b>
Overview .....	B-78
Parameters .....	B-78
Probability Density Function .....	B-78
Relationship to Other Distributions .....	B-79
<b>Lognormal Distribution</b> .....	<b>B-80</b>

Overview .....	B-80
Parameters .....	B-80
Probability Density Function .....	B-80
Descriptive Statistics .....	B-81
Relationship to Other Distributions .....	B-81
Example .....	B-81
<b>Multinomial Distribution .....</b>	<b>B-83</b>
Overview .....	B-83
Parameter .....	B-83
Probability Density Function .....	B-83
Descriptive Statistics .....	B-84
Relationship to Other Distributions .....	B-84
<b>Multivariate Gaussian Distribution .....</b>	<b>B-85</b>
<b>Multivariate Normal Distribution .....</b>	<b>B-86</b>
Definition .....	B-86
Background .....	B-86
Example .....	B-87
<b>Multivariate <math>t</math> Distribution .....</b>	<b>B-91</b>
Definition .....	B-91
Background .....	B-91
Example .....	B-92
<b>Nakagami Distribution .....</b>	<b>B-97</b>
Definition .....	B-97
Background .....	B-97
Parameters .....	B-97
<b>Negative Binomial Distribution .....</b>	<b>B-98</b>
Definition .....	B-98
Background .....	B-98
Parameters .....	B-99
Example .....	B-101
<b>Noncentral Chi-Square Distribution .....</b>	<b>B-104</b>
Definition .....	B-104
Background .....	B-104
Example .....	B-105

<b>Noncentral F Distribution</b> .....	<b>B-106</b>
Definition .....	<b>B-106</b>
Background .....	<b>B-106</b>
Example .....	<b>B-107</b>
<b>Noncentral t Distribution</b> .....	<b>B-108</b>
Definition .....	<b>B-108</b>
Background .....	<b>B-108</b>
Example .....	<b>B-109</b>
<b>Normal Distribution</b> .....	<b>B-110</b>
Definition .....	<b>B-110</b>
Background .....	<b>B-110</b>
Parameters .....	<b>B-111</b>
Example .....	<b>B-112</b>
<b>Pareto Distribution</b> .....	<b>B-113</b>
<b>Pearson System</b> .....	<b>B-114</b>
<b>Piecewise Linear Distribution</b> .....	<b>B-115</b>
Overview .....	<b>B-115</b>
Parameters .....	<b>B-115</b>
Cumulative Distribution Function .....	<b>B-115</b>
Relationship to Other Distributions .....	<b>B-117</b>
<b>Poisson Distribution</b> .....	<b>B-118</b>
Definition .....	<b>B-118</b>
Background .....	<b>B-118</b>
Parameters .....	<b>B-119</b>
Examples .....	<b>B-119</b>
<b>Rayleigh Distribution</b> .....	<b>B-121</b>
Definition .....	<b>B-121</b>
Background .....	<b>B-121</b>
Parameters .....	<b>B-122</b>
Example .....	<b>B-122</b>
<b>Rician Distribution</b> .....	<b>B-123</b>
Definition .....	<b>B-123</b>

Background .....	B-123
Parameters .....	B-123
<b>Student's t Distribution</b> .....	<b>B-124</b>
Definition .....	B-124
Background .....	B-124
Example .....	B-125
<b>t Location-Scale Distribution</b> .....	<b>B-126</b>
Definition .....	B-126
Background .....	B-126
Parameters .....	B-126
<b>Triangular Distribution</b> .....	<b>B-127</b>
Overview .....	B-127
Parameters .....	B-127
Probability Density Function .....	B-128
Cumulative Distribution Function .....	B-129
Descriptive Statistics .....	B-131
<b>Uniform Distribution (Continuous)</b> .....	<b>B-133</b>
Overview .....	B-133
Parameters .....	B-133
Probability Density Function .....	B-134
Cumulative Distribution Function .....	B-135
Descriptive Statistics .....	B-137
Relationship to Other Distributions .....	B-138
<b>Uniform Distribution (Discrete)</b> .....	<b>B-139</b>
Definition .....	B-139
Background .....	B-139
Example .....	B-139
<b>Weibull Distribution</b> .....	<b>B-141</b>
Definition .....	B-141
Background .....	B-141
Parameters .....	B-142
Example .....	B-142
<b>Wishart Distribution</b> .....	<b>B-144</b>
Definition .....	B-144

Background ..... B-144  
Example ..... B-145  
See Also ..... B-145

**Bibliography**

**C**

**Index**





# Getting Started

---

## Statistics Toolbox Product Description

### Perform statistical modeling and analysis

Statistics Toolbox™ provides algorithms and tools for organizing, analyzing, and modeling data. You can use regression or classification for predictive modeling, generate random numbers for Monte Carlo simulations, use statistical plots for exploratory data analysis, and perform hypothesis tests.

For analyzing multidimensional data, Statistics Toolbox includes algorithms that let you identify key variables that impact your model with sequential feature selection, transform your data with principal component analysis, apply regularization and shrinkage, or use partial least-squares regression.

Statistics Toolbox includes specialized data types for organizing and accessing heterogeneous data. Dataset arrays store numeric data, text, and metadata in a single data container. Built-in methods enable you to merge datasets using a common key (join), calculate summary statistics on grouped data, and convert between tall and wide data representations. Categorical arrays provide a memory-efficient data container for storing information drawn from a finite, discrete set of categories.

### Key Features

- Statistical arrays for storing heterogeneous and categorical data
- Regression techniques, including linear, nonlinear, robust, and ridge, and nonlinear mixed-effects models
- Classification algorithms, including boosted and bagged decision trees, k-Nearest Neighbor, and linear discriminant analysis
- Analysis of variance (ANOVA)
- Probability distributions, including copulas and Gaussian mixtures
- Random number generation
- Hypothesis testing
- Design of experiments and statistical process control

# Organizing Data

---

- “Create Nominal and Ordinal Arrays” on page 2-3
- “Change Category Labels” on page 2-8
- “Reorder Category Levels” on page 2-10
- “Categorize Numeric Data” on page 2-15
- “Merge Category Levels” on page 2-18
- “Add and Drop Category Levels” on page 2-20
- “Plot Data Grouped by Category” on page 2-24
- “Test Differences Between Category Means” on page 2-28
- “Summary Statistics Grouped by Category” on page 2-37
- “Sort Ordinal Arrays” on page 2-39
- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43
- “Index and Search Using Categorical Arrays” on page 2-46
- “Grouping Variables” on page 2-51
- “Dummy Indicator Variables” on page 2-55
- “Regression with Categorical Covariates” on page 2-59
- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Create a Dataset Array from a File” on page 2-71
- “Add and Delete Observations” on page 2-79
- “Add and Delete Variables” on page 2-83
- “Access Data in Dataset Array Variables” on page 2-87

- “Select Subsets of Observations” on page 2-93
- “Sort Observations in Dataset Arrays” on page 2-97
- “Merge Dataset Arrays” on page 2-101
- “Stack or Unstack Dataset Arrays” on page 2-106
- “Calculations on Dataset Arrays” on page 2-111
- “Export Dataset Arrays” on page 2-115
- “Clean Messy and Missing Data” on page 2-117
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Dataset Arrays” on page 2-135
- “Index and Search Dataset Arrays” on page 2-138

## Create Nominal and Ordinal Arrays

### In this section...

“Create Nominal Arrays” on page 2-3

“Create Ordinal Arrays” on page 2-5

### Create Nominal Arrays

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

#### Load sample data.

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

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

```
ans =

    'setosa'
    'versicolor'
    'virginica'
```

#### Create a nominal array.

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

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

```
ans =

nominal
```

#### Explore category levels.

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

```
getlevels(speciesNom)

ans =

    setosa    versicolor    virginica
```

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

### Explore category labels.

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

```
getlabels(speciesNom)

ans =

    'setosa'    'versicolor'    'virginica'
```

### Specify your own category labels.

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

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

ans =

    'seto'    'vers'    'virg'
```

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

**Verify new category labels.**

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

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

```
ans =
```

```
1
```

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

**Create Ordinal Arrays**

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

**Load sample data.**

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

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

**Create an ordinal array.**

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

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

```
ans =
```

```
large    medium    small
```

**Explore category labels.**

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

```
getlabels(sizeOrd)

ans =

    'large'    'medium'    'small'
```

### **Add additional categories.**

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

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

ans =

    xsmall    small    medium    large    xlarge
```

### **Explore category labels.**

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

```
unique(sizeOrd2)

ans =

    small    medium    large
```

### **Specify the category order.**

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

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

ans =
```



```
small    medium    large
```

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

### **Compare elements.**

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

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

```
ans =
```

```
1
```

The logical value `1` indicates that the inequality holds.

## **See Also**

```
nominal | ordinal | getlabels | getlevels | unique | isequal |
```

## **Related Examples**

- “Change Category Labels” on page 2-8
- “Reorder Category Levels” on page 2-10
- “Merge Category Levels” on page 2-18
- “Index and Search Using Categorical Arrays” on page 2-46

## **Concepts**

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43

## Change Category Labels

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

### Load sample data.

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

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

```
ans =
```

```
4
6
8
```

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

### Create an ordinal array.

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

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

```
ans =
```

```
'4'    '6'    '8'
```

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

### Change category labels.

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

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

```
ans =  
  
      'Four'      'Six'      'Eight'
```

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

## See Also

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

## Related Examples

- “Reorder Category Levels” on page 2-10
- “Add and Drop Category Levels” on page 2-20
- “Index and Search Using Categorical Arrays” on page 2-46

## Concepts

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43

## Reorder Category Levels

<b>In this section...</b>
“Reorder Category Levels in Ordinal Arrays” on page 2-10
“Reorder Category Levels in Nominal Arrays” on page 2-11

### Reorder Category Levels in Ordinal Arrays

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

#### Load sample data.

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

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

#### Create an ordinal array.

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

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

```
ans =
```

```
       large       medium       small
```

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

#### Compare elements.

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

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

```
ans =  
  
      0
```

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

### **Reorder category levels.**

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

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

```
ans =  
  
      small      medium      large
```

### **Compare elements.**

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

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

```
ans =  
  
      1
```

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

## **Reorder Category Levels in Nominal Arrays**

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

### **Load sample data.**

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

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

```
ans =
```

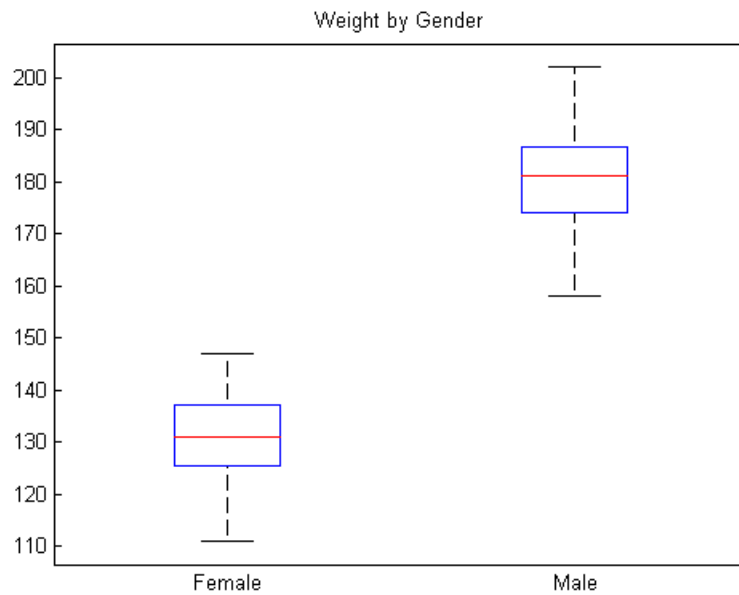
```
      Female      Male
```

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

### **Plot data grouped by category level.**

Draw box plots of weight, grouped by gender.

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



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

**Change the category order.**

Change the order of the category levels.

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

```
ans =
```

```
    Male    Female
```

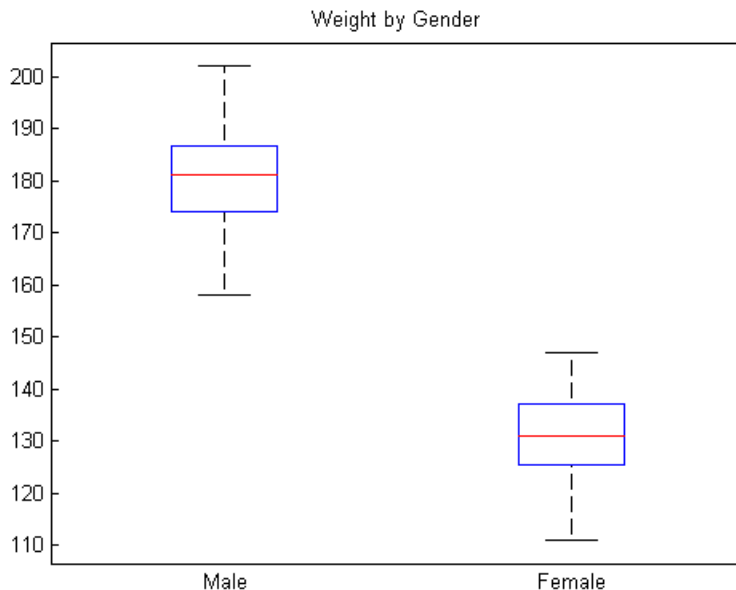
The levels are in the newly specified order.

**Plot data in new order.**

Draw box plots of weight by gender.

```
figure()
```

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



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

### See Also

`nominal` | `ordinal` | `getlevels` | `fitlmreorderlevels` |

### Related Examples

- “Change Category Labels” on page 2-8
- “Merge Category Levels” on page 2-18
- “Add and Drop Category Levels” on page 2-20
- “Index and Search Using Categorical Arrays” on page 2-46

### Concepts

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43



## Categorize Numeric Data

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

### Load sample data.

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

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

```
ans =
    25    39    50
```

The patient ages range from 25 to 50.

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

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

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

```
ans =
    Under 30    30-39    Over 40
```

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

### Explore categories.

Display the age and age category for the second patient.

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

```
{hospital.AgeCat(2), 'AgeCategory'}  
  
ans =  
  
    Age    AgeCategory  
    43    Over 40
```

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

### **Categorize a numeric array into quartiles.**

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

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

### **Explore categories.**

Display the weight and weight quartile for the second patient.

```
dataset({hospital.Weight(2), 'Weight'},...  
    {hospital.WeightQ(2), 'WeightQuartile'})  
  
ans =  
  
    Weight    WeightQuartile  
    163      Q3
```

### **Summary statistics grouped by category levels.**

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

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

```
ans =
```

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

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

## See Also

`ordinal` | `grpstats`

## Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-3
- “Merge Category Levels” on page 2-18
- “Plot Data Grouped by Category” on page 2-24
- “Index and Search Using Categorical Arrays” on page 2-46

## Concepts

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43

## Merge Category Levels

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

### Load sample data.

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

### Create a nominal array.

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

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

```
ans =
```

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

There are six unique countries of origin in the data.

### Tabulate category counts.

Explore the elements of the categorical array.

```
tabulate(Origin)
```

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

There are relatively few observations in each European country.

### Merge categories.

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

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

ans =
```

```
      Japan      USA      Europe
```

The variable `Origin` now has only three category levels.

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

Explore the elements of the merged categories.

```
tabulate(Origin)

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

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

### **See Also**

`nominal` | `mergelevels` |

### **Related Examples**

- “Create Nominal and Ordinal Arrays” on page 2-3
- “Add and Drop Category Levels” on page 2-20
- “Index and Search Using Categorical Arrays” on page 2-46

### **Concepts**

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43

## Add and Drop Category Levels

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

### Load sample data.

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

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

### Create an ordinal array.

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

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

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

```
ans =
```

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

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

### Check for undefined categories.

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

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

```
ans =
      1      0      1      1      0
```

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

**Identify elements in undefined categories.**

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

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

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

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

**Add a new category.**

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

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

```
Warning: Categorical level 'D-' being added.
```

```
> In categorical.subsasgn at 55
```

```
ans =
```

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

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

### Reorder category levels.

Reorder the categories so that `D- < D`.

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

```
ans =
```

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

### Compare elements.

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

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

```
ans =
```

```
32
```

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

### Explore categories.

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

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

```
ans =
```

```
      0      0      0      0      0
```



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

**Drop a category.**

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

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

```
ans =
```

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

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

**See Also**

[ordinal](#) | [isundefined](#) | [reorderlevels](#) | [droplevels](#) |

**Related Examples**

- “Create Nominal and Ordinal Arrays” on page 2-3
- “Reorder Category Levels” on page 2-10
- “Merge Category Levels” on page 2-18
- “Index and Search Using Categorical Arrays” on page 2-46

**Concepts**

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43

## Plot Data Grouped by Category

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

### **Load sample data.**

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

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

### **Create a nominal array.**

Convert `Origin` to a nominal array.

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

```
ans =
```

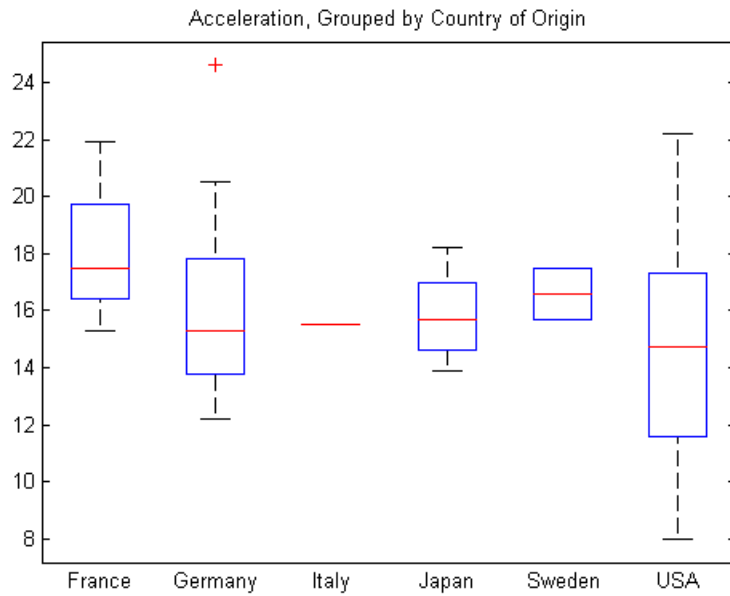
```
      France      Germany      Italy      Japan      Sweden      USA
```

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

### **Plot data grouped by category.**

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

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



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

Few observations have Italy as the country of origin.

**Tabulate category counts.**

Tabulate the number of sample cars from each country.

```
tabulate(Origin)
```

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

Only one car is made in Italy.

**Drop a category.**

Delete the Italian car from the sample.

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

```
ans =
```

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

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

**Drop a category level.**

Use `droplevels` to remove the `Italy` category.

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

```
tabulate(Origin2)  
  Value  Count  Percent  
  France    4    4.04%  
  Germany    9    9.09%  
  Japan   15   15.15%  
  Sweden    2    2.02%  
  USA     69   69.70%
```

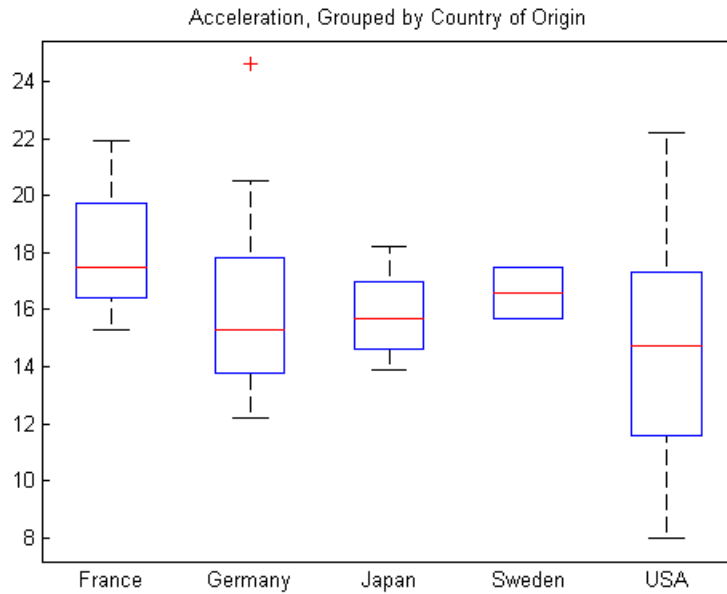
The `Italy` category is no longer in the nominal array, `Origin2`.

**Plot data grouped by category.**

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

```
figure()  
boxplot(Acceleration2, Origin2)
```

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



The plot no longer includes the car from Italy.

## See Also

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

## Related Examples

- “Test Differences Between Category Means” on page 2-28
- “Summary Statistics Grouped by Category” on page 2-37
- “Regression with Categorical Covariates” on page 2-59

## Concepts

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43
- “Grouping Variables” on page 2-51

## Test Differences Between Category Means

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

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

### Load sample data.

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

```
ans =
```

```
70
76
82
```

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

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

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

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

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

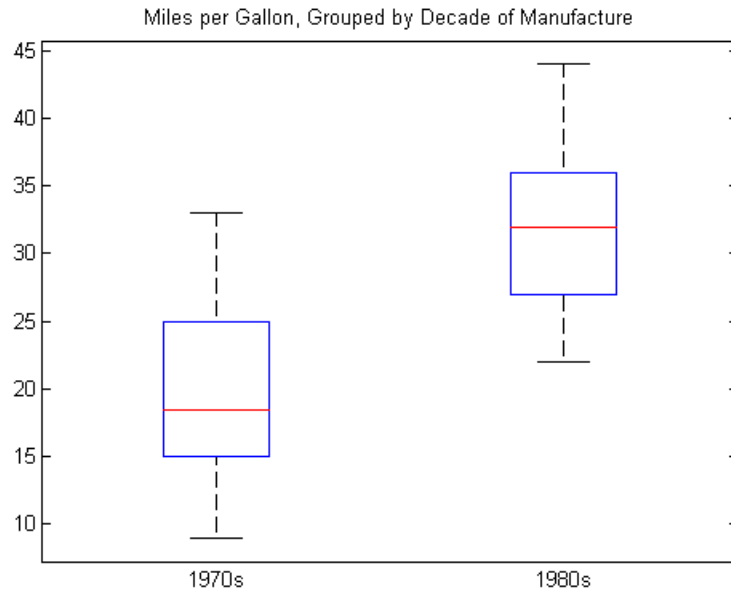
```
ans =
```

```
1970s    1980s
```

**Plot data grouped by category.**

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

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



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

**Compute summary statistics.**

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

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

```
xbar =
```

```
19.7857
31.7097

s2 =

35.1429
29.0796

grp =

'1970s'
'1980s'
```

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

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

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

$$H_0 : \mu_{70} = \mu_{80}$$
$$H_A : \mu_{70} \neq \mu_{80}.$$

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

```
h =

1

p =

3.4809e-15
```



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

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

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

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

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

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

**Merge categories.**

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

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

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

**Compute summary statistics.**

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

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

```
xbar =
```

```
31.8000  
21.1328  
26.6667
```

```
grp =
```

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

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

### **Conduct two-way ANOVA.**

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

The statistical model is

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

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

The hypotheses to test are equality of decade effects,

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

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

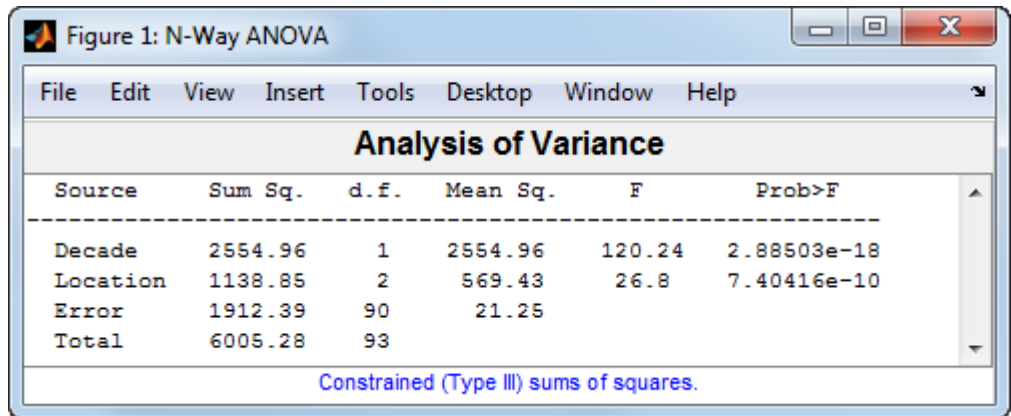
and equality of location effects,

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

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

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

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



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

### Conduct ANOCOVA analysis.

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

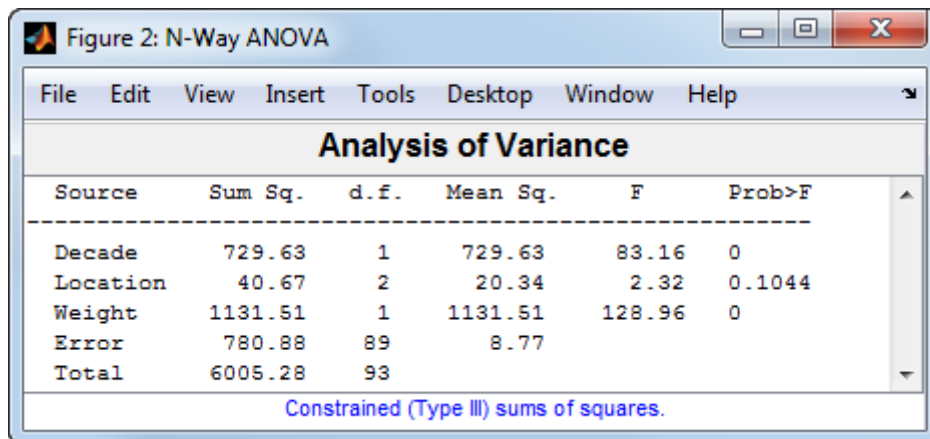
Assuming parallel lines, the statistical model is

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

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

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

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



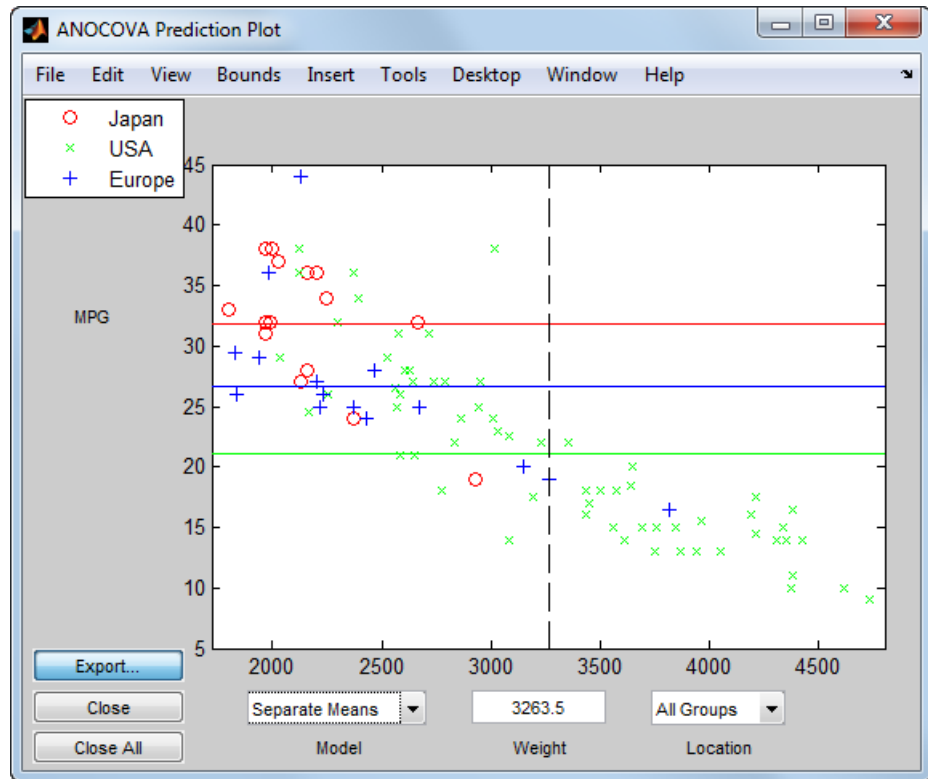
This output shows that when car weight is considered, there is insufficient evidence of a manufacturing location effect (p-value = 0.1044).

#### Use interactive tool.

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

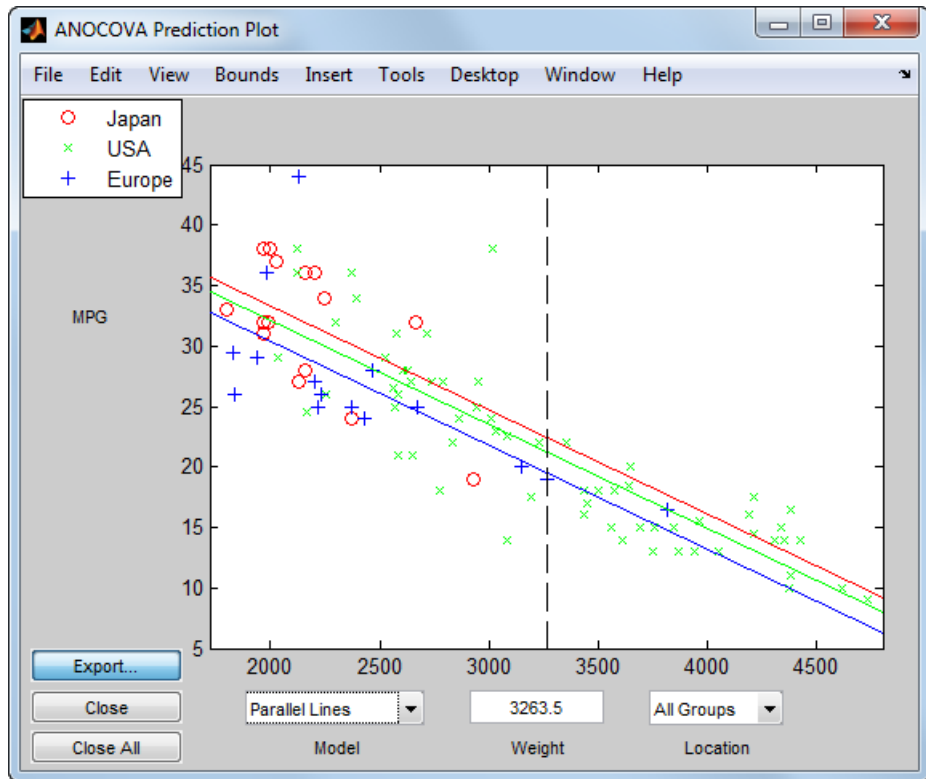
```
aoctool(Weight,MPG,Location);
```

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



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

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



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

**See Also**

`ordinal` | `nominal` | `boxplot` | `grpstats` | `ttest2` | `anovan` | `aocool`

**Related Examples**

- “Plot Data Grouped by Category” on page 2-24
- “Summary Statistics Grouped by Category” on page 2-37
- “Regression with Categorical Covariates” on page 2-59

**Concepts**

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43
- “Grouping Variables” on page 2-51

## Summary Statistics Grouped by Category

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

### Load sample data.

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

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

### Compute summary statistics by category.

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

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

```
stats =
```

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

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

### Compute summary statistics by multiple categories.

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

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

```
stats =
```

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

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

### See Also

`dataset | nominal | grpstats`

### Related Examples

- “Plot Data Grouped by Category” on page 2-24
- “Test Differences Between Category Means” on page 2-28
- “Calculations on Dataset Arrays” on page 2-111

### Concepts

- “Grouping Variables” on page 2-51
- “Categorical Arrays” on page 2-41
- “Dataset Arrays” on page 2-135



## Sort Ordinal Arrays

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

### Load sample data.

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

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

### Create an ordinal array.

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

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

```
ans =  
  
    small    medium    large
```

### Sort the ordinal array.

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

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

```
ans =  
  
    small  
    small  
    small  
    medium  
    medium  
    medium  
    large  
    large
```

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

### **See Also**

`ordinal` | `sort` |

### **Related Examples**

- “Reorder Category Levels” on page 2-10
- “Add and Drop Category Levels” on page 2-20

### **Concepts**

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43

## Categorical Arrays

### In this section...

“What Are Categorical Arrays?” on page 2-41

“Categorical Array Conversion” on page 2-41

### What Are Categorical Arrays?

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

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

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

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

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

### Categorical Array Conversion

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

- Numeric array

- Logical array
- Character array
- Cell array of strings

### **See Also**

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

### **Related Examples**

- “Create Nominal and Ordinal Arrays” on page 2-3
- “Summary Statistics Grouped by Category” on page 2-37
- “Plot Data Grouped by Category” on page 2-24
- “Index and Search Using Categorical Arrays” on page 2-46

### **Concepts**

- “Advantages of Using Categorical Arrays” on page 2-43
- “Grouping Variables” on page 2-51

## Advantages of Using Categorical Arrays

In this section...
“Manipulate Category Levels” on page 2-43
“Analysis Using Categorical Arrays” on page 2-43
“Reduce Memory Requirements” on page 2-44

### Manipulate Category Levels

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

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

### Analysis Using Categorical Arrays

You can use categorical arrays in a variety of statistical analyses. For example, you might want to compute descriptive statistics for data grouped by the category levels, conduct statistical tests on differences between category means, or perform regression analysis using categorical predictors.

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

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

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

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

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



### **Reduce Memory Requirements**

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

For example, load some sample data:



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

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

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

Convert species to a nominal array:

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

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

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

## See Also

`ordinal` | `nominal` |

## Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-3
- “Test Differences Between Category Means” on page 2-28
- “Regression with Categorical Covariates” on page 2-59
- “Index and Search Using Categorical Arrays” on page 2-46

## Concepts

- “Categorical Arrays” on page 2-41
- “Grouping Variables” on page 2-51
- “Dummy Indicator Variables” on page 2-55

## Index and Search Using Categorical Arrays

### Index By Category

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

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

### Common Indexing and Searching

The following brief examples illustrate several indexing and searching methods.

**Load sample data.**

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

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

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



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

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

```
ans =
```

```
0
```

There are no sample cars manufactured in Canada. List the countries that are levels of `Origin`.

```
getlevels(Origin)
```

```
ans =
```

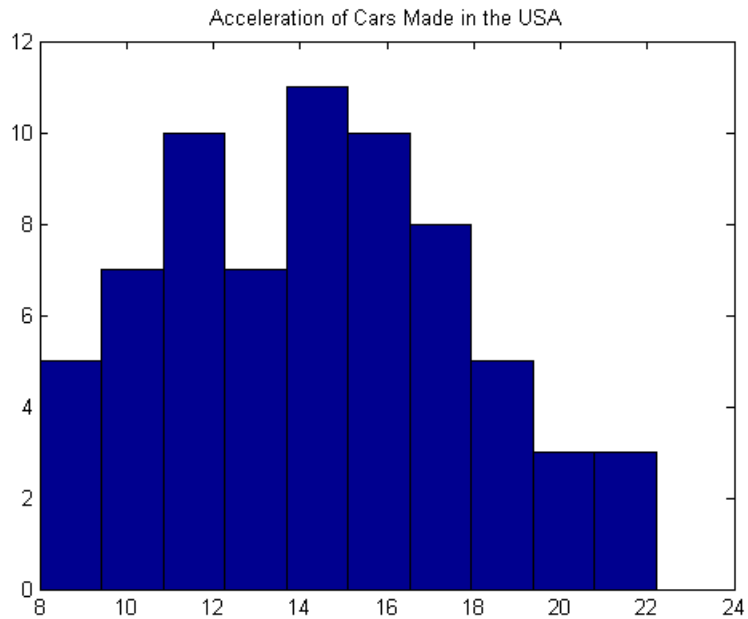
```
France      Germany      Italy      Japan      Sweden      USA
```

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

```
figure()
```

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

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



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

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

```
ans =
```

```
0
```

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

```
getlevels(Origin)
```

```
ans =
```

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

Remove Sweden from the levels of `Origin`.

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

```
ans =
```

```
      France      Germany      Italy      Japan      USA
```

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

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

```
ix =
```

```
    11
    27
    39
    61
```

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

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

```
Warning: OLDLEVELS contains categorical levels that were present in the
input array A. As a result, the output array B contains array elements
that have undefined levels.
```

```
> In categorical.droplevels at 52
```

The warning indicates that you are dropping a category level that has elements in it. These observations are no longer in a defined category, indicated by `<undefined>`.

```
Origin(ix)
```

```
ans =
```

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

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

```
find(isundefined(Origin))
```

```
ans =
```

```
11  
27  
39  
61
```

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

### See Also

`nominal` | `ordinal` | `ismember` | `isundefined` | `droplevels` |

### Related Examples

- “Create Nominal and Ordinal Arrays” on page 2-3
- “Reorder Category Levels” on page 2-10
- “Merge Category Levels” on page 2-18
- “Add and Drop Category Levels” on page 2-20

### Concepts

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43

## Grouping Variables

### In this section...

“What Are Grouping Variables?” on page 2-51

“Group Definition” on page 2-52

“Analysis Using Grouping Variables” on page 2-52

“Missing Group Values” on page 2-53

### What Are Grouping Variables?

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

- Numeric vector
- Logical vector
- String array (also called character arrays)
- Cell array of strings
- Categorical vector

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

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

Data Type	Grouping Variable
Numeric vector	[ 1 2 2 1 2 ]
Logical vector	[ 0 1 1 0 1 ]

Data Type	Grouping Variable
Cell array of strings	{'Male', 'Female', 'Female', 'Male', 'Female'}
Categorical vector	Male Female Female Male Female

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

### Group Definition

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

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

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

### Analysis Using Grouping Variables

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

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

## Missing Group Values

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

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

### See Also

categorical | ordinal | nominal |

### Related Examples

- “Plot Data Grouped by Category” on page 2-24
- “Summary Statistics Grouped by Category” on page 2-37

### **Concepts**

- “Categorical Arrays” on page 2-41
- “Advantages of Using Categorical Arrays” on page 2-43



## Dummy Indicator Variables

### In this section...

“What Are Dummy Variables?” on page 2-55

“Creating Dummy Variables” on page 2-56

### What Are Dummy Variables?

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

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

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

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

In this example,  $X_1$  is a dummy variable that has value 1 for the `Medium` group, and 0 otherwise.  $X_2$  is a dummy variable that has value 1 for the `Large` group, and 0 otherwise. Together, these two variables represent the three categories. Observations in the `Small` group have 0s for both dummy variables.

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

## Creating Dummy Variables

### Automatic Creation of Dummy Variables

The regression fitting functions, `fitlm`, `fitglm`, and `fitnlm`, recognize categorical array inputs as categorical predictors. That is, if you input your categorical predictor as a `nominal` or `ordinal` array, the fitting function automatically creates the required dummy variables. The first level returned by `getlevels` is the reference group. To use a different reference group, use `reorderlevels` to change the level order.

If there are  $c$  unique levels in the categorical array, then the fitting function estimates  $c - 1$  regression coefficients for the categorical predictor.

---

**Note** The fitting functions use every level of the categorical array returned by `getlevels`, even if there are levels with no observations. To remove levels from the categorical array, use `droplevels`.

---

### Manual Creation of Dummy Variables

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

For example,

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

```

dv =
    0     1
    1     0
    1     0
    0     1
    1     0

```

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

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

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

```

X =
    1     0     1
    1     1     0
    1     1     0
    1     0     1
    1     1     0

```

```
rank(X)
```

```

ans =
    2

```

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

## See Also

`nominal` | `ordinal` | `fitlm` | `fitglm` | `fitnlm` | `dummyvar`

### **Related Examples**

- “Regression with Categorical Covariates” on page 2-59
- “Test Differences Between Category Means” on page 2-28

## Regression with Categorical Covariates

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

### **Load sample data.**

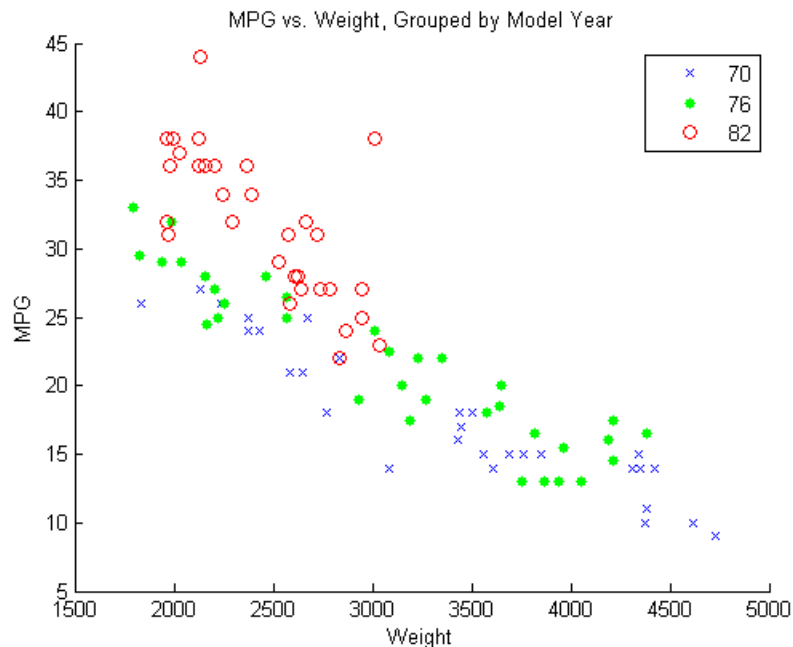
```
load('carsmall')
```

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

### **Plot grouped data.**

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

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



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

### Create dataset and nominal arrays.

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

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

### Fit a regression model.

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

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

The proposed model is

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

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

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

```
fit =
```

Linear regression model:

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

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	37.399	2.1466	17.423	2.8607
Weight	-0.0058437	0.00061765	-9.4612	4.6077
Model_Year_76	4.6903	2.8538	1.6435	0.1
Model_Year_82	21.051	4.157	5.0641	2.2364
Weight:Model_Year_76	-0.00082009	0.00085468	-0.95953	0.3
Weight:Model_Year_82	-0.0050551	0.0015636	-3.2329	0.001

Number of observations: 94, Error degrees of freedom: 88

Root Mean Squared Error: 2.79

R-squared: 0.886, Adjusted R-Squared 0.88

F-statistic vs. constant model: 137, p-value = 5.79e-40

The regression output shows:

- `fitlm` recognizes `Model_Year` as a nominal variable, and constructs the required indicator (dummy) variables. By default, the first level, 70, is the reference group (use `reorderlevels` to change the reference group).

- The model specification,  $MPG \sim Weight * Model\_Year$ , specifies the first-order terms for *Weight* and *Model\_Year*, and all interactions.
- The model  $R^2 = 0.886$ , meaning the variation in miles per gallon is reduced by 88.6% when you consider weight, model year, and their interactions.
- The fitted model is

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

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

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

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

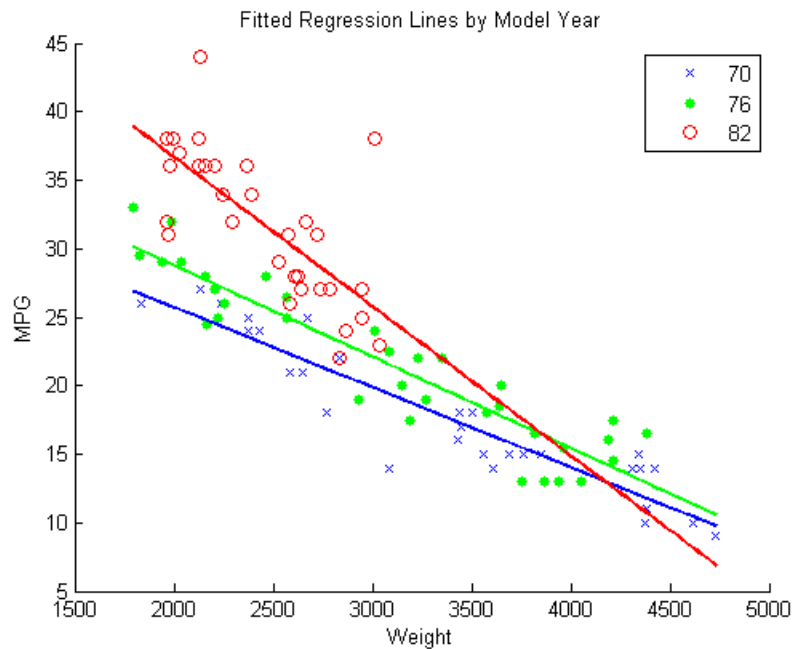
### Plot fitted regression lines.

Plot the data and fitted regression lines.

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

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





### Test for different slopes.

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

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

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

```
anova(fit)
```

```
ans =
```

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

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

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

### See Also

`fitlmdataset` | `nominal` | `reorderlevels` |

### Related Examples

- “Plot Data Grouped by Category” on page 2-24
- “Test Differences Between Category Means” on page 2-28
- “Summary Statistics Grouped by Category” on page 2-37

### Concepts

- “Advantages of Using Categorical Arrays” on page 2-43
- “Grouping Variables” on page 2-51
- “Dummy Indicator Variables” on page 2-55

## Create a Dataset Array from Workspace Variables

### In this section...

“Create a Dataset Array from a Numeric Array” on page 2-65

“Create Dataset Array from Heterogeneous Workspace Variables” on page 2-68

### Create a Dataset Array from a Numeric Array

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

#### Load sample data.

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

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

#### Create a dataset array.

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

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

```
ans =
```

meas1	meas2	meas3	meas4
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5	3.6	1.4	0.2
5.4	3.9	1.7	0.4
4.6	3.4	1.4	0.3
5	3.4	1.5	0.2
4.4	2.9	1.4	0.2
4.9	3.1	1.5	0.1

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

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

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

### **Examine the dataset array.**

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

```
size(ds)
```

```
ans =
```

```
    150     4
```

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

### **Explore dataset array metadata.**

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

```
ds.Properties
```

```
ans =
```

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

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

### Access data in a dataset array variable.

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

```
min(ds.meas1)

ans =

    4.3000
```

### Change variable names.

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

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

### Add description.

you can add a description for the dataset array.

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

```
ans =

    Description: 'Fisher iris data'
  VarDescription: {}
         Units: {}
   DimNames: {'Observations' 'Variables'}
   UserData: []
   ObsNames: {}
   VarNames: {'SLength' 'SWidth' 'PLength' 'PWidth'}
```

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

### Add a variable to the dataset array.

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

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

```
ans =
```

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

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

## Create Dataset Array from Heterogeneous Workspace Variables

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

### Load sample data.

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

### Create a dataset array.

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

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

```
ans =
```

```
'Origin'  
'Acceleration'
```

```
'Cylinders'
'MPG'
```

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

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

### Examine a dataset array.

Display the first five observations in the dataset array.

```
ds(1:5,:)
```

```
ans =
```

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

### Apply a function to a dataset array.

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

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

```
ans =
```

```
'char'
'double'
'double'
'double'
```

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

### Modify a dataset array.

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

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

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

```
ans =
```

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

The variable `Cylinders` has a new data type.

### See Also

[dataset](#) | [datasetfun](#) | [mat2datasetnominal](#) |

### Related Examples

- “Create a Dataset Array from a File” on page 2-71
- “Export Dataset Arrays” on page 2-115
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

### Concepts

- “Dataset Arrays” on page 2-135



## Create a Dataset Array from a File

### In this section...

“Create a Dataset Array from a Tab-Delimited Text File” on page 2-71

“Create a Dataset Array from a Comma-Separated Text File” on page 2-74

“Create a Dataset Array from an Excel File” on page 2-77

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

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

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

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

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

```
ds =
```

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

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

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

### Summarize the dataset array.

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

```
summary(ds)
```

```
name: [14x1 cell string]
```

```
sex: [14x1 cell string]
```

```
age: [14x1 double]
```

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

```
wgt: [14x1 double]
```

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

```
smoke: [14x1 double]
```

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

### Import observation names.

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

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

```
ds =

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

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

### **Change dataset array properties.**

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

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

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

```
ans =
```

```
      Description: ''
      VarDescription: {}
           Units: {}
      DimNames: {'LastName' 'Variables'}
      UserData: []
```

```
ObsNames: {14x1 cell}
VarNames: {'sex' 'age' 'wgt' 'smoke'}
```

### Index into dataset array.

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

```
ds('BROWN', :)
```

```
ans =
```

	sex	age	wgt	smoke
BROWN	'f'	49	119	0

Note that observation names must be unique.

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

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

### Create a dataset array.

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

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

```
ds =
```

id	name	sex	age	wgt	smoke
'YPL-320'	'SMITH'	'm'	38	176	1
'GLI-532'	'JOHNSON'	'm'	43	163	0
'PNI-258'	'WILLIAMS'	'f'	38	131	0
'MIJ-579'	'JONES'	'f'	40	133	0
'XLK-030'	'BROWN'	'f'	49	119	0

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

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

### Add observation names.

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

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

```
ds =
```

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

### Delete observations.

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

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

```
ds =
```

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

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

### **Return size of dataset array.**

The array now has 13 observations.

```
size(ds)
```

```
ans =
```

```
13    5
```

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

## Create a Dataset Array from an Excel File

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

### Create a dataset array.

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

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

```
ds =
```

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

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

### Specify which worksheet to import.

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

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

ds2 =

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

### See Also

[dataset](#) | [summary](#) |

### Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Clean Messy and Missing Data” on page 2-117
- “Export Dataset Arrays” on page 2-115
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

### Concepts

- “Dataset Arrays” on page 2-135



## Add and Delete Observations

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

### Load sample data.

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

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

```
ans =
```

```
    14     6
```

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

### Add observations by concatenation.

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

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

```
ans =
```

```
    22     6
```

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

### Add observations from a cell array.

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

```
cellObs = {'id', 'name', 'sex', 'age', 'wgt', 'smoke';  
          'YQR-965', 'BAKER', 'M', 36, 160, 0;  
          'LFG-497', 'WALL', 'F', 28, 125, 1;  
          'KSD-003', 'REED', 'M', 32, 187, 0};  
dsNew = [dsNew; cell2dataset(cellObs)];  
size(dsNew)
```

```
ans =  
  
    25     6
```

### **Add observations from a structure.**

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

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

```
ans =  
  
    27     6
```

### **Delete duplicate observations.**

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

```
dsNew = unique(dsNew);  
size(dsNew)
```

```
ans =  
  
    21     6
```

One duplicated observation is deleted.

#### **Delete observations by observation number.**

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

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

```
ans =  
  
    18     6
```

The dataset array has only 18 observations now.

#### **Delete observations by observation name.**

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

```
dsNew.Properties.ObsNames = dsNew.id;  
dsNew.id = [];  
dsNew('K0Q-996',:) = [];  
size(dsNew)
```

```
ans =  
  
    17     5
```

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

#### **Search for observations to delete.**

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

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

```
ans =
```

```
16    5
```

The dataset array now has one less observation.

### See Also

[dataset](#) | [cell2dataset](#) | [struct2dataset](#)

### Related Examples

- “Add and Delete Variables” on page 2-83
- “Select Subsets of Observations” on page 2-93
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

### Concepts

- “Dataset Arrays” on page 2-135

## Add and Delete Variables

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

### Load sample data.

Navigate to the folder containing sample data.

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

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

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

```
ans =
```

```
    14     6
```

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

### Add variables by concatenating dataset arrays.

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

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

```
ans =
```

```
    14     7
```

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

```
ds.Properties.VarNames{end}
```

```
ans =
```

```
hgt
```

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

### Delete variables by variable name.

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

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

```
ans =
```

```
14    6
```

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

```
ds.Properties.VarNames(:)
```

```
ans =
```

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

There is no longer a variable called `id`.

### Add a new variable by name.

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

```
ds.bmi = ds.wgt*703./ds.hgt.^2;
```

```
ds(:, {'name', 'sex', 'bmi'})

ans =
```

	name	sex	bmi
YPL-320	'SMITH'	'm'	24.544
GLI-532	'JOHNSON'	'm'	24.068
PNI-258	'WILLIAMS'	'f'	23.958
MIJ-579	'JONES'	'f'	25.127
XLK-030	'BROWN'	'f'	21.078
TFP-518	'DAVIS'	'f'	27.729
LPD-746	'MILLER'	'f'	26.828
ATA-945	'WILSON'	'm'	24.41
VNL-702	'MOORE'	'm'	27.822
LQW-768	'TAYLOR'	'f'	22.655
QFY-472	'ANDERSON'	'f'	23.409
UJG-627	'THOMAS'	'f'	25.883
XUE-826	'JACKSON'	'm'	24.265
TRW-072	'WHITE'	'm'	29.827

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

### Delete variables by variable number.

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

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

ans =
```

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

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

**See Also** dataset |

**Related  
Examples**

- “Add and Delete Observations” on page 2-79
- “Merge Dataset Arrays” on page 2-101
- “Calculations on Dataset Arrays” on page 2-111
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

**Concepts**

- “Dataset Arrays” on page 2-135



## Access Data in Dataset Array Variables

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

### Access variables by name.

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

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

```
ans =
```

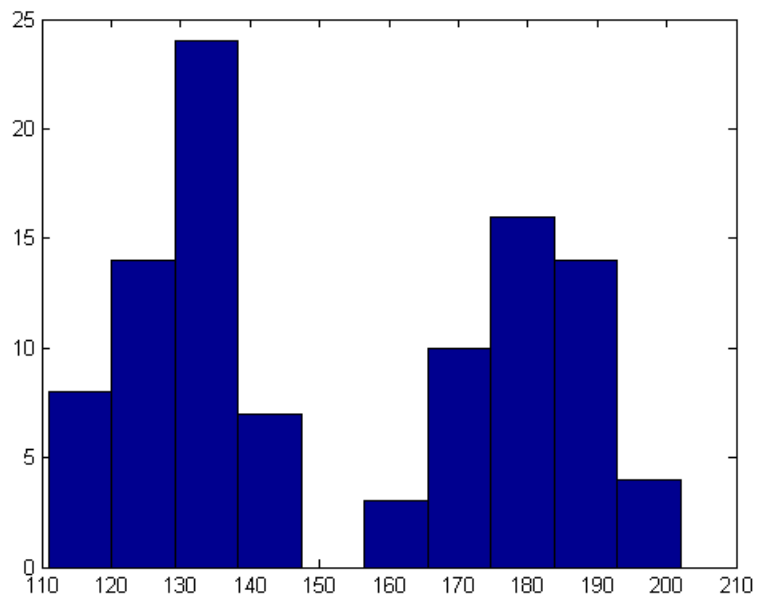
```
    'LastName'  
    'Sex'  
    'Age'  
    'Weight'  
    'Smoker'  
    'BloodPressure'  
    'Trials'
```

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

### Plot histogram.

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

```
figure()  
hist(hospital.Weight)
```

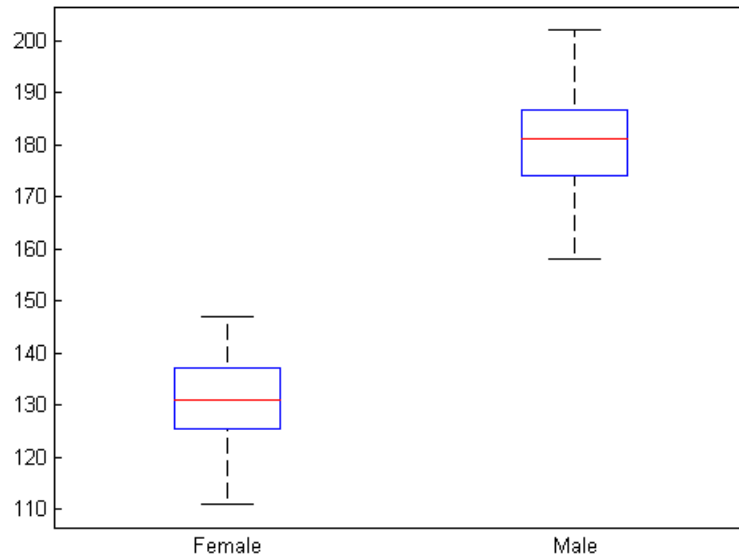


The histogram shows that the weight distribution is bimodal.

**Plot data grouped by category.**

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

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



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

**Select a subset of variables.**

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

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

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

**Convert the variable data type.**

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

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

```
ans =
```

```
nominal
```

### **Explore data.**

Display the first 10 elements of `Smoker`.

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

```
ans =
```

```
Yes
```

```
No
```

```
No
```

```
No
```

```
No
```

```
No
```

```
Yes
```

```
No
```

```
No
```

```
No
```

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

### **Add variables.**

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

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

```
ans =
```

```

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

```

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

### **Search for variables by name.**

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

```

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

```

```

ans =

    'BloodPressure'
    'SysPressure'
    'DiaPressure'

```

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

### **Delete variables.**

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

```

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

```

```

ans =

    'LastName'

```

```
'Sex '  
'Age '  
'Weight '  
'Smoker '  
'Trials '  
'SysPressure '  
'DiaPressure '
```

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

**See Also** `dataset |`

### Related Examples

- “Add and Delete Variables” on page 2-83
- “Calculations on Dataset Arrays” on page 2-111
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

### Concepts

- “Dataset Arrays” on page 2-135
- “Grouping Variables” on page 2-51

## Select Subsets of Observations

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

### Load sample data.

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

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

```
ans =
```

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

These are the first 10 observation names.

### Index an observation by name.

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

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

```
ans =
```

	LastName	Sex	Age
XLK-030	'BROWN'	Female	49

### Index a subset of observations by number.

Create a new dataset array containing the first 50 patients.

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

```
ans =  
  
    50     7
```

### Search observations using a logical condition.

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

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

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

### Search observations using multiple conditions.

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

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



```
ans =
```

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

**Select a random subset of observations.**

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

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

```
ans =
```

```
'DAU-529'
'AGR-528'
'RBO-332'
'Q00-305'
'RVS-253'
'QEQ-082'
'EHE-616'
'HVR-372'
'KOQ-996'
'REV-997'
'PUE-347'
'LQW-768'
'YLN-495'
'HJQ-495'
'ELG-976'
```

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

### Delete observations by name.

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

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

```
ans =
```

```
99      7
```

The dataset array has one less observation.

### See Also `dataset` |

### Related Examples

- “Add and Delete Observations” on page 2-79
- “Clean Messy and Missing Data” on page 2-117
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Sort Observations in Dataset Arrays” on page 2-97
- “Index and Search Dataset Arrays” on page 2-138

### Concepts

- “Dataset Arrays” on page 2-135

## Sort Observations in Dataset Arrays

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

### Sort observations in ascending order.

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

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

ans =

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

The youngest patients are age 25.

### Sort observations in descending order.

Sort the observations by `Age` in descending order.

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

ans =

	LastName	Age
XBA-581	'ROBINSON'	50

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

The oldest patients are age 50.

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

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

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

```
ans =
```

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

Now the names are sorted alphabetically within increasing age groups.

### **Sort observations in mixed order.**

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

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

```
dsWeight(1:10,{'LastName','Age','Weight'})
```

```
ans =
```

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

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

### Sort observations by observation name.

Sort the observations in hospital by the observation names.

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

```
ans =
```

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

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

### **See Also**

dataset | sortrows |

### **Related Examples**

- “Select Subsets of Observations” on page 2-93
- “Stack or Unstack Dataset Arrays” on page 2-106
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

### **Concepts**

- “Dataset Arrays” on page 2-135

## Merge Dataset Arrays

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

### Load sample data.

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

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

```
ds1 =
```

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

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

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

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

```
ds2 =
```

id	hgt
'LPD-746'	61
'PNI-258'	62
'XUE-826'	71
'ATA-945'	72
'XLK-030'	63

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

### **Merge only the matching subset of observations.**

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

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

```
JoinSmall =
```

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

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

### **Include incomplete observations in the merge.**

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

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

```
joinAll =
```

id	hgt	name	sex	age
'ATA-945'	72	'WILSON'	'm'	40
'GLI-532'	NaN	'JOHNSON'	'm'	43



'LPD-746'	61	'MILLER'	'f'	33
'LQW-768'	NaN	'TAYLOR'	'f'	31
'MIJ-579'	NaN	'JONES'	'f'	40
'PNI-258'	62	'WILLIAMS'	'f'	38
'QFY-472'	NaN	'ANDERSON'	'f'	45
'TFP-518'	NaN	'DAVIS'	'f'	46
'TRW-072'	NaN	'WHITE'	'm'	39
'UJG-627'	NaN	'THOMAS'	'f'	42
'VNL-702'	NaN	'MOORE'	'm'	28
'XLK-030'	63	'BROWN'	'f'	49
'XUE-826'	71	'JACKSON'	'm'	25
'YPL-320'	NaN	'SMITH'	'm'	38

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

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

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

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

```
ds3 =
```

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

```
'XUE-826'      71
'TRW-072'      69
```

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

### Specify key variable.

You can easily change the variable name of the key variable in `ds3` by setting `d3.Properties.VarNames` or using the Variables editor, but it is not required to perform a merge. Instead, you can specify the name of the key variable in each dataset array using `LeftKeys` and `RightKeys`.

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

```
joinDiff =
```

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

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

### See Also

`dataset` | `join` |

**Related  
Examples**

- “Add and Delete Variables” on page 2-83
- “Stack or Unstack Dataset Arrays” on page 2-106
- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

**Concepts**

- “Dataset Arrays” on page 2-135

## Stack or Unstack Dataset Arrays

This example shows how to reformat dataset arrays between wide and tall (or long) format using `stack` and `unstack`.

### Load sample data.

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

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

```
ds =
```

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

Each of the 10 students has 4 test scores, displayed here in wide format.

### Perform calculations on dataset array.

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

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

```
ans =
```

LastName	Sex	TestAve
----------	-----	---------

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

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

**Reformat the dataset array into tall format.**

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

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

```
dsTall =
```

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

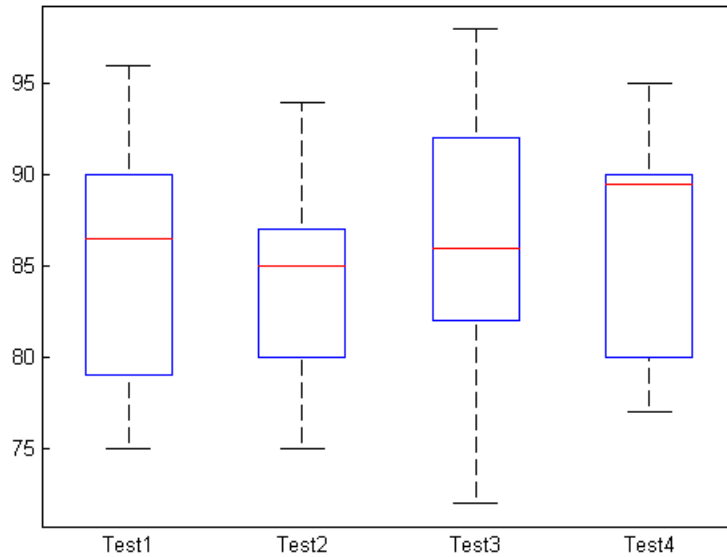
'GRAY'	'female'	88	Test3	87
'GRAY'	'female'	88	Test4	90
'RAMIREZ'	'female'	95.25	Test1	96
'RAMIREZ'	'female'	95.25	Test2	92
'RAMIREZ'	'female'	95.25	Test3	98
'RAMIREZ'	'female'	95.25	Test4	95
'JAMES'	'male'	76.75	Test1	78
'JAMES'	'male'	76.75	Test2	75
'JAMES'	'male'	76.75	Test3	77
'JAMES'	'male'	76.75	Test4	77
'WATSON'	'female'	91.75	Test1	91
'WATSON'	'female'	91.75	Test2	94
'WATSON'	'female'	91.75	Test3	92
'WATSON'	'female'	91.75	Test4	90
'BROOKS'	'female'	85.75	Test1	86
'BROOKS'	'female'	85.75	Test2	83
'BROOKS'	'female'	85.75	Test3	85
'BROOKS'	'female'	85.75	Test4	89
'KELLY'	'male'	79.25	Test1	79
'KELLY'	'male'	79.25	Test2	76
'KELLY'	'male'	79.25	Test3	82
'KELLY'	'male'	79.25	Test4	80

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

### **Plot data grouped by category.**

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

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



**Reformat the dataset array into wide format.**

Reformat dsTall back into its original wide format.

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

ans =

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

'BROOKS'	86	83	85	89
'KELLY'	79	76	82	80

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

### See Also

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

### Related Examples

- “Access Data in Dataset Array Variables” on page 2-87
- “Calculations on Dataset Arrays” on page 2-111
- “Index and Search Dataset Arrays” on page 2-138

### Concepts

- “Dataset Arrays” on page 2-135
- “Grouping Variables” on page 2-51



## Calculations on Dataset Arrays

This example shows how to perform calculations on dataset arrays.

### Load sample data.

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

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

```
ds =
```

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

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

### Average dataset array variables.

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

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

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

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

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

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

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

```
stats =
```

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

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

### Replace data values.

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

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

```
ds =
```

LastName	Sex	Test1	Test2	Test3	Test4	T
----------	-----	-------	-------	-------	-------	---

'HOWARD'	'male'	22.5	21.75	23.25	23	
'WARD'	'male'	21.75	21.25	20.75	22.5	8
'TORRES'	'male'	21.5	21.25	22	21.5	8
'PETERSON'	'female'	18.75	20	18	19.25	
'GRAY'	'female'	22.25	21.5	21.75	22.5	
'RAMIREZ'	'female'	24	23	24.5	23.75	9
'JAMES'	'male'	19.5	18.75	19.25	19.25	7
'WATSON'	'female'	22.75	23.5	23	22.5	9
'BROOKS'	'female'	21.5	20.75	21.25	22.25	8
'KELLY'	'male'	19.75	19	20.5	20	7

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

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

### Change variable name.

Change the variable name to `Final`.

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

```
ds =
```

LastName	Sex	Test1	Test2	Test3	Test4	F
'HOWARD'	'male'	22.5	21.75	23.25	23	
'WARD'	'male'	21.75	21.25	20.75	22.5	8
'TORRES'	'male'	21.5	21.25	22	21.5	8
'PETERSON'	'female'	18.75	20	18	19.25	
'GRAY'	'female'	22.25	21.5	21.75	22.5	
'RAMIREZ'	'female'	24	23	24.5	23.75	9
'JAMES'	'male'	19.5	18.75	19.25	19.25	7
'WATSON'	'female'	22.75	23.5	23	22.5	9
'BROOKS'	'female'	21.5	20.75	21.25	22.25	8
'KELLY'	'male'	19.75	19	20.5	20	7

## See Also

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

### **Related Examples**

- “Stack or Unstack Dataset Arrays” on page 2-106
- “Access Data in Dataset Array Variables” on page 2-87
- “Select Subsets of Observations” on page 2-93
- “Index and Search Dataset Arrays” on page 2-138

### **Concepts**

- “Dataset Arrays” on page 2-135

## Export Dataset Arrays

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

### Load sample data.

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

The dataset array has 100 observations and 7 variables.

### Export to a text file.

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

```
export(hospital)
```

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

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

### Export without variable names.

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

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

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

### Export to a comma-delimited format.

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

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

### **Export to an Excel spreadsheet.**

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

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

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

### **See Also**

`dataset` | `export` |

### **Related Examples**

- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Create a Dataset Array from a File” on page 2-71

### **Concepts**

- “Dataset Arrays” on page 2-135

## Clean Messy and Missing Data

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

### Load sample data.

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

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

```
messyData =
```

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

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

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

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

### **Find observations with missing values.**

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

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

ans =

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

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

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



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

### Convert string arrays to double arrays.

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

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

```
messyData =
```

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

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

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

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

### Replace missing value indicators.

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

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

```
messyData =
```

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

### Create a dataset array with complete observations.

Create a new dataset array that contains only the complete observations—those without missing data.

```
ix = ismissing(messyData);
completeData = messyData(~any(ix,2),:)
```

```
completeData =
```

var1	var2	var3	var4	var5
'afe1'	3	'yes'	3	3
'wth4'	3	'yes'	3	3
'atn2'	23	'no'	23	23
'arg1'	5	'yes'	5	5
'jre3'	34.6	'yes'	34.6	34.6
'wen9'	234	'yes'	234	234
'ple2'	2	'no'	2	2
'dbo8'	5	'no'	5	5
'oii4'	5	'yes'	5	5
'wnk3'	245	'yes'	245	245
'pnj5'	463	'no'	463	463
'wnn3'	6	'no'	6	6
'oks9'	23	'yes'	23	23
'pkn4'	2	'no'	2	2
'adw3'	22	'no'	22	22
'bas8'	23	'no'	23	23

## See Also

[dataset](#) | [ismissing](#) | [replaceWithMissing](#) |

## Related Examples

- “Select Subsets of Observations” on page 2-93
- “Calculations on Dataset Arrays” on page 2-111
- “Index and Search Dataset Arrays” on page 2-138

## Concepts

- “Dataset Arrays” on page 2-135

## Dataset Arrays in the Variables Editor

### In this section...

“Open Dataset Arrays in the Variables Editor” on page 2-122

“Modify Variable and Observation Names” on page 2-123

“Reorder or Delete Variables” on page 2-125

“Add New Data” on page 2-127

“Sort Observations” on page 2-129

“Select a Subset of Data” on page 2-130

“Create Plots” on page 2-133

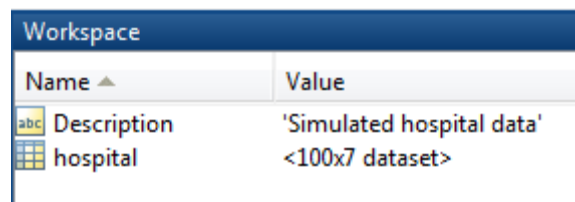
### Open Dataset Arrays in the Variables Editor

The MATLAB Variables editor provides a convenient interface for viewing, modifying, and plotting dataset arrays.

First, load the sample data set, `hospital`.

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

The dataset array, `hospital`, is created in the MATLAB workspace.

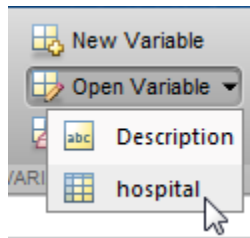


The screenshot shows the MATLAB Workspace window with a table of variables. The table has two columns: 'Name' and 'Value'. The first row shows 'Description' with a value of 'Simulated hospital data'. The second row shows 'hospital' with a value of '<100x7 dataset>'. The 'hospital' row is highlighted with a blue background.

Name	Value
Description	'Simulated hospital data'
hospital	<100x7 dataset>

The dataset array has 100 observations and 7 variables.

To open `hospital` in the Variables editor, click **Open Variable**, and select `hospital`.



The Variables editor opens, displaying the contents of the dataset array (only the first 10 observations are shown here).

hospital <100x7 dataset>							
	1	2	3	4	5	6	7
	LastName	Sex	Age	Weight	Smoker	BloodPressure	Trials
1	YPL-320	'SMITH'	Male	38	176	1	124 93 18
2	GLI-532	'JOHNSON'	Male	43	163	0	109 77 [11,13,22]
3	PNI-258	'WILLIAMS'	Female	38	131	0	125 83 []
4	MIJ-579	'JONES'	Female	40	133	0	117 75 [6,12]
5	XLK-030	'BROWN'	Female	49	119	0	122 80 [14,23]
6	TFP-518	'DAVIS'	Female	46	142	0	121 70 19
7	LPD-746	'MILLER'	Female	33	142	1	130 88 13
8	ATA-945	'WILSON'	Male	40	180	0	115 82 []
9	VNL-702	'MOORE'	Male	28	183	0	115 78 2
10	LQW-768	'TAYLOR'	Female	31	132	0	118 86 11

In the Variables editor, you can see the names of the seven variables along the top row, and the observations names down the first column.

## Modify Variable and Observation Names

You can modify variable and observation names by double-clicking a name, and then typing new text.

	1	2
	LastName	Sex
1 YPL-320	'SMITH'	Male
2 GLI-532	'JOHNSON'	Male
3 PNI-258	'WILLIAMS'	Female

	1	2
	LastName	Gender
1 YPL-320	'SMITH'	Male
2 GLI-532	'JOHNSON'	Male
3 PNI-258	'WILLIAMS'	Female
4 MIJ-579	'JONES'	Female
5 XLK-030	'BROWN'	Female

All changes made in the Variables editor are also sent to the command line.

```
Command Window
>> hospital.Properties.VarNames{2} = 'Gender';
fx >>
```

The sixth variable in the data set, `BloodPressure`, is a numeric array with two columns. The first column shows systolic blood pressure, and the second column shows diastolic blood pressure. Click the arrow that appears on the right side of the variable name cell to see the units and description of the variable. You can type directly in the units and description fields to modify the text. The variable data type and size are shown under the variable description.

6	7	8	9
BloodPressure	Trials		
124	Ascending		
109	Descending		
125	UNITS		
117	mm Hg		
122	DESCRIPTION		
121	Systolic/Diastolic		
130			
115			
115			

## Reorder or Delete Variables

You can reorder variables in a dataset array using the Variables editor. Hover over the left side of a variable name cell until a four-headed arrow appears.

4	5
Weight	Smoker
176	1
163	0
131	0
133	0
119	0
142	0

After the arrow appears, click and drag the variable column to a new location.

4 Weight	5 Smoke	5 Smoker	6 BloodPressure
176	1	1	124
163	0	0	109
131	0	0	125
133	0	0	117
119	0	0	122
142	0	0	121
142	1	1	130
180	0	0	115
183	0	0	115
132	0	0	118
128	0	0	114
137	0	0	115
174	0	0	127

5 BloodPressure	6 Smoker	7 Trials
124	93	1 18
109	77	0 [11,13,22]
125	83	0 []
117	75	0 [6,12]
122	80	0 [14,23]
121	70	0 19
130	88	1 17

The command for the variable reordering appears in the command line.

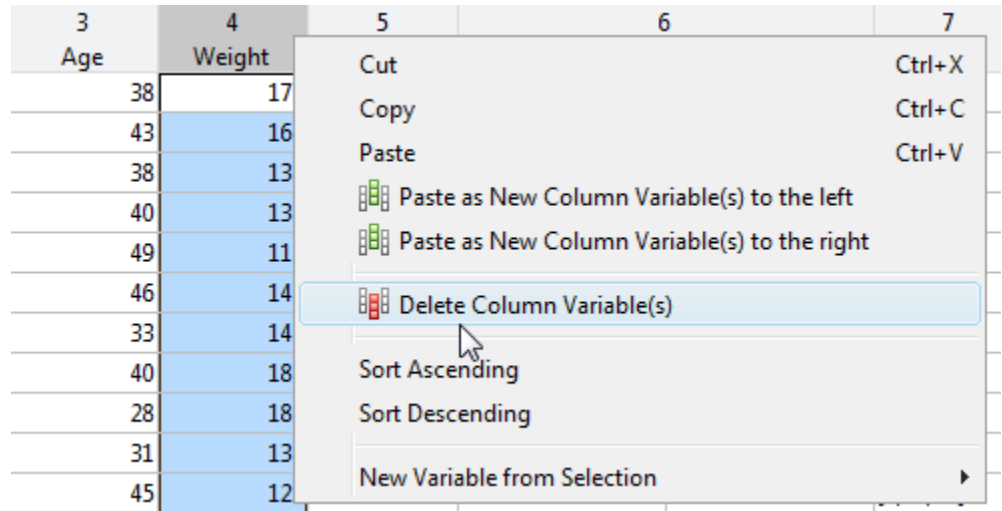
```

Command Window
>> hospital = hospital(:, [1:4 6 5 end]);
fx >>

```



You can delete a variable in the Variables editor by selecting the variable column, right-clicking, and selecting **Delete Column Variable(s)**.



The command for the variable deletion appears in the command line.

```
Command Window
>> hospital(:, 'Weight') = [];
fx >>
```

## Add New Data

You can enter new data values directly into the Variables editor. For example, you can add a new patient observation to the `hospital` data set. To enter a new last name, add a string to the end of the variable `LastName`.

100	ZZB-405	'HAYES'	Male	48	114	86
101		'JONES'				
102						

The variable `Gender` is a nominal array. The levels of the categorical variable appear in a drop-down list when you double-click a cell in the `Gender` column.

You can choose one of the levels previously used, or create a new level by selecting **New Item**.

100	ZZB-405	'HAYES'	Male
101	Obs101	'JONES'	Female
102			Female
103			Male
104			New item
105			

You can continue to add data for the remaining variables.

To change the observation name, click the observation name and type the new name.

100	ZZB-405	'HAYES'	Male	48
101	Obs101	'JONES'	Female	45
102				

The commands for entering the new data appear at the command line.

```

Command Window
>> hospital.LastName{101} = 'JONES';
Warning: Observations with default values added to dataset
variables.
> In dataset.subsasgn at 584
>> hospital.Sex(101) = 'Female';
>> hospital.Age(101) = 45;
>> hospital.BloodPressure(101,2) = 85;
>> hospital.BloodPressure(101,1) = 120;
>> hospital.Properties.ObsNames{101} = 'QPO-187';
fx >>
    
```

Notice the warning that appears after the first assignment. When you enter the first piece of data in the new observation row—here, the last name—default values are assigned to all other variables. Default assignments are:

- 0 for numeric variables
- <undefined> for categorical variables
- [ ] for cell arrays

You can also copy and paste data from one dataset array to another using the Variables editor.

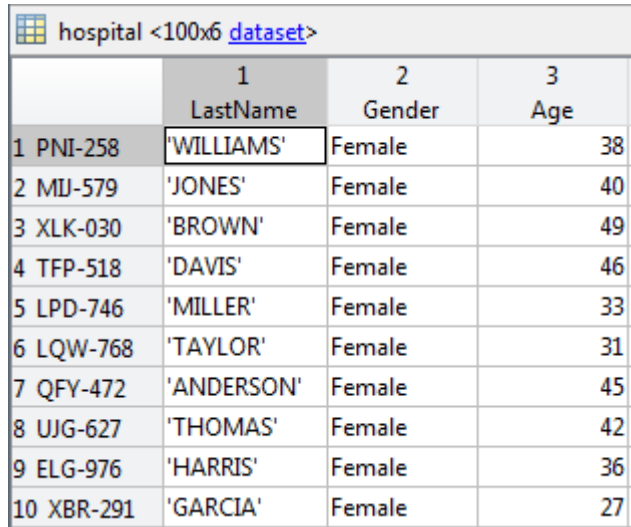
## Sort Observations

You can use the Variables editor to sort dataset array observations by the values of one or more variables. To sort by gender, for example, select the variable Gender. Then click **Sort**, and choose to sort rows by ascending or descending values of the selected variable.

The screenshot shows the Variables Editor interface with the 'VIEW' tab selected. The 'Sort' menu is open, showing options for 'Sort Ascending by Selected Columns' and 'Sort Descending by Selected Columns'. The data table below shows the 'hospital' dataset with columns: LastName, Gender, Age, BloodPressure, Smoker, and Trials. The rows are sorted by Gender, with females grouped together and males grouped together.

	1 LastName	2 Gender	3 Age	4 BloodPressure	5 Smoker	6 Trials		
1	YPL-320	'SMITH'	Male	38	124	93	1	18
2	GLI-532	'JOHNSON'	Male	43	109	77	0	[11,13,22]
3	PNI-258	'WILLIAMS'	Female	38	125	83	0	[]
4	MIJ-579	'JONES'	Female	40	117	75	0	[6,12]
5	XLK-030	'BROWN'	Female	49	122	80	0	[14,23]

When sorting by variables that are cell arrays of strings or of nominal data type, observations are sorted alphabetically. For ordinal variables, rows are sorted by the ordering of the levels. For example, when the observations of `hospital` are sorted by the values in `Gender`, the females are grouped together, followed by the males.



	1 LastName	2 Gender	3 Age	
1	PNI-258	'WILLIAMS'	Female	38
2	MU-579	'JONES'	Female	40
3	XLK-030	'BROWN'	Female	49
4	TFP-518	'DAVIS'	Female	46
5	LPD-746	'MILLER'	Female	33
6	LQW-768	'TAYLOR'	Female	31
7	QFY-472	'ANDERSON'	Female	45
8	UJG-627	'THOMAS'	Female	42
9	ELG-976	'HARRIS'	Female	36
10	XBR-291	'GARCIA'	Female	27

To sort by the values of multiple variables, press **Ctrl** while you select multiple variables.

When you use the Variables editor to sort rows, it is the same as calling `sortrows`. You can see this at the command line after executing the sorting.

```
Command Window
>> hospital = sortrows(hospital, 'Gender', 'ascend');
fx >>
```

### Select a Subset of Data

You can select a subset of data from a dataset array in the Variables editor, and create a new dataset array from the selection. For example, to create a dataset array containing only the variables `LastName` and `Age`:

- 1 Hold **Ctrl** while you click the variables `LastName` and `Age`.
- 2 Right-click, and select **New Variable from Selection > New Dataset Array**.

hospital <100x6 dataset>							
	1	2	3	4	5	6	7
	LastName	Gender	Age	BloodPressure	Smoker	Trials	
1	PNI-258	'WILLIAMS'	Female				
2	MJ-579	'JONES'	Female				
3	XLK-030	'BROWN'	Female				
4	TFP-518	'DAVIS'	Female				
5	LPD-746	'MILLER'	Female				
6	LQW-768	'TAYLOR'	Female				
7	QFY-472	'ANDERSON'	Female				
8	UJG-627	'THOMAS'	Female				
9	ELG-976	'HARRIS'	Female				
10	XBR-291	'GARCIA'	Female				
11	BKD-785	'CLARK'	Female				
12	JHV-416	'RODRIGUEZ'	Female				
13	VWL-936	'LEWIS'	Female	41	114	88	0 8
14	AAX-056	'LEE'	Female	44	128	90	1 []
15	DTT-578	'WALKER'	Female	28	129	96	17

The new dataset array appears in the Workspace window with the name LastName. The Command Window shows the commands that execute the selection.

Workspace		Command Window
Name	Value	
Description	'Simulated hospital data'	>> LastName = hospital(:, [1,3]);
LastName	<100x2 dataset>	fx >>
hospital	<100x6 dataset>	

You can use the same steps to select any subset of data. To select observations according to some logical condition, you can use a combination of sorting and selecting. For example, to create a new dataset array containing only males aged 45 and older:

- 1 Sort the observations of hospital by the values in Gender and Age, descending.
- 2 Select the male observations with age 45 and older.

hospital <100x6 dataset>

	1 LastName	2 Gender	3 Age	4 BloodPressure	5 Smoker	6 Trials
1 XBA-581	'ROBINSON'	Male	50	125	76	0 [20,26,30]
2 DAU-529	'REED'	Male	50	129	89	1 22
3 FLJ-908	'STEWART'	Male	49	129	95	1 15
4 MEZ-469	'GRIFFIN'	Male	49	119	74	0 9
5 KOQ-996	'MARTIN'	Male	48	130	92	1 [13,15,21,27]
6 FCD-425	'GONZALES'	Male	48	123	79	0 []
7 ZZB-405	'HAYES'	Male	48	114	86	0 28
8 YVY-570	'SCOTT'	Male	47	127	84	0 8
9 VDZ-577	'PHILLIPS'	Male	45	117	89	0 [10,15,17,19]
10 WCJ-997	'BELL'	Male	45	138	82	1 7
11 AGR-528	'SIMMONS'	Male	45	124	77	0 [8,18,25]
12 BEZ-311	'DIAZ'	Male	45	136	93	1 []
13 MSL-692	'GREEN'	Male	44	121	92	0 []

- 3** Right-click, and select **New Variables from Selection > New Dataset Array**. The new dataset array, LastName1, is created in the Workspace window.

Workspace

Name	Value
Description	'Simulated hospital data'
LastName	<100x2 dataset>
LastName1	<12x6 dataset>
hospital	<100x6 dataset>

Command Window

```
>> hospital = sortrows(hospital, {'Gender', 'Age'}, 'descend');
>> LastName1 = hospital(1:12, 1:6);
fx >>
```

- 4** You can rename the dataset array in the Workspace window.

Workspace

Name	Value
Description	'Simulated hospital data'
LastName	<100x2 dataset>
Male45	<12x6 dataset>
hospital	<100x6 dataset>

## Create Plots

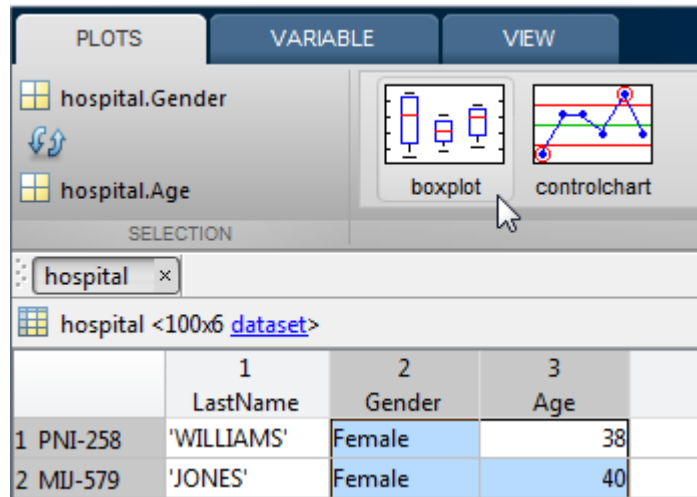
You can plot data from a dataset array using plotting options in the Variables editor. Available plot choices depend on the data types of variables to be plotted.

For example, if you select the variable **Age**, you can see in the **Plots** tab some plotting options that are appropriate for a univariate, numeric variable.

The screenshot shows the Variables Editor interface. The 'PLOTS' tab is active, displaying five plot options: 'plot' (line graph), 'bar' (bar chart), 'area' (area chart), 'pie' (pie chart), and 'hist' (histogram). Below the plot options is a 'SELECTION' area with a dropdown menu showing 'hospital' and a table view of the dataset. The table has columns for LastName, Gender, Age, BloodPressure, and Sm, with the Age column highlighted in blue.

	1	2	3	4		
	LastName	Gender	Age	BloodPressure	Sm	
1	PNI-258	'WILLIAMS'	Female	38	125	83
2	MJ-579	'JONES'	Female	40	117	75
3	XLK-030	'BROWN'	Female	49	122	80
4	TFP-518	'DAVIS'	Female	46	121	70
5	LPD-746	'MILLER'	Female	33	130	88

Sometimes, there are plot options for multiple variables, depending on their data types. For example, if you select both **Age** and **Gender**, you can draw box plots of age, grouped by gender.



The screenshot shows the SAS software interface. At the top, there are three tabs: 'PLOTS', 'VARIABLE', and 'VIEW'. Under the 'PLOTS' tab, two variables are listed: 'hospital.Gender' and 'hospital.Age'. To the right of these variables are two plot icons: a 'boxplot' icon and a 'controlchart' icon. Below the plot icons is a 'SELECTION' section with a search box containing 'hospital' and a dropdown menu showing 'hospital <100x6 dataset>'. At the bottom of the screenshot is a table view of the 'hospital' dataset with the following data:

	1	2	3	
	LastName	Gender	Age	
1	PNI-258	'WILLIAMS'	Female	38
2	MIJ-579	'JONES'	Female	40

### See Also

dataset | sortrows |

### Related Examples

- “Add and Delete Observations” on page 2-79
- “Add and Delete Variables” on page 2-83
- “Access Data in Dataset Array Variables” on page 2-87
- “Select Subsets of Observations” on page 2-93
- “Sort Observations in Dataset Arrays” on page 2-97

### Concepts

- “Dataset Arrays” on page 2-135



## Dataset Arrays

### In this section...

“What Are Dataset Arrays?” on page 2-135

“Dataset Array Conversion” on page 2-135

“Dataset Array Properties” on page 2-136

### What Are Dataset Arrays?

Statistics Toolbox has *dataset arrays* for storing variables with heterogeneous data types. For example, you can combine numeric data, logical data, cell arrays of strings, and categorical arrays in one dataset array variable.

Within a dataset array, each variable (column) must be one homogeneous data type, but the different variables can be of heterogeneous data types. A dataset array is usually interpreted as a set of variables measured on many units of observation. That is, each row in a dataset array corresponds to an observation, and each column to a variable. In this sense, a dataset array organizes data like a typical spreadsheet.

Dataset arrays are a unique data type, with a corresponding set of valid operations. Even if a dataset array contains only numeric variables, you cannot operate on the dataset array like a numeric variable. The valid operations for dataset arrays are the methods of the `dataset` class.

### Dataset Array Conversion

You can create a dataset array by combining variables that exist in the MATLAB workspace, or directly importing data from a file, such as a text file or spreadsheet. This table summarizes the functions you can use to create dataset arrays.

Data Source	Conversion to Dataset Array
Data from a file	<code>dataset</code>
Heterogeneous collection of workspace variables	<code>dataset</code>

<b>Data Source</b>	<b>Conversion to Dataset Array</b>
Numeric array	mat2dataset
Cell array	cell2dataset
Structure array	struct2dataset
Table	table2dataset

You can export dataset arrays to text or spreadsheet files using `export`. To convert a dataset array to a cell array or structure array, use `dataset2cell` or `dataset2struct`. To convert a dataset array to a table, use `dataset2table`.

## **Dataset Array Properties**

In addition to storing data in a dataset array, you can store metadata such as:

- Variable and observation names
- Data descriptions
- Units of measurement
- Variable descriptions

This information is stored as dataset array properties. For a dataset array named `ds`, you can view the dataset array metadata by entering `ds.Properties` at the command line. You can access a specific property, such as variable names—property `VarNames`—using `ds.Properties.VarNames`. You can both retrieve and modify property values using this syntax.

Variable and observation names are included in the display of a dataset array. Variable names display across the top row, and observation names, if present, appear in the first column. Note that variable and observation names do not affect the size of a dataset array.

### **See Also**

`dataset` | `export` | `dataset2cell` | `dataset2struct` | `dataset2table` | `mat2dataset` | `cell2dataset` | `struct2dataset` | `table2dataset`

### **Related Examples**

- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Create a Dataset Array from a File” on page 2-71
- “Export Dataset Arrays” on page 2-115

- “Dataset Arrays in the Variables Editor” on page 2-122
- “Index and Search Dataset Arrays” on page 2-138

## Index and Search Dataset Arrays

### Ways To Index and Search

There are many ways to index into dataset arrays. For example, for a dataset array, `ds`, you can:

- Use `()` to create a new dataset array from a subset of `ds`. For example, `ds1 = ds(1:5, :)` creates a new dataset array, `ds1`, consisting of the first five rows of `ds`. Metadata, including variable and observation names, transfers to the new dataset array.
- Use variable names with dot notation to index individual variables in a dataset array. For example, `ds.Height` indexes the variable named `Height`.
- Use observation names to index individual observations in a dataset array. For example, `ds('Obs1', :)` gives data for the observation named `Obs1`.
- Use observation or variable numbers. For example, `ds(:, [1,3,5])` gives the data in the first, third, and fifth variables (columns) of `ds`.
- Use logical indexing to search for observations in `ds` that satisfy a logical condition. For example, `ds(ds.Gender=='Male', :)` gives the observations in `ds` where the variable named `Gender`, a nominal array, has the value `Male`.
- Use `ismissing` to find missing data in the dataset array.

### Common Indexing and Searching

The following brief examples illustrate several indexing and searching methods.

#### Load sample data.

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

```
ans =
```

```
    100     7
```

The dataset array has 100 observations and 7 variables.

**Index a variable by name.** Return the minimum age in the dataset array.

```
min(hospital.Age)
```

```
ans =
```

```
    25
```

Delete the variable Trials.

```
hospital.Trials = [];
size(hospital)
```

```
ans =
```

```
    100     6
```

**Index an observation by name.** Display measurements on the first five variables for the observation named PUE-347.

```
hospital('PUE-347',1:5)
```

```
ans =
```

	LastName	Sex	Age	Weight	Smoker
PUE-347	'YOUNG'	Female	25	114	false

**Index variables by number.** Create a new dataset array containing the first four variables of hospital.

```
dsNew = hospital(:,1:4);
dsNew.Properties.VarNames(:)
```

```
ans =
```

```
'LastName'
'Sex'
'Age'
'Weight'
```

**Index observations by number.** Delete the last 10 observations.

```
hospital(end-9:end,:) = [];  
size(hospital)
```

```
ans =
```

```
90    7
```

**Search for observations by logical condition.** Create a new dataset array containing only females who smoke.

```
dsFS = hospital(hospital.Sex=='Female' & hospital.Smoker==true,:);  
dsFS(:,{'LastName','Sex','Smoker'})
```

```
ans =
```

	LastName	Sex	Smoker
LPD-746	'MILLER'	Female	true
XBR-291	'GARCIA'	Female	true
AAX-056	'LEE'	Female	true
DTT-578	'WALKER'	Female	true
AFK-336	'WRIGHT'	Female	true
RBA-579	'SANCHEZ'	Female	true
HAK-381	'MORRIS'	Female	true
NSK-403	'RAMIREZ'	Female	true
ILS-109	'WATSON'	Female	true
JDR-456	'SANDERS'	Female	true
HWZ-321	'PATTERSON'	Female	true
GGU-691	'HUGHES'	Female	true
WUS-105	'FLORES'	Female	true

**See Also** [dataset](#) |

### Related Examples

- “Access Data in Dataset Array Variables” on page 2-87
- “Select Subsets of Observations” on page 2-93

### Concepts

- “Dataset Arrays” on page 2-135

# Descriptive Statistics

---

- “Introduction to Descriptive Statistics” on page 3-2
- “Measures of Central Tendency” on page 3-3
- “Measures of Dispersion” on page 3-5
- “Quantiles and Percentiles” on page 3-7
- “Exploratory Analysis of Data” on page 3-11
- “Resampling Statistics” on page 3-16
- “Data with Missing Values” on page 3-21

## **Introduction to Descriptive Statistics**

You may need to summarize large, complex data sets—both numerically and visually—to convey their essence to the data analyst and to allow for further processing.



## Measures of Central Tendency

Measures of central tendency locate a distribution of data along an appropriate scale.

The following table lists the functions that calculate the measures of central tendency.

Function Name	Description
geomean	Geometric mean
harmmean	Harmonic mean
mean	Arithmetic average
median	50th percentile
mode	Most frequent value
trimmean	Trimmed mean

The average is a simple and popular estimate of location. If the data sample comes from a normal distribution, then the sample mean is also optimal (minimum variance unbiased estimator (MVUE) of  $\mu$ ).

Unfortunately, outliers, data entry errors, or glitches exist in almost all real data. The sample mean is sensitive to these problems. One bad data value can move the average away from the center of the rest of the data by an arbitrarily large distance.

The median and trimmed mean are two measures that are resistant (robust) to outliers. The median is the 50th percentile of the sample, which will only change slightly if you add a large perturbation to any value. The idea behind the trimmed mean is to ignore a small percentage of the highest and lowest values of a sample when determining the center of the sample.

The geometric mean and harmonic mean, like the average, are not robust to outliers. They are useful when the sample is distributed lognormal or heavily skewed.

The following example shows the behavior of the measures of location for a sample with one outlier.

```
x = [ones(1,6) 100]
```

```
x =  
    1    1    1    1    1    1   100
```

```
locate = [geomean(x) harmmean(x) mean(x) median(x)...  
          trimmean(x,25)]
```

```
locate =  
    1.9307    1.1647   15.1429    1.0000    1.0000
```

You can see that the mean is far from any data value because of the influence of the outlier. The median and trimmed mean ignore the outlying value and describe the location of the rest of the data values.

## Measures of Dispersion

The purpose of measures of dispersion is to find out how spread out the data values are on the number line. Another term for these statistics is measures of spread.

The table gives the function names and descriptions.

Function Name	Description
iqr	Interquartile range
mad	Mean absolute deviation
moment	Central moment of all orders
range	Range
std	Standard deviation
var	Variance

The range (the difference between the maximum and minimum values) is the simplest measure of spread. But if there is an outlier in the data, it will be the minimum or maximum value. Thus, the range is not robust to outliers.

The standard deviation and the variance are popular measures of spread that are optimal for normally distributed samples. The sample variance is the minimum variance unbiased estimator (MVUE) of the normal parameter  $\sigma^2$ . The standard deviation is the square root of the variance and has the desirable property of being in the same units as the data. That is, if the data is in meters, the standard deviation is in meters as well. The variance is in meters<sup>2</sup>, which is more difficult to interpret.

Neither the standard deviation nor the variance is robust to outliers. A data value that is separate from the body of the data can increase the value of the statistics by an arbitrarily large amount.

The mean absolute deviation (MAD) is also sensitive to outliers. But the MAD does not move quite as much as the standard deviation or variance in response to bad data.

The interquartile range (IQR) is the difference between the 75th and 25th percentile of the data. Since only the middle 50% of the data affects this measure, it is robust to outliers.

The following example shows the behavior of the measures of dispersion for a sample with one outlier.

```
x = [ones(1,6) 100]

x =
     1     1     1     1     1     1    100

stats = [iqr(x) mad(x) range(x) std(x)]

stats =
     0    24.2449    99.0000    37.4185
```

## Quantiles and Percentiles

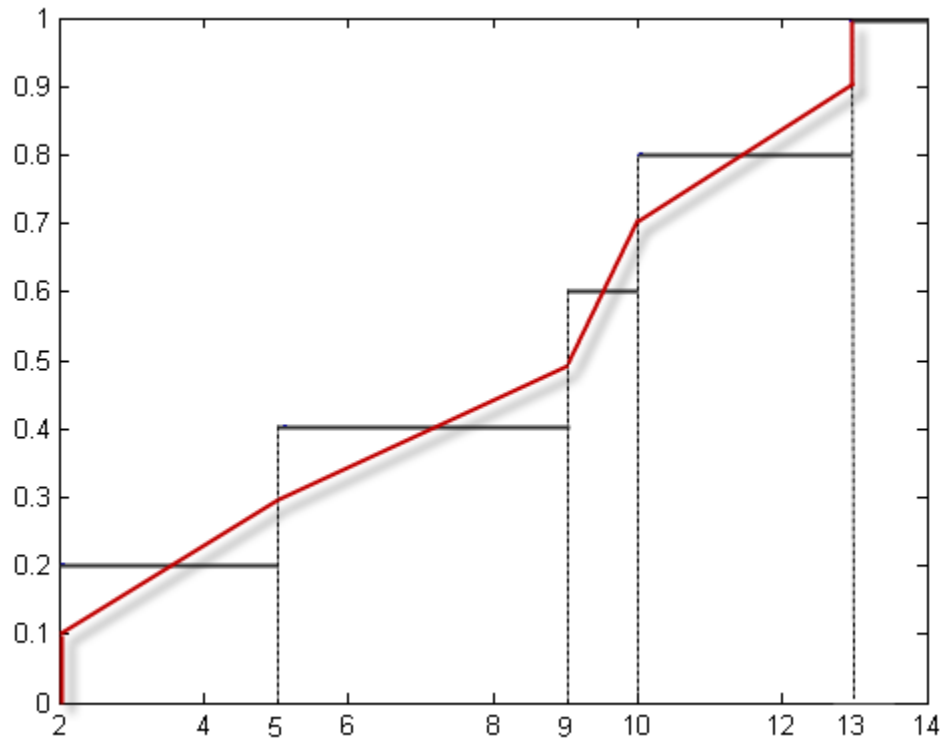
This section explains how the Statistics Toolbox functions `quantile` and `prctile` compute quantiles and percentiles.

The `prctile` function calculates the percentiles in a similar way as `quantile` calculates quantiles. The following steps in the computation of quantiles are also true for percentiles, given the fact that, for the same data sample, the quantile at the value  $Q$  is the same as the percentile at the value  $P = 100 * Q$ .

**1** `quantile` initially assigns the sorted values in  $X$  to the  $(0.5/n)$ ,  $(1.5/n)$ , ...,  $([n - 0.5]/n)$  quantiles. For example:

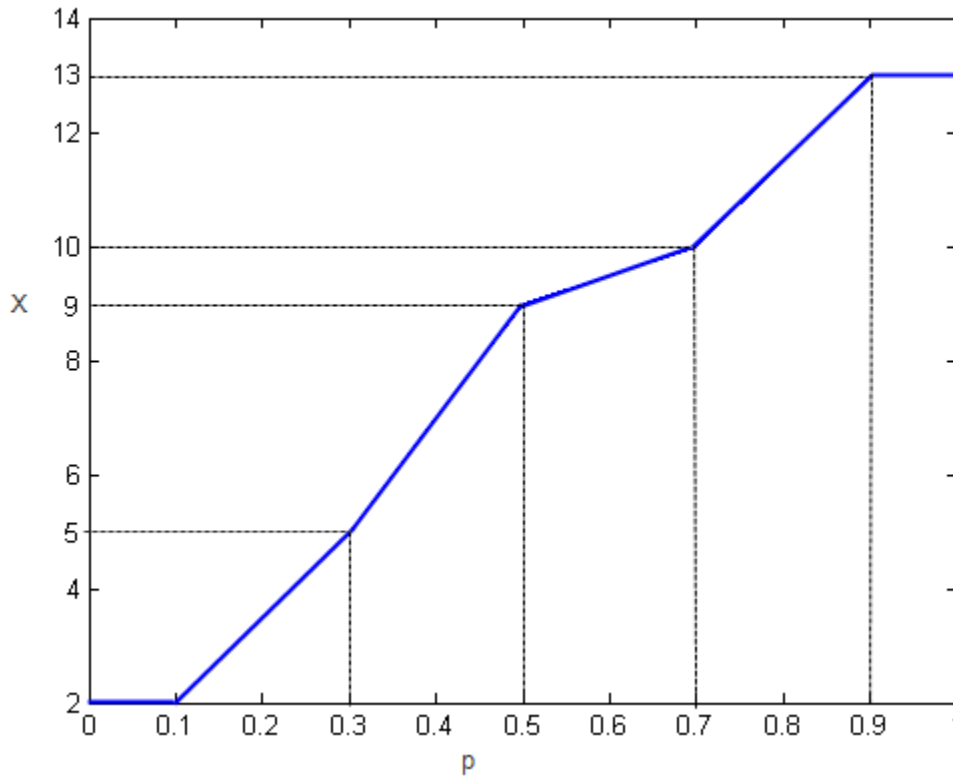
- For a data vector of six elements such as {6, 3, 2, 10, 8, 1}, the sorted elements {1, 2, 3, 6, 8, 10} respectively correspond to the  $(0.5/6)$ ,  $(1.5/6)$ ,  $(2.5/6)$ ,  $(3.5/6)$ ,  $(4.5/6)$ , and  $(5.5/6)$  quantiles.
- For a data vector of five elements such as {2, 10, 5, 9, 13}, the sorted elements {2, 5, 9, 10, 13} respectively correspond to the 0.1, 0.3, 0.5, 0.7, and 0.9 quantiles.

The following figure illustrates this approach for data vector  $X = \{2, 10, 5, 9, 13\}$ . The first observation corresponds to the cumulative probability  $1/5 = 0.2$ , the second observation corresponds to the cumulative probability  $2/5 = 0.4$ , and so on. The step function in this figure shows these cumulative probabilities. `quantile` instead places the observations in midpoints, such that the first corresponds to  $0.5/5 = 0.1$ , the second corresponds to  $1.5/5 = 0.3$ , and so on, and then connects these midpoints. The red lines in the following figure connect the midpoints.



**Assigning Observations to Quantiles**

By switching the axes, as the next figure, you can see the values of the variable  $X$  that correspond to the  $p$  quantiles.



### Quantiles of X

- 2** quantile finds any quantiles between the data values using linear interpolation.

*Linear interpolation* uses linear polynomials to approximate a function  $f(x)$  and construct new data points within the range of a known set of data points. Algebraically, given the data points  $(x_1, y_1)$  and  $(x_2, y_2)$ , where  $y_1 = f(x_1)$  and  $y_2 = f(x_2)$ , linear interpolation finds  $y = f(x)$  for a given  $x$  between  $x_1$  and  $x_2$  as follows:

$$y = f(x) = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1).$$

Similarly, if the  $1.5/n$  quantile is  $y_{1.5/n}$  and the  $2.5/n$  quantile is  $y_{2.5/n}$ , then linear interpolation finds the  $2.3/n$  quantile  $y_{2.3/n}$  as

$$y_{\frac{2.3}{n}} = y_{\frac{1.5}{n}} + \frac{\left(\frac{2.3}{n} - \frac{1.5}{n}\right)}{\left(\frac{2.5}{n} - \frac{1.5}{n}\right)} \left( y_{\frac{2.5}{n}} - y_{\frac{1.5}{n}} \right).$$

**3** quantile assigns the first and last values of  $X$  to the quantiles for probabilities less than  $(0.5/n)$  and greater than  $([n-0.5]/n)$ , respectively.

**See Also**

quantile | prctile | median



## Exploratory Analysis of Data

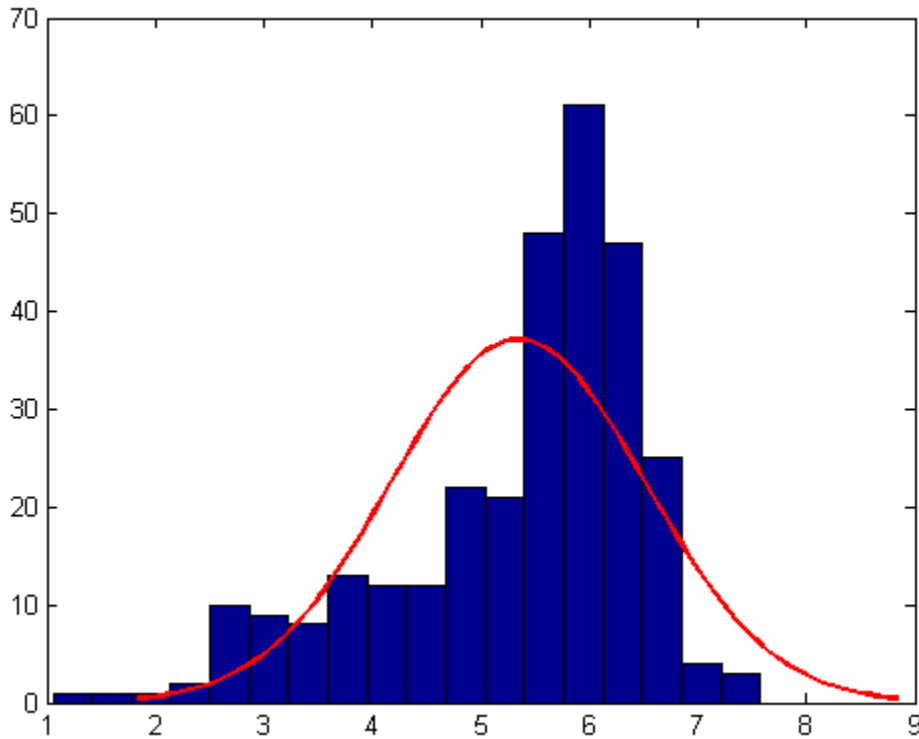
This example shows how to explore the distribution of data using descriptive statistics.

### Generate sample data.

```
rng('default') % for reproducibility  
x = [normrnd(4,1,1,100) normrnd(6,0.5,1,200)];
```

### Create a histogram of data with normal density fit.

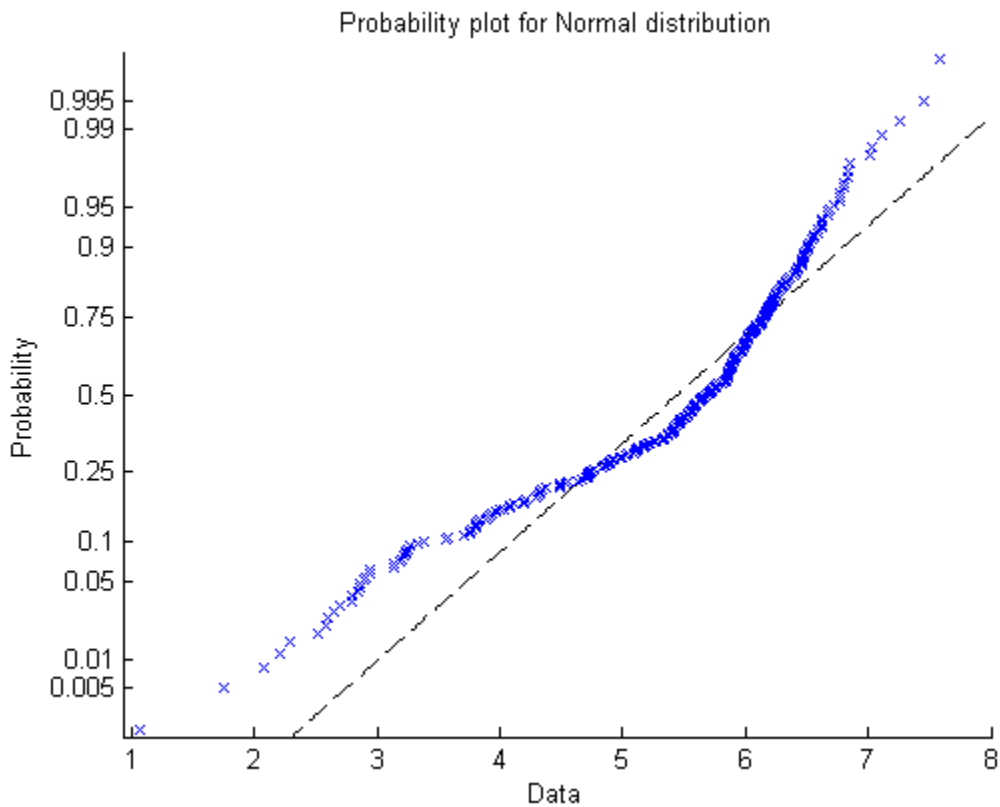
```
histfit(x)
```



The distribution of the data seems left skewed and normal distribution does not look like a good fit to this distribution.

**Obtain a normal probability plot.**

```
probplot('normal',x)
```



This probability plot also clearly shows the deviation of data from normality.

**Compute quantiles of data.**

```
p = 0:0.25:1;  
y = quantile(x,p);
```

```
z = [p;y]
```

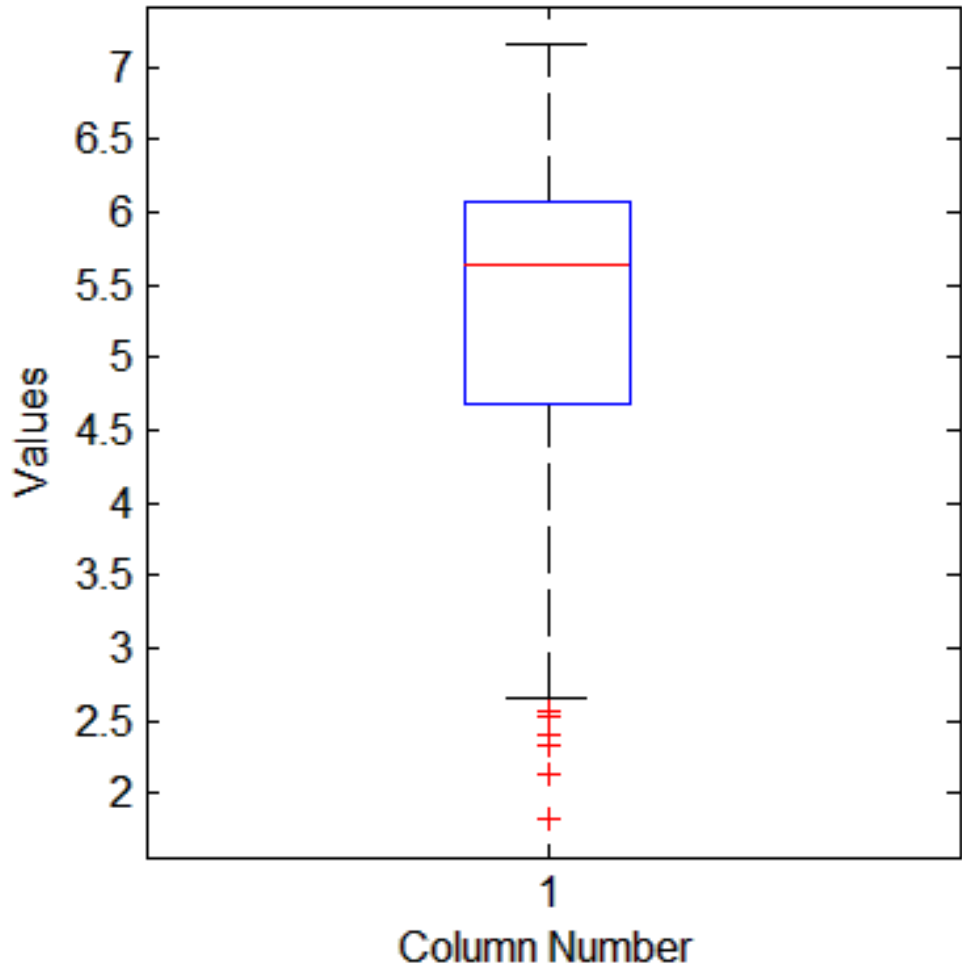
```
z =
```

```
          0    0.2500    0.5000    0.7500    1.0000
1.0557    4.7375    5.6872    6.1526    7.5784
```

**Plot a box plot.**

A box plot helps to visualize the statistics.

```
boxplot(x)
```



You can also see the 0.25, 0.5, and 0.75 quantiles in the box plot. The long lower tail and plus signs also show the lack of symmetry in the sample values.

**Compute the mean and median of data.**

`y = [mean(x) median(x)]`

`y =`

```
5.3438    5.6872
```

The mean and median values seem close to each other, but a mean smaller than the median usually flags left skewness of the data.

**Compute the skewness and kurtosis of data.**

```
y = [skewness(x) kurtosis(x)]
```

```
y =
```

```
-1.0417    3.5895
```

A negative skewness value means the data is left skewed. The data has a larger peakedness than a normal distribution because the kurtosis value is greater than 3.

**Identify possible outliers.**

Compute z-scores. Find the z-scores that are greater than 3 or less than  $-3$ .

```
Z = zscore(x);
find(abs(Z)>3);
```

```
ans =
```

```
3    35
```

The 3rd and 35th observations might be outliers.

**See Also**

```
quantile | prctile | histfit | boxplot | mean | median | kurtosis
| skewness
```

**Concepts**

- “Box Plots” on page 4-7
- “Measures of Central Tendency” on page 3-3
- “Measures of Dispersion” on page 3-5
- “Quantiles and Percentiles” on page 3-7

## Resampling Statistics

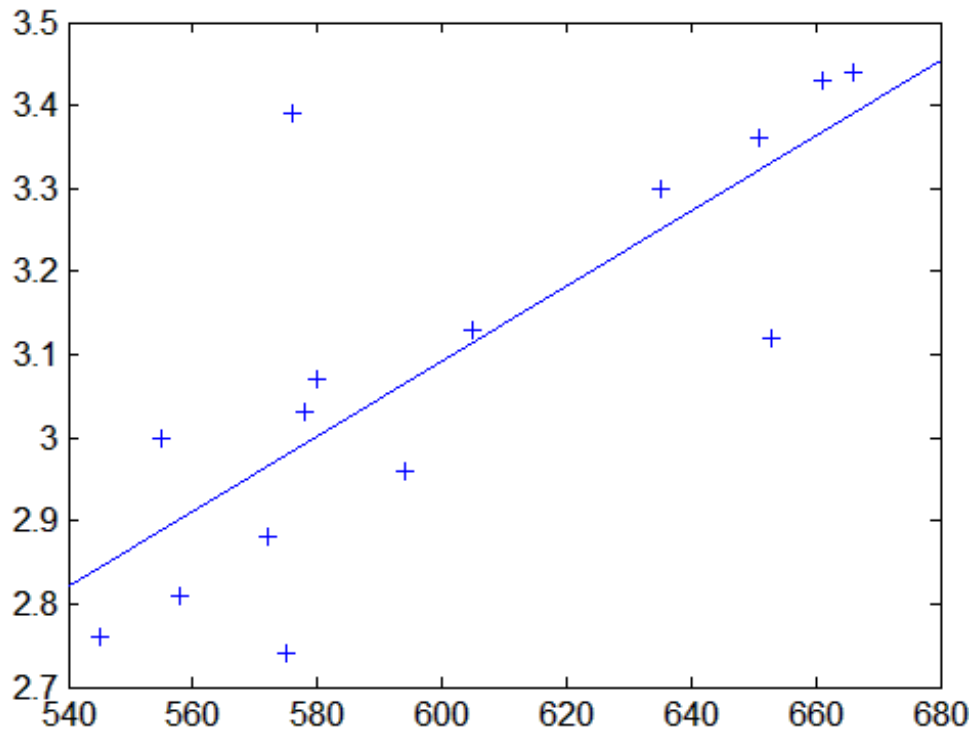
In this section...
“Bootstrap” on page 3-16
“Jackknife” on page 3-19
“Parallel Computing Support for Resampling Methods” on page 3-20

### Bootstrap

The *bootstrap* procedure involves choosing random samples with replacement from a data set and analyzing each sample the same way. Sampling with replacement means that each observation is selected separately at random from the original dataset. So a particular data point from the original data set could appear multiple times in a given bootstrap sample. The number of elements in each bootstrap sample equals the number of elements in the original data set. The range of sample estimates you obtain enables you to establish the uncertainty of the quantity you are estimating.

This example from Efron and Tibshirani [33] compares Law School Admission Test (LSAT) scores and subsequent law school grade point average (GPA) for a sample of 15 law schools.

```
load lawdata
plot(lsat,gpa, '+')
lsline
```



The least-squares fit line indicates that higher LSAT scores go with higher law school GPAs. But how certain is this conclusion? The plot provides some intuition, but nothing quantitative.

You can calculate the correlation coefficient of the variables using the `corr` function.

```
rho_hat = corr(lsat, gpa)
rho_hat =
    0.7764
```

Now you have a number describing the positive connection between LSAT and GPA; though it may seem large, you still do not know if it is statistically significant.

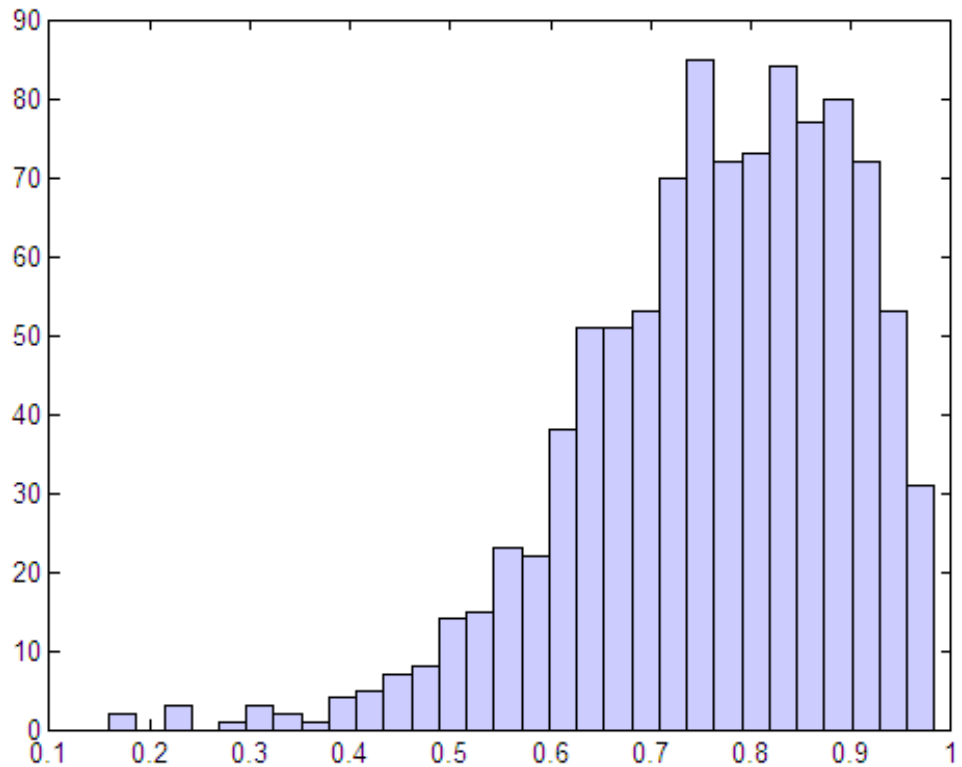
Using the `bootstrp` function you can resample the `lsat` and `gpa` vectors as many times as you like and consider the variation in the resulting correlation coefficients.

Here is an example.

```
rhos1000 = bootstrp(1000, 'corr', lsat, gpa);
```

This command resamples the `lsat` and `gpa` vectors 1000 times and computes the `corr` function on each sample. Here is a histogram of the result.

```
hist(rhos1000, 30)  
set(get(gca, 'Children'), 'FaceColor', [.8 .8 1])
```



Nearly all the estimates lie on the interval  $[0.4 \ 1.0]$ .



It is often desirable to construct a confidence interval for a parameter estimate in statistical inferences. Using the `bootci` function, you can use bootstrapping to obtain a confidence interval. The confidence interval for the `lsat` and `gpa` data is computed as:

```
ci = bootci(5000,@corr,lsat,gpa)
```

```
ci =  
    0.3313  
    0.9427
```

Therefore, a 95% confidence interval for the correlation coefficient between LSAT and GPA is [0.33 0.94]. This is strong quantitative evidence that LSAT and subsequent GPA are positively correlated. Moreover, this evidence does not require any strong assumptions about the probability distribution of the correlation coefficient.

Although the `bootci` function computes the Bias Corrected and accelerated (BCa) interval as the default type, it is also able to compute various other types of bootstrap confidence intervals, such as the studentized bootstrap confidence interval.

## Jackknife

Similar to the bootstrap is the *jackknife*, which uses resampling to estimate the bias of a sample statistic. Sometimes it is also used to estimate standard error of the sample statistic. The jackknife is implemented by the Statistics Toolbox function `jackknife`.

The jackknife resamples systematically, rather than at random as the bootstrap does. For a sample with  $n$  points, the jackknife computes sample statistics on  $n$  separate samples of size  $n-1$ . Each sample is the original data with a single observation omitted.

In the previous bootstrap example you measured the uncertainty in estimating the correlation coefficient. You can use the jackknife to estimate the bias, which is the tendency of the sample correlation to over-estimate or under-estimate the true, unknown correlation. First compute the sample correlation on the data:

```
load lawdata
```

```
rhohat = corr(lsat,gpa)
```

```
rhohat =  
    0.7764
```

Next compute the correlations for jackknife samples, and compute their mean:

```
jackrho = jackknife(@corr,lsat,gpa);  
meanrho = mean(jackrho)
```

```
meanrho =  
    0.7759
```

Now compute an estimate of the bias:

```
n = length(lsat);  
biasrho = (n-1) * (meanrho-rhohat)
```

```
biasrho =  
   -0.0065
```

The sample correlation probably underestimates the true correlation by about this amount.

## **Parallel Computing Support for Resampling Methods**

For information on computing resampling statistics in parallel, see Parallel Computing Toolbox™.

## Data with Missing Values

Many data sets have one or more missing values. It is convenient to code missing values as NaN (Not a Number) to preserve the structure of data sets across multiple variables and observations.

For example:

```
X = magic(3);
X([1 5]) = [NaN NaN]
X =
     NaN     1     6
     3     NaN     7
     4     9     2
```

Normal MATLAB arithmetic operations yield NaN values when operands are NaN:

```
s1 = sum(X)
s1 =
     NaN     NaN    15
```

Removing the NaN values would destroy the matrix structure. Removing the rows containing the NaN values would discard data. Statistics Toolbox functions in the following table remove NaN values only for the purposes of computation.

Function	Description
nancov	Covariance matrix, ignoring NaN values
nanmax	Maximum, ignoring NaN values
nanmean	Mean, ignoring NaN values
nanmedian	Median, ignoring NaN values
nanmin	Minimum, ignoring NaN values
nanstd	Standard deviation, ignoring NaN values
nansum	Sum, ignoring NaN values
nanvar	Variance, ignoring NaN values

For example:

```
s2 = nansum(X)
s2 =
     7     10     15
```

Other Statistics Toolbox functions also ignore NaN values. These include `iqr`, `kurtosis`, `mad`, `prctile`, `range`, `skewness`, and `trimmean`.

# Statistical Visualization

---

- “Introduction to Statistical Visualization” on page 4-2
- “Create Scatter Plots Using Grouped Data” on page 4-3
- “Box Plots” on page 4-7
- “Distribution Plots” on page 4-9

## Introduction to Statistical Visualization

Statistics Toolbox data visualization functions add to the extensive graphics capabilities already in MATLAB.

- Scatter plots are a basic visualization tool for multivariate data. They are used to identify relationships among variables. Grouped versions of these plots use different plotting symbols to indicate group membership. The `gname` function is used to label points on these plots with a text label or an observation number.
- Box plots display a five-number summary of a set of data: the median, the two ends of the interquartile range (the box), and two extreme values (the whiskers) above and below the box. Because they show less detail than histograms, box plots are most useful for side-by-side comparisons of two distributions.
- Distribution plots help you identify an appropriate distribution family for your data. They include normal and Weibull probability plots, quantile-quantile plots, and empirical cumulative distribution plots.

Advanced Statistics Toolbox visualization functions are available for specialized statistical analyses.

## Create Scatter Plots Using Grouped Data

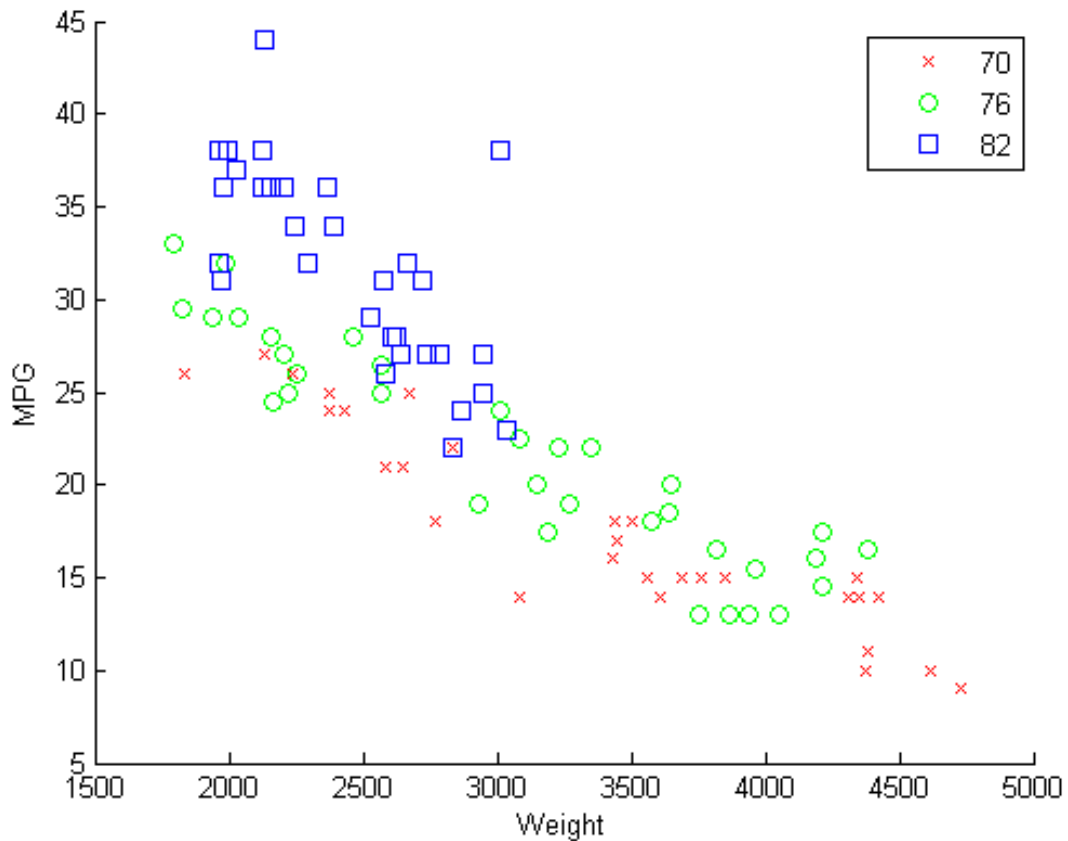
This example shows how to create scatter plots using grouped sample data.

A scatter plot is a simple plot of one variable against another. The MATLAB functions `plot` and `scatter` produce scatter plots. The MATLAB function `plotmatrix` can produce a matrix of such plots showing the relationship between several pairs of variables.

Statistics Toolbox functions `gscatter` and `gplotmatrix` produce grouped versions of these plots. These are useful for determining whether the values of two variables or the relationship between those variables is the same in each group.

Suppose you want to examine the weight and mileage of cars from three different model years.

```
load carsmall
gscatter(Weight,MPG,Model_Year,' ','xos')
```



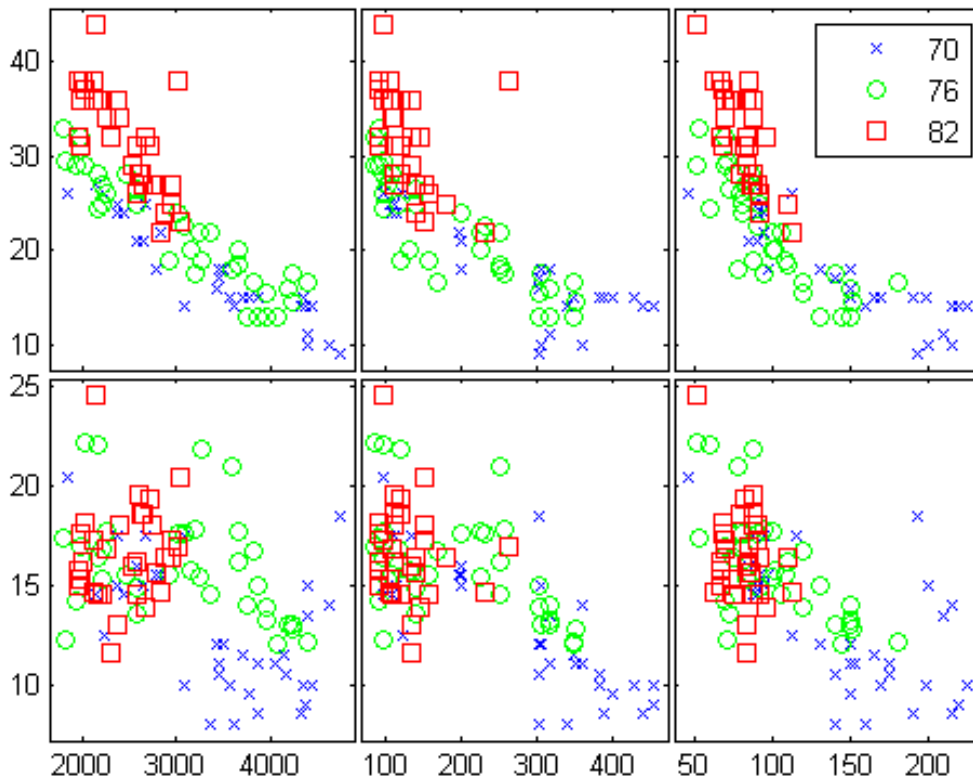
This shows that not only is there a strong relationship between the weight of a car and its mileage, but also that newer cars tend to be lighter and have better gas mileage than older cars.

The default arguments for `gscatter` produce a scatter plot with the different groups shown with the same symbol but different colors. The last two arguments above request that all groups be shown in default colors and with different symbols.



The carsmall data set contains other variables that describe different aspects of cars. You can examine several of them in a single display by creating a grouped plot matrix.

```
xvars = [Weight Displacement Horsepower];
yvars = [MPG Acceleration];
gplotmatrix(xvars,yvars,Model_Year,' ','xos')
```



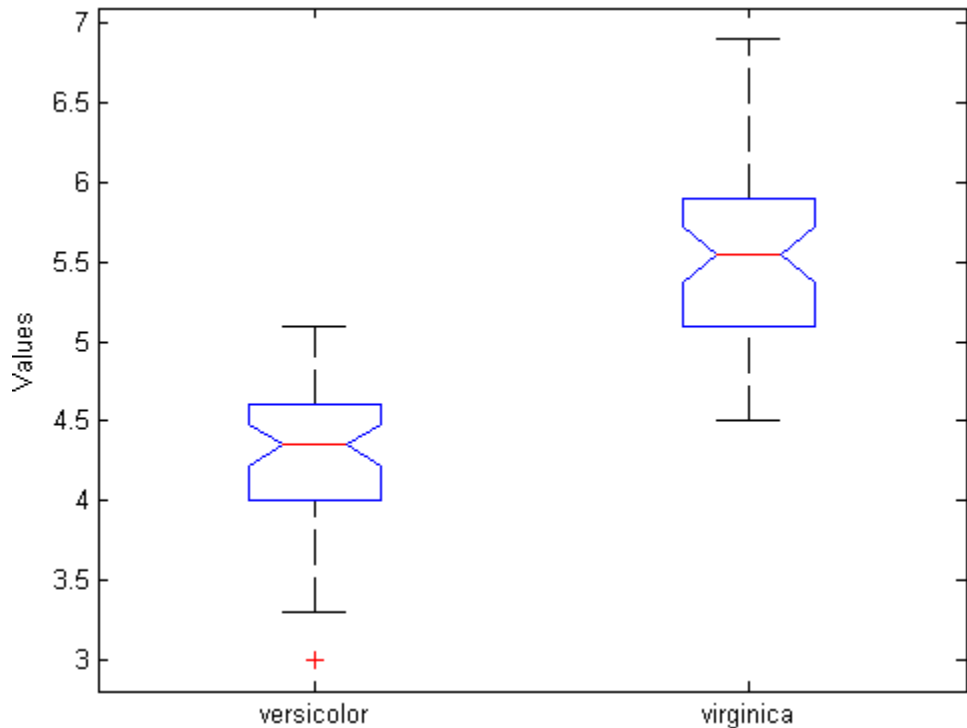
The upper right subplot displays MPG against Horsepower, and shows that over the years the horsepower of the cars has decreased but the gas mileage has improved.

The `gplotmatrix` function can also graph all pairs from a single list of variables, along with histograms for each variable. See MANOVA.

## Box Plots

The graph below, created with the `boxplot` command, compares petal lengths in samples from two species of iris.

```
load fisheriris
s1 = meas(51:100,3);
s2 = meas(101:150,3);
boxplot([s1 s2], 'notch', 'on', ...
        'labels', {'versicolor', 'virginica'})
```



This plot has the following features:

- The tops and bottoms of each “box” are the 25th and 75th percentiles of the samples, respectively. The distances between the tops and bottoms are the interquartile ranges.

- The line in the middle of each box is the sample median. If the median is not centered in the box, it shows sample skewness.
- The whiskers are lines extending above and below each box. Whiskers are drawn from the ends of the interquartile ranges to the furthest observations within the whisker length (the *adjacent values*).
- Observations beyond the whisker length are marked as outliers. By default, an outlier is a value that is more than 1.5 times the interquartile range away from the top or bottom of the box, but this value can be adjusted with additional input arguments. Outliers are displayed with a red + sign.
- Notches display the variability of the median between samples. The width of a notch is computed so that box plots whose notches do not overlap (as above) have different medians at the 5% significance level. The significance level is based on a normal distribution assumption, but comparisons of medians are reasonably robust for other distributions. Comparing box-plot medians is like a visual hypothesis test, analogous to the  $t$  test used for means.

## Distribution Plots

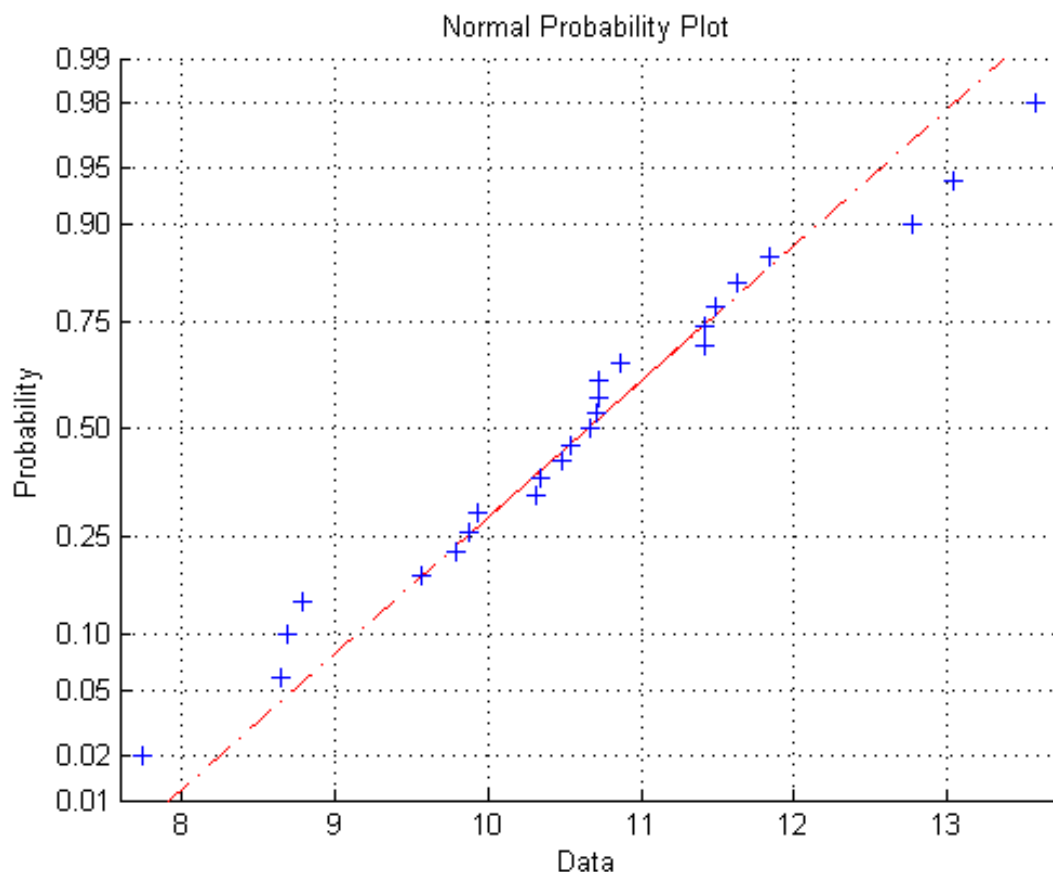
In this section...
“Normal Probability Plots” on page 4-9
“Quantile-Quantile Plots” on page 4-12
“Cumulative Distribution Plots” on page 4-15
“Other Probability Plots” on page 4-16

### Normal Probability Plots

Normal probability plots are used to assess whether data comes from a normal distribution. Many statistical procedures make the assumption that an underlying distribution is normal, so normal probability plots can provide some assurance that the assumption is justified, or else provide a warning of problems with the assumption. An analysis of normality typically combines normal probability plots with hypothesis tests for normality.

This example generates a data sample of 25 random numbers from a normal distribution with  $\mu = 10$  and  $\sigma = 1$ , and creates a normal probability plot of the data.

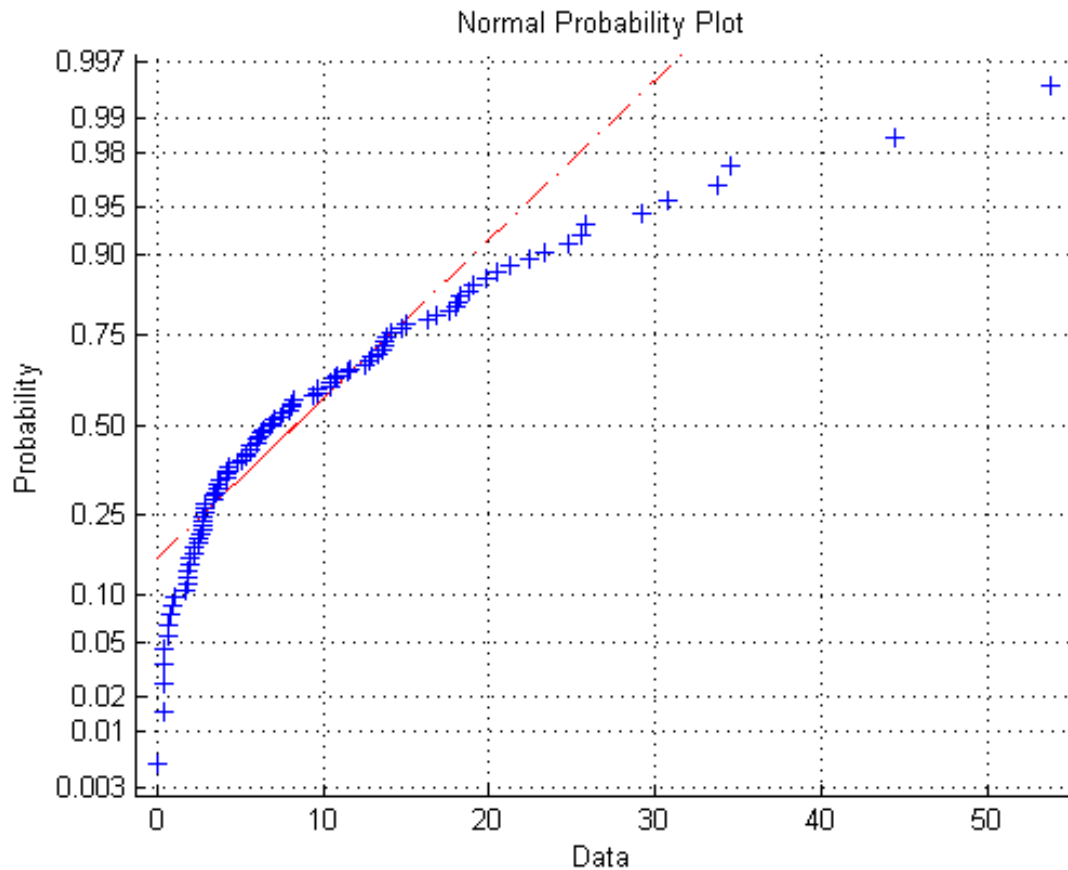
```
rng('default'); % For reproducibility
x = normrnd(10,1,25,1);
normplot(x)
```



The plus signs plot the empirical probability versus the data value for each point in the data. A solid line connects the 25th and 75th percentiles in the data, and a dashed line extends it to the ends of the data. The y-axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks on the y-axis matches the distance between the quantiles of a normal distribution. The quantiles are close together near the median (probability = 0.5) and stretch out symmetrically as you move away from the median.

In a normal probability plot, if all the data points fall near the line, an assumption of normality is reasonable. Otherwise, the points will curve away from the line, and an assumption of normality is not justified. For example, the following generates a data sample of 100 random numbers from an exponential distribution with  $\mu = 10$ , and creates a normal probability plot of the data.

```
x = exprnd(10,100,1);  
normplot(x)
```



The plot is strong evidence that the underlying distribution is not normal.

### Quantile-Quantile Plots

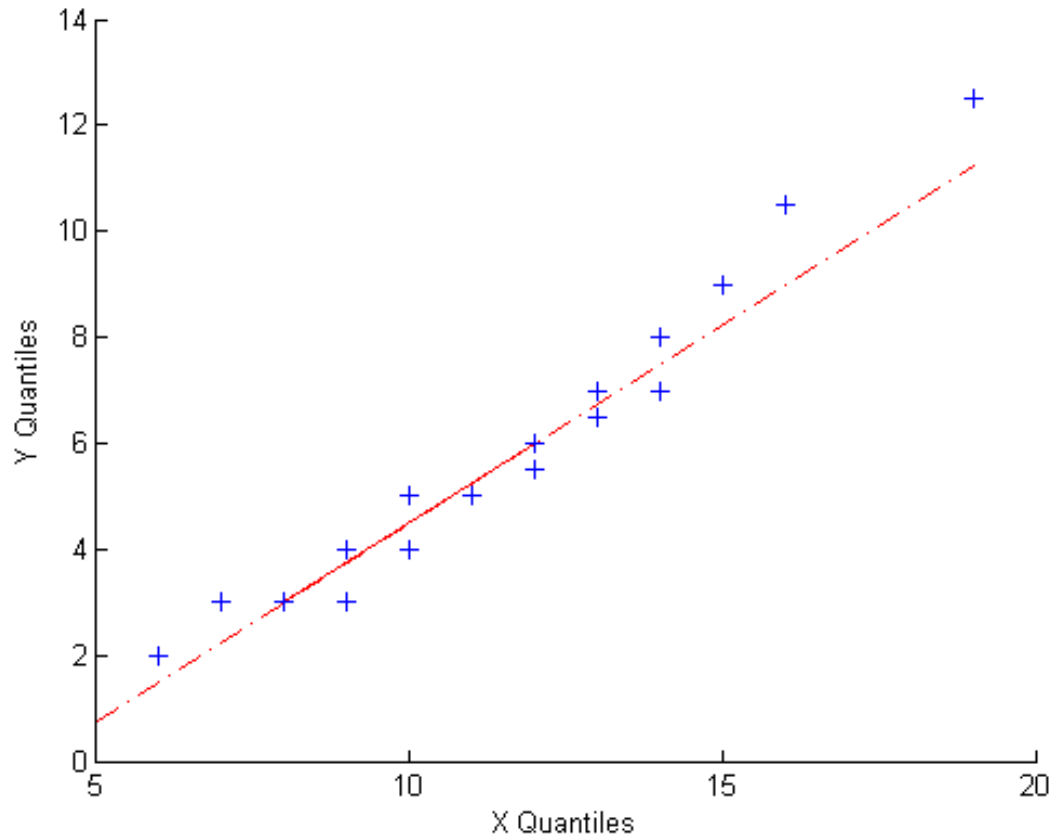
Quantile-quantile plots are used to determine whether two samples come from the same distribution family. They are scatter plots of quantiles computed from each sample, with a line drawn between the first and third quartiles. If the data falls near the line, it is reasonable to assume that the two samples come from the same distribution. The method is robust with respect to changes in the location and scale of either distribution.

To create a quantile-quantile plot, use the `qqplot` function.

The following example generates two data samples containing random numbers from Poisson distributions with different parameter values, and creates a quantile-quantile plot. The data in `x` is from a Poisson distribution with `lambda = 10`, and the data in `y` is from a Poisson distribution with `lambda = 5`.

```
x = poissrnd(10,50,1);  
y = poissrnd(5,100,1);  
qqplot(x,y);
```



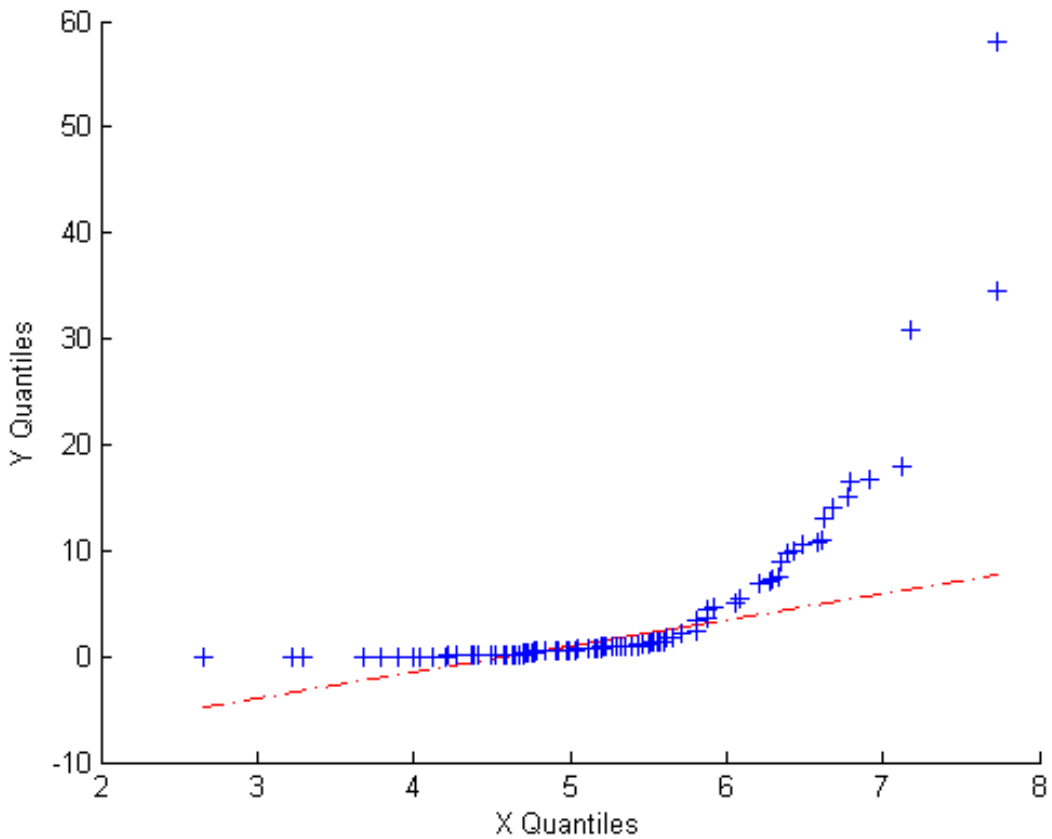


Even though the parameters and sample sizes are different, the approximate linear relationship suggests that the two samples may come from the same distribution family. As with normal probability plots, hypothesis tests can provide additional justification for such an assumption. For statistical procedures that depend on the two samples coming from the same distribution, however, a linear quantile-quantile plot is often sufficient.

The following example shows what happens when the underlying distributions are not the same. Here, `x` contains 100 random numbers generated from a

normal distribution with  $\mu = 5$  and  $\sigma = 1$ , while  $y$  contains 100 random numbers generated from a Weibull distribution with  $A = 2$  and  $B = 0.5$ .

```
x = normrnd(5,1,100,1);  
y = wblrnd(2,0.5,100,1);  
qqplot(x,y);
```



These samples clearly are not from the same distribution family.

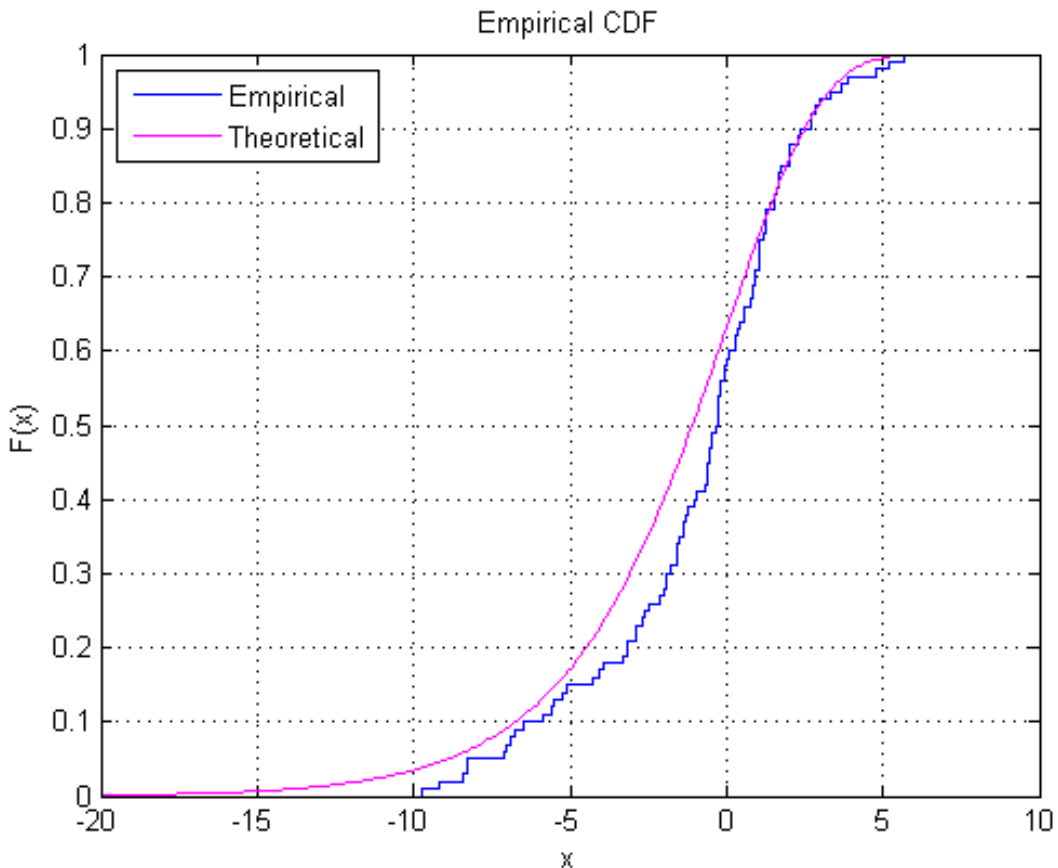
## Cumulative Distribution Plots

An empirical cumulative distribution function (cdf) plot shows the proportion of data less than each  $x$  value, as a function of  $x$ . The scale on the  $y$ -axis is linear; in particular, it is not scaled to any particular distribution. Empirical cdf plots are used to compare data cdfs to cdfs for particular distributions.

To create an empirical cdf plot, use the `cdfplot` function (or `ecdf` and `stairs`).

The following example compares the empirical cdf for a sample from an extreme value distribution with a plot of the cdf for the sampling distribution. In practice, the sampling distribution would be unknown, and would be chosen to match the empirical cdf.

```
y = evrnd(0,3,100,1);
cdfplot(y)
hold on
x = -20:0.1:10;
f = evcdf(x,0,3);
plot(x,f,'m')
legend('Empirical','Theoretical','Location','NW')
```



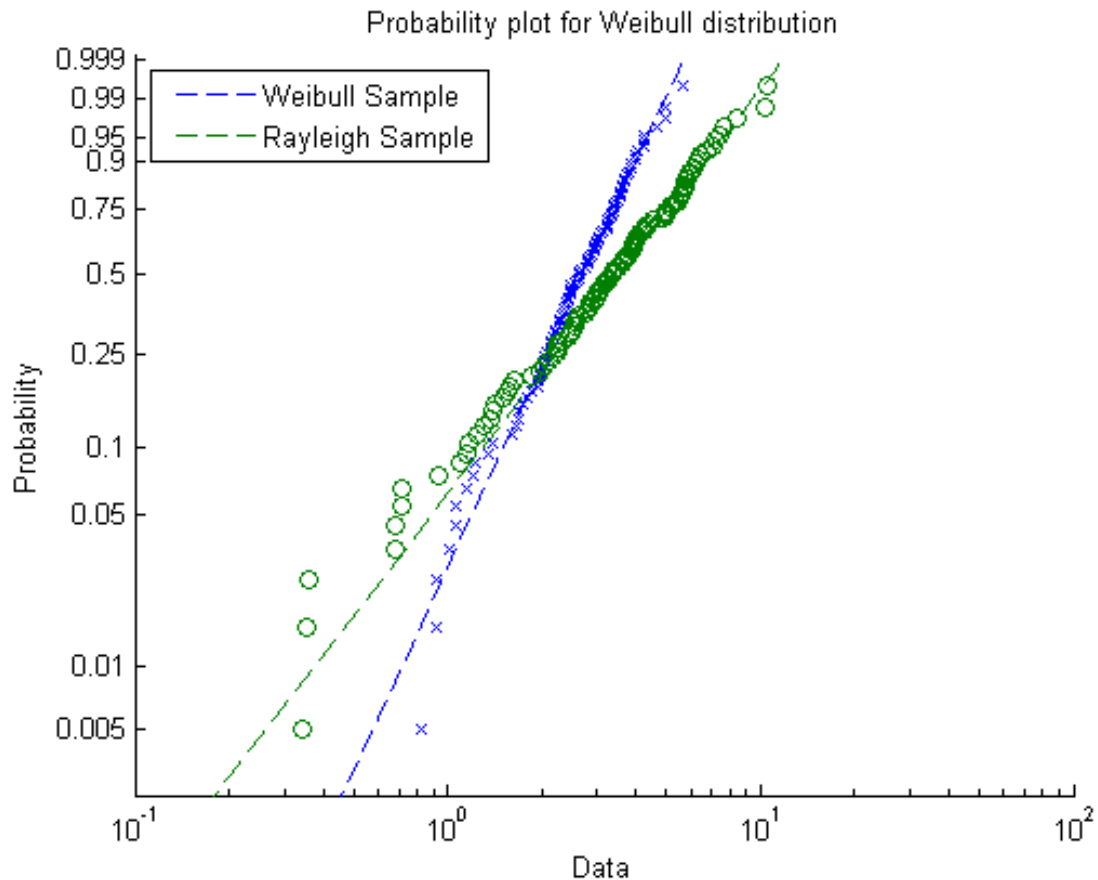
### Other Probability Plots

A probability plot, like the normal probability plot, is just an empirical cdf plot scaled to a particular distribution. The  $y$ -axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks is the distance between quantiles of the distribution. In the plot, a line is drawn between the first and third quartiles in the data. If the data falls near the line, it is reasonable to choose the distribution as a model for the data.

To create probability plots for different distributions, use the `probplot` function.

The following example assesses two samples, one from a Weibull distribution with  $A = 3$  and  $B = 3$ , and one from a Rayleigh distribution with  $B = 3$ , to see if either distribution may have come from a Weibull population.

```
x1 = wblrnd(3,3,100,1);  
x2 = raylrnd(3,100,1);  
probplot('weibull',[x1 x2])  
legend('Weibull Sample','Rayleigh Sample','Location','NW')
```



The plot gives justification for modeling the first sample with a Weibull distribution; much less so for the second sample.

A distribution analysis typically combines probability plots with hypothesis tests for a particular distribution.

# Probability Distributions

---

- “Working with Probability Distributions” on page 5-3
- “Maximum Likelihood Estimation” on page 5-5
- “Negative Loglikelihood Functions” on page 5-8
- “Random Number Generation” on page 5-11
- “Nonparametric and Empirical Probability Distributions” on page 5-14
- “Fit Kernel Distribution Object to Data” on page 5-16
- “Fit Kernel Distribution Using `ksdensity`” on page 5-21
- “Fit Kernel Distributions to Grouped Data Using `ksdensity`” on page 5-23
- “Create and Plot Empirical Cumulative Distribution Functions” on page 5-26
- “Fit a Nonparametric Distribution with Pareto Tails” on page 5-27
- “Generate Random Numbers Using the Triangular Distribution” on page 5-32
- “Explore the Probability Distribution Function UI” on page 5-37
- “Model Data Using the Distribution Fitting App” on page 5-40
- “Custom Distributions Using the Distribution Fitting App” on page 5-77
- “Explore the Random Number Generation UI” on page 5-80
- “Compare Multiple Distribution Fits” on page 5-83
- “Fit Probability Distribution Objects to Grouped Data” on page 5-89
- “Multinomial Probability Distribution Objects” on page 5-94
- “Multinomial Probability Distribution Functions” on page 5-98

- “Generate Random Numbers Using Uniform Distribution Inversion” on page 5-101
- “Use  $t$  Location-Scale to Represent Cauchy Distribution ” on page 5-104
- “Use Student’s  $t$  to Generate Standard Cauchy Random Numbers” on page 5-108
- “Probability Distributions Used for Multivariate Modeling” on page 5-110



## Working with Probability Distributions

In this section...
“Types of Probability Distributions” on page 5-3
“How to Work with Probability Distributions” on page 5-4

### Types of Probability Distributions

Probability distributions are theoretical distributions based on assumptions about a source population. The distributions assign probability to the event that a random variable has a specific, discrete value, or falls within a specified range of continuous values. The two main types of distributions are parametric and nonparametric.

Parametric distributions are probability distributions that you can describe using a finite set of parameters. You can choose a distribution model for your data based on a parametric family of probability distributions, and then adjust the parameters to fit the data. You can then perform further analyses by computing summary statistics, evaluating the probability density function (pdf) and cumulative distribution function (cdf), and assessing the fit of the distribution to your data.

Nonparametric and empirical distributions provide estimates of probability density functions (pdf) or cumulative distribution functions (cdf) based on sample data. You can use this approach when the data cannot be described accurately by the supported parametric distributions. Statistics Toolbox supports the following nonparametric and empirical techniques:

- Kernel distribution
- Piecewise linear distribution
- Empirical cumulative distribution function
- Piecewise distribution with Pareto tails
- Triangular distribution

### **How to Work with Probability Distributions**

Statistics Toolbox provides several ways to work with parametric and nonparametric probability distributions.

Apps and user interfaces provide an interactive approach to working with parametric and nonparametric probability distributions.

- Use the Distribution Fitting app to fit a distribution to your data and export a probability distribution object to your workspace for further analysis.
- Use the Probability Distribution Function user interface to visually explore probability distributions.
- Use the Random Number Generation user interface to generate random data from a specified distribution and export the results to your workspace.

Probability distribution objects provide a convenient way to explore and fit your data to a parametric or nonparametric distribution, save the results to a single entity, and generate random data from the resulting parameters.

Command-line functions let you further explore parametric and nonparametric distributions, fit relevant models to your data, and generate random data. You can use command-line functions to work with distributions that are not currently supported as probability distribution objects.

## Maximum Likelihood Estimation

The Statistics Toolbox function `mle` is a convenient front end to the individual distribution fitting functions, and more. The function computes maximum likelihood estimates (MLEs) for distributions beyond those for which Statistics Toolbox software provides specific pdf functions.

For some pdfs, MLEs can be given in closed form and computed directly. For other pdfs, a search for the maximum likelihood must be employed. The search can be controlled with an `options` input argument, created using the `statset` function. For efficient searches, it is important to choose a reasonable distribution model and set appropriate convergence tolerances.

MLEs can be heavily biased, especially for small samples. As sample size increases, however, MLEs become unbiased minimum variance estimators with approximate normal distributions. This is used to compute confidence bounds for the estimates.

For example, consider the following distribution of means from repeated random samples of an exponential distribution:

```
mu = 1; % Population parameter
n = 1e3; % Sample size
ns = 1e4; % Number of samples
samples = exprnd(mu,n,ns); % Population samples
means = mean(samples); % Sample means
```

The Central Limit Theorem says that the means will be approximately normally distributed, regardless of the distribution of the data in the samples. The `normfit` function can be used to find the normal distribution that best fits the means:

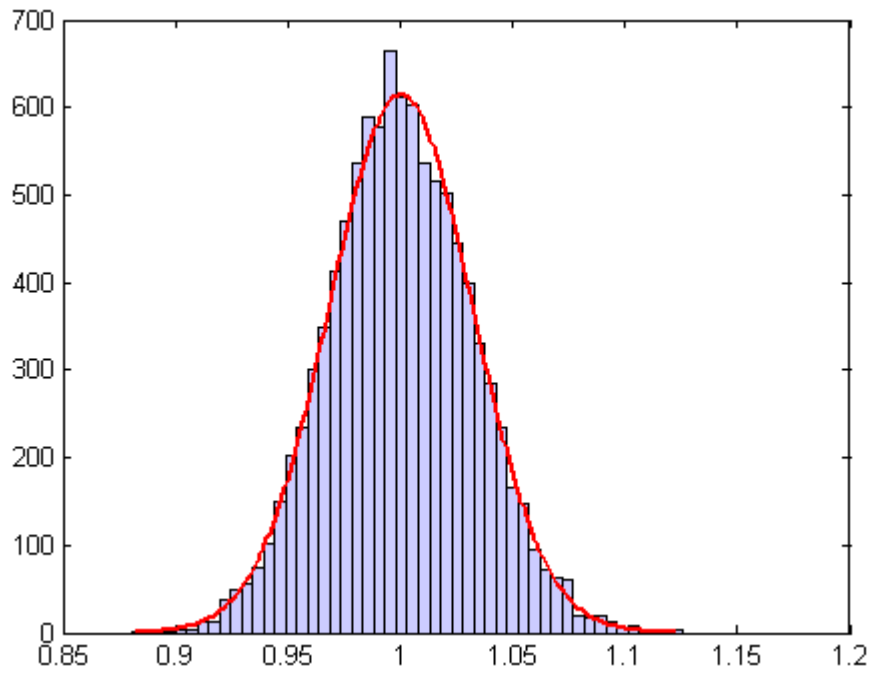
```
[muhat,sigmahat,muci,sigmaci] = normfit(means)
muhat =
    1.0003
sigmahat =
    0.0319
muci =
    0.9997
    1.0010
```

```
sigmaci =  
    0.0314  
    0.0323
```

The function returns MLEs for the mean and standard deviation and their 95% confidence intervals.

To visualize the distribution of sample means together with the fitted normal distribution, you must scale the fitted pdf, with area = 1, to the area of the histogram being used to display the means:

```
numbins = 50;  
hist(means,numbins)  
set(get(gca,'Children'),'FaceColor',[.8 .8 1])  
hold on  
[bincounts,binpositions] = hist(means,numbins);  
binwidth = binpositions(2) - binpositions(1);  
histarea = binwidth*sum(bincounts);  
x = binpositions(1):0.001:binpositions(end);  
y = normpdf(x,muhat,sigmahat);  
plot(x,histarea*y,'r','LineWidth',2)
```



## Negative Loglikelihood Functions

Negative loglikelihood functions for supported Statistics Toolbox distributions all end with `like`, as in `explike`. Each function represents a parametric family of distributions. Input arguments are lists of parameter values specifying a particular member of the distribution family followed by an array of data. Functions return the negative log-likelihood of the parameters, given the data.

Negative log-likelihood functions are used as objective functions in search algorithms such as the one implemented by the MATLAB function `fminsearch`. Additional search algorithms are implemented by Optimization Toolbox™ functions and Global Optimization Toolbox functions.

When used to compute maximum likelihood estimates (MLEs), negative log-likelihood functions allow you to choose a search algorithm and exercise low-level control over algorithm execution. By contrast, the functions discussed in “Maximum Likelihood Estimation” on page 5-5 use preset algorithms with options limited to those set by the `statset` function.

Likelihoods are conditional probability densities. A parametric family of distributions is specified by its pdf  $f(x,a)$ , where  $x$  and  $a$  represent the variables and parameters, respectively. When  $a$  is fixed, the pdf is used to compute the density at  $x$ ,  $f(x|a)$ . When  $x$  is fixed, the pdf is used to compute the *likelihood* of the parameters  $a$ ,  $f(a|x)$ . The joint likelihood of the parameters over an independent random sample  $X$  is

$$L(a) = \prod_{x \in X} f(a|x)$$

Given  $X$ , MLEs maximize  $L(a)$  over all possible  $a$ .

In numerical algorithms, the log-likelihood function,  $\log(L(a))$ , is (equivalently) optimized. The logarithm transforms the product of potentially small likelihoods into a sum of logs, which is easier to distinguish from 0 in computation. For convenience, Statistics Toolbox negative log-likelihood functions return the *negative* of this sum, since the optimization algorithms to which the values are passed typically search for minima rather than maxima.

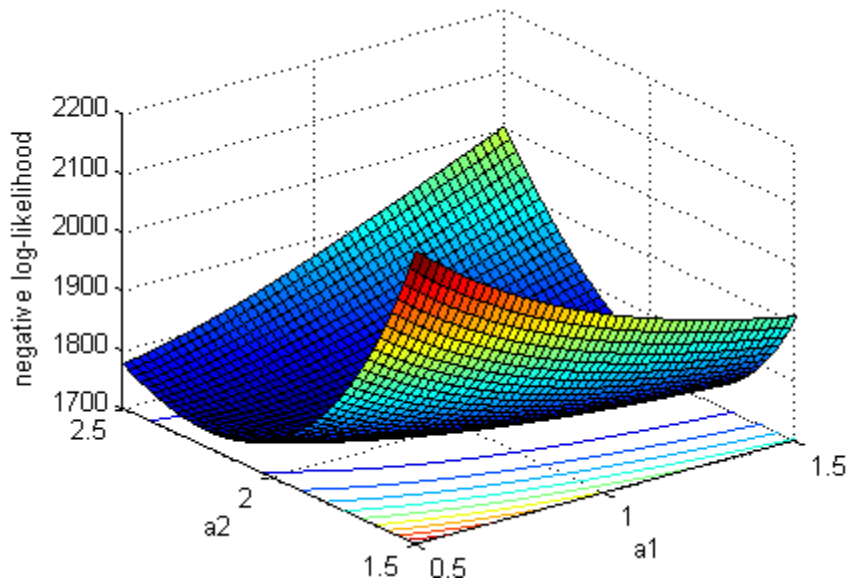
For example, use `gamrnd` to generate a random sample from a specific gamma distribution:

```
a = [1,2];
X = gamrnd(a(1),a(2),1e3,1);
```

Given `X`, the `gamlike` function can be used to visualize the likelihood surface in the neighborhood of `a`:

```
mesh = 50;
delta = 0.5;
a1 = linspace(a(1)-delta,a(1)+delta,mesh);
a2 = linspace(a(2)-delta,a(2)+delta,mesh);
logL = zeros(mesh); % Preallocate memory
for i = 1:mesh
    for j = 1:mesh
        logL(i,j) = gamlike([a1(i),a2(j)],X);
    end
end
end
```

```
[A1,A2] = meshgrid(a1,a2);
surfc(A1,A2,logL)
```



The MATLAB function `fminsearch` is used to search for the minimum of the likelihood surface:

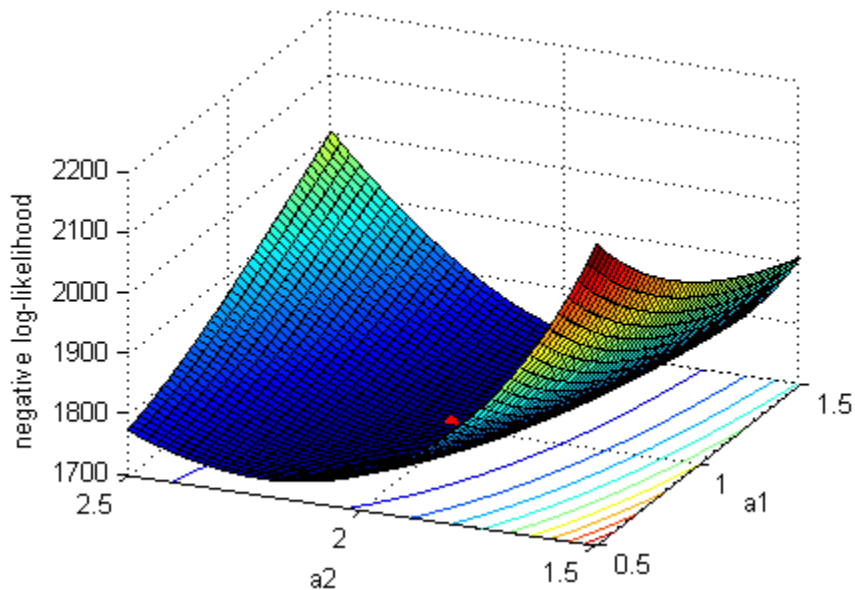
```
LL = @(u)gamlike([u(1),u(2)],X); % Likelihood given X
MLES = fminsearch(LL,[1,2])
MLES =
    1.0231    1.9729
```

These can be compared to the MLEs returned by the `gamfit` function, which uses a combination search and solve algorithm:

```
ahat = gamfit(X)
ahat =
    1.0231    1.9728
```

The MLEs can be added to the surface plot (rotated to show the minimum):

```
hold on
plot3(MLES(1),MLES(2),LL(MLES),...
      'ro','MarkerSize',5,...
      'MarkerFaceColor','r')
```





## Random Number Generation

The Statistics Toolbox supports the generation of random numbers from various distributions. Each RNG represents a parametric family of distributions. RNGs return random numbers from the specified distribution in an array of the specified dimensions.

Other random number generation functions which do not support specific distributions include:

- `cvpartition`
- `datasample`
- `hmmgenerate`
- `lhsdesign`
- `lhsnorm`
- `mhsample`
- `random`
- `randsample`
- `slicesample`

RNGs in Statistics Toolbox software depend on MATLAB's default random number stream via the `rand` and `randn` functions, each RNG uses one of the techniques discussed in “Common Generation Methods” on page 6-5 to generate random numbers from a given distribution.

By controlling the default random number stream and its state, you can control how the RNGs in Statistics Toolbox software generate random values. For example, to reproduce the same sequence of values from an RNG, you can save and restore the default stream's state, or reset the default stream. For details on managing the default random number stream, see “Managing the Global Stream”.

MATLAB initializes the default random number stream to the same state each time it starts up. Thus, RNGs in Statistics Toolbox software will generate the same sequence of values for each MATLAB session unless you

modify that state at startup. One simple way to do that is to add commands to `startup.m` such as

```
rng('shuffle');
```

that initialize MATLAB's default random number stream to a different state for each session.

The following table lists the dependencies of Statistics Toolbox RNGs on the MATLAB base RNGs `rand`, `randi`, and/or `randn`.

<b>RNG</b>	<b>MATLAB Base RNG</b>
<code>betarnd</code>	<code>rand</code> , <code>randn</code>
<code>binornd</code>	<code>rand</code>
<code>chi2rnd</code>	<code>rand</code> , <code>randn</code>
<code>evrnd</code>	<code>rand</code>
<code>exprnd</code>	<code>rand</code>
<code>datasample</code>	<code>rand</code> , <code>randi</code> , <code>randperm</code>
<code>frnd</code>	<code>rand</code> , <code>randn</code>
<code>gamrnd</code>	<code>rand</code> , <code>randn</code>
<code>geornd</code>	<code>rand</code>
<code>gevrnd</code>	<code>rand</code>
<code>gprnd</code>	<code>rand</code>
<code>hygernd</code>	<code>rand</code>
<code>iwishrnd</code>	<code>rand</code> , <code>randn</code>
<code>johnsrnd</code>	<code>randn</code>
<code>lhsdesign</code>	<code>rand</code>
<code>lhsnorm</code>	<code>rand</code>
<code>lognrnd</code>	<code>randn</code>

<b>RNG</b>	<b>MATLAB Base RNG</b>
mhsample	rand or randn, depending on the RNG given for the proposal distribution
mvnrnd	randn
mvtrnd	rand, randn
nbinrnd	rand, randn
ncfrnd	rand, randn
nctrnd	rand, randn
ncx2rnd	randn
normrnd	randn
pearsrnd	rand or randn, depending on the distribution type
poissrnd	rand, randn
random	rand or randn, depending on the specified distribution
randsample	rand
raylrnd	randn
slicesample	rand
trnd	rand, randn
unidrnd	rand
unifrnd	rand
wblrnd	rand
wishrnd	rand, randn

## Nonparametric and Empirical Probability Distributions

In this section...
“Overview” on page 5-14
“Kernel Distribution” on page 5-14
“Piecewise Linear Distribution” on page 5-14
“Empirical Cumulative Distribution Function” on page 5-15
“Pareto Tails” on page 5-15
“Triangular Distribution” on page 5-15

### Overview

In some situations, you cannot accurately describe a data sample using a parametric distribution. Instead, the probability density function (pdf) or cumulative distribution function (cdf) must be estimated from the data. Several options exist in Statistics Toolbox for estimating the pdf or cdf from sample data.

### Kernel Distribution

“Kernel Distribution” on page B-67 produces a nonparametric probability density estimate that adapts itself to the data, rather than selecting a density with a particular parametric form and estimating the parameters. This distribution is defined by a kernel density estimator, a smoothing function that determines the shape of the curve used to generate the pdf, and a bandwidth value that controls the smoothness of the resulting density curve.

### Piecewise Linear Distribution

A piecewise distribution is a probability distribution formed by estimating the pdf or cdf in pieces from the sample data. “Piecewise Linear Distribution” on page B-115 estimates an overall cdf by computing the cdf value at each individual point in the sample data, and linearly connects these values to form a continuous curve.

## **Empirical Cumulative Distribution Function**

An empirical cumulative distribution function (ecdf) estimates the cdf of a random variable by assigning equal probability to each observation in a sample. It creates an exact match between the empirical cdf and the distribution of the sample data. However, the empirical cdf is not a smooth distribution, especially in the tails where data might be sparse. In this case, you can use Pareto tails to smooth the distribution in the tails.

For more information, see `ecdf`, `paretotails`, and `coxphfit`.

## **Pareto Tails**

Pareto tails use a piecewise approach to improve the fit of a nonparametric cdf or pdf by smoothing the tails of the distribution. You can fit a kernel distribution, piecewise linear distribution, or empirical cdf to the middle data values, then fit generalized Pareto distributions (GPDs) to the tails. This technique is especially useful when the sample data is sparse in the tails.

For more information, see `paretotails`.

## **Triangular Distribution**

“Triangular Distribution” on page B-127 provides a simplistic representation of the probability distribution when limited sample data is available. It uses the minimum, mode, and maximum value of the sample data as parameters, and linearly connects these points to estimate the pdf.

## Fit Kernel Distribution Object to Data

This example shows how to fit a kernel probability distribution object to sample data.

### **Step 1. Load sample data.**

Load the sample data.

```
load carsmall;
```

This data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (**Origin**), model year (**Year**), and other vehicle characteristics.

### **Step 2. Fit a kernel distribution object.**

Use `fitdist` to fit a kernel probability distribution object to the miles per gallon (MPG) data for all makes of cars.

```
pd = fitdist(MPG,'Kernel')
```

```
pd =
```

```
KernelDistribution
```

```
Kernel = normal  
Bandwidth = 4.11428  
Support = unbounded
```

This creates a `prob.KernelDistribution` object. By default, `fitdist` uses a normal kernel smoothing function and chooses an optimal bandwidth for estimating normal densities, unless you specify otherwise. You can access information about the fit and perform further calculations using the related object functions.

### **Step 3. Compute descriptive statistics.**

Compute the mean, median, and standard deviation of the fitted kernel distribution.

```
m = mean(pd)
med = median(pd)
s = std(pd)
```

```
m =
    23.7181
```

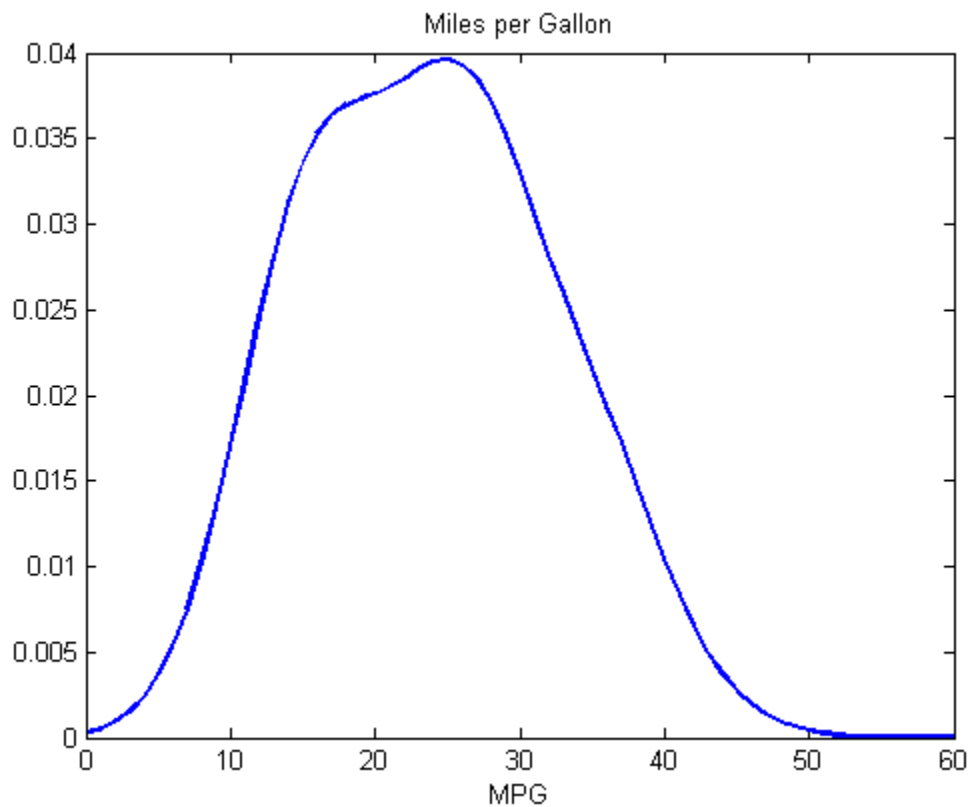
```
med =
    23.4841
```

```
s =
    8.9896
```

#### **Step 4. Compute and plot the pdf.**

Compute and plot the pdf of the fitted kernel distribution.

```
figure;
x = 0:1:60;
y = pdf(pd,x);
plot(x,y,'LineWidth',2);
title('Miles per Gallon');
xlabel('MPG');
```



The plot shows the pdf of the kernel distribution fit to the MPG data across all makes of cars. The distribution is smooth and fairly symmetrical, although it is slightly skewed with a heavier right tail.

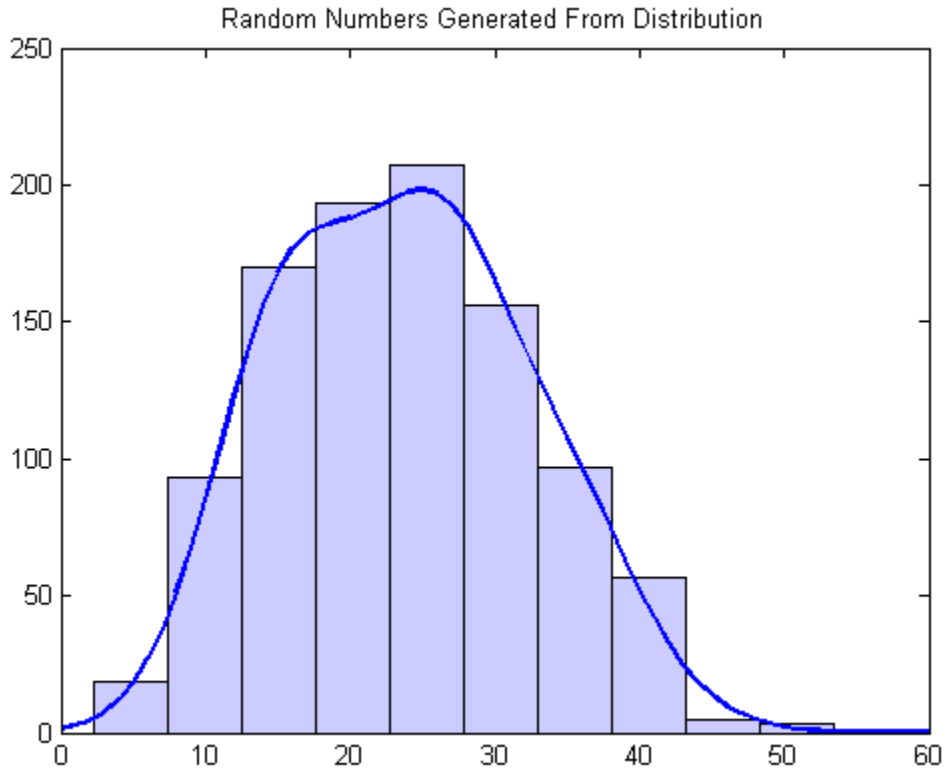
### **Step 5. Generate random numbers.**

Generate a vector of random numbers from the fitted kernel distribution.

```
rng('default') % For reproducibility
r = random(pd,1000,1);
figure;
hist(r);
set(get(gca,'Children'),'FaceColor',[.8 .8 1]);
```



```
hold on;  
y = y*5000; % Scale pdf to overlay on histogram  
plot(x,y,'LineWidth',2);  
title('Random Numbers Generated From Distribution');  
hold off;
```



The histogram has a similar shape to the pdf plot because the random numbers generate from the nonparametric kernel distribution fit to the sample data.

**See Also** `fitdist`

### **Related Examples**

- “Fit Kernel Distribution Using `ksdensity`” on page 5-21

### **Concepts**

- “Kernel Distribution” on page B-67

## Fit Kernel Distribution Using `ksdensity`

This example shows how to generate a kernel probability density estimate from sample data using the `ksdensity` function.

### **Step 1. Load sample data.**

Load the sample data.

```
load carsmall;
```

This data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (`Origin`), model year (`Year`), and other vehicle characteristics.

### **Step 2. Generate a kernel probability density estimate.**

Use `ksdensity` to generate a kernel probability density estimate for the miles per gallon (MPG) data.

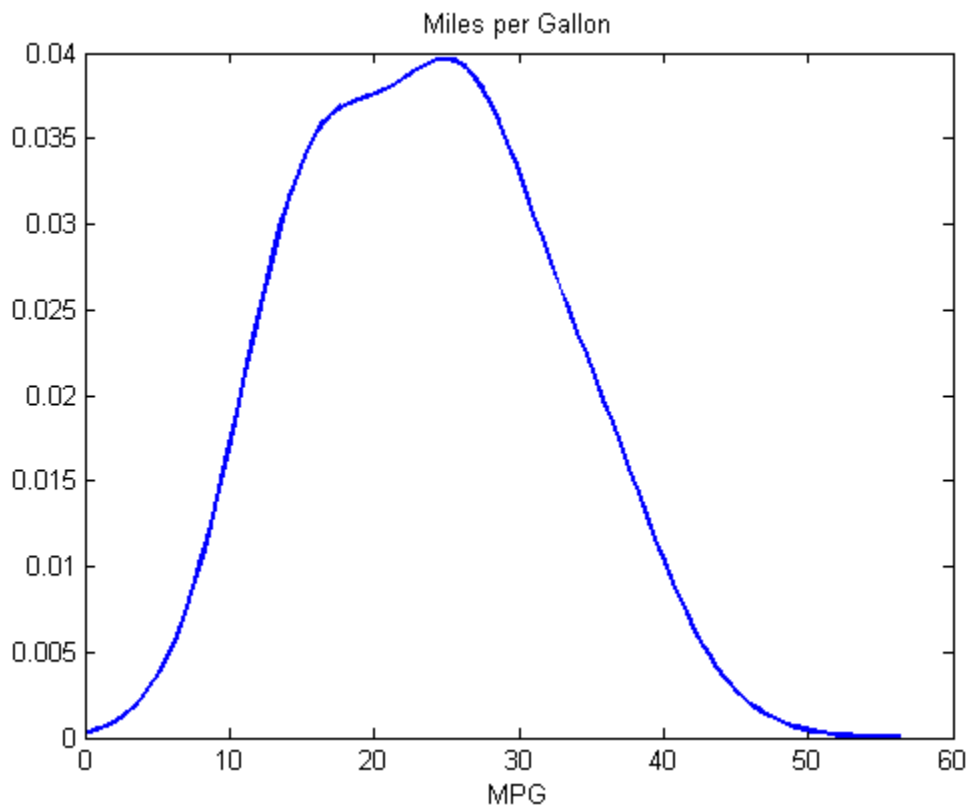
```
[f,xi] = ksdensity(MPG);
```

By default, `ksdensity` uses a normal kernel smoothing function and chooses an optimal bandwidth for estimating normal densities, unless you specify otherwise.

### **Step 3. Plot the kernel probability density estimate.**

Plot the kernel probability density estimate to visualize the |MPG| distribution.

```
figure;  
plot(xi,f,'LineWidth',2);  
title('Miles per Gallon');  
xlabel('MPG');
```



The plot shows the pdf of the kernel distribution fit to the MPG data across all makes of cars. The distribution is smooth and fairly symmetrical, although it is slightly skewed with a heavier right tail.

**See Also** `ksdensity`

### Related Examples

- “Fit Kernel Distribution Object to Data” on page 5-16

### Concepts

- “Kernel Distribution” on page B-67

## Fit Kernel Distributions to Grouped Data Using ksdensity

This example shows how to fit kernel distributions to grouped sample data using the ksdensity function.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

The data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (Origin), model year (Model\_Year), and other vehicle characteristics.

### Step 2. Group sample data by origin.

Group the MPG data by origin (Origin) for cars made in the USA, Japan, and Germany.

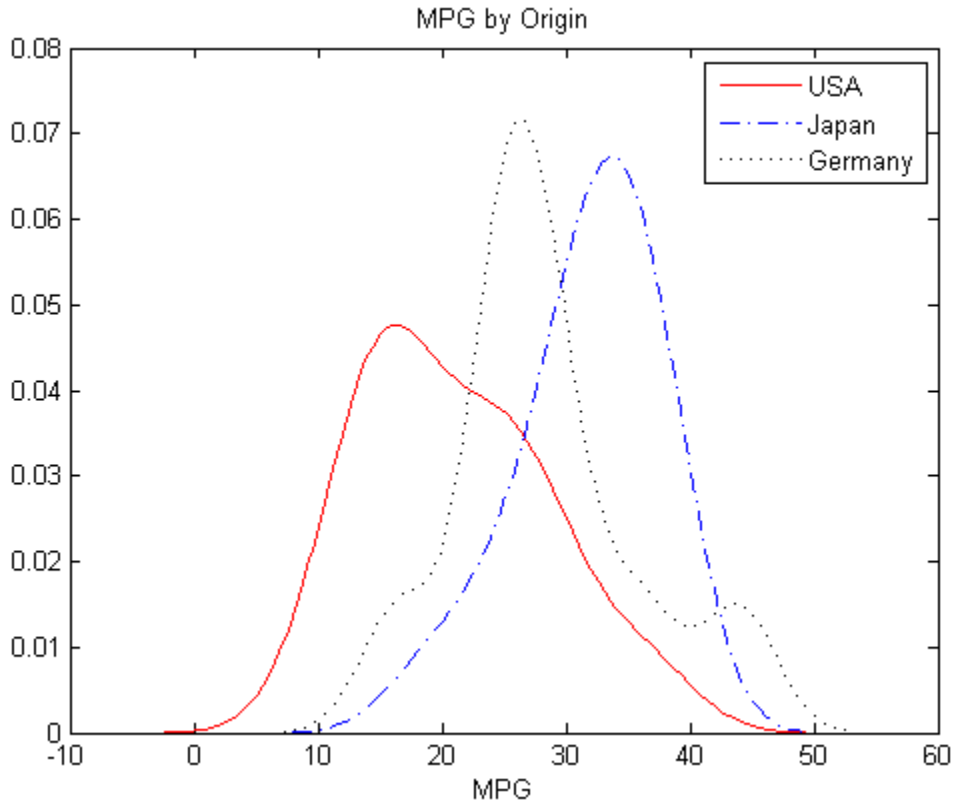
```
Origin = nominal(Origin);  
MPG_USA = MPG(Origin=='USA');  
MPG_Japan = MPG(Origin=='Japan');  
MPG_Germany = MPG(Origin=='Germany');
```

### Step 3. Compute and plot the pdf.

Compute and plot the pdf for each group.

```
figure;  
  
[fi,xi] = ksdensity(MPG_USA);  
plot(xi,fi,'r-');  
hold on;  
  
[fj,xj] = ksdensity(MPG_Japan);  
plot(xj,fj,'b-.');  
  
[fk,xk] = ksdensity(MPG_Germany);  
plot(xk,fk,'k:');
```

```
legend('USA', 'Japan', 'Germany')  
title('MPG by Origin');  
xlabel('MPG');  
hold off;
```



The plot shows how miles per gallon (MPG) performance differs by country of origin (Origin). Using this data, the USA has the widest distribution, and its peak is at the lowest MPG value of the three origins. Japan has the most regular distribution with a slightly heavier left tail, and its peak is at the highest MPG value of the three origins. The peak for Germany is between the USA and Japan, and the second bump near 44 miles per gallon suggests that there might be multiple modes in the data.

**See Also** `makedist`

**Related  
Examples**

- “Fit Kernel Distribution Using `ksdensity`” on page 5-21
- “Fit Probability Distribution Objects to Grouped Data” on page 5-89

**Concepts**

- “Kernel Distribution” on page B-67
- “Grouping Variables” on page 2-51

## Create and Plot Empirical Cumulative Distribution Functions

This example shows how to create and plot an empirical cumulative distribution function based on sample data.

### **Step 1. Load sample data.**

Load the sample data.

### **Step 2. Compute the empirical cumulative distribution.**

The empirical cdf assigns the probability  $1/n$  to each of  $n$  observations in a data sample. It returns the values of a function  $F$  such that  $F(x)$  represents the proportion of observations in a sample less than or equal to  $x$ . The empirical distributions computed by `ecdf` assign equal probability to each observation in a sample, providing an exact match of the sample distribution. However, the distributions are not smooth, especially in the tails where data may be sparse. In this situation, you can use Pareto tails to smooth the cdf in the tails.

### **Step 3. Plot the ecdf.**

The graph of an empirical cdf has a stair-step appearance. If a sample comes from a distribution in a parametric family (such as a normal distribution), its empirical cdf is likely to resemble the parametric distribution. If not, its empirical distribution still gives an estimate of the cdf for the distribution that generated the data.



## Fit a Nonparametric Distribution with Pareto Tails

This example shows how to fit a nonparametric probability distribution to sample data using Pareto tails to smooth the distribution in the tails.

### Step 1. Generate sample data.

Generate sample data that contains more outliers than expected from a standard normal distribution.

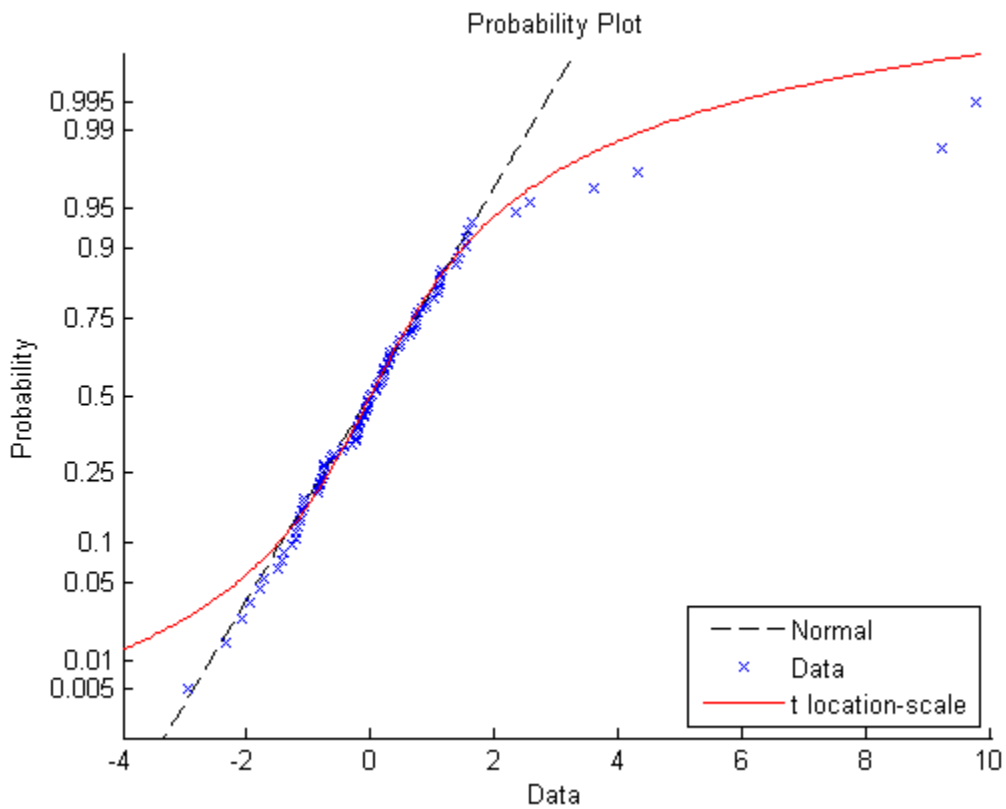
```
rng('default') % For reproducibility
left_tail = -exprnd(1,10,1);
right_tail = exprnd(5,10,1);
center = randn(80,1);
data = [left_tail;center;right_tail];
```

The data contains 80% values from a standard normal distribution, 10% from an exponential distribution with a mean of 5, and 10% from an exponential distribution with mean of -1. The data contains random numbers from an exponential distribution. Compared to a standard normal distribution, the exponential values are more likely to be outliers, especially in the upper tail.

### Step 2. Fit probability distributions to the data.

Fit a normal distribution and a  $t$  location-scale distribution to the data, and plot for a visual comparison.

```
figure;
probplot(data);
p = fitdist(data,'tlocation-scale');
h = probplot(gca,p);
set(h,'color','r','linestyle','-');
title('Probability Plot');
legend('Normal','Data','t location-scale','Location','SE');
```

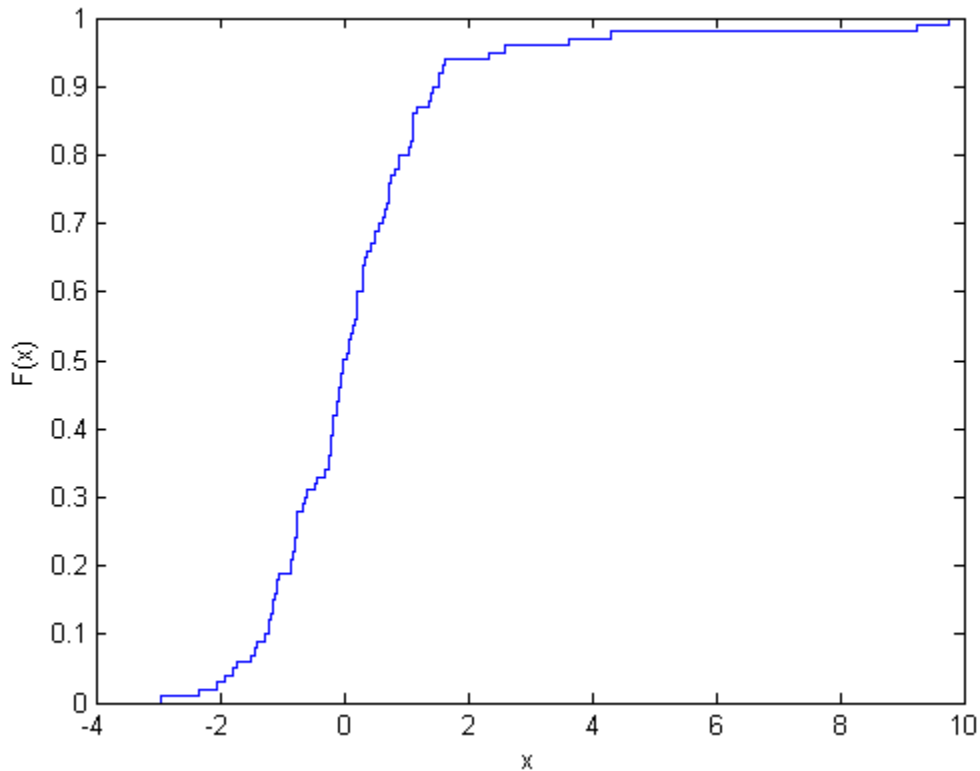


Both distributions appear to fit reasonably well in the center, but neither the normal distribution nor the  $t$  location-scale distribution fit the tails very well.

**Step 3. Generate an empirical distribution.**

To obtain a better fit, use `ecdf` to generate an empirical cdf based on the sample data.

```
figure;  
ecdf(data)
```



The empirical distribution provides a perfect fit, but the outliers make the tails very discrete. Random samples generated from this distribution using the inversion method might include, for example, values near 4.33 and 9.25, but no values in between.

**Step 4. Fit a distribution using Pareto tails.**

Use `paretotails` to generate an empirical cdf for the middle 80% of the data and fit generalized Pareto distributions to the lower and upper 10%.

```
pfit = paretotails(data,0.1,0.9)
```

```
pfit =
```

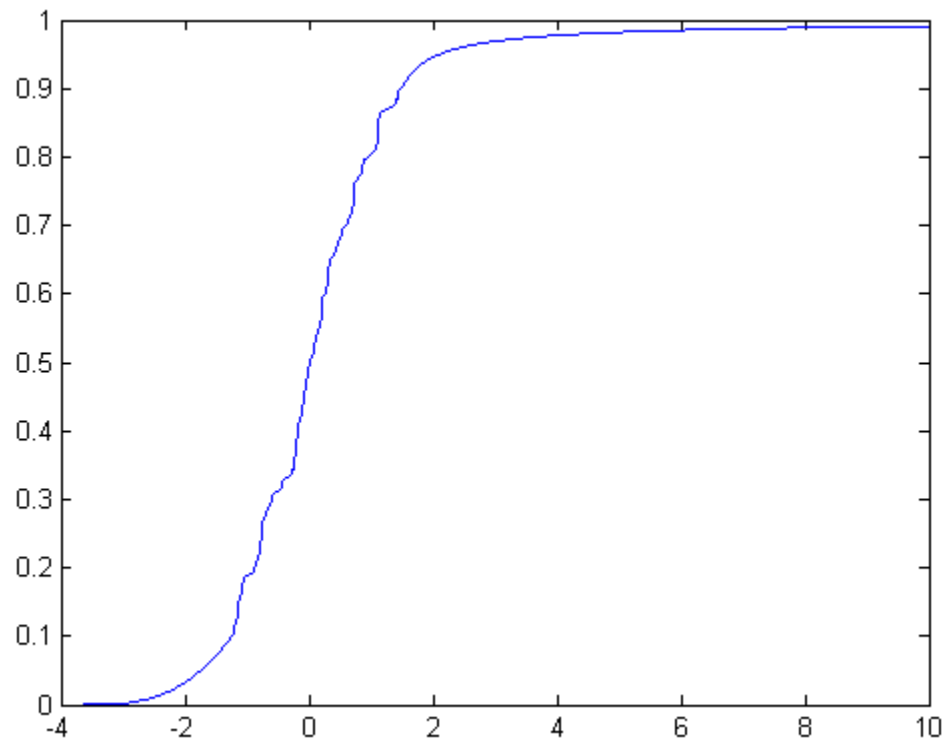
```
Piecewise distribution with 3 segments
  -Inf < x < -1.24623    (0 < p < 0.1): lower tail, GPD(-0.334156,0.798
  -1.24623 < x < 1.48551 (0.1 < p < 0.9): interpolated empirical cdf
    1.48551 < x < Inf    (0.9 < p < 1): upper tail, GPD(1.23681,0.58186
```

To obtain a better fit, `paretotails` fits a distribution by piecing together an ecdf or kernel distribution in the center of the sample, and smooth generalized Pareto distributions (GPDs) in the tails. The `paretotails` function creates a `paretotails` probability distribution object. You can access information about the fit and perform further calculations on the object using the methods of the `paretotails` class. For example, you can evaluate the cdf or generate random numbers from the distribution.

### **Step 5. Compute and plot the cdf.**

Compute and plot the cdf of the fitted `paretotails` distribution.

```
x = -4:0.01:10;
plot(x,cdf(pfit,x));
```



The `paretotails` cdf closely fits the data but is smoother in the tails than the ecdf generated in Step 3.

## Generate Random Numbers Using the Triangular Distribution

This example shows how to create a triangular probability distribution object based on sample data, and generate random numbers for use in a simulation.

### Step 1. Input sample data.

Input the data vector `time`, which contains the observed length of time (in seconds) that 10 different cars stopped at a highway tollbooth.

```
time = [6 14 8 7 16 8 23 6 7 15];
```

The data shows that, while most cars stopped for 6 to 16 seconds, one outlier stopped for 23 seconds.

### Step 2. Estimate distribution parameters.

Estimate the triangular distribution parameters from the sample data.

```
lower = min(time);  
peak = median(time);  
upper = max(time);
```

A triangular distribution provides a simplistic representation of the probability distribution when sample data is limited. Estimate the lower and upper boundaries of the distribution by finding the minimum and maximum values of the sample data. For the peak parameter, the median might provide a better estimate of the mode than the mean, since the data includes an outlier.

### Step 3. Create a probability distribution object.

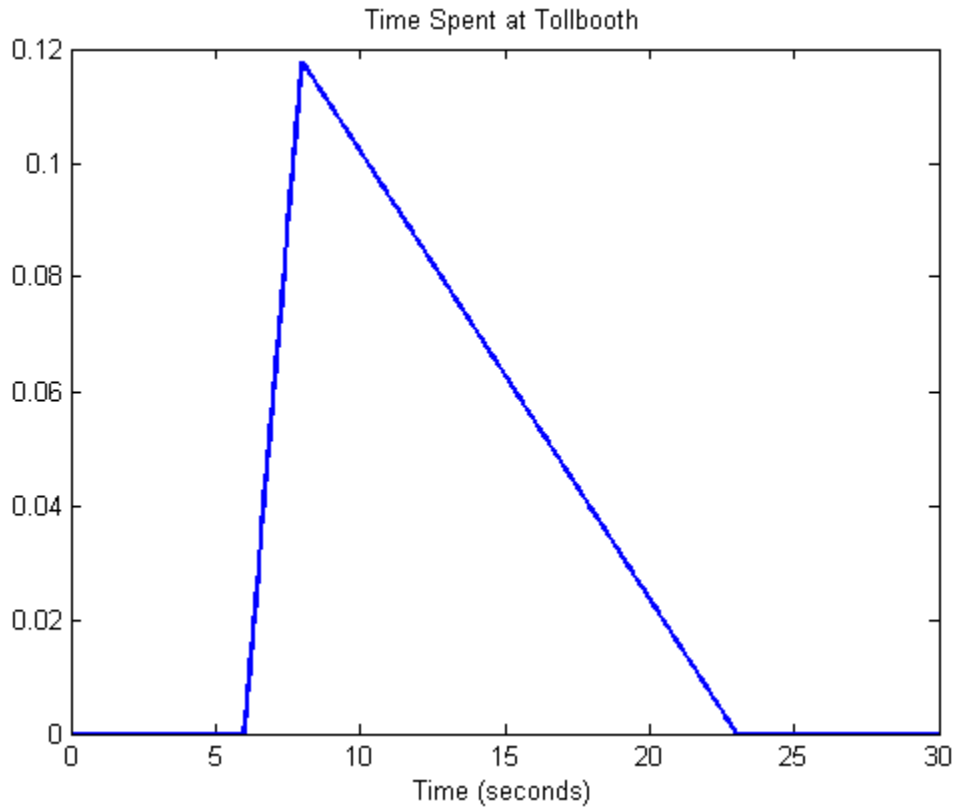
Create a triangular probability distribution object using the estimated parameter values.

```
pd = makedist('Triangular','a',lower,'b',peak,'c',upper)  
  
pd =  
  
TriangularDistribution
```

$A = 6, B = 8, C = 23$

Compute and plot the pdf of the triangular distribution.

```
figure;  
x = 0:.1:230;  
y = pdf(pd,x);  
plot(x,y);  
title('Time Spent at Tollbooth');  
xlabel('Time (seconds)');
```



The plot shows that this triangular distribution is skewed to the right. However, since the estimated peak value is the sample median, the distribution should be symmetrical about the peak. Because of its skew, this model might, for example, generate random numbers that seem unusually high when compared to the initial sample data.

### **Step 4. Generate random numbers.**

Generate random numbers from this distribution to simulate future traffic flow through the tollbooth.

```
rng('default'); % For reproducibility
r = random(pd,10,1)
```

```
r =
```

```
16.1265
18.0987
 8.0796
18.3001
13.3176
 7.8211
 9.4360
12.2508
19.7082
20.0078
```

The returned values in `r` are the time in seconds that the next 10 simulated cars spend at the tollbooth. These values seem high compared to the values in the original data vector `time` because the outlier skewed the distribution to the right. Using the second-highest value as the upper limit parameter might mitigate the effects of the outlier and generate a set of random numbers more similar to the initial sample data.

### **Step 5. Revise estimated parameters.**

Estimate the upper boundary of the distribution using the second largest value in the sample data.

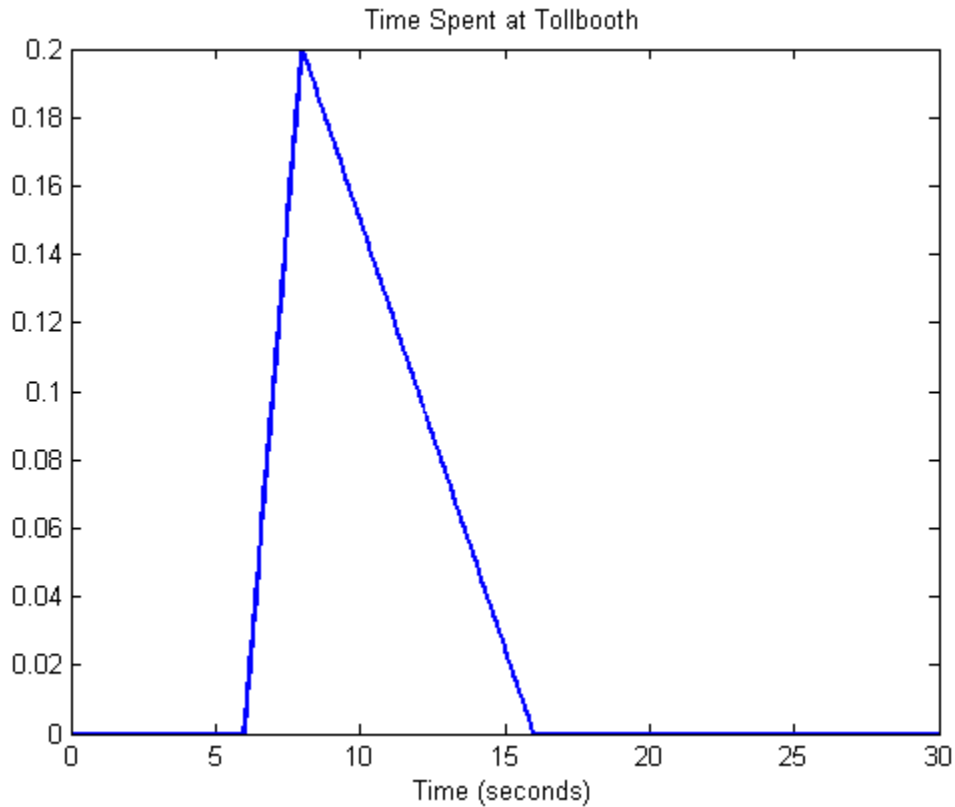
```
sort_time = sort(time,'descend');
secondLargest = sort_time(2);
```



**Step 6. Create a new distribution object and plot the pdf.**

Create a new triangular probability distribution object using the revised estimated parameters, and plot its pdf.

```
figure;  
pd2 = makedist('Triangular','a',lower,'b',peak,'c',secondLargest);  
y2 = pdf(pd2,x);  
plot(x,y2,'LineWidth',2);  
title('Time Spent at Tollbooth');  
xlabel('Time (seconds)');
```



The plot shows that this triangular distribution is still slightly skewed to the right. However, it is much more symmetrical about the peak than the distribution that used the maximum sample data value to estimate the upper limit.

**Step 7. Generate new random numbers.**

Generate new random numbers from the revised distribution.

```
rng('default'); % For reproducibility
r2 = random(pd2,10,1)
```

```
r2 =
    12.1501
    13.2547
     7.5937
    13.3675
    10.5768
     7.3967
     8.4026
     9.9792
    14.1562
    14.3240
```

These new values more closely resemble those in the original data vector `time`. They are also closer to the sample median than the random numbers generated by the distribution that used the outlier to estimate its upper limit. This example does not remove the outlier from the sample data when computing the median. Other options for parameter estimation include removing outliers from the sample data altogether, or using the mean or mode of the sample data as the peak value.

**See Also**

`pdf` | `random` | `makedist`

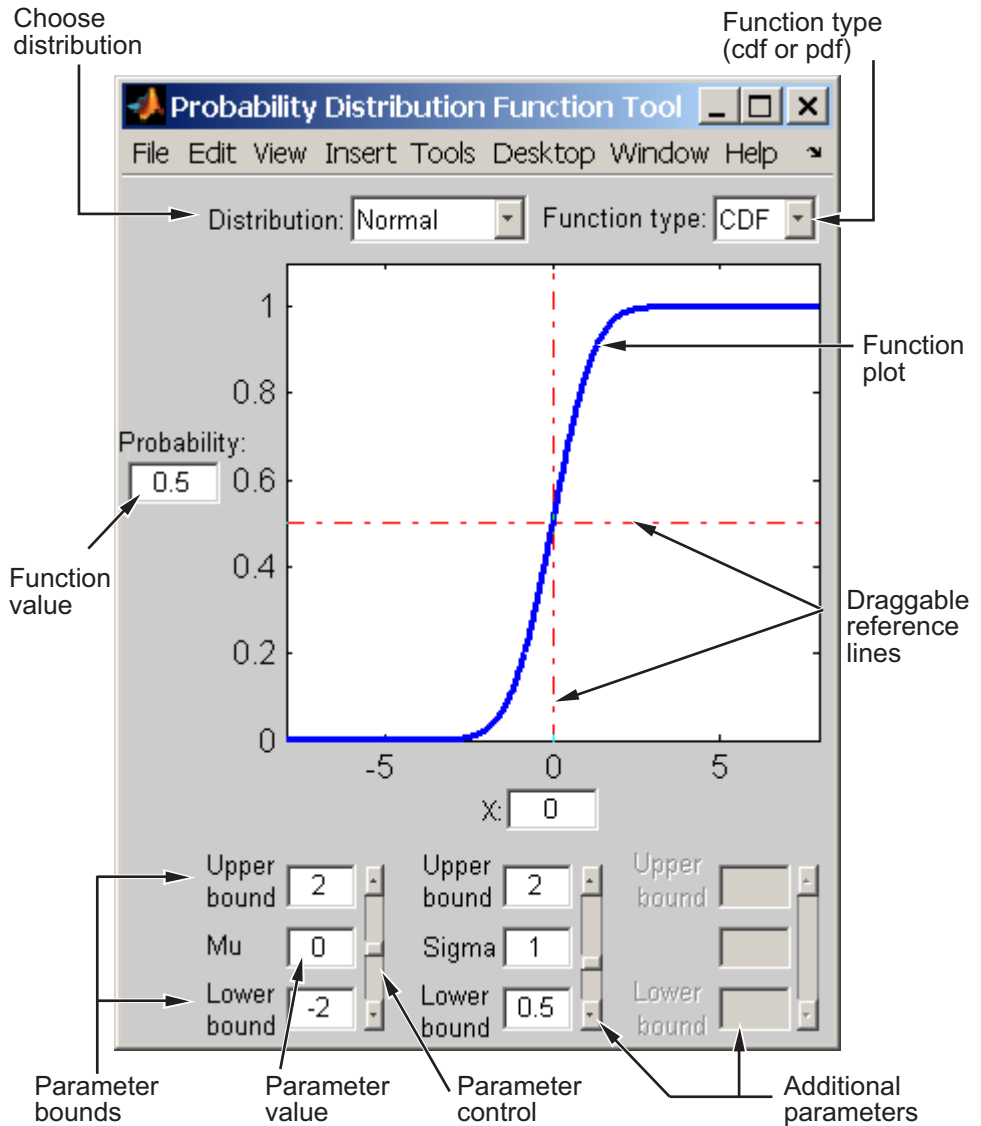
**Concepts**

- “Triangular Distribution” on page B-127

## Explore the Probability Distribution Function UI

The Probability Distribution Function user interface (UI) interactively displays the influence of parameter changes on the shapes of the pdfs and cdfs of supported Statistics Toolbox distributions.

Run the user interface by typing `disttool` at the command line.



Start by selecting a distribution. Then choose the function type: probability density function (pdf) or cumulative distribution function (cdf).

After the plot appears, you can

- Calculate a new function value by
  - Typing a new  $x$  value in the text box on the  $x$ -axis
  - Dragging the vertical reference line.
  - Clicking in the figure where you want the line to be.The new function value appears in the text box to the left of the plot.
- For cdf plots, find critical values corresponding to a specific probability by typing the desired probability in the text box on the  $y$ -axis or by dragging the horizontal reference line.
- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.

## Model Data Using the Distribution Fitting App

The Distribution Fitting app fits univariate distributions to data. This section describes how to use the Distribution Fitting app and covers the following topics:

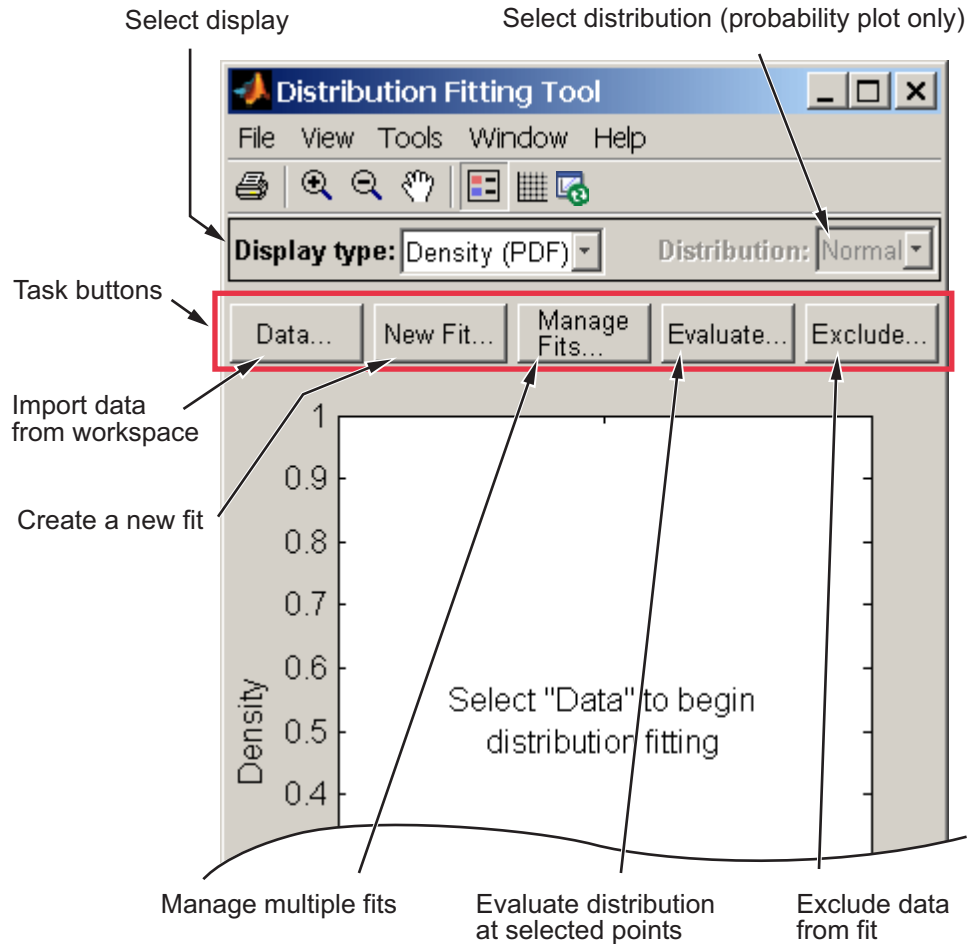
<b>In this section...</b>
“Opening the Distribution Fitting App” on page 5-40
“Creating and Managing Data Sets” on page 5-43
“Creating a New Fit” on page 5-48
“Displaying Results” on page 5-53
“Managing Fits” on page 5-55
“Evaluating Fits” on page 5-57
“Excluding Data” on page 5-61
“Saving and Loading Sessions” on page 5-67
“Example: Fitting a Distribution” on page 5-68
“Generating a File to Fit and Plot Distributions” on page 5-75

### Opening the Distribution Fitting App

Open the Distribution Fitting app by entering:




```
dfittool
```

Alternatively, click Distribution Fitting on the Apps tab.



## Adjusting the Plot

Buttons at the top of the tool allow you to adjust the plot displayed in this window:

-  — Toggle the legend on (default) or off.
-  — Toggle grid lines on or off (default).
-  — Restore default axes limits.

### Displaying the Data

The **Display type** field specifies the type of plot displayed in the main window. Each type corresponds to a probability function, for example, a probability density function. The following display types are available:

- **Density (PDF)** — Display a probability density function (PDF) plot for the fitted distribution.
- **Cumulative probability (CDF)** — Display a cumulative probability plot of the data.
- **Quantile (inverse CDF)** — Display a quantile (inverse CDF) plot.
- **Probability plot** — Display a probability plot.
- **Survivor function** — Display a survivor function plot of the data.
- **Cumulative hazard** — Display a cumulative hazard plot of the data.

### Inputting and Fitting Data

The task buttons enable you to perform the tasks necessary to fit distributions to data. Each button opens a new dialog box in which you perform the task. The buttons include:

- **Data** — Import and manage data sets.
- **New Fit** — Create new fits.
- **Manage Fits** — Manage existing fits.
- **Evaluate** — Evaluate fits at any points you choose.
- **Exclude** — Create rules specifying which values to exclude when fitting a distribution.

The display pane displays plots of the data sets and fits you create. Whenever you make changes in one of the dialog boxes, the results in the display pane update.

### Saving and Customizing Distributions

The Distribution Fitting app menus contain items that enable you to do the following:

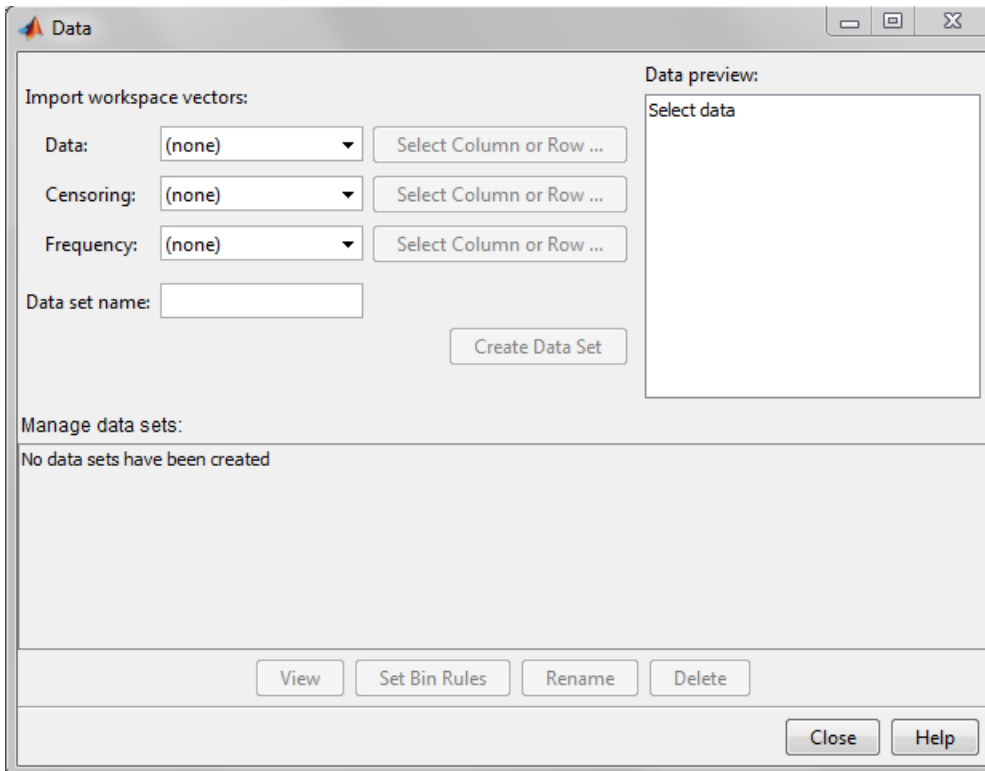


- Save and load sessions.
- Generate a file with which you can fit distributions to data and plot the results independently of the Distribution Fitting app.
- Define and import custom distributions.

## Creating and Managing Data Sets

This section describes how to create and manage data sets.

To begin, click the **Data** button in the Distribution Fitting app to open the Data dialog box shown in the following figure.



## Importing Data

The **Import workspace vectors** pane enables you to create a data set by importing a vector from the MATLAB workspace. The following sections describe the fields in this pane and give appropriate values for vectors imported from the MATLAB workspace:

- **Data** — The drop-down list in the **Data** field contains the names of all matrices and vectors, other than 1-by-1 matrices (scalars) in the MATLAB workspace. Select the array containing the data you want to fit. The actual data you import must be a vector. If you select a matrix in the **Data** field, the first column of the matrix is imported by default. To select a different column or row of the matrix, click **Select Column or Row**. This displays the matrix in the Variables editor, where you can select a row or column by highlighting it with the mouse.

Alternatively, you can enter any valid MATLAB expression in the **Data** field.

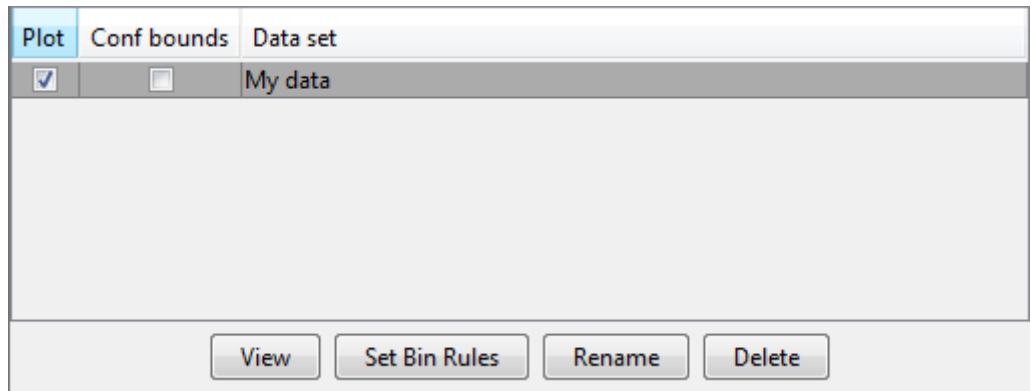
When you select a vector in the **Data** field, a histogram of the data appears in the **Data preview** pane.

- **Censoring** — If some of the points in the data set are censored, enter a Boolean vector, of the same size as the data vector, specifying the censored entries of the data. A 1 in the censoring vector specifies that the corresponding entry of the data vector is censored, while a 0 specifies that the entry is not censored. If you enter a matrix, you can select a column or row by clicking **Select Column or Row**. If you do not want to censor any data, leave the **Censoring** field blank.
- **Frequency** — Enter a vector of positive integers of the same size as the data vector to specify the frequency of the corresponding entries of the data vector. For example, a value of 7 in the 15th entry of frequency vector specifies that there are 7 data points corresponding to the value in the 15th entry of the data vector. If all entries of the data vector have frequency 1, leave the **Frequency** field blank.
- **Data set name** — Enter a name for the data set you import from the workspace, such as `My data`.

After you have entered the information in the preceding fields, click **Create Data Set** to create the data set `My data`.

## Managing Data Sets

The **Manage data sets** pane enables you to view and manage the data sets you create. When you create a data set, its name appears in the **Data sets** list. The following figure shows the **Manage data sets** pane after creating the data set **My data**.



For each data set in the **Data sets** list, you can:

- Select the **Plot** check box to display a plot of the data in the main Distribution Fitting app window. When you create a new data set, **Plot** is selected by default. Clearing the **Plot** check box removes the data from the plot in the main window. You can specify the type of plot displayed in the **Display type** field in the main window.
- If **Plot** is selected, you can also select **Bounds** to display confidence interval bounds for the plot in the main window. These bounds are pointwise confidence bounds around the empirical estimates of these functions. The bounds are only displayed when you set **Display Type** in the main window to one of the following:
  - Cumulative probability (CDF)
  - Survivor function
  - Cumulative hazard

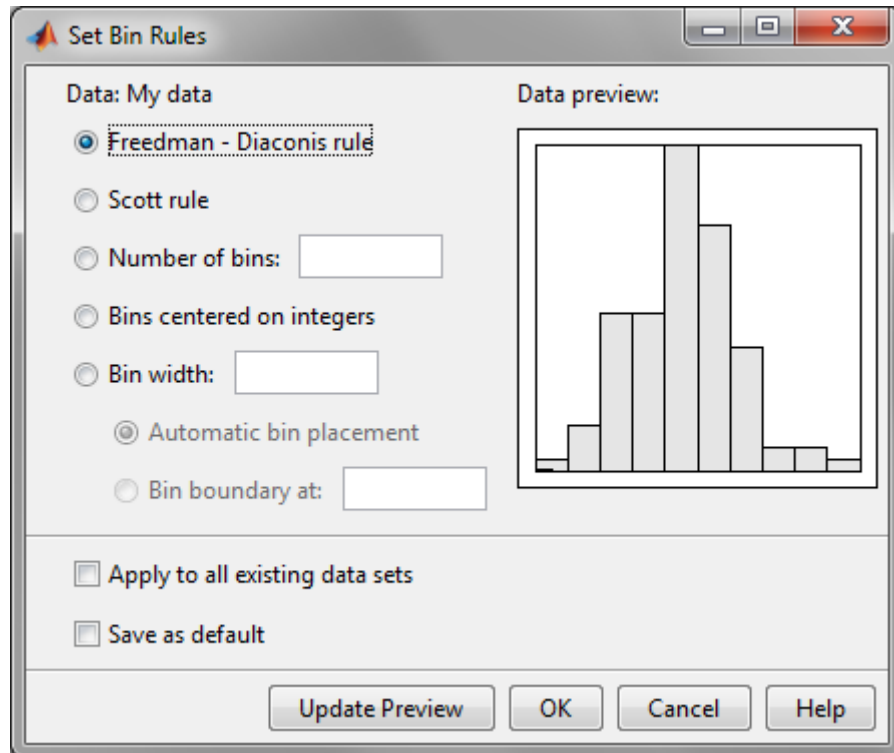
The Distribution Fitting app cannot display confidence bounds on density (PDF), quantile (inverse CDF), or probability plots. Clearing the **Bounds** check box removes the confidence bounds from the plot in the main window.

When you select a data set from the list, the following buttons are enabled:

- **View** — Display the data in a table in a new window.
- **Set Bin Rules** — Defines the histogram bins used in a density (PDF) plot.
- **Rename** — Rename the data set.
- **Delete** — Delete the data set.

### **Setting Bin Rules**

To set bin rules for the histogram of a data set, click **Set Bin Rules**. This opens the **Set Bin Width Rules** dialog box.



You can select from the following rules:

- **Freedman-Diaconis rule** — Algorithm that chooses bin widths and locations automatically, based on the sample size and the spread of the data. This rule, which is the default, is suitable for many kinds of data.
- **Scott rule** — Algorithm intended for data that are approximately normal. The algorithm chooses bin widths and locations automatically.
- **Number of bins** — Enter the number of bins. All bins have equal widths.
- **Bins centered on integers** — Specifies bins centered on integers.
- **Bin width** — Enter the width of each bin. If you select this option, you can also select:
  - **Automatic bin placement** — Place the edges of the bins at integer multiples of the **Bin width**.

- **Bin boundary at** — Enter a scalar to specify the boundaries of the bins. The boundary of each bin is equal to this scalar plus an integer multiple of the **Bin width**.

The Set Bin Width Rules dialog box also provides the following options:

- **Apply to all existing data sets** — Apply the rule to all data sets. Otherwise, the rule is only applied to the data set currently selected in the Data dialog box.
- **Save as default** — Apply the current rule to any new data sets that you create. You can also set default bin width rules by selecting **Set Default Bin Rules** from the **Tools** menu in the main window.

### Creating a New Fit

This section describes how to create a new fit. To begin, click the **New Fit** button at the top of the main window to open the New Fit dialog box. If you created the data set `My data`, it appears in the **Data** field.

New Fit

Fit name:

Data:

Distribution:

Exclusion rule:

Normal

Distribution parameters:

- mu (location)
- sigma (scale)

Apply

Results:

Click "Apply" to fit this distribution

Save to workspace... Manage Fits Close Help

<b>Field Name</b>	<b>Description</b>
Fit Name	Enter a name for the fit in the <b>Fit Name</b> field.
Data	The <b>Data</b> field contains a drop-down list of the data sets you have created. Select the data set to which you want to fit a distribution.
Distribution	<p>Select the type of distribution to fit from the <b>Distribution</b> drop-down list.</p> <p>Only the distributions that apply to the values of the selected data set appear in the <b>Distribution</b> field. For example, positive distributions are not displayed when the data include values that are zero or negative.</p> <p>You can specify either a parametric or a nonparametric distribution. When you select a parametric distribution from the drop-down list, a description of its parameters appears in the Normal pane. The Distribution Fitting Tool estimates these parameters to fit the distribution to the data set. When you select <b>Nonparametric fit</b>, options for the fit appear in the pane, as described in “Further Options for Nonparametric Fits” on page 5-52.</p>
Exclusion rule	Specify a rule to exclude some data in the <b>Exclusion rule</b> field. Create an exclusion rule by clicking <b>Exclude</b> in the Distribution Fitting app. For more information, see “Excluding Data” on page 5-61.

### **Apply the New Fit**

Click **Apply** to fit the distribution. For a parametric fit, the **Results** pane displays the values of the estimated parameters. For a nonparametric fit, the **Results** pane displays information about the fit.

When you click **Apply**, the Distribution Fitting app displays a plot of the distribution, along with the corresponding data.



---

**Note** When you click **Apply**, the title of the dialog box changes to **Edit Fit**. You can now make changes to the fit you just created and click **Apply** again to save them. After closing the **Edit Fit** dialog box, you can reopen it from the **Fit Manager** dialog box at any time to edit the fit.

---

After applying the fit, you can save the information to the workspace using probability distribution objects by clicking **Save to workspace**.

### Available Distributions

Most, but not all, of the distributions available in the Distribution Fitting app are supported elsewhere in Statistics Toolbox software, and have dedicated distribution fitting functions. These functions compute the majority of the fits in the Distribution Fitting app, and are referenced in the list below. Other fits are computed using functions internal to the Distribution Fitting app.

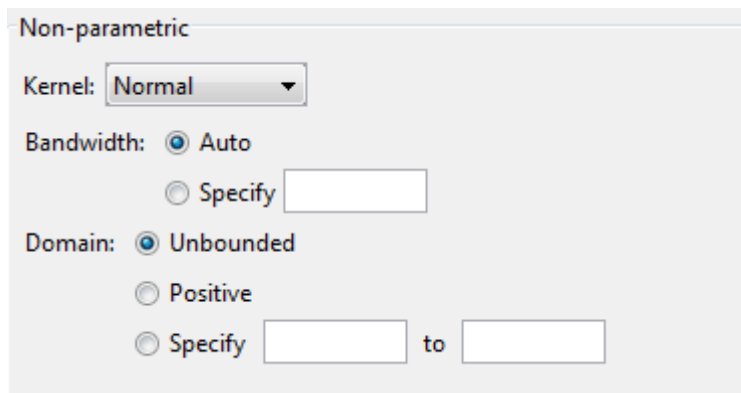
Not all of the distributions listed below are available for all data sets. The Distribution Fitting app determines the extent of the data (nonnegative, unit interval, etc.) and displays appropriate distributions in the **Distribution** drop-down list. Distribution data ranges are given parenthetically in the list below.

- Beta (unit interval values) distribution, fit using the function `betafit`.
- Binomial (nonnegative values) distribution, fit using the function `binopdf`.
- Birnbaum-Saunders (positive values) distribution.
- Burr Type XII (positive values) distribution.
- Exponential (nonnegative values) distribution, fit using the function `expfit`.
- Extreme value (all values) distribution, fit using the function `evfit`.
- Gamma (positive values) distribution, fit using the function `gamfit`.
- Generalized extreme value (all values) distribution, fit using the function `gevfit`.
- Generalized Pareto (all values) distribution, fit using the function `gpfit`.
- Inverse Gaussian (positive values) distribution.

- Logistic (all values) distribution.
- Loglogistic (positive values) distribution.
- Lognormal (positive values) distribution, fit using the function `lognfit`.
- Nakagami (positive values) distribution.
- Negative binomial (nonnegative values) distribution, fit using the function `nbinpdf`.
- Nonparametric (all values) distribution, fit using the function `ksdensity`.
- Normal (all values) distribution, fit using the function `normfit`.
- Poisson (nonnegative integer values) distribution, fit using the function `poisspdf`.
- Rayleigh (positive values) distribution using the function `raylfit`.
- Rician (positive values) distribution.
- *t* location-scale (all values) distribution.
- Weibull (positive values) distribution using the function `wblfit`.

### Further Options for Nonparametric Fits

When you select Non-parametric in the **Distribution** field, a set of options appears in the **Non-parametric** pane, as shown in the following figure.



The options for nonparametric distributions are:

- **Kernel** — Type of kernel function to use.
  - Normal
  - Box
  - Triangle
  - Epanechnikov
- **Bandwidth** — The bandwidth of the kernel smoothing window. Select **Auto** for a default value that is optimal for estimating normal densities. This value appears in the **Fit results** pane after you click **Apply**. Select **Specify** and enter a smaller value to reveal features such as multiple modes or a larger value to make the fit smoother.
- **Domain** — The allowed  $x$ -values for the density.
  - **Unbounded** — The density extends over the whole real line.
  - **Positive** — The density is restricted to positive values.
  - **Specify** — Enter lower and upper bounds for the domain of the density.

When you select **Positive** or **Specify**, the nonparametric fit has zero probability outside the specified domain.

## Displaying Results

This section explains the different ways to display results in the Distribution Fitting app window. This window displays plots of:

- The data sets for which you select **Plot** in the Data dialog box
- The fits for which you select **Plot** in the Fit Manager dialog box
- Confidence bounds for:
  - Data sets for which you select **Bounds** in the Data dialog box
  - Fits for which you select **Bounds** in the Fit Manager dialog box

The following fields are available.

## Display Type

The **Display Type** field in the main window specifies the type of plot displayed. Each type corresponds to a probability function, for example, a probability density function. The following display types are available:

- **Density (PDF)** — Display a probability density function (PDF) plot for the fitted distribution. The main window displays data sets using a probability histogram, in which the height of each rectangle is the fraction of data points that lie in the bin divided by the width of the bin. This makes the sum of the areas of the rectangles equal to 1.
- **Cumulative probability (CDF)** — Display a cumulative probability plot of the data. The main window displays data sets using a cumulative probability step function. The height of each step is the cumulative sum of the heights of the rectangles in the probability histogram.
- **Quantile (inverse CDF)** — Display a quantile (inverse CDF) plot.
- **Probability plot** — Display a probability plot of the data. You can specify the type of distribution used to construct the probability plot in the **Distribution** field, which is only available when you select **Probability plot**. The choices for the distribution are:
  - Exponential
  - Extreme value
  - Logistic
  - Log-Logistic
  - Lognormal
  - Normal
  - Rayleigh
  - Weibull

In addition to these choices, you can create a probability plot against a parametric fit that you create in the **New Fit** pane. These fits are added at the bottom of the **Distribution** drop-down list when you create them.

- **Survivor function** — Display survivor function plot of the data.
- **Cumulative hazard** — Display cumulative hazard plot of the data.

---

**Note** Some distributions are unavailable if the plotted data includes 0 or negative values.

---

## Confidence Bounds

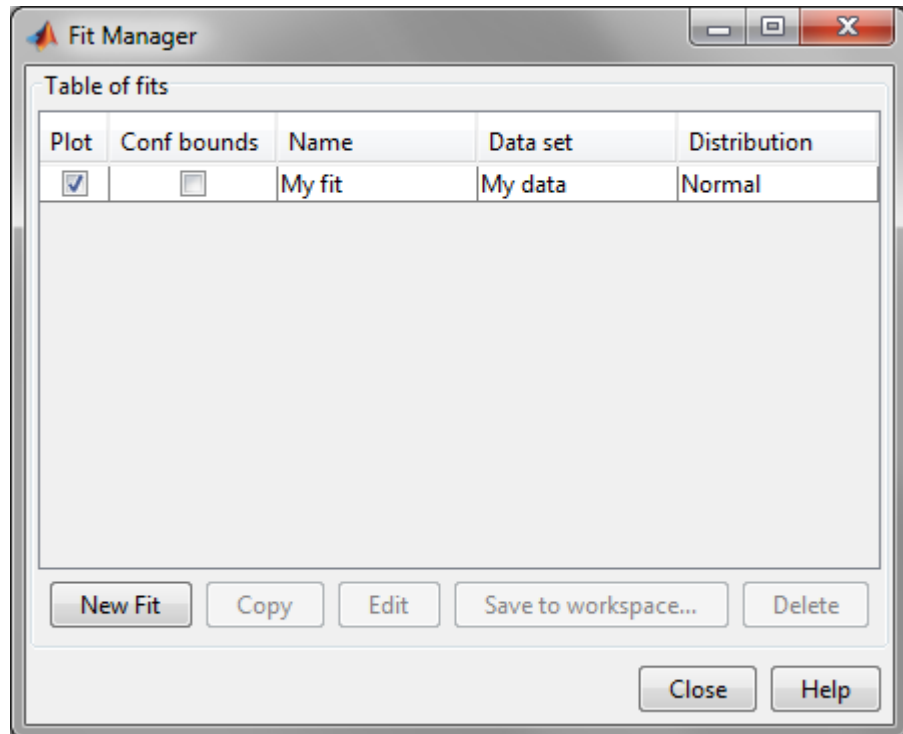
You can display confidence bounds for data sets and fits when you set **Display Type** to **Cumulative probability (CDF)**, **Survivor function**, **Cumulative hazard**, or, for fits only, **Quantile (inverse CDF)**.

- To display bounds for a data set, select **Bounds** next to the data set in the **Data sets** pane of the Data dialog box.
- To display bounds for a fit, select **Bounds** next to the fit in the Fit Manager dialog box. Confidence bounds are not available for all fit types.

To set the confidence level for the bounds, select **Confidence Level** from the **View** menu in the main window and choose from the options.

## Managing Fits

This section describes how to manage fits that you have created. To begin, click the **Manage Fits** button in the Distribution Fitting app. This opens the Fit Manager dialog box as shown in the following figure.



The **Table of fits** displays a list of the fits you create, with the following options:

- **Plot** — Select **Plot** to display a plot of the fit in the main window of the Distribution Fitting app. When you create a new fit, **Plot** is selected by default. Clearing the **Plot** check box removes the fit from the plot in the main window.
- **Bounds** — If **Plot** is selected, you can also select **Bounds** to display confidence bounds in the plot. The bounds are displayed when you set **Display Type** in the main window to one of the following:
  - Cumulative probability (CDF)
  - Quantile (inverse CDF)
  - Survivor function

- **Cumulative hazard**

The Distribution Fitting app cannot display confidence bounds on density (PDF) or probability plots. In addition, bounds are not supported for nonparametric fits and some parametric fits.

Clearing the **Bounds** check box removes the confidence intervals from the plot in the main window.

When you select a fit in the **Table of fits**, the following buttons are enabled below the table:

- **New Fit** — Open a New Fit window.
- **Copy** — Create a copy of the selected fit.
- **Edit** — Open an Edit Fit dialog box, where you can edit the fit.

---

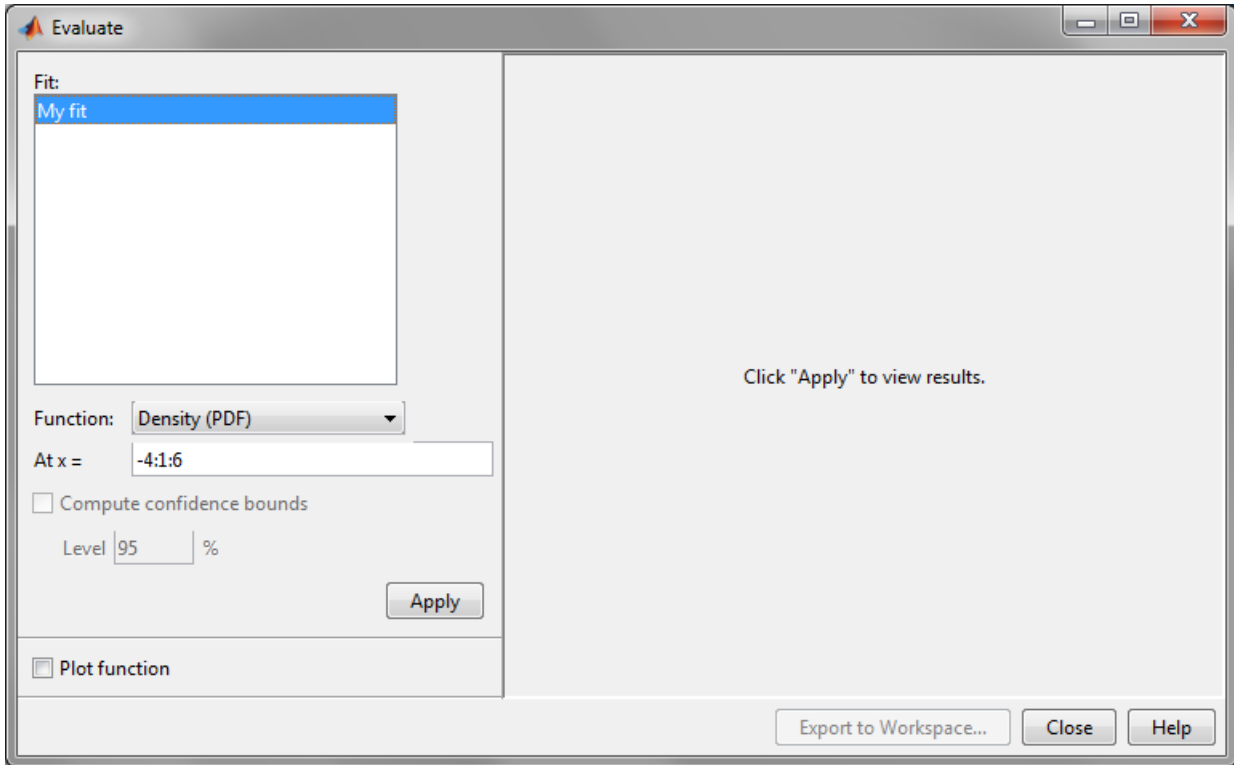
**Note** You can only edit the currently selected fit in the Edit Fit dialog box. To edit a different fit, select it in the **Table of fits** and click **Edit** to open another Edit Fit dialog box.

---

- **Save to workspace** — Save the selected fit as a distribution object.
- **Delete** — Delete the selected fit.

## Evaluating Fits

The Evaluate dialog box enables you to evaluate any fit at whatever points you choose. To open the dialog box, click the **Evaluate** button in the Distribution Fitting app. The following figure shows the Evaluate dialog box.



The Evaluate dialog box contains the following items:

- **Fit** pane — Display the names of existing fits. Select one or more fits that you want to evaluate. Using your platform specific functionality, you can select multiple fits.
- **Function** — Select the type of probability function you want to evaluate for the fit. The available functions are
  - Density (PDF) — Computes a probability density function.
  - Cumulative probability (CDF) — Computes a cumulative probability function.
  - Quantile (inverse CDF) — Computes a quantile (inverse CDF) function.



- Survivor function — Computes a survivor function.
- Cumulative hazard — Computes a cumulative hazard function.
- Hazard rate — Computes the hazard rate.
- **At  $\mathbf{x} =$**  — Enter a vector of points or the name of a workspace variable containing a vector of points at which you want to evaluate the distribution function. If you change **Function** to **Quantile (inverse CDF)**, the field name changes to **At  $\mathbf{p} =$**  and you enter a vector of probability values.
- **Compute confidence bounds** — Select this box to compute confidence bounds for the selected fits. The check box is only enabled if you set **Function** to one of the following:
  - Cumulative probability (CDF)
  - Quantile (inverse CDF)
  - Survivor function
  - Cumulative hazard

The Distribution Fitting app cannot compute confidence bounds for nonparametric fits and for some parametric fits. In these cases, it returns NaN for the bounds.

- **Level** — Set the level for the confidence bounds.
- **Plot function** — Select this box to display a plot of the distribution function, evaluated at the points you enter in the **At  $\mathbf{x} =$**  field, in a new window.

---

**Note** The settings for **Compute confidence bounds**, **Level**, and **Plot function** do not affect the plots that are displayed in the main window of the Distribution Fitting app. The settings only apply to plots you create by clicking **Plot function** in the Evaluate window.

---

Click **Apply** to apply these settings to the selected fit. The following figure shows the results of evaluating the cumulative density function for the fit `My fit`, at the points in the vector `-4:1:6`.

X	My fit		
	F(X)	LB	UB
-4	0.00268	0.00064	0.0094
-3	0.01498	0.00568	0.03518
-2	0.05984	0.03225	0.10321
-1	0.1732	0.11948	0.24022
0	0.37179	0.29863	0.44992
1	0.61309	0.53479	0.68703
2	0.81644	0.74822	0.87178
3	0.93529	0.89025	0.96446
4	0.98345	0.96201	0.99356
5	0.99698	0.98969	0.99925
6	0.99961	0.99782	0.99995

The window displays the following values in the columns of the table to the right of the **Fit** pane:

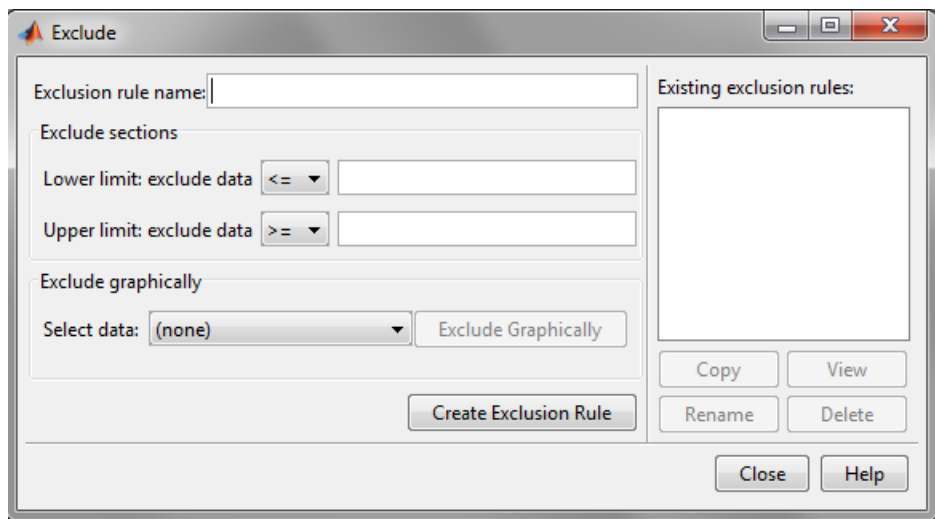
- X — The entries of the vector you enter in **At x =** field
- F(X)— The corresponding values of the CDF at the entries of X
- LB — The lower bounds for the confidence interval, if you select **Compute confidence bounds**

- UB — The upper bounds for the confidence interval, if you select **Compute confidence bounds**

To save the data displayed in the Evaluate window, click **Export to Workspace**. This saves the values in the table to a matrix in the MATLAB workspace.

## Excluding Data

To exclude values from fit, click the **Exclude** button in the main window of the Distribution Fitting app. This opens the Exclude window, in which you can create rules for excluding specified values. You can use these rules to exclude data when you create a new fit in the New Fit window. The following figure shows the Exclude window.



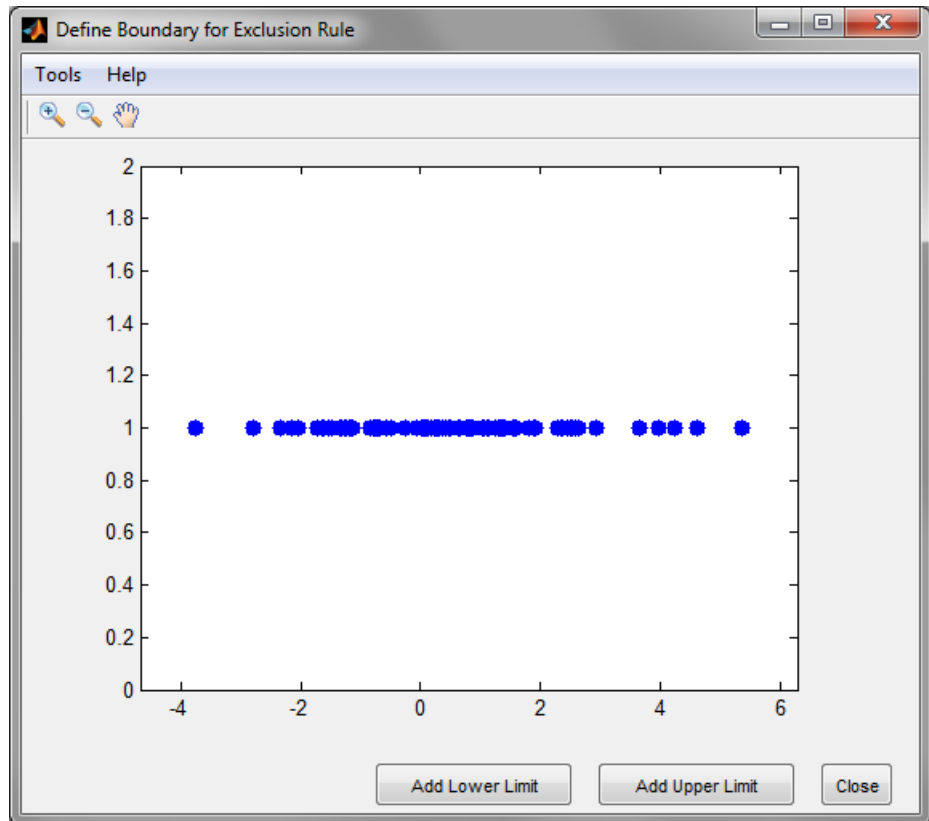
To create an exclusion rule:

- 1 Exclusion Rule Name**—Enter a name for the exclusion rule in the **Exclusion rule name** field.
- 2 Exclude Sections**—In the **Exclude sections** pane, you can specify bounds for the excluded data:

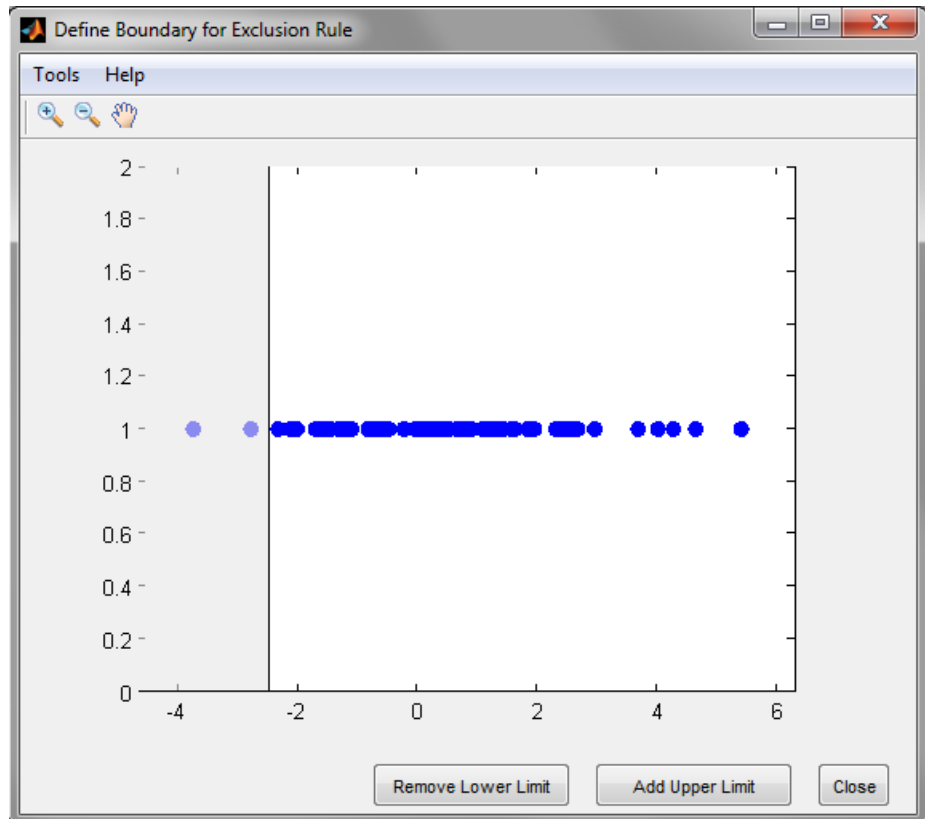
- In the **Lower limit: exclude Y** drop-down list, select  $\leq$  or  $<$  from the drop-down list and enter a scalar in the field to the right. This excludes values that are either less than or equal to or less than that scalar, respectively.
- In the **Upper limit: exclude Y** drop-down list, select  $\geq$  or  $>$  from the drop-down list and enter a scalar in the field to the right to exclude values that are either greater than or equal to or greater than the scalar, respectively.

**OR**

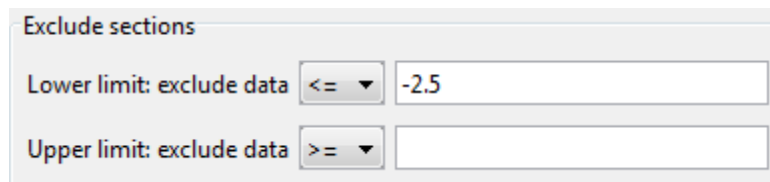
**Exclude Graphically**—The **Exclude Graphically** button enables you to define the exclusion rule by displaying a plot of the values in a data set and selecting the bounds for the excluded data with the mouse. For example, if you created the data set `My data`, described in *Creating and Managing Data Sets*, select it from the drop-down list next to **Exclude graphically** and then click the **Exclude graphically** button. This displays the values in `My data` in a new window as shown in the following figure.



To set a lower limit for the boundary of the excluded region, click **Add Lower Limit**. This displays a vertical line on the left side of the plot window. Move the line with the mouse to the point you where you want the lower limit, as shown in the following figure.



Moving the vertical line changes the value displayed in the **Lower limit: exclude data** field in the Exclude window, as shown in the following figure.



The value displayed corresponds to the  $x$ -coordinate of the vertical line.

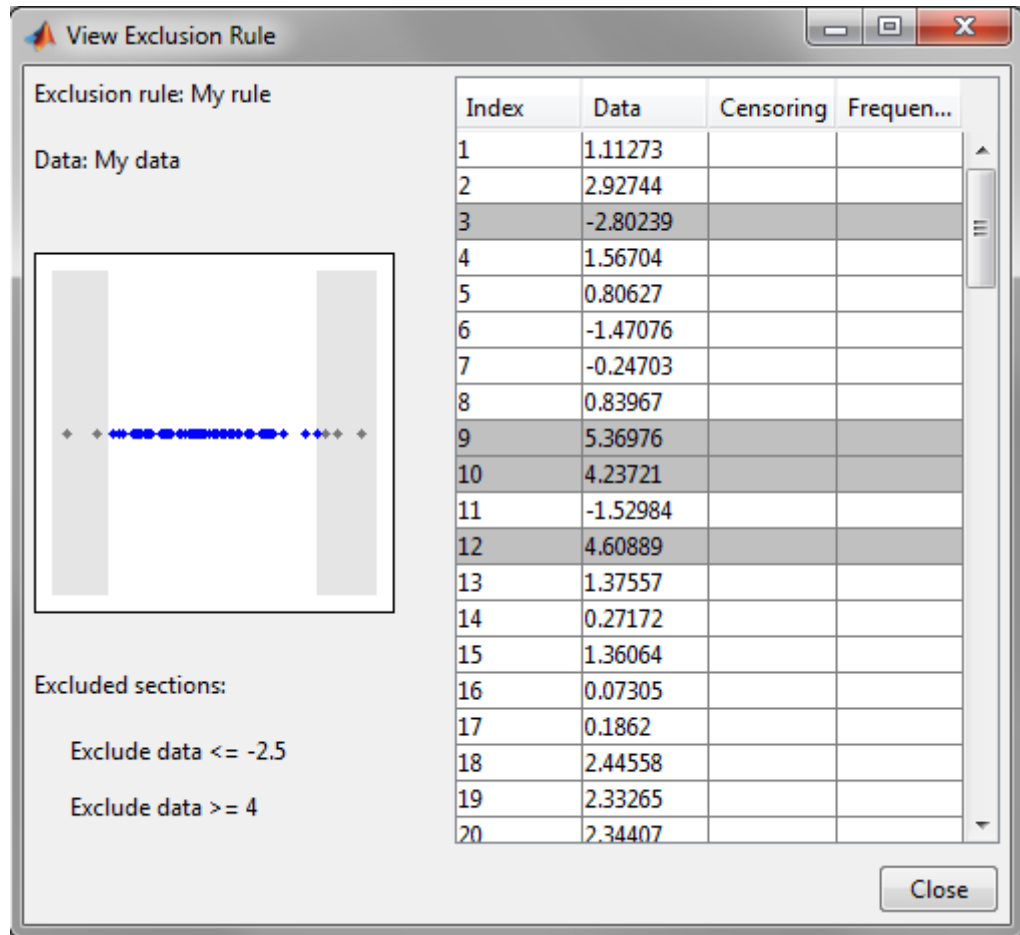
Similarly, you can set the upper limit for the boundary of the excluded region by clicking **Add Upper Limit** and moving the vertical line that

appears at the right side of the plot window. After setting the lower and upper limits, click **Close** and return to the Exclude window.

- 3 Create Exclusion Rule**—Once you have set the lower and upper limits for the boundary of the excluded data, click **Create Exclusion Rule** to create the new rule. The name of the new rule now appears in the **Existing exclusion rules** pane.

When you select an exclusion rule in the **Existing exclusion rules** pane, the following buttons are enabled:

- **Copy** — Creates a copy of the rule, which you can then modify. To save the modified rule under a different name, click **Create Exclusion Rule**.
- **View** — Opens a new window in which you can see which data points are excluded by the rule. The following figure shows a typical example.



The shaded areas in the plot graphically display which data points are excluded. The table to the right lists all data points. The shaded rows indicate excluded points:

- **Rename** — Renames the rule
- **Delete** — Deletes the rule

Once you define an exclusion rule, you can use it when you fit a distribution to your data. The rule does not exclude points from the display of the data set.



## Saving and Loading Sessions

This section explains how to save your work in the current Distribution Fitting app session and then load it in a subsequent session, so that you can continue working where you left off.

### Saving a Session

To save the current session, select **Save Session** from the **File** menu in the main window. This opens a dialog box that prompts you to enter a filename, such as `my_session.dfit`, for the session. Clicking **Save** saves the following items created in the current session:

- Data sets
- Fits
- Exclusion rules
- Plot settings
- Bin width rules

### Loading a Session

To load a previously saved session, select **Load Session** from the **File** menu in the main window and enter the name of a previously saved session. Clicking **Open** restores the information from the saved session to the current session of the Distribution Fitting app.

## Example: Fitting a Distribution

This section presents an example that illustrates how to use the Distribution Fitting app. The example involves the following steps:

- “Step 1: Generate Random Data” on page 5-68
- “Step 2: Import Data” on page 5-68
- “Step 3: Create a New Fit” on page 5-71

### Step 1: Generate Random Data

To try the example, first generate some random data to which you will fit a distribution. The following command generates a vector `data`, of length 100, whose entries are random numbers from a normal distribution with mean `.36` and standard deviation `1.4`.

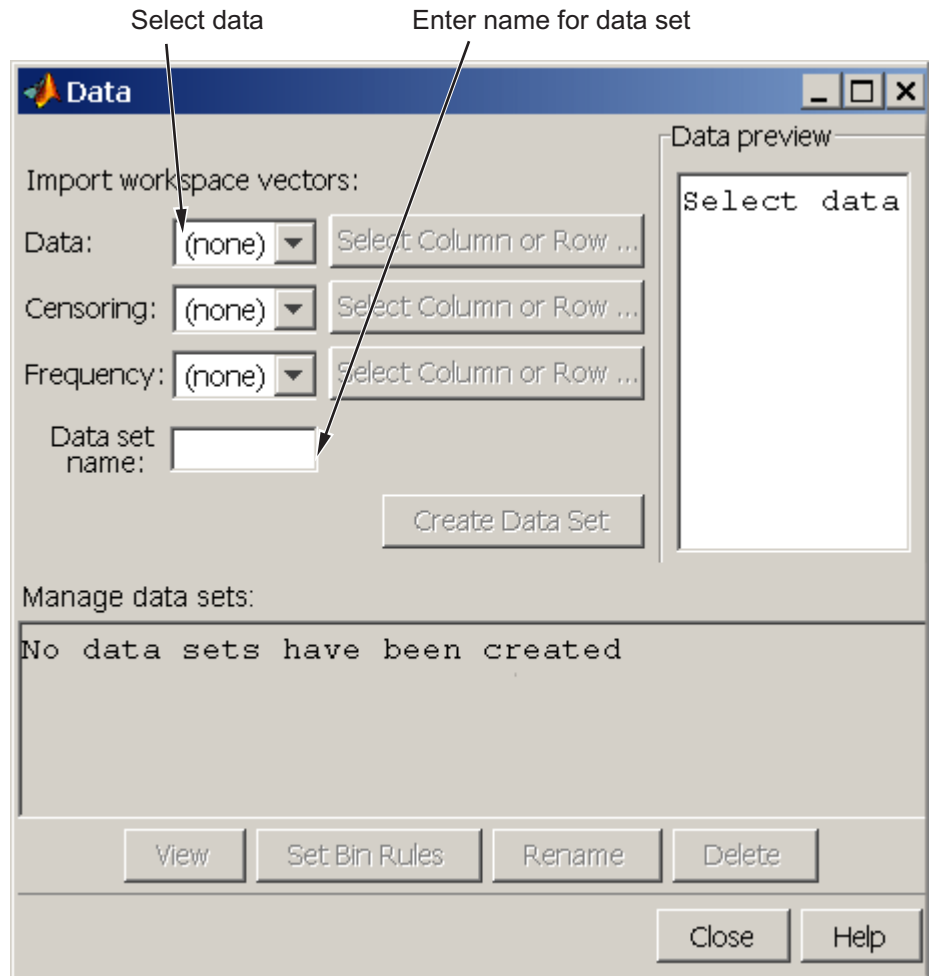
```
rng('default')  
data = normrnd(.36, 1.4, 100, 1);
```

### Step 2: Import Data

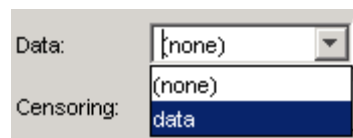
Open the distribution fitting tool:

```
dfittool
```

To import the vector `data` into the Distribution Fitting app, click the **Data** button in main window. This opens the window shown in the following figure.

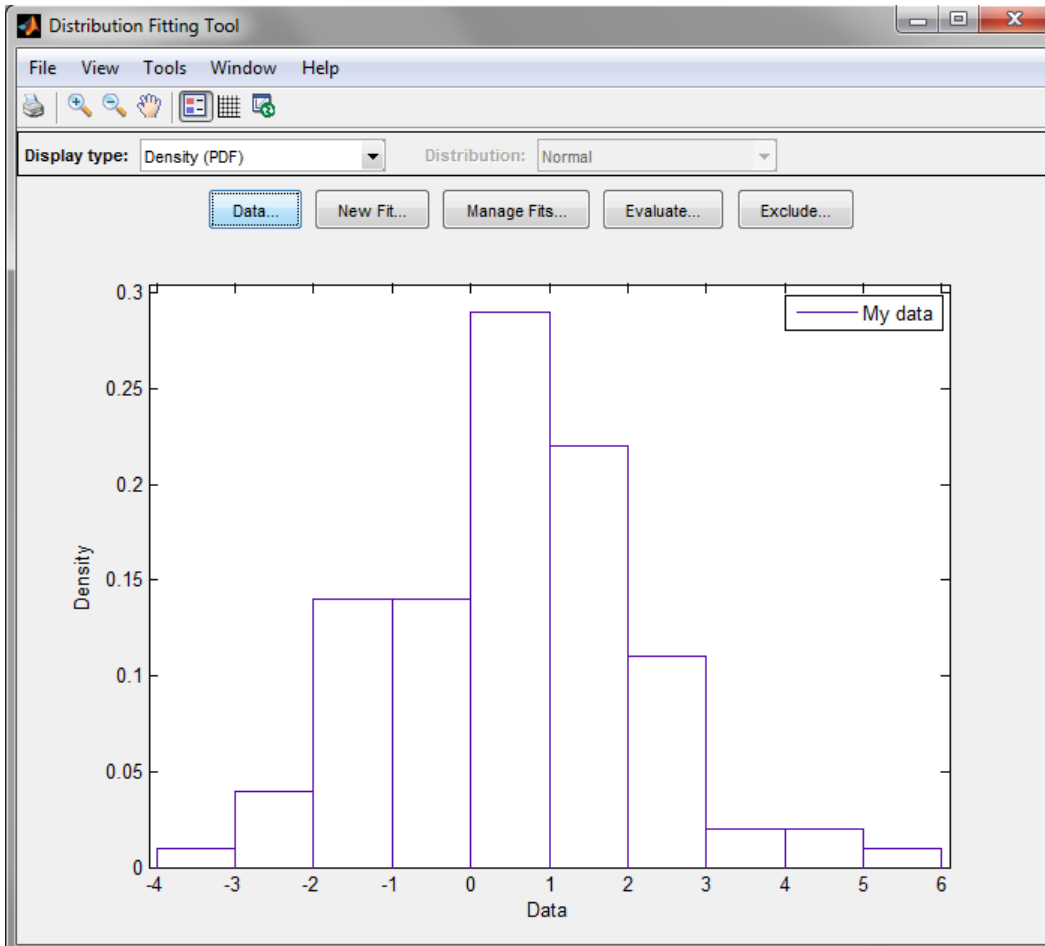


The **Data** field displays all numeric arrays in the MATLAB workspace. Select data from the drop-down list, as shown in the following figure.



This displays a histogram of the data in the **Data preview** pane.

In the **Data set name** field, type a name for the data set, such as **My data**, and click **Create Data Set** to create the data set. The main window of the Distribution Fitting app now displays a larger version of the histogram in the **Data preview** pane, as shown in the following figure.



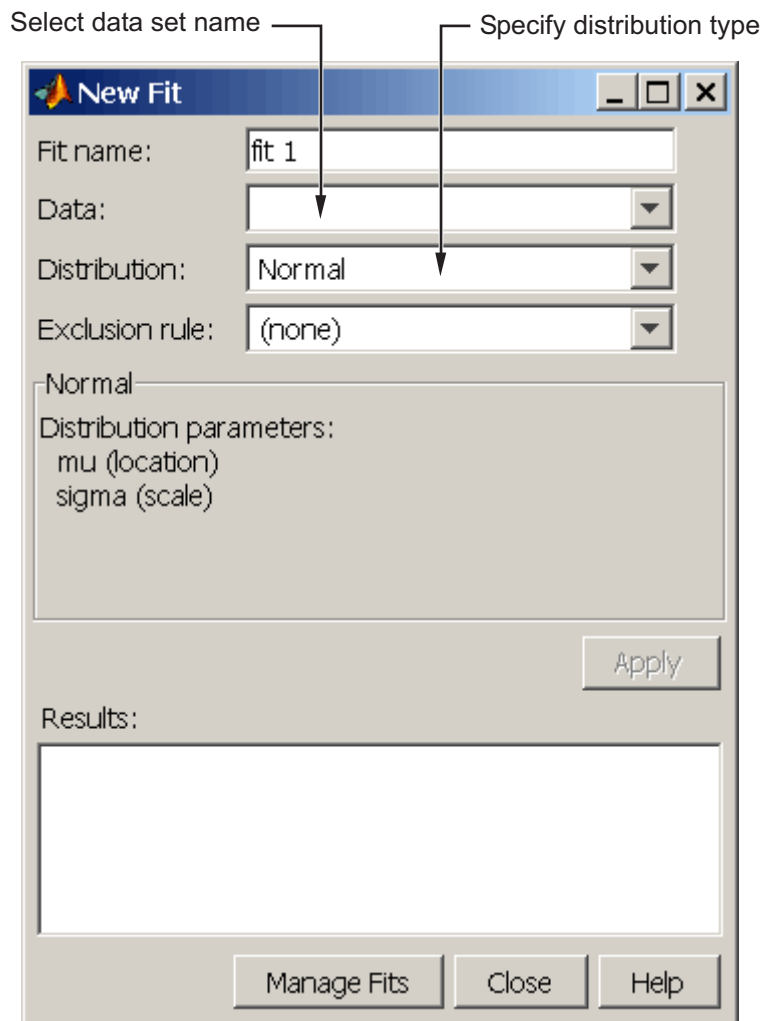
---

**Note** Because the example uses random data, you might see a slightly different histogram if you try this example for yourself.

---

### **Step 3: Create a New Fit**

To fit a distribution to the data, click **New Fit** in the main window of the Distribution Fitting app. This opens the window shown in the following figure.

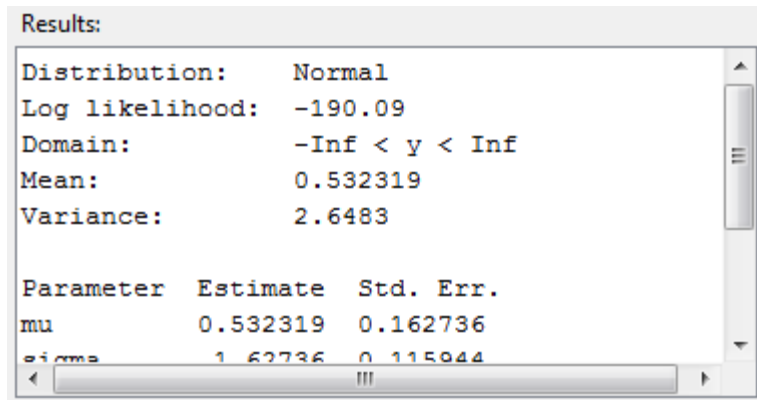


To fit a normal distribution, the default entry of the **Distribution** field, to My data:

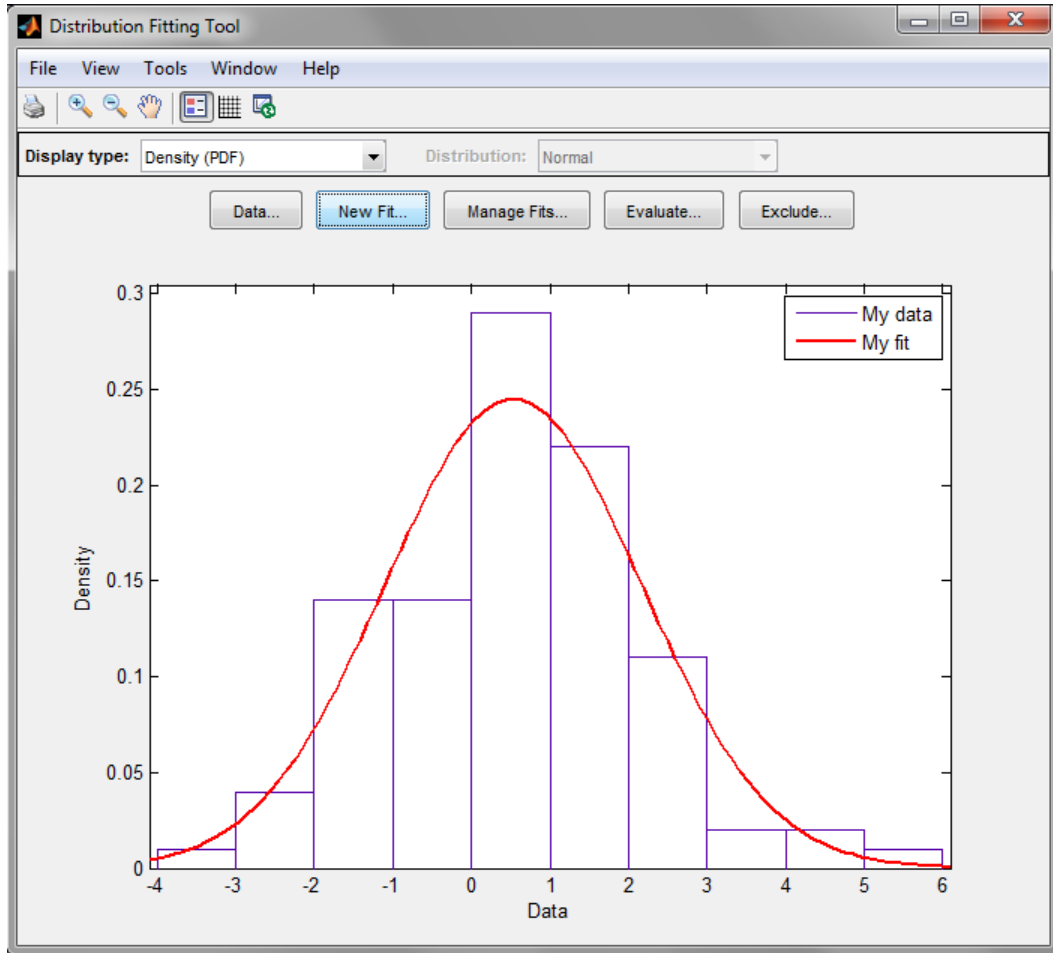
- 1 Enter a name for the fit, such as My fit, in the **Fit name** field.
- 2 Select My data from the drop-down list in the **Data** field.

**3** Click **Apply**.

The **Results** pane displays the mean and standard deviation of the normal distribution that best fits My data, as shown in the following figure.



The main window of the Distribution Fitting app displays a plot of the normal distribution with this mean and standard deviation, as shown in the following figure.





## Generating a File to Fit and Plot Distributions

The **Generate Code** option in the **File** menu enables you to create a file that

- Fits the distributions used in the current session to any data vector in the MATLAB workspace.
- Plots the data and the fits.

After you end the current session, you can use the file to create plots in a standard MATLAB figure window, without having to reopen the Distribution Fitting app.

As an example, assuming you created the fit described in “Creating a New Fit” on page 5-48, do the following steps:

- 1** Select **Generate Code** from the **File** menu.
- 2** Choose **File > Save as** in the MATLAB Editor window. Save the file as `normal_fit.m` in a folder on the MATLAB path.

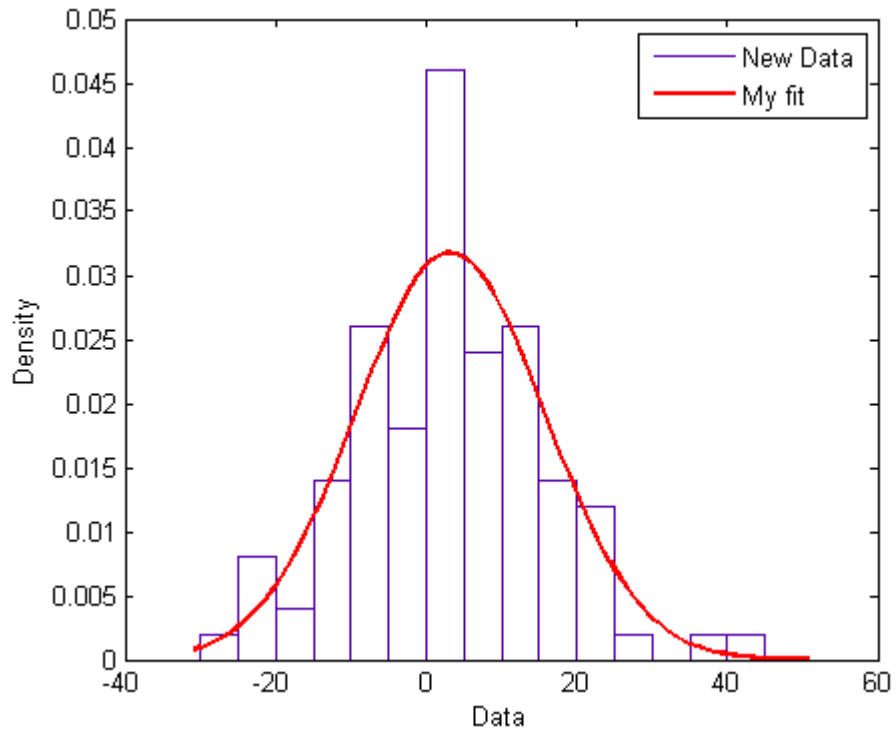
You can then apply the function `normal_fit` to any vector of data in the MATLAB workspace. For example, the following commands

```
new_data = normrnd(4.1, 12.5, 100, 1);
newfit = normal_fit(new_data)
legend('New Data', 'My fit')
```

generate `newfit`, a fitted normal distribution of the data, and generates a plot of the data and the fit.

```
newfit =
normal distribution

    mu = 3.19148
    sigma = 12.5631
```



---

**Note** By default, the file labels the data in the legend using the same name as the data set in the Distribution Fitting app. You can change the label using the `legend` command, as illustrated by the preceding example.

---

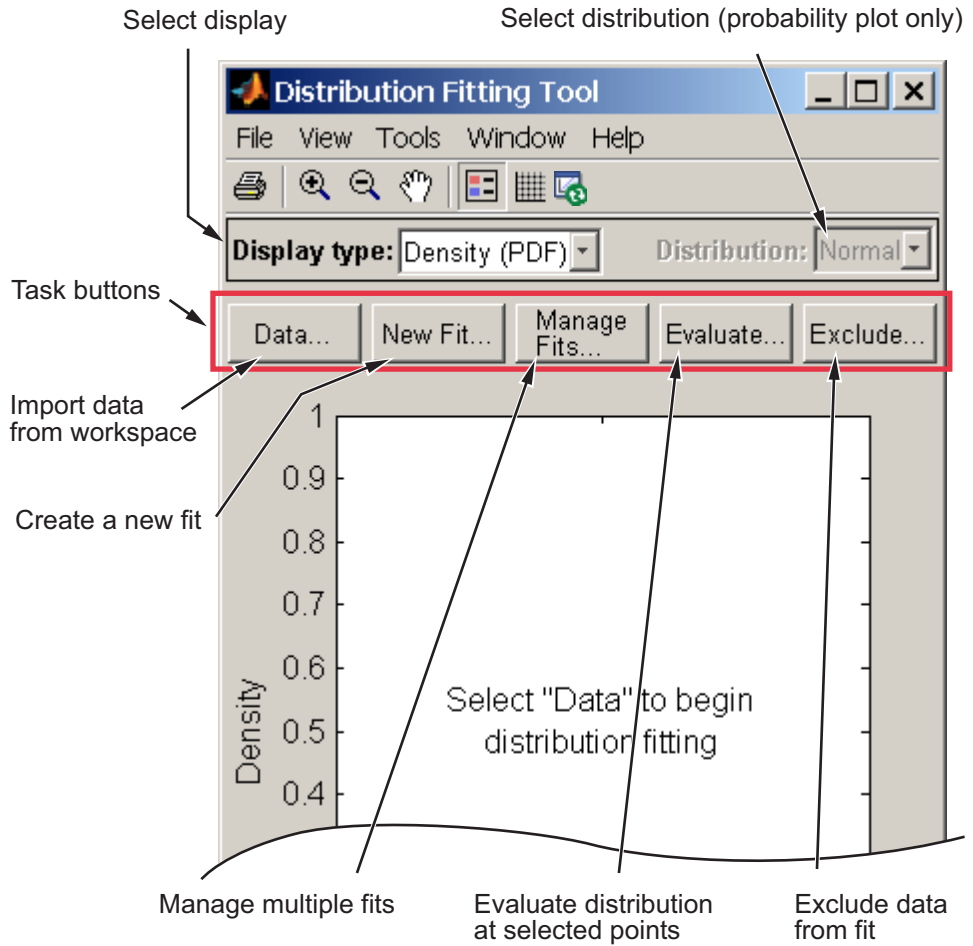
## Custom Distributions Using the Distribution Fitting App

You can use the Distribution Fitting app to fit distributions not supported by the Statistics Toolbox by defining a custom distribution.

### Opening the Distribution Fitting App

To open the Distribution Fitting app, enter the command

```
dfittool
```



Alternatively, click Distribution Fitting on the Apps tab.

## Defining Custom Distributions

To define a custom distribution, select **Define Custom Distribution** from the **File** menu. This opens a file template in the MATLAB editor. You then edit this file so that it computes the distribution you want.

The template includes example code that computes the Laplace distribution. Follow the instructions in the template to define your own custom distribution.

To save your custom distribution, create a directory called `+prob` on your path. Save the file in this directory using a name that matches your distribution name. If you save the template in a folder on the MATLAB path, under its default name `dfittooldists.m`, the Distribution Fitting app reads it in automatically when you start the tool. You can also save the template under a different name, such as `laplace.m`, and then import the custom distribution as described in the following section.

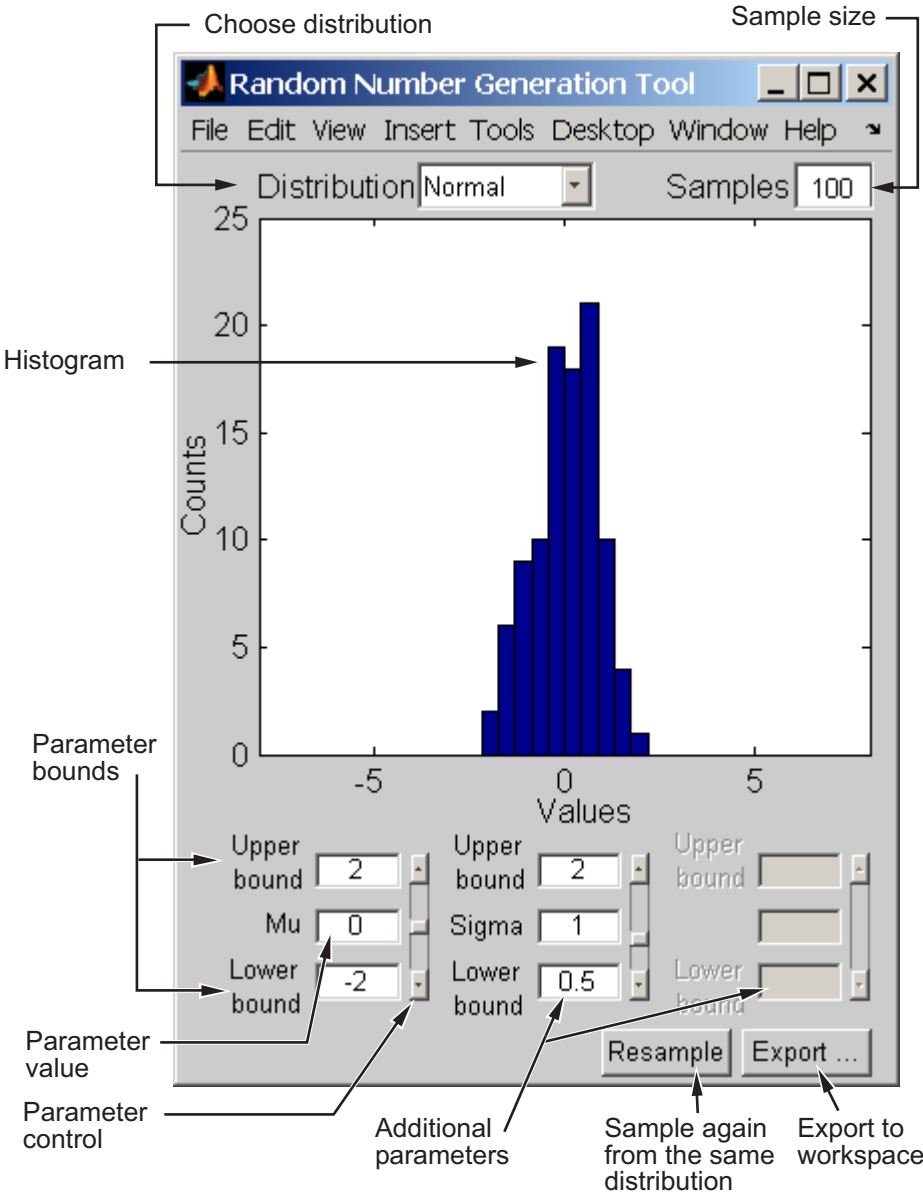
## Importing Custom Distributions

To import a custom distribution, select **Import Custom Distributions** from the **File** menu. This opens a dialog box in which you can select the file that defines the distribution. For example, if you created the file `laplace.m`, as described in the preceding section, you can enter `laplace.m` and select **Open** in the dialog box. The **Distribution** field of the New Fit window now contains the option **Laplace**.

### Explore the Random Number Generation UI

The Random Number Generation user interface (UI) generates random samples from specified probability distributions, and displays the samples as histograms. Use the interface to explore the effects of changing parameters and sample size on the distributions.

Run the user interface by typing `randtool` at the command line.



Start by selecting a distribution, then enter the desired sample size.

You can also

- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.
- Draw another sample from the same distribution, with the same size and parameters.
- Export the current sample to your workspace. A dialog box enables you to provide a name for the sample.



## Compare Multiple Distribution Fits

This example shows how to fit multiple probability distribution objects to the same set of sample data, and obtain a visual comparison of how well each distribution fits the data.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

This data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (`Origin`), model year (`Model_Year`), and other vehicle characteristics.

### Step 2. Create a nominal array.

Transform `Origin` into a nominal array and remove the Italian car from the sample data.

```
Origin = nominal(Origin);
MPG2 = MPG(Origin~='Italy');
Origin2 = Origin(Origin~='Italy');
Origin2 = droplevels(Origin2, 'Italy');
```

Since there is only one Italian car, `fitdist` cannot fit a distribution to that group. Removing the Italian car from the sample data prevents `fitdist` from producing an error.

### Step 3. Fit multiple distributions by group.

Use `fitdist` to fit Weibull, normal, logistic, and kernel distributions to each country of origin group in the MPG data.

```
[WeiByOrig,Country] = fitdist(MPG2, 'weibull', 'by', Origin2);
[NormByOrig,Country] = fitdist(MPG2, 'normal', 'by', Origin2);
[LogByOrig,Country] = fitdist(MPG2, 'logistic', 'by', Origin2);
[KerByOrig,Country] = fitdist(MPG2, 'kernel', 'by', Origin2);
```

```
WeiByOrig
```

Country

WeiByOrig =

Column 1

[1x1 prob.WeibullDistribution]

Column 2

[1x1 prob.WeibullDistribution]

Column 3

[1x1 prob.WeibullDistribution]

Column 4

[1x1 prob.WeibullDistribution]

Column 5

[1x1 prob.WeibullDistribution]

Country =

'France'

'Germany'

'Japan'

'Sweden'

'USA'

Each country group now has four distribution objects associated with it. For example, the cell array `WeiByOrig` contains five Weibull distribution objects, one for each country represented in the sample data. Likewise, the cell array `NormByOrig` contains five normal distribution objects, and so on. Each object contains properties that hold information about the data, distribution, and parameters. The array `Country` lists the country of origin for each group in the same order as the distribution objects are stored in the cell arrays.

**Step 4. Compute the pdf for each distribution.**

Extract the four probability distribution objects for USA and compute the pdf for each distribution. As shown in Step 3, USA is in position 5 in each cell array.

```
WeiUSA = WeiByOrig{5};
NormUSA = NormByOrig{5};
LogUSA = LogByOrig{5};
KerUSA = KerByOrig{5};

x = 0:1:50;
pdf_Wei = pdf(WeiUSA,x);
pdf_Norm = pdf(NormUSA,x);
pdf_Log = pdf(LogUSA,x);
pdf_Ker = pdf(KerUSA,x);
```

**Step 5. Plot pdf the for each distribution.**

Plot the pdf for each distribution fit to the USA data, superimposed on a histogram of the sample data. Scale the density by the histogram area for easier display.

```
% Create a histogram of the USA sample data
data = MPG(Origin2=='USA');
figure;
[n,y] = hist(data,10);
b = bar(y,n,'hist');
set(b,'FaceColor',[1,0.8,0]);

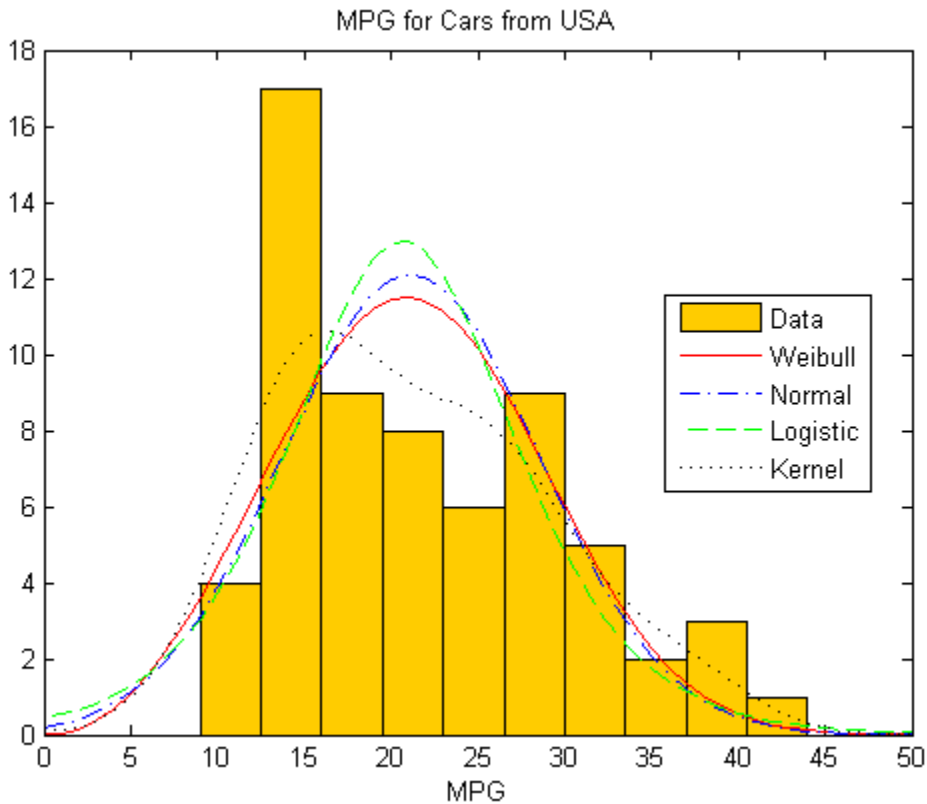
% Scale the density by the histogram area for easier display
area = sum(n)*(y(2)-y(1));
time = 0:50;
pdfWei = pdf(WeiUSA,time);
pdfNorm = pdf(NormUSA,time);
pdfLog = pdf(LogUSA,time);
pdfKer = pdf(KerUSA,time);

% Plot the pdf of each fitted distribution
line(x,pdfWei*area,'LineStyle','-','Color','r');
hold on;
```

```

line(x,pdfNorm*area,'LineStyle','-.-','Color','b');
line(x,pdfLog*area,'LineStyle','--','Color','g');
line(x,pdfKer*area,'LineStyle',':','Color','k');
l = legend('Data','Weibull','Normal','Logistic','Kernel');
set(l,'Location','Best');
title('MPG for Cars from USA');
xlabel('MPG');
hold off;

```



Superimposing the pdf plots over a histogram of the sample data provides a visual comparison of how well each type of distribution fits the data. Only the nonparametric kernel distribution `KerUSA` comes close to revealing the two modes in the original data.

**Step 6. Further group USA data by year.**

To investigate the two modes revealed in Step 5, group the MPG data by both country of origin (`Origin`) and model year (`Model_Year`), and use `fitdist` to fit kernel distributions to each group.

```
[KerByYearOrig,Names] = fitdist(MPG,'Kernel','By',{Origin Model_Year});
```

Each unique combination of origin and model year now has a kernel distribution object associated with it.

**Step 7. Compute the pdf for each group.**

Extract the three probability distributions for each USA model year, which are in positions 12, 13, and 14 in the cell array `KerByYearOrig`. Compute each pdf.

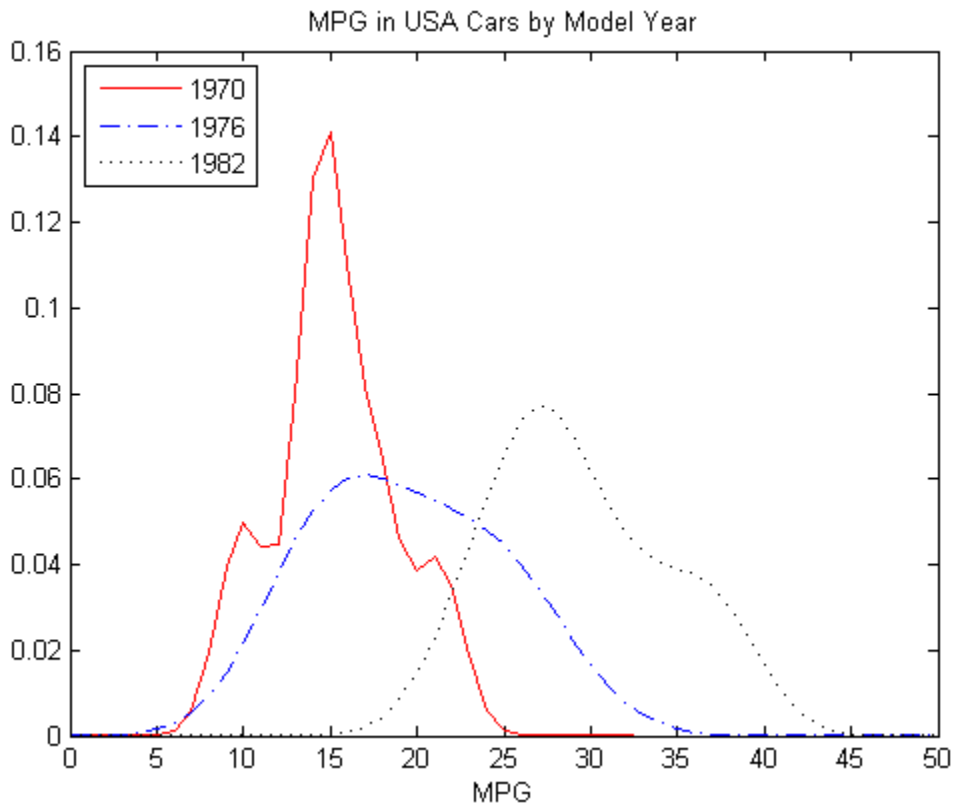
```
USA70 = KerByYearOrig{12};
USA76 = KerByYearOrig{13};
USA82 = KerByYearOrig{14};
```

```
pdf70 = pdf(USA70,x);
pdf76 = pdf(USA76,x);
pdf82 = pdf(USA82,x);
```

**Step 8. Plot pdf for each group.**

Plot the pdf for each group on the same figure.

```
figure;
plot(x,pdf70,'r-');
hold on;
plot(x,pdf76,'b-.');
plot(x,pdf82,'k:');
legend({'1970','1976','1982'],'Location','NW');
title('MPG in USA Cars by Model Year');
xlabel('MPG');
hold off;
```



When further grouped by model year, the pdf plots reveal two distinct peaks in the MPG data for cars made in the USA — one for the model year 1970, and one for the model year 1982. This explains why the smooth curve produced by the kernel distribution for the combined USA miles per gallon data shows two peaks instead of one.

**See Also** `fitdist`

### Related Examples

- “Fit Probability Distribution Objects to Grouped Data” on page 5-89

### Concepts

- “Grouping Variables” on page 2-51

## Fit Probability Distribution Objects to Grouped Data

This example shows how to fit probability distribution objects to grouped sample data, and create a plot to visually compare the pdf of each group.

### Step 1. Load sample data.

Load the sample data.

```
load carsmall;
```

The data contains miles per gallon (MPG) measurements for different makes and models of cars, grouped by country of origin (`Origin`), model year (`Model_Year`), and other vehicle characteristics.

### Step 2. Create a nominal array.

Transform `Origin` into a nominal array and remove the Italian car from the sample data. Since there is only one Italian car, `fitdist` cannot fit a distribution to that group. Removing the Italian car from the sample data prevents `fitdist` from returning an error.

```
Origin = nominal(Origin);
MPG2 = MPG(Origin~='Italy');
Origin2 = Origin(Origin~='Italy');
Origin2 = droplevels(Origin2,'Italy');
```

### Step 3. Fit kernel distributions to each group.

Use `fitdist` to fit kernel distributions to each country of origin group in the MPG data.

```
[KerByOrig,Country] = fitdist(MPG2,'Kernel','by',Origin2)
```

```
KerByOrig =
```

```
Column 1
```

```
[1x1 prob.KernelDistribution]
```

```
Column 2
```

```
[1x1 prob.KernelDistribution]

Column 3

[1x1 prob.KernelDistribution]

Column 4

[1x1 prob.KernelDistribution]

Column 5

[1x1 prob.KernelDistribution]

Country =

    'France'
    'Germany'
    'Japan'
    'Sweden'
    'USA'
```

The cell array `KerByOrig` contains five kernel distribution objects, one for each country represented in the sample data. Each object contains properties that hold information about the data, the distribution, and the parameters. The array `Country` lists the country of origin for each group in the same order as the distribution objects are stored in `KerByOrig`.

#### **Step 4. Compute the pdf for each group.**

Extract the probability distribution objects for Germany, Japan, and USA. Use the positions of each country in `KerByOrig` shown in Step 3, which indicates that Germany is the second country, Japan is the third country, and USA is the fifth country. Compute the pdf for each group.

```
Germany = KerByOrig{2};
Japan = KerByOrig{3};
USA = KerByOrig{5};
```



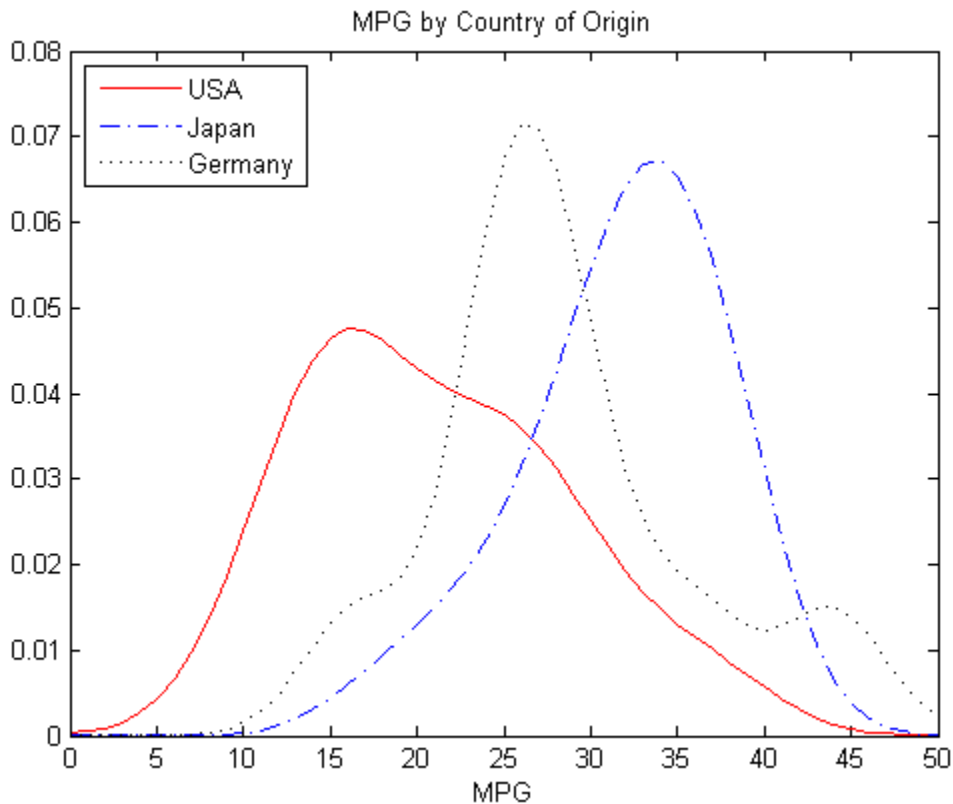
```
x = 0:1:50;

USA_pdf = pdf(USA,x);
Japan_pdf = pdf(Japan,x);
Germany_pdf = pdf(Germany,x);
```

**Step 5. Plot the pdf for each group.**

Plot the pdf for each group on the same figure.

```
figure;
plot(x,USA_pdf,'r-');
hold on;
plot(x,Japan_pdf,'b-.');
plot(x,Germany_pdf,'k:');
legend({'USA','Japan','Germany'},'Location','NW');
title('MPG by Country of Origin');
xlabel('MPG');
```



The resulting plot shows how miles per gallon (MPG) performance differs by country of origin (Origin). Using this data, the USA has the widest distribution, and its peak is at the lowest MPG value of the three origins. Japan has the most regular distribution with a slightly heavier left tail, and its peak is at the highest MPG value of the three origins. The peak for Germany is between the USA and Japan, and the second bump near 44 miles per gallon suggests that there might be multiple modes in the data.

### See Also

pdf | fitdist

### Related Examples

- “Fit Kernel Distributions to Grouped Data Using `ksdensity`” on page 5-23

## **Concepts**

- “Kernel Distribution” on page B-67
- “Grouping Variables” on page 2-51

## Multinomial Probability Distribution Objects

This example shows how to generate random numbers, compute and plot the pdf, and compute descriptive statistics of a multinomial distribution using probability distribution objects.

### Step 1. Define the distribution parameters.

Create a vector `p` containing the probability of each outcome. Outcome 1 has a probability of 1/2, outcome 2 has a probability of 1/3, and outcome 3 has a probability of 1/6. The number of trials `n` in each experiment is 5, and the number of repetitions `reps` of the experiment is 8.

```
p = [1/2 1/3 1/6];  
n = 5;  
reps = 8;
```

### Step 2. Create a multinomial probability distribution object.

Create a multinomial probability distribution object using the specified value `p` for the `Probabilities` parameter.

```
pd = makedist('Multinomial','Probabilities',p)
```

```
pd =  
  
    MultinomialDistribution  
  
    Probabilities:  
    0.5000    0.3333    0.1667
```

### Step 3. Generate one random number.

Generate one random number from the multinomial distribution, which is the outcome of a single trial.

```
rng('default') % For reproducibility  
r = random(pd)  
  
r =
```

2

This trial resulted in outcome 2.

**Step 4. Generate a matrix of random numbers.**

You can also generate a matrix of random numbers from the multinomial distribution, which reports the results of multiple experiments that each contain multiple trials. Generate a matrix that contains the outcomes of an experiment with  $n = 5$  trials and  $\text{reps} = 8$  repetitions.

```
r = random(pd, reps, n)
```

```
r =
```

```

3     3     3     2     1
1     1     2     2     1
3     3     3     1     2
2     3     2     2     2
1     1     1     1     1
1     2     3     2     3
2     1     3     1     1
3     1     2     1     1

```

Each element in the resulting matrix is the outcome of one trial. The columns correspond to the five trials in each experiment, and the rows correspond to the eight experiments. For example, in the first experiment (corresponding to the first row), one of the five trials resulted in outcome 1, one of the five trials resulted in outcome 2, and three of the five trials resulted in outcome 3.

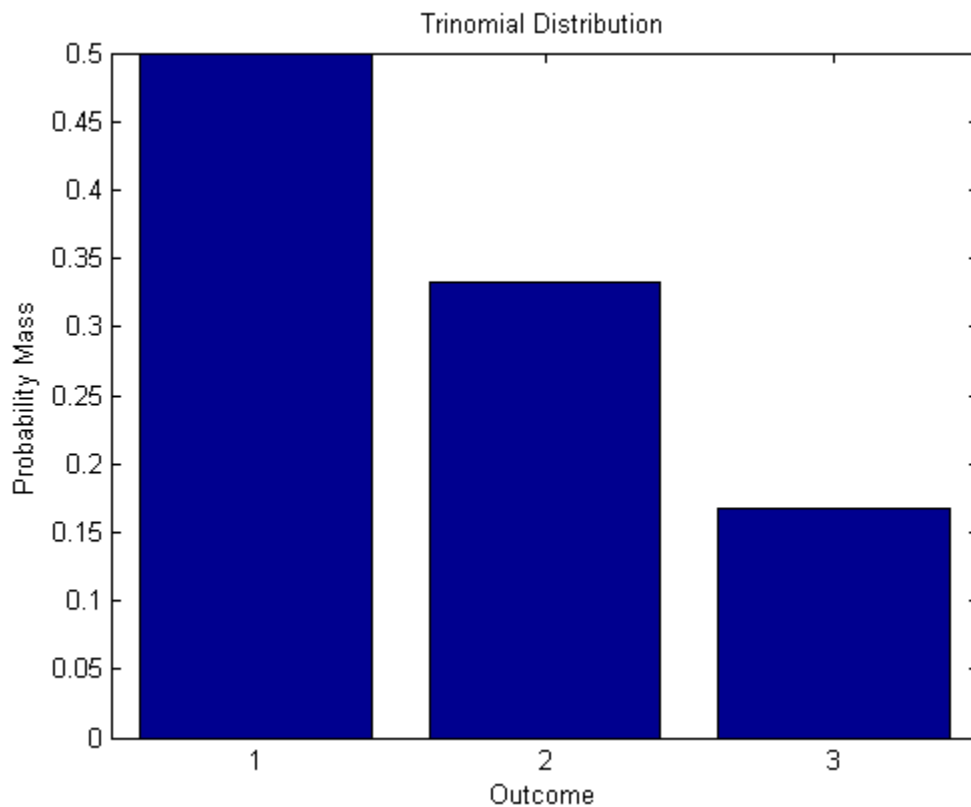
**Step 5. Compute and plot the pdf.**

Compute the pdf of the distribution.

```

x = 1:3;
y = pdf(pd, x);
bar(x, y);
xlabel('Outcome');
ylabel('Probability Mass');
title('Trinomial Distribution');

```



The plot shows the probability mass for each  $k$  possible outcome. For this distribution, the pdf value for any  $x$  other than 1, 2, or 3 is 0.

**Step 6. Compute descriptive statistics.**

Compute the mean, median, and standard deviation of the distribution.

```
m = mean(pd)
med = median(pd)
s = std(pd)
```

```
m =
```

1.6667

med =

1

s =

0.7454

## Multinomial Probability Distribution Functions

This example shows how to generate random numbers and compute and plot the pdf of a multinomial distribution using probability distribution functions.

### Step 1. Define the distribution parameters.

Create a vector  $p$  containing the probability of each outcome. Outcome 1 has a probability of  $1/2$ , outcome 2 has a probability of  $1/3$ , and outcome 3 has a probability of  $1/6$ . The number of trials in each experiment  $n$  is 5, and the number of repetitions of the experiment  $reps$  is 8.

```
p = [1/2 1/3 1/6];  
n = 5;  
reps = 8;
```

### Step 2. Generate one random number.

Generate one random number from the multinomial distribution, which is the outcome of a single trial.

```
rng('default') % For reproducibility  
r = mnrnd(1,p,1)
```

```
r =  
  
    0     1     0
```

The returned vector  $r$  contains three elements, which show the counts for each possible outcome. This single trial resulted in outcome 2.

### Step 3. Generate a matrix of random numbers.

You can also generate a matrix of random numbers from the multinomial distribution, which reports the results of multiple experiments that each contain multiple trials. Generate a matrix that contains the outcomes of an experiment with  $n = 5$  trials and  $reps = 8$  repetitions.

```
r = mnrnd(n,p,reps)  
  
r =
```



1	1	3
3	2	0
1	1	3
0	4	1
5	0	0
1	2	2
3	1	1
3	1	1

Each row in the resulting matrix contains counts for each of the  $k$  multinomial bins. For example, in the first experiment (corresponding to the first row), one of the five trials resulted in outcome 1, one of the five trials resulted in outcome 2, and three of the five trials resulted in outcome 3.

#### Step 4. Compute the pdf.

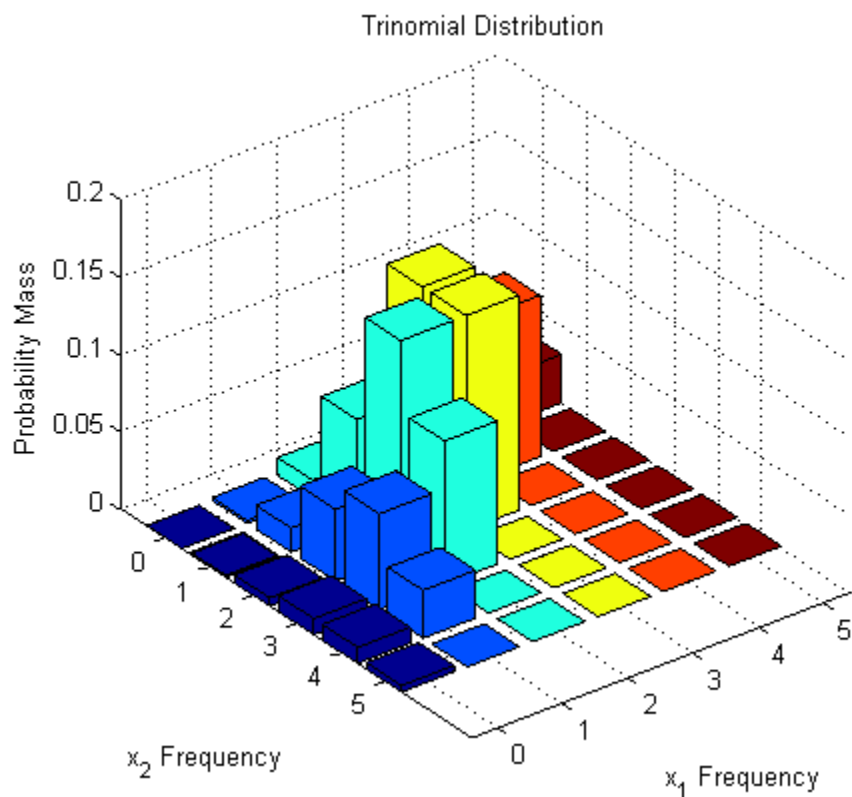
Since multinomial functions work with bin counts, create a multidimensional array of all possible outcome combinations, and compute the pdf using `mnpdf`.

```
count1 = 1:n;
count2 = 1:n;
[x1,x2] = meshgrid(count1,count2);
x3 = n-(x1+x2);
y = mnpdf([x1(:),x2(:),x3(:)], repmat(p, (n)^2, 1));
```

#### Step 5. Plot the pdf.

Create a 3-D bar graph to visualize the pdf for each combination of outcome frequencies.

```
figure;
y = reshape(y,n,n);
bar3(y);
set(gca, 'XTickLabel', 1:n);
set(gca, 'YTickLabel', 1:n);
xlabel('x_1 Frequency');
ylabel('x_2 Frequency');
zlabel('Probability Mass');
```



The plot shows the probability mass for each possible combination of outcomes. It does not show  $x_3$ , which is determined by the constraint  $x_1 + x_2 + x_3 = n$ .

## Generate Random Numbers Using Uniform Distribution Inversion

This example shows how to generate random numbers using the uniform distribution inversion method. This is useful for distributions when it is possible to compute the inverse cumulative distribution function, but there is no support for sampling from the distribution directly.

### Step 1. Generate random numbers from the standard uniform distribution.

Use `rand` to generate 1000 random numbers from the uniform distribution on the interval (0,1).

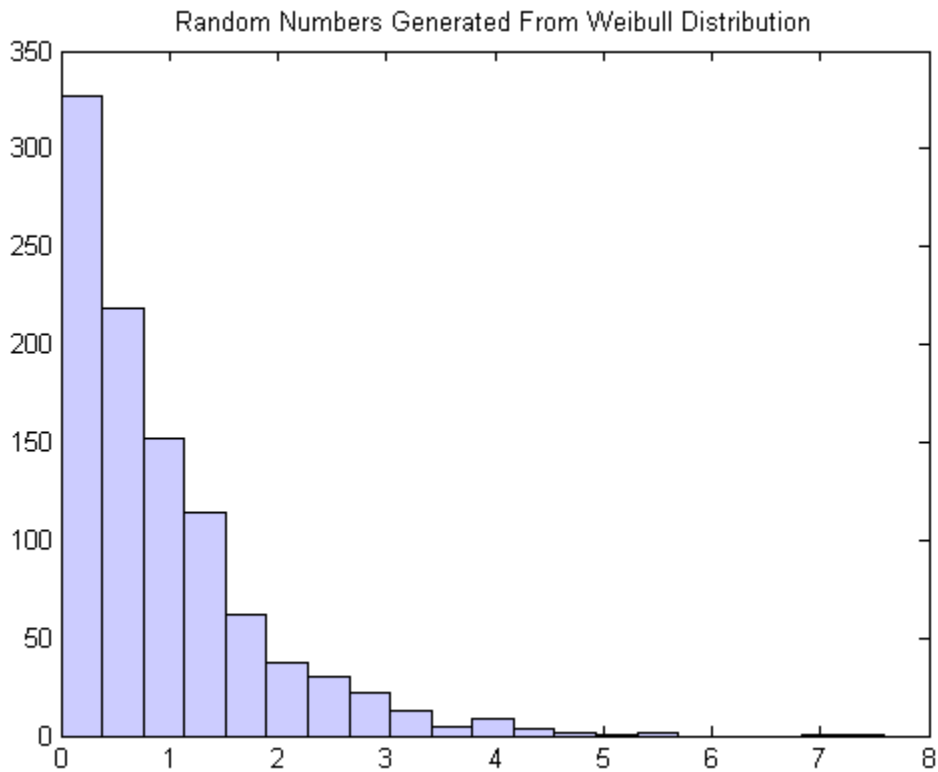
```
rng('default') % For reproducibility
u = rand(1000,1);
```

The inversion method relies on the principle that continuous cumulative distribution functions (cdfs) range uniformly over the open interval (0,1). If  $u$  is a uniform random number on (0,1), then  $x = F^{-1}(u)$  generates a random number  $x$  from any continuous distribution with the specified cdf  $F$ .

### Step 2. Generate random numbers from the Weibull distribution.

Use the inverse cumulative distribution function to generate the random numbers from a Weibull distribution with parameters  $A = 1$  and  $B = 1$  that correspond to the probabilities in  $u$ . Plot the results.

```
figure;
x = wblinv(u,1,1);
hist(x,20);
```

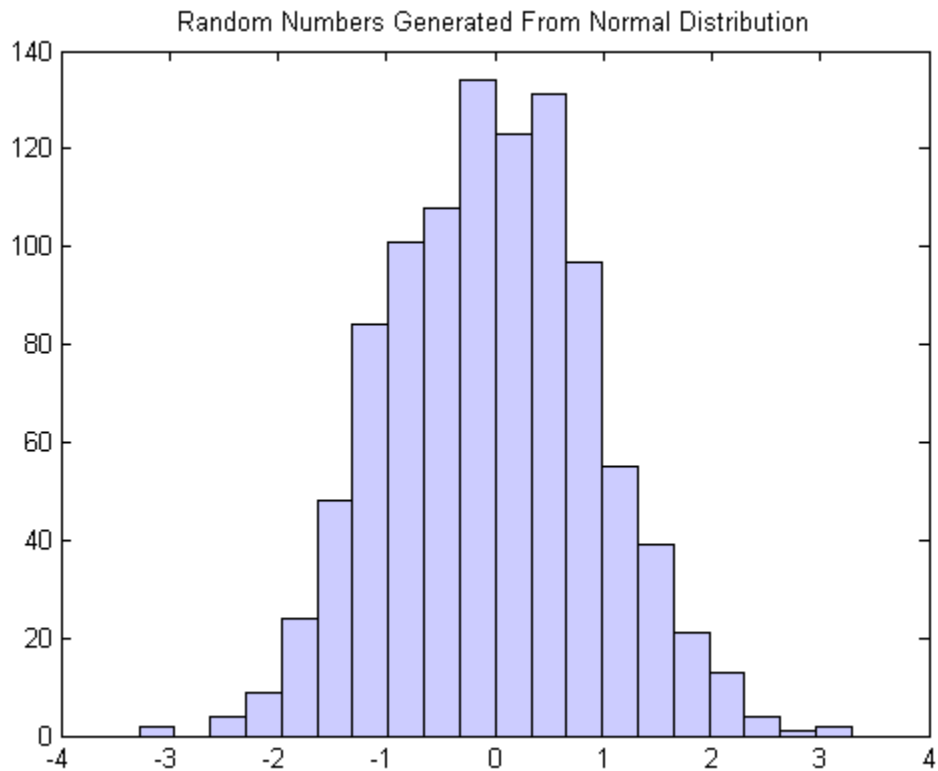


The histogram shows that the random numbers generated using the Weibull inverse cdf function `wblinv` have a Weibull distribution.

**Step 3. Generate random numbers from the standard normal distribution.**

The same values in `u` can generate random numbers from any distribution, for example the standard normal, by following the same procedure using the inverse cdf of the desired distribution.

```
figure;
x_norm = norminv(u,1,0);
hist = (x_norm,20);
```



The histogram shows that, by using the standard normal inverse cdf `norminv`, the random numbers generated from `u` now have a standard normal distribution.

## See Also

`wblinv` | `norminv` | `rand` | `hist`

## Concepts

- “Uniform Distribution (Continuous)” on page B-133
- “Weibull Distribution” on page B-141
- “Normal Distribution” on page B-110

## Use $t$ Location-Scale to Represent Cauchy Distribution

This example shows how to use the  $t$  location-scale probability distribution object to work with a Cauchy distribution with nonstandard parameter values.

### Step 1. Create a probability distribution object.

Create a  $t$  location-scale probability distribution object with degrees of freedom `nu = 1`. Specify `mu = 3` to set the location parameter equal to 3, and `sigma = 1` to set the scale parameter equal to 1.

```
pd = makedist('tLocationScale','mu',3,'sigma',1,'nu',1)
```

```
pd =
```

```
tLocationScaleDistribution  
  
t Location-Scale distribution  
    mu = 3  
    sigma = 1  
    nu = 1
```

### Step 2. Compute descriptive statistics.

Use object functions to compute descriptive statistics for the Cauchy distribution.

```
med = median(pd)  
r = iqr(pd)  
m = mean(pd)  
s = std(pd)
```

```
med =
```

```
3
```

```
r =
```

```
2
```

```
m =
```

```
NaN
```

```
s =
```

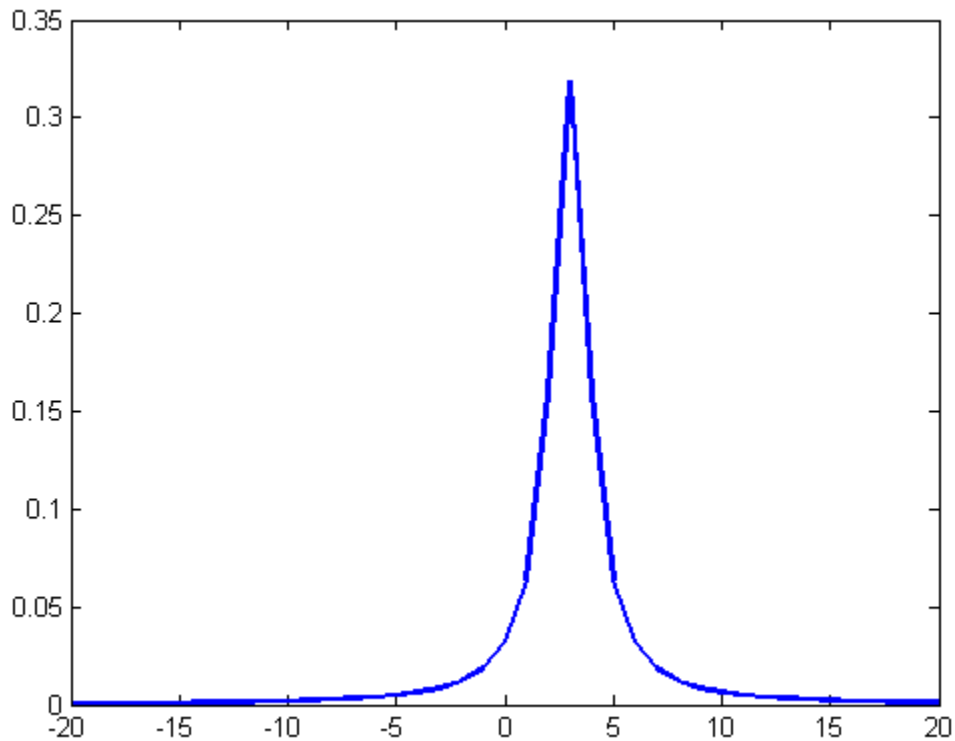
```
Inf
```

The median of the Cauchy distribution is equal to its location parameter, and the interquartile range is equal to two times its scale parameter. Its mean and standard deviation are undefined.

**Step 3. Compute and plot the pdf.**

Compute and plot the pdf of the Cauchy distribution.

```
figure;  
x = -20:1:20;  
y = pdf(pd,x);  
plot(x,y,'LineWidth',2);
```



The peak of the pdf is centered at the location parameter  $\mu = 3$ .

**Step 4. Generate a vector of Cauchy random numbers.**

Generate a column vector containing 10 random numbers from the Cauchy distribution using the random function for the  $t$  location-scale probability distribution object.

```
rng('default'); % For reproducibility  
r = random(pd,10,1)
```

```
r =
```



3.2678  
 4.6547  
 2.0604  
 4.7322  
 3.1810  
 1.6649  
 1.8471  
 4.2466  
 5.4647  
 8.8874

**Step 5. Generate a matrix of Cauchy random numbers.**

Generate a 5-by-5 matrix of Cauchy random numbers.

```
r = random(pd,5,5)
```

```
r =
```

```
2.2867    2.9692   -1.7003    5.5949    1.9806
2.7421    2.7180    3.2210    2.4233    3.1394
3.5966    3.9806    1.0182    6.4180    5.1367
5.4791   15.6472    0.7558    2.8908    5.9031
1.6863    4.0985    2.9934   13.9506    4.8792
```

**See Also** `makedist`

**Related Examples**

- “Use Student’s  $t$  to Generate Standard Cauchy Random Numbers” on page 5-108

**Concepts**

- “ $t$  Location-Scale Distribution” on page B-126

## Use Student's $t$ to Generate Standard Cauchy Random Numbers

This example shows how to use the Student's  $t$  distribution to generate random numbers from a standard Cauchy distribution.

### Step 1. Generate a vector of random numbers.

Generate a column vector containing 10 random numbers from a standard Cauchy distribution, which has a location parameter  $\mu = 0$  and scale parameter  $\sigma = 1$ . Use `trnd` with degrees of freedom  $V = 1$ .

```
rng('default'); % For reproducibility
r = trnd(1,10,1)
```

```
r =
    0.2678
    1.6547
   -0.9396
    1.7322
    0.1810
   -1.3351
   -1.1529
    1.2466
    2.4647
    5.8874
```

### Step 2. Generate a matrix of random numbers.

Generate a 5-by-5 matrix of random numbers from a standard Cauchy distribution.

```
r = trnd(1,5,5)
```

```
r =
   -0.7133   -0.0308   -4.7003    2.5949   -1.0194
   -0.2579   -0.2820    0.2210   -0.5767    0.1394
    0.5966    0.9806   -1.9818    3.4180    2.1367
```

2.4791	12.6472	-2.2442	-0.1092	2.9031
-1.3137	1.0985	-0.0066	10.9506	1.8792

**See Also** `trnd`

**Related  
Examples**

- “Use  $t$  Location-Scale to Represent Cauchy Distribution ” on page 5-104

**Concepts**

- “Student’s  $t$  Distribution” on page B-124

## Probability Distributions Used for Multivariate Modeling

### In this section...

“Gaussian Mixture Models” on page 5-110

“Copulas: Generate Correlated Samples” on page 5-120

### Gaussian Mixture Models

- “Creating Gaussian Mixture Models” on page 5-110
- “Simulating Gaussian Mixtures” on page 5-118

Gaussian mixture models are formed by combining multivariate normal density components. In Statistics Toolbox software, use the `gmdistribution` class to fit data using an expectation maximization (EM) algorithm, which assigns posterior probabilities to each component density with respect to each observation. The fitting method uses an iterative algorithm that converges to a local optimum. Clustering using Gaussian mixture models is sometimes considered a soft clustering method. The posterior probabilities for each point indicate that each data point has some probability of belonging to each cluster.

For more information on clustering with Gaussian mixture models, see “Gaussian Mixture Models” on page 14-30. This section describes their creation.

### Creating Gaussian Mixture Models

- “Specifying a Model” on page 5-110
- “Fitting a Model to Data” on page 5-114

**Specifying a Model.** Use the `gmdistribution` constructor to create Gaussian mixture models with specified means, covariances, and mixture proportions.

First, define the means, covariances, and mixture proportions.

```
MU = [1 2; -3 -5]; % Means
```

```
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]); % Covariances
p = ones(1,2)/2; % Mixing proportions
```

Then, create an object of the `gmdistribution` class defining a two-component mixture of bivariate Gaussian distributions:

```
obj = gmdistribution(MU,SIGMA,p);
```

Display properties of the object with the MATLAB function `fieldnames`:

```
properties = fieldnames(obj)
```

```
properties =
    'NDimensions'
    'DistName'
    'NComponents'
    'PComponents'
    'mu'
    'Sigma'
    'NlogL'
    'AIC'
    'BIC'
    'Converged'
    'Iters'
    'SharedCov'
    'CovType'
    'RegV'
```

The `gmdistribution` reference page describes these properties. To access the value of a property, use dot indexing. For example, access the dimensions of the object.

```
dimension = obj.NDimensions
```

```
dimension =
```

2

Access the distribution name.

```
name = obj.DistName
```

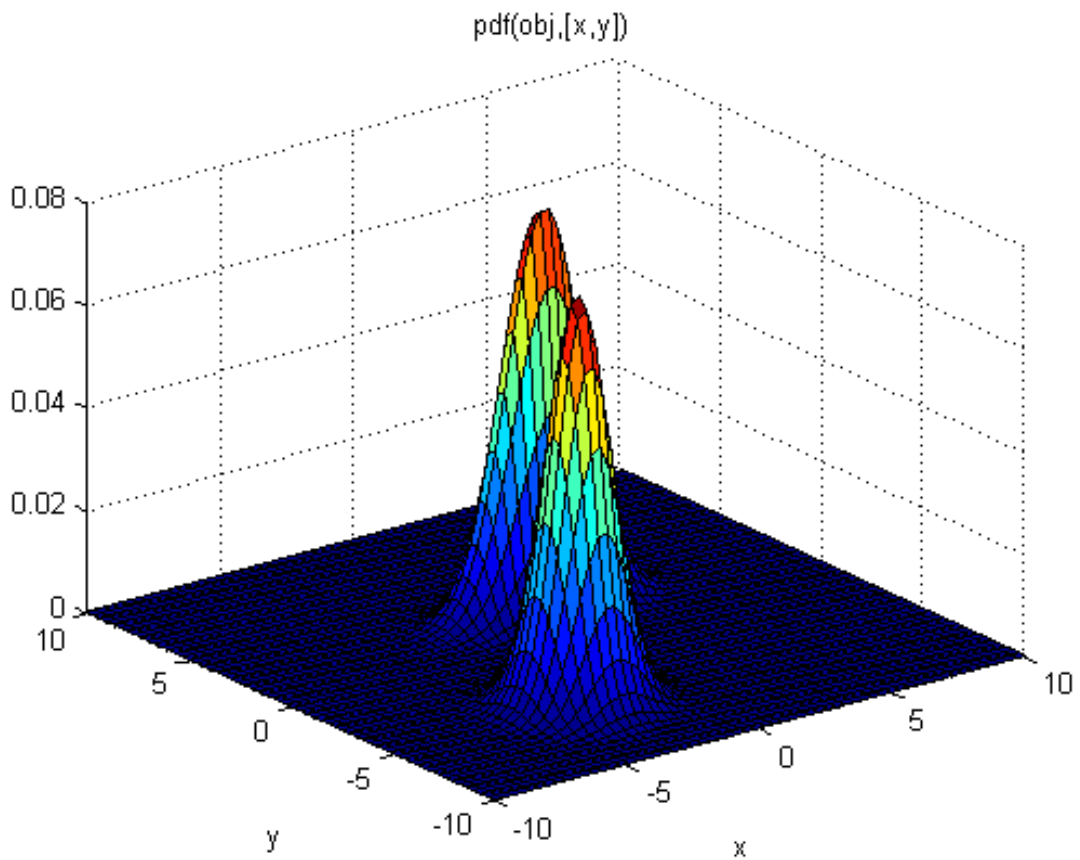
```
name =
```

```
gaussian mixture distribution
```

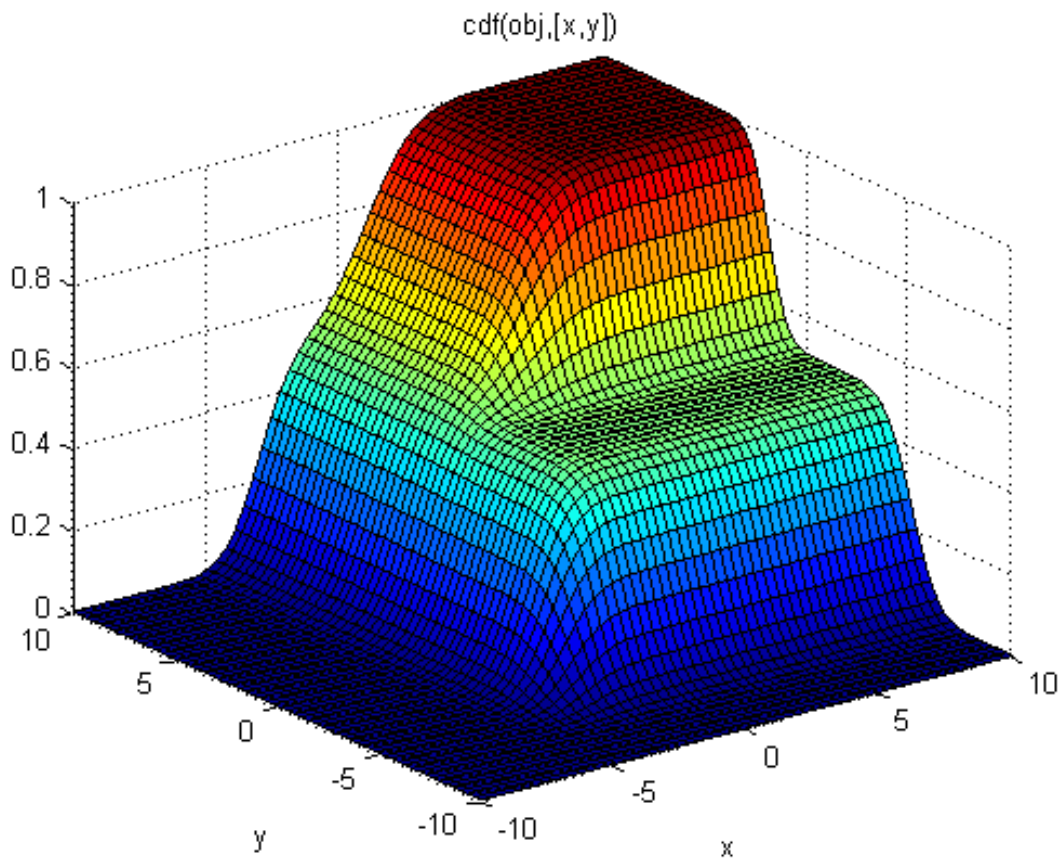
Use the methods `pdf` and `cdf` to compute values and visualize the object:

```
figure
```

```
ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
```



```
figure  
ezsurf(@ (x,y) cdf(obj,[x y]), [-10 10], [-10 10])
```



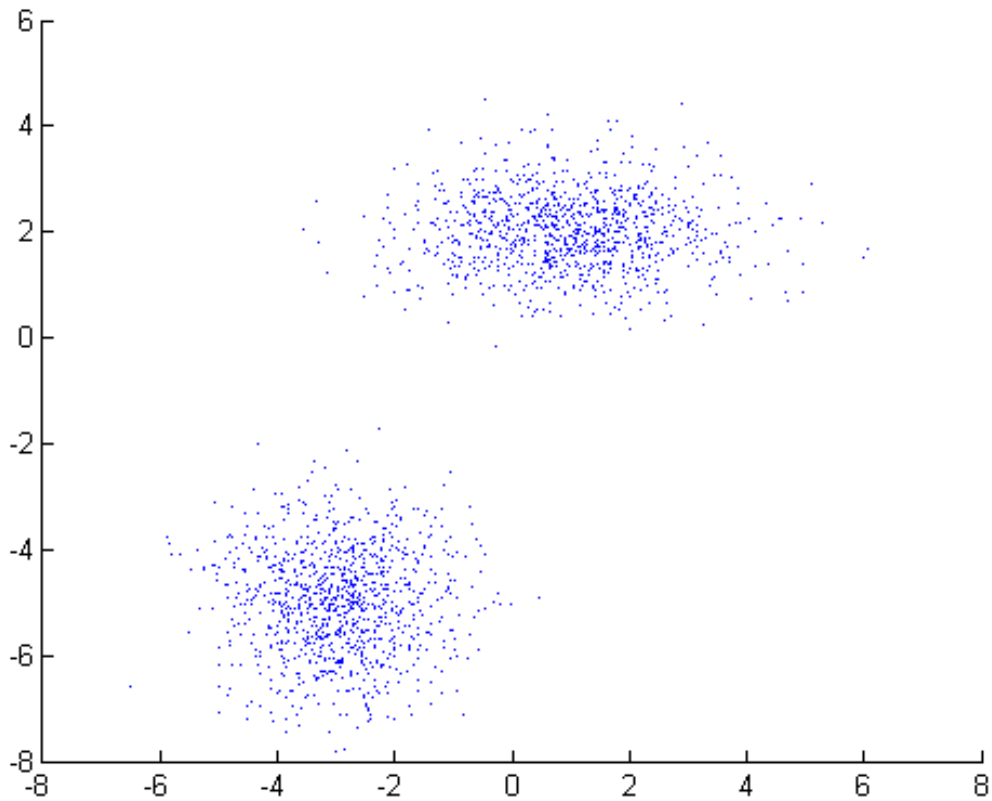
**Fitting a Model to Data.** You can also create Gaussian mixture models by fitting a parametric model with a specified number of components to data. The `fit` method of the `gmdistribution` class uses the syntax `obj = gmdistribution.fit(X,k)`, where `X` is a data matrix and `k` is the specified number of components. Choosing a suitable number of components `k` is essential for creating a useful model of the data—too few components fails to model the data accurately; too many components leads to an over-fit model with singular covariance matrices.

The following example illustrates this approach.



First, create some data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function:

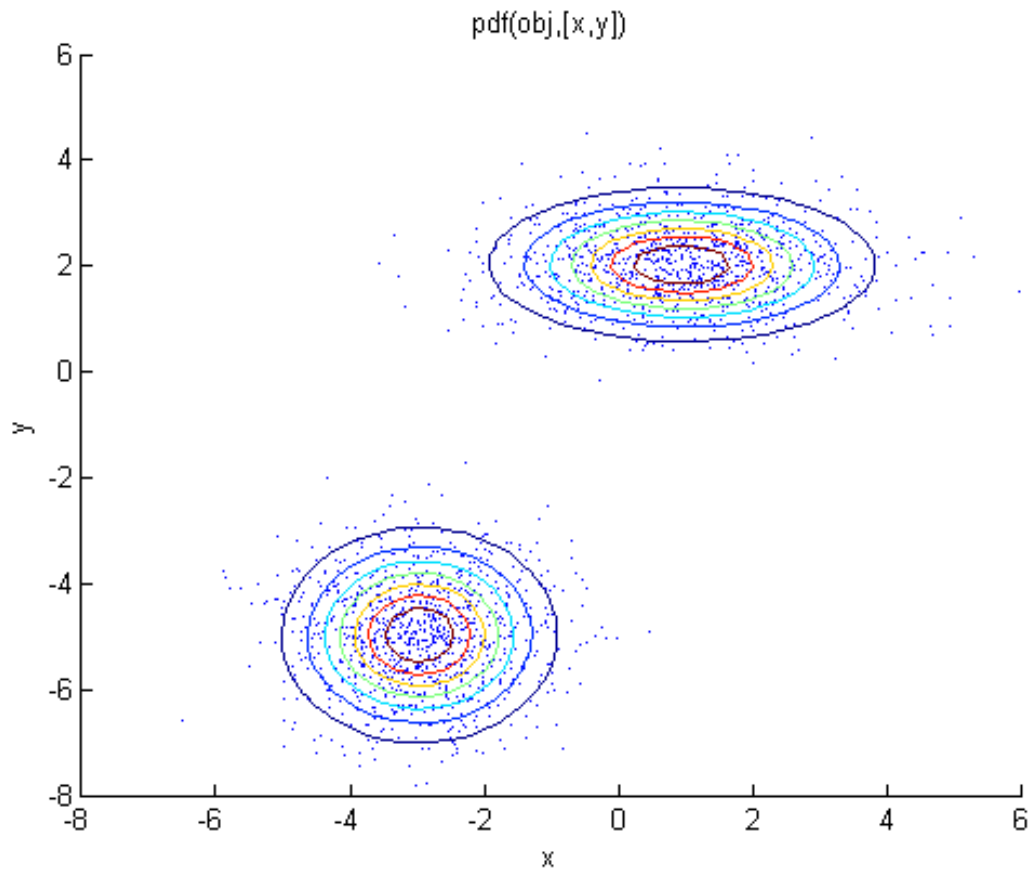
```
MU1 = [1 2];  
SIGMA1 = [2 0; 0 .5];  
MU2 = [-3 -5];  
SIGMA2 = [1 0; 0 1];  
X = [mvnrnd(MU1,SIGMA1,1000);  
mvnrnd(MU2,SIGMA2,1000)];  
figure  
scatter(X(:,1),X(:,2),10, '.')
```



Next, fit a two-component Gaussian mixture model:

```
options = statset('Display','final');  
obj = gmdistribution.fit(X,2,'Options',options);  
hold on  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);  
hold off
```

18 iterations, log-likelihood = -7058.35



Among the properties of the fit are the parameter estimates.

Display the estimates for mu, sigma, and mixture proportions

```
ComponentMeans = obj.mu  
ComponentCovariances = obj.Sigma  
MixtureProportions = obj.PComponents
```

```
ComponentMeans =
```

```
    -2.9617    -4.9727  
     0.9539     2.0261
```

```
ComponentCovariances(:, :, 1) =
```

```
    1.0100    0.0059  
    0.0059    0.9897
```

```
ComponentCovariances(:, :, 2) =
```

```
    1.9939   -0.0092  
   -0.0092    0.4981
```

```
MixtureProportions =
```

```
    0.5000    0.5000
```

The two-component model minimizes the Akaike information:

```
AIC = zeros(1,4);  
obj = cell(1,4);  
for k = 1:4  
    obj{k} = gmdistribution.fit(X,k);  
    AIC(k) = obj{k}.AIC;  
end  
  
[minAIC,numComponents] = min(AIC);
```

```
numComponents
```

```
numComponents =
```

```
2
```

Display the model.

```
model = obj{2}
```

```
model =
```

```
Gaussian mixture distribution with 2 components in 2 dimensions
```

```
Component 1:
```

```
Mixing proportion: 0.500000
```

```
Mean: -2.9617 -4.9727
```

```
Component 2:
```

```
Mixing proportion: 0.500000
```

```
Mean: 0.9539 2.0261
```

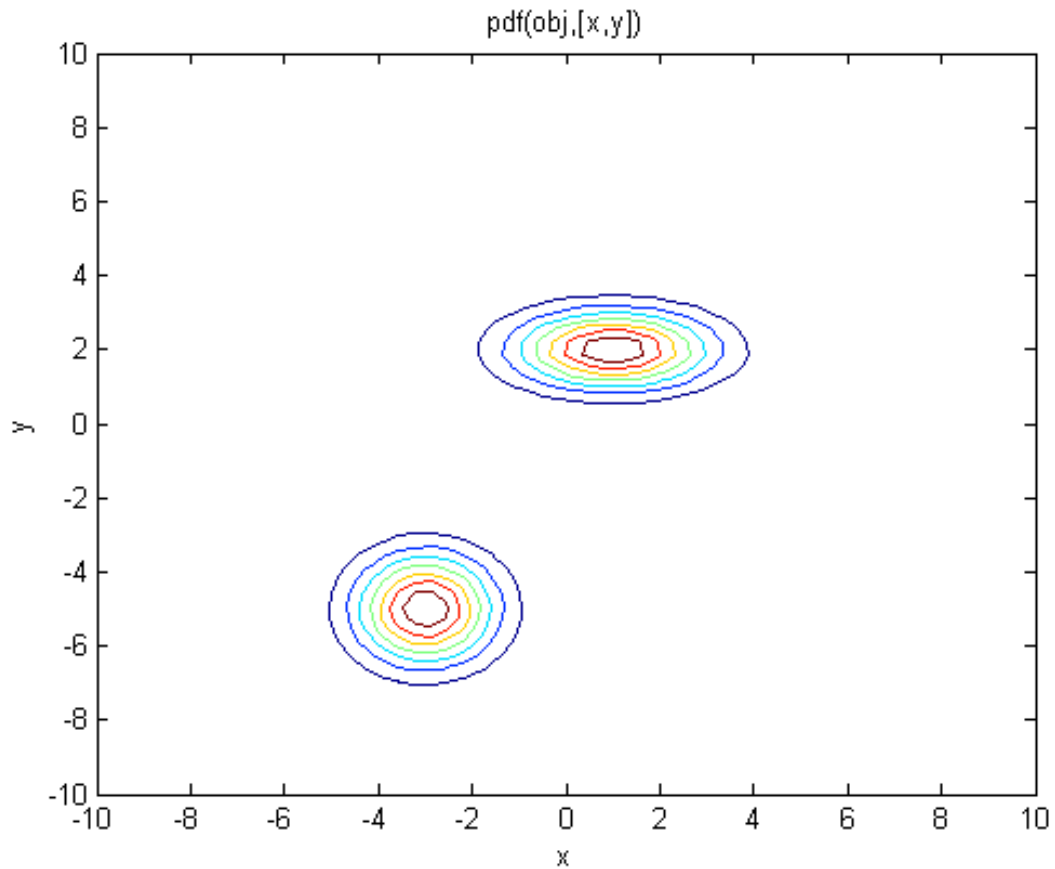
Both the Akaike and Bayes information are negative log-likelihoods for the data with penalty terms for the number of estimated parameters. You can use them to determine an appropriate number of components for a model when the number of components is unspecified.

### **Simulating Gaussian Mixtures**

Use the method `random` of the `gmdistribution` class to generate random data from a Gaussian mixture model created with `gmdistribution` or `fit`.

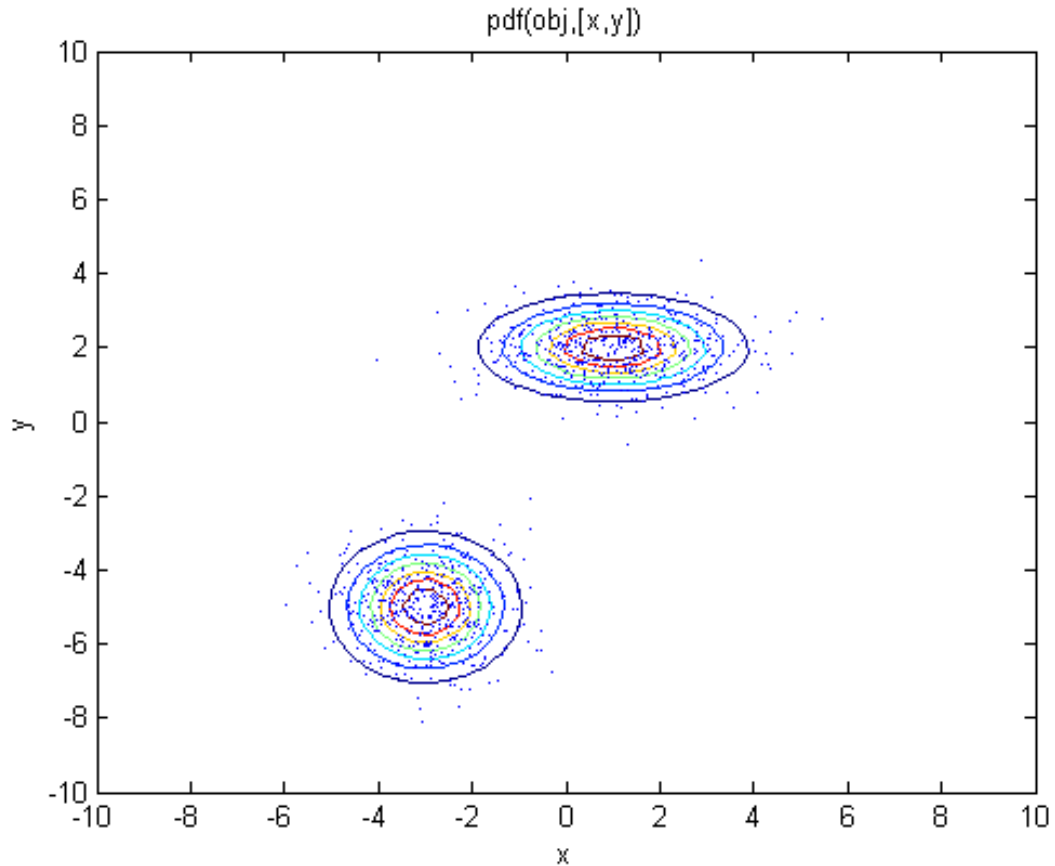
For example, the following specifies a `gmdistribution` object consisting of a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2; -3 -5];  
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);  
p = ones(1,2)/2;  
obj = gmdistribution(MU,SIGMA,p);  
  
figure  
ezcontour(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])  
hold on
```



Use random (gmdistribution) to generate 1000 random values:

```
Y = random(obj,1000);  
scatter(Y(:,1),Y(:,2),10,'.')
```



## Copulas: Generate Correlated Samples

- “Determining Dependence Between Simulation Inputs” on page 5-121
- “Constructing Dependent Bivariate Distributions” on page 5-125
- “Using Rank Correlation Coefficients” on page 5-130

- “Using Bivariate Copulas” on page 5-133
- “Higher Dimension Copulas” on page 5-143
- “Archimedean Copulas” on page 5-145
- “Simulating Dependent Multivariate Data Using Copulas” on page 5-147
- “Fitting Copulas to Data” on page 5-153

*Copulas* are functions that describe dependencies among variables, and provide a way to create distributions that model correlated multivariate data. Using a copula, you can construct a multivariate distribution by specifying marginal univariate distributions, and then choose a copula to provide a correlation structure between variables. Bivariate distributions, as well as distributions in higher dimensions, are possible.

### **Determining Dependence Between Simulation Inputs**

One of the design decisions for a Monte Carlo simulation is a choice of probability distributions for the random inputs. Selecting a distribution for each individual variable is often straightforward, but deciding what dependencies should exist between the inputs may not be. Ideally, input data to a simulation should reflect what you know about dependence among the real quantities you are modeling. However, there may be little or no information on which to base any dependence in the simulation. In such cases, it is useful to experiment with different possibilities in order to determine the model’s sensitivity.

It can be difficult to generate random inputs with dependence when they have distributions that are not from a standard multivariate distribution. Further, some of the standard multivariate distributions can model only limited types of dependence. It is always possible to make the inputs independent, and while that is a simple choice, it is not always sensible and can lead to the wrong conclusions.

For example, a Monte-Carlo simulation of financial risk could have two random inputs that represent different sources of insurance losses. You could model these inputs as lognormal random variables. A reasonable question to ask is how dependence between these two inputs affects the results of the simulation. Indeed, you might know from real data that the same random

conditions affect both sources; ignoring that in the simulation could lead to the wrong conclusions.

**Example: Generate and Exponentiate Normal Random Variables.**

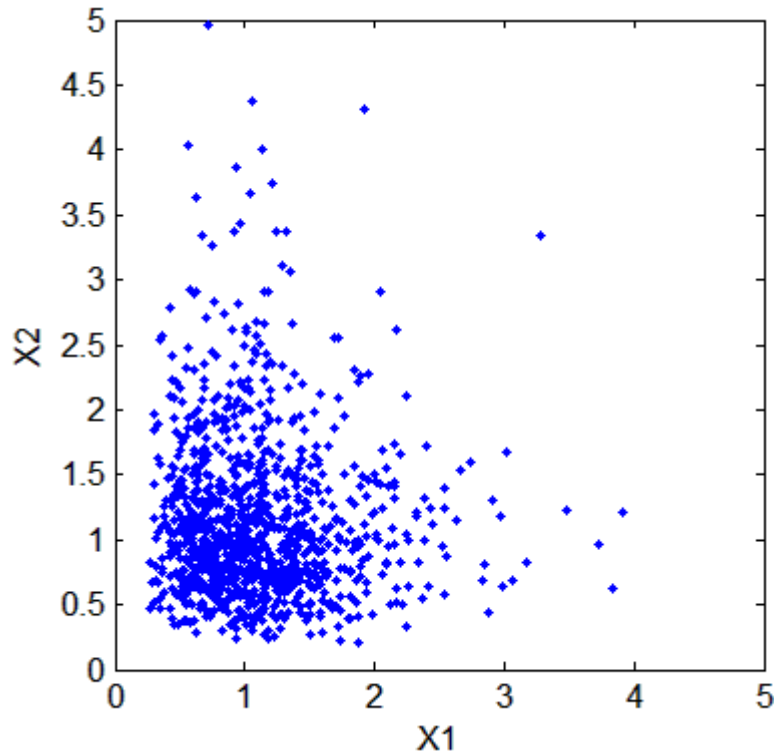
The `lognrnd` function simulates independent lognormal random variables. In the following example, the `mvnrnd` function generates  $n$  pairs of independent normal random variables, and then exponentiates them. Notice that the covariance matrix used here is diagonal:

```
n = 1000;

sigma = .5;
SigmaInd = sigma.^2 .* [1 0; 0 1]
SigmaInd =
    0.25    0
    0    0.25
ZInd = mvnrnd([0 0],SigmaInd,n);
XInd = exp(ZInd);

plot(XInd(:,1),XInd(:,2),'.')
axis([0 5 0 5])
axis equal
xlabel('X1')
ylabel('X2')
```





Dependent bivariate lognormal random variables are also easy to generate using a covariance matrix with nonzero off-diagonal terms:

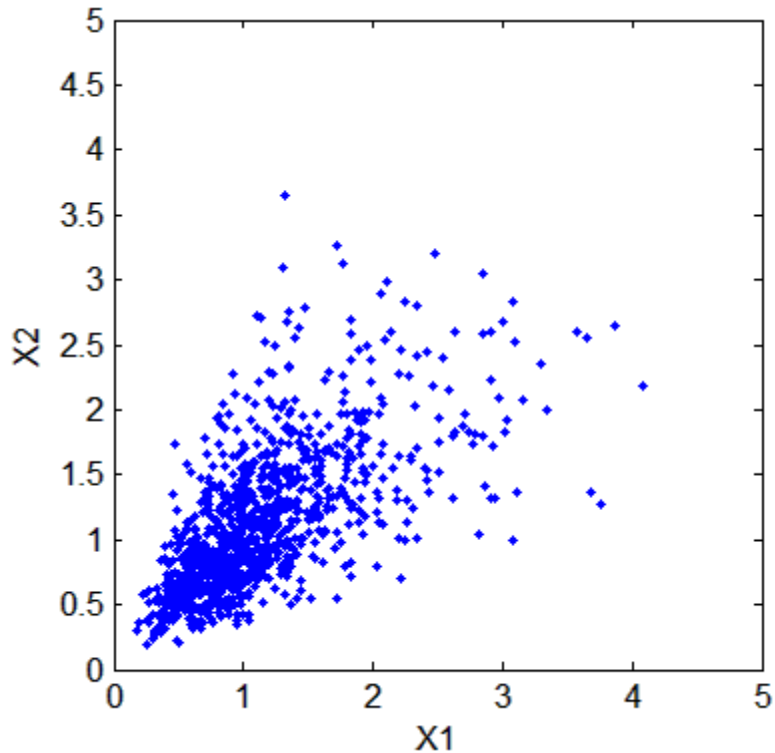
```
rho = .7;

SigmaDep = sigma.^2 .* [1 rho; rho 1]
SigmaDep =
    0.25    0.175
    0.175    0.25

ZDep = mvnrnd([0 0],SigmaDep,n);
XDep = exp(ZDep);
```

A second scatter plot demonstrates the difference between these two bivariate distributions:

```
plot(XDep(:,1),XDep(:,2),'.')
axis([0 5 0 5])
axis equal
xlabel('X1')
ylabel('X2')
```



It is clear that there is a tendency in the second data set for large values of  $X_1$  to be associated with large values of  $X_2$ , and similarly for small values. The correlation parameter,  $\rho$ , of the underlying bivariate normal determines this dependence. The conclusions drawn from the simulation could well depend on whether you generate  $X_1$  and  $X_2$  with dependence. The bivariate lognormal distribution is a simple solution in this case; it easily generalizes to higher dimensions in cases where the marginal distributions are different lognormals.

Other multivariate distributions also exist. For example, the multivariate  $t$  and the Dirichlet distributions simulate dependent  $t$  and beta random variables, respectively. But the list of simple multivariate distributions is not long, and they only apply in cases where the marginals are all in the same family (or even the exact same distributions). This can be a serious limitation in many situations.

### Constructing Dependent Bivariate Distributions

Although the construction discussed in the previous section creates a bivariate lognormal that is simple, it serves to illustrate a method that is more generally applicable.

- 1 Generate pairs of values from a bivariate normal distribution. There is statistical dependence between these two variables, and each has a normal marginal distribution.
- 2 Apply a transformation (the exponential function) separately to each variable, changing the marginal distributions into lognormals. The transformed variables still have a statistical dependence.

If a suitable transformation can be found, this method can be generalized to create dependent bivariate random vectors with other marginal distributions. In fact, a general method of constructing such a transformation does exist, although it is not as simple as exponentiation alone.

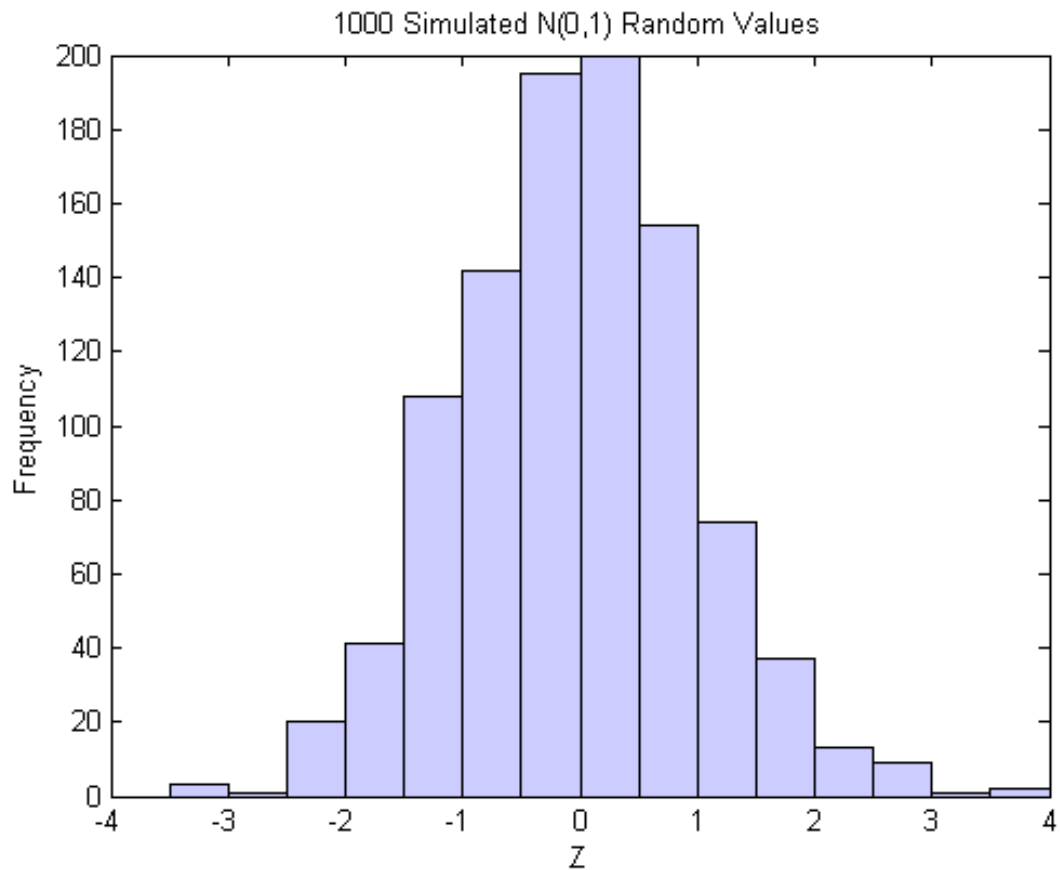
By definition, applying the normal cumulative distribution function (cdf), denoted here by  $\Phi$ , to a standard normal random variable results in a random variable that is uniform on the interval  $[0,1]$ . To see this, if  $Z$  has a standard normal distribution, then the cdf of  $U = \Phi(Z)$  is

$$\Pr\{U \leq u\} = \Pr\{\Phi(Z) \leq u\} = \Pr\{Z \leq \Phi^{-1}(u)\} = u$$

and that is the cdf of a  $\text{Unif}(0,1)$  random variable. Histograms of some simulated normal and transformed values demonstrate that fact:

```
n = 1000;
rng('default') % for reproducibility
z = normrnd(0,1,n,1); % generate standard normal data
```

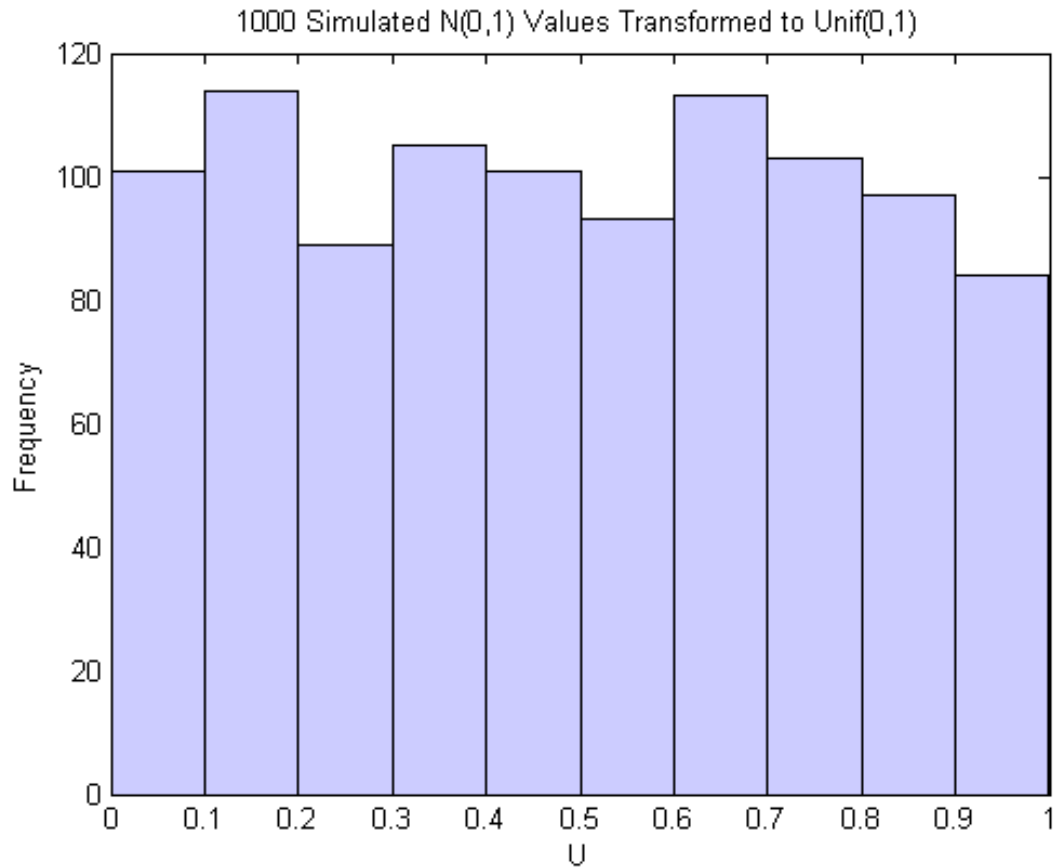
```
hist(z,-3.75:.5:3.75) % plot the histogram of data
xlim([-4 4])
title('1000 Simulated N(0,1) Random Values')
xlabel('Z')
ylabel('Frequency')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```



```
u = normcdf(z); % compute the cdf values of the sample data

figure
hist(u,.05:.1:.95) % plot the histogram of the cdf values
```

```
title('1000 Simulated N(0,1) Values Transformed to Unif(0,1)')
xlabel('U')
ylabel('Frequency')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```

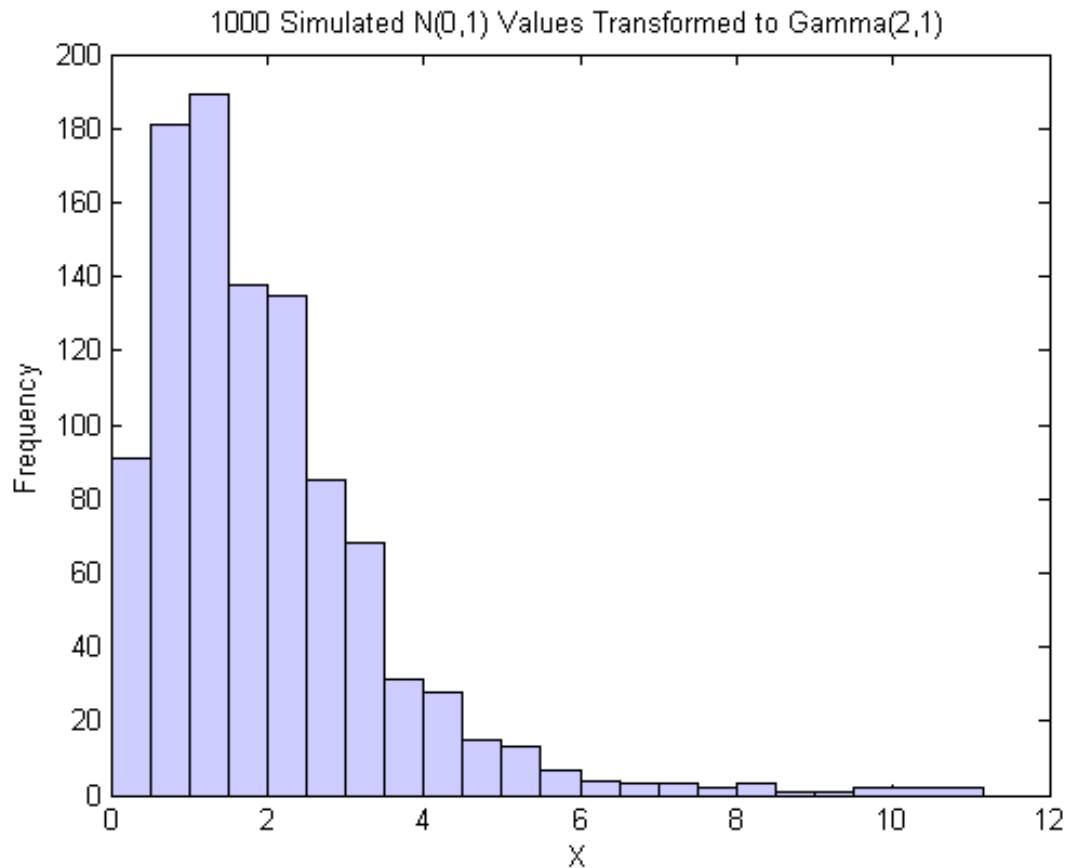


Borrowing from the theory of univariate random number generation, applying the inverse cdf of any distribution,  $F$ , to a  $\text{Unif}(0,1)$  random variable results in a random variable whose distribution is exactly  $F$  (see “Inversion Methods” on page 6-7). The proof is essentially the opposite of the preceding proof for

the forward case. Another histogram illustrates the transformation to a gamma distribution:

```
x = gaminv(u,2,1); % transform to gamma values
```

```
figure  
hist(x,.25:.5:9.75) % plot the histogram of gamma values  
title('1000 Simulated N(0,1) Values Transformed to Gamma(2,1)')  
xlabel('X')  
ylabel('Frequency')  
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```



You can apply this two-step transformation to each variable of a standard bivariate normal, creating dependent random variables with arbitrary marginal distributions. Because the transformation works on each component separately, the two resulting random variables need not even have the same marginal distributions. The transformation is defined as:

$$Z = [Z_1, Z_2] \square N\left([0, 0], \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}\right)$$

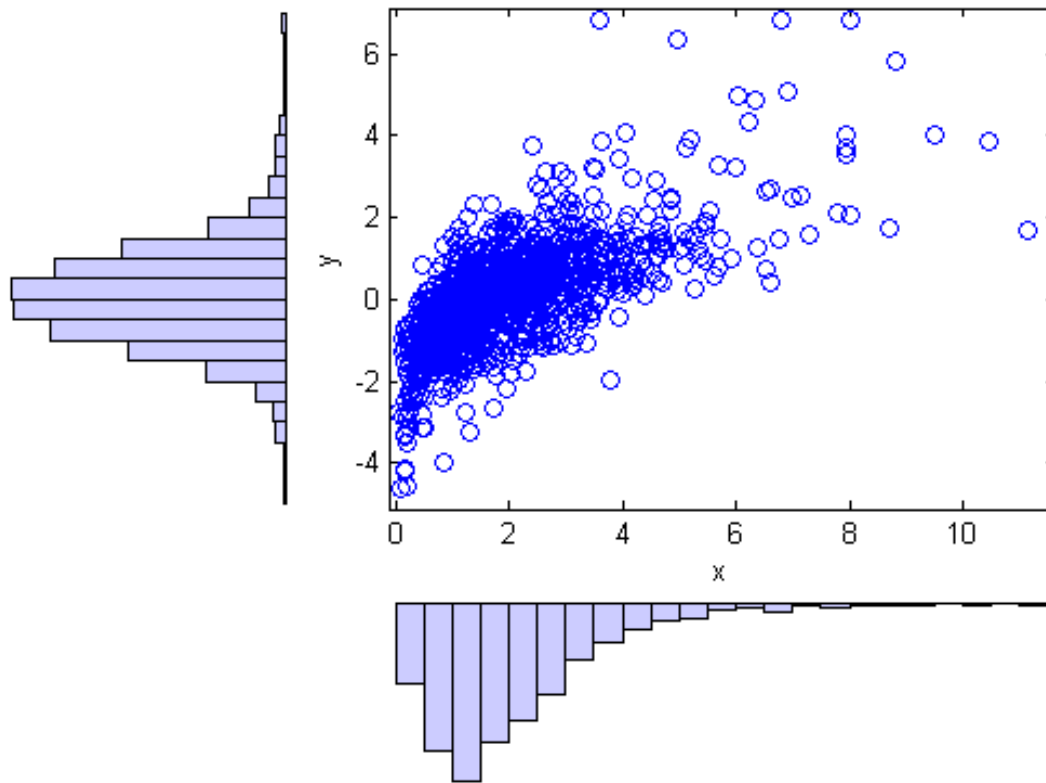
$$U = [\Phi(Z_1), \Phi(Z_2)]$$

$$X = [G_1(U_1), G_2(U_2)]$$

where  $G_1$  and  $G_2$  are inverse cdfs of two possibly different distributions. For example, the following generates random vectors from a bivariate distribution with  $t_5$  and Gamma(2,1) marginals:

```
n = 1000; rho = .7;
Z = mvnrnd([0 0],[1 rho; rho 1],n);
U = normcdf(Z);
X = [gaminv(U(:,1),2,1) tinv(U(:,2),5)];

% draw the scatter plot of data with histograms
figure
scatterhist(X(:,1),X(:,2),'Direction','out')
```



This plot has histograms alongside a scatter plot to show both the marginal distributions, and the dependence.

### Using Rank Correlation Coefficients

The correlation parameter,  $\rho$ , of the underlying bivariate normal determines the dependence between  $X_1$  and  $X_2$  in this construction. However, the linear correlation of  $X_1$  and  $X_2$  is not  $\rho$ . For example, in the original lognormal case, a closed form for that correlation is:



$$\text{cor}(X1, X2) = \frac{e^{\rho\sigma^2} - 1}{e^{\sigma^2} - 1}$$

which is strictly less than  $\rho$ , unless  $\rho$  is exactly 1. In more general cases such as the Gamma/ $t$  construction, the linear correlation between  $X1$  and  $X2$  is difficult or impossible to express in terms of  $\rho$ , but simulations show that the same effect happens.

That is because the linear correlation coefficient expresses the linear dependence between random variables, and when nonlinear transformations are applied to those random variables, linear correlation is not preserved. Instead, a rank correlation coefficient, such as Kendall's  $\tau$  or Spearman's  $\rho$ , is more appropriate.

Roughly speaking, these rank correlations measure the degree to which large or small values of one random variable associate with large or small values of another. However, unlike the linear correlation coefficient, they measure the association only in terms of ranks. As a consequence, the rank correlation is preserved under any monotonic transformation. In particular, the transformation method just described preserves the rank correlation. Therefore, knowing the rank correlation of the bivariate normal  $Z$  exactly determines the rank correlation of the final transformed random variables,  $X$ . While the linear correlation coefficient,  $\rho$ , is still needed to parameterize the underlying bivariate normal, Kendall's  $\tau$  or Spearman's  $\rho$  are more useful in describing the dependence between random variables, because they are invariant to the choice of marginal distribution.

For the bivariate normal, there is a simple one-to-one mapping between Kendall's  $\tau$  or Spearman's  $\rho$ , and the linear correlation coefficient  $\rho$ :

$$\tau = \frac{2}{\pi} \arcsin(\rho) \quad \text{or} \quad \rho = \sin\left(\tau \frac{\pi}{2}\right)$$

$$\rho_s = \frac{6}{\pi} \arcsin\left(\frac{\rho}{2}\right) \quad \text{or} \quad \rho = 2\sin\left(\rho_s \frac{\pi}{6}\right)$$

The following plot shows the relationship:

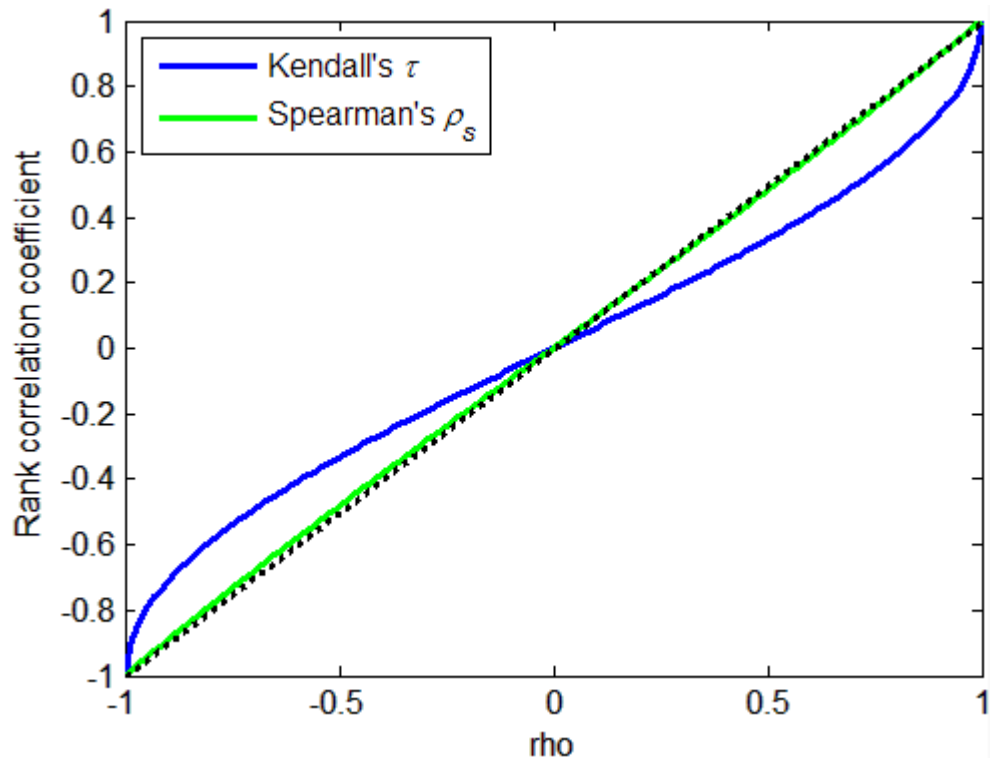
```
rho = -1:.01:1;
```

```

tau = 2.*asin(rho)./pi;
rho_s = 6.*asin(rho./2)./pi;

plot(rho,tau,'b-','LineWidth',2)
hold on
plot(rho,rho_s,'g-','LineWidth',2)
plot([-1 1],[-1 1],'k:','LineWidth',2)
axis([-1 1 -1 1])
xlabel('rho')
ylabel('Rank correlation coefficient')
legend('Kendall''s  $\tau$ ', ...
       'Spearman''s  $\rho_s$ ', ...
       'location','NW')

```



Thus, it is easy to create the desired rank correlation between  $X_1$  and  $X_2$ , regardless of their marginal distributions, by choosing the correct  $\rho$  parameter value for the linear correlation between  $Z_1$  and  $Z_2$ .

For the multivariate normal distribution, Spearman's rank correlation is almost identical to the linear correlation. However, this is not true once you transform to the final random variables.

### Using Bivariate Copulas

The first step of the construction described in the previous section defines what is known as a bivariate Gaussian copula. A copula is a multivariate probability distribution, where each random variable has a uniform marginal distribution on the unit interval  $[0,1]$ . These variables may be completely independent, deterministically related (e.g.,  $U_2 = U_1$ ), or anything in between. Because of the possibility for dependence among variables, you can use a copula to construct a new multivariate distribution for dependent variables. By transforming each of the variables in the copula separately using the inversion method, possibly using different cdfs, the resulting distribution can have arbitrary marginal distributions. Such multivariate distributions are often useful in simulations, when you know that the different random inputs are not independent of each other.

Statistics Toolbox functions compute:

- Probability density functions (`copulapdf`) and the cumulative distribution functions (`copulacdf`) for Gaussian copulas
- Rank correlations from linear correlations (`copulastat`) and vice versa (`copulaparam`)
- Random vectors (`copularnd`)
- Parameters for copulas fit to data (`copulafit`)

For example, use the `copularnd` function to create scatter plots of random values from a bivariate Gaussian copula for various levels of  $\rho$ , to illustrate the range of different dependence structures. The family of bivariate Gaussian copulas is parameterized by the linear correlation matrix:

$$P = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

$U_1$  and  $U_2$  approach linear dependence as  $\rho$  approaches  $\pm 1$ , and approach complete independence as  $\rho$  approaches zero:

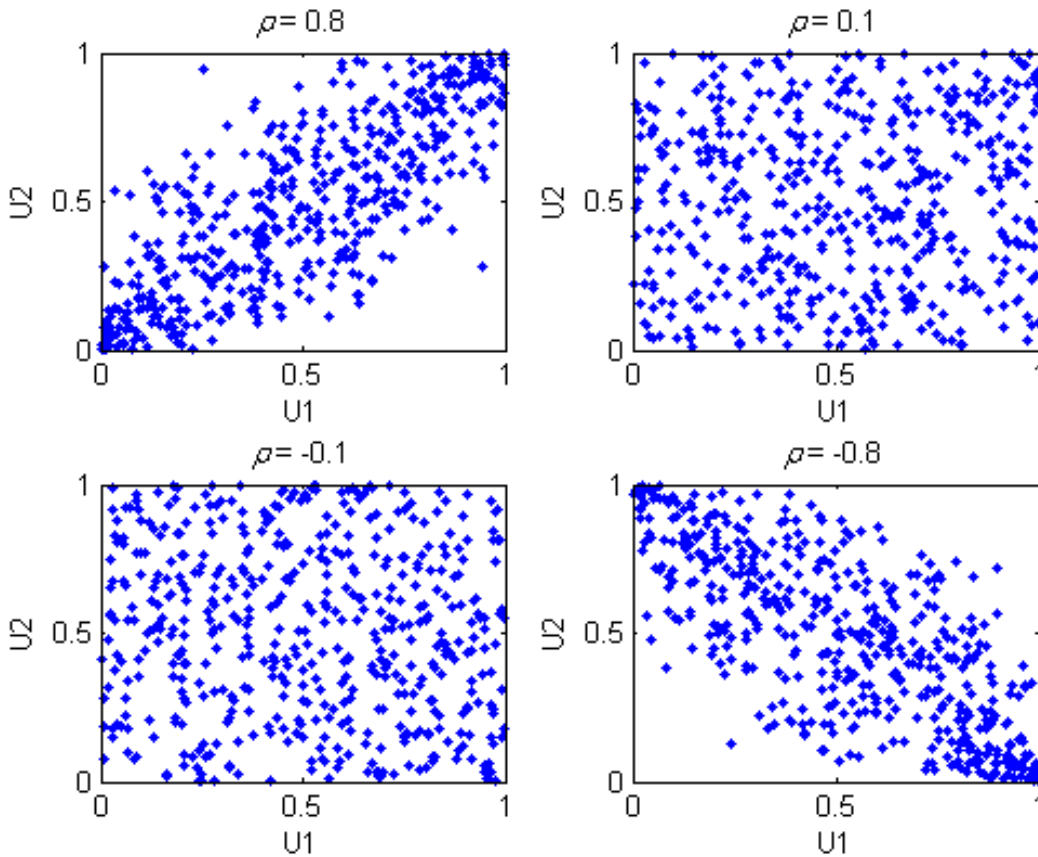
```
n = 500;
```

```
rng('default') % for reproducibility
U = copularnd('Gaussian',[1 .8; .8 1],n);
subplot(2,2,1)
plot(U(:,1),U(:,2),'.')
title('\it\rho = 0.8')
xlabel('U1')
ylabel('U2')

U = copularnd('Gaussian',[1 .1; .1 1],n);
subplot(2,2,2)
plot(U(:,1),U(:,2),'.')
title('\it\rho = 0.1')
xlabel('U1')
ylabel('U2')

U = copularnd('Gaussian',[1 -.1; -.1 1],n);
subplot(2,2,3)
plot(U(:,1),U(:,2),'.')
title('\it\rho = -0.1')
xlabel('U1')
ylabel('U2')

U = copularnd('Gaussian',[1 -.8; -.8 1],n);
subplot(2,2,4)
plot(U(:,1),U(:,2),'.')
title('\it\rho = -0.8')
xlabel('U1')
ylabel('U2')
```

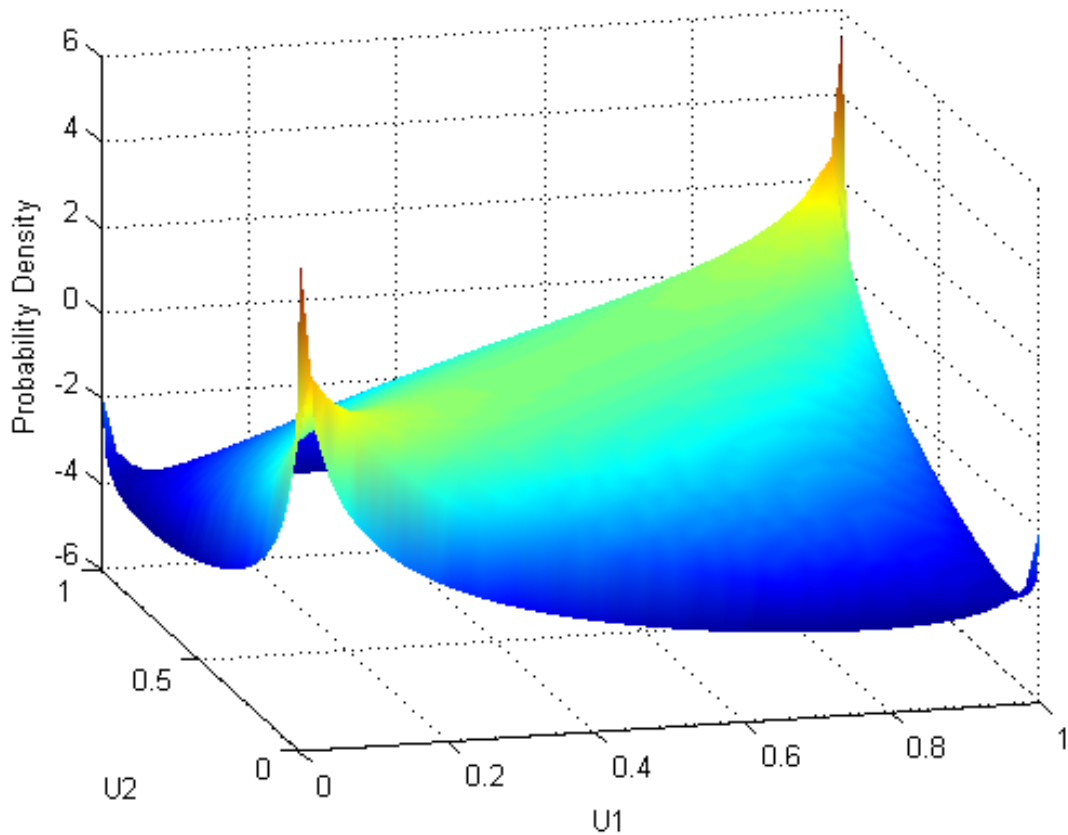


The dependence between  $U_1$  and  $U_2$  is completely separate from the marginal distributions of  $X_1 = G(U_1)$  and  $X_2 = G(U_2)$ .  $X_1$  and  $X_2$  can be given any marginal distributions, and still have the same rank correlation. This is one of the main appeals of copulas—they allow this separate specification of dependence and marginal distribution. You can also compute the pdf (`copulapdf`) and the cdf (`copulacdf`) for a copula. For example, these plots show the pdf and cdf for  $\rho = .8$ :

```
u1 = linspace(1e-3,1-1e-3,50);
u2 = linspace(1e-3,1-1e-3,50);
```

```
[U1,U2] = meshgrid(u1,u2);
Rho = [1 .8; .8 1];
f = copulapdf('t',[U1(:) U2(:)],Rho,5);
f = reshape(f,size(U1));

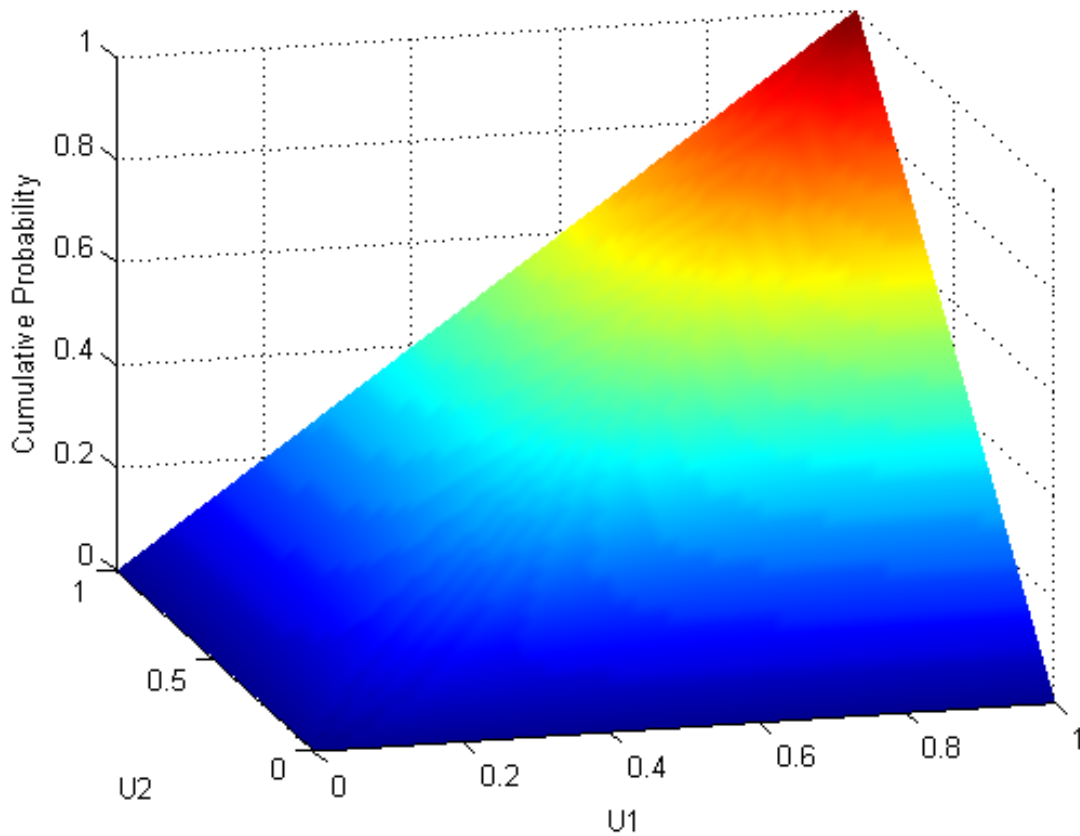
figure()
surf(u1,u2,log(f),'FaceColor','interp','EdgeColor','none')
view([-15,20])
xlabel('U1')
ylabel('U2')
zlabel('Probability Density')
```



```
u1 = linspace(1e-3,1-1e-3,50);  
u2 = linspace(1e-3,1-1e-3,50);  
[U1,U2] = meshgrid(u1,u2);  
F = copulacdf('t',[U1(:) U2(:)],Rho,5);  
F = reshape(F,size(U1));
```

```
figure()  
surf(u1,u2,F,'FaceColor','interp','EdgeColor','none')  
view([-15,20])  
xlabel('U1')
```

```
ylabel('U2')
zlabel('Cumulative Probability')
```



A different family of copulas can be constructed by starting from a bivariate  $t$  distribution and transforming using the corresponding  $t$  cdf. The bivariate  $t$  distribution is parameterized with  $P$ , the linear correlation matrix, and  $\nu$ , the degrees of freedom. Thus, for example, you can speak of a  $t_1$  or a  $t_5$  copula, based on the multivariate  $t$  with one and five degrees of freedom, respectively.



Just as for Gaussian copulas, Statistics Toolbox functions for  $t$  copulas compute:

- Probability density functions (`copulapdf`) and the cumulative distribution functions (`copulacdf`) for Gaussian copulas
- Rank correlations from linear correlations (`copulastat`) and vice versa (`copulaparam`)
- Random vectors (`copularnd`)
- Parameters for copulas fit to data (`copulafit`)

For example, use the `copularnd` function to create scatter plots of random values from a bivariate  $t_1$  copula for various levels of  $\rho$ , to illustrate the range of different dependence structures:

```
n = 500;
nu = 1;

rng('default') % for reproducibility
U = copularnd('t',[1 .8; .8 1],nu,n);
subplot(2,2,1)
plot(U(:,1),U(:,2),'.')
title('\it\rho = 0.8')
xlabel('U1')
ylabel('U2')

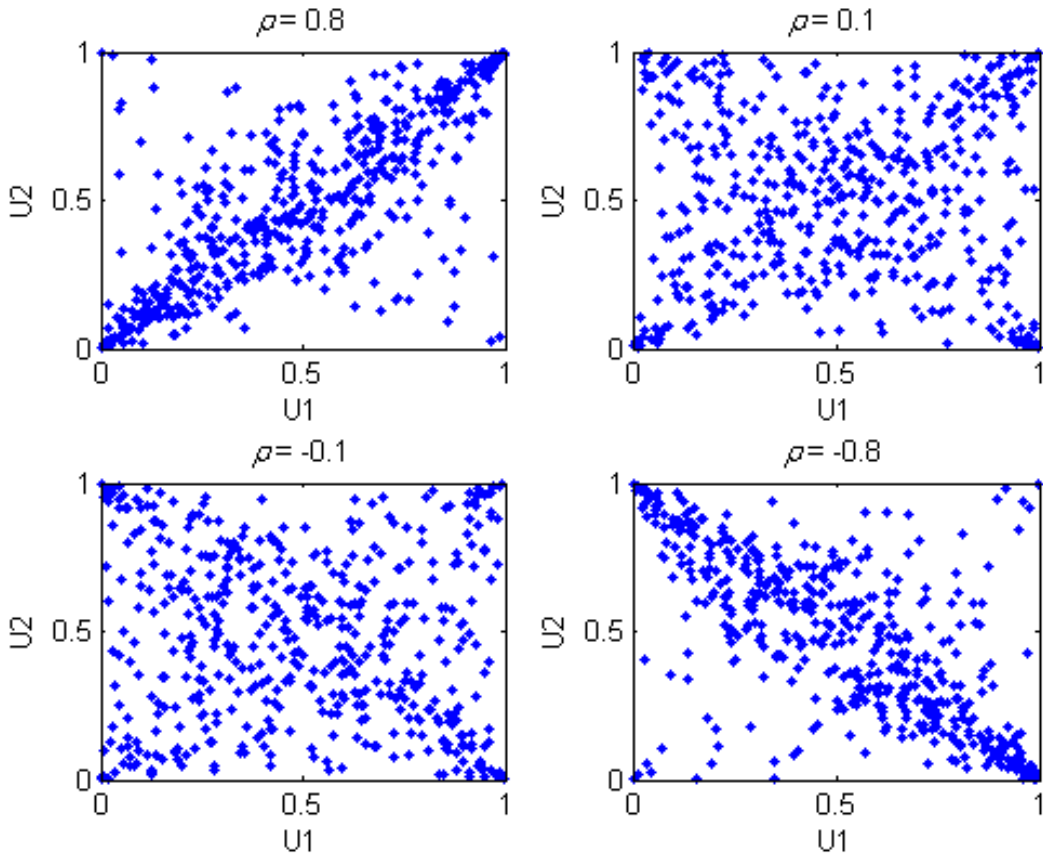
U = copularnd('t',[1 .1; .1 1],nu,n);
subplot(2,2,2)
plot(U(:,1),U(:,2),'.')
title('\it\rho = 0.1')
xlabel('U1')
ylabel('U2')

U = copularnd('t',[1 -.1; -.1 1],nu,n);
subplot(2,2,3)
plot(U(:,1),U(:,2),'.')
title('\it\rho = -0.1')
xlabel('U1')
ylabel('U2')
```

```

U = copularnd('t',[1 -.8; -.8 1],nu, n);
subplot(2,2,4)
plot(U(:,1),U(:,2),'.')
title('\it\rho = -0.8')
xlabel('U1')
ylabel('U2')

```



A  $t$  copula has uniform marginal distributions for  $U_1$  and  $U_2$ , just as a Gaussian copula does. The rank correlation  $\tau$  or  $\rho_s$  between components in a  $t$  copula is also the same function of  $\rho$  as for a Gaussian. However, as these plots

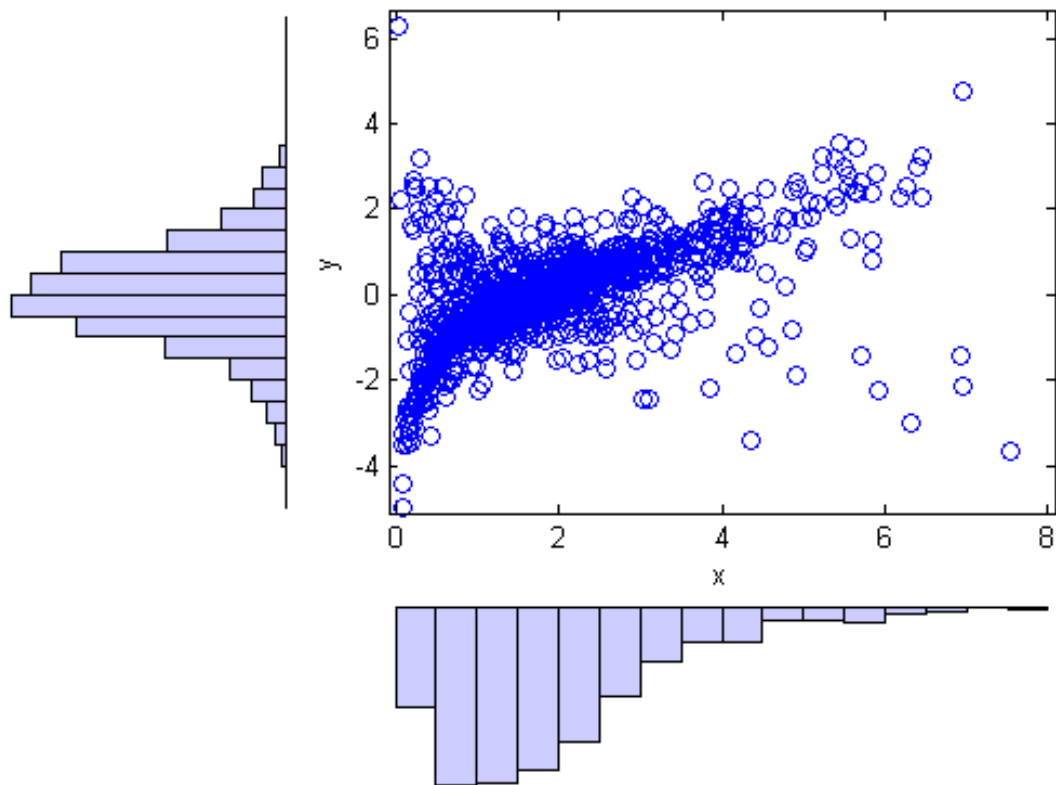
demonstrate, a  $t_1$  copula differs quite a bit from a Gaussian copula, even when their components have the same rank correlation. The difference is in their dependence structure. Not surprisingly, as the degrees of freedom parameter  $\nu$  is made larger, a  $t_\nu$  copula approaches the corresponding Gaussian copula.

As with a Gaussian copula, any marginal distributions can be imposed over a  $t$  copula. For example, using a  $t$  copula with 1 degree of freedom, you can again generate random vectors from a bivariate distribution with Gamma(2,1) and  $t_5$  marginals using `copularnd`:

```
n = 1000;
rho = .7;
nu = 1;

rng('default') % for reproducibility
U = copularnd('t',[1 rho; rho 1],nu,n);
X = [gaminv(U(:,1),2,1) tinv(U(:,2),5)];

figure()
scatterhist(X(:,1),X(:,2),'Direction','out')
```



Compared to the bivariate Gamma/ $t$  distribution constructed earlier, which was based on a Gaussian copula, the distribution constructed here, based on a  $t_1$  copula, has the same marginal distributions and the same rank correlation between variables but a very different dependence structure. This illustrates the fact that multivariate distributions are not uniquely defined by their marginal distributions, or by their correlations. The choice of a particular copula in an application may be based on actual observed data, or different copulas may be used as a way of determining the sensitivity of simulation results to the input distribution.

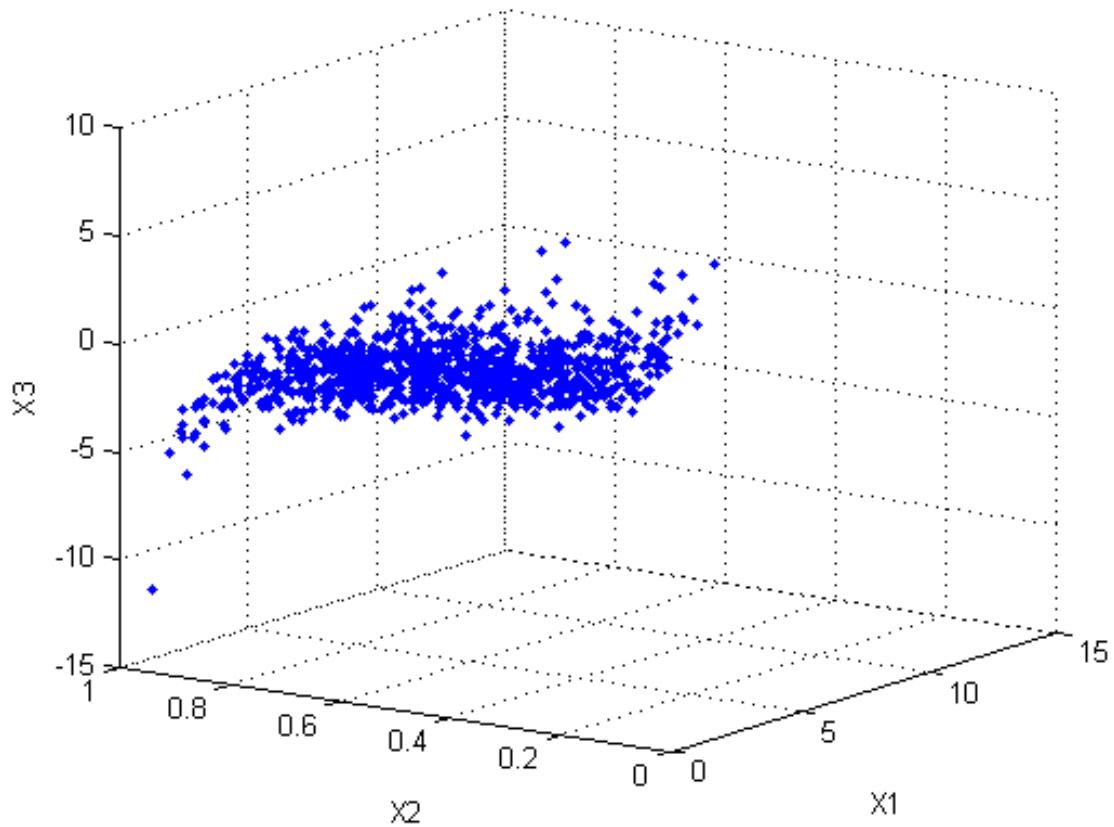
## Higher Dimension Copulas

The Gaussian and  $t$  copulas are known as elliptical copulas. It is easy to generalize elliptical copulas to a higher number of dimensions. For example, simulate data from a trivariate distribution with Gamma(2,1), Beta(2,2), and  $t_5$  marginals using a Gaussian copula and `copularnd`, as follows:

```
n = 1000;
Rho = [1 .4 .2; .4 1 -.8; .2 -.8 1];
rng('default') % for reproducibility
U = copularnd('Gaussian',Rho,n);
X = [gaminv(U(:,1),2,1) betainv(U(:,2),2,2) tinv(U(:,3),5)];
```

Plot the data.

```
subplot(1,1,1)
plot3(X(:,1),X(:,2),X(:,3),'.')
grid on
view([-55, 15])
xlabel('X1')
ylabel('X2')
zlabel('X3')
```



Notice that the relationship between the linear correlation parameter  $\rho$  and, for example, Kendall's  $\tau$ , holds for each entry in the correlation matrix  $P$  used here. You can verify that the sample rank correlations of the data are approximately equal to the theoretical values:

```
tauTheoretical = 2.*asin(Rho)./pi
```

```
tauTheoretical =
```

```

1.0000    0.2620    0.1282
0.2620    1.0000   -0.5903
0.1282   -0.5903    1.0000

```

```
tauSample = corr(X,'type','Kendall')
```

```
tauSample =
```

```

1.0000    0.2581    0.1414
0.2581    1.0000   -0.5790
0.1414   -0.5790    1.0000

```

## Archimedean Copulas

Statistics Toolbox functions are available for three bivariate Archimedean copula families:

- Clayton copulas
- Frank copulas
- Gumbel copulas

These are one-parameter families that are defined directly in terms of their cdfs, rather than being defined constructively using a standard multivariate distribution.

To compare these three Archimedean copulas to the Gaussian and  $t$  bivariate copulas, first use the `copulastat` function to find the rank correlation for a Gaussian or  $t$  copula with linear correlation parameter of 0.8, and then use the `copulaparam` function to find the Clayton copula parameter that corresponds to that rank correlation:

```
tau = copulastat('Gaussian',.8,'type','kendall')
```

```
tau =
```

```
0.5903
```

```
alpha = copulaparam('Clayton',tau,'type','kendall')
```

```
alpha =
```

```
2.8820
```

Finally, plot a random sample from the Clayton copula with `copularnd`. Repeat the same procedure for the Frank and Gumbel copulas:

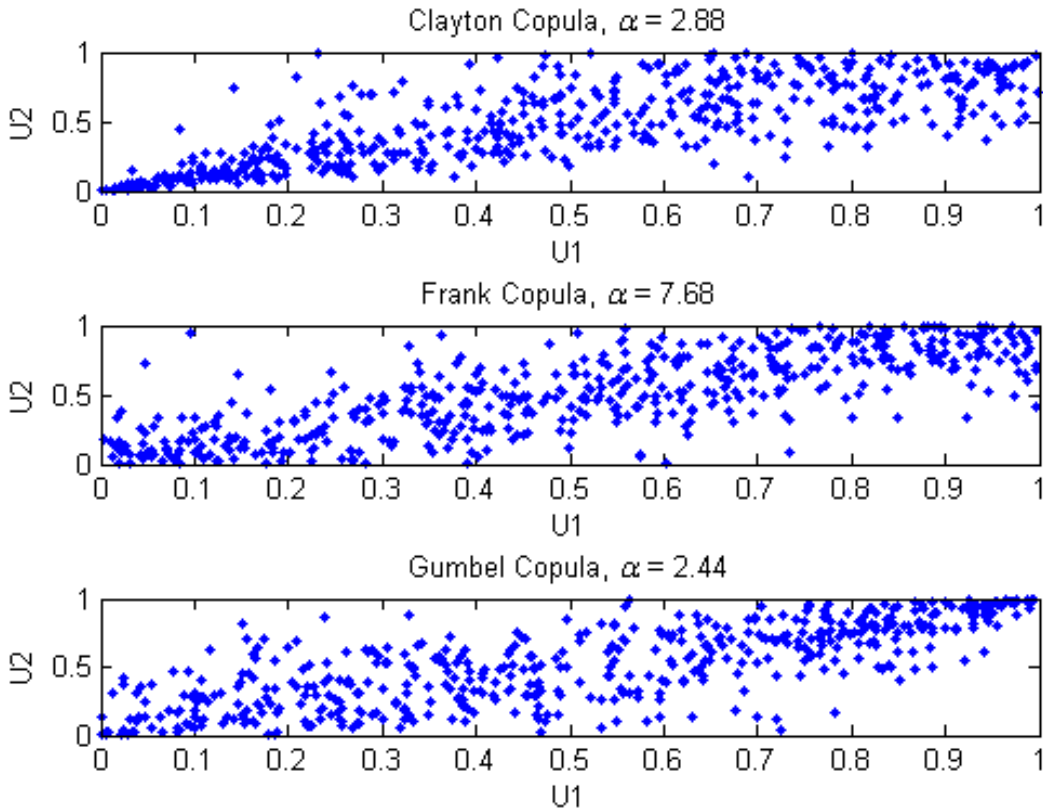
```
n = 500;
```

```
U = copularnd('Clayton',alpha,n);
subplot(3,1,1)
plot(U(:,1),U(:,2),'.');
title(['Clayton Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')
```

```
alpha = copulaparam('Frank',tau,'type','kendall');
U = copularnd('Frank',alpha,n);
subplot(3,1,2)
plot(U(:,1),U(:,2),'.');
title(['Frank Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')
```

```
alpha = copulaparam('Gumbel',tau,'type','kendall');
U = copularnd('Gumbel',alpha,n);
subplot(3,1,3)
plot(U(:,1),U(:,2),'.');
title(['Gumbel Copula, {\it\alpha} = ',sprintf('%0.2f',alpha)])
xlabel('U1')
ylabel('U2')
```





### Simulating Dependent Multivariate Data Using Copulas

To simulate dependent multivariate data using a copula, you must specify each of the following:

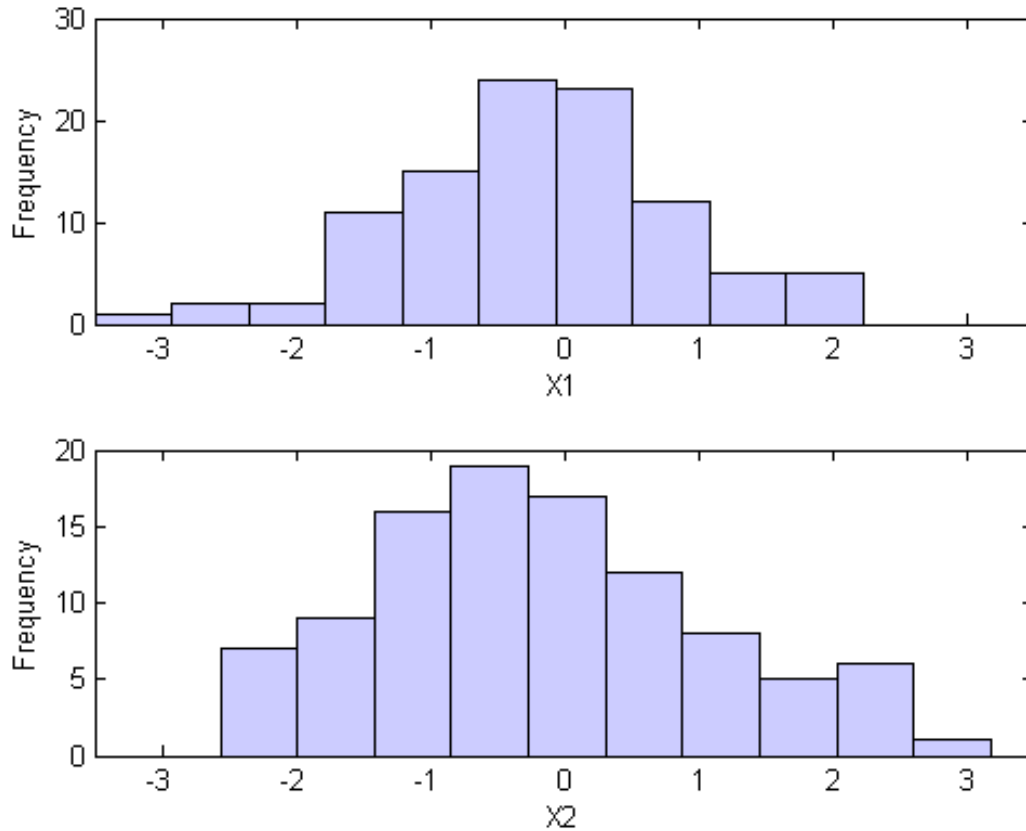
- The copula family (and any shape parameters)
- The rank correlations among variables
- Marginal distributions for each variable

Suppose you have return data for two stocks and want to run a Monte Carlo simulation with inputs that follow the same distributions as the data:

```
load stockreturns
nobs = size(stocks,1);

subplot(2,1,1)
hist(stocks(:,1),10)
xlim([-3.5 3.5])
xlabel('X1')
ylabel('Frequency')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])

subplot(2,1,2)
hist(stocks(:,2),10)
xlim([-3.5 3.5])
xlabel('X2')
ylabel('Frequency')
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```



You could fit a parametric model separately to each dataset, and use those estimates as the marginal distributions. However, a parametric model may not be sufficiently flexible. Instead, you can use a nonparametric model to transform to the marginal distributions. All that is needed is a way to compute the inverse cdf for the nonparametric model.

The simplest nonparametric model is the empirical cdf, as computed by the `ecdf` function. For a discrete marginal distribution, this is appropriate. However, for a continuous distribution, use a model that is smoother than the step function computed by `ecdf`. One way to do that is to estimate

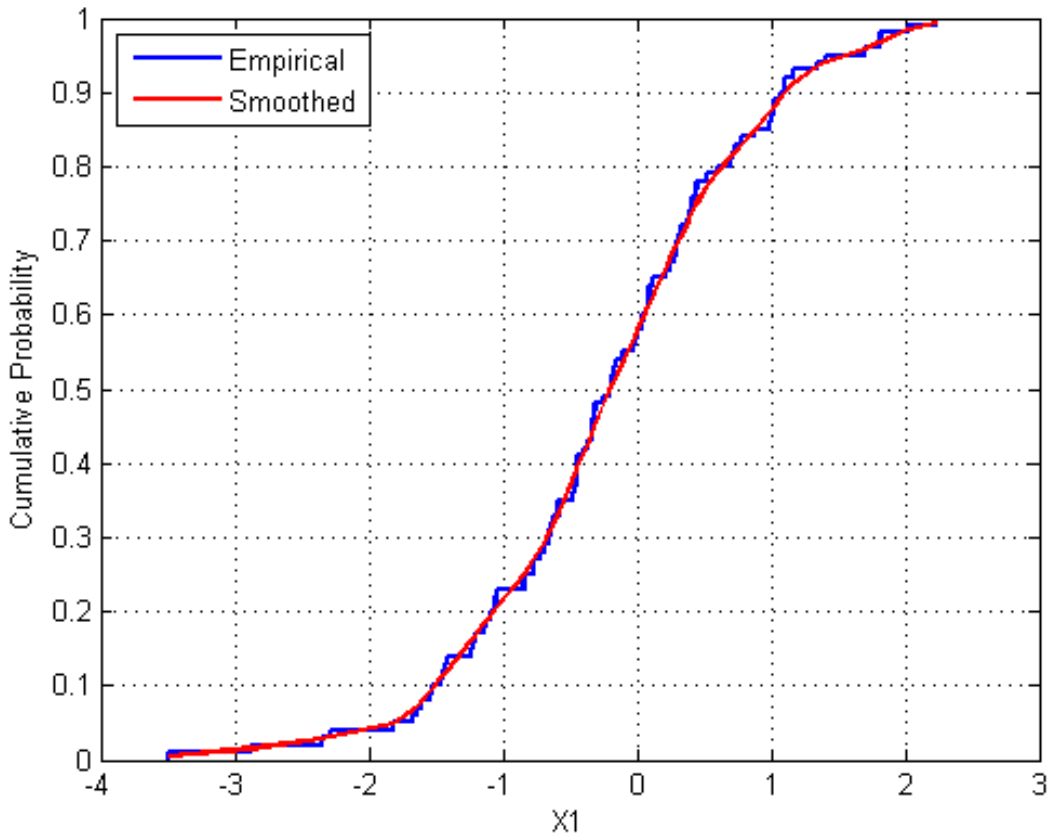
the empirical cdf and interpolate between the midpoints of the steps with a piecewise linear function. Another way is to use kernel smoothing with `ksdensity`. For example, compare the empirical cdf to a kernel smoothed cdf estimate for the first variable:

```
[Fi,xi] = ecdf(stocks(:,1));

figure()
stairs(xi,Fi,'b','LineWidth',2)
hold on

Fi_sm = ksdensity(stocks(:,1),xi,'function','cdf','width',.15);

plot(xi,Fi_sm,'r-','LineWidth',1.5)
xlabel('X1')
ylabel('Cumulative Probability')
legend('Empirical','Smoothed','Location','NW')
grid on
```



For the simulation, experiment with different copulas and correlations. Here, you will use a bivariate  $t$  copula with a fairly small degrees of freedom parameter. For the correlation parameter, you can compute the rank correlation of the data.

```
nu = 5;
tau = corr(stocks(:,1),stocks(:,2),'type','kendall')
```

```
tau =
```

```
0.5180
```

Find the corresponding linear correlation parameter for the  $t$  copula using `copulaparam`.

```
rho = copulaparam('t', tau, nu, 'type','kendall')
```

```
rho =
```

```
0.7268
```

Next, use `copularnd` to generate random values from the  $t$  copula and transform using the nonparametric inverse cdfs. The `ksdensity` function allows you to make a kernel estimate of distribution and evaluate the inverse cdf at the copula points all in one step:

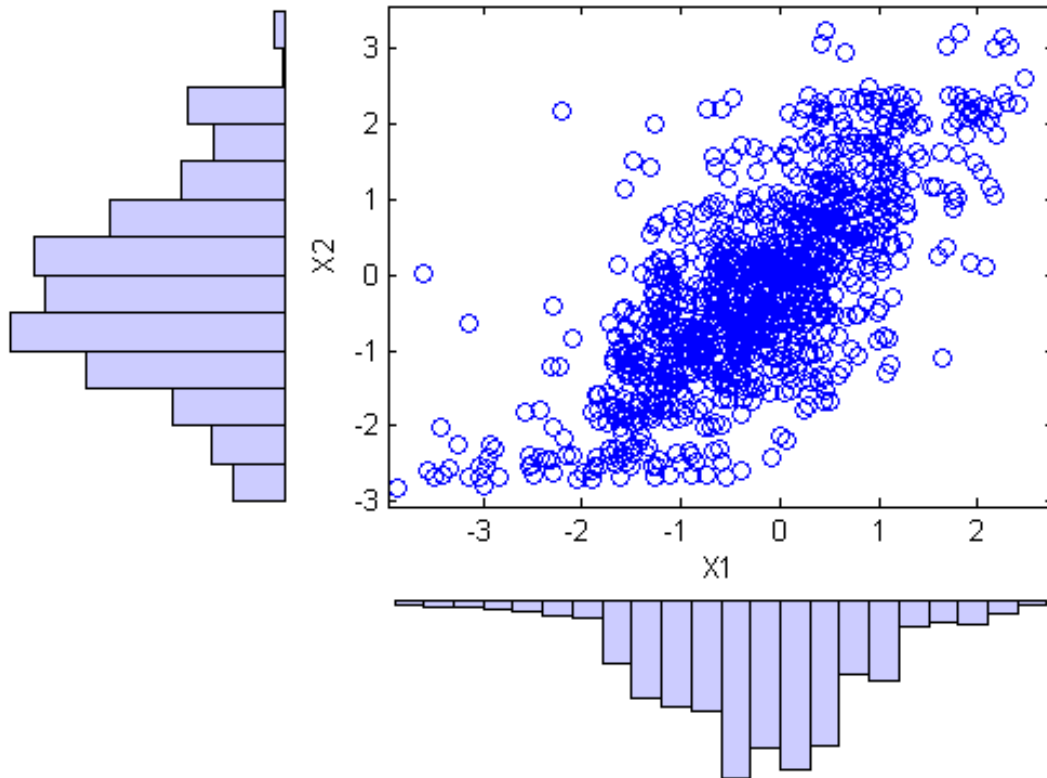
```
n = 1000;
U = copularnd('t',[1 rho; rho 1],nu,n);

X1 = ksdensity(stocks(:,1),U(:,1),...
               'function','icdf','width',.15);
X2 = ksdensity(stocks(:,2),U(:,2),...
               'function','icdf','width',.15);
```

Alternatively, when you have a large amount of data or need to simulate more than one set of values, it may be more efficient to compute the inverse cdf over a grid of values in the interval (0,1) and use interpolation to evaluate it at the copula points:

```
p = linspace(0.00001,0.99999,1000);
G1 = ksdensity(stocks(:,1),p,'function','icdf','width',0.15);
X1 = interp1(p,G1,U(:,1),'spline');
G2 = ksdensity(stocks(:,2),p,'function','icdf','width',0.15);
X2 = interp1(p,G2,U(:,2),'spline');

scatterhist(X1,X2,'Direction','out')
```



The marginal histograms of the simulated data are a smoothed version of the histograms for the original data. The amount of smoothing is controlled by the bandwidth input to `ksdensity`.

### Fitting Copulas to Data

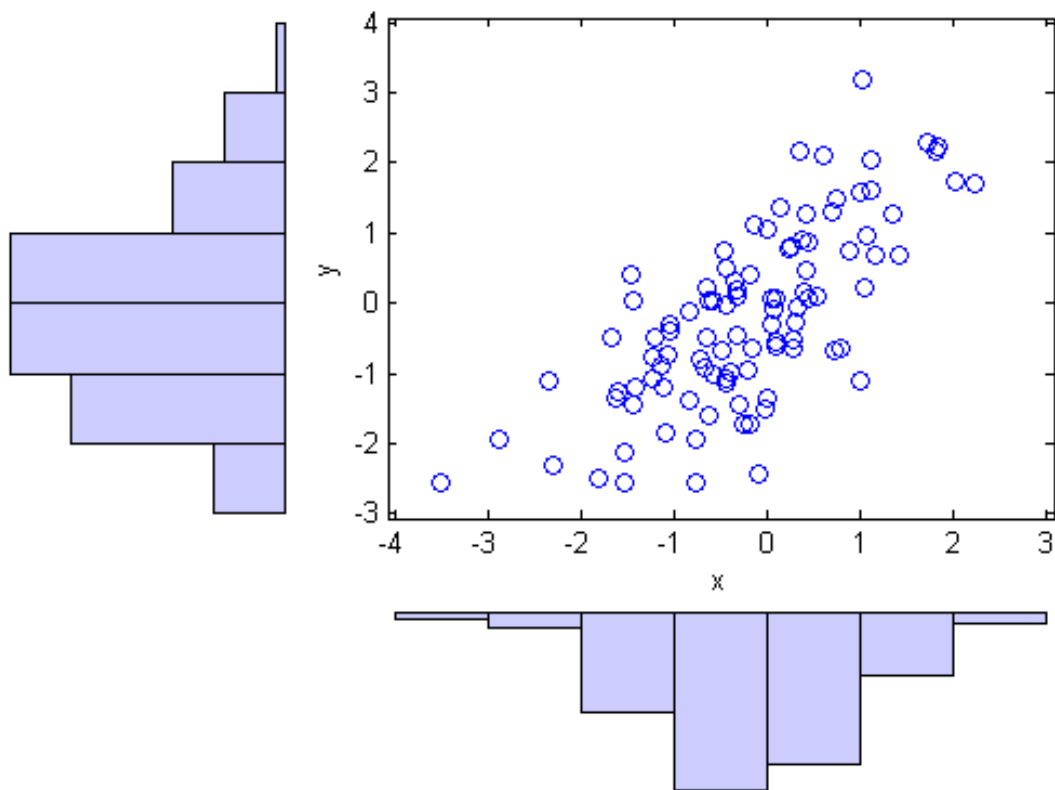
This example shows how to use `copulafit` to calibrate copulas with data. To generate data  $X_{\text{sim}}$  with a distribution "just like" (in terms of marginal distributions and correlations) the distribution of data in the matrix  $X$ , you need to fit marginal distributions to the columns of  $X$ , use appropriate cdf

functions to transform  $X$  to  $U$ , so that  $U$  has values between 0 and 1, use `copulafit` to fit a copula to  $U$ , generate new data  $U_{\text{sim}}$  from the copula, and use appropriate inverse cdf functions to transform  $U_{\text{sim}}$  to  $X_{\text{sim}}$ .

Load and plot the simulated stock return data.

```
load stockreturns
x = stocks(:,1);
y = stocks(:,2);

scatterhist(x,y,'Direction','out')
```

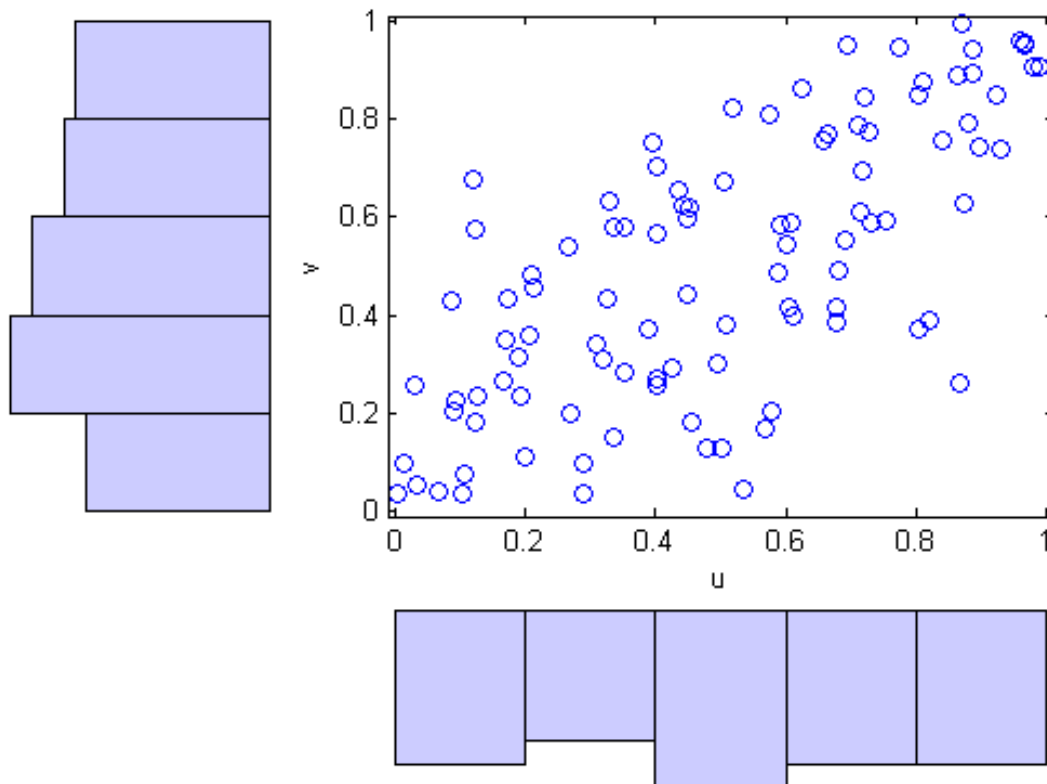




Transform the data to the copula scale (unit square) using a kernel estimator of the cumulative distribution function.

```
u = ksdensity(x,x,'function','cdf');
v = ksdensity(y,y,'function','cdf');
```

```
scatterhist(u,v,'Direction','out')
xlabel('u')
ylabel('v')
```



Fit a  $t$  copula.

```
[Rho,nu] = copulafit('t',[u v],'Method','ApproximateML')
```

```
Rho =
```

```
    1.0000    0.7220  
    0.7220    1.0000
```

```
nu =
```

```
    3.2763e+06
```

Generate a random sample from the  $t$  copula.

```
r = copularnd('t',Rho,nu,1000);
```

```
u1 = r(:,1);
```

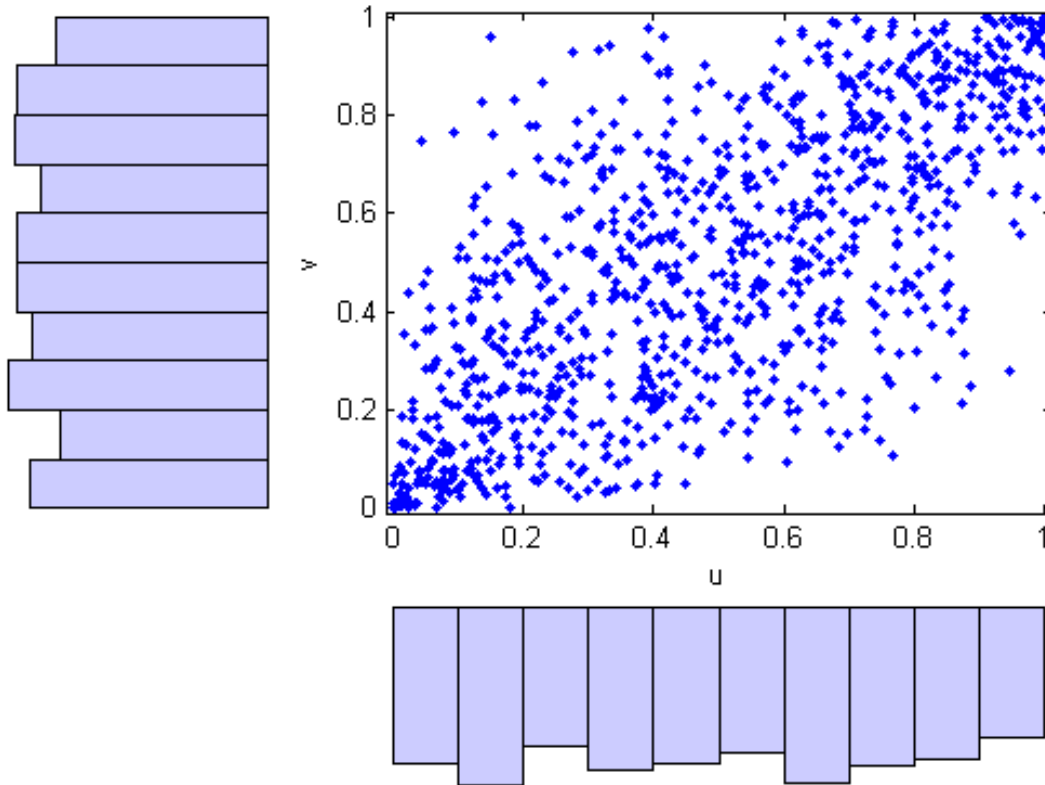
```
v1 = r(:,2);
```

```
scatterhist(u1,v1,'Direction','out')
```

```
xlabel('u')
```

```
ylabel('v')
```

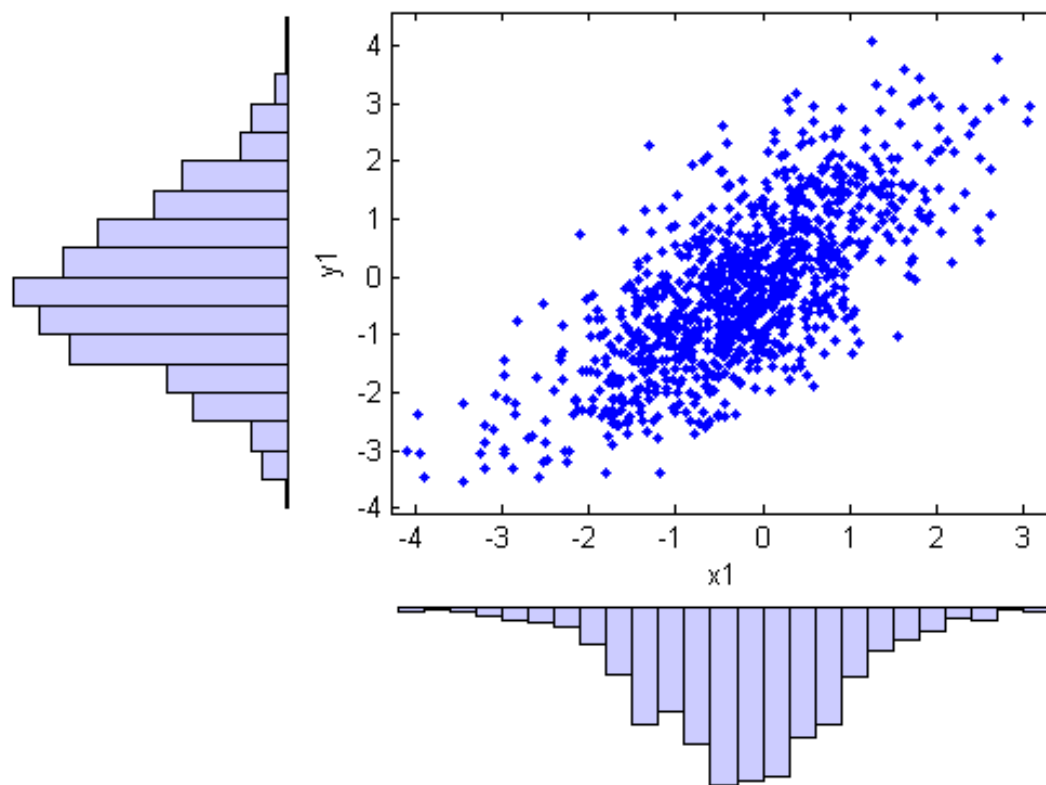
```
set(get(gca,'children'),'marker','.')
```



Transform the random sample back to the original scale of the data.

```
x1 = ksdensity(x,u1,'function','icdf');  
y1 = ksdensity(y,v1,'function','icdf');
```

```
scatterhist(x1,y1,'Direction','out')  
set(get(gca,'children'),'marker','.')
```



As the example illustrates, copulas integrate naturally with other distribution fitting functions.

# Random Number Generation

---

- “Generating Random Data” on page 6-2
- “Random Number Generation Functions” on page 6-3
- “Common Generation Methods” on page 6-5
- “Representing Sampling Distributions Using Markov Chain Samplers” on page 6-13
- “Generating Quasi-Random Numbers” on page 6-15
- “Generating Data Using Flexible Families of Distributions” on page 6-26

### Generating Random Data

*Pseudorandom* numbers are generated by deterministic algorithms. They are "random" in the sense that, on average, they pass statistical tests regarding their distribution and correlation. They differ from true random numbers in that they are generated by an algorithm, rather than a truly random process.

*Random number generators* (RNGs) like those in MATLAB are algorithms for generating pseudorandom numbers with a specified distribution.

For more information on the GUI for generating random numbers from supported distributions, see "Explore the Random Number Generation UI" on page 5-80.

## Random Number Generation Functions

The following table lists the supported distributions and their respective random number generation functions.

Distribution	Random Number Generation Function
Beta	betarnd, random, randtool
Binomial	binornd, random, randtool
Chi-square	chi2rnd, random, randtool
Clayton copula	copularnd
Exponential	exprnd, random, randtool
Extreme value	evrnd, random, randtool
$F$	frnd, random, randtool
Frank copula	copularnd
Gamma	gamrnd, randg, random, randtool
Gaussian copula	copularnd
Gaussian mixture	random
Generalized extreme value	gevrnd, random, randtool
Generalized Pareto	gprnd, random, randtool
Geometric	geornd, random, randtool
Gumbel copula	copularnd
Hypergeometric	hygernd, random
Inverse Wishart	iwishrnd
Johnson system	johnsrnd
Lognormal	lognrnd, random, randtool
Multinomial	mnrnd
Multivariate normal	mvnrnd
Multivariate $t$	mvtrnd

<b>Distribution</b>	<b>Random Number Generation Function</b>
Negative binomial	nbinrnd, random, randtool
Noncentral chi-square	ncx2rnd, random, randtool
Noncentral $F$	ncfrnd, random, randtool
Noncentral $t$	nctrnd, random, randtool
Normal (Gaussian)	normrnd, randn, random, randtool
Pearson system	pearsrnd
Piecewise	random
Poisson	poissrnd, random, randtool
Rayleigh	raylrnd, random, randtool
Student's $t$	trnd, random, randtool
$t$ copula	copularnd
Uniform (continuous)	unifrnd, rand, random
Uniform (discrete)	unidrnd, random, randtool
Weibull	wblrnd, random
Wishart	wishrnd



## Common Generation Methods

### In this section...

“Direct Methods” on page 6-5

“Inversion Methods” on page 6-7

“Acceptance-Rejection Methods” on page 6-9

Methods for generating pseudorandom numbers usually start with uniform random numbers, like the MATLAB `rand` function produces. The methods described in this section detail how to produce random numbers from other distributions.

### Direct Methods

Direct methods directly use the definition of the distribution.

For example, consider binomial random numbers. A binomial random number is the number of heads in  $N$  tosses of a coin with probability  $p$  of a heads on any single toss. If you generate  $N$  uniform random numbers on the interval  $(0,1)$  and count the number less than  $p$ , then the count is a binomial random number with parameters  $N$  and  $p$ .

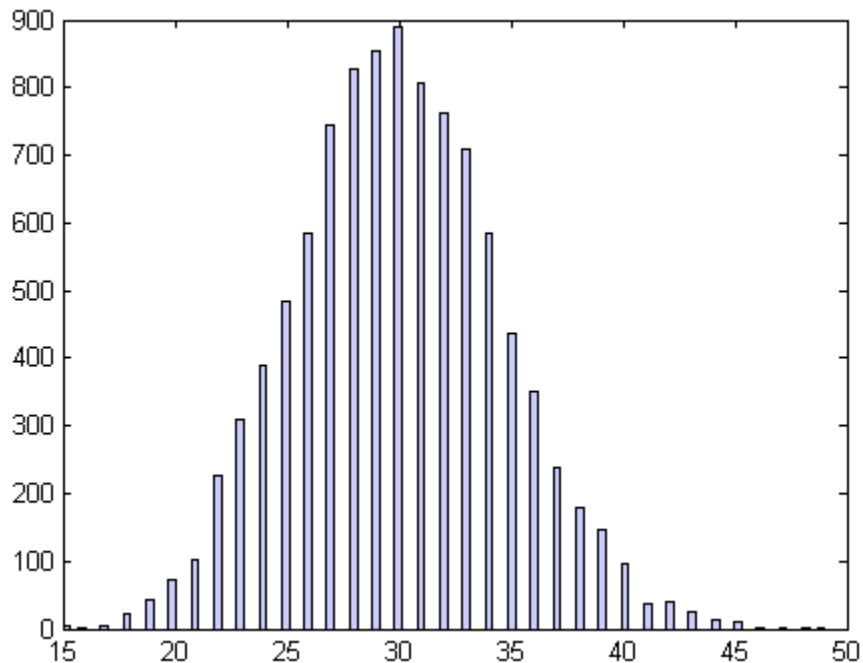
This function is a simple implementation of a binomial RNG using the direct approach:

```
function X = directbinornd(N,p,m,n)

X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    u = rand(N,1);
    X(i) = sum(u < p);
end
```

For example:

```
X = directbinornd(100,0.3,1e4,1);
hist(X,101)
set(get(gca, 'Children'), 'FaceColor', [.8 .8 1])
```



The Statistics Toolbox function `binornd` uses a modified direct method, based on the definition of a binomial random variable as the sum of Bernoulli random variables.

You can easily convert the previous method to a random number generator for the Poisson distribution with parameter  $\lambda$ . The Poisson distribution is the limiting case of the binomial distribution as  $N$  approaches infinity,  $p$  approaches zero, and  $Np$  is held fixed at  $\lambda$ . To generate Poisson random numbers, create a version of the previous generator that inputs  $\lambda$  rather than  $N$  and  $p$ , and internally sets  $N$  to some large number and  $p$  to  $\lambda/N$ .

The Statistics Toolbox function `poissrnd` actually uses two direct methods:

- A waiting time method for small values of  $\lambda$
- A method due to Ahrens and Dieter for larger values of  $\lambda$

## Inversion Methods

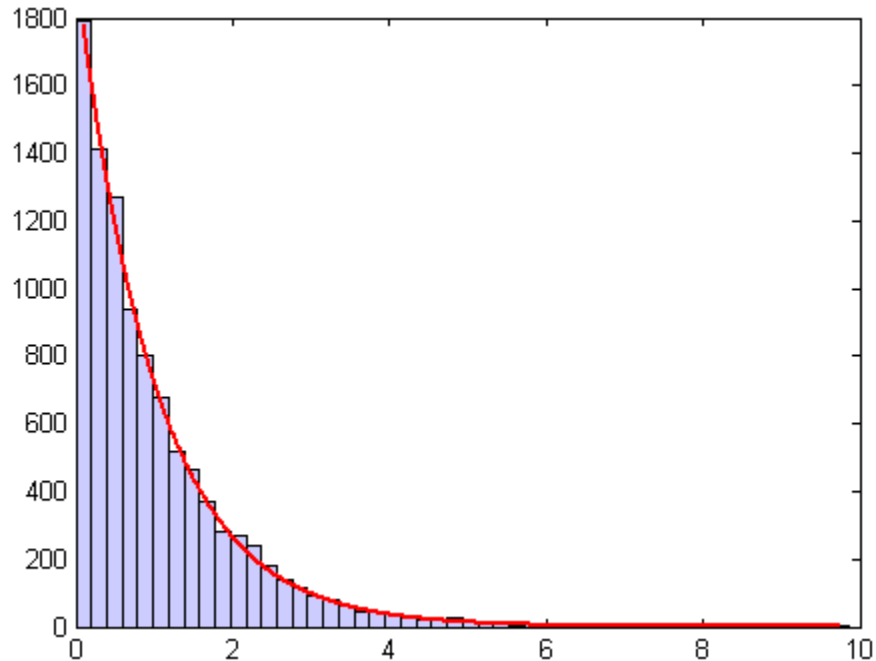
*Inversion methods* are based on the observation that continuous cumulative distribution functions (cdfs) range uniformly over the interval (0,1). If  $u$  is a uniform random number on (0,1), then using  $X = F^{-1}(U)$  generates a random number  $X$  from a continuous distribution with specified cdf  $F$ .

For example, the following code generates random numbers from a specific exponential distribution using the inverse cdf and the MATLAB uniform random number generator `rand`:

```
mu = 1;  
X = expinv(rand(1e4,1),mu);
```

Compare the distribution of the generated random numbers to the pdf of the specified exponential by scaling the pdf to the area of the histogram used to display the distribution:

```
numbins = 50;  
hist(X,numbins)  
set(get(gca,'Children'),'FaceColor',[.8 .8 1])  
hold on  
  
[bincounts,binpositions] = hist(X,numbins);  
binwidth = binpositions(2) - binpositions(1);  
histarea = binwidth*sum(bincounts);  
  
x = binpositions(1):0.001:binpositions(end);  
y = exppdf(x,mu);  
plot(x,histarea*y,'r','LineWidth',2)
```



Inversion methods also work for discrete distributions. To generate a random number  $X$  from a discrete distribution with probability mass vector  $P(X=x_i) = p_i$  where  $x_0 < x_1 < x_2 < \dots$ , generate a uniform random number  $u$  on  $(0,1)$  and then set  $X = x_i$  if  $F(x_{i-1}) < u < F(x_i)$ .

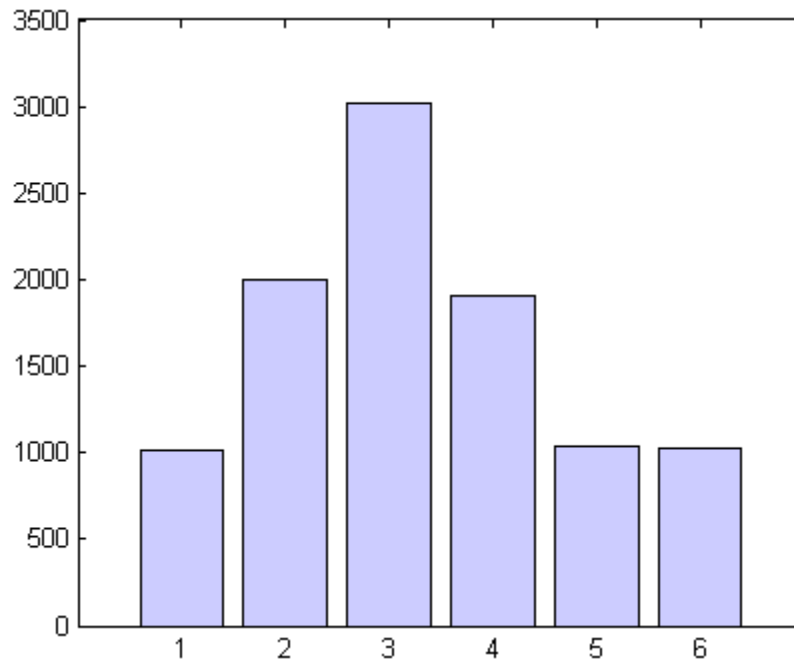
For example, the following function implements an inversion method for a discrete distribution with probability mass vector  $p$ :

```
function X = discreteinvrnd(p,m,n)

X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    u = rand;
    I = find(u < cumsum(p));
    X(i) = min(I);
end
```

Use the function to generate random numbers from any discrete distribution:

```
p = [0.1 0.2 0.3 0.2 0.1 0.1]; % Probability mass vector
X = discreteinvrnd(p,1e4,1);
[n,x] = hist(X,length(p));
bar(1:length(p),n)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```

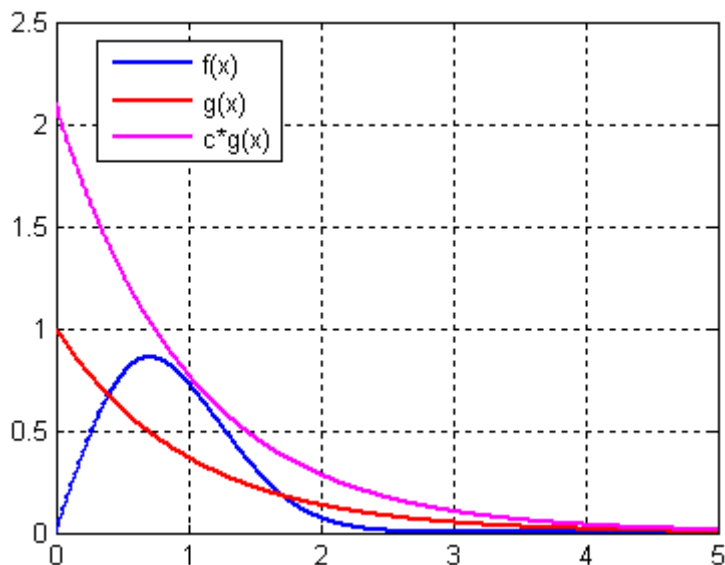


## Acceptance-Rejection Methods

The functional form of some distributions makes it difficult or time-consuming to generate random numbers using direct or inversion methods.

Acceptance-rejection methods provide an alternative in these cases.

*Acceptance-rejection methods* begin with uniform random numbers, but require an additional random number generator. If your goal is to generate a random number from a continuous distribution with pdf  $f$ , acceptance-rejection methods first generate a random number from a continuous distribution with pdf  $g$  satisfying  $f(x) \leq cg(x)$  for some  $c$  and all  $x$ .



A continuous acceptance-rejection RNG proceeds as follows:

- 1** Chooses a density  $g$ .
- 2** Finds a constant  $c$  such that  $f(x)/g(x) \leq c$  for all  $x$ .
- 3** Generates a uniform random number  $u$ .
- 4** Generates a random number  $v$  from  $g$ .
- 5** If  $cu \leq f(v)/g(v)$ , accepts and returns  $v$ .
- 6** Otherwise, rejects  $v$  and goes to step 3.

For efficiency, a “cheap” method is necessary for generating random numbers from  $g$ , and the scalar  $c$  should be small. The expected number of iterations to produce a single random number is  $c$ .

The following function implements an acceptance-rejection method for generating random numbers from pdf  $f$ , given  $f$ ,  $g$ , the RNG `grnd` for  $g$ , and the constant  $c$ :

```

function X = accrejrnd(f,g,grnd,c,m,n)

X = zeros(m,n); % Preallocate memory
for i = 1:m*n
    accept = false;
    while accept == false
        u = rand();
        v = grnd();
        if c*u <= f(v)/g(v)
            X(i) = v;
            accept = true;
        end
    end
end
end

```

For example, the function  $f(x) = xe^{-x/2}$  satisfies the conditions for a pdf on  $[0, \infty)$  (nonnegative and integrates to 1). The exponential pdf with mean 1,  $f(x) = e^{-x}$ , dominates  $g$  for  $c$  greater than about 2.2. Thus, you can use `rand` and `exprnd` to generate random numbers from  $f$ :

```

f = @(x)x.*exp(-(x.^2)/2);
g = @(x)exp(-x);
grnd = @()exprnd(1);
X = accrejrnd(f,g,grnd,2.2,1e4,1);

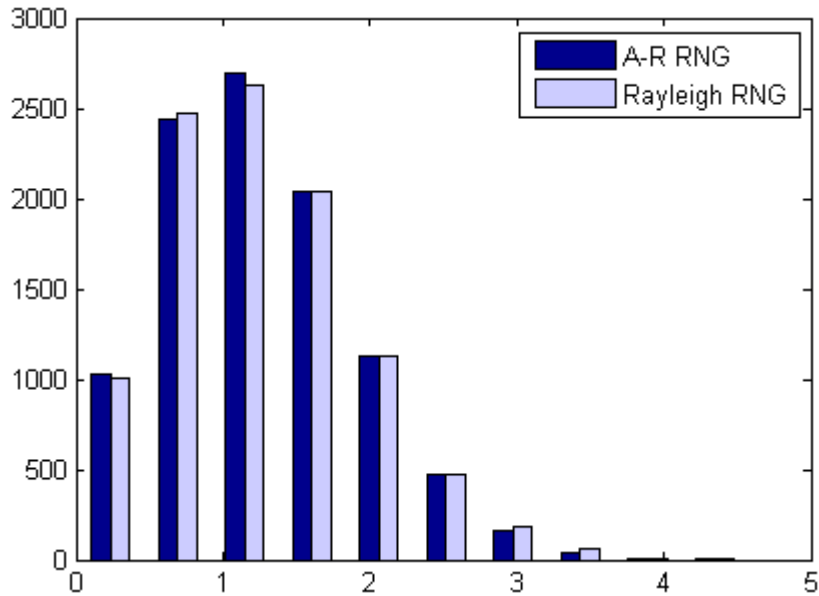
```

The pdf  $f$  is actually a Rayleigh distribution with shape parameter 1. This example compares the distribution of random numbers generated by the acceptance-rejection method with those generated by `raylrnd`:

```

Y = raylrnd(1,1e4,1);
hist([X Y])
h = get(gca,'Children');
set(h(1),'FaceColor',[.8 .8 1])
legend('A-R RNG','Rayleigh RNG')

```



The Statistics Toolbox function `raylrnd` uses a transformation method, expressing a Rayleigh random variable in terms of a chi-square random variable, which you compute using `randn`.

Acceptance-rejection methods also work for discrete distributions. In this case, the goal is to generate random numbers from a distribution with probability mass  $P_p(X = i) = p_i$ , assuming that you have a method for generating random numbers from a distribution with probability mass  $P_q(X = i) = q_i$ . The RNG proceeds as follows:

- 1** Chooses a density  $P_q$ .
- 2** Finds a constant  $c$  such that  $p_i/q_i \leq c$  for all  $i$ .
- 3** Generates a uniform random number  $u$ .
- 4** Generates a random number  $v$  from  $P_q$ .
- 5** If  $cu \leq p_i/q_i$ , accepts and returns  $v$ .
- 6** Otherwise, rejects  $v$  and goes to step 3.



# Representing Sampling Distributions Using Markov Chain Samplers

## In this section...

“Using the Metropolis-Hastings Algorithm” on page 6-13

“Using Slice Sampling” on page 6-14

The methods in “Common Generation Methods” on page 6-5 might be inadequate when sampling distributions are difficult to represent in computations. Such distributions arise, for example, in Bayesian data analysis and in the large combinatorial problems of Markov chain Monte Carlo (MCMC) simulations. An alternative is to construct a Markov chain with a stationary distribution equal to the target sampling distribution, using the states of the chain to generate random numbers after an initial burn-in period in which the state distribution converges to the target.

## Using the Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm draws samples from a distribution that is only known up to a constant. Random numbers are generated from a distribution with a probability density function that is equal to or proportional to a proposal function.

To generate random numbers:

- 1 Assume an initial value  $x(t)$ .
- 2 Draw a sample,  $y(t)$ , from a proposal distribution  $q(y | x(t))$ .
- 3 Accept  $y(t)$  as the next sample  $x(t + 1)$  with probability  $r(x(t), y(t))$ , and keep  $x(t)$  as the next sample  $x(t + 1)$  with probability  $1 - r(x(t), y(t))$ , where:

$$r(x, y) = \min \left\{ \frac{f(y) q(x | y)}{f(x) q(y | x)}, 1 \right\}$$

- 4 Increment  $t \rightarrow t+1$ , and repeat steps 2 and 3 until you get the desired number of samples.

Generate random numbers using the Metropolis-Hastings method with the `mhsample` function. To produce quality samples efficiently with the Metropolis-Hastings algorithm, it is crucial to select a good proposal distribution. If it is difficult to find an efficient proposal distribution, use the slice sampling algorithm (`slicesample`) without explicitly specifying a proposal distribution.

## Using Slice Sampling

In instances where it is difficult to find an efficient Metropolis-Hastings proposal distribution, the slice sampling algorithm does not require an explicit specification. The slice sampling algorithm draws samples from the region under the density function using a sequence of vertical and horizontal steps. First, it selects a height at random from 0 to the density function  $f(x)$ . Then, it selects a new  $x$  value at random by sampling from the horizontal “slice” of the density above the selected height. A similar slice sampling algorithm is used for a multivariate distribution.

If a function  $f(x)$  proportional to the density function is given, then do the following to generate random numbers:

- 1** Assume an initial value  $x(t)$  within the domain of  $f(x)$ .
- 2** Draw a real value  $y$  uniformly from  $(0, f(x(t)))$ , thereby defining a horizontal “slice” as  $S = \{x: y < f(x)\}$ .
- 3** Find an interval  $I = (L, R)$  around  $x(t)$  that contains all, or much of the “slice”  $S$ .
- 4** Draw the new point  $x(t+1)$  within this interval.
- 5** Increment  $t \rightarrow t+1$  and repeat steps 2 through 4 until you get the desired number of samples.

Slice sampling can generate random numbers from a distribution with an arbitrary form of the density function, provided that an efficient numerical procedure is available to find the interval  $I = (L, R)$ , which is the “slice” of the density.

Generate random numbers using the slice sampling method with the `slicesample` function.

## Generating Quasi-Random Numbers

In this section...
“Quasi-Random Sequences” on page 6-15
“Quasi-Random Point Sets” on page 6-16
“Quasi-Random Streams” on page 6-24

### Quasi-Random Sequences

*Quasi-random number generators* (QRNGs) produce highly uniform samples of the unit hypercube. QRNGs minimize the *discrepancy* between the distribution of generated points and a distribution with equal proportions of points in each sub-cube of a uniform partition of the hypercube. As a result, QRNGs systematically fill the “holes” in any initial segment of the generated quasi-random sequence.

Unlike the pseudorandom sequences described in “Common Generation Methods” on page 6-5, quasi-random sequences fail many statistical tests for randomness. Approximating true randomness, however, is not their goal. Quasi-random sequences seek to fill space uniformly, and to do so in such a way that initial segments approximate this behavior up to a specified density.

QRNG applications include:

- **Quasi-Monte Carlo (QMC) integration.** Monte Carlo techniques are often used to evaluate difficult, multi-dimensional integrals without a closed-form solution. QMC uses quasi-random sequences to improve the convergence properties of these techniques.
- **Space-filling experimental designs.** In many experimental settings, taking measurements at every factor setting is expensive or infeasible. Quasi-random sequences provide efficient, uniform sampling of the design space.
- **Global optimization.** Optimization algorithms typically find a local optimum in the neighborhood of an initial value. By using a quasi-random sequence of initial values, searches for global optima uniformly sample the basins of attraction of all local minima.

**Example: Using Scramble, Leap, and Skip**

Imagine a simple 1-D sequence that produces the integers from 1 to 10. This is the basic sequence and the first three points are [1,2,3]:

1 2 3 4 5 6 7 8 9 10

Now look at how Scramble, Leap, and Skip work together:

- **Scramble** — Scrambling shuffles the points in one of several different ways. In this example, assume a scramble turns the sequence into 1,3,5,7,9,2,4,6,8,10. The first three points are now [1,3,5]:

1 3 5 7 9 2 4 6 8 10

- **Skip** — A Skip value specifies the number of initial points to ignore. In this example, set the Skip value to 2. The sequence is now 5,7,9,2,4,6,8,10 and the first three points are [5,7,9]:

1 3 5 7 9 2 4 6 8 10

- **Leap** — A Leap value specifies the number of points to ignore for each one you take. Continuing the example with the Skip set to 2, if you set the Leap to 1, the sequence uses every other point. In this example, the sequence is now 5,9,4,8 and the first three points are [5,9,4]:

1 3 5 7 9 2 4 6 8 10

**Quasi-Random Point Sets**

Statistics Toolbox functions support these quasi-random sequences:

- **Halton sequences.** Produced by the `haltonset` function. These sequences use different prime bases to form successively finer uniform partitions of the unit interval in each dimension.

- **Sobol sequences.** Produced by the `sobolset` function. These sequences use a base of 2 to form successively finer uniform partitions of the unit interval, and then reorder the coordinates in each dimension.
- **Latin hypercube sequences.** Produced by the `lhsdesign` function. Though not quasi-random in the sense of minimizing discrepancy, these sequences nevertheless produce sparse uniform samples useful in experimental designs.

Quasi-random sequences are functions from the positive integers to the unit hypercube. To be useful in application, an initial *point set* of a sequence must be generated. Point sets are matrices of size  $n$ -by- $d$ , where  $n$  is the number of points and  $d$  is the dimension of the hypercube being sampled. The functions `haltonset` and `sobolset` construct point sets with properties of a specified quasi-random sequence. Initial segments of the point sets are generated by the `net` method of the `grandset` class (parent class of the `haltonset` class and `sobolset` class), but points can be generated and accessed more generally using parenthesis indexing.

Because of the way in which quasi-random sequences are generated, they may contain undesirable correlations, especially in their initial segments, and especially in higher dimensions. To address this issue, quasi-random point sets often *skip*, *leap* over, or *scramble* values in a sequence. The `haltonset` and `sobolset` functions allow you to specify both a `Skip` and a `Leap` property of a quasi-random sequence, and the `scramble` method of the `grandset` class allows you apply a variety of scrambling techniques. Scrambling reduces correlations while also improving uniformity.

## Generate a Quasi-Random Point Set

This example shows how to use `haltonset` to construct a 2-D Halton quasi-random point set.

Create a `haltonset` object `p`, that skips the first 1000 values of the sequence and then retains every 101st point.

```
rng('default'); % For reproducibility
p = haltonset(2, 'Skip', 1e3, 'Leap', 1e2)
```

```
p =
```

```
Halton point set in 2 dimensions (89180190640991 points)
```

```
Properties:
```

```
    Skip : 1000  
    Leap : 100  
    ScrambleMethod : none
```

The object `p` encapsulates properties of the specified quasi-random sequence. The point set is finite, with a length determined by the `Skip` and `Leap` properties and by limits on the size of point set indices.

Use `scramble` to apply reverse-radix scrambling.

```
p = scramble(p, 'RR2')
```

```
p =
```

```
Halton point set in 2 dimensions (89180190640991 points)
```

```
Properties:
```

```
    Skip : 1000  
    Leap : 100  
    ScrambleMethod : RR2
```

Use `net` to generate the first 500 points.

```
X0 = net(p,500);
```

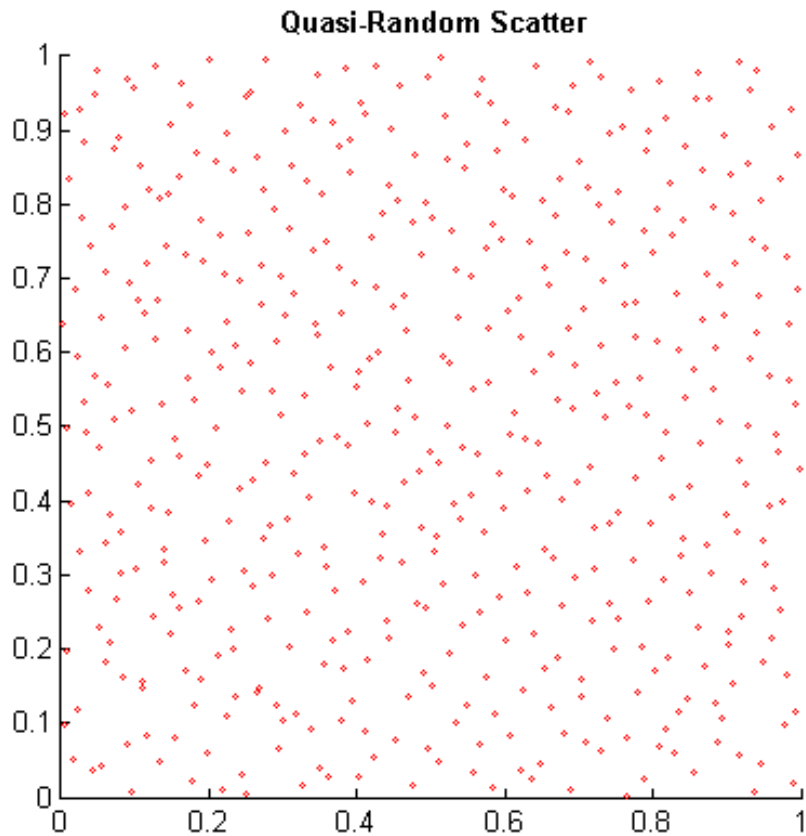
This is equivalent to

```
X0 = p(1:500,:);
```

Values of the point set `X0` are not generated and stored in memory until you access `p` using `net` or parenthesis indexing.

To appreciate the nature of quasi-random numbers, create a scatter plot of the two dimensions in  $X_0$ .

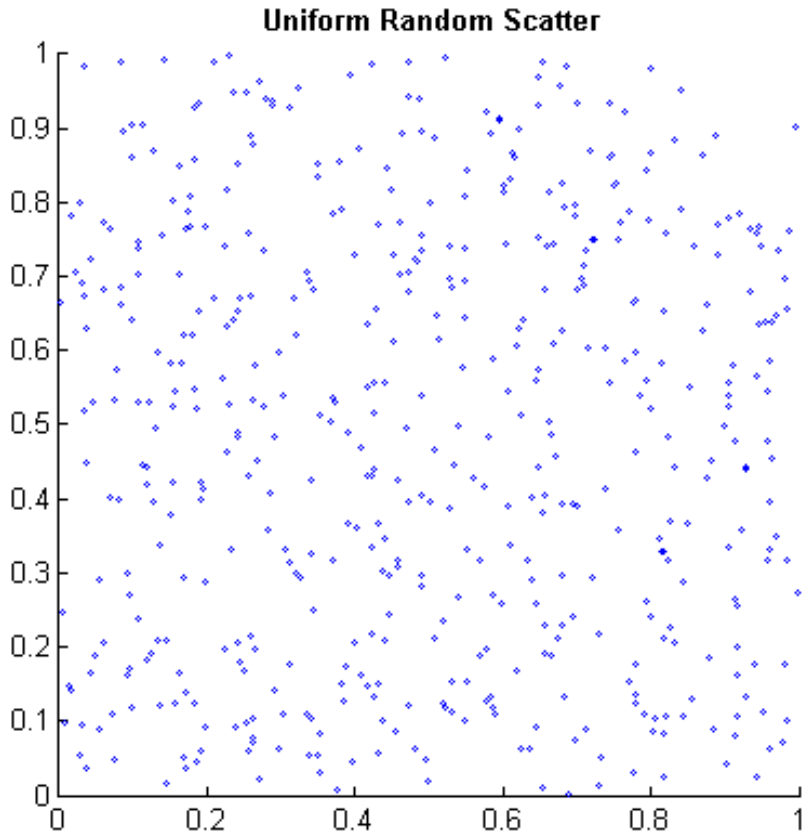
```
scatter(X0(:,1),X0(:,2),5,'r')  
axis square  
title('\bf Quasi-Random Scatter')
```



Compare this to a scatter of uniform pseudorandom numbers generated by the `rand` function.

```
X = rand(500,2);
```

```
scatter(X(:,1),X(:,2),5,'b')
axis square
title('{\bf Uniform Random Scatter}')
```



The quasi-random scatter appears more uniform, avoiding the clumping in the pseudorandom scatter.

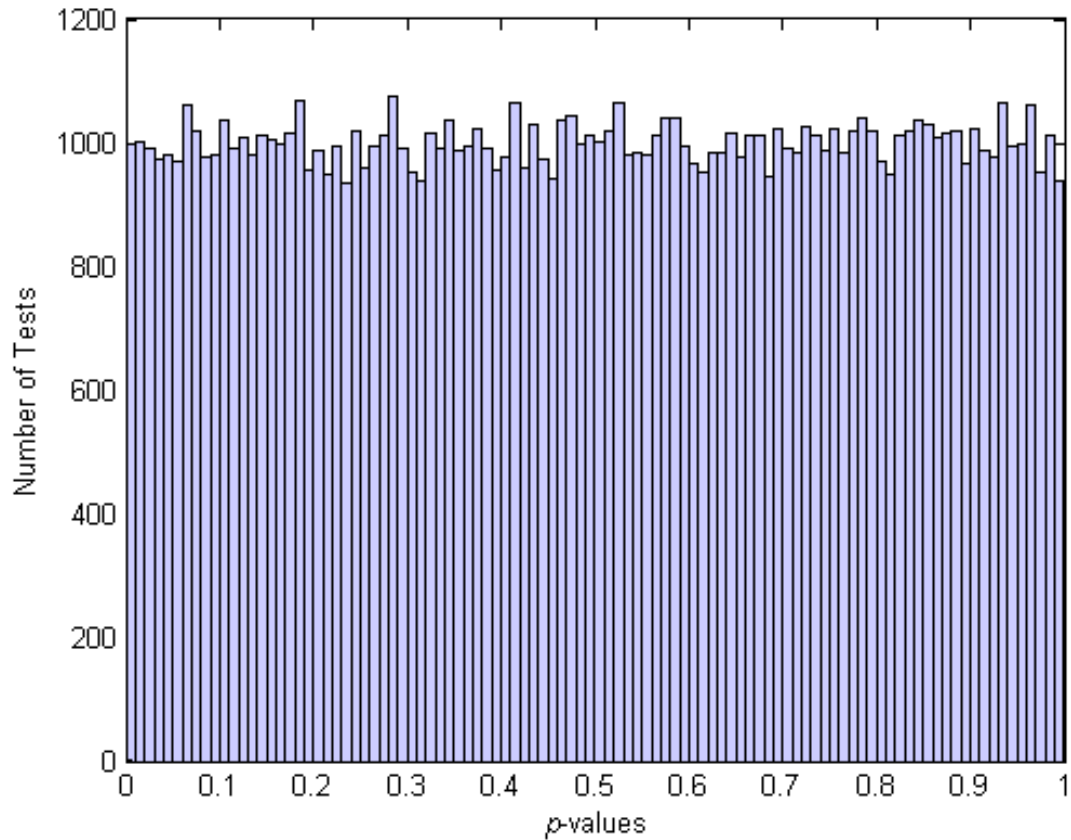
In a statistical sense, quasi-random numbers are too uniform to pass traditional tests of randomness. For example, a Kolmogorov-Smirnov test, performed by `kstest`, is used to assess whether or not a point set has a



uniform random distribution. When performed repeatedly on uniform pseudorandom samples, such as those generated by `rand`, the test produces a uniform distribution of  $p$ -values.

```
nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
    x = rand(sampSize,1);
    [h,pval] = kstest(x,[x,x]);
    PVALS(test) = pval;
end

hist(PVALS,100)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('\it p}-values')
ylabel('Number of Tests')
```

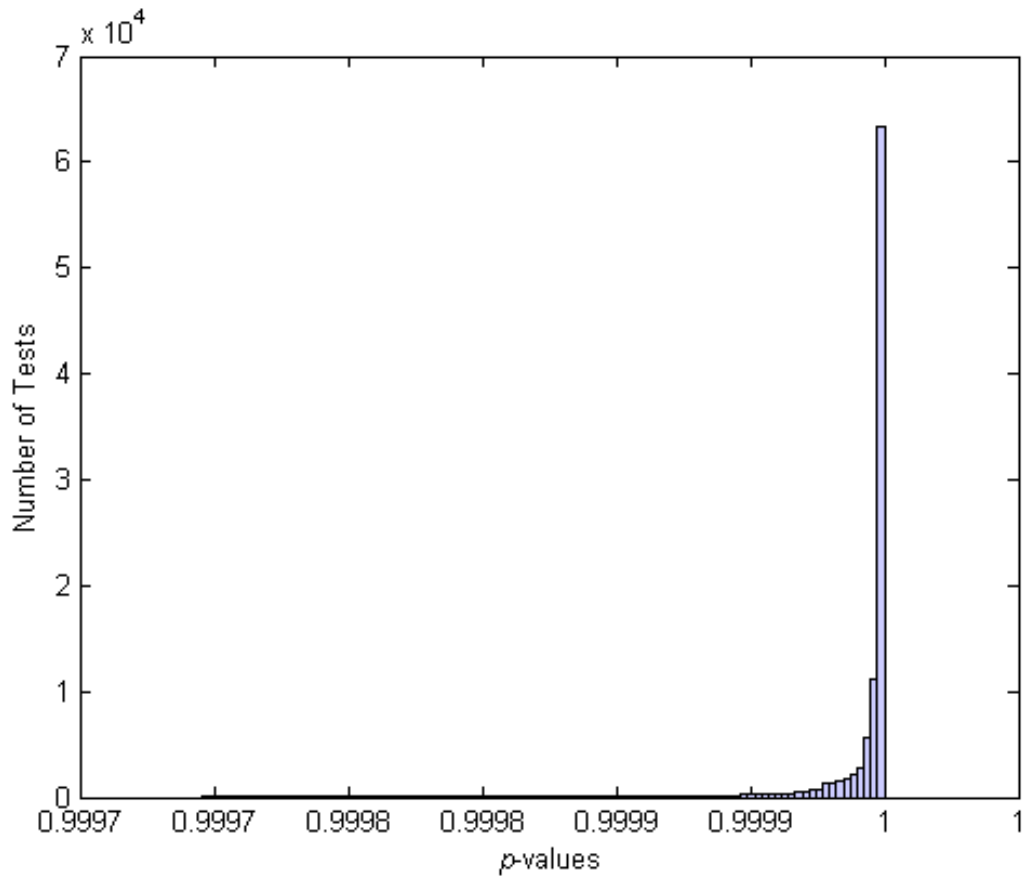


The results are quite different when the test is performed repeatedly on uniform quasi-random samples.

```
p = haltonset(1, 'Skip', 1e3, 'Leap', 1e2);  
p = scramble(p, 'RR2');
```

```
nTests = 1e5;  
sampSize = 50;  
PVALS = zeros(nTests, 1);  
for test = 1:nTests
```

```
x = p(test:test+(sampSize-1),:);  
[h,pval] = kstest(x,[x,x]);  
PVALS(test) = pval;  
end  
  
hist(PVALS,100)  
set(get(gca,'Children'),'FaceColor',[.8 .8 1])  
xlabel('\it p}-values')  
ylabel('Number of Tests')
```



Small  $p$ -values call into question the null hypothesis that the data are uniformly distributed. If the hypothesis is true, about 5% of the  $p$ -values are expected to fall below 0.05. The results are remarkably consistent in their failure to challenge the hypothesis.

## Quasi-Random Streams

Quasi-random *streams*, produced by the `grandstream` function, are used to generate sequential quasi-random outputs, rather than point sets of a specific size. Streams are used like pseudoRNGS, such as `rand`, when client applications require a source of quasi-random numbers of indefinite size that can be accessed intermittently. Properties of a quasi-random stream, such as its type (Halton or Sobol), dimension, skip, leap, and scramble, are set when the stream is constructed.

In implementation, quasi-random streams are essentially very large quasi-random point sets, though they are accessed differently. The *state* of a quasi-random stream is the scalar index of the next point to be taken from the stream. Use the `grand` method of the `grandstream` class to generate points from the stream, starting from the current state. Use the `reset` method to reset the state to 1. Unlike point sets, streams do not support parenthesis indexing.

## Generate a Quasi-Random Stream

This example shows how to generate samples from a quasi-random point set.

Use `haltonset` to create a quasi-random point set `p`, then repeatedly increment the index into the point set `test` to generate different samples.

```
p = haltonset(1, 'Skip', 1e3, 'Leap', 1e2);
p = scramble(p, 'RR2');

nTests = 1e5;
sampSize = 50;
PVALS = zeros(nTests,1);
for test = 1:nTests
    x = p(test:test+(sampSize-1),:);
    [h,pval] = kstest(x,[x,x]);
    PVALS(test) = pval;
```

```
end
```

The same results are obtained by using `grandstream` to construct a quasi-random stream `q` based on the point set `p` and letting the stream take care of increments to the index.

```
p = haltonset(1,'Skip',1e3,'Leap',1e2);  
p = scramble(p,'RR2');  
q = grandstream(p);
```

```
nTests = 1e5;  
sampSize = 50;  
PVALS = zeros(nTests,1);  
for test = 1:nTests  
    X = grand(q,sampSize);  
    [h,pval] = kstest(X,[X,X]);  
    PVALS(test) = pval;  
end
```

## Generating Data Using Flexible Families of Distributions

### In this section...

“Pearson and Johnson Systems” on page 6-26

“Generating Data Using the Pearson System” on page 6-28

“Generating Data Using the Johnson System” on page 6-30

### Pearson and Johnson Systems

As described in “Working with Probability Distributions” on page 5-3, choosing an appropriate parametric family of distributions to model your data can be based on *a priori* or *a posteriori* knowledge of the data-producing process, but the choice is often difficult. The *Pearson and Johnson systems* can make such a choice unnecessary. Each system is a flexible parametric family of distributions that includes a wide range of distribution shapes, and it is often possible to find a distribution within one of these two systems that provides a good match to your data.

#### Data Input

The following parameters define each member of the Pearson and Johnson systems

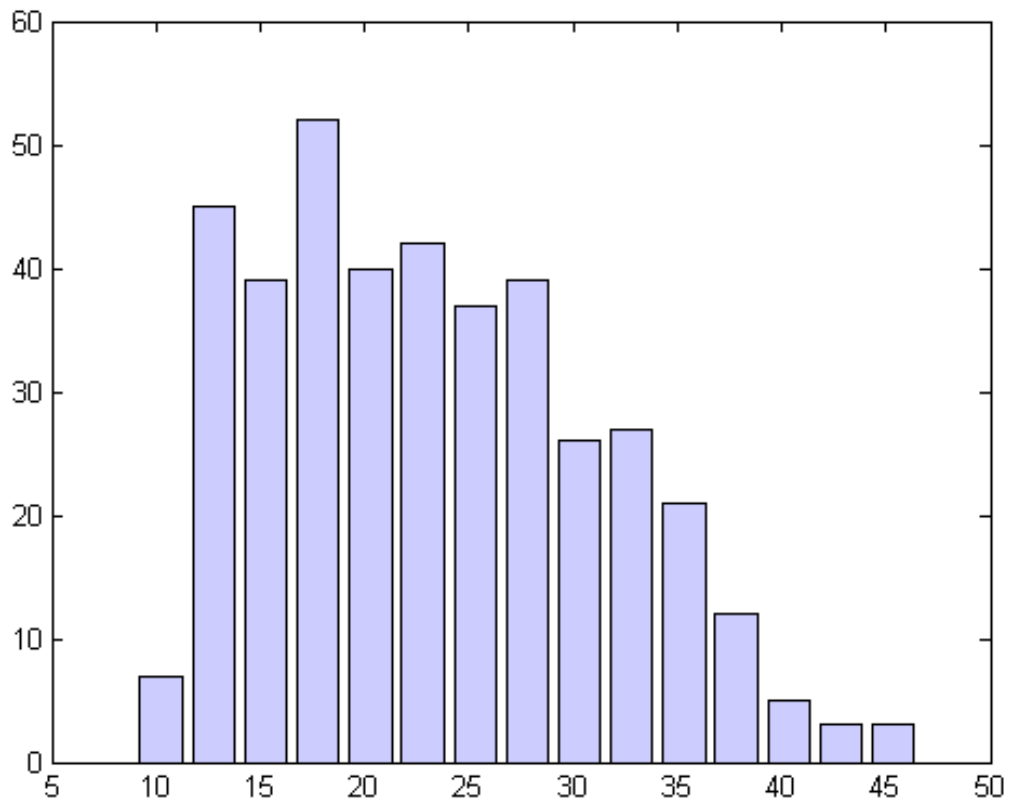
- Mean — Estimated by mean
- Standard deviation — Estimated by std
- Skewness — Estimated by skewness
- Kurtosis — Estimated by kurtosis

These statistics can also be computed with the `moment` function. The Johnson system, while based on these four parameters, is more naturally described using quantiles, estimated by the `quantile` function.

The Statistics Toolbox functions `pearsrnd` and `johnsrnd` take input arguments defining a distribution (parameters or quantiles, respectively) and return the type and the coefficients of the distribution in the corresponding system. Both functions also generate random numbers from the specified distribution.

As an example, load the data in `carbig.mat`, which includes a variable `MPG` containing measurements of the gas mileage for each car.

```
load carbig
MPG = MPG(~isnan(MPG));
[n,x] = hist(MPG,15);
bar(x,n)
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
```



The following two sections model the distribution with members of the Pearson and Johnson systems, respectively.

## Generating Data Using the Pearson System

The statistician Karl Pearson devised a system, or family, of distributions that includes a unique distribution corresponding to every valid combination of mean, standard deviation, skewness, and kurtosis. If you compute sample values for each of these moments from data, it is easy to find the distribution in the Pearson system that matches these four moments and to generate a random sample.

The Pearson system embeds seven basic types of distribution together in a single parametric framework. It includes common distributions such as the normal and  $t$  distributions, simple transformations of standard distributions such as a shifted and scaled beta distribution and the inverse gamma distribution, and one distribution—the Type IV—that is not a simple transformation of any standard distribution.

For a given set of moments, there are distributions that are not in the system that also have those same first four moments, and the distribution in the Pearson system may not be a good match to your data, particularly if the data are multimodal. But the system does cover a wide range of distribution shapes, including both symmetric and skewed distributions.

To generate a sample from the Pearson distribution that closely matches the MPG data, simply compute the four sample moments and treat those as distribution parameters.

```
moments = {mean(MPG),std(MPG),skewness(MPG),kurtosis(MPG)};
rng('default'); % For reproducibility
[r,type] = pearsrnd(moments{:},10000,1);
```

The optional second output from `pearsrnd` indicates which type of distribution within the Pearson system matches the combination of moments.

```
type
```

```
type =
```

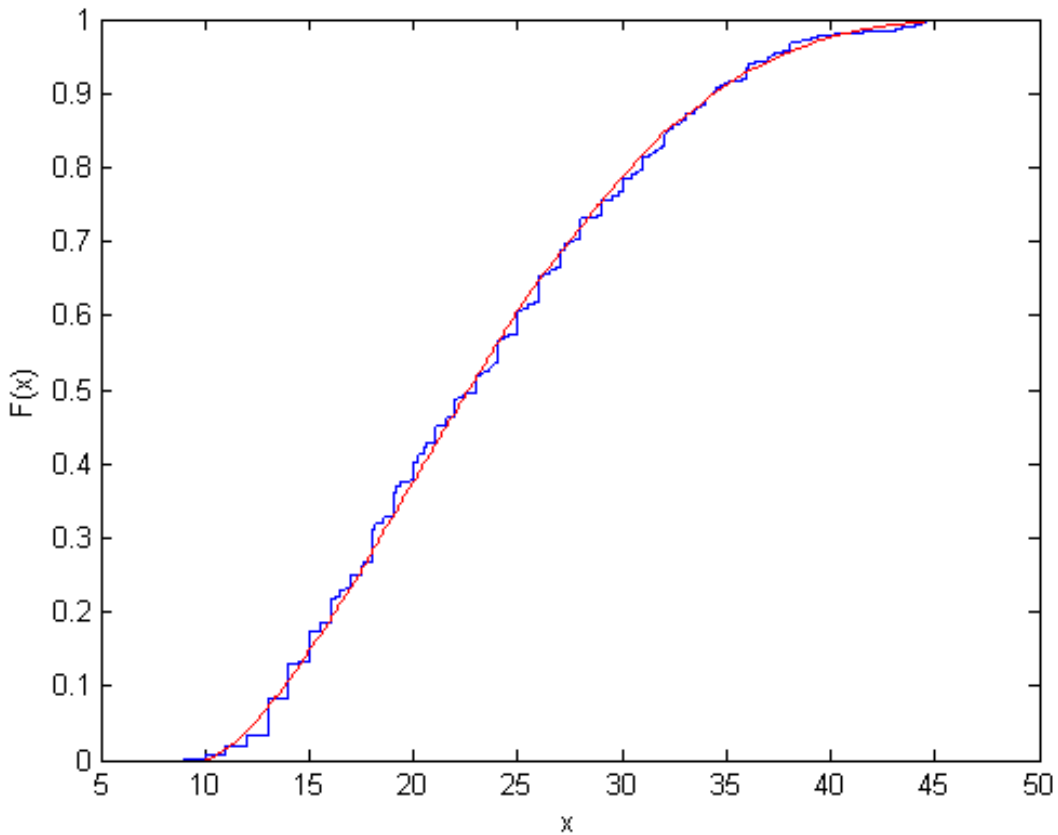
```
1
```



In this case, `pearsrnd` has determined that the data are best described with a Type I Pearson distribution, which is a shifted, scaled beta distribution.

Verify that the sample resembles the original data by overlaying the empirical cumulative distribution functions.

```
ecdf(MPG);  
[Fi,xi] = ecdf(r);  
hold on;  
stairs(xi,Fi,'r');  
hold off
```



## Generating Data Using the Johnson System

Statistician Norman Johnson devised a different system of distributions that also includes a unique distribution for every valid combination of mean, standard deviation, skewness, and kurtosis. However, since it is more natural to describe distributions in the Johnson system using quantiles, working with this system is different than working with the Pearson system.

The Johnson system is based on three possible transformations of a normal random variable, plus the identity transformation. The three nontrivial cases are known as SL, SU, and SB, corresponding to exponential, logistic, and hyperbolic sine transformations. All three can be written as

$$X = \gamma + \delta \cdot \Gamma\left(\frac{(Z-\xi)}{\lambda}\right)$$

where  $Z$  is a standard normal random variable,  $\Gamma$  is the transformation, and  $\gamma$ ,  $\delta$ ,  $\xi$ , and  $\lambda$  are scale and location parameters. The fourth case, SN, is the identity transformation.

To generate a sample from the Johnson distribution that matches the MPG data, first define the four quantiles to which the four evenly spaced standard normal quantiles of -1.5, -0.5, 0.5, and 1.5 should be transformed. That is, you compute the sample quantiles of the data for the cumulative probabilities of 0.067, 0.309, 0.691, and 0.933.

```
probs = normcdf([-1.5 -0.5 0.5 1.5])

probs =
    0.0668    0.3085    0.6915    0.9332

quantiles = quantile(MPG,probs)

quantiles =
    13.0000    18.0000    27.2000    36.0000
```

Then treat those quantiles as distribution parameters.

```
[r1,type] = johnsrnd(quantiles,10000,1);
```

The optional second output from `johnsrnd` indicates which type of distribution within the Johnson system matches the quantiles.

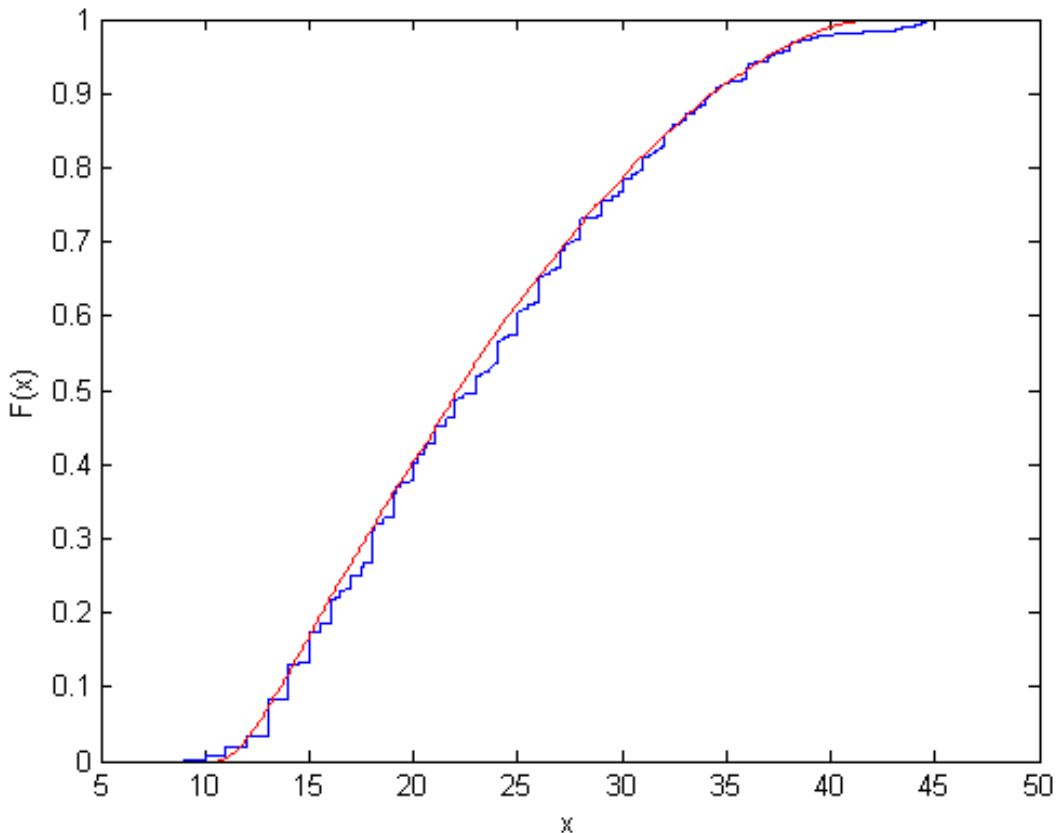
```
type
```

```
type =
```

```
SB
```

You can verify that the sample resembles the original data by overlaying the empirical cumulative distribution functions.

```
ecdf(MPG);  
[Fi,xi] = ecdf(r1);  
hold on;  
stairs(xi,Fi,'r');  
hold off
```



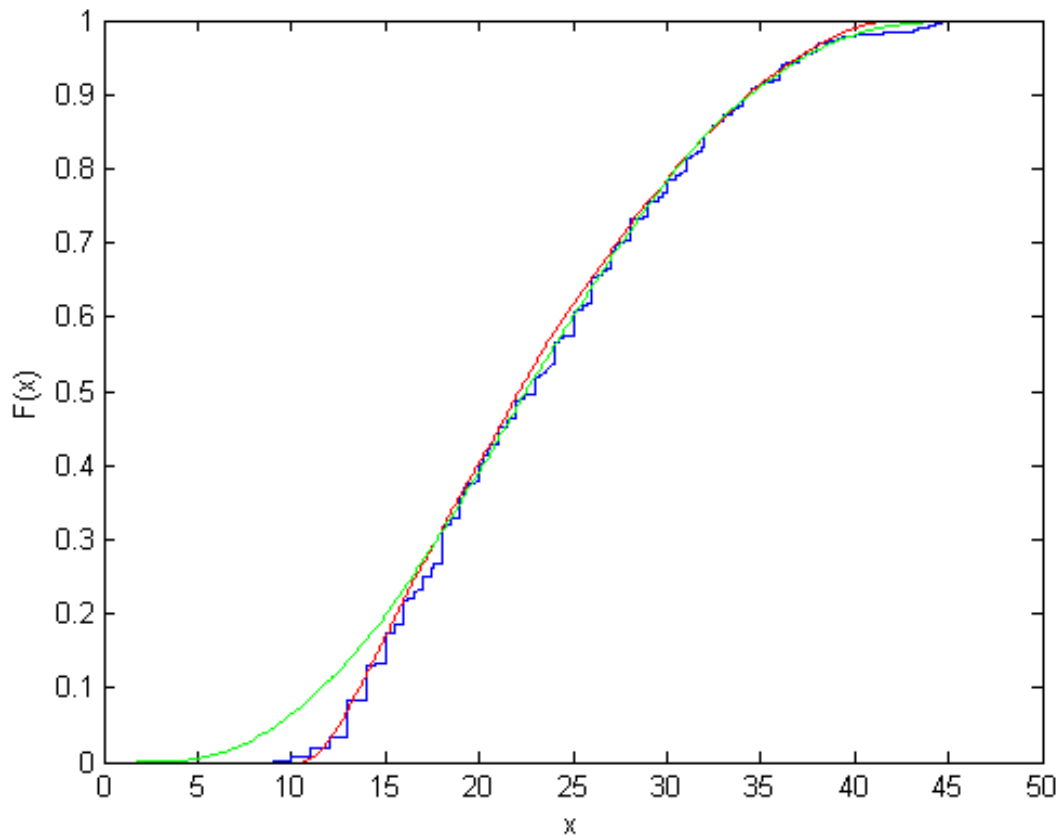
In some applications, it may be important to match the quantiles better in some regions of the data than in others. To do that, specify four evenly spaced standard normal quantiles at which you want to match the data, instead of the default -1.5, -0.5, 0.5, and 1.5. For example, you might care more about matching the data in the right tail than in the left, and so you specify standard normal quantiles that emphasizes the right tail.

```
qnorm = [-.5 .25 1 1.75];  
probs = normcdf(qnorm);  
qemp = quantile(MPG,probs);
```

```
r2 = johnsrnd([qnorm; qemp],10000,1);
```

However, while the new sample matches the original data better in the right tail, it matches much worse in the left tail.

```
[Fj,xj] = ecdf(r2);  
hold on;  
stairs(xj,Fj,'g');  
hold off
```





# Hypothesis Tests

---

- “Introduction to Hypothesis Tests” on page 7-2
- “Hypothesis Test Terminology” on page 7-3
- “Hypothesis Test Assumptions” on page 7-5
- “Hypothesis Testing” on page 7-7
- “Available Hypothesis Tests” on page 7-14

## Introduction to Hypothesis Tests

Hypothesis testing is a common method of drawing inferences about a population based on statistical evidence from a sample.

As an example, suppose someone says that at a certain time in the state of Massachusetts the average price of a gallon of regular unleaded gas was \$1.15. How could you determine the truth of the statement? You could try to find prices at every gas station in the state at the time. That approach would be definitive, but it could be time-consuming, costly, or even impossible.

A simpler approach would be to find prices at a small number of randomly selected gas stations around the state, and then compute the sample average.

Sample averages differ from one another due to chance variability in the selection process. Suppose your sample average comes out to be \$1.18. Is the \$0.03 difference an artifact of random sampling or significant evidence that the average price of a gallon of gas was in fact greater than \$1.15? Hypothesis testing is a statistical method for making such decisions.



## Hypothesis Test Terminology

All hypothesis tests share the same basic terminology and structure.

- A *null hypothesis* is an assertion about a population that you would like to test. It is “null” in the sense that it often represents a status quo belief, such as the absence of a characteristic or the lack of an effect. It may be formalized by asserting that a population parameter, or a combination of population parameters, has a certain value. In the example given in the “Introduction to Hypothesis Tests” on page 7-2, the null hypothesis would be that the average price of gas across the state was \$1.15. This is written  $H_0: \mu = 1.15$ .
- An *alternative hypothesis* is a contrasting assertion about the population that can be tested against the null hypothesis. In the example given in the “Introduction to Hypothesis Tests” on page 7-2, possible alternative hypotheses are:
  - $H_1: \mu \neq 1.15$  — State average was different from \$1.15 (two-tailed test)
  - $H_1: \mu > 1.15$  — State average was greater than \$1.15 (right-tail test)
  - $H_1: \mu < 1.15$  — State average was less than \$1.15 (left-tail test)
- To conduct a hypothesis test, a random sample from the population is collected and a relevant *test statistic* is computed to summarize the sample. This statistic varies with the type of test, but its distribution under the null hypothesis must be known (or assumed).
- The *p value* of a test is the probability, under the null hypothesis, of obtaining a value of the test statistic as extreme or more extreme than the value computed from the sample.
- The *significance level* of a test is a threshold of probability  $\alpha$  agreed to before the test is conducted. A typical value of  $\alpha$  is 0.05. If the *p* value of a test is less than  $\alpha$ , the test rejects the null hypothesis. If the *p* value is greater than  $\alpha$ , there is insufficient evidence to reject the null hypothesis. Note that lack of evidence for rejecting the null hypothesis is not evidence for accepting the null hypothesis. Also note that substantive “significance” of an alternative cannot be inferred from the statistical significance of a test.
- The significance level  $\alpha$  can be interpreted as the probability of rejecting the null hypothesis when it is actually true—a *type I error*. The distribution of the test statistic under the null hypothesis determines the probability

$\alpha$  of a type I error. Even if the null hypothesis is not rejected, it may still be false—a *type II error*. The distribution of the test statistic under the alternative hypothesis determines the probability  $\beta$  of a type II error. Type II errors are often due to small sample sizes. The *power* of a test,  $1 - \beta$ , is the probability of correctly rejecting a false null hypothesis.

- Results of hypothesis tests are often communicated with a *confidence interval*. A confidence interval is an estimated range of values with a specified probability of containing the true population value of a parameter. Upper and lower bounds for confidence intervals are computed from the sample estimate of the parameter and the known (or assumed) sampling distribution of the estimator. A typical assumption is that estimates will be normally distributed with repeated sampling (as dictated by the Central Limit Theorem). Wider confidence intervals correspond to poor estimates (smaller samples); narrow intervals correspond to better estimates (larger samples). If the null hypothesis asserts the value of a population parameter, the test rejects the null hypothesis when the hypothesized value lies outside the computed confidence interval for the parameter.

## Hypothesis Test Assumptions

Different hypothesis tests make different assumptions about the distribution of the random variable being sampled in the data. These assumptions must be considered when choosing a test and when interpreting the results.

For example, the  $z$ -test (`ztest`) and the  $t$ -test (`ttest`) both assume that the data are independently sampled from a normal distribution. Statistics Toolbox functions are available for testing this assumption, such as `chi2gof`, `jbtest`, `lillietest`, and `normplot`.

Both the  $z$ -test and the  $t$ -test are relatively robust with respect to departures from this assumption, so long as the sample size  $n$  is large enough. Both tests compute a sample mean  $\bar{x}$ , which, by the Central Limit Theorem, has an approximately normal sampling distribution with mean equal to the population mean  $\mu$ , regardless of the population distribution being sampled.

The difference between the  $z$ -test and the  $t$ -test is in the assumption of the standard deviation  $\sigma$  of the underlying normal distribution. A  $z$ -test assumes that  $\sigma$  is known; a  $t$ -test does not. As a result, a  $t$ -test must compute an estimate  $s$  of the standard deviation from the sample.

Test statistics for the  $z$ -test and the  $t$ -test are, respectively,

$$z = \frac{\bar{x} - \mu}{\sigma / \sqrt{n}}$$
$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}}$$

Under the null hypothesis that the population is distributed with mean  $\mu$ , the  $z$ -statistic has a standard normal distribution,  $N(0,1)$ . Under the same null hypothesis, the  $t$ -statistic has Student's  $t$  distribution with  $n - 1$  degrees of freedom. For small sample sizes, Student's  $t$  distribution is flatter and wider than  $N(0,1)$ , compensating for the decreased confidence in the estimate  $s$ . As sample size increases, however, Student's  $t$  distribution approaches the standard normal distribution, and the two tests become essentially equivalent.

Knowing the distribution of the test statistic under the null hypothesis allows for accurate calculation of  $p$ -values. Interpreting  $p$ -values in the context of the test assumptions allows for critical analysis of test results.

Assumptions underlying Statistics Toolbox hypothesis tests are given in the reference pages for implementing functions.

## Hypothesis Testing

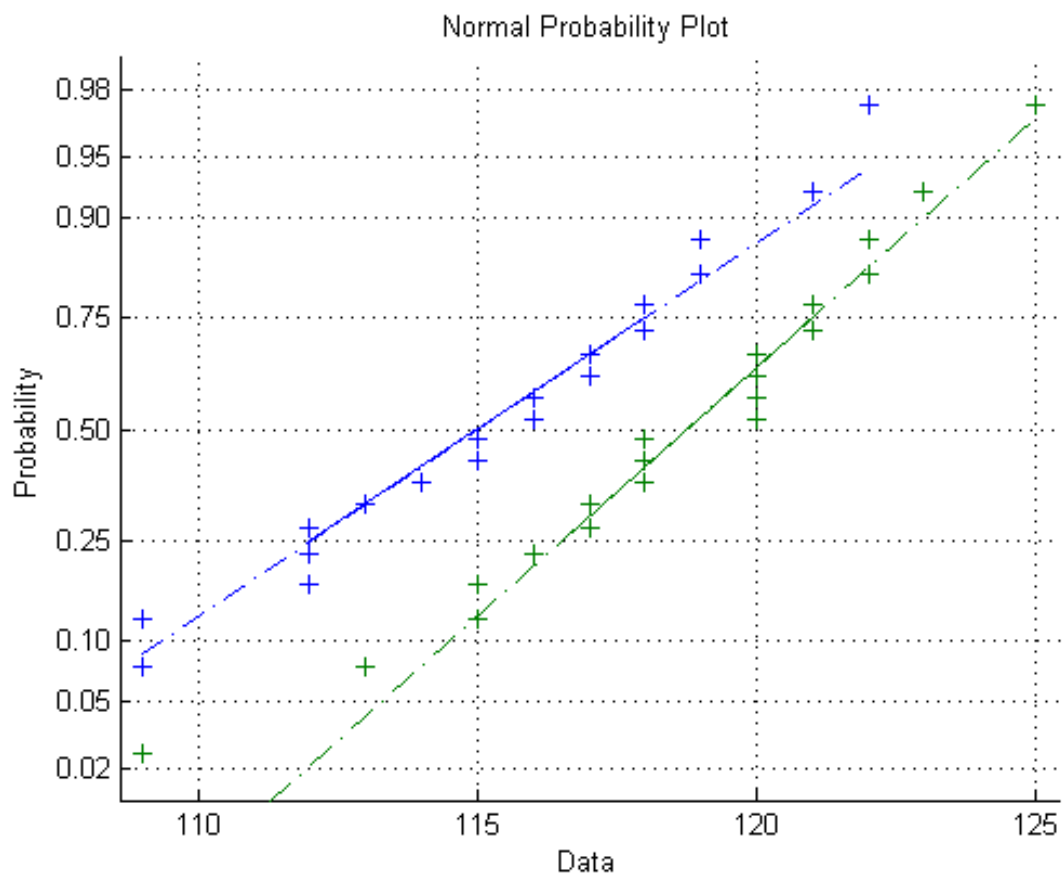
This example shows how to use hypothesis testing to analyze gas prices measured across the state of Massachusetts during two separate months.

This example uses the gas price data in the file `gas.mat`. The file contains two random samples of prices for a gallon of gas around the state of Massachusetts in 1993. The first sample, `price1`, contains 20 random observations around the state on a single day in January. The second sample, `price2`, contains 20 random observations around the state one month later.

```
load gas
prices = [price1 price2];
```

As a first step, you might want to test the assumption that the samples come from normal distributions. A normal probability plot gives a quick idea.

```
normplot(prices)
```



Both scatters approximately follow straight lines through the first and third quartiles of the samples, indicating approximate normal distributions. The February sample (the right-hand line) shows a slight departure from normality in the lower tail. A shift in the mean from January to February is evident. A hypothesis test is used to quantify the test of normality. Since each sample is relatively small, a Lilliefors test is recommended.

```
lillietest(price1)  
lillietest(price2)
```

```
ans =  
0
```

```
ans =  
0
```

The default significance level of `lillietest` is 5%. The logical 0 returned by each test indicates a failure to reject the null hypothesis that the samples are normally distributed. This failure may reflect normality in the population or it may reflect a lack of strong evidence against the null hypothesis due to the small sample size.

Now compute the sample means.

```
sample_means = mean(prices)
```

```
sample_means =  
115.1500 118.5000
```

You might want to test the null hypothesis that the mean price across the state on the day of the January sample was \$1.15. If you know that the standard deviation in prices across the state has historically, and consistently, been \$0.04, then a *z*-test is appropriate.

```
[h,pvalue,ci] = ztest(price1/100,1.15,0.04)
```

```
h =  
0
```

```
pvalue =  
  
    0.8668  
  
ci =  
  
    1.1340  
    1.1690
```

The logical output  $h = 0$  indicates a failure to reject the null hypothesis at the default significance level of 5%. This is a consequence of the high probability under the null hypothesis, indicated by the  $p$  value, of observing a value as extreme or more extreme of the  $z$ -statistic computed from the sample. The 95% confidence interval on the mean [1.1340 1.1690] includes the hypothesized population mean of \$1.15.

Does the later sample offer stronger evidence for rejecting a null hypothesis of a state-wide average price of \$1.15 in February? The shift shown in the probability plot and the difference in the computed sample means suggest this. The shift might indicate a significant fluctuation in the market, raising questions about the validity of using the historical standard deviation. If a known standard deviation cannot be assumed, a  $t$ -test is more appropriate.

```
[h,pvalue,ci] = ttest(price2/100,1.15)
```

```
h =  
  
    1  
  
pvalue =  
  
    4.9517e-04  
  
ci =
```



```
1.1675
1.2025
```

The logical output  $h = 1$  indicates a rejection of the null hypothesis at the default significance level of 5%. In this case, the 95% confidence interval on the mean does not include the hypothesized population mean of \$1.15.

You might want to investigate the shift in prices a little more closely. The function `ttest2` tests if two independent samples come from normal distributions with equal but unknown standard deviations and the same mean, against the alternative that the means are unequal.

```
[h,sig,ci] = ttest2(price1,price2)
```

```
h =
```

```
1
```

```
sig =
```

```
0.0083
```

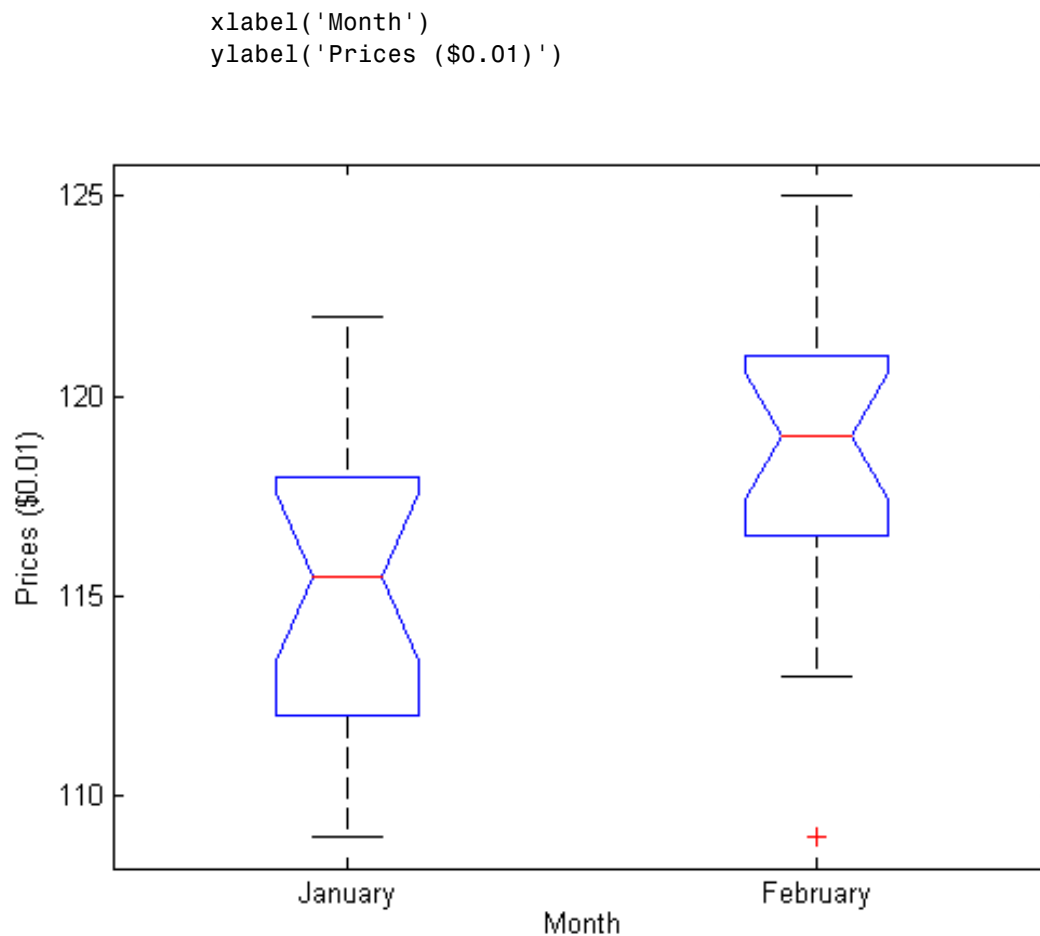
```
ci =
```

```
-5.7845
```

```
-0.9155
```

The null hypothesis is rejected at the default 5% significance level, and the confidence interval on the difference of means does not include the hypothesized value of 0. A notched box plot is another way to visualize the shift.

```
boxplot(prices,1)
set(gca,'XTick',[1 2])
set(gca,'XtickLabel',{'January','February'})
```



The plot displays the distribution of the samples around their medians. The heights of the notches in each box are computed so that the side-by-side boxes have nonoverlapping notches when their medians are different at a default 5% significance level. The computation is based on an assumption of normality in the data, but the comparison is reasonably robust for other distributions. The side-by-side plots provide a kind of visual hypothesis test,

comparing medians rather than means. The plot above appears to barely reject the null hypothesis of equal medians.

The nonparametric Wilcoxon rank sum test, implemented by the function `ranksum`, can be used to quantify the test of equal medians. It tests if two independent samples come from identical continuous (not necessarily normal) distributions with equal medians, against the alternative that they do not have equal medians.

```
[p,h] = ranksum(price1,price2)
```

```
p =
```

```
0.0095
```

```
h =
```

```
1
```

The test rejects the null hypothesis of equal medians at the default 5% significance level.

## Available Hypothesis Tests

Function	Description
ansaribradley	Ansari-Bradley test. Tests if two independent samples come from the same distribution, against the alternative that they come from distributions that have the same median and shape but different variances.
barttest	Bartlett's test. Tests if the variances of the data values along each principal component are equal, against the alternative that the variances are not all equal.
chi2gof	Chi-square goodness-of-fit test. Tests if a sample comes from a specified distribution, against the alternative that it does not come from that distribution.
dwtest	Durbin-Watson test. Tests if the residuals from a linear regression are uncorrelated, against the alternative that there is autocorrelation among them.
friedman	Friedman's test. Tests if the column effects in a two-way layout are all the same, against the alternative that the column effects are not all the same.
jbtest	Jarque-Bera test. Tests if a sample comes from a normal distribution with unknown mean and variance, against the alternative that it does not come from a normal distribution.
kruskalwallis	Kruskal-Wallis test. Tests if multiple samples are all drawn from the same populations (or equivalently, from different populations with the same distribution), against the alternative that they are not all drawn from the same population.
kstest	One-sample Kolmogorov-Smirnov test. Tests if a sample comes from a continuous distribution with specified parameters, against the alternative that it does not come from that distribution.

<b>Function</b>	<b>Description</b>
<code>kstest2</code>	Two-sample Kolmogorov-Smirnov test. Tests if two samples come from the same continuous distribution, against the alternative that they do not come from the same distribution.
<code>lillietest</code>	Lilliefors test. Tests if a sample comes from a distribution in the normal family, against the alternative that it does not come from a normal distribution.
<code>linhypstest</code>	Linear hypothesis test. Tests if $H^*b = c$ for parameter estimates $b$ with estimated covariance $H$ and specified $c$ , against the alternative that $H^*b \neq c$ .
<code>ranksum</code>	Wilcoxon rank sum test. Tests if two independent samples come from identical continuous distributions with equal medians, against the alternative that they do not have equal medians.
<code>runstest</code>	Runs test. Tests if a sequence of values comes in random order, against the alternative that the ordering is not random.
<code>signrank</code>	One-sample or paired-sample Wilcoxon signed rank test. Tests if a sample comes from a continuous distribution symmetric about a specified median, against the alternative that it does not have that median.
<code>signtest</code>	One-sample or paired-sample sign test. Tests if a sample comes from an arbitrary continuous distribution with a specified median, against the alternative that it does not have that median.
<code>ttest</code>	One-sample or paired-sample $t$ -test. Tests if a sample comes from a normal distribution with unknown variance and a specified mean, against the alternative that it does not have that mean.

Function	Description
<code>ttest2</code>	Two-sample $t$ -test. Tests if two independent samples come from normal distributions with unknown but equal (or, optionally, unequal) variances and the same mean, against the alternative that the means are unequal.
<code>vartest</code>	One-sample chi-square variance test. Tests if a sample comes from a normal distribution with specified variance, against the alternative that it comes from a normal distribution with a different variance.
<code>vartest2</code>	Two-sample $F$ -test for equal variances. Tests if two independent samples come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances.
<code>vartestn</code>	Bartlett multiple-sample test for equal variances. Tests if multiple samples come from normal distributions with the same variance, against the alternative that they come from normal distributions with different variances.
<code>ztest</code>	One-sample $z$ -test. Tests if a sample comes from a normal distribution with known variance and specified mean, against the alternative that it does not have that mean.

---

**Note** In addition to the previous functions, Statistics Toolbox functions are available for analysis of variance (ANOVA), which perform hypothesis tests in the context of linear modeling.

---

# Analysis of Variance

---

- “Introduction to Analysis of Variance” on page 8-2
- “ANOVA” on page 8-3
- “MANOVA” on page 8-39

## **Introduction to Analysis of Variance**

Analysis of variance (ANOVA) is a procedure for assigning sample variance to different sources and deciding whether the variation arises within or among different population groups. Samples are described in terms of variation around group means and variation of group means around an overall mean. If variations within groups are small relative to variations between groups, a difference in group means may be inferred. Hypothesis tests are used to quantify decisions.



# ANOVA

**In this section...**

- “One-Way ANOVA” on page 8-3
- “Two-Way ANOVA” on page 8-9
- “N-Way ANOVA” on page 8-12
- “Other ANOVA Models” on page 8-26
- “Analysis of Covariance” on page 8-27
- “Nonparametric Methods” on page 8-35

## One-Way ANOVA

- “Introduction to One-Way ANOVA” on page 8-3
- “Example: One-Way ANOVA” on page 8-4
- “Multiple Comparisons” on page 8-6
- “Example: Multiple Comparisons” on page 8-7

### Introduction to One-Way ANOVA

The purpose of one-way ANOVA is to find out whether data from several groups have a common mean. That is, to determine whether the groups are actually different in the measured characteristic.

One-way ANOVA is a simple special case of the linear model. The one-way ANOVA form of the model is

$$y_{ij} = \alpha_{.j} + \varepsilon_{ij}$$

where:

- $y_{ij}$  is a matrix of observations in which each column represents a different group.

- $\alpha_j$  is a matrix whose columns are the group means. (The “dot  $j$ ” notation means that  $\alpha$  applies to all rows of column  $j$ . That is, the value  $\alpha_{ij}$  is the same for all  $i$ .)
- $\varepsilon_{ij}$  is a matrix of random disturbances.

The model assumes that the columns of  $y$  are a constant plus a random disturbance. You want to know if the constants are all the same.

### Example: One-Way ANOVA

The data below comes from a study by Hogg and Ledolter [48] of bacteria counts in shipments of milk. The columns of the matrix `hogg` represent different shipments. The rows are bacteria counts from cartons of milk chosen randomly from each shipment. Do some shipments have higher counts than others?

```
load hogg
hogg
hogg =
```

```
    24    14    11    7    19
    15     7     9    7    24
    21    12     7    4    19
    27    17    13    7    15
    33    14    12   12    10
    23    16    18   18    20
```

```
[p,tbl,stats] = anova1(hogg);
p
p =
    1.1971e-04
```

The standard ANOVA table has columns for the sums of squares, degrees of freedom, mean squares (SS/df),  $F$  statistic, and  $p$  value.

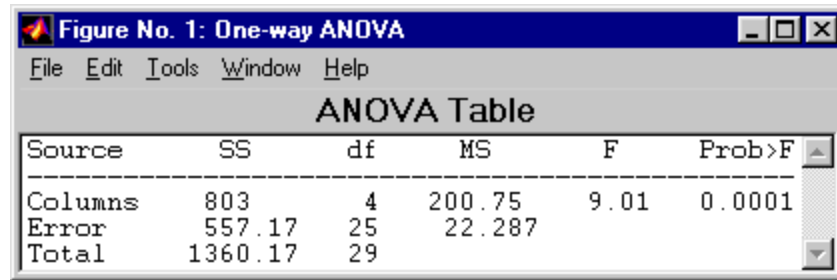


Figure No. 1: One-way ANOVA

File Edit Tools Window Help

ANOVA Table

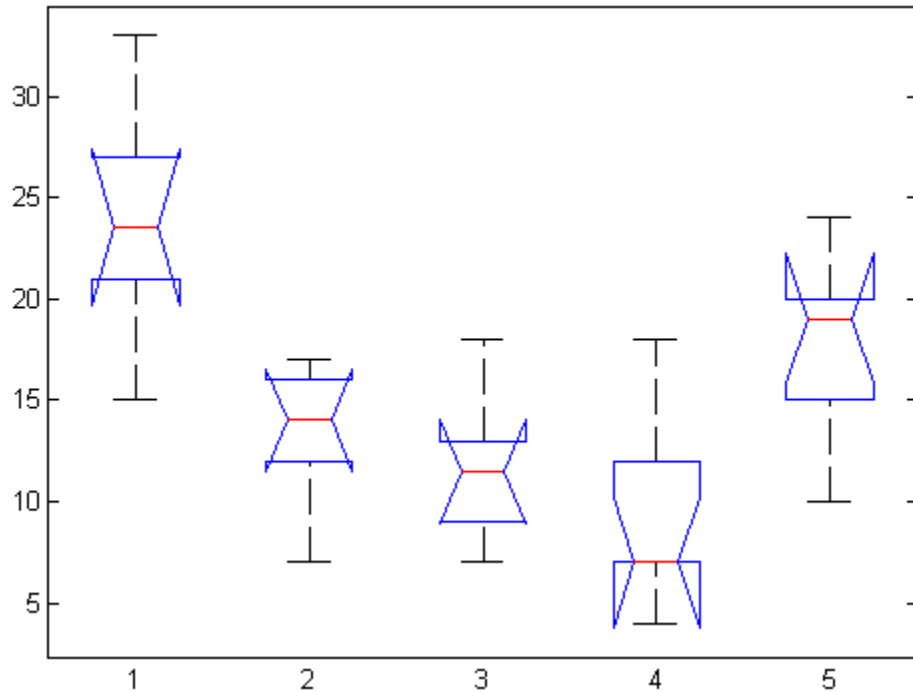
Source	SS	df	MS	F	Prob>F
Columns	803	4	200.75	9.01	0.0001
Error	557.17	25	22.287		
Total	1360.17	29			

You can use the  $F$  statistic to do a hypothesis test to find out if the bacteria counts are the same. `anova1` returns the  $p$  value from this hypothesis test.

In this case the  $p$  value is about 0.0001, a very small value. This is a strong indication that the bacteria counts from the different shipments are not the same. An  $F$  statistic as extreme as the observed  $F$  would occur by chance only once in 10,000 times if the counts were truly equal.

The  $p$  value returned by `anova1` depends on assumptions about the random disturbances  $\varepsilon_{ij}$  in the model equation. For the  $p$  value to be correct, these disturbances need to be independent, normally distributed, and have constant variance.

You can get some graphical assurance that the means are different by looking at the box plots in the second figure window displayed by `anova1`. Note, however, that the notches are used for a comparison of medians, not a comparison of means. For more information on this display, see “Box Plots” on page 4-7.



### Multiple Comparisons

Sometimes you need to determine not just whether there are any differences among the means, but specifically which pairs of means are significantly different. It is tempting to perform a series of  $t$  tests, one for each pair of means, but this procedure has a pitfall.

In a  $t$  test, you compute a  $t$  statistic and compare it to a critical value. The critical value is chosen so that when the means are really the same (any apparent difference is due to random chance), the probability that the  $t$  statistic will exceed the critical value is small, say 5%. When the means are different, the probability that the statistic will exceed the critical value is larger.

In this example there are five means, so there are 10 pairs of means to compare. It stands to reason that if all the means are the same, and if there is a 5% chance of incorrectly concluding that there is a difference in one pair,

then the probability of making at least one incorrect conclusion among all 10 pairs is much larger than 5%.

Fortunately, there are procedures known as *multiple comparison procedures* that are designed to compensate for multiple tests.

### Example: Multiple Comparisons

You can perform a multiple comparison test using the `multcompare` function and supplying it with the `stats` output from `anova1`.

```
load hogg
[p,tbl,stats] = anova1(hogg);
[c,m] = multcompare(stats)
c =
    1.0000    2.0000    2.4953   10.5000   18.5047
    1.0000    3.0000    4.1619   12.1667   20.1714
    1.0000    4.0000    6.6619   14.6667   22.6714
    1.0000    5.0000   -2.0047    6.0000   14.0047
    2.0000    3.0000   -6.3381    1.6667    9.6714
    2.0000    4.0000   -3.8381    4.1667   12.1714
    2.0000    5.0000  -12.5047   -4.5000    3.5047
    3.0000    4.0000   -5.5047    2.5000   10.5047
    3.0000    5.0000  -14.1714   -6.1667    1.8381
    4.0000    5.0000  -16.6714   -8.6667   -0.6619
m =
    23.8333    1.9273
    13.3333    1.9273
    11.6667    1.9273
     9.1667    1.9273
    17.8333    1.9273
```

The first output from `multcompare` has one row for each pair of groups, with an estimate of the difference in group means and a confidence interval for that group. For example, the second row has the values

```
1.0000    3.0000    4.1619   12.1667   20.1714
```

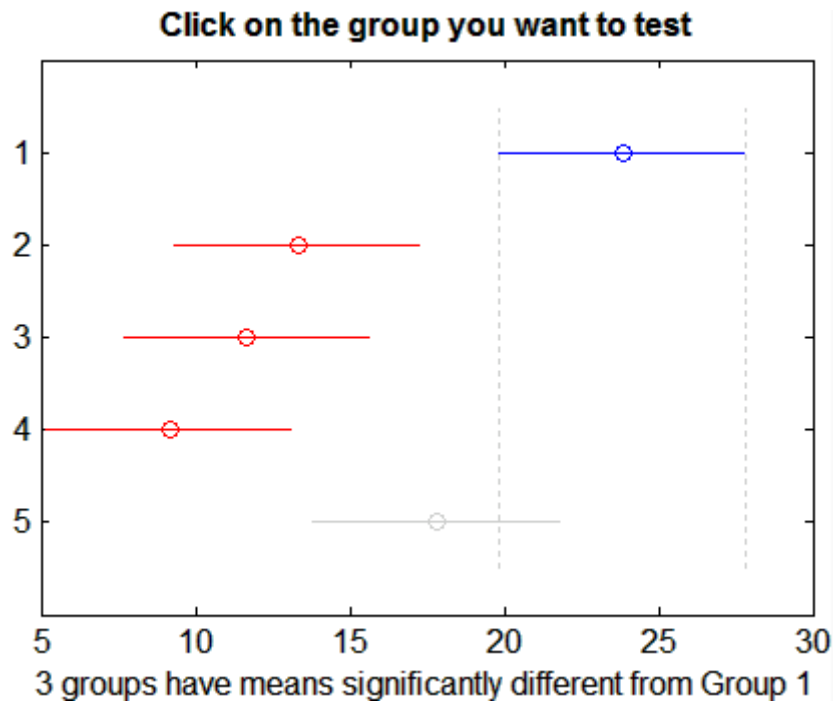
indicating that the mean of group 1 minus the mean of group 3 is estimated to be 12.1667, and a 95% confidence interval for this difference is

[4.1619, 20.1714]. This interval does not contain 0, so you can conclude that the means of groups 1 and 3 are different.

The second output contains the mean and its standard error for each group.

It is easier to visualize the difference between group means by looking at the graph that `multcompare` produces.

There are five groups. The graph instructs you to **Click on the group you want to test**. Three groups have slopes significantly different from group one.



The graph shows that group 1 is significantly different from groups 2, 3, and 4. By using the mouse to select group 4, you can determine that it is also significantly different from group 5. Other pairs are not significantly different.

## Two-Way ANOVA

- “Introduction to Two-Way ANOVA” on page 8-9
- “Example: Two-Way ANOVA” on page 8-10

### Introduction to Two-Way ANOVA

The purpose of two-way ANOVA is to find out whether data from several groups have a common mean. One-way ANOVA and two-way ANOVA differ in that the groups in two-way ANOVA have two categories of defining characteristics instead of one.

Suppose an automobile company has two factories, and each factory makes the same three models of car. It is reasonable to ask if the gas mileage in the cars varies from factory to factory as well as from model to model. There are two predictors, factory and model, to explain differences in mileage.

There could be an overall difference in mileage due to a difference in the production methods between factories. There is probably a difference in the mileage of the different models (irrespective of the factory) due to differences in design specifications. These effects are called *additive*.

Finally, a factory might make high mileage cars in one model (perhaps because of a superior production line), but not be different from the other factory for other models. This effect is called an *interaction*. It is impossible to detect an interaction unless there are duplicate observations for some combination of factory and car model.

Two-way ANOVA is a special case of the linear model. The two-way ANOVA form of the model is

$$y_{ijk} = \mu + \alpha_{.j} + \beta_i + \gamma_{ij} + \varepsilon_{ijk}$$

where, with respect to the automobile example above:

- $y_{ijk}$  is a matrix of gas mileage observations (with row index  $i$ , column index  $j$ , and repetition index  $k$ ).
- $\mu$  is a constant matrix of the overall mean gas mileage.

- $\alpha_j$  is a matrix whose columns are the deviations of each car's gas mileage (from the mean gas mileage  $\mu$ ) that are attributable to the car's *model*. All values in a given column of  $\alpha_j$  are identical, and the values in each row of  $\alpha_j$  sum to 0.
- $\beta_i$  is a matrix whose rows are the deviations of each car's gas mileage (from the mean gas mileage  $\mu$ ) that are attributable to the car's *factory*. All values in a given row of  $\beta_i$  are identical, and the values in each column of  $\beta_i$  sum to 0.
- $\gamma_{ij}$  is a matrix of interactions. The values in each row of  $\gamma_{ij}$  sum to 0, and the values in each column of  $\gamma_{ij}$  sum to 0.
- $\varepsilon_{ijk}$  is a matrix of random disturbances.

### Example: Two-Way ANOVA

The purpose of the example is to determine the effect of car model and factory on the mileage rating of cars.

```
load mileage
mileage
```

```
mileage =
```

```

33.3000  34.5000  37.4000
33.4000  34.8000  36.8000
32.9000  33.8000  37.6000
32.6000  33.4000  36.6000
32.5000  33.7000  37.0000
33.0000  33.9000  36.7000
```

```
cars = 3;
[p,tbl,stats] = anova2(mileage,cars);
p
```

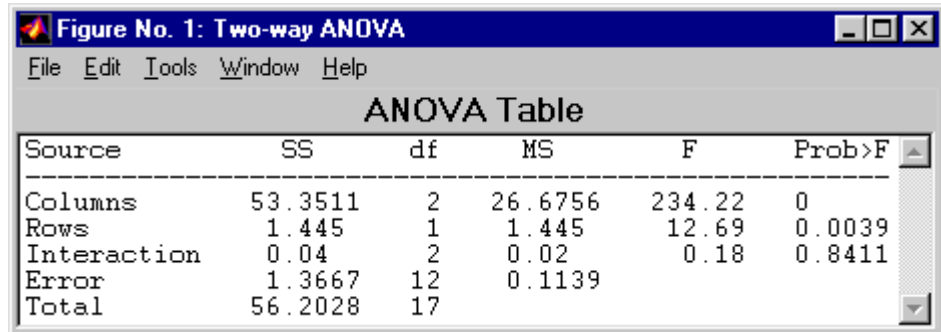
```
p =
0.0000    0.0039    0.8411
```

There are three models of cars (columns) and two factories (rows). The reason there are six rows in `mileage` instead of two is that each factory provides



three cars of each model for the study. The data from the first factory is in the first three rows, and the data from the second factory is in the last three rows.

The standard ANOVA table has columns for the sums of squares, degrees-of-freedom, mean squares (SS/df),  $F$  statistics, and  $p$ -values.



Source	SS	df	MS	F	Prob>F
Columns	53.3511	2	26.6756	234.22	0
Rows	1.445	1	1.445	12.69	0.0039
Interaction	0.04	2	0.02	0.18	0.8411
Error	1.3667	12	0.1139		
Total	56.2028	17			

You can use the  $F$  statistics to do hypotheses tests to find out if the mileage is the same across models, factories, and model-factory pairs (after adjusting for the additive effects). `anova2` returns the  $p$  value from these tests.

The  $p$  value for the model effect is zero to four decimal places. This is a strong indication that the mileage varies from one model to another. An  $F$  statistic as extreme as the observed  $F$  would occur by chance less than once in 10,000 times if the gas mileage were truly equal from model to model. If you used the `multcompare` function to perform a multiple comparison test, you would find that each pair of the three models is significantly different.

The  $p$  value for the factory effect is 0.0039, which is also highly significant. This indicates that one factory is out-performing the other in the gas mileage of the cars it produces. The observed  $p$  value indicates that an  $F$  statistic as extreme as the observed  $F$  would occur by chance about four out of 1000 times if the gas mileage were truly equal from factory to factory.

There does not appear to be any interaction between factories and models. The  $p$  value, 0.8411, means that the observed result is quite likely (84 out of 100 times) given that there is no interaction.

The  $p$ -values returned by `anova2` depend on assumptions about the random disturbances  $\varepsilon_{ijk}$  in the model equation. For the  $p$ -values to be correct these disturbances need to be independent, normally distributed, and have constant variance.

In addition, `anova2` requires that data be *balanced*, which in this case means there must be the same number of cars for each combination of model and factory. The next section discusses a function that supports unbalanced data with any number of predictors.

## N-Way ANOVA

- “Introduction to N-Way ANOVA” on page 8-12
- “N-Way ANOVA with a Small Data Set” on page 8-13
- “N-Way ANOVA with a Large Data Set” on page 8-15
- “ANOVA with Random Effects” on page 8-19

### Introduction to N-Way ANOVA

You can use N-way ANOVA to determine if the means in a set of data differ when grouped by multiple factors. If they do differ, you can determine which factors or combinations of factors are associated with the difference.

N-way ANOVA is a generalization of two-way ANOVA. For three factors, the model can be written

$$y_{ijkl} = \mu + \alpha_{.j.} + \beta_{i..} + \gamma_{..k} + (\alpha\beta)_{ij.} + (\alpha\gamma)_{i.k} + (\beta\gamma)_{.jk} + (\alpha\beta\gamma)_{ijk} + \varepsilon_{ijkl}$$

In this notation parameters with two subscripts, such as  $(\alpha\beta)_{ij.}$ , represent the interaction effect of two factors. The parameter  $(\alpha\beta\gamma)_{ijk}$  represents the three-way interaction. An ANOVA model can have the full set of parameters or any subset, but conventionally it does not include complex interaction terms unless it also includes all simpler terms for those factors. For example, one would generally not include the three-way interaction without also including all two-way interactions.

The `anovan` function performs N-way ANOVA. Unlike the `anova1` and `anova2` functions, `anovan` does not expect data in a tabular form. Instead, it expects a vector of response measurements and a separate vector (or text array) containing the values corresponding to each factor. This input data format is more convenient than matrices when there are more than two factors or when the number of measurements per factor combination is not constant.

### N-Way ANOVA with a Small Data Set

Consider the following two-way example using `anova2`.

```
m = [23 15 20;27 17 63;43 3 55;41 9 90]
m =
    23    15    20
    27    17    63
    43     3    55
    41     9    90
```

```
anova2(m,2)
```

```
ans =
    0.0197    0.2234    0.2663
```

The factor information is implied by the shape of the matrix `m` and the number of measurements at each factor combination (2). Although `anova2` does not actually require arrays of factor values, for illustrative purposes you could create them as follows.

```
cfactor = repmat(1:3,4,1)

cfactor =
     1     2     3
     1     2     3
     1     2     3
     1     2     3

rfactor = [ones(2,3); 2*ones(2,3)]

rfactor =
     1     1     1
```

```
1    1    1
2    2    2
2    2    2
```

The `cfactor` matrix shows that each column of `m` represents a different level of the column factor. The `rfactor` matrix shows that the top two rows of `m` represent one level of the row factor, and bottom two rows of `m` represent a second level of the row factor. In other words, each value `m(i, j)` represents an observation at column factor level `cfactor(i, j)` and row factor level `rfactor(i, j)`.

To solve the above problem with `anovan`, you need to reshape the matrices `m`, `cfactor`, and `rfactor` to be vectors.

```
m = m(:);
cfactor = cfactor(:);
rfactor = rfactor(:);
```

```
[m cfactor rfactor]
```

```
ans =
```

```
23    1    1
27    1    1
43    1    2
41    1    2
15    2    1
17    2    1
 3    2    2
 9    2    2
20    3    1
63    3    1
55    3    2
90    3    2
```

```
anovan(m, {cfactor rfactor}, 2)
```

```
ans =
```

```
0.0197
```

```
0.2234
0.2663
```

## N-Way ANOVA with a Large Data Set

The previous example used `anova2` to study a small data set measuring car mileage. This example illustrates how to analyze a larger set of car data with mileage and other information on 406 cars made between 1970 and 1982. First, load the data set and look at the variable names.

```
load carbig
whos
```

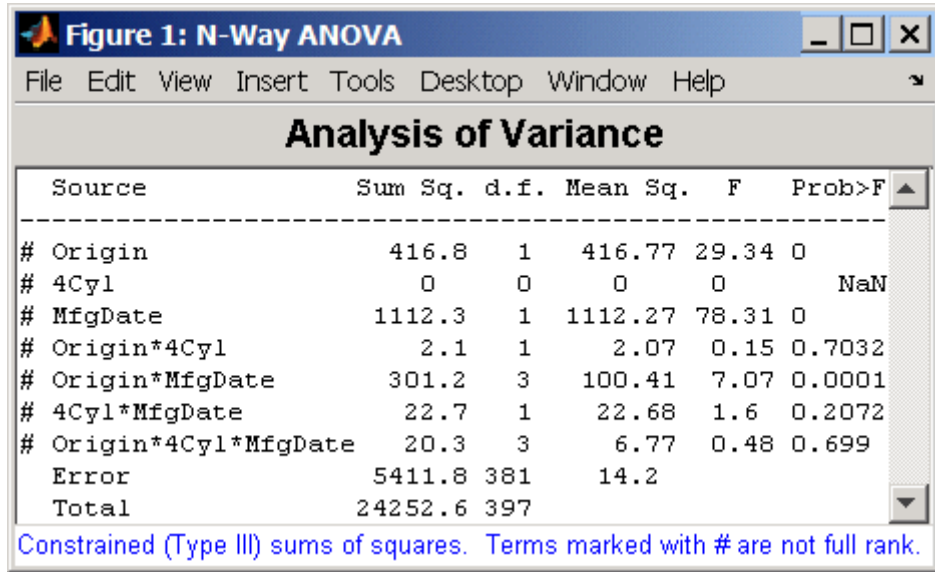
Name	Size	Bytes	Class
Acceleration	406x1	3248	double array
Cylinders	406x1	3248	double array
Displacement	406x1	3248	double array
Horsepower	406x1	3248	double array
MPG	406x1	3248	double array
Model	406x36	29232	char array
Model_Year	406x1	3248	double array
Origin	406x7	5684	char array
Weight	406x1	3248	double array
cyl4	406x5	4060	char array
org	406x7	5684	char array
when	406x5	4060	char array

The example focusses on four variables. MPG is the number of miles per gallon for each of 406 cars (though some have missing values coded as NaN). The other three variables are factors: `cyl4` (four-cylinder car or not), `org` (car originated in Europe, Japan, or the USA), and `when` (car was built early in the period, in the middle of the period, or late in the period).

First, fit the full model, requesting up to three-way interactions and Type 3 sums-of-squares.

```
varnames = {'Origin';'4Cyl';'MfgDate'};
anovan(MPG,{org cyl4 when},3,3,varnames)
ans =
    0.0000
```

NaN  
 0  
 0.7032  
 0.0001  
 0.2072  
 0.6990



Note that many terms are marked by a # symbol as not having full rank, and one of them has zero degrees of freedom and is missing a  $p$  value. This can happen when there are missing factor combinations and the model has higher-order terms. In this case, the cross-tabulation below shows that there are no cars made in Europe during the early part of the period with other than four cylinders, as indicated by the 0 in `table(2,1,1)`.

```
[table, chi2, p, factorvals] = crosstab(org,when,cyl4)
```

```
table(:, :, 1) =
```

```
82  75  25
 0   4   3
 3   3   4
```

```

table(:, :, 2) =
    12    22    38
    23    26    17
    12    25    32

chi2 =

    207.7689

p =

    0

factorvals =

    'USA'      'Early'    'Other'
    'Europe'  'Mid'      'Four'
    'Japan'   'Late'     []

```

Consequently it is impossible to estimate the three-way interaction effects, and including the three-way interaction term in the model makes the fit singular.

Using even the limited information available in the ANOVA table, you can see that the three-way interaction has a  $p$  value of 0.699, so it is not significant. So this time you examine only two-way interactions.

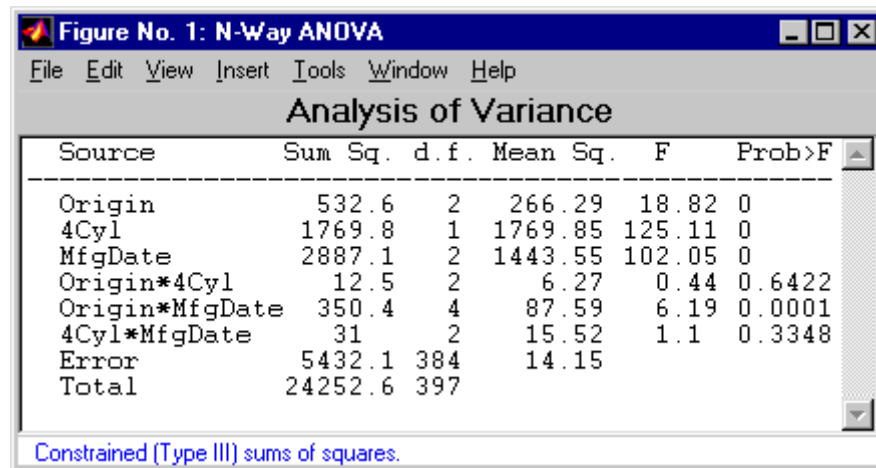
```

[p,tbl,stats,terms] = anovan(MPG,{org cyl4 when},2,3,varnames);
terms

terms =

    1     0     0
    0     1     0
    0     0     1
    1     1     0
    1     0     1
    0     1     1

```



Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
Origin	532.6	2	266.29	18.82	0
4Cyl	1769.8	1	1769.85	125.11	0
MfgDate	2887.1	2	1443.55	102.05	0
Origin*4Cyl	12.5	2	6.27	0.44	0.6422
Origin*MfgDate	350.4	4	87.59	6.19	0.0001
4Cyl*MfgDate	31	2	15.52	1.1	0.3348
Error	5432.1	384	14.15		
Total	24252.6	397			

Constrained (Type III) sums of squares.

Now all terms are estimable. The  $p$ -values for interaction term 4 (Origin\*4Cyl) and interaction term 6 (4Cyl\*MfgDate) are much larger than a typical cutoff value of 0.05, indicating these terms are not significant. You could choose to omit these terms and pool their effects into the error term. The output terms variable returns a matrix of codes, each of which is a bit pattern representing a term. You can omit terms from the model by deleting their entries from terms and running anovan again, this time supplying the resulting vector as the model argument.

```
terms([4 6],:) = []
```

```
terms =
```

```

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

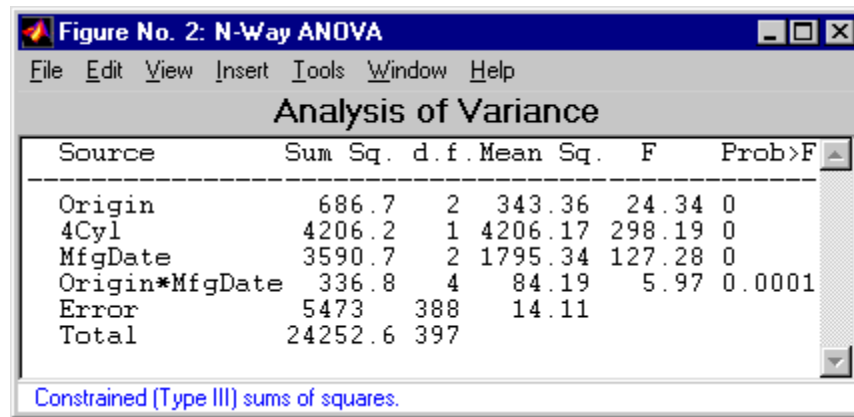
```
anovan(MPG,{org cyl4 when},terms,3,varnames)
```

```
ans =
```

```
1.0e-003 *
```



0.0000  
0  
0  
0.1140



Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
Origin	686.7	2	343.36	24.34	0
4Cyl	4206.2	1	4206.17	298.19	0
MfgDate	3590.7	2	1795.34	127.28	0
Origin*MfgDate	336.8	4	84.19	5.97	0.0001
Error	5473	388	14.11		
Total	24252.6	397			

Constrained (Type III) sums of squares.

Now you have a more parsimonious model indicating that the mileage of these cars seems to be related to all three factors, and that the effect of the manufacturing date depends on where the car was made.

## ANOVA with Random Effects

- “Introduction to ANOVA with Random Effects” on page 8-19
- “Setting Up the Model” on page 8-20
- “Fitting a Random Effects Model” on page 8-21
- “F Statistics for Models with Random Effects” on page 8-22
- “Variance Components” on page 8-24

**Introduction to ANOVA with Random Effects.** In an ordinary ANOVA model, each grouping variable represents a fixed factor. The levels of that factor are a fixed set of values. Your goal is to determine whether different factor levels lead to different response values. This section presents an example that shows how to use anovan to fit models where a factor’s levels represent a random selection from a larger (infinite) set of possible levels.

**Setting Up the Model.** To set up the example, first load the data, which is stored in a 6-by-3 matrix, `mileage`.

```
load mileage
```

The `anova2` function works only with balanced data, and it infers the values of the grouping variables from the row and column numbers of the input matrix. The `anovan` function, on the other hand, requires you to explicitly create vectors of grouping variable values. To create these vectors, do the following steps:

- 1 Create an array indicating the factory for each value in `mileage`. This array is 1 for the first column, 2 for the second, and 3 for the third.

```
factory = repmat(1:3,6,1);
```

- 2 Create an array indicating the car model for each mileage value. This array is 1 for the first three rows of `mileage`, and 2 for the remaining three rows.

```
carmod = [ones(3,3); 2*ones(3,3)];
```

- 3 Turn these matrices into vectors and display them.

```
mileage = mileage(:);  
factory = factory(:);  
carmod = carmod(:);  
[mileage factory carmod]
```

```
ans =
```

```
33.3000    1.0000    1.0000  
33.4000    1.0000    1.0000  
32.9000    1.0000    1.0000  
32.6000    1.0000    2.0000  
32.5000    1.0000    2.0000  
33.0000    1.0000    2.0000  
34.5000    2.0000    1.0000  
34.8000    2.0000    1.0000  
33.8000    2.0000    1.0000  
33.4000    2.0000    2.0000  
33.7000    2.0000    2.0000
```

33.9000	2.0000	2.0000
37.4000	3.0000	1.0000
36.8000	3.0000	1.0000
37.6000	3.0000	1.0000
36.6000	3.0000	2.0000
37.0000	3.0000	2.0000
36.7000	3.0000	2.0000

**Fitting a Random Effects Model.** Continuing the example from the preceding section, suppose you are studying a few factories but you want information about what would happen if you build these same car models in a different factory—either one that you already have or another that you might construct. To get this information, fit the analysis of variance model, specifying a model that includes an interaction term and that the factory factor is random.

```
[pvals,tbl,stats] = anovan(mileage, {factory carmod}, ...
    'model',2, 'random',1,'varnames',{'Factory' 'Car Model'});
```

Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
Factory	53.3511	2	26.6756	1333.78	0.0007
Car Model	1.445	1	1.445	72.25	0.0136
Factory*Car Model	0.04	2	0.02	0.18	0.8411
Error	1.3667	12	0.1139		
Total	56.2028	17			

Constrained (Type III) sums of squares.

In the fixed effects version of this fit, which you get by omitting the inputs 'random', 1 in the preceding code, the effect of car model is significant, with a  $p$  value of 0.0039. But in this example, which takes into account the random variation of the effect of the variable 'Car Model' from one factory to another, the effect is still significant, but with a higher  $p$  value of 0.0136.

**F Statistics for Models with Random Effects.** The  $F$  statistic in a model having random effects is defined differently than in a model having all fixed effects. In the fixed effects model, you compute the  $F$  statistic for any term by taking the ratio of the mean square for that term with the mean square for error. In a random effects model, however, some  $F$  statistics use a different mean square in the denominator.

In the example described in “Setting Up the Model” on page 8-20, the effect of the variable 'Factory' could vary across car models. In this case, the interaction mean square takes the place of the error mean square in the  $F$  statistic. The  $F$  statistic for factory is:

$$F = 26.6756 / 0.02$$

$$F =$$

$$1333.78$$

The degrees of freedom for the statistic are the degrees of freedom for the numerator (2) and denominator (2) mean squares. Therefore the  $p$  value for the statistic is:

$$pval = 1 - fcdf(F,2,2)$$

$$pval =$$

$$7.4919e-04$$

With random effects, the expected value of each mean square depends not only on the variance of the error term, but also on the variances contributed by the random effects. You can see these dependencies by writing the expected values as linear combinations of contributions from the various model terms. To find the coefficients of these linear combinations, enter `stats.ems`, which returns the `ems` field of the `stats` structure:

```
stats.ems
```

```
ans =
```

```
6.0000    0.0000    3.0000    1.0000
0.0000    9.0000    3.0000    1.0000
```

```

0.0000    0.0000    3.0000    1.0000
      0          0          0    1.0000

```

To see text representations of the linear combinations, enter

```
stats.txtems
```

```
ans =
```

```

'6*V(Factory)+3*V(Factory*Car Model)+V(Error) '
'9*Q(Car Model)+3*V(Factory*Car Model)+V(Error) '
'3*V(Factory*Car Model)+V(Error) '
'V(Error) '

```

The expected value for the mean square due to car model (second term) includes contributions from a quadratic function of the car model effects, plus three times the variance of the interaction term's effect, plus the variance of the error term. Notice that if the car model effects were all zero, the expression would reduce to the expected mean square for the third term (the interaction term). That is why the  $F$  statistic for the car model effect uses the interaction mean square in the denominator.

In some cases there is no single term whose expected value matches the one required for the denominator of the  $F$  statistic. In that case, the denominator is a linear combination of mean squares. The `stats` structure contains fields giving the definitions of the denominators for each  $F$  statistic. The `txtdenom` field, `stats.txtdenom`, gives a text representation, and the `denom` field gives a matrix that defines a linear combination of the variances of terms in the model. For balanced models like this one, the `denom` matrix, `stats.denom`, contains zeros and ones, because the denominator is just a single term's mean square:

```
stats.txtdenom
```

```
ans =
```

```

'MS(Factory*Car Model) '
'MS(Factory*Car Model) '
'MS(Error) '

```

```
stats.denom
```

```
ans =  
  
-0.0000    1.0000    0.0000  
 0.0000    1.0000   -0.0000  
 0.0000         0    1.0000
```

**Variance Components.** For the model described in “Setting Up the Model” on page 8-20, consider the mileage for a particular car of a particular model made at a random factory. The variance of that car is the sum of components, or contributions, one from each of the random terms.

```
stats.rtnames
```

```
ans =  
  
  'Factory'  
  'Factory*Car Model'  
  'Error'
```

You do not know those variances, but you can estimate them from the data. Recall that the `ems` field of the `stats` structure expresses the expected value of each term’s mean square as a linear combination of unknown variances for random terms, and unknown quadratic forms for fixed terms. If you take the expected mean square expressions for the random terms, and equate those expected values to the computed mean squares, you get a system of equations that you can solve for the unknown variances. These solutions are the variance component estimates. The `varest` field contains a variance component estimate for each term. The `rtnames` field contains the names of the random terms.

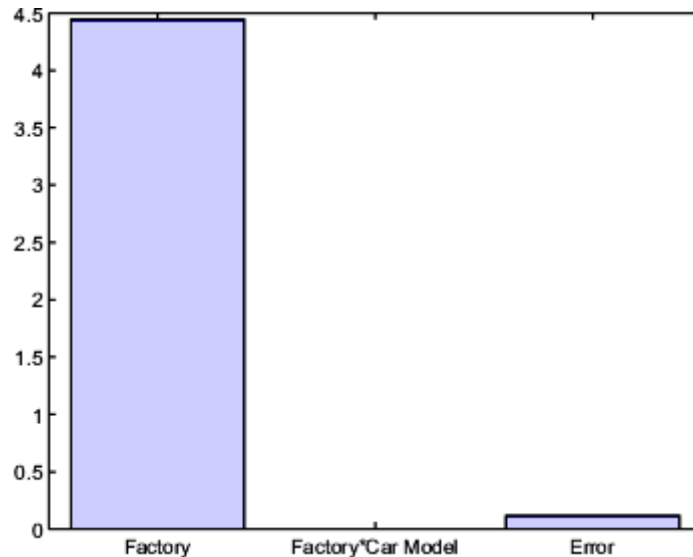
```
stats.varest
```

```
ans =  
  
  4.4426  
 -0.0313  
  0.1139
```

Under some conditions, the variability attributed to a term is unusually low, and that term’s variance component estimate is negative. In those cases it

is common to set the estimate to zero, which you might do, for example, to create a bar graph of the components.

```
bar(max(0,stats.varest))
set(gca,'xtick',1:3,'xticklabel',stats.rtnames)
```



You can also compute confidence bounds for the variance estimate. The `anovan` function does this by computing confidence bounds for the variance expected mean squares, and finding lower and upper limits on each variance component containing all of these bounds. This procedure leads to a set of bounds that is conservative for balanced data. (That is, 95% confidence bounds will have a probability of at least 95% of containing the true variances if the number of observations for each combination of grouping variables is the same.) For unbalanced data, these are approximations that are not guaranteed to be conservative.

```
[{'Term' 'Estimate' 'Lower' 'Upper'};
 stats.rtnames, num2cell([stats.varest stats.varci])]
```

```
ans =
```

```
    'Term'                'Estimate'    'Lower'    'Upper'
```

```

'Factory'          [ 4.4426]    [1.0736]    [175.6038]
'Factory*Car Model' [ -0.0313]   [   NaN]    [   NaN]
'Error'           [ 0.1139]   [0.0586]    [ 0.3103]

```

## Other ANOVA Models

The `anovan` function also has arguments that enable you to specify two other types of model terms. First, the `'nested'` argument specifies a matrix that indicates which factors are nested within other factors. A nested factor is one that takes different values within each level its nested factor.

For example, the mileage data from the previous section assumed that the two car models produced in each factory were the same. Suppose instead, each factory produced two distinct car models for a total of six car models, and we numbered them 1 and 2 for each factory for convenience. Then, the car model is nested in factory. A more accurate and less ambiguous numbering of car model would be as follows:

<b>Factory</b>	<b>Car Model</b>
1	1
1	2
2	3
2	4
3	5
3	6

However, it is common with nested models to number the nested factor the same way in each nested factor.

Second, the `'continuous'` argument specifies that some factors are to be treated as continuous variables. The remaining factors are categorical variables. Although the `anovan` function can fit models with multiple continuous and categorical predictors, the simplest model that combines one predictor of each type is known as an *analysis of covariance* model. The next section describes a specialized tool for fitting this model.



## Analysis of Covariance

- “Introduction to Analysis of Covariance” on page 8-27
- “Analysis of Covariance Tool” on page 8-27
- “Confidence Bounds” on page 8-31
- “Multiple Comparisons” on page 8-34

### Introduction to Analysis of Covariance

Analysis of covariance is a technique for analyzing grouped data having a response ( $y$ , the variable to be predicted) and a predictor ( $x$ , the variable used to do the prediction). Using analysis of covariance, you can model  $y$  as a linear function of  $x$ , with the coefficients of the line possibly varying from group to group.

### Analysis of Covariance Tool

The `aocool` function opens an interactive graphical environment for fitting and prediction with analysis of covariance (ANOCOVA) models. It fits the following models for the  $i$ th group:

Same mean	$y = a + \varepsilon$
Separate means	$y = (a + a_i) + \varepsilon$
Same line	$y = a + \beta x + \varepsilon$
Parallel lines	$y = (a + a_i) + \beta x + \varepsilon$
Separate lines	$y = (a + a_i) + (\beta + \beta_i)x + \varepsilon$

For example, in the parallel lines model the intercept varies from one group to the next, but the slope is the same for each group. In the same mean model, there is a common intercept and no slope. In order to make the group coefficients well determined, the tool imposes the constraints

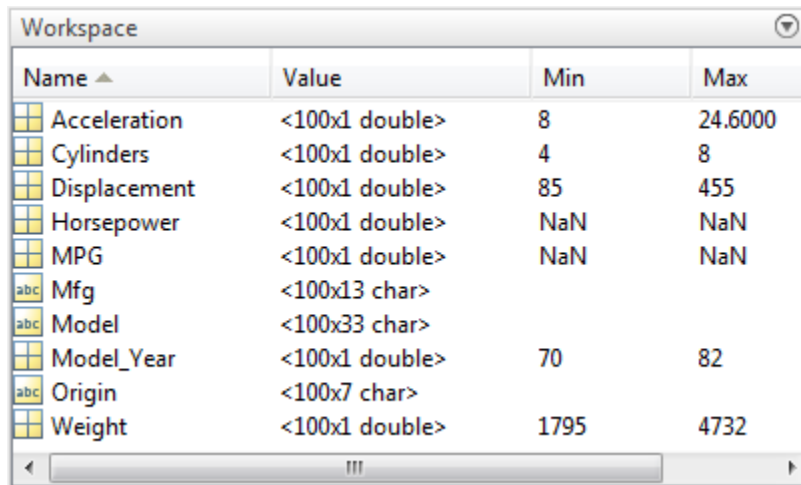
$$\sum \alpha_j = \sum \beta_j = 0$$

The following steps describe the use of `aocool`.

- 1 Load the data.** The Statistics Toolbox data set `carsmall.mat` contains information on cars from the years 1970, 1976, and 1982. This example studies the relationship between the weight of a car and its mileage, and whether this relationship has changed over the years. To start the demonstration, load the data set.

```
load carsmall
```

The Workspace Browser shows the variables in the data set.



Name	Value	Min	Max
Acceleration	<100x1 double>	8	24.6000
Cylinders	<100x1 double>	4	8
Displacement	<100x1 double>	85	455
Horsepower	<100x1 double>	NaN	NaN
MPG	<100x1 double>	NaN	NaN
Mfg	<100x13 char>		
Model	<100x33 char>		
Model_Year	<100x1 double>	70	82
Origin	<100x7 char>		
Weight	<100x1 double>	1795	4732

You can also use `aoctool` with your own data.

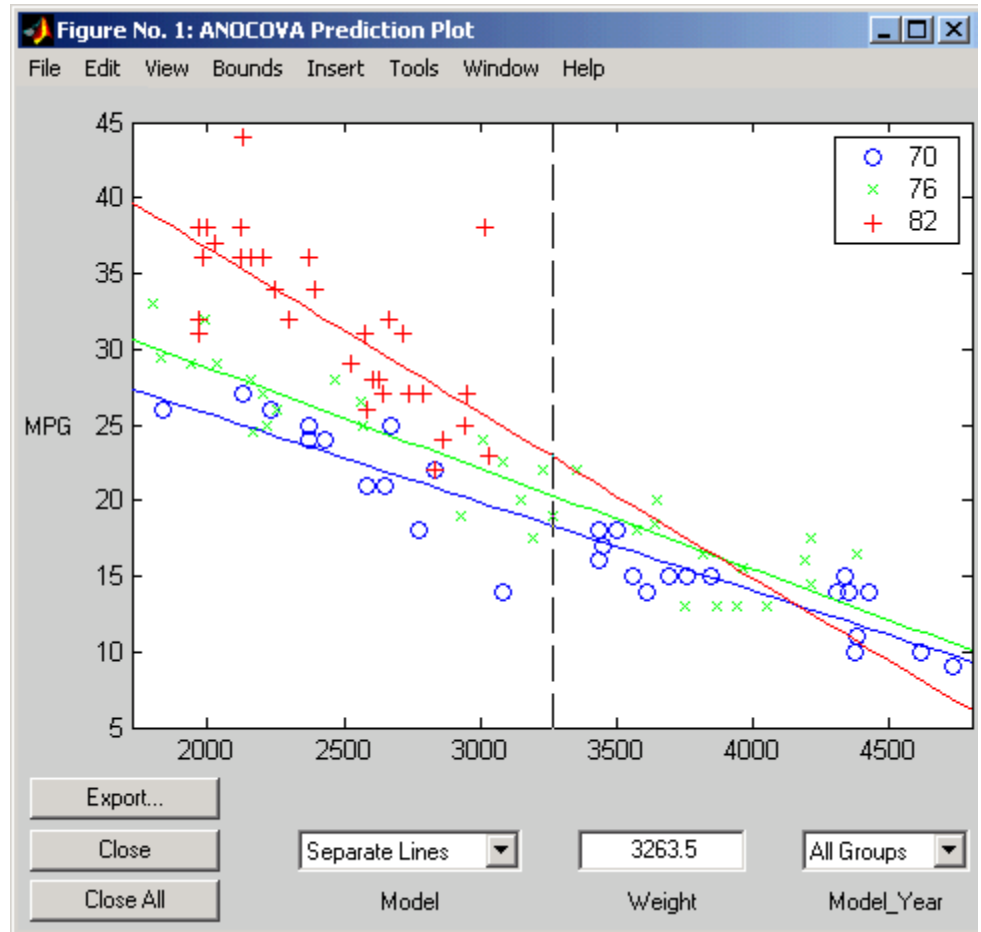
- 2 Start the tool.** The following command calls `aoctool` to fit a separate line to the column vectors `Weight` and `MPG` for each of the three model group defined in `Model_Year`. The initial fit models the  $y$  variable, `MPG`, as a linear function of the  $x$  variable, `Weight`.

```
[h,atab,ctab,stats] = aoctool(Weight,MPG,Model_Year);
```

See the `aoctool` function reference page for detailed information about calling `aoctool`.

- 3 Examine the output.** The graphical output consists of a main window with a plot, a table of coefficient estimates, and an analysis of variance table. In the plot, each `Model_Year` group has a separate line. The data

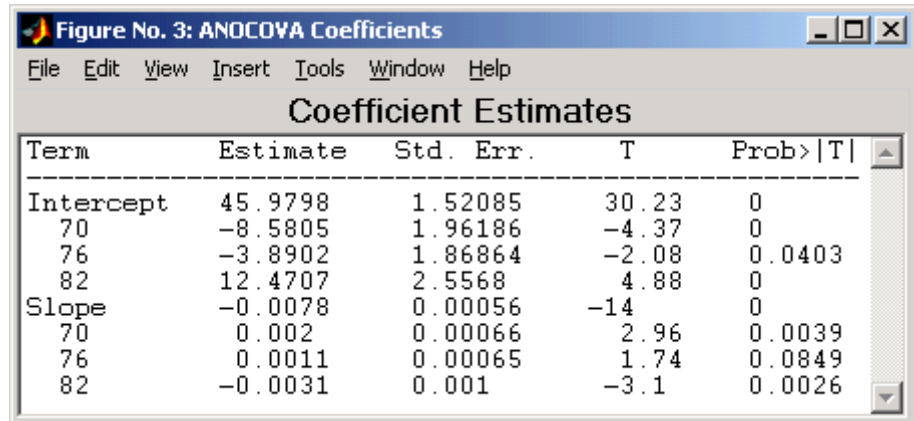
points for each group are coded with the same color and symbol, and the fit for each group has the same color as the data points.



The coefficients of the three lines appear in the figure titled ANCOVA Coefficients. You can see that the slopes are roughly  $-0.0078$ , with a small deviation for each group:

- Model year 1970:  $y = (45.9798 - 8.5805) + (-0.0078 + 0.002)x + \varepsilon$
- Model year 1976:  $y = (45.9798 - 3.8902) + (-0.0078 + 0.0011)x + \varepsilon$

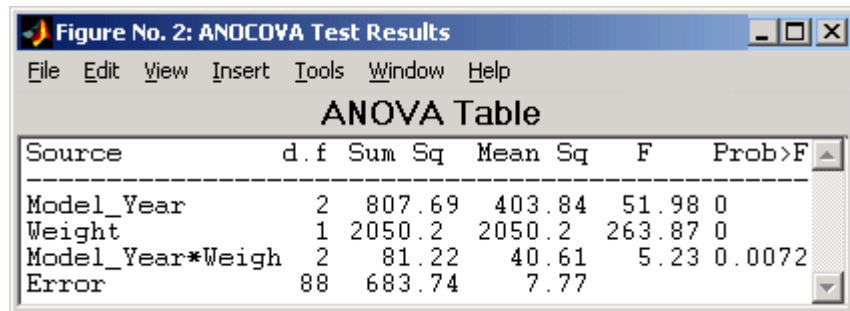
- Model year 1982:  $y = (45.9798 + 12.4707) + (-0.0078 - 0.0031)x + \varepsilon$



**Figure No. 3: ANCOVA Coefficients**

Term	Estimate	Std. Err.	T	Prob> T
Intercept	45.9798	1.52085	30.23	0
70	-8.5805	1.96186	-4.37	0
76	-3.8902	1.86864	-2.08	0.0403
82	12.4707	2.5568	4.88	0
Slope	-0.0078	0.00056	-14	0
70	0.002	0.00066	2.96	0.0039
76	0.0011	0.00065	1.74	0.0849
82	-0.0031	0.001	-3.1	0.0026

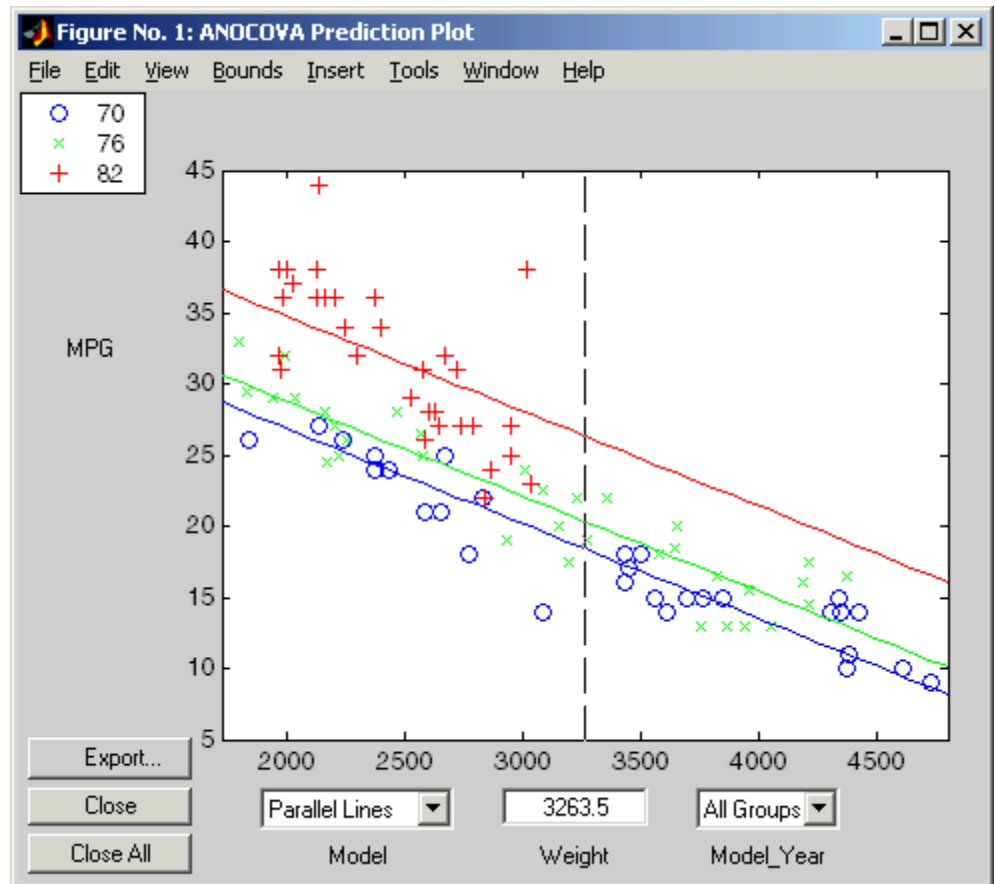
Because the three fitted lines have slopes that are roughly similar, you may wonder if they really are the same. The `Model_Year*Weight` interaction expresses the difference in slopes, and the ANOVA table shows a test for the significance of this term. With an  $F$  statistic of 5.23 and a  $p$  value of 0.0072, the slopes are significantly different.



**Figure No. 2: ANCOVA Test Results**

Source	d.f	Sum Sq	Mean Sq	F	Prob>F
Model_Year	2	807.69	403.84	51.98	0
Weight	1	2050.2	2050.2	263.87	0
Model_Year*Weigh	2	81.22	40.61	5.23	0.0072
Error	88	683.74	7.77		

- 4 Constrain the slopes to be the same.** To examine the fits when the slopes are constrained to be the same, return to the ANCOVA Prediction Plot window and use the **Model** pop-up menu to select a **Parallel Lines** model. The window updates to show the following graph.

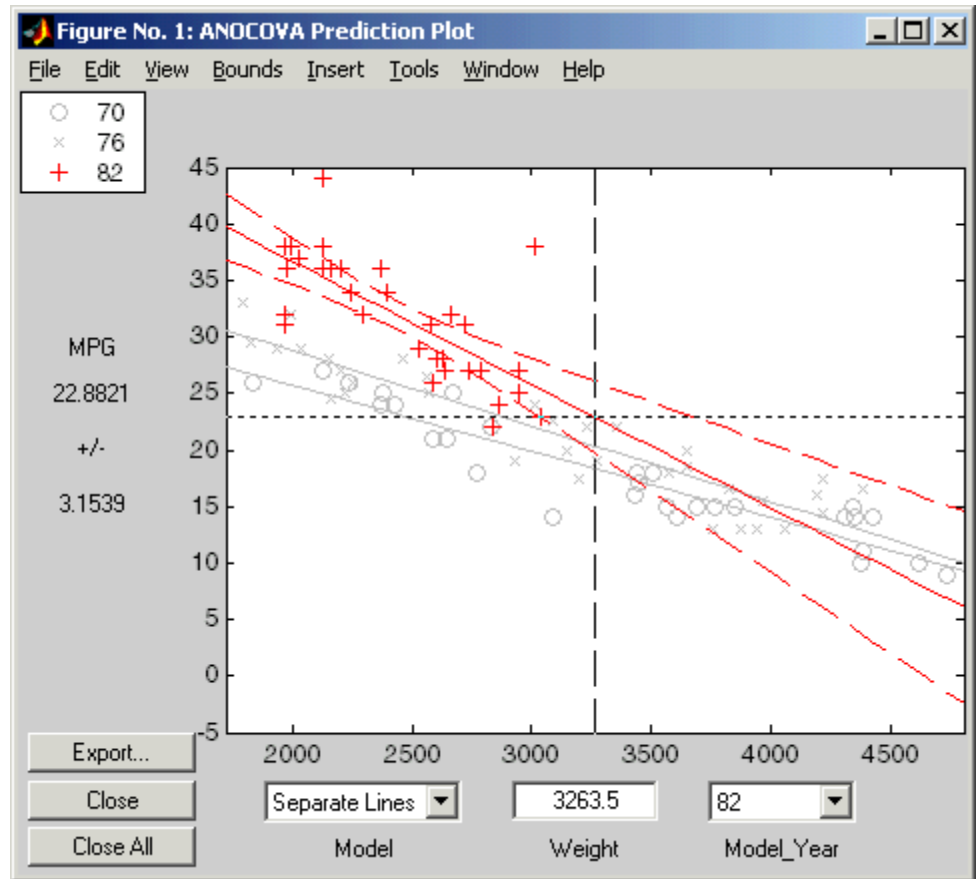


Though this fit looks reasonable, it is significantly worse than the Separate Lines model. Use the **Model** pop-up menu again to return to the original model.

### Confidence Bounds

The example in “Analysis of Covariance Tool” on page 8-27 provides estimates of the relationship between MPG and Weight for each Model\_Year, but how accurate are these estimates? To find out, you can superimpose confidence bounds on the fits by examining them one group at a time.

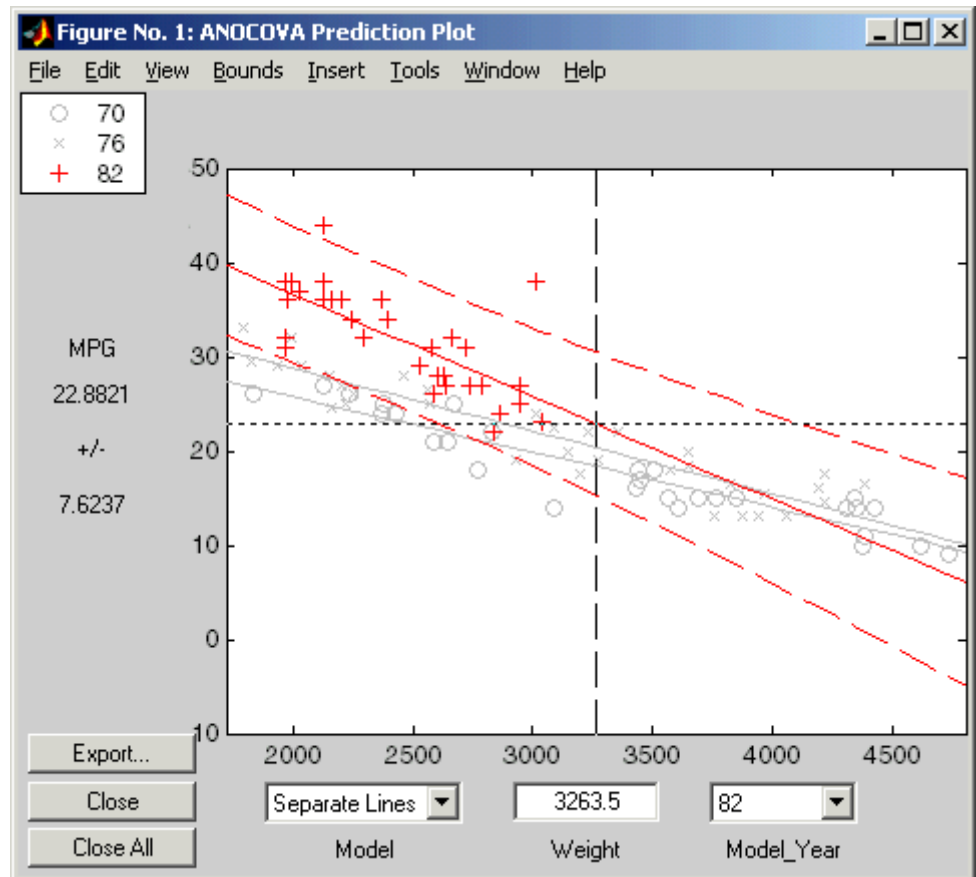
- 1 In the **Model\_Year** menu at the lower right of the figure, change the setting from All Groups to 82. The data and fits for the other groups are dimmed, and confidence bounds appear around the 82 fit.



The dashed lines form an envelope around the fitted line for model year 82. Under the assumption that the true relationship is linear, these bounds provide a 95% confidence region for the true line. Note that the fits for the other model years are well outside these confidence bounds for Weight values between 2000 and 3000.

- 2 Sometimes it is more valuable to be able to predict the response value for a new observation, not just estimate the average response value. Use the

aoctool function **Bounds** menu to change the definition of the confidence bounds from Line to Observation. The resulting wider intervals reflect the uncertainty in the parameter estimates as well as the randomness of a new observation.



Like the polytool function, the aoctool function has cross hairs that you can use to manipulate the Weight and watch the estimate and confidence bounds along the y-axis update. These values appear only when a single group is selected, not when All Groups is selected.

## Multiple Comparisons

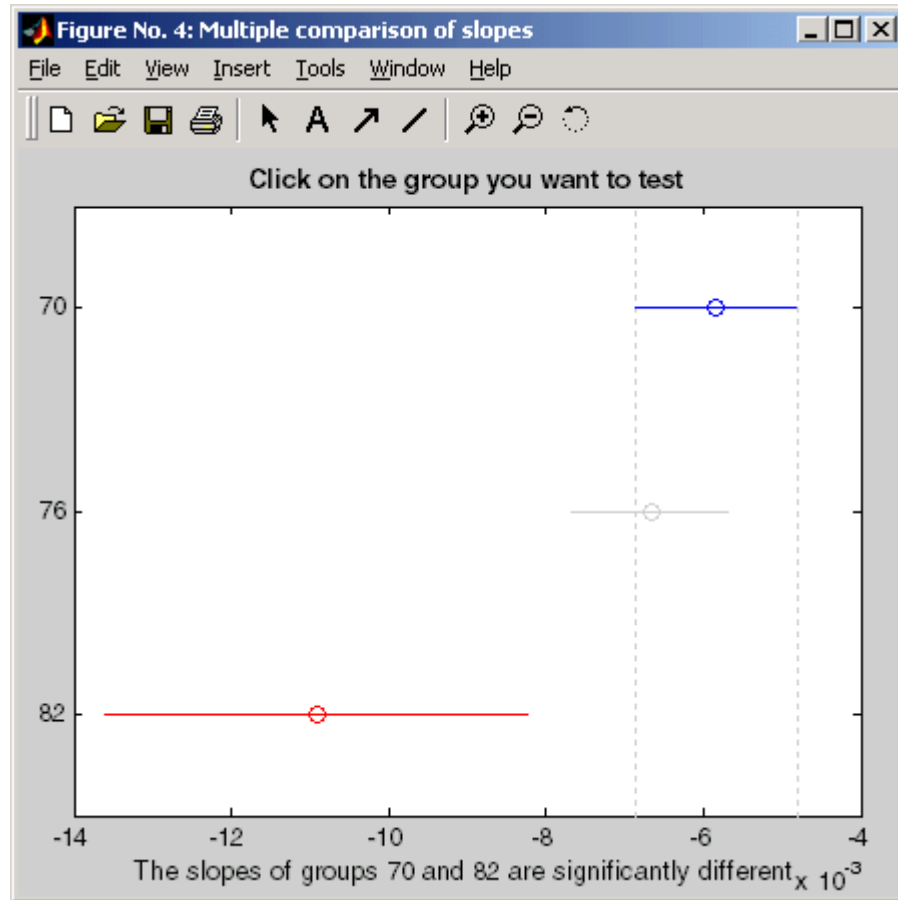
You can perform a multiple comparison test by using the `stats` output structure from `aocool` as input to the `multcompare` function. The `multcompare` function can test either slopes, intercepts, or population marginal means (the predicted MPG of the mean weight for each group). The example in “Analysis of Covariance Tool” on page 8-27 shows that the slopes are not all the same, but could it be that two are the same and only the other one is different? You can test that hypothesis.

```
multcompare(stats,0.05,'on',' ','s')
```

```
ans =  
    1.0000    2.0000   -0.0012    0.0008    0.0029  
    1.0000    3.0000    0.0013    0.0051    0.0088  
    2.0000    3.0000    0.0005    0.0042    0.0079
```

This matrix shows that the estimated difference between the intercepts of groups 1 and 2 (1970 and 1976) is 0.0008, and a confidence interval for the difference is  $[-0.0012, 0.0029]$ . There is no significant difference between the two. There are significant differences, however, between the intercept for 1982 and each of the other two. The graph shows the same information.





Note that the `stats` structure was created in the initial call to the `aocool` function, so it is based on the initial model fit (typically a separate-lines model). If you change the model interactively and want to base your multiple comparisons on the new model, you need to run `aocool` again to get another `stats` structure, this time specifying your new model as the initial model.

## Nonparametric Methods

- “Introduction to Nonparametric Methods” on page 8-36

- “Kruskal-Wallis Test” on page 8-36
- “Friedman’s Test” on page 8-37

## Introduction to Nonparametric Methods

Statistics Toolbox functions include nonparametric versions of one-way and two-way analysis of variance. Unlike classical tests, nonparametric tests make only mild assumptions about the data, and are appropriate when the distribution of the data is non-normal. On the other hand, they are less powerful than classical methods for normally distributed data.

Both of the nonparametric functions described here will return a `stats` structure that can be used as an input to the `multcompare` function for multiple comparisons.

## Kruskal-Wallis Test

The example “Example: One-Way ANOVA” on page 8-4 uses one-way analysis of variance to determine if the bacteria counts of milk varied from shipment to shipment. The one-way analysis rests on the assumption that the measurements are independent, and that each has a normal distribution with a common variance and with a mean that was constant in each column. You can conclude that the column means were not all the same. The following example repeats that analysis using a nonparametric procedure.

The Kruskal-Wallis test is a nonparametric version of one-way analysis of variance. The assumption behind this test is that the measurements come from a continuous distribution, but not necessarily a normal distribution. The test is based on an analysis of variance using the ranks of the data values, not the data values themselves. Output includes a table similar to an ANOVA table, and a box plot.

You can run this test as follows:

```
load hogg

p = kruskalwallis(hogg)
p =
    0.0020
```

The low  $p$  value means the Kruskal-Wallis test results agree with the one-way analysis of variance results.

### Friedman's Test

“Example: Two-Way ANOVA” on page 8-10 uses two-way analysis of variance to study the effect of car model and factory on car mileage. The example tests whether either of these factors has a significant effect on mileage, and whether there is an interaction between these factors. The conclusion of the example is there is no interaction, but that each individual factor has a significant effect. The next example examines whether a nonparametric analysis leads to the same conclusion.

Friedman's test is a nonparametric test for data having a two-way layout (data grouped by two categorical factors). Unlike two-way analysis of variance, Friedman's test does not treat the two factors symmetrically and it does not test for an interaction between them. Instead, it is a test for whether the columns are different after adjusting for possible row differences. The test is based on an analysis of variance using the ranks of the data across categories of the row factor. Output includes a table similar to an ANOVA table.

You can run Friedman's test as follows.

```
load mileage
p = friedman(mileage,3)
p =
    7.4659e-004
```

Recall the classical analysis of variance gave a  $p$  value to test column effects, row effects, and interaction effects. This  $p$  value is for column effects. Using either this  $p$  value or the  $p$  value from ANOVA ( $p < 0.0001$ ), you conclude that there are significant column effects.

In order to test for row effects, you need to rearrange the data to swap the roles of the rows in columns. For a data matrix  $x$  with no replications, you could simply transpose the data and type

```
p = friedman(x')
```

With replicated data it is slightly more complicated. A simple way is to transform the matrix into a three-dimensional array with the first dimension

representing the replicates, swapping the other two dimensions, and restoring the two-dimensional shape.

```
x = reshape(mileage, [3 2 3]);  
x = permute(x,[1 3 2]);  
x = reshape(x,[9 2])  
x =  
    33.3000    32.6000  
    33.4000    32.5000  
    32.9000    33.0000  
    34.5000    33.4000  
    34.8000    33.7000  
    33.8000    33.9000  
    37.4000    36.6000  
    36.8000    37.0000  
    37.6000    36.7000
```

```
friedman(x,3)  
ans =  
    0.0082
```

Again, the conclusion is similar to that of the classical analysis of variance. Both this  $p$  value and the one from ANOVA ( $p = 0.0039$ ) lead you to conclude that there are significant row effects.

You cannot use Friedman's test to test for interactions between the row and column factors.

# MANOVA

## In this section...

“Introduction to MANOVA” on page 8-39

“ANOVA with Multiple Responses” on page 8-39

## Introduction to MANOVA

The analysis of variance technique in “Example: One-Way ANOVA” on page 8-4 takes a set of grouped data and determine whether the mean of a variable differs significantly among groups. Often there are multiple response variables, and you are interested in determining whether the entire set of means is different from one group to the next. There is a multivariate version of analysis of variance that can address the problem.

## ANOVA with Multiple Responses

The carsmall data set has measurements on a variety of car models from the years 1970, 1976, and 1982. Suppose you are interested in whether the characteristics of the cars have changed over time.

Load the sample data.

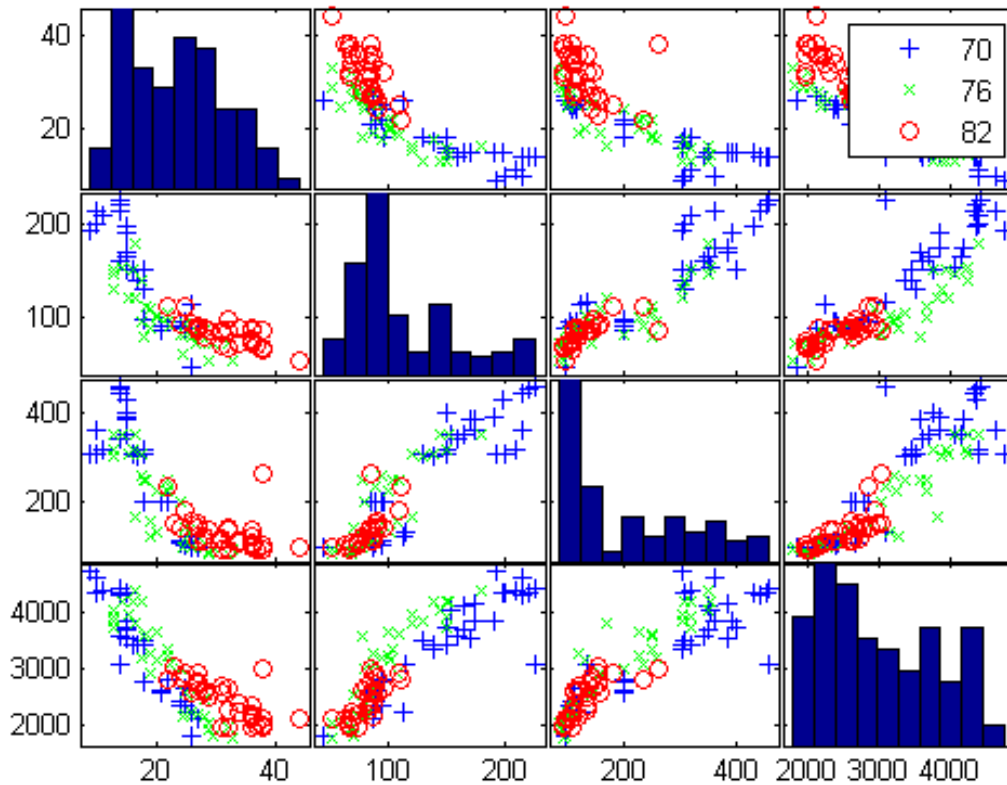
```
load carsmall
whos
```

Name	Size	Bytes	Class	Attributes
Acceleration	100x1	800	double	
Cylinders	100x1	800	double	
Displacement	100x1	800	double	
Horsepower	100x1	800	double	
MPG	100x1	800	double	
Mfg	100x13	2600	char	
Model	100x33	6600	char	
Model_Year	100x1	800	double	
Origin	100x7	1400	char	
Weight	100x1	800	double	

Four of these variables (Acceleration, Displacement, Horsepower, and MPG) are continuous measurements on individual car models. The variable Model\_Year indicates the year in which the car was made. You can create a grouped plot matrix of these variables using the `gplotmatrix` function.

Create a grouped plot matrix of these variables using the `gplotmatrix` function.

```
x = [MPG Horsepower Displacement Weight];  
gplotmatrix(x,[],Model_Year,[],'+xo')
```



(When the second argument of `gplotmatrix` is empty, the function graphs the columns of the `x` argument against each other, and places histograms along the diagonals. The empty fourth argument produces a graph with the default colors. The fifth argument controls the symbols used to distinguish between groups.)

It appears the cars do differ from year to year. The upper right plot, for example, is a graph of MPG versus Weight. The 1982 cars appear to have higher mileage than the older cars, and they appear to weigh less on average. But as a group, are the three years significantly different from one another? The `manova1` function can answer that question.

```
[d,p,stats] = manova1(x,Model_Year)
```

```
d =
```

```
2
```

```
p =
```

```
1.0e-06 *
```

```
0.0000
```

```
0.1141
```

```
stats =
```

```
W: [4x4 double]
```

```
B: [4x4 double]
```

```
T: [4x4 double]
```

```
dfW: 90
```

```
dfB: 2
```

```
dfT: 92
```

```
lambda: [2x1 double]
```

```
chisq: [2x1 double]
```

```
chisqdf: [2x1 double]
```

```
eigenval: [4x1 double]
```

```
eigenvec: [4x4 double]
  canon: [100x4 double]
  mdist: [1x100 double]
  gmdist: [3x3 double]
  gnames: {3x1 cell}
```

The `manova1` function produces three outputs:

- The first output, `d`, is an estimate of the dimension of the group means. If the means were all the same, the dimension would be 0, indicating that the means are at the same point. If the means differed but fell along a line, the dimension would be 1. In the example the dimension is 2, indicating that the group means fall in a plane but not along a line. This is the largest possible dimension for the means of three groups.
- The second output, `p`, is a vector of  $p$ -values for a sequence of tests. The first  $p$  value tests whether the dimension is 0, the next whether the dimension is 1, and so on. In this case both  $p$ -values are small. That's why the estimated dimension is 2.
- The third output, `stats`, is a structure containing several fields, described in the following section.

### The Fields of the `stats` Structure

The `W`, `B`, and `T` fields are matrix analogs to the within, between, and total sums of squares in ordinary one-way analysis of variance. The next three fields are the degrees of freedom for these matrices. Fields `lambda`, `chisq`, and `chisqdf` are the ingredients of the test for the dimensionality of the group means. (The  $p$ -values for these tests are the first output argument of `manova1`.)

The next three fields are used to do a canonical analysis. Recall that in principal components analysis (“Principal Component Analysis (PCA)” on page 13-80) you look for the combination of the original variables that has the largest possible variation. In multivariate analysis of variance, you instead look for the linear combination of the original variables that has the largest separation between groups. It is the single variable that would give the most significant result in a univariate one-way analysis of variance. Having found that combination, you next look for the combination with the second highest separation, and so on.

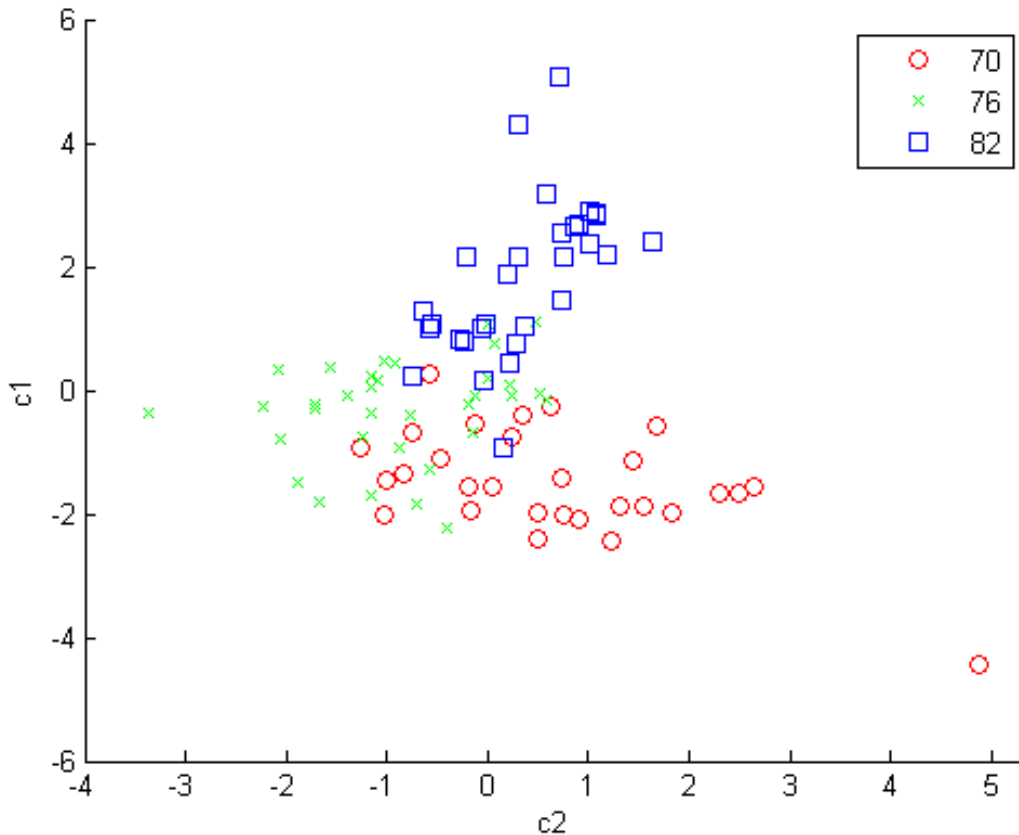


The `eigenvec` field is a matrix that defines the coefficients of the linear combinations of the original variables. The `eigenval` field is a vector measuring the ratio of the between-group variance to the within-group variance for the corresponding linear combination. The `canon` field is a matrix of the canonical variable values. Each column is a linear combination of the mean-centered original variables, using coefficients from the `eigenvec` matrix.

```
c1 = stats.canon(:,1);  
c2 = stats.canon(:,2);
```

Plot the grouped scatter plot of the first two canonical variables.

```
figure()  
gscatter(c2,c1,Model_Year,[],'oxs')
```



A grouped scatter plot of the first two canonical variables shows more separation between groups than a grouped scatter plot of any pair of original variables. In this example, it shows three clouds of points, overlapping but with distinct centers. One point in the bottom right sits apart from the others. You can mark this point on the plot using the `gname` function.

Roughly speaking, the first canonical variable,  $c_1$ , separates the 1982 cars (which have high values of  $c_1$ ) from the older cars. The second canonical variable,  $c_2$ , reveals some separation between the 1970 and 1976 cars.

The final two fields of the `stats` structure are Mahalanobis distances. The `mdist` field measures the distance from each point to its group mean. Points with large values may be outliers. In this data set, the largest outlier is the one in the scatter plot, the Buick Estate station wagon. (Note that you could have supplied the model name to the `gname` function above if you wanted to label the point with its model name rather than its row number.)

Find the largest distance from the group mean.

```
max(stats.mdist)
```

```
ans =
```

```
31.5273
```

Find the point that has the largest distance from the group mean.

```
find(stats.mdist == ans)
```

```
ans =
```

```
20
```

Find the car model that corresponds to the largest distance from the group mean.

```
Model(20,:)
```

```
ans =
```

```
buick estate wagon (sw)
```

The `gmdist` field measures the distances between each pair of group means. Examine the group means using `grpstats`.

```
grpstats(x, Model_Year)
```

```
ans =
```

```
1.0e+03 *
```

```
0.0177    0.1489    0.2869    3.4413
0.0216    0.1011    0.1978    3.0787
0.0317    0.0815    0.1289    2.4535
```

Find the distances between the each pair of group means.

```
stats.gmdist
```

```
ans =
```

```
0    3.8277    11.1106
3.8277    0    6.1374
11.1106    6.1374    0
```

As might be expected, the multivariate distance between the extreme years 1970 and 1982 (11.1) is larger than the difference between more closely spaced years (3.8 and 6.1). This is consistent with the scatter plots, where the points seem to follow a progression as the year changes from 1970 through 1976 to 1982. If you had more groups, you might find it instructive to use the `manovacluster` function to draw a diagram that presents clusters of the groups, formed using the distances between their means.

# Parametric Regression Analysis

---

- “Parametric Regression Analysis” on page 9-3
- “What Are Linear Regression Models?” on page 9-8
- “Linear Regression” on page 9-12
- “Regression Using Dataset Arrays” on page 9-51
- “Linear Regression with Interaction Effects” on page 9-54
- “Interpret Linear Regression Results” on page 9-64
- “Cook’s Distance” on page 9-72
- “Coefficient Standard Errors and Confidence Intervals” on page 9-76
- “Coefficient of Determination (R-Squared)” on page 9-80
- “Delete-1 Statistics” on page 9-83
- “Durbin-Watson Test” on page 9-93
- “F-statistic and t-statistic” on page 9-95
- “Hat Matrix and Leverage” on page 9-102
- “Residuals” on page 9-107
- “Summary of Output and Diagnostic Statistics” on page 9-117
- “Stepwise Regression” on page 9-119
- “Robust Regression — Reduce Outlier Effects” on page 9-125
- “Ridge Regression” on page 9-128
- “Lasso and Elastic Net” on page 9-132

- “Partial Least Squares” on page 9-147
- “Linear Mixed-Effects Models” on page 9-153
- “Prepare Data for Linear Mixed-Effects Models” on page 9-158
- “Relationship Between Formula and Design Matrices” on page 9-164
- “Estimating Parameters in Linear Mixed-Effects Models” on page 9-172

## Parametric Regression Analysis

### In this section...

“What Is Parametric Regression?” on page 9-3

“Choose a Regression Function” on page 9-3

“Update Legacy Code with New Fitting Methods” on page 9-4

### What Is Parametric Regression?

Regression is the process of fitting models to data. The models must have numerical responses. For models with categorical responses, see “Parametric Classification” on page 15-2 or “Supervised Learning (Machine Learning) Workflow and Algorithms” on page 16-2. The regression process depends on the model. If a model is parametric, regression estimates the parameters from the data. If a model is linear in the parameters, estimation is based on methods from linear algebra that minimize the norm of a residual vector. If a model is nonlinear in the parameters, estimation is based on search methods from optimization that minimize the norm of a residual vector.

### Choose a Regression Function

You have:	You want:	Use this:
Continuous or categorical predictors, continuous response, linear model	Fitted model coefficients	<code>fitlm</code> . See “Linear Regression” on page 9-12.
Continuous or categorical predictors, continuous response, linear model of unknown complexity	Fitted model and fitted coefficients	<code>stepwiselm</code> . See “Stepwise Regression” on page 9-119.
Continuous or categorical predictors, response possibly with restrictions such as nonnegative or integer-valued, generalized linear model	Fitted generalized linear model coefficients	<code>fitglm</code> or <code>stepwiseglm</code> . See “Generalized Linear Models” on page 10-12.

<b>You have:</b>	<b>You want:</b>	<b>Use this:</b>
Continuous predictors with a continuous nonlinear response, parametrized nonlinear model	Fitted nonlinear model coefficients	<code>fitnlm</code> . See “Nonlinear Regression” on page 11-2.
Continuous predictors, continuous response, linear model	Set of models from ridge, lasso, or elastic net regression	<code>lasso</code> or <code>ridge</code> See “Lasso and Elastic Net” on page 9-132 or “Ridge Regression” on page 9-128.
Correlated continuous predictors, continuous response, linear model	Fitted model and fitted coefficients	<code>plsregress</code> See “Partial Least Squares” on page 9-147.
Continuous or categorical predictors, continuous response, unknown model	Nonparametric model	<code>RegressionTree.fit</code> or <code>fitensemble</code> See “Classification Trees and Regression Trees” on page 16-31 or “Ensemble Methods” on page 16-59.
Categorical predictors only	ANOVA	<code>anova</code> , <code>anova1</code> , <code>anova2</code> , <code>anovan</code>
Continuous predictors, multivariable response, linear model	Fitted multivariate regression model coefficients	<code>mvregress</code>
Continuous predictors, continuous response, mixed-effects model	Fitted mixed-effects model coefficients	<code>nlmefit</code> or <code>nlmefitsa</code> See “Mixed-Effects Models” on page 11-23.

## Update Legacy Code with New Fitting Methods

There are several Statistics Toolbox functions for performing regression. The following sections describe how to replace calls to older functions to new versions:

- “regress into `fitlm`” on page 9-5
- “regstats into `fitlm`” on page 9-5



- “robustfit into fitlm” on page 9-6
- “stepwisefit into stepwiselm” on page 9-6
- “glmfit into fitglm” on page 9-7
- “nlinfit into fitnlm” on page 9-7

## **regress into fitlm**

### **Previous Syntax.**

```
[b,bint,r,rint,stats] = regress(y,X)
```

where  $X$  contains a column of ones.

### **Current Syntax.**

```
mdl = fitlm(X,y)
```

where you do not add a column of ones to  $X$ .

Equivalent values of the previous outputs:

- $b$  — `mdl.Coefficients.Estimate`
- $bint$  — `coefCI(mdl)`
- $r$  — `mdl.Residuals.Raw`
- $rint$  — There is no exact equivalent. Try examining `mdl.Residuals.Studentized` to find outliers.
- $stats$  — `mdl` contains various properties that replace components of `stats`.

## **regstats into fitlm**

### **Previous Syntax.**

```
stats = regstats(y,X,model,whichstats)
```

### **Current Syntax.**

```
mdl = fitlm(X,y,model)
```

Obtain statistics from the properties and methods of `mdl`. For example, see the `mdl.Diagnostics` and `mdl.Residuals` properties.

### **robustfit into fitlm**

#### **Previous Syntax.**

```
[b,stats] = robustfit(X,y,wfun,tune,const)
```

#### **Current Syntax.**

```
mdl = fitlm(X,y,'robust','on') % bisquare
```

Or to use the *wfun* weight and the *tune* tuning parameter:

```
opt.RobustWgtFun = 'wfun';  
opt.Tune = tune; % optional  
mdl = fitlm(X,y,'robust',opt)
```

Obtain statistics from the properties and methods of `mdl`. For example, see the `mdl.Diagnostics` and `mdl.Residuals` properties.

### **stepwisefit into stepwiselm**

#### **Previous Syntax.**

```
[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(X,y,Name,Value)
```

#### **Current Syntax.**

```
mdl = stepwiselm(ds,modelspec,Name,Value)
```

or

```
mdl = stepwiselm(X,y,modelspec,Name,Value)
```

Obtain statistics from the properties and methods of `mdl`. For example, see the `mdl.Diagnostics` and `mdl.Residuals` properties.

## glmfit into fitglm

### Previous Syntax.

```
[b,dev,stats] = glmfit(X,y,distr,param1,val1,...)
```

### Current Syntax.

```
mdl = fitglm(X,y,distr,...)
```

Obtain statistics from the properties and methods of `mdl`. For example, the deviance is `mdl.Deviance`, and to compare `mdl` against a constant model, use `devianceTest(mdl)`.

## nlinfit into fitnlm

### Previous Syntax.

```
[beta,r,J,COVB,mse] = nlinfit(X,y,fun,beta0,options)
```

### Current Syntax.

```
mdl = fitnlm(X,y,fun,beta0,'Options',options)
```

Equivalent values of the previous outputs:

- `beta` — `mdl.Coefficients.Estimate`
- `r` — `mdl.Residuals.Raw`
- `covb` — `mdl.CoefficientCovariance`
- `mse` — `mdl.mse`

`mdl` does not provide the Jacobian (`J`) output. The primary purpose of `J` was to pass it into `nlparci` or `nlpredci` to obtain confidence intervals for the estimated coefficients (parameters) or predictions. Obtain those confidence intervals as:

```
parci = coefCI(mdl)  
[pred,predci] = predict(mdl)
```

## What Are Linear Regression Models?

Regression models describe the relationship between a *dependent variable*,  $y$ , and *independent variable* or variables,  $X$ . The dependent variable is also called the *response variable*. Independent variables are also called *explanatory* or *predictor variables*. Continuous predictor variables might be called *covariates*, whereas categorical predictor variables might be also referred to as *factors*. The matrix,  $X$ , of observations on predictor variables is usually called the *design matrix*.

A multiple linear regression model is

$$y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \cdots + \beta_p X_{ip} + \varepsilon_i, \quad i = 1, \dots, n,$$

where

- $y_i$  is the  $i$ th response.
- $\beta_k$  is the  $k$ th coefficient, where  $\beta_0$  is the constant term in the model. Sometimes, design matrices might include information about the constant term. However, `fitlm` or `stepwiselm` by default includes a constant term in the model, so you must not enter a column of 1s into your design matrix  $X$ .
- $X_{ij}$  is the  $i$ th observation on the  $j$ th predictor variable,  $j = 1, \dots, p$ .
- $\varepsilon_i$  is the  $i$ th noise term, that is, random error.

In general, a linear regression model can be a model of the form

$$y_i = \beta_0 + \sum_{k=1}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + \varepsilon_i, \quad i = 1, \dots, n,$$

where  $f(\cdot)$  is a scalar-valued function of the independent variables,  $X_{ij}$ s. The functions,  $f(X)$ , might be in any form including nonlinear functions or polynomials. The linearity, in the linear regression models, refers to the linearity of the coefficients  $\beta_k$ . That is, the response variable,  $y$ , is a linear function of the coefficients,  $\beta_k$ .

Some examples of linear models are:

$$y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{3i} + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{1i}^3 + \beta_4 X_{2i}^2 + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{1i} X_{2i} + \beta_4 \log X_{3i} + \varepsilon_i$$

The following, however, are not linear models since they are not linear in the unknown coefficients,  $\beta_k$ .

$$\log y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 X_{1i} + \frac{1}{\beta_2 X_{2i}} + e^{\beta_3 X_{1i} X_{2i}} + \varepsilon_i$$

The usual assumptions for linear regression models are:

- The noise terms,  $\varepsilon_i$ , are uncorrelated.
- The noise terms,  $\varepsilon_i$ , have independent and identical normal distributions with mean zero and constant variance,  $\sigma^2$ . Thus

$$\begin{aligned} E(y_i) &= E\left(\sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + \varepsilon_i\right) \\ &= \sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + E(\varepsilon_i) \\ &= \sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) \end{aligned}$$

and

$$V(y_i) = V\left(\sum_{k=0}^K \beta_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}) + \varepsilon_i\right) = V(\varepsilon_i) = \sigma^2$$

So the variance of  $y_i$  is the same for all levels of  $X_{ij}$ .

- The responses  $y_i$  are uncorrelated.

The fitted linear function is

$$\hat{y}_i = b_0 + \sum_{k=1}^K b_k f_k(X_{i1}, X_{i2}, \dots, X_{ip}), \quad i = 1, \dots, n,$$

where  $\hat{y}_i$  is the estimated response and  $b_k$ s are the fitted coefficients. The coefficients are estimated so as to minimize the mean squared difference between the prediction vector  $bf(X)$  and the true response vector  $y$ , that is  $\hat{y} - y$ . This method is called the *method of least squares*. Under the assumptions on the noise terms, these coefficients also maximize the likelihood of the prediction vector.

In a linear regression model of the form  $y = \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p$ , the coefficient  $\beta_k$  expresses the impact of a one-unit change in predictor variable,  $X_j$ , on the mean of the response,  $E(y)$  provided that all other variables are held constant. The sign of the coefficient gives the direction of the effect. For example, if the linear model is  $E(y) = 1.8 - 2.35X_1 + X_2$ , then  $-2.35$  indicates a 2.35 unit decrease in the mean response with a one-unit increase in  $X_1$ , given  $X_2$  is held constant. If the model is  $E(y) = 1.1 + 1.5X_1^2 + X_2$ , the coefficient of  $X_1^2$  indicates a 1.5 unit increase in the mean of  $Y$  with a one-unit increase in  $X_1^2$  given all else held constant. However, in the case of  $E(y) = 1.1 + 2.1X_1 + 1.5X_1^2$ , it is difficult to interpret the coefficients similarly, since it is not possible to hold  $X_1$  constant when  $X_1^2$  changes or vice versa.

## References

- [1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. IRWIN, The McGraw-Hill Companies, Inc., 1996.
- [2] Seber, G. A. F. *Linear Regression Analysis*. Wiley Series in Probability and Mathematical Statistics. John Wiley and Sons, Inc., 1977.

## See Also

LinearModel | fitlm | stepwiselm

## Related Examples

- “Interpret Linear Regression Results” on page 9-64
- “Regression Using Dataset Arrays” on page 9-51
- “Linear Regression with Interaction Effects” on page 9-54
- “Regression with Categorical Covariates” on page 2-59

- “Linear Regression Workflow” on page 9-44

## Linear Regression

### In this section...

“Prepare Data” on page 9-12

“Choose a Fitting Method” on page 9-14

“Choose a Model or Range of Models” on page 9-15

“Fit Model to Data” on page 9-21

“Examine Quality and Adjust the Fitted Model” on page 9-21

“Predict or Simulate Responses to New Data” on page 9-40

“Share Fitted Models” on page 9-43

“Linear Regression Workflow” on page 9-44

### Prepare Data

To begin fitting a regression, put your data into a form that fitting functions expect. All regression techniques begin with input data in an array  $X$  and response data in a separate vector  $y$ , or input data in a dataset array  $ds$  and response data as a column in  $ds$ . Each row of the input data represents one observation. Each column represents one predictor (variable).

For a dataset array  $ds$ , indicate the response variable with the 'ResponseVar' name-value pair:

```
mdl = fitlm(ds, 'ResponseVar', 'BloodPressure');  
% or  
mdl = fitglm(ds, 'ResponseVar', 'BloodPressure');
```

The response variable is the last column by default.

You can use numeric *categorical* predictors. A categorical predictor is one that takes values from a fixed set of possibilities.

- For a numeric array  $X$ , indicate the categorical predictors using the 'Categorical' name-value pair. For example, to indicate that predictors 2 and 3 out of six are categorical:



```
mdl = fitlm(X,y,'Categorical',[2,3]);
% or
mdl = fitglm(X,y,'Categorical',[2,3]);
% or equivalently
mdl = fitlm(X,y,'Categorical',logical([0 1 1 0 0 0]));
```

- For a dataset array `ds`, fitting functions assume that these data types are categorical:
  - Logical
  - Categorical (nominal or ordinal)
  - String or character array

If you want to indicate that a numeric predictor is categorical, use the 'Categorical' name-value pair.

Represent missing numeric data as NaN. To represent missing data for other data types, see “Missing Group Values” on page 2-53.

### **Dataset Array for Input and Response Data**

For example, to create a dataset array from an Excel spreadsheet:

```
ds = dataset('XLSFile','hospital.xls',...
            'ReadObsNames',true);
```

To create a dataset array from workspace variables:

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
```

### **Numeric Matrix for Input Data, Numeric Vector for Response**

For example, to create numeric arrays from workspace variables:

```
load carsmall
X = [Weight Horsepower Cylinders Model_Year];
y = MPG;
```

To create numeric arrays from an Excel spreadsheet:

```
[X Xnames] = xlsread('hospital.xls');  
y = X(:,4); % response y is systolic pressure  
X(:,4) = []; % remove y from the X matrix
```

Notice that the nonnumeric entries, such as `sex`, do not appear in `X`.

## Choose a Fitting Method

There are three ways to fit a model to data:

- “Least-Squares Fit” on page 9-14
- “Robust Fit” on page 9-14
- “Stepwise Fit” on page 9-14

### Least-Squares Fit

Use `fitlm` to construct a least-squares fit of a model to the data. This method is best when you are reasonably certain of the model’s form, and mainly need to find its parameters. This method is also useful when you want to explore a few models. The method requires you to examine the data manually to discard outliers, though there are techniques to help (see “Residuals — Model Quality for Training Data” on page 9-25).

### Robust Fit

Use `fitlm` with the `RobustOpts` name-value pair to create a model that is little affected by outliers. Robust fitting saves you the trouble of manually discarding outliers. However, `step` does not work with robust fitting. This means that when you use robust fitting, you cannot search stepwise for a good model.

### Stepwise Fit

Use `stepwiselm` to find a model, and fit parameters to the model. `stepwiselm` starts from one model, such as a constant, and adds or subtracts terms one at a time, choosing an optimal term each time in a greedy fashion, until it cannot improve further. Use stepwise fitting to find a good model, which is one that has only relevant terms.

The result depends on the starting model. Usually, starting with a constant model leads to a small model. Starting with more terms can lead to a more complex model, but one that has lower mean squared error. See “Compare large and small stepwise models” on page 9-119.

You cannot use robust options along with stepwise fitting. So after a stepwise fit, examine your model for outliers (see “Residuals — Model Quality for Training Data” on page 9-25).

## Choose a Model or Range of Models

There are several ways of specifying a model for linear regression. Use whichever you find most convenient.

- “Brief String” on page 9-16
- “Terms Matrix” on page 9-16
- “Formula” on page 9-20

For `fitlm`, the model specification you give is the model that is fit. If you do not give a model specification, the default is `'linear'`.

For `stepwiselm`, the model specification you give is the starting model, which the stepwise procedure tries to improve. If you do not give a model specification, the default starting model is `'constant'`, and the default upper bounding model is `'interactions'`. Change the upper bounding model using the `Upper` name-value pair.

---

**Note** There are other ways of selecting models, such as using `lasso`, `lassoglm`, `sequentialfs`, or `plsregress`.

---

**Brief String**

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

For example, to specify an interaction model using `fitlm` with matrix predictors:

```
mdl = fitlm(X,y,'interactions');
```

To specify a model using `stepwiselm` and a dataset array `ds` of predictors, suppose you want to start from a constant and have a linear model upper bound. Assume the response variable in `ds` is in the third column.

```
mdl2 = stepwiselm(ds,'constant',...
    'Upper','linear','ResponseVar',3);
```

**Terms Matrix**

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =  
  
    0    0    0    0  
    0    1    0    0  
    0    0    1    0  
    0    0    0    1
```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall  
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =  
  
    0    0    0  
    1    0    0  
    0    1    0  
    1    1    0  
    0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

```
Linear regression model:  
y ~ 1 + x1*x2 + x2^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

```
mdl =
```

Linear regression model:

$$y \sim 1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for a model specification is a string of the form

`'Y ~ terms'`,

- *Y* is the response name.
- *terms* contains
  - Variable names
  - + to include the next variable
  - - to exclude the next variable
  - : to define an interaction, a product of terms
  - \* to define an interaction and all lower-order terms
  - ^ to raise the predictor to a power, exactly as in \* repeated, so ^ includes lower order terms as well
  - () to group terms

---

**Tip** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include -1 in the formula.

---

Examples:

`'Y ~ A + B + C'` is a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a  $B^2$  term.

`'Y ~ A + B^2 + C'` is the same as the previous example, since  $B^2$  includes a B term.

`'Y ~ A + B + C + A:B'` includes an  $A*B$  term.

`'Y ~ A*B + C'` is the same as the previous example, since  $A*B = A + B + A:B$ .

`'Y ~ A*B*C - A:B:C'` has all interactions among A, B, and C, except the three-way interaction.

`'Y ~ A*(B + C + D)'` has all linear terms, plus products of A with each of the other variables.



For example, to specify an interaction model using `fitlm` with matrix predictors:

```
mdl = fitlm(X,y,'y ~ x1*x2*x3 - x1:x2:x3');
```

To specify a model using `stepwiselm` and a dataset array `ds` of predictors, suppose you want to start from a constant and have a linear model upper bound. Assume the response variable in `ds` is named `'y'`, and the predictor variables are named `'x1'`, `'x2'`, and `'x3'`.

```
mdl2 = stepwiselm(ds,'y ~ 1','Upper','y ~ x1 + x2 + x3');
```

## Fit Model to Data

The most common optional arguments for fitting:

- For robust regression in `fitlm`, set the `'RobustOpts'` name-value pair to `'on'`.
- Specify an appropriate upper bound model in `stepwiselm`, such as set `'Upper'` to `'linear'`.
- Indicate which variables are categorical using the `'CategoricalVars'` name-value pair. Provide a vector with column numbers, such as `[1 6]` to specify that predictors 1 and 6 are categorical. Alternatively, give a logical vector the same length as the data columns, with a 1 entry indicating that variable is categorical. If there are seven predictors, and predictors 1 and 6 are categorical, specify `logical([1,0,0,0,0,1,0])`.
- For a dataset array, specify the response variable using the `'ResponseVar'` name-value pair. The default is the last column in the array.

For example,

```
mdl = fitlm(X,y,'linear',...
    'RobustOpts','on','CategoricalVars',3);
mdl2 = stepwiselm(ds,'constant',...
    'ResponseVar','MPG','Upper','quadratic');
```

## Examine Quality and Adjust the Fitted Model

After fitting a model, examine the result.

- “Model Display” on page 9-22

- “ANOVA” on page 9-23
- “Diagnostic Plots” on page 9-24
- “Residuals — Model Quality for Training Data” on page 9-25
- “Plots to Understand Predictor Effects” on page 9-30
- “Plots to Understand Terms Effects” on page 9-35
- “Change Models” on page 9-37

### Model Display

A linear regression model shows several diagnostics when you enter its name or enter `disp mdl`. This display gives some of the basic information to check whether the fitted model represents the data adequately.

For example, fit a linear model to data constructed with two out of five predictors not present and with no intercept term:

```
X = randn(100,5);
y = X*[1;0;3;0;-1]+randn(100,1);
mdl = fitlm(X,y)
```

```
mdl =
```

Linear regression model:

$$y \sim 1 + x_1 + x_2 + x_3 + x_4 + x_5$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.038164	0.099458	0.38372	0.70205
x1	0.92794	0.087307	10.628	8.5494e-18
x2	-0.075593	0.10044	-0.75264	0.45355
x3	2.8965	0.099879	29	1.1117e-48
x4	0.045311	0.10832	0.41831	0.67667
x5	-0.99708	0.11799	-8.4504	3.593e-13

Number of observations: 100, Error degrees of freedom: 94

Root Mean Squared Error: 0.972

R-squared: 0.93, Adjusted R-Squared 0.926

F-statistic vs. constant model: 248, p-value = 1.5e-52

Notice that:

- The display contains the estimated values of each coefficient in the Estimate column. These values are reasonably near the true values `[0;1;0;3;0;-1]`.
- There is a standard error column for the coefficient estimates.
- The reported pValue (which are derived from the  $t$  statistics under the assumption of normal errors) for predictors 1, 3, and 5 are extremely small. These are the three predictors that were used to create the response data  $y$ .
- The pValue for (Intercept),  $x_2$  and  $x_4$  are much larger than 0.01. These three predictors were not used to create the response data  $y$ .
- The display contains  $R^2$ , adjusted  $R^2$ , and  $F$  statistics.

## ANOVA

To examine the quality of the fitted model, consult an ANOVA table. For example, use `anova` on a linear model with five predictors:

```
X = randn(100,5);
y = X*[1;0;3;0;-1]+randn(100,1);
mdl = fitlm(X,y);
tbl = anova(mdl)
```

```
tbl =
```

	SumSq	DF	MeanSq	F	pValue
x1	106.62	1	106.62	112.96	8.5494e-18
x2	0.53464	1	0.53464	0.56646	0.45355
x3	793.74	1	793.74	840.98	1.1117e-48
x4	0.16515	1	0.16515	0.17498	0.67667
x5	67.398	1	67.398	71.41	3.593e-13
Error	88.719	94	0.94382		

This table gives somewhat different results than the default display (see “Model Display” on page 9-22). The table clearly shows that the effects of  $x_2$  and  $x_4$  are not significant. Depending on your goals, consider removing  $x_2$  and  $x_4$  from the model.

## Diagnostic Plots

Diagnostic plots help you identify outliers, and see other problems in your model or fit. For example, load the `carsmall` data, and make a model of MPG as a function of `Cylinders` (nominal) and `Weight`:

```
load carsmall
ds = dataset(Weight,MPG,Cylinders);
ds.Cylinders = ordinal(ds.Cylinders);
mdl = fitlm(ds,'MPG ~ Cylinders*Weight + Weight^2');
```

Make a leverage plot of the data and model.

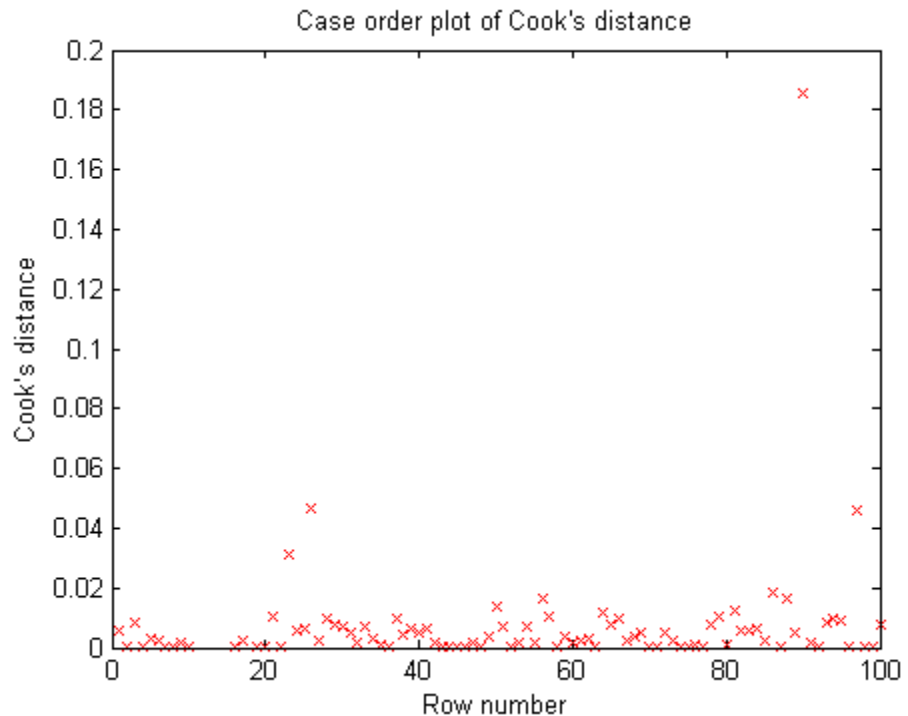
```
plotDiagnostics(mdl)
```



There are a few points with high leverage. But this plot does not reveal whether the high-leverage points are outliers.

Look for points with large Cook's distance.

```
plotDiagnostics(md1, 'cookd')
```



There is one point with large Cook's distance. Identify it and remove it from the model. You can use the Data Cursor to click the outlier and identify it, or identify it programmatically:

```
[~,larg] = max(md1.Diagnostics.CooksDistance);
md12 = fitlm(ds,'MPG ~ Cylinders*Weight + Weight^2',...
    'Exclude',larg);
```

## Residuals – Model Quality for Training Data

There are several residual plots to help you discover errors, outliers, or correlations in the model or data. The simplest residual plots are the default histogram plot, which shows the range of the residuals and their frequencies,

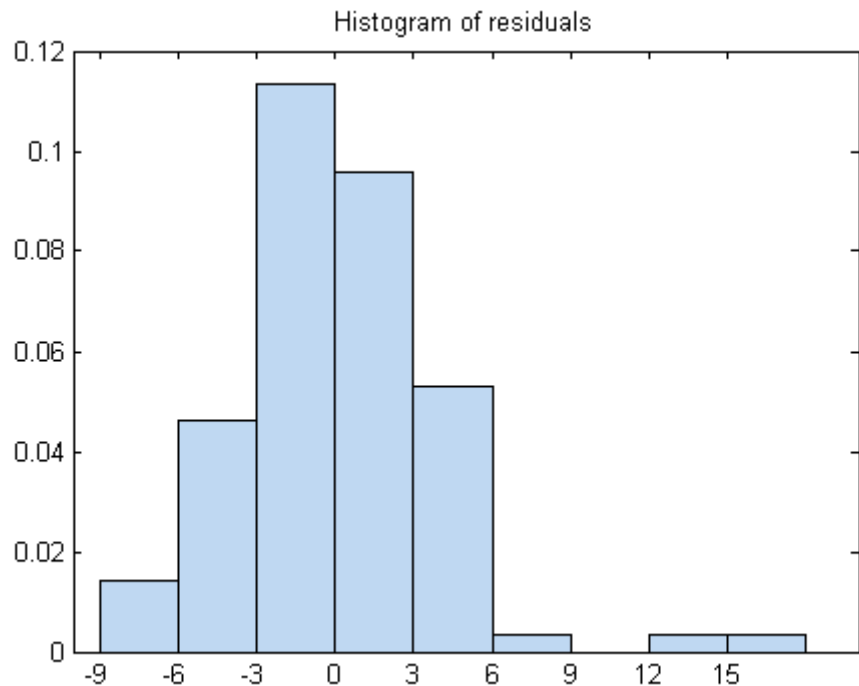
and the probability plot, which shows how the distribution of the residuals compares to a normal distribution with matched variance.

Load the `carsmall` data, and make a model of MPG as a function of Cylinders (nominal) and Weight:

```
load carsmall
ds = dataset(Weight,MPG,Cylinders);
ds.Cylinders = ordinal(ds.Cylinders);
mdl = fitlm(ds,'MPG ~ Cylinders*Weight + Weight^2');
```

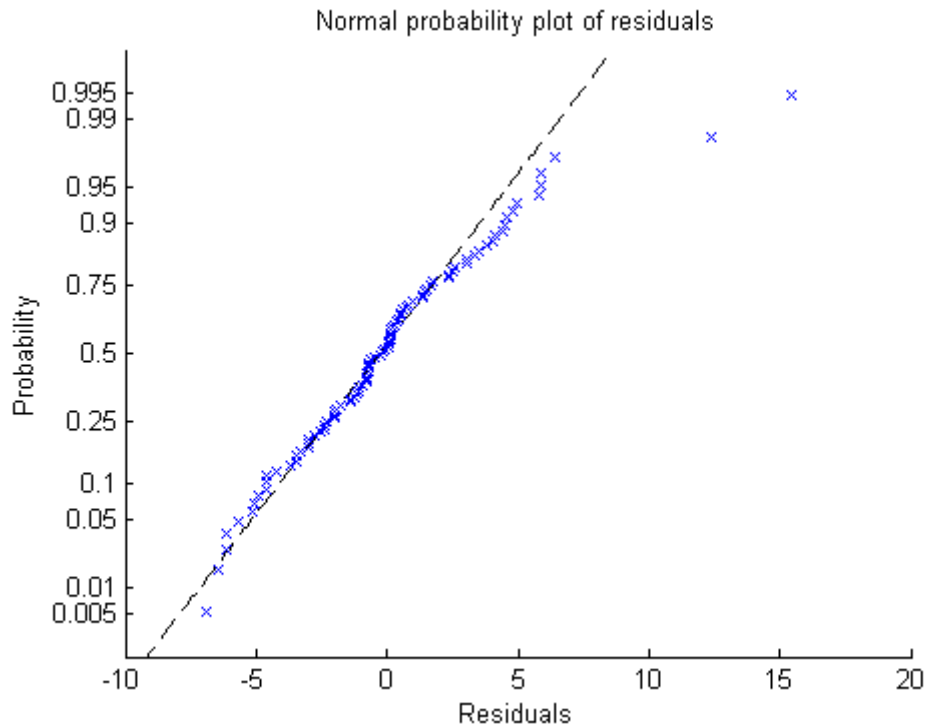
Examine the residuals:

```
plotResiduals(mdl)
```



The observations above 12 are potential outliers.

```
plotResiduals mdl, 'probability'
```



The two potential outliers appear on this plot as well. Otherwise, the probability plot seems reasonably straight, meaning a reasonable fit to normally distributed residuals.

You can identify the two outliers and remove them from the data:

```
out1 = find(mdl.Residuals.Raw > 12)
```

```
out1 =
```

```
90
```

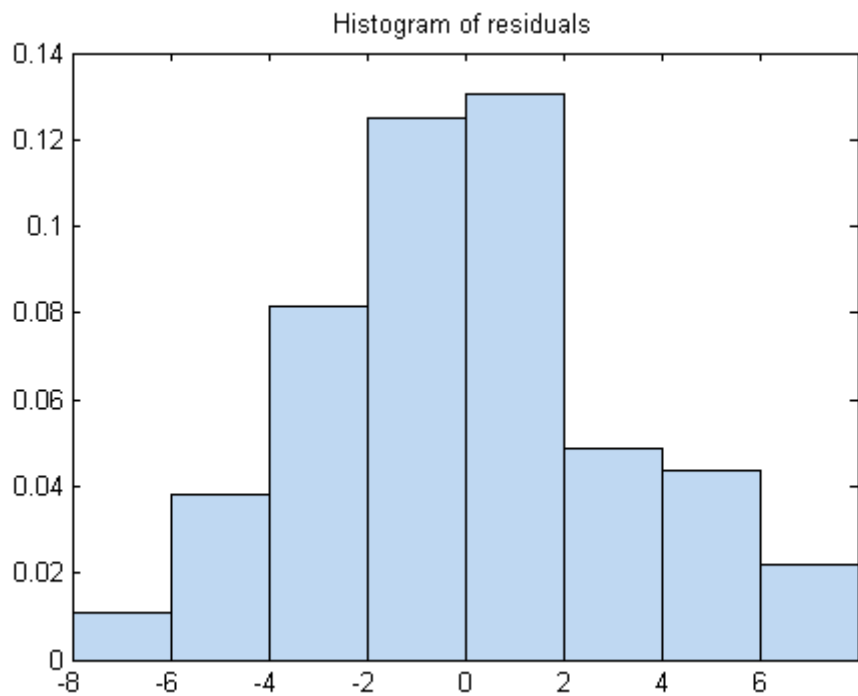
```
97
```

To remove the outliers, use the Exclude name-value pair:

```
md12 = fitlm(ds,'MPG ~ Cylinders*Weight + Weight^2',...  
            'Exclude',out1);
```

Examine a residuals plot of md12:

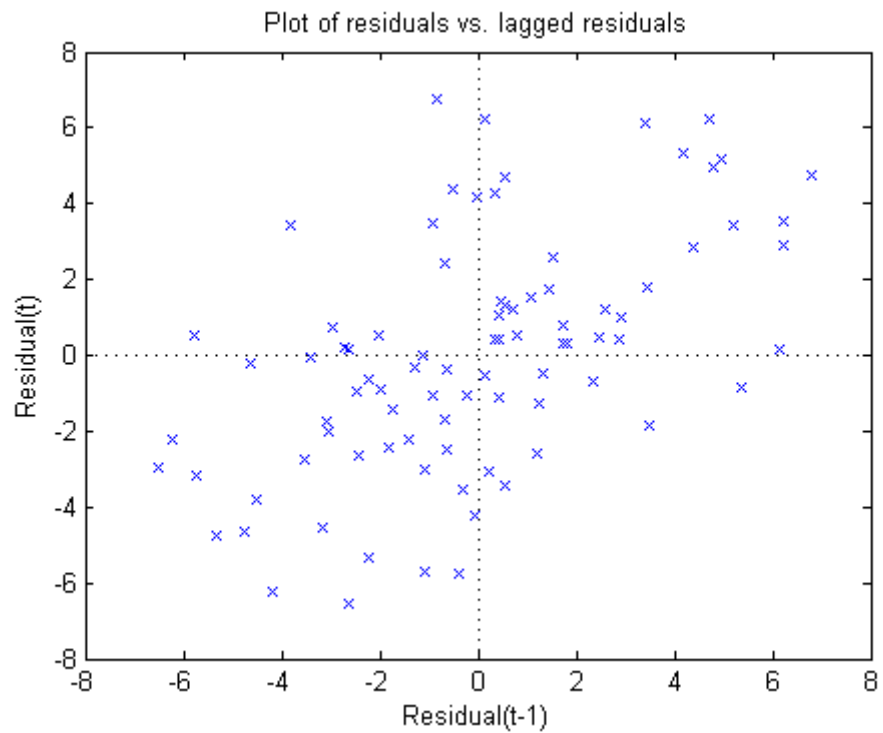
```
plotResiduals(md12)
```



The new residuals plot looks fairly symmetric, without obvious problems. However, there might be some serial correlation among the residuals. Create a new plot to see if such an effect exists.

```
plotResiduals(md12,'lagged')
```

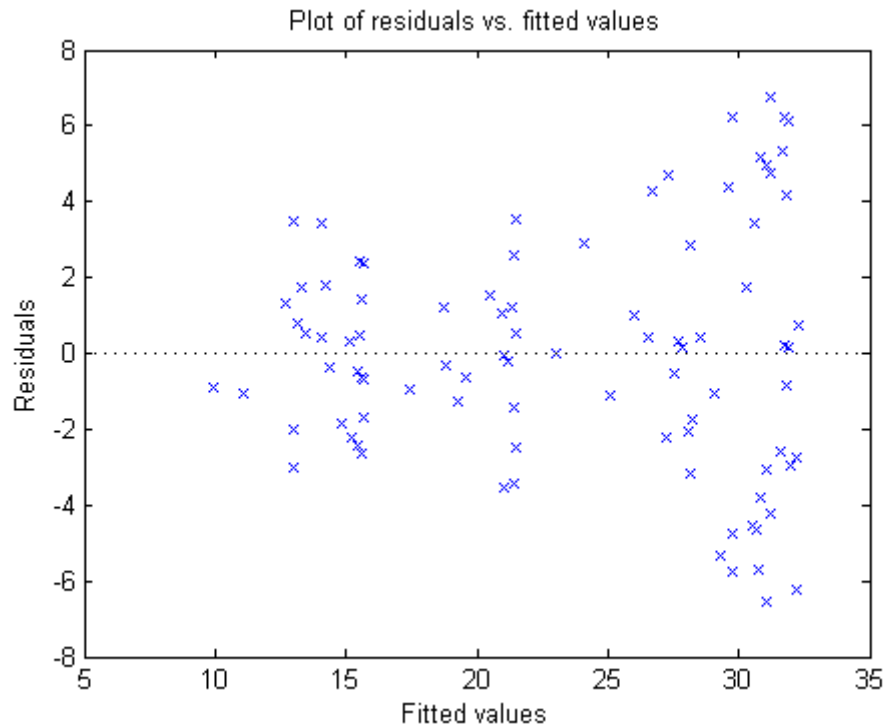




The scatter plot shows many more crosses in the upper-right and lower-left quadrants than in the other two quadrants, indicating positive serial correlation among the residuals.

Another potential issue is when residuals are large for large observations. See if the current model has this issue.

```
plotResiduals(md12, 'fitted')
```



There is some tendency for larger fitted values to have larger residuals. Perhaps the model errors are proportional to the measured values.

### Plots to Understand Predictor Effects

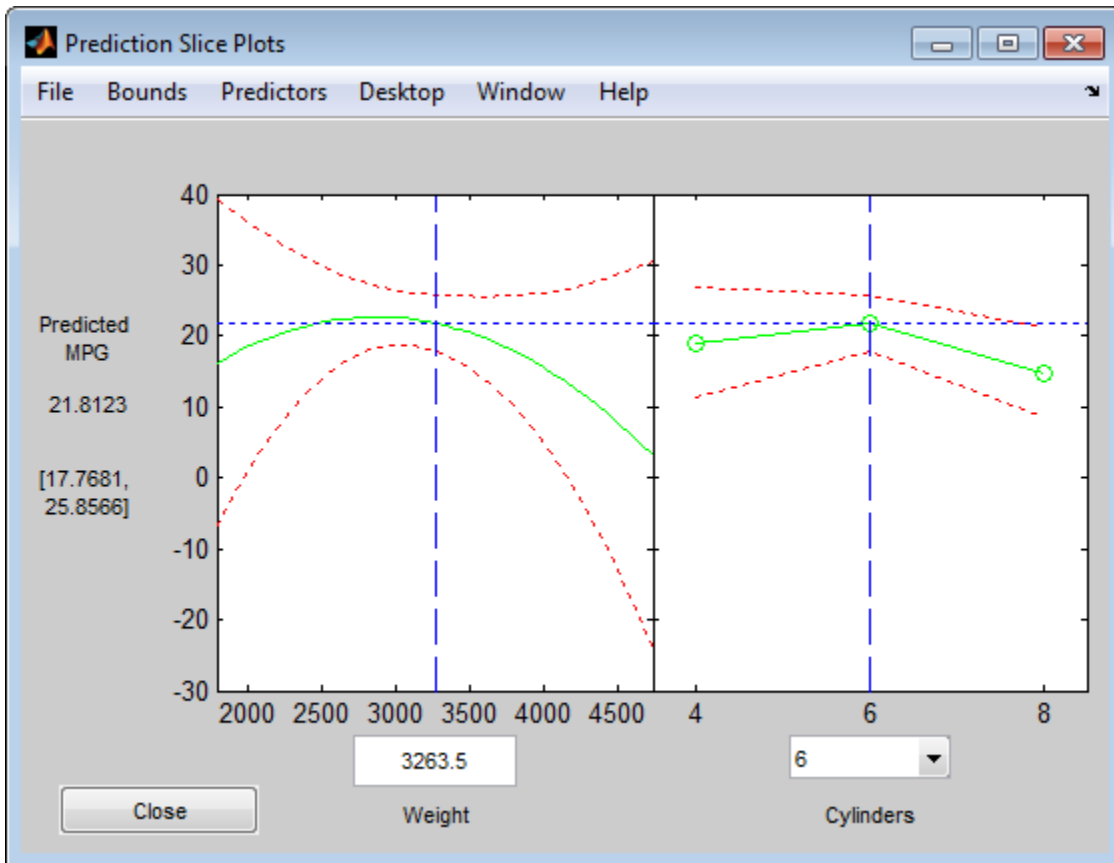
This example shows how to understand the effect each predictor has on a regression model using a variety of available plots.

- 1 Create a model of mileage from some predictors in the carsmall data.

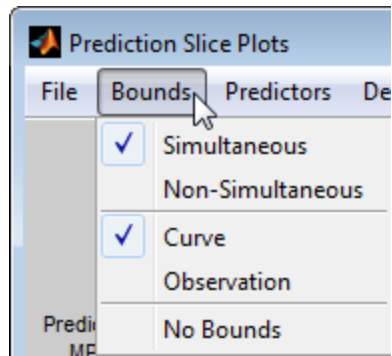
```
load carsmall
ds = dataset(Weight,MPG,Cylinders);
ds.Cylinders = ordinal(ds.Cylinders);
mdl = fitlm(ds,'MPG ~ Cylinders*Weight + Weight^2');
```

- 2** Examine a slice plot of the responses. This displays the effect of each predictor separately.

```
plotSlice mdl
```

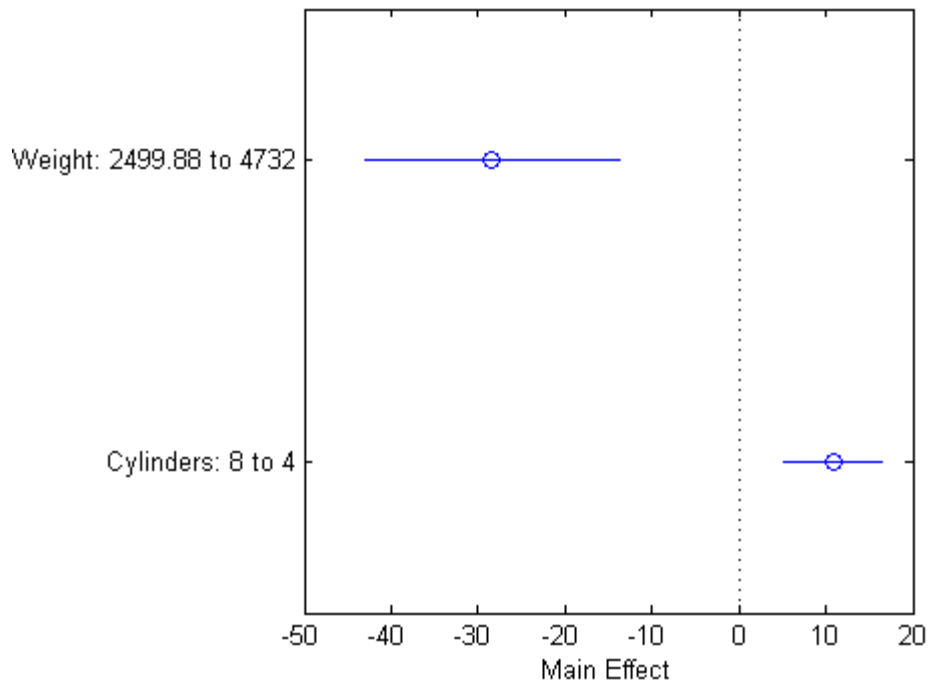


You can drag the individual predictor values, which are represented by dashed blue vertical lines. You can also choose between simultaneous and non-simultaneous confidence bounds, which are represented by dashed red curves.



- 3 Use an effects plot to show another view of the effect of predictors on the response.

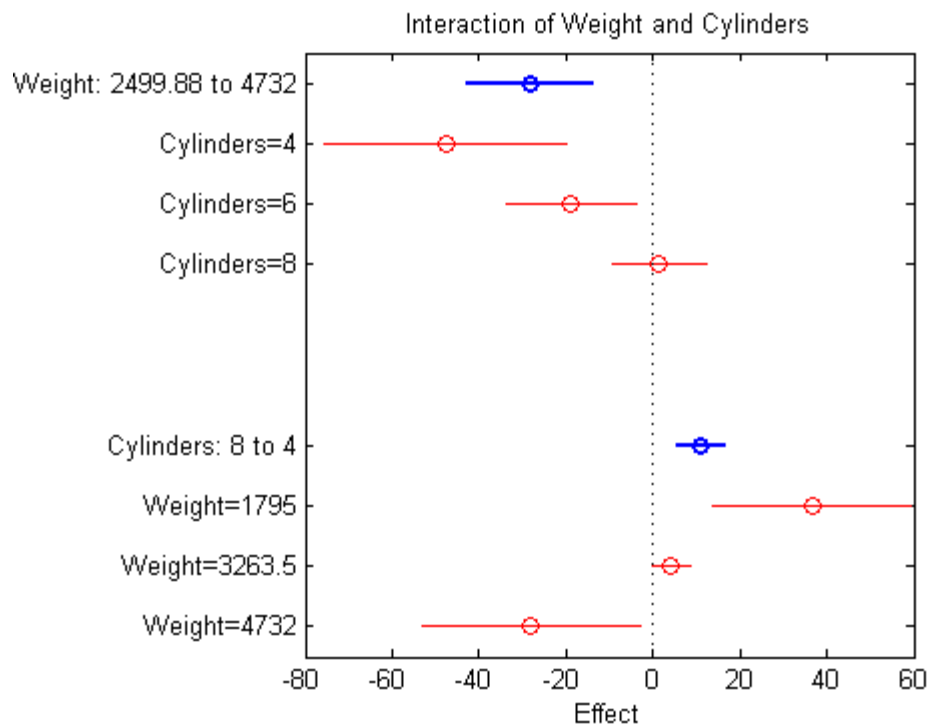
```
plotEffects mdl
```



This plot shows that changing Weight from about 2500 to 4732 lowers MPG by about 30 (the location of the upper blue circle). It also shows that changing the number of cylinders from 8 to 4 raises MPG by about 10 (the lower blue circle). The horizontal blue lines represent confidence intervals for these predictions. The predictions come from averaging over one predictor as the other is changed. In cases such as this, where the two predictors are correlated, be careful when interpreting the results.

- 4 Instead of viewing the effect of averaging over a predictor as the other is changed, examine the joint interaction in an interaction plot.

```
plotInteraction(md1, 'Weight', 'Cylinders')
```

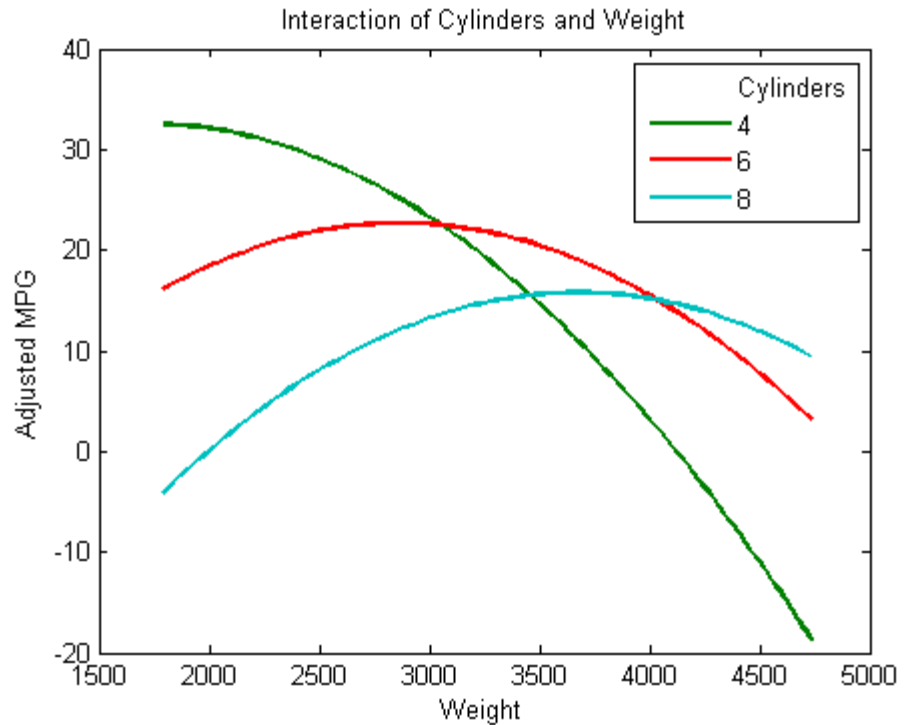


The interaction plot shows the effect of changing one predictor with the other held fixed. In this case, the plot is much more informative. It shows, for example, that lowering the number of cylinders in a relatively light car

(Weight = 1795) leads to an increase in mileage, but lowering the number of cylinders in a relatively heavy car (Weight = 4732) leads to a decrease in mileage.

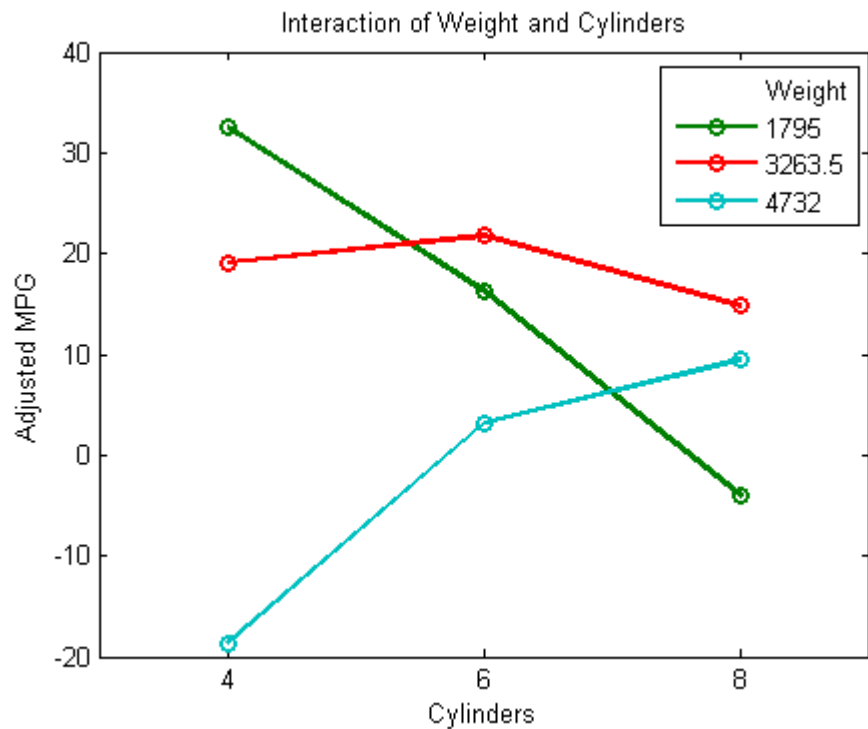
- For an even more detailed look at the interactions, look at an interaction plot with predictions. This plot holds one predictor fixed while varying the other, and plots the effect as a curve. Look at the interactions for various fixed numbers of cylinders.

```
plotInteraction(md1, 'Cylinders', 'Weight', 'predictions')
```



Now look at the interactions with various fixed levels of weight.

```
plotInteraction(md1, 'Weight', 'Cylinders', 'predictions')
```



### Plots to Understand Terms Effects

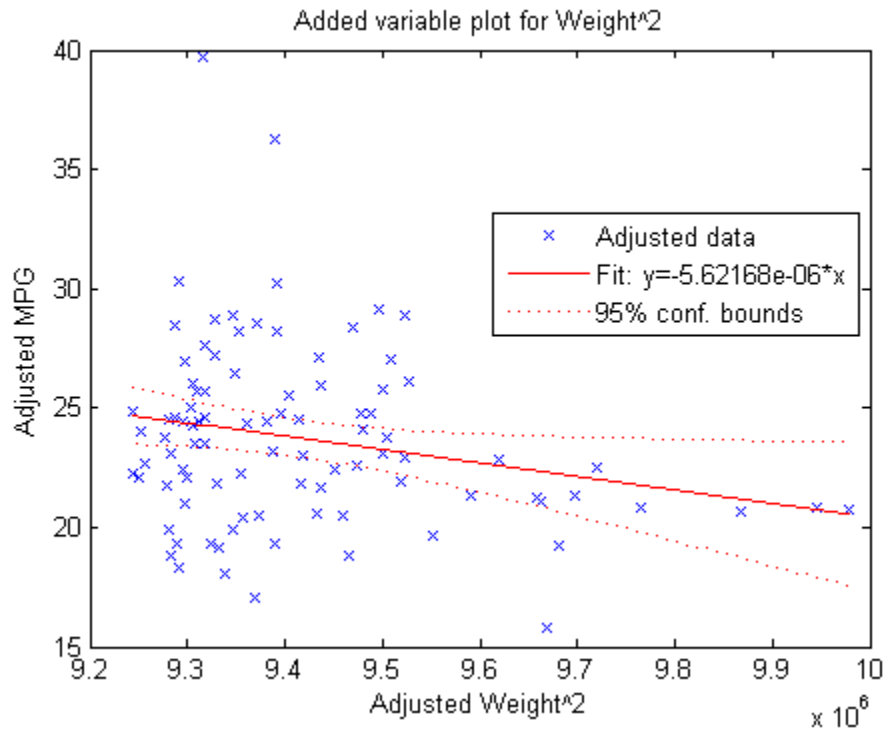
This example shows how to understand the effect of each term in a regression model using a variety of available plots.

- 1 Create a model of mileage from some predictors in the carsmall data.

```
load carsmall
ds = dataset(Weight,MPG,Cylinders);
ds.Cylinders = ordinal(ds.Cylinders);
mdl = fitlm(ds,'MPG ~ Cylinders*Weight + Weight^2');
```

- 2 Create an added variable plot with  $\text{Weight}^2$  as the added variable.

```
plotAdded(mdl, 'Weight^2')
```

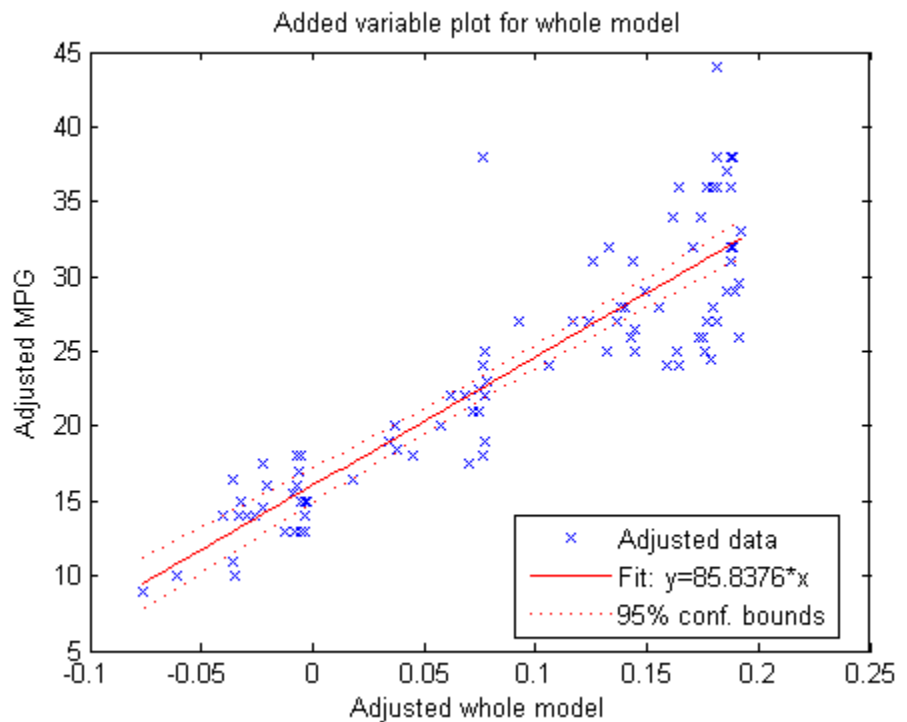


This plot shows the results of fitting both  $\text{Weight}^2$  and MPG to the terms other than  $\text{Weight}^2$ . The reason to use `plotAdded` is to understand what additional improvement in the model you get by adding  $\text{Weight}^2$ . The coefficient of a line fit to these points is the coefficient of  $\text{Weight}^2$  in the full model. The  $\text{Weight}^2$  predictor is just over the edge of significance ( $p\text{Value} < 0.05$ ) as you can see in the coefficients table display. You can see that in the plot as well. The confidence bounds look like they could not contain a horizontal line (constant  $y$ ), so a zero-slope model is not consistent with the data.

- 3 Create an added variable plot for the model as a whole.

```
plotAdded(md1)
```





The model as a whole is very significant, so the bounds don't come close to containing a horizontal line. The slope of the line is the slope of a fit to the predictors projected onto their best-fitting direction, or in other words, the norm of the coefficient vector.

## Change Models

There are two ways to change a model:

- `step` — Add or subtract terms one at a time, where `step` chooses the most important term to add or remove.
- `addTerms` and `removeTerms` — Add or remove specified terms. Give the terms in any of the forms described in "Choose a Model or Range of Models" on page 9-15.

If you created a model using `stepwiselm`, `step` can have an effect only if you give different upper or lower models. `step` does not work when you fit a model using `RobustOpts`.

For example, start with a linear model of mileage from the `carbig` data:

```
load carbig
ds = dataset(Acceleration,Displacement,Horsepower,Weight,MPG);
mdl = fitlm(ds,'linear','ResponseVar','MPG')
```

```
mdl =
```

Linear regression model:

```
MPG ~ 1 + Acceleration + Displacement + Horsepower + Weight
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	45.251	2.456	18.424	7.0721e-55
Acceleration	-0.023148	0.1256	-0.1843	0.85388
Displacement	-0.0060009	0.0067093	-0.89441	0.37166
Horsepower	-0.043608	0.016573	-2.6312	0.008849
Weight	-0.0052805	0.00081085	-6.5123	2.3025e-10

Number of observations: 392, Error degrees of freedom: 387

Root Mean Squared Error: 4.25

R-squared: 0.707, Adjusted R-Squared 0.704

F-statistic vs. constant model: 233, p-value = 9.63e-102

Try to improve the model using `step` for up to 10 steps:

```
mdl1 = step(mdl,'NSteps',10)
```

```
1. Adding Displacement:Horsepower, FStat = 87.4802, pValue = 7.05273e-19
```

```
mdl1 =
```

Linear regression model:

```
MPG ~ 1 + Acceleration + Weight + Displacement*Horsepower
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	61.285	2.8052	21.847	1.8593e-69
Acceleration	-0.34401	0.11862	-2.9	0.0039445
Displacement	-0.081198	0.010071	-8.0623	9.5014e-15
Horsepower	-0.24313	0.026068	-9.3265	8.6556e-19
Weight	-0.0014367	0.00084041	-1.7095	0.088166
Displacement:Horsepower	0.00054236	5.7987e-05	9.3531	7.0527e-19

Number of observations: 392, Error degrees of freedom: 386

Root Mean Squared Error: 3.84

R-squared: 0.761, Adjusted R-Squared 0.758

F-statistic vs. constant model: 246, p-value = 1.32e-117

**step stopped after just one change.**

To try to simplify the model, remove the Acceleration and Weight terms from mdl1:

```
mdl2 = removeTerms(mdl1, 'Acceleration + Weight')
```

```
mdl2 =
```

Linear regression model:

```
MPG ~ 1 + Displacement*Horsepower
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	53.051	1.526	34.765	3.0201e-121
Displacement	-0.098046	0.0066817	-14.674	4.3203e-39
Horsepower	-0.23434	0.019593	-11.96	2.8024e-28
Displacement:Horsepower	0.00058278	5.193e-05	11.222	1.6816e-25

Number of observations: 392, Error degrees of freedom: 388

Root Mean Squared Error: 3.94

R-squared: 0.747, Adjusted R-Squared 0.745

F-statistic vs. constant model: 381, p-value = 3e-115

mdl2 uses just Displacement and Horsepower, and has nearly as good a fit to the data as mdl1 in the Adjusted R-Squared metric.

### Predict or Simulate Responses to New Data

There are three ways to use a linear model to predict or simulate the response to new data:

- “predict” on page 9-40
- “feval” on page 9-41
- “random” on page 9-42

#### predict

This example shows how to predict and obtain confidence intervals on the predictions using the predict method.

- 1 Load the carbig data and make a default linear model of the response MPG to the Acceleration, Displacement, Horsepower, and Weight predictors.

```
load carbig
X = [Acceleration,Displacement,Horsepower,Weight];
mdl = fitlm(X,MPG);
```

- 2 Create a three-row array of predictors from the minimal, mean, and maximal values. There are some NaN values, so use functions that ignore NaN values.

```
Xnew = [nanmin(X);nanmean(X);nanmax(X)]; % new data
```

- 3 Find the predicted model responses and confidence intervals on the predictions.

```
[NewMPG NewMPGCI] = predict(mdl,Xnew)
```

```
NewMPG =
    34.1345
    23.4078
     4.7751
```

```
NewMPGCI =
    31.6115    36.6575
    22.9859    23.8298
     0.6134     8.9367
```

The confidence bound on the mean response is narrower than those for the minimum or maximum responses, which is quite sensible.

## feval

When you construct a model from a dataset array, `feval` is often more convenient for predicting mean responses than `predict`. However, `feval` does not provide confidence bounds.

This example shows how to predict mean responses using the `feval` method.

- 1 Load the `carbig` data and make a default linear model of the response MPG to the Acceleration, Displacement, Horsepower, and Weight predictors.

```
load carbig
ds = dataset(Acceleration,Displacement,Horsepower,Weight,MPG);
mdl = fitlm(ds,'linear','ResponseVar','MPG');
```

- 2 Create a three-row array of predictors from the minimal, mean, and maximal values. There are some NaN values, so use functions that ignore NaN values.

```
X = [Acceleration,Displacement,Horsepower,Weight];
Xnew = [nanmin(X);nanmean(X);nanmax(X)]; % new data
```

The `Xnew` array is not a dataset array. It has the right number of columns for prediction, so `feval` can use it for predictions.

- 3 Find the predicted model responses.

```
NewMPG = feval(mdl,Xnew)
```

```
NewMPG =
    34.1345
    23.4078
     4.7751
```

**random**

The `random` method simulates new random response values, equal to the mean prediction plus a random disturbance with the same variance as the training data.

This example shows how to simulate responses using the `random` method.

- 1 Load the `carbig` data and make a default linear model of the response MPG to the Acceleration, Displacement, Horsepower, and Weight predictors.

```
load carbig
X = [Acceleration,Displacement,Horsepower,Weight];
mdl = fitlm(X,MPG);
```

- 2 Create a three-row array of predictors from the minimal, mean, and maximal values. There are some NaN values, so use functions that ignore NaN values.

```
Xnew = [nanmin(X);nanmean(X);nanmax(X)]; % new data
```

- 3 Generate new predicted model responses including some randomness.

```
rng('default') % for reproducibility
NewMPG = random(mdl,Xnew)
```

```
NewMPG =

    36.4178
    31.1958
    -4.8176
```

- 4 Because a negative value of MPG does not seem sensible, try predicting two more times.

```
NewMPG = random(mdl,Xnew)
```

```
NewMPG =

    37.7959
    24.7615
    -0.7783
```

```
NewMPG = random(md1,Xnew)
```

```
NewMPG =
```

```
32.2931
24.8628
19.9715
```

Clearly, the predictions for the third (maximal) row of `Xnew` are not reliable.

## Share Fitted Models

Suppose you have a linear regression model, such as `mdl` from the following commands:

```
load carbig
ds = dataset(Acceleration,Displacement,Horsepower,Weight,MPG);
mdl = fitlm(ds,'linear','ResponseVar','MPG');
```

To share the model with other people, you can:

- Provide the model display.

```
mdl
```

```
mdl =
```

```
Linear regression model:
```

```
MPG ~ 1 + Acceleration + Displacement + Horsepower + Weight
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	45.251	2.456	18.424	7.0721e-55
Acceleration	-0.023148	0.1256	-0.1843	0.85388
Displacement	-0.0060009	0.0067093	-0.89441	0.37166
Horsepower	-0.043608	0.016573	-2.6312	0.008849
Weight	-0.0052805	0.00081085	-6.5123	2.3025e-10

```
Number of observations: 392, Error degrees of freedom: 387
```

```
Root Mean Squared Error: 4.25
```

```
R-squared: 0.707, Adjusted R-Squared 0.704  
F-statistic vs. constant model: 233, p-value = 9.63e-102
```

- Provide just the model definition and coefficients.

```
mdl.CoefficientNames
```

```
ans =
```

```
      '(Intercept)'      'Acceleration'      'Displacement'      'Horsepower'
```

```
mdl.Coefficients.Estimate
```

```
ans =
```

```
45.2511  
-0.0231  
-0.0060  
-0.0436  
-0.0053
```

```
mdl.Formula
```

```
ans =
```

```
MPG ~ 1 + Acceleration + Displacement + Horsepower + Weight
```

### Linear Regression Workflow

This example shows how to fit a linear regression model. A typical workflow involves the following: import data, fit a regression, test its quality, modify it to improve the quality, and share it.

#### Step 1. Import the data into a dataset array.

hospital.xls is an Excel spreadsheet containing patient names, sex, age, weight, blood pressure, and dates of treatment in an experimental protocol. First read the data into a dataset array.

```
patients = dataset('XLSFile','hospital.xls',...  
                  'ReadObsNames',true);
```



Examine the first row of data.

```
patients(1,:)
```

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

      sys      dia      trial1      trial2      trial3      trial4
YPL-320  124      93      18      -99      -99      -99
```

The sex and smoke fields seem to have two choices each. So change these fields to nominal.

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
patients.sex = nominal(patients.sex);
```

### Step 2. Create a fitted model.

Your goal is to model the systolic pressure as a function of a patient's age, weight, sex, and smoking status. Create a linear formula for 'sys' as a function of 'age', 'wgt', 'sex', and 'smoke'.

```
modelspec = 'sys ~ age + wgt + sex + smoke';
mdl = fitlm(patients,modelspec)
```

```
mdl =
```

Linear regression model:

```
sys ~ 1 + sex + age + wgt + smoke
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	118.28	7.6291	15.504	9.1557e-28
sex_m	0.88162	2.9473	0.29913	0.76549
age	0.08602	0.06731	1.278	0.20438
wgt	-0.016685	0.055714	-0.29947	0.76524
smoke_Yes	9.884	1.0406	9.498	1.9546e-15

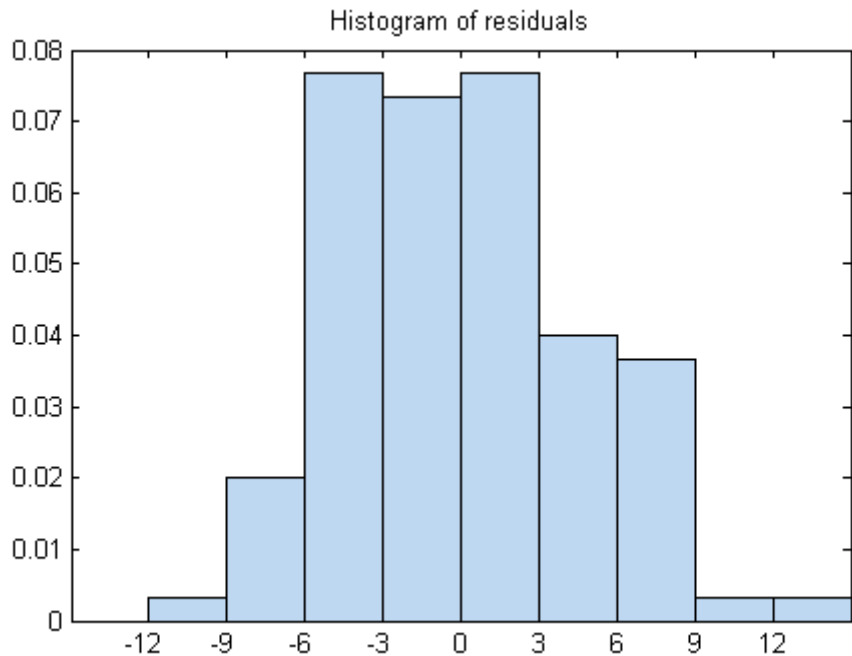
Number of observations: 100, Error degrees of freedom: 95  
Root Mean Squared Error: 4.81  
R-squared: 0.508, Adjusted R-Squared 0.487  
F-statistic vs. constant model: 24.5, p-value = 5.99e-14

The sex, age, and weight predictors have rather high  $p$ -values, indicating that some of these predictors might be unnecessary.

**Step 3. Locate and remove outliers.**

See if there are outliers in the data that should be excluded from the fit. Plot the residuals.

```
plotResiduals(md1)
```



There is one possible outlier, with a value greater than 12. This is probably not truly an outlier. For demonstration, here is how to find and remove it.

Find the outlier.

```
outlier = mdl.Residuals.Raw > 12;
find(outlier)
```

```
ans =
     84
```

Remove the outlier.

```
mdl = fitlm(patients,modelspec,...
    'Exclude',84);
```

```
mdl.ObservationInfo(84,:)
```

```
ans =
           WXM-486      Weights      Excluded      Missing      Subset
           1           1           true           false           false
```

Observation 84 is no longer in the model.

#### Step 4. Simplify the model.

Try to obtain a simpler model, one with fewer predictors but the same predictive accuracy. `step` looks for a better model by adding or removing one term at a time. Allow `step` take up to 10 steps.

```
mdl1 = step(mdl,'NSteps',10)
```

1. Removing wgt, FStat = 4.6001e-05, pValue = 0.9946
2. Removing sex, FStat = 0.063241, pValue = 0.80199

```
mdl1 =
```

```
Linear regression model:
    sys ~ 1 + age + smoke
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	115.11	2.5364	45.383	1.1407e-66
age	0.10782	0.064844	1.6628	0.09962

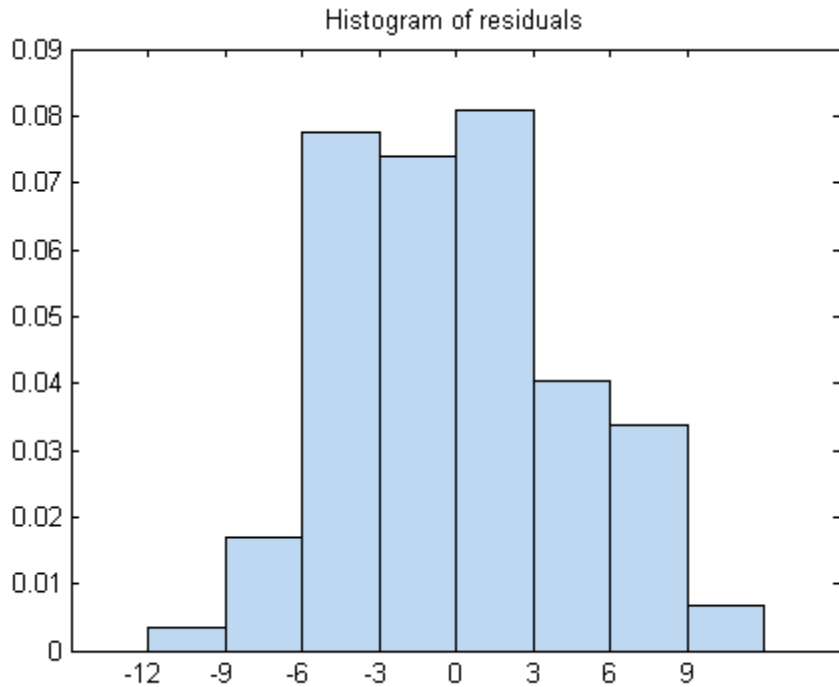
```
smoke_Yes      10.054      0.97696      10.291      3.5276e-17
```

```
Number of observations: 99, Error degrees of freedom: 96  
Root Mean Squared Error: 4.61  
R-squared: 0.536, Adjusted R-Squared 0.526  
F-statistic vs. constant model: 55.4, p-value = 1.02e-16
```

step took two steps. This means it could not improve the model further by adding or subtracting a single term.

Plot the effectiveness of the simpler model on the training data.

```
plotResiduals(md11)
```



The residuals look about as small as those of the original model.

**Step 5. Predict responses to new data.**

Suppose you have four new people, aged 25, 30, 40, and 65, and the first and third smoke. Predict their systolic pressure using `mdl1`.

```
ages = [25;30;40;65];
smoker = {'Yes';'No';'Yes';'No'};
systolicnew = feval(mdl1,ages,smoker)
```

```
systolicnew =
```

```
127.8561
118.3412
129.4734
122.1149
```

To make predictions, you need only the variables that `mdl1` uses.

**Step 6. Share the model.**

You might want others to be able to use your model for prediction. Access the terms in the linear model.

```
coefnames = mdl1.CoefficientNames
```

```
coefnames =
```

```
 '(Intercept)'      'age'      'smoke_Yes'
```

View the model formula.

```
mdl1.Formula
```

```
ans =
```

```
sys ~ 1 + age + smoke
```

Access the coefficients of the terms.

```
coefvals = mdl1.Coefficients(:,1); % dataset array
coefvals = double(coefvals)
```

```
coefvals =
```

```
115.1066
```

```
0.1078
```

```
10.0540
```

The model is  $\text{sys} = 115.1066 + 0.1078 \cdot \text{age} + 10.0540 \cdot \text{smoke}$ , where  $\text{smoke}$  is 1 for a smoker, and 0 otherwise.

## Regression Using Dataset Arrays

This example shows how to perform linear and stepwise regression analyses using dataset arrays.

### Load sample data.

```
load imports-85
```

### Store predictor and response variables in dataset array.

```
ds = dataset(X(:,7),X(:,8),X(:,9),X(:,15),'Varnames',...
{'curb_weight','engine_size','bore','price'});
```

### Fit linear regression model.

Fit a linear regression model that explains the price of a car in terms of its curb weight, engine size, and bore.

```
fitlm(ds,'price~curb_weight+engine_size+bore')
```

```
ans =
```

```
Linear regression model:
```

```
price ~ 1 + curb_weight + engine_size + bore
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	64.095	3.703	17.309	2.0481e-41
curb_weight	-0.0086681	0.0011025	-7.8623	2.42e-13
engine_size	-0.015806	0.013255	-1.1925	0.23452
bore	-2.6998	1.3489	-2.0015	0.046711

```
Number of observations: 201, Error degrees of freedom: 197
```

```
Root Mean Squared Error: 3.95
```

```
R-squared: 0.674, Adjusted R-Squared 0.669
```

```
F-statistic vs. constant model: 136, p-value = 1.14e-47
```

The command `fitlm(ds)` also returns the same result because `fitlm`, by default, assumes the predictor variable is in the last column of the dataset array `ds`.

### Recreate dataset array and repeat analysis.

This time, put the response variable in the first column of the dataset array.

```
ds = dataset(X(:,15),X(:,7),X(:,8),X(:,9), 'Varnames', ...
{'price', 'curb_weight', 'engine_size', 'bore'});
```

When the response variable is in the first column of `ds`, define its location. For example, `fitlm`, by default, assumes that `bore` is the response variable. You can define the response variable in the model using either:

```
fitlm(ds, 'ResponseVar', 'price');
```

or

```
fitlm(ds, 'ResponseVar', logical([1 0 0 0]));
```

### Perform stepwise regression.

```
stepwiselm(ds, 'quadratic', 'lower', 'price~1', ...
'ResponseVar', 'price')
```

1. Removing `bore^2`, `FStat` = 0.01282, `pValue` = 0.90997
2. Removing `engine_size^2`, `FStat` = 0.078043, `pValue` = 0.78027
3. Removing `curb_weight:bore`, `FStat` = 0.70558, `pValue` = 0.40195

```
ans =
```

Linear regression model:

```
price ~ 1 + curb_weight*engine_size + engine_size*bore + curb_weight^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pVal
(Intercept)	131.13	14.273	9.1873	6.23
curb_weight	-0.043315	0.0085114	-5.0891	8.46
engine_size	-0.17102	0.13844	-1.2354	0
bore	-12.244	4.999	-2.4493	0.



curb_weight:engine_size	-6.3411e-05	2.6577e-05	-2.386	0.
engine_size:bore	0.092554	0.037263	2.4838	0.
curb_weight^2	8.0836e-06	1.9983e-06	4.0451	7.54

Number of observations: 201, Error degrees of freedom: 194

Root Mean Squared Error: 3.59

R-squared: 0.735, Adjusted R-Squared 0.726

F-statistic vs. constant model: 89.5, p-value = 3.58e-53

The initial model is a quadratic formula, and the lowest model considered is the constant. Here, `stepwiselm` performs a backward elimination technique to determine the terms in the model. The final model is `price ~ 1 + curb_weight*engine_size + engine_size*bore + curb_weight^2`, which corresponds to

$$P = \beta_0 + \beta_C C + \beta_E E + \beta_B B + \beta_{CE} CE + \beta_{EB} EB + \beta_{C^2} C^2 + \varepsilon$$

where  $P$  is price,  $C$  is curb weight,  $E$  is engine size,  $B$  is bore,  $\beta_i$  is the coefficient for the corresponding term in the model, and  $\varepsilon$  is the error term. The final model includes all three main effects, the interaction effects for curb weight and engine size and engine size and bore, and the second-order term for curb weight.

## See Also

`LinearModel` | `fitlm` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

## Linear Regression with Interaction Effects

This example shows how to construct and analyze a linear regression model with interaction effects and interpret the results.

### Load sample data.

```
load hospital
```

To retain only the first column of blood pressure, store data in a new dataset array.

```
ds = dataset(hospital.Sex,hospital.Age,hospital.Weight,hospital.Smoker,...
hospital.BloodPressure(:,1),'Varnames',{'Sex','Age','Weight','Smoker',...
'BloodPressure'});
```

### Perform stepwise linear regression.

For the initial model, use the full model with all terms and their pairwise interactions.

```
mdl = stepwiselm(ds,'interactions')
```

1. Removing Sex:Smoker, FStat = 0.050738, pValue = 0.8223
2. Removing Weight:Smoker, FStat = 0.07758, pValue = 0.78124
3. Removing Age:Weight, FStat = 1.9717, pValue = 0.16367
4. Removing Sex:Age, FStat = 0.32389, pValue = 0.57067
5. Removing Age:Smoker, FStat = 2.4939, pValue = 0.11768

```
mdl =
```

Linear regression model:

```
BloodPressure ~ 1 + Age + Smoker + Sex*Weight
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	133.17	10.337	12.883	1.76e-22
Sex_Male	-35.269	17.524	-2.0126	0.047015
Age	0.11584	0.067664	1.712	0.090198
Weight	-0.1393	0.080211	-1.7367	0.085722

Smoker_1	9.8307	1.0229	9.6102	1.2391e-15
Sex_Male:Weight	0.2341	0.11192	2.0917	0.039162

Number of observations: 100, Error degrees of freedom: 94  
 Root Mean Squared Error: 4.72  
 R-squared: 0.53, Adjusted R-Squared 0.505  
 F-statistic vs. constant model: 21.2, p-value = 4e-14

The final model in formula form is **BloodPressure ~ 1 + Age + Smoker + Sex\*Weight**. This model includes all four main effects (Age, Smoker, Sex, Weight) and the two-way interaction between Sex and Weight. This model corresponds to

$$BP = \beta_0 + \beta_A X_A + \beta_{Sm} I_{Sm} + \beta_S I_S + \beta_W X_W + \beta_{SW} X_W I_S + \varepsilon,$$

where

- $BP$  is the blood pressure
- $\beta_i$ s are the coefficients
- $I_{Sm}$  is the indicator variable for smoking;  $I_{Sm} = 1$  indicates a smoking patient whereas  $I_{Sm} = 0$  indicates a nonsmoking patient
- $I_S$  is the indicator variable for sex;  $I_S = 1$  indicates a male patient whereas  $I_S = 0$  indicates a female patient
- $X_A$  is the Age variable
- $X_W$  is the Weight variable
- $\varepsilon$  is the error term

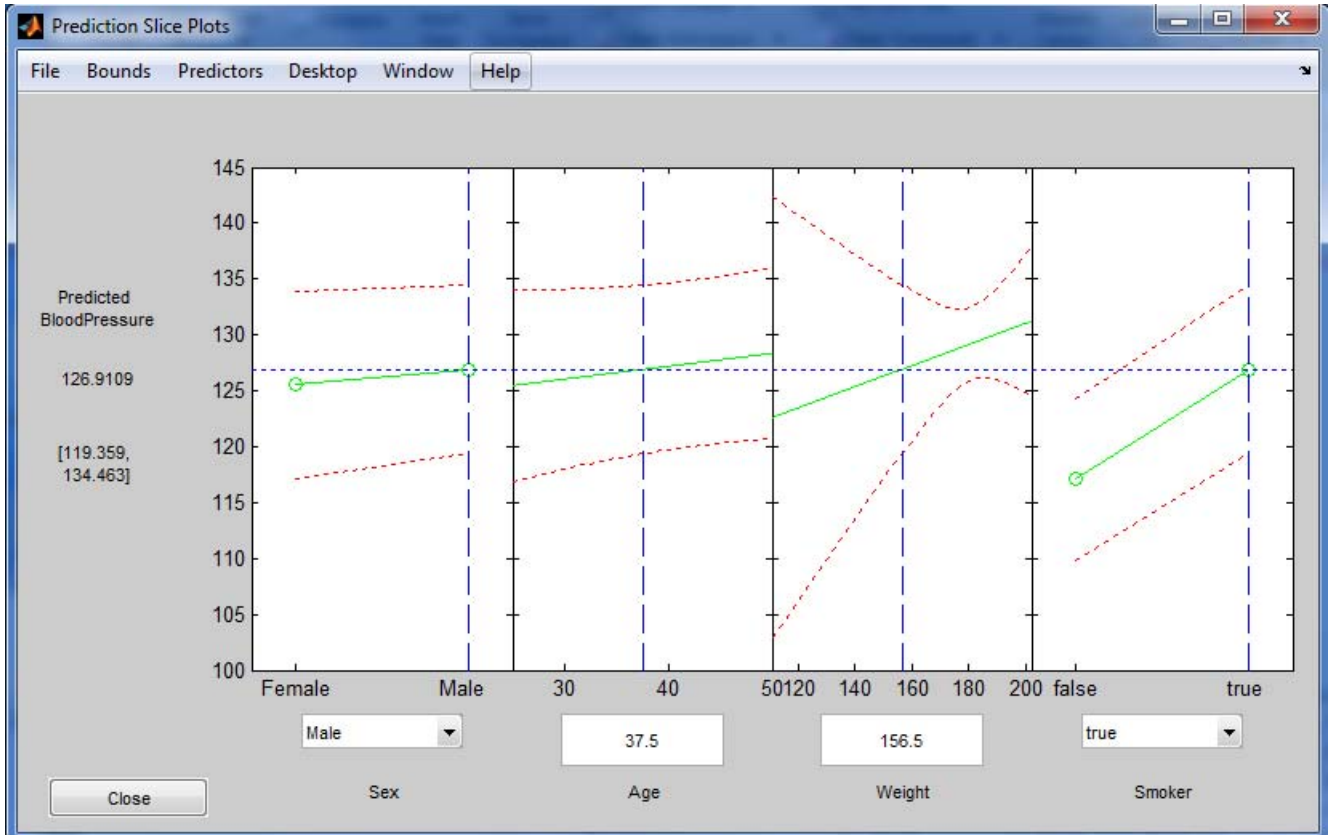
The following table shows the fitted linear model for each gender and smoking combination.

$I_{Sm}$	$I_S$	Linear Model
1 (Male)	1 (Smoker)	$BP = (\beta_0 + \beta_{Sm} + \beta_S) + \beta_A X_A + (\beta_W + \beta_{SW}) X_W$ $\hat{BP} = 107.5617 + 0.11584 X_A + 0.11826 X_W$
1 (Male)	0 (Nonsmoker)	$BP = (\beta_0 + \beta_{Sm}) + \beta_A X_A + \beta_W X_W$
0 (Female)	1 (Smoker)	$BP = (\beta_0 + \beta_S) + \beta_A X_A + (\beta_W + \beta_{SW}) X_W$ $\hat{BP} = 97.901 + 0.11584 X_A + 0.11826 X_W$
0 (Female)	0 (Nonsmoker)	$BP = \beta_0 + \beta_A X_A + \beta_W X_W$ $\hat{BP} = 133.17 + 0.11584 X_A - 0.1393 X_W$

As seen from these models,  $\beta_{Sm}$  and  $\beta_S$  show how much the intercept of the response function changes when the indicator variable takes the value 1 compared to when it takes the value 0.  $\beta_{SW}$ , however, shows the effect of the Weight variable on the response variable when the indicator variable for sex takes the value 1 compared to when it takes the value 0. You can explore the main and interaction effects in the final model using the methods of the `LinearModel` class as follows.

**Plot prediction slice plots.**

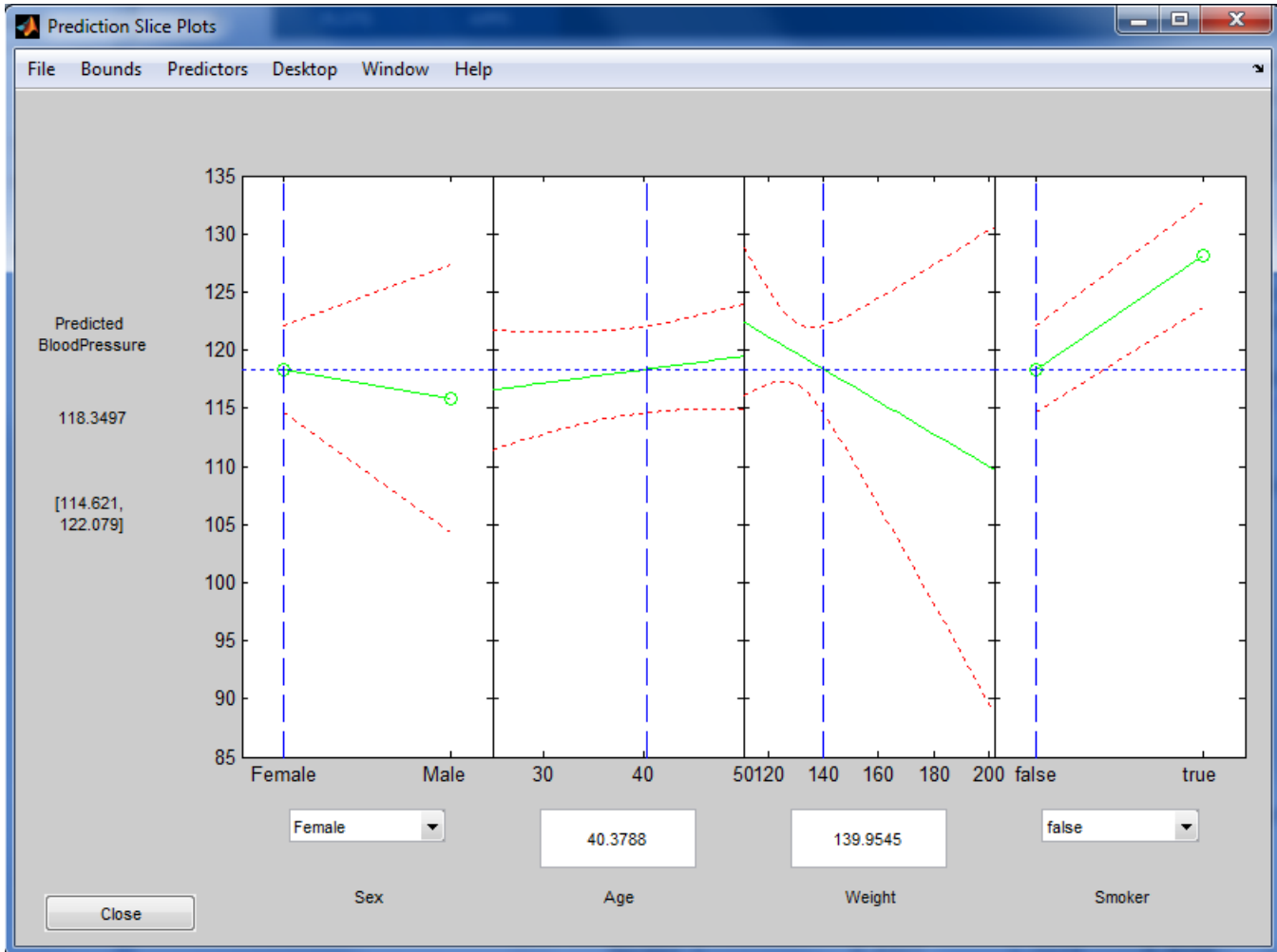
```
figure()
plotSlice mdl
```



This plot shows the main effects for all predictor variables. The green line in each panel shows the change in the response variable as a function of the predictor variable when all other predictor variables are held constant. For example, for a smoking male patient aged 37.5, the expected blood pressure increases as the weight of the patient increases, given all else the same.

The dashed red curves in each panel show the 95% confidence bounds for the predicted response values.

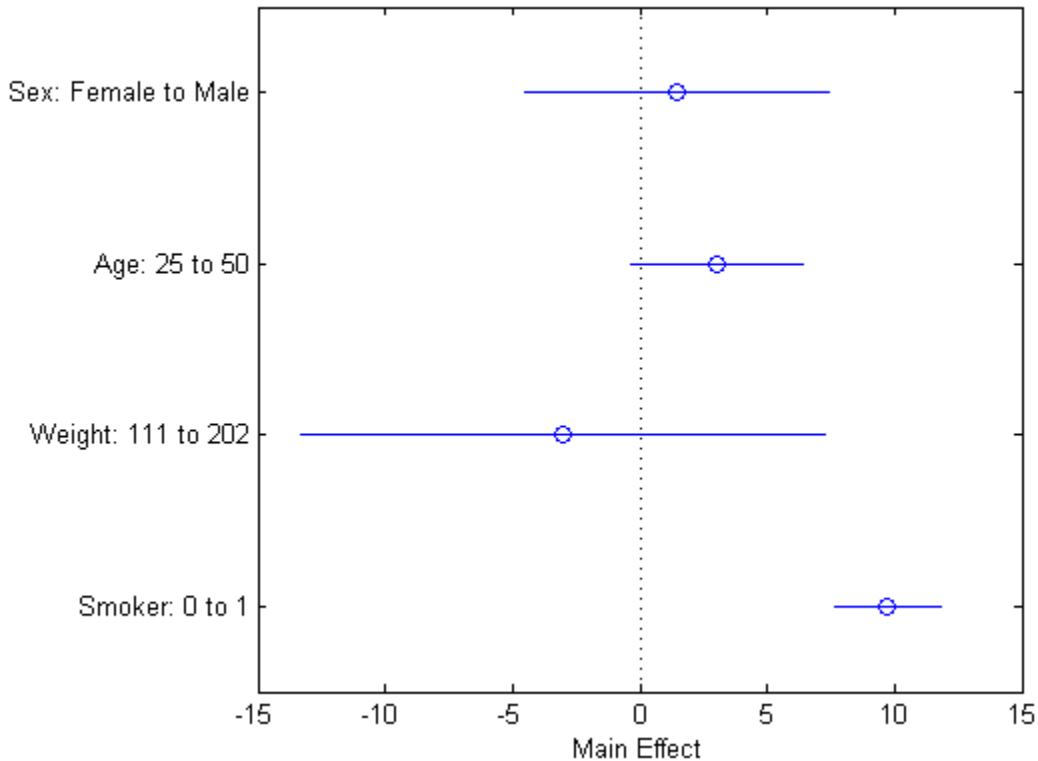
The horizontal dashed blue line in each panel shows the predicted response for the specific value of the predictor variable corresponding to the vertical dashed blue line. You can drag these lines to get the predicted response values at other predictor values, as shown next.



For example, the predicted value of the response variable is 118.3497 when a patient is female, nonsmoking, age 40.3788, and weighs 139.9545 pounds. The values in the square brackets, [114.621, 122.079], show the lower and upper limits of a 95% confidence interval for the estimated response. Note that, for a nonsmoking female patient, the expected blood pressure decreases as the weight increases, given all else is held constant.

**Plot main effects.**

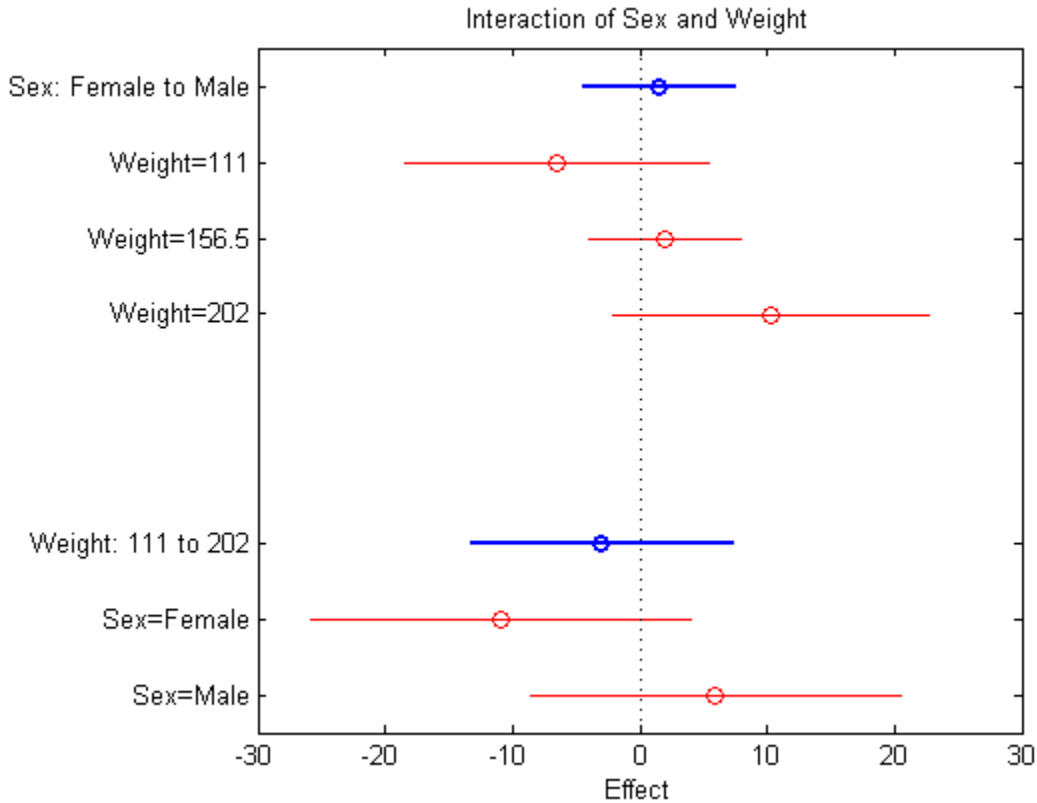
```
figure()  
plotEffects mdl
```



This plot displays the main effects. The circles show the magnitude of the effect and the blue lines show the upper and lower confidence limits for the main effect. For example, being a smoker increases the expected blood pressure by 10 units, compared to being a nonsmoker, given all else is held constant. Expected blood pressure increases about two units for males compared to females, again, given other predictors held constant. An increase in age from 25 to 50 causes an expected increase of 4 units, whereas a change in weight from 111 to 202 causes about a 4-unit decrease in the expected blood pressure, given all else held constant.

**Plot interaction effects.**

```
figure()
plotInteraction mdl, 'Sex', 'Weight'
```



This plot displays the impact of a change in one factor given the other factor is fixed at a value.

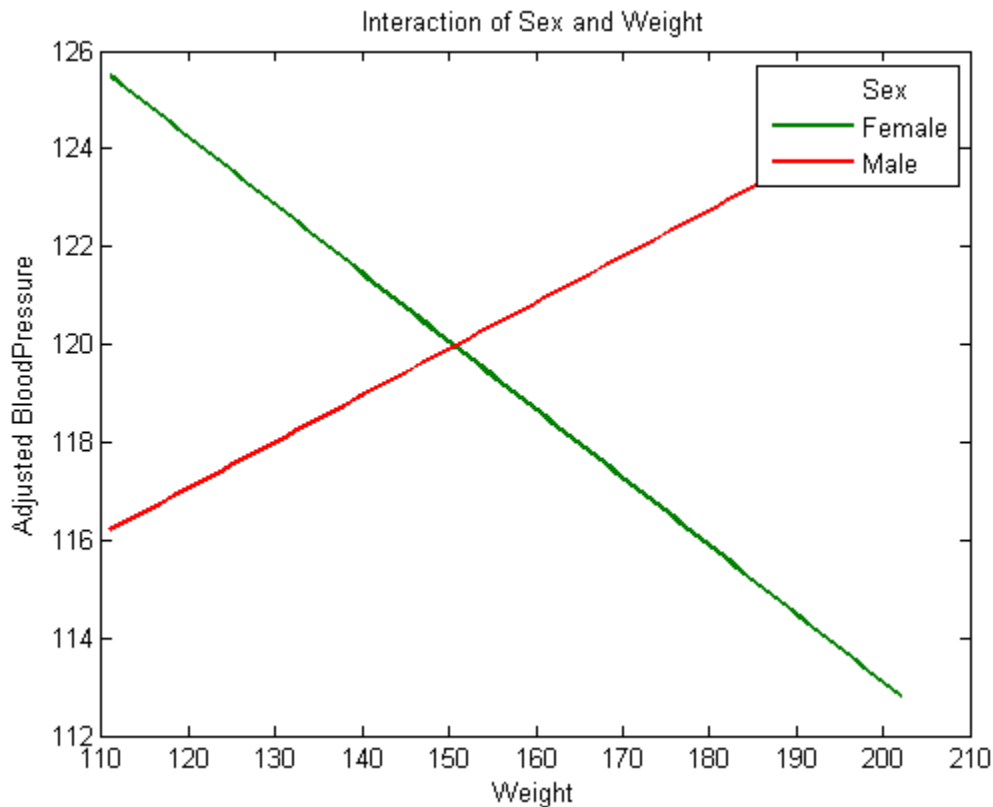
Be cautious while interpreting the interaction effects. When there is not enough data on all factor combinations or the data is highly correlated, it might be difficult to determine the interaction effect of changing one factor while keeping the other fixed. In such cases, the estimated interaction effect is an extrapolation from the data.



The blue circles show the main effect of a specific term, as in the main effects plot. The red circles show the impact of a change in one term for fixed values of the other term. For example, in the bottom half of this plot, the red circles show the impact of a weight change in female and male patients, separately. You can see that an increase in a female's weight from 111 to 202 pounds causes about a 14-unit decrease in the expected blood pressure, while an increase of the same amount in the weight of a male patient causes about a 5-unit increase in the expected blood pressure, again given other predictors are held constant.

**Plot prediction effects.**

```
figure()  
plotInteraction(md1, 'Sex', 'Weight', 'predictions')
```



This plot shows the effect of changing one variable as the other predictor variable is held constant. In this example, the last figure shows the response variable, blood pressure, as a function of weight, when the variable sex is fixed at males and females. The lines for males and females are crossing which indicates a strong interaction between weight and sex. You can see that the expected blood pressure increases as the weight of a male patient increases, but decreases as the weight of a female patient increases.

### See Also

`LinearModel` | `LinearModel.fit` | `LinearModel.stepwise` | `plotSlice` | `plotEffects` | `plotInteraction` |

## **Related Examples**

- “Plots to Understand Predictor Effects” on page 9-30

## Interpret Linear Regression Results

This example shows how to display and interpret linear regression output statistics.

### Load sample data and define predictor variables.

```
load carsmall
X = [Weight,Horsepower,Acceleration];
```

### Fit linear regression model.

```
lm = fitlm(X,MPG,'linear')
```

```
lm =
```

```
Linear regression model:
```

```
y ~ 1 + x1 + x2 + x3
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	47.977	3.8785	12.37	4.8957e-21
x1	-0.0065416	0.0011274	-5.8023	9.8742e-08
x2	-0.042943	0.024313	-1.7663	0.08078
x3	-0.011583	0.19333	-0.059913	0.95236

```
Number of observations: 93, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 4.09
```

```
R-squared: 0.752, Adjusted R-Squared 0.744
```

```
F-statistic vs. constant model: 90, p-value = 7.38e-27
```

This linear regression outputs display shows the following.

$y \sim 1 + x_1 + x_2 + x_3$	Linear regression model in the formula form using Wilkinson notation. Here it corresponds to:  $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \varepsilon.$
First column (under Estimated Coefficients)	Terms included in the model.
Estimate	Coefficient estimates for each corresponding term in the model. For example, the estimate for the constant term (intercept) is 47.977.
SE	Standard error of the coefficients.
tStat	<i>t</i> -statistic for each coefficient to test the null hypothesis that the corresponding coefficient is zero against the alternative that it is different from zero, given the other predictors in the model. Note that $tStat = Estimate/SE$ . For example, the <i>t</i> -statistic for the intercept is $47.977/3.8785 = 12.37$ .
pValue	<i>p</i> -value for the F statistic of the hypotheses test that the corresponding coefficient is equal to zero or not. For example, the <i>p</i> -value of the F-statistic for $x_2$ is greater than 0.05, so this term is not significant at the 5% significance level given the other terms in the model.
Number of observations	Number of rows without any NaN values. For example, Number of observations is 93 because the MPG data vector has 6 NaN values and one of the data vectors, Horsepower, has one NaN value for a different observation.
Error degrees of freedom	$n - p$ , where $n$ is the number of observations, and $p$ is the number of coefficients in the model, including the intercept. For example, the model has four predictors, so the Error degrees of freedom is $93 - 4 = 89$ .
Root mean squared error	Square root of the mean squared error, which estimates the standard deviation of the error distribution.

R-squared and Adjusted R-squared	Coefficient of determination and adjusted coefficient of determination, respectively. For example, the R-squared value suggests that the model explains approximately 75% of the variability in the response variable MPG.
F-statistic vs. constant model	Test statistic for the F-test on the regression model. It tests for a significant linear regression relationship between the response variable and the predictor variables.
p-value	<i>p</i> -value for the F-test on the model. For example, the model is significant with a <i>p</i> -value of 7.3816e-27.

You can request this display by using `disp`. For example, if you name your model `lm`, then you can display the outputs using `disp(lm)`.

**Perform analysis of variance (ANOVA) for the model.**

```
anova(lm, 'summary')
```

```
ans =
```

	SumSq	DF	MeanSq	F	pValue
Total	6004.8	92	65.269		
Model	4516	3	1505.3	89.987	7.3816e-27
Residual	1488.8	89	16.728		

This ANOVA display shows the following.

SumSq	Sum of squares for the regression model, Model, the error term, Residual, and the total, Total.
DF	Degrees of freedom for each term. Degrees of freedom is $n - 1$ for the total, $p - 1$ for the model, and $n - p$ for the error term, where $n$ is the number of observations, and $p$ is the number of coefficients in the model, including the intercept. For example, MPG data vector has six NaN values and one of the data vectors, Horsepower, has one NaN value for a different observation, so the total degrees of freedom is $93 - 1 = 92$ . There are four coefficients in the model, so the model DF is $4 - 1 = 3$ , and the DF for error term is $93 - 4 = 89$ .
MeanSq	Mean squared error for each term. Note that $\text{MeanSq} = \text{SumSq}/\text{DF}$ . For example, the mean squared error for the error term is $1488.8/89 = 16.728$ . The square root of this value is the root mean squared error in the linear regression display, or 4.09.
F	F-statistic value, which is the same as F-statistic vs. constant model in the linear regression display. In this example, it is 89.987, and in the linear regression display this F-statistic value is rounded up to 90.
pValue	p-value for the F-test on the model. In this example, it is 7.3816e-27.

If there are higher-order terms in the regression model, anova partitions the model SumSq into the part explained by the higher-order terms and the rest. The corresponding F-statistics are for testing the significance of the linear terms and higher-order terms as separate groups.

If the data includes replicates, or multiple measurements at the same predictor values, then the anova partitions the error SumSq into the part for the replicates and the rest. The corresponding F-statistic is for testing the lack-of-fit by comparing the model residuals with the model-free variance estimate computed on the replicates.

See the anova method for details.

**Decompose ANOVA table for model terms.**

```
anova(lm)
```

```
ans =
```

	SumSq	DF	MeanSq	F	pValue
x1	563.18	1	563.18	33.667	9.8742e-08
x2	52.187	1	52.187	3.1197	0.08078
x3	0.060046	1	0.060046	0.0035895	0.95236
Error	1488.8	89	16.728		

This anova display shows the following:

First column	Terms included in the model.
SumSq	Sum of squared error for each term except for the constant.
DF	Degrees of freedom. In this example, DF is 1 for each term in the model and $n - p$ for the error term, where $n$ is the number of observations, and $p$ is the number of coefficients in the model, including the intercept. For example, the DF for the error term in this model is $93 - 4 = 89$ . If any of the variables in the model is a categorical variable, the DF for that variable is the number of indicator variables created for its categories (number of categories - 1).
MeanSq	Mean squared error for each term. Note that $\text{MeanSq} = \text{SumSq}/\text{DF}$ . For example, the mean squared error for the error term is $1488.8/89 = 16.728$ .
F	F-values for each coefficient. The F-value is the ratio of the mean squared of each term and mean squared error, that is, $F = \text{MeanSq}(x_i) / \text{MeanSq}(\text{Error})$ . Each F-statistic has an F distribution, with the numerator degrees of freedom, DF value for the corresponding term, and the denominator degrees of freedom, $n - p$ . $n$ is the number of observations, and $p$ is the number of coefficients in the model. In this example, each F-statistic has an $F_{(1, 89)}$ distribution.
pValue	$p$ -value for each hypothesis test on the coefficient of the corresponding term in the linear model. For example, the $p$ -value for the F-statistic coefficient of $x_2$ is 0.08078, and is not



significant at the 5% significance level given the other terms in the model.

### Display coefficient confidence intervals.

```
coefCI(lm)
```

```
ans =
```

```

40.2702  55.6833
-0.0088 -0.0043
-0.0913  0.0054
-0.3957  0.3726

```

The values in each row are the lower and upper confidence limits, respectively, for the default 95% confidence intervals for the coefficients. For example, the first row shows the lower and upper limits, 40.2702 and 55.6833, for the intercept,  $\beta_0$ . Likewise, the second row shows the limits for  $\beta_1$  and so on. Confidence intervals provide a measure of precision for linear regression coefficient estimates. A  $100(1-\alpha)\%$  confidence interval gives the range the corresponding regression coefficient will be in with  $100(1-\alpha)\%$  confidence.

You can also change the confidence level. Find the 99% confidence intervals for the coefficients.

```
coefCI(lm,0.01)
```

```
ans =
```

```

37.7677  58.1858
-0.0095 -0.0036
-0.1069  0.0211
-0.5205  0.4973

```

### Perform hypothesis test on coefficients.

Test the null hypothesis that all predictor variable coefficients are equal to zero versus the alternate hypothesis that at least one of them is different from zero.

```
[p,F,d] = coefTest(lm)
```

```
p =  
7.3816e-27
```

```
F =  
89.9874
```

```
d =  
3
```

Here, `coefTest` performs an F-test for the hypothesis that all regression coefficients (except for the intercept) are zero versus at least one differs from zero, which essentially is the hypothesis on the model. It returns  $p$ , the  $p$ -value,  $F$ , the F-statistic, and  $d$ , the numerator degrees of freedom. The F-statistic and  $p$ -value are the same as the ones in the linear regression display and ANOVA for the model. The degrees of freedom is  $4 - 1 = 3$  because there are four predictors (including the intercept) in the model.

Now, perform a hypothesis test on the coefficients of the first and second predictor variables.

```
H = [0 1 0 0; 0 0 1 0];  
[p,F,d] = coefTest(lm,H)
```

```
p =  
5.1702e-23
```

```
F =  
96.4873
```

```
d =
```

2

The numerator degrees of freedom is the number of coefficients tested, which is 2 in this example. The results indicate that at least one of  $\beta_2$  and  $\beta_3$  differs from zero.

**See Also**

`LinearModel` | `fitlm` | `stepwiselmanova` |

**Related Examples**

- “Examine Quality and Adjust the Fitted Model” on page 9-21

**Concepts**

- “Coefficient Standard Errors and Confidence Intervals” on page 9-76
- “Coefficient of Determination (R-Squared)” on page 9-80
- “F-statistic and t-statistic” on page 9-95
- “Summary of Output and Diagnostic Statistics” on page 9-117

## Cook's Distance

### Purpose

Cook's distance is useful for identifying outliers in the  $X$  values (observations for predictor variables). It also shows the influence of each observation on the fitted response values. An observation with Cook's distance larger than three times the mean Cook's distance might be an outlier.

### Definition

Cook's distance is the scaled change in fitted values. Each element in `CooksDistance` is the normalized change in the vector of coefficients due to the deletion of an observation. The Cook's distance,  $D_i$ , of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{pMSE},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $MSE$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{pMSE} \left( \frac{h_{ii}}{(1-h_{ii})^2} \right),$$

where  $r_i$  is the  $i$ th residual, and  $h_{ii}$  is the  $i$ th leverage value.

`CooksDistance` is an  $n$ -by-1 column vector in the `Diagnostics` dataset array of the `LinearModel` object.

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the Cook's distance values by indexing into the property using dot notation,

```
mdl.Diagnostics.CooksDistance
```

- Plot the Cook's distance values using

```
plotDiagnostics(mdl, 'cookd')
```

For details, see the `plotDiagnostics` method of the `LinearModel` class.

## Example

This example shows how to use Cook's Distance to determine the outliers in the data.

Load the sample data and define the independent and response variables.

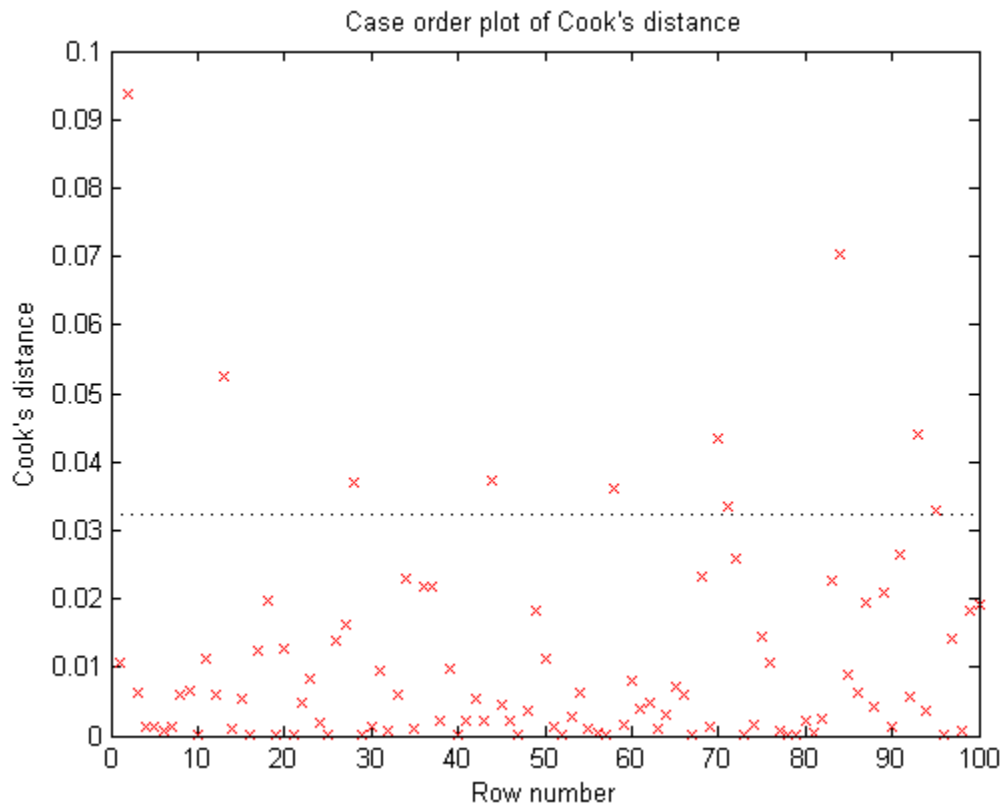
```
load hospital
X = double(hospital(:,2:5));
y = hospital.BloodPressure(:,1);
```

Fit the linear regression model.

```
mdl = fitlm(X,y);
```

Plot the Cook's distance values.

```
plotDiagnostics(mdl, 'cookd')
```



The dashed line in the figure corresponds to the recommended threshold value,  $3 \cdot \text{mean}(\text{mdl.Diagnostics.CooksDistance})$ . The plot has some observations with Cook's distance values greater than the threshold value, which for this example is  $3 \cdot (0.0108) = 0.0324$ . In particular, there are two Cook's distance values that are relatively higher than the others, which exceed the threshold value. You might want to find and omit these from your data and rebuild your model.

Find the observations with Cook's distance values that exceed the threshold value.

```
find((mdl.Diagnostics.CooksDistance)>3*mean(mdl.Diagnostics.CooksDistance))
```

```
ans =  
    2  
   13  
   28  
   44  
   58  
   70  
   71  
   84  
   93  
   95
```

Find the observations with Cook's distance values that are relatively larger than the other observations with Cook's distances exceeding the threshold value.

```
find((mdl.Diagnostics.CooksDistance)>5*mean(mdl.Diagnostics.CooksDistance))
```

```
ans =  
    2  
   84
```

## See Also

[LinearModel](#) | [fitlm](#) | [stepwiselplotDiagnostics](#) |

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

## Coefficient Standard Errors and Confidence Intervals

### In this section...

“Coefficient Covariance and Standard Errors” on page 9-76

“Compute Coefficient Covariance and Standard Errors” on page 9-76

“Coefficient Confidence Intervals” on page 9-77

“Compute Coefficient Confidence Intervals” on page 9-78

## Coefficient Covariance and Standard Errors

### Purpose

Estimated coefficient variances and covariances capture the precision of regression coefficient estimates. The coefficient variances and their square root, the standard errors, are useful in testing hypotheses for coefficients.

### Definition

The estimated covariance matrix is

$$\Sigma = MSE(XX)^{-1},$$

where  $MSE$  is the mean squared error, and  $X$  is the matrix of observations on the predictor variables. `CoefficientCovariance`, a property of the fitted model, is a  $p$ -by- $p$  covariance matrix of regression coefficient estimates.  $p$  is the number of coefficients in the regression model. The diagonal elements are the variances of the individual coefficients.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can display the coefficient covariances using

```
mdl.CoefficientCovariance
```

## Compute Coefficient Covariance and Standard Errors

This example shows how to compute the covariance matrix and standard errors of the coefficients.



Load the sample data and define the predictor and response variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Display the coefficient covariance matrix.

```
CM = mdl.CoefficientCovariance
```

```
CM =

    27.5113    11.0027   -0.1542   -0.2444    0.2702
    11.0027     8.6864    0.0021   -0.1547   -0.0838
   -0.1542     0.0021    0.0045   -0.0001   -0.0029
   -0.2444   -0.1547   -0.0001    0.0031   -0.0026
    0.2702   -0.0838   -0.0029   -0.0026    1.0829
```

Compute the coefficient standard errors.

```
SE = diag(sqrt(CM))
```

```
SE =

    5.2451
    2.9473
    0.0673
    0.0557
    1.0406
```

## Coefficient Confidence Intervals

### Purpose

The coefficient confidence intervals provide a measure of precision for linear regression coefficient estimates. A  $100(1-\alpha)\%$  confidence interval gives the range that the corresponding regression coefficient will be in with  $100(1-\alpha)\%$  confidence.

**Definition**

The  $100*(1-\alpha)\%$  confidence intervals for linear regression coefficients are

$$b_i \pm t_{(1-\alpha/2, n-p)} SE(b_i),$$

where  $b_i$  is the coefficient estimate,  $SE(b_i)$  is the standard error of the coefficient estimate, and  $t_{(1-\alpha/2, n-p)}$  is the  $100(1-\alpha/2)$  percentile of  $t$ -distribution with  $n - p$  degrees of freedom.  $n$  is the number of observations and  $p$  is the number of regression coefficients.

**How To**

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can obtain the default 95% confidence intervals for coefficients using

```
coefCI(mdl)
```

You can also change the confidence level using

```
coefCI(mdl, alpha)
```

For details, see the `coefCI` and `coefTest` methods of `LinearModel` class.

**Compute Coefficient Confidence Intervals**

This example shows how to compute coefficient confidence intervals.

Load the sample data and fit a linear regression model.

```
load hald  
mdl = fitlm(ingredients, heat);
```

Display the 95% coefficient confidence intervals.

```
coefCI(mdl)  
  
ans =  
  
-99.1786  223.9893  
-0.1663   3.2685  
-1.1589   2.1792
```

```
-1.6385    1.8423
-1.7791    1.4910
```

The values in each row are the lower and upper confidence limits, respectively, for the default 95% confidence intervals for the coefficients. For example, the first row shows the lower and upper limits,  $-99.1786$  and  $223.9893$ , for the intercept,  $\beta_0$ . Likewise, the second row shows the limits for  $\beta_1$  and so on.

Display the 90% confidence intervals for the coefficients ( $\alpha = 0.1$ ).

```
coefCI mdl, 0.1)
```

```
ans =
```

```
-67.8949  192.7057
  0.1662   2.9360
-0.8358   1.8561
-1.3015   1.5053
-1.4626   1.1745
```

The confidence interval limits become narrower as the confidence level decreases.

## See Also

[LinearModel](#) | [fitlm](#) | [stepwiselmplotDiagnostics](#) | [anova](#) | [coefCI](#) | [coefTest](#) |

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

## Coefficient of Determination (R-Squared)

### Purpose

Coefficient of determination (R-squared) indicates the proportionate amount of variation in the response variable  $y$  explained by the independent variables  $X$  in the linear regression model. The larger the R-squared is, the more variability is explained by the linear regression model.

### Definition

R-squared is the proportion of the total sum of squares explained by the model. Rsquared, a property of the fitted model, is a structure with two fields:

- Ordinary — Ordinary (unadjusted) R-squared

$$R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}.$$

- Adjusted — R-squared adjusted for the number of coefficients

$$R_{adj}^2 = 1 - \left( \frac{n-1}{n-p} \right) \frac{SSE}{SST}.$$

$SSE$  is the sum of squared error,  $SSR$  is the sum of squared regression,  $SST$  is the sum of squared total,  $n$  is the number of observations, and  $p$  is the number of regression coefficients (including the intercept). Because R-squared increases with added predictor variables in the regression model, the adjusted R-squared adjusts for the number of predictor variables in the model. This makes it more useful for comparing models with a different number of predictors.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can obtain either R-squared value as a scalar by indexing into the property using dot notation, for example,

```
mdl.Rsquared.Ordinary  
mdl.Rsquared.Adjusted
```

You can also obtain the SSE, SSR, and SST using the properties with the same name.

```
mdl.SSE
mdl.SSR
mdl.SST
```

## Example

This example shows how to display Rsquared (coefficient of determination) and adjusted R-squared.

Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y)
```

```
mdl =
```

```
Linear regression model:
y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	117.4	5.2451	22.383	1.1667e-39
x1	0.88162	2.9473	0.29913	0.76549
x2	0.08602	0.06731	1.278	0.20438
x3	-0.016685	0.055714	-0.29947	0.76524
x4	9.884	1.0406	9.498	1.9546e-15

Number of observations: 100, Error degrees of freedom: 95

Root Mean Squared Error: 4.81

R-squared: 0.508, Adjusted R-Squared 0.487

F-statistic vs. constant model: 24.5, p-value = 5.99e-14

The R-squared and adjusted R-squared values are 0.508 and 0.487, respectively. Model explains about 50% of the variability in the response variable.

Access the R-squared and adjusted R-squared values using the property of the fitted `LinearModel` object.

```
mdl.Rsquared.Ordinary
```

```
ans =
```

```
0.5078
```

```
mdl.Rsquared.Adjusted
```

```
ans =
```

```
0.4871
```

The adjusted R-squared value is smaller than the ordinary R-squared value.

### See Also

`LinearModel` | `fitlm` | `stepwiselmanova` |

### Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

## Delete-1 Statistics

### In this section...

“Delete-1 Change in Covariance (covratio)” on page 9-83

“CovRatio Example” on page 9-84

“Delete-1 Scaled Difference in Coefficient Estimates (Dfbetas)” on page 9-86

“Dfbetas Example” on page 9-87

“Delete-1 Scaled Change in Fitted Values (Dffits)” on page 9-87

“Dffits Example” on page 9-88

“Delete-1 Variance (S2\_i)” on page 9-90

“Delete-1 Variance Example” on page 9-91

## Delete-1 Change in Covariance (covratio)

### Purpose

Delete-1 change in covariance (covratio) identifies the observations that are influential in the regression fit. An influential observation is one where its exclusion from the model might significantly alter the regression function. Values of covratio larger than  $1 + 3*p/n$  or smaller than  $1 - 3*p/n$  indicate influential points, where  $p$  is the number of regression coefficients, and  $n$  is the number of observations.

### Definition

The covratio statistic is the ratio of the determinant of the coefficient covariance matrix with observation  $i$  deleted to the determinant of the covariance matrix for the full model:

$$\text{covratio} = \frac{\det\{MSE(i)[X'(i)X(i)]^{-1}\}}{\det[MSE(XX)^{-1}]}$$

CovRatio is an  $n$ -by-1 vector in the Diagnostics dataset array of the fitted LinearModel object. Each element is the ratio of the generalized variance of

the estimated coefficients when the corresponding element is deleted to the generalized variance of the coefficients using all the data.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the `CovRatio` by indexing into the property using dot notation

```
mdl.Diagnostics.CovRatio
```

- Plot the delete-1 change in covariance using

```
plotDiagnostics(mdl, 'CovRatio')
```

For details, see the `plotDiagnostics` method of the `LinearModel` class.

### CovRatio Example

This example shows how to use the `CovRatio` statistics to determine the influential points in data.

Load the sample data and define the response and predictor variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

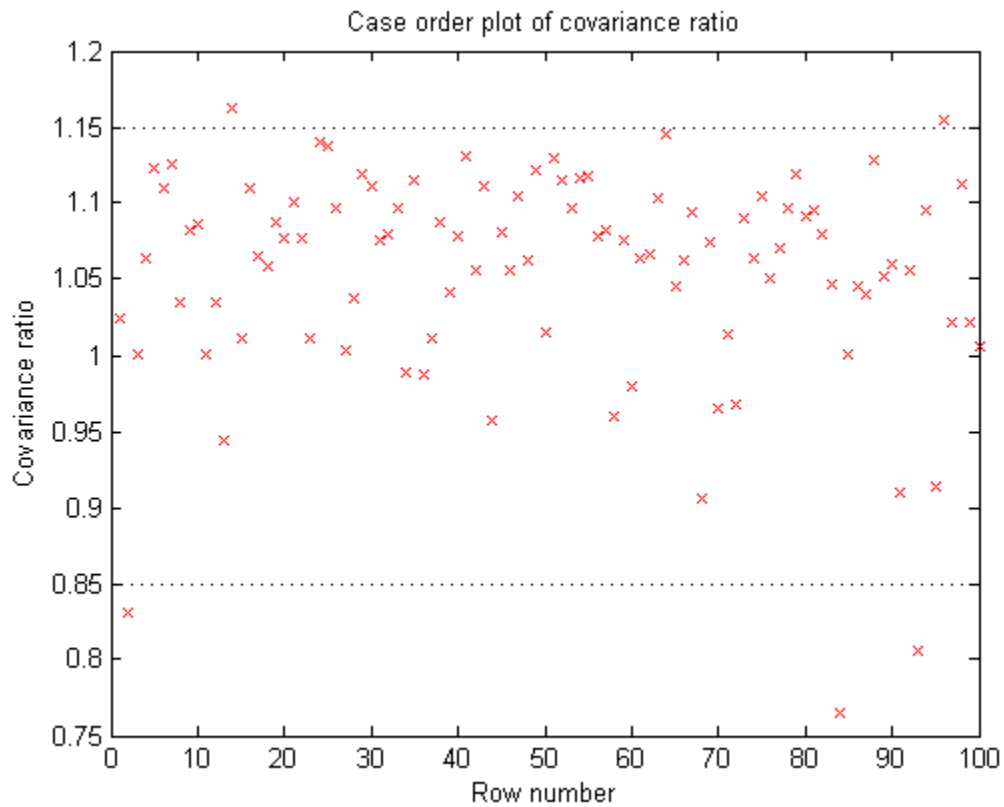
Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Plot the `CovRatio` statistics.

```
plotDiagnostics(mdl, 'CovRatio')
```





For this example, the threshold limits are  $1 + 3*5/100 = 1.15$  and  $1 - 3*5/100 = 0.85$ . There are a few points beyond the limits, which might be influential points.

Find the observations that are beyond the limits.

```
find((mdl.Diagnostics.CovRatio)>1.15|(mdl.Diagnostics.CovRatio)<0.85)
```

```
ans =
```

```
2  
14  
84
```

93

96

## Delete-1 Scaled Difference in Coefficient Estimates (Dfbetas)

### Purpose

The sign of a delete-1 scaled difference in coefficient estimate (Dfbetas) for coefficient  $j$  and observation  $i$  indicates whether that observation causes an increase or decrease in the estimate of the regression coefficient. The absolute value of a Dfbetas indicates the magnitude of the difference relative to the estimated standard deviation of the regression coefficient. A Dfbetas value larger than  $3/\sqrt{n}$  in absolute value indicates that the observation has a large influence on the corresponding coefficient.

### Definition

Dfbetas for coefficient  $j$  and observation  $i$  is the ratio of the difference in the estimate of coefficient  $j$  using all observations and the one obtained by removing observation  $i$ , and the standard error of the coefficient estimate obtained by removing observation  $i$ . The Dfbetas for coefficient  $j$  and observation  $i$  is

$$Dfbetas_{ij} = \frac{b_j - b_{j(i)}}{\sqrt{MSE_{(i)}(1 - h_{ii})}},$$

where  $b_j$  is the estimate for coefficient  $j$ ,  $b_{j(i)}$  is the estimate for coefficient  $j$  by removing observation  $i$ ,  $MSE_{(i)}$  is the mean squared error of the regression fit by removing observation  $i$ , and  $h_{ii}$  is the leverage value for observation  $i$ . Dfbetas is an  $n$ -by- $p$  matrix in the `Diagnostics` dataset array of the fitted `LinearModel` object. Each cell of Dfbetas corresponds to the Dfbetas value for the corresponding coefficient obtained by removing the corresponding observation.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can obtain the Dfbetas values as an  $n$ -by- $p$  matrix by indexing into the property using dot notation,

```
mdl.Diagnostics.Dfbetas
```

## Dfbetas Example

This example shows how to determine the observations that have large influence on coefficients using Dfbetas.

Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Find the Dfbetas values that are high in absolute value.

```
[row,col] = find(abs(mdl.Diagnostics.Dfbetas)>3/sqrt(100));
disp([row col])
```

```

 2     1
28     1
84     1
93     1
 2     2
13     3
84     3
 2     4
84     4
```

## Delete-1 Scaled Change in Fitted Values (Dffits)

### Purpose

The delete-1 scaled change in fitted values (Dffits) show the influence of each observation on the fitted response values. Dffits values with an absolute value larger than  $2\sqrt{p/n}$  might be influential.

**Definition**

Dffits for observation  $i$  is

$$Dffits_i = sr_i \sqrt{\frac{h_{ii}}{1-h_{ii}}},$$

where  $sr_i$  is the studentized residual, and  $h_{ii}$  is the leverage value of the fitted `LinearModel` object. `Dffits` is an  $n$ -by-1 column vector in the `Diagnostics` dataset array of the fitted `LinearModel` object. Each element in `Dffits` is the change in the fitted value caused by deleting the corresponding observation and scaling by the standard error.

**How To**

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the `Dffits` values by indexing into the property using dot notation

```
mdl.Diagnostics.Dffits
```

- Plot the delete-1 scaled change in fitted values using

```
plotDiagnostics(mdl, 'Dffits')
```

For details, see the `plotDiagnostics` method of the `LinearModel` class for details.

**Dffits Example**

This example shows how to determine the observations that are influential on the fitted response values using `Dffits` values.

Load the sample data and define the response and independent variables.

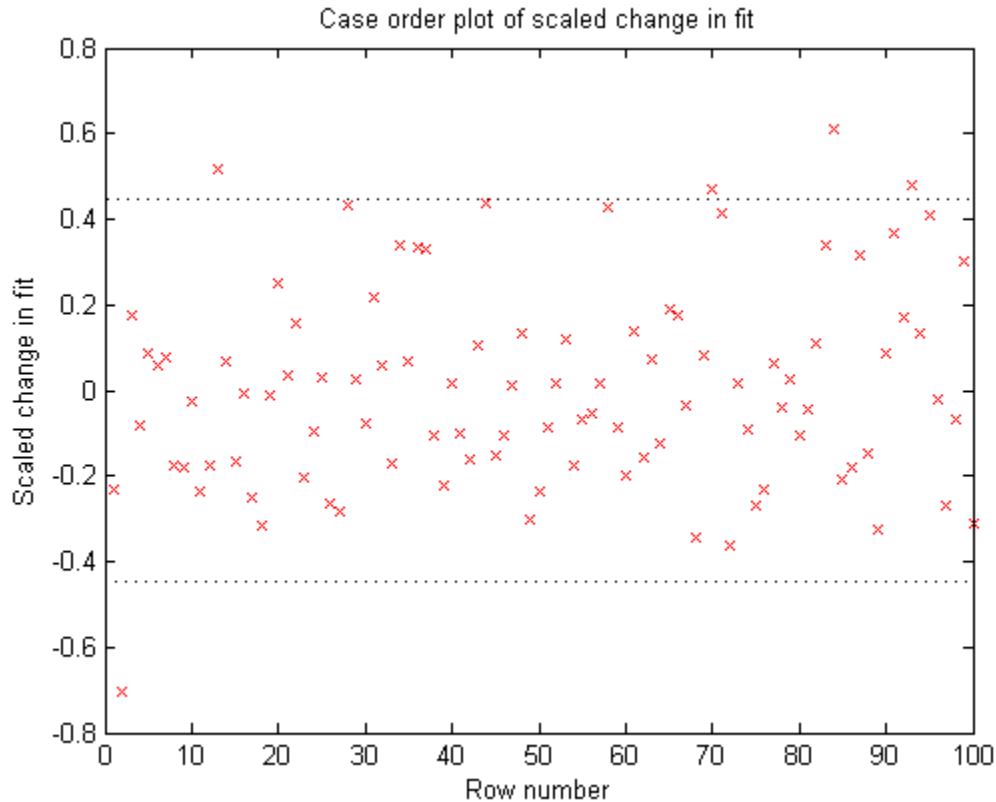
```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Plot the Dffits values.

```
plotDiagnostics mdl, 'Dffits')
```



The influential threshold limit for the absolute value of Dffits in this example is  $2\sqrt{5/100} = 0.45$ . Again, there are some observations with Dffits values beyond the recommended limits.

Find the Dffits values that are large in absolute value.

```
find(abs(mdl.Diagnostics.Dffits)>2*sqrt(4/100))
```

```
ans =
```

2  
13  
28  
44  
58  
70  
71  
84  
93  
95

## Delete-1 Variance (S2\_i)

### Purpose

The delete-1 variance (S2\_i) shows how the mean squared error changes when an observation is removed from the data set. You can compare the S2\_i values with the value of the mean squared error.

### Definition

S2\_i is a set of residual variance estimates obtained by deleting each observation in turn. The S2\_i value for observation  $i$  is

$$S2\_i = MSE_{(i)} = \frac{\sum_{j \neq i}^n [y_j - y_{j(i)}]^2}{n - p - 1},$$

where  $y_j$  is the  $j$ th observed response value. S2\_i is an  $n$ -by-1 vector in the Diagnostics dataset array of the fitted LinearModel object. Each element in S2\_i is the mean squared error of the regression obtained by deleting that observation.

### How To

After obtaining a fitted model, say, mdl, using fitlm or stepwiselm, you can:

- Display the S2\_i vector by indexing into the property using dot notation

```
mdl.Diagnostics.S2_i
```

- Plot the delete-1 variance values using

```
plotDiagnostics mdl, 'S2_i' )
```

For details, see the `plotDiagnostics` method of the `LinearModel` class.

## Delete-1 Variance Example

This example shows how to compute and plot `S2_i` values to examine the change in the mean squared error when an observation is removed from the data.

Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Display the MSE value for the model.

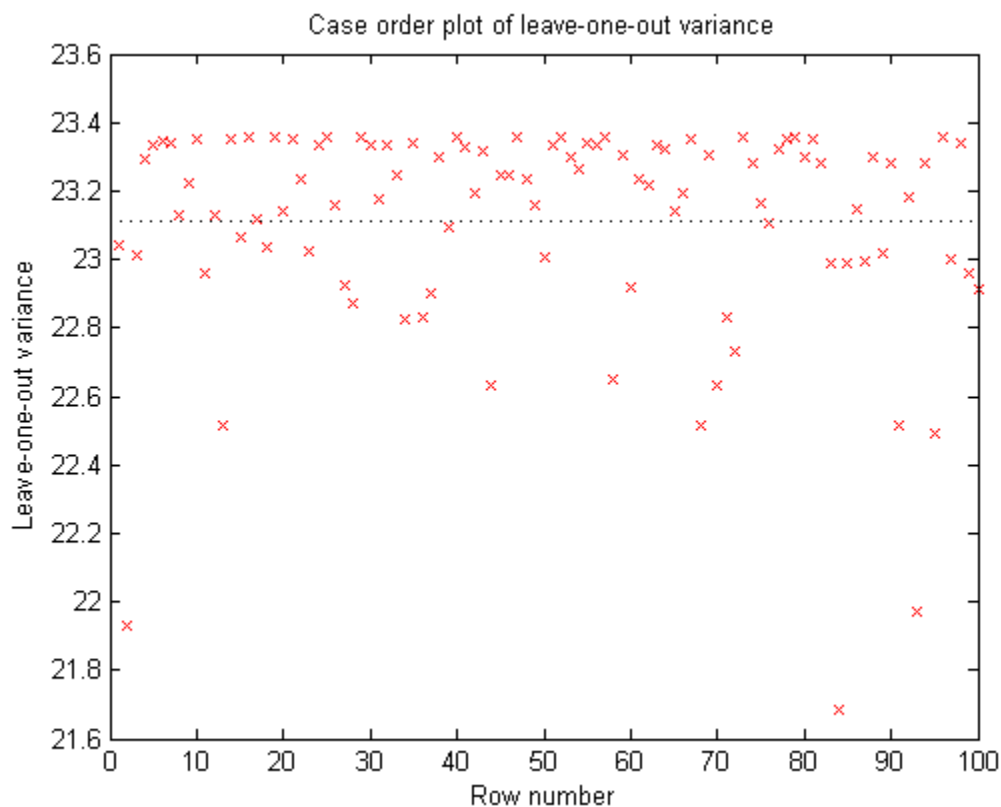
```
mdl.MSE
```

```
ans =
```

```
23.1140
```

Plot the `S2_i` values.

```
plotDiagnostics(mdl, 'S2_i' )
```



This plot makes it easy to compare the  $S^2_i$  values to the MSE value of 23.114, indicated by the horizontal dashed lines. You can see how deleting one observation changes the error variance.

### See Also

`LinearModel` | `fitlm` | `stepwiselmplotDiagnostics` | `plotResiduals` |

### Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64



## Durbin-Watson Test

### Purpose

The Durbin-Watson test assesses whether there is autocorrelation among the residuals or not.

### Definition

The Durbin-Watson test statistic,  $DW$ , is

$$DW = \frac{\sum_{i=1}^{n-1} (r_{i+1} - r_i)^2}{\sum_{i=1}^n r_i^2}.$$

Here,  $r_i$  is the  $i$ th raw residual, and  $n$  is the number of observations.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can perform the Durbin-Watson test using

```
dwtest(mdl)
```

For details, see the `dwtest` method of the `LinearModel` class.

### Example

This example shows how to test for autocorrelation among the residuals of a linear regression model.

Load the sample data and fit a linear regression model.

```
load hald
mdl = fitlm(ingredients,heat);
```

Perform a two-sided Durbin-Watson test to determine if there is any autocorrelation among the residuals of the linear model, `mdl`.

```
[p,DW] = dwtest mdl,'exact','both')
```

```
p =
```

```
0.6285
```

```
DW =
```

```
2.0526
```

The value of the Durbin-Watson test statistic is 2.0526. The  $p$ -value of 0.6285 suggest that the residuals are not autocorrelated.

### See Also

[LinearModel](#) | [fitlm](#) | [stepwiselplotResiduals](#) | [dwtest](#) |

### Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

## F-statistic and t-statistic

### In this section...

“F-statistic” on page 9-95

“F-statistic Example” on page 9-96

“t-statistic” on page 9-98

“t-statistic Example” on page 9-99

## F-statistic

### Purpose

In linear regression, the F-statistic is the test statistic for the analysis of variance (ANOVA) approach to test the significance of the model or the components in the model.

### Definition

The F-statistic in the linear model output display is the test statistic for testing the statistical significance of the model. The F-statistic values in the anova display are for assessing the significance of the terms or components in the model.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Find the F-statistic vs. constant model in the output display or by using

```
disp(mdl)
```

- Display the ANOVA for the model using

```
anova(mdl, 'summary')
```

- Obtain the F-statistic values for the components, except for the constant term using

```
anova mdl)
```

For details, see the `anova` method of the `LinearModel` class.

## F-statistic Example

This example shows how to use assess the fit of the model and the significance of the regression coefficients using F-statistic.

Load the sample data.

```
load carbig
ds = dataset(Acceleration,Cylinders,Weight,MPG);
ds.Cylinders = ordinal(Cylinders);
```

Fit a linear regression model.

```
mdl = fitlm(ds,'MPG~Acceleration*Weight+Cylinders+Weight^2')
```

```
mdl =
```

Linear regression model:

```
MPG ~ 1 + Cylinders + Acceleration*Weight + Weight^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	50.816	7.5669	6.7156	6.661e-
Acceleration	0.023343	0.33931	0.068796	0.945
Cylinders_4	7.167	2.0596	3.4798	0.00055
Cylinders_5	10.963	3.1299	3.5028	0.000513
Cylinders_6	4.7415	2.1257	2.2306	0.0262
Cylinders_8	5.057	2.2981	2.2005	0.0283
Weight	-0.017497	0.0034674	-5.0461	6.9371e-
Acceleration:Weight	7.0745e-05	0.00011171	0.6333	0.526
Weight^2	1.5767e-06	3.6909e-07	4.2719	2.4396e-

Number of observations: 398, Error degrees of freedom: 389

Root Mean Squared Error: 4.02

R-squared: 0.741, Adjusted R-Squared 0.736

F-statistic vs. constant model: 139, p-value = 2.94e-109

The F-statistic of the linear fit versus the constant model is 139, with a  $p$ -value of 2.94e-109. The model is significant at the 5% significance level. The R-squared value of 0.741 means the model explains about 74% of the variability in the response.

Display the ANOVA table for the fitted model.

```
anova(md1, 'summary')
```

```
ans =
```

	SumSq	DF	MeanSq	F	pValue
Total	24253	397	61.09		
Model	17981	8	2247.6	139.41	2.9432e-109
. Linear	17667	6	2944.4	182.63	7.5446e-110
. Nonlinear	314.36	2	157.18	9.7492	7.3906e-05
Residual	6271.6	389	16.122		
. Lack of fit	6267.1	387	16.194	7.1973	0.12968
. Pure error	4.5	2	2.25		

This display separates the variability in the model into linear and nonlinear terms. Since there are two non-linear terms ( $Weight^2$  and the interaction between *Weight* and *Acceleration*), the nonlinear degrees of freedom in the DF column is 2. There are six linear terms in the model (four *Cylinders* indicator variables, *Weight*, and *Acceleration*). The corresponding F-statistics in the F column are for testing the significance of the linear and nonlinear terms as separate groups.

The residual term is also separated into two parts; first is the error due to the lack of fit, and second is the pure error independent from the model, obtained from the replicated observations. The corresponding F-statistics in the F column are for testing the lack of fit, that is, whether the proposed model is an adequate fit or not.

Display the ANOVA table for the model terms.

```
anova(md1)
```

```
ans =
```

	SumSq	DF	MeanSq	F	pValue
Acceleration	104.99	1	104.99	6.5122	0.011095
Cylinders	408.94	4	102.23	6.3412	5.9573e-05
Weight	2187.5	1	2187.5	135.68	4.1974e-27
Acceleration:Weight	6.4662	1	6.4662	0.40107	0.52691
Weight^2	294.22	1	294.22	18.249	2.4396e-05
Error	6271.6	389	16.122		

This display decomposes the ANOVA table into the model terms. The corresponding F-statistics in the F column are for assessing the statistical significance of each term. The F-test for Cylinders test whether at least one of the coefficients of indicator variables for cylinders categories is different from zero or not. That is, whether different numbers of cylinders have a significant effect on MPG or not. The degrees of freedom for each model term is the numerator degrees of freedom for the corresponding F-test. Most of the terms have 1 degree of freedom, but the degrees of freedom for Cylinders is 4. Because there are four indicator variables for this term.

## t-statistic

### Purpose

In linear regression, the  $t$ -statistic is useful for making inferences about the regression coefficients. The hypothesis test on coefficient  $i$  tests the null hypothesis that it is equal to zero – meaning the corresponding term is not significant – versus the alternate hypothesis that the coefficient is different from zero.

### Definition

For a hypotheses test on coefficient  $i$ , with

$$H_0 : \beta_i = 0$$

$$H_1 : \beta_i \neq 0,$$

the  $t$ -statistic is:

$$t = \frac{b_i}{SE(b_i)},$$

where  $SE(b_i)$  is the standard error of the estimated coefficient  $b_i$ .

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Find the coefficient estimates, the standard errors of the estimates (SE), and the  $t$ -statistic values of hypothesis tests for the corresponding coefficients (`tStat`) in the output display.
- Call for the display using

```
display(mdl)
```

## t-statistic Example

This example shows how to test for the significance of the regression coefficients using  $t$ -statistic.

Load the sample data and fit the linear regression model.

```
load hald
mdl = fitlm(ingredients,heat)
```

```
mdl =
```

```
Linear regression model:
y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	62.405	70.071	0.8906	0.39913
x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

Number of observations: 13, Error degrees of freedom: 8

```

Root Mean Squared Error: 2.45
R-squared: 0.982, Adjusted R-Squared 0.974
F-statistic vs. constant model: 111, p-value = 4.76e-07

```

You can see that for each coefficient,  $tStat = Estimate/SE$ . The  $p$ -values for the hypotheses tests are in the `pValue` column. Each  $t$ -statistic tests for the significance of each term given other terms in the model. According to these results, none of the coefficients seem significant at the 5% significance level, although the R-squared value for the model is really high at 0.97. This often indicates possible multicollinearity among the predictor variables.

Use stepwise regression to decide which variables to include in the model.

```

load hald
mdl = stepwiselm(ingredients,heat)

1. Adding x4, FStat = 22.7985, pValue = 0.000576232
2. Adding x1, FStat = 108.2239, pValue = 1.105281e-06

mdl =

```

```

Linear regression model:
y ~ 1 + x1 + x4

```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	103.1	2.124	48.54	3.3243e-13
x1	1.44	0.13842	10.403	1.1053e-06
x4	-0.61395	0.048645	-12.621	1.8149e-07

```

Number of observations: 13, Error degrees of freedom: 10
Root Mean Squared Error: 2.73
R-squared: 0.972, Adjusted R-Squared 0.967
F-statistic vs. constant model: 177, p-value = 1.58e-08

```

In this example, `stepwiselm` starts with the constant model (default) and uses forward selection to incrementally add `x4` and `x1`. Each predictor variable in the final model is significant given the other one is in the model. The algorithm stops when adding none of the other predictor variables



significantly improves in the model. For details on stepwise regression, see `stepwiselm`.

**See Also**

`LinearModel` | `fitlm` | `stepwiselmanova` | `coefCI` | `coefTest` |

**Related Examples**

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

## Hat Matrix and Leverage

**In this section...**

“Hat Matrix” on page 9-102

“Leverage” on page 9-103

“Determine High Leverage Observations” on page 9-104

### Hat Matrix

**Purpose**

The hat matrix provides a measure of leverage. It is useful for investigating whether one or more observations are outlying with regard to their  $X$  values, and therefore might be excessively influencing the regression results.

**Definition**

The hat matrix is also known as the *projection matrix* because it projects the vector of observations,  $y$ , onto the vector of predictions,  $\hat{y}$ , thus putting the "hat" on  $y$ . The hat matrix  $H$  is defined in terms of the data matrix  $X$ :

$$H = X(X^T X)^{-1} X^T$$

and determines the fitted or predicted values since

$$\hat{y} = Hy = Xb.$$

The diagonal elements of  $H$ ,  $h_{ii}$ , are called leverages and satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $p$  is the number of coefficients, and  $n$  is the number of observations (rows of  $X$ ) in the regression model. `HatMatrix` is an  $n$ -by- $n$  matrix in the `Diagnostics` dataset array.

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the `HatMatrix` by indexing into the property using dot notation

```
mdl.Diagnostics.HatMatrix
```

When  $n$  is large, `HatMatrix` might be computationally expensive. In those cases, you can obtain the diagonal values directly, using

```
mdl.Diagnostics.Leverage
```

## Leverage

### Purpose

Leverage is a measure of the effect of a particular observation on the regression predictions due to the position of that observation in the space of the inputs. In general, the farther a point is from the center of the input space, the more leverage it has. Because the sum of the leverage values is  $p$ , an observation  $i$  can be considered as an outlier if its leverage substantially exceeds the mean leverage value,  $p/n$ , for example, a value larger than  $2^*p/n$ .

### Definition

The leverage of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix,  $H$ , where

$$H = X(X^T X)^{-1} X^T.$$

The diagonal terms satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $p$  is the number of coefficients in the regression model, and  $n$  is the number of observations. The minimum value of  $h_{ii}$  is  $1/n$  for a model with a constant term. If the fitted model goes through the origin, then the minimum leverage value is 0 for an observation at  $x = 0$ .

It is possible to express the fitted values,  $\hat{y}$ , by the observed values,  $y$ , since

$$\hat{y} = Hy = Xb.$$

Hence,  $h_{ii}$  expresses how much the observation  $y_i$  has impact on  $\hat{y}_i$ . A large value of  $h_{ii}$  indicates that the  $i$ th case is distant from the center of all X values for all  $n$  cases and has more leverage. Leverage is an  $n$ -by-1 column vector in the Diagnostics dataset array.

### How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Display the Leverage vector by indexing into the property using dot notation

```
mdl.Diagnostics.Leverage
```

- Plot the leverage for the values fitted by your model using

```
plotDiagnostics(mdl)
```

See the `plotDiagnostics` method of the `LinearModel` class for details.

### Determine High Leverage Observations

This example shows how to compute Leverage values and assess high leverage observations.

Load the sample data and define the response and independent variables.

```
load hospital
y = hospital.BloodPressure(:,1);
X = double(hospital(:,2:5));
```

Fit a linear regression model.

```
mdl = fitlm(X,y);
```

Plot the leverage values.

```
plotDiagnostics(mdl)
```



For this example, the recommended threshold value is  $2*5/100 = 0.1$ . There is no indication of high leverage observations.

## See Also

[LinearModel](#) | [fitlm](#) | [stepwiselmplotDiagnostics](#) |

**Related  
Examples**

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

# Residuals

## Purpose

Residuals are useful for detecting outlying  $y$  values and checking the linear regression assumptions with respect to the error term in the regression model. High-leverage observations have smaller residuals because they often shift the regression line or surface closer to them. You can also use residuals to detect some forms of heteroscedasticity and autocorrelation.

## Definition

The Residuals matrix is an  $n$ -by-4 dataset array containing a table of four types of residuals, with one row for each observation.

## Raw Residuals

Observed minus fitted values, that is,

$$r_i = y_i - \hat{y}_i.$$

## Pearson Residuals

Raw residuals divided by the root mean squared error, that is,

$$pr_i = \frac{r_i}{\sqrt{MSE}},$$

where  $r_i$  is the raw residual and  $MSE$  is the mean squared error.

## Standardized Residuals

Standardized residuals are raw residuals divided by their estimated standard deviation. The standardized residual for observation  $i$  is

$$st_i = \frac{r_i}{\sqrt{MSE(1 - h_{ii})}},$$

where  $MSE$  is the mean squared error and  $h_{ii}$  is the leverage value for observation  $i$ .

## Studentized Residuals

Studentized residuals are the raw residuals divided by an independent estimate of the residual standard deviation. The residual for observation  $i$  is divided by an estimate of the error standard deviation based on all observations except for observation  $i$ .

$$sr_i = \frac{r_i}{\sqrt{MSE_{(i)}(1 - h_{ii})}},$$

where  $MSE_{(i)}$  is the mean squared error of the regression fit calculated by removing observation  $i$ , and  $h_{ii}$  is the leverage value for observation  $i$ . The studentized residual  $sr_i$  has a  $t$ -distribution with  $n - p - 1$  degrees of freedom.

## How To

After obtaining a fitted model, say, `mdl`, using `fitlm` or `stepwiselm`, you can:

- Find the `Residuals` dataset array under `mdl` object.
- Obtain any of these columns as a vector by indexing into the property using dot notation, for example,

```
mdl.Residuals.Raw
```

- Plot any of the residuals for the values fitted by your model using

```
plotResiduals(mdl)
```

For details, see the `plotResiduals` method of the `LinearModel` class.

## Example

This example shows how to assess the model assumptions by examining the residuals of a fitted linear regression model.

Load the sample data and store the independent and response variables in a dataset array.

```
load imports-85
ds = dataset(X(:,7),X(:,8),X(:,9),X(:,15),'VarNames',...
{'curb_weight','engine_size','bore','price'});
```



Fit a linear regression model.

```
mdl = fitlm(ds)
```

```
mdl =
```

Linear regression model:

price ~ 1 + curb\_weight + engine\_size + bore

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	64.095	3.703	17.309	2.0481e-41
curb_weight	-0.0086681	0.0011025	-7.8623	2.42e-13
engine_size	-0.015806	0.013255	-1.1925	0.23452
bore	-2.6998	1.3489	-2.0015	0.046711

Number of observations: 201, Error degrees of freedom: 197

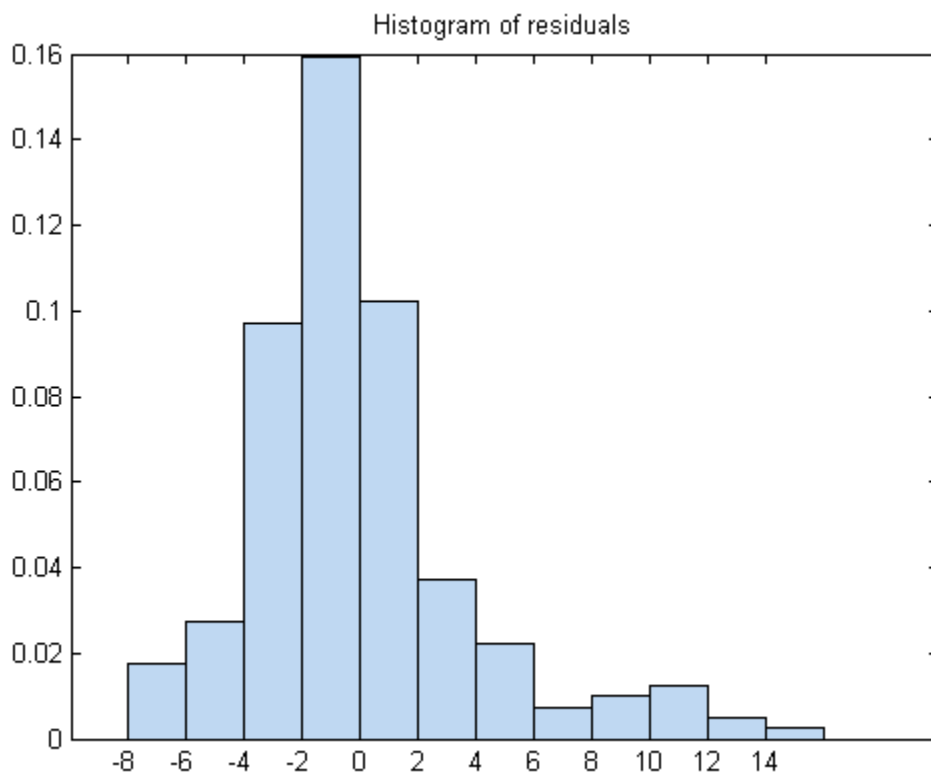
Root Mean Squared Error: 3.95

R-squared: 0.674, Adjusted R-Squared 0.669

F-statistic vs. constant model: 136, p-value = 1.14e-47

Plot the histogram of raw residuals.

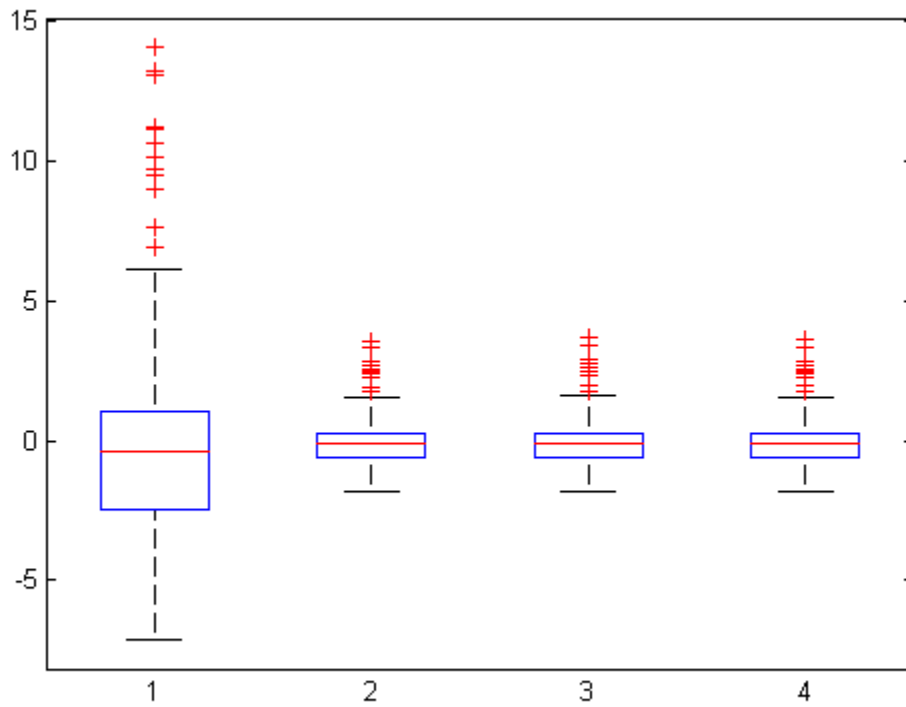
```
plotResiduals(mdl)
```



The histogram shows that the residuals are slightly right skewed.

Plot the box plot of all four types of residuals.

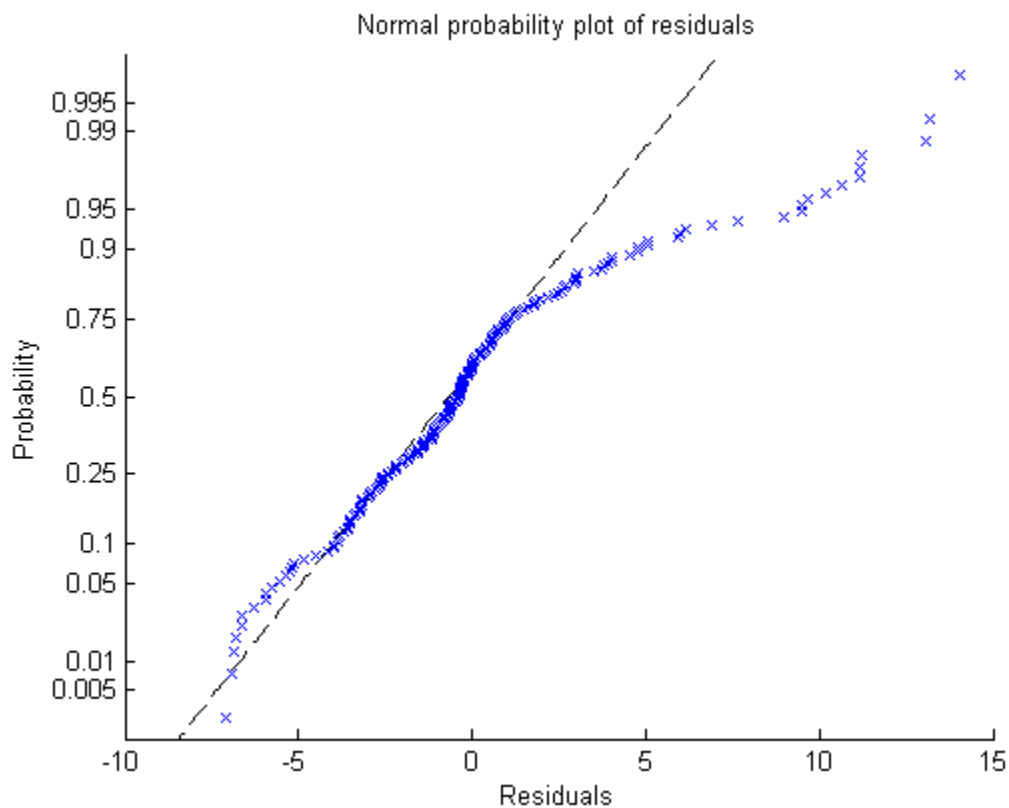
```
Res = double mdl.Residuals;
```



You can see the right-skewed structure of the residuals in the box plot as well.

Plot the normal probability plot of the raw residuals.

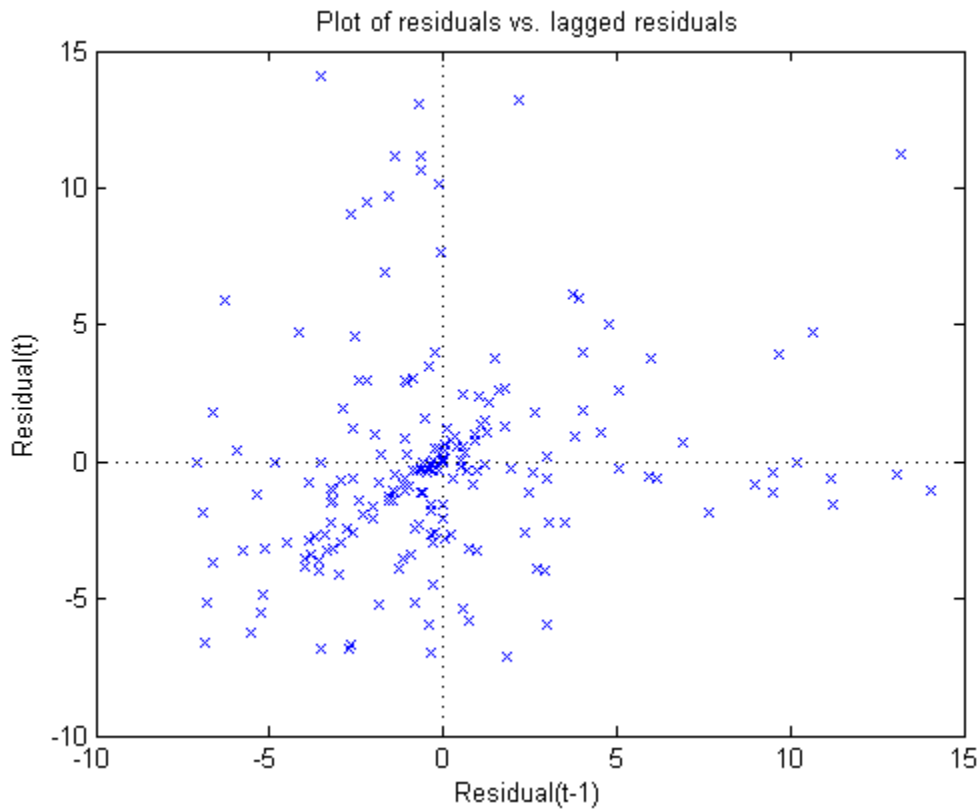
```
plotResiduals mdl, 'probability'  
boxplot(Res)
```



This normal probability plot also shows the deviation from normality and the skewness on the right tail of the distribution of residuals.

Plot the residuals versus lagged residuals.

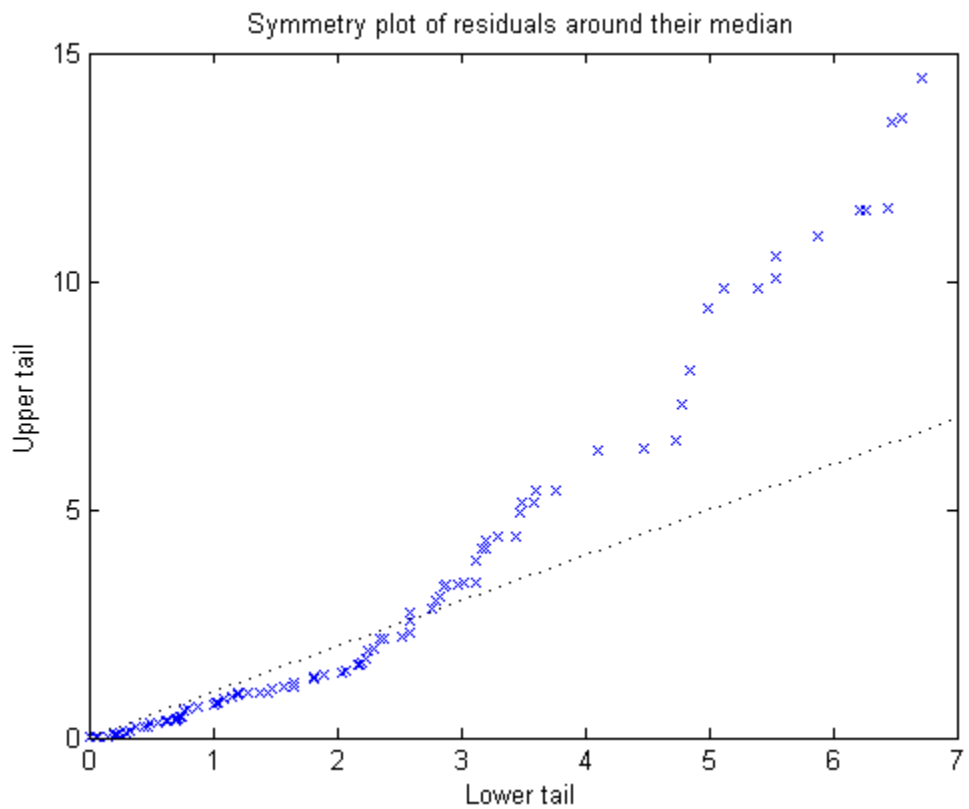
```
plotResiduals mdl, 'lagged'
```



This graph shows a trend, which indicates a possible correlation among the residuals. You can further check this using `dwtest(mdl)`. Serial correlation among residuals usually means that the model can be improved.

Plot the symmetry plot of residuals.

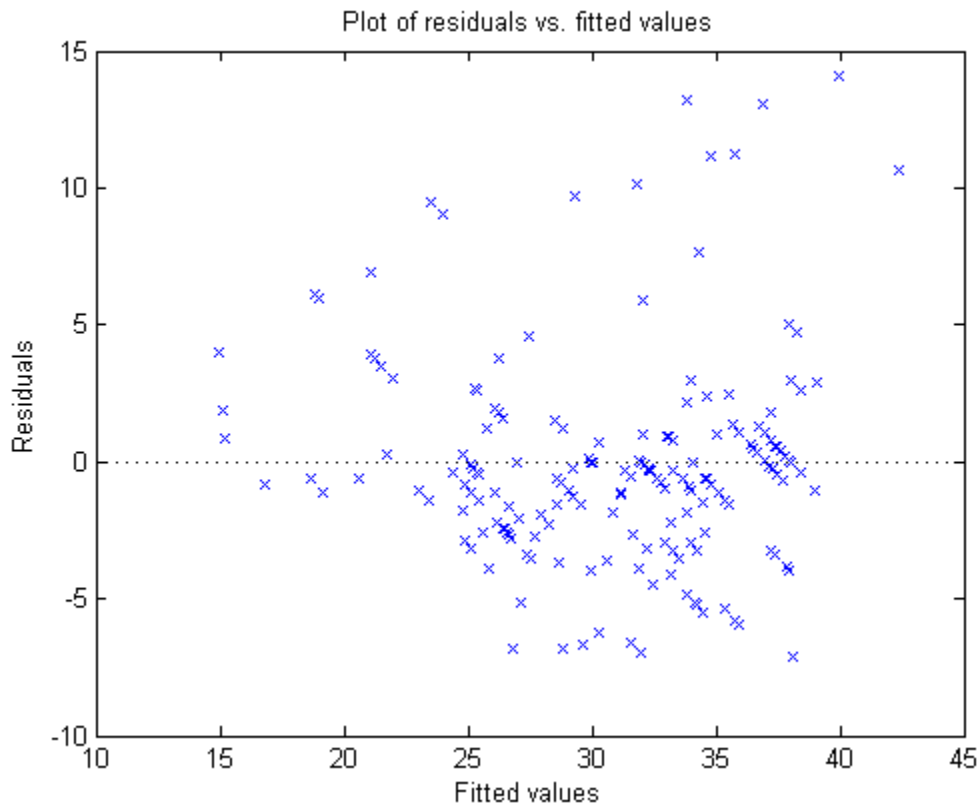
```
plotResiduals(mdl, 'symmetry')
```



This plot also suggests that the residuals are not distributed equally around their median, as would be expected for normal distribution.

Plot the residuals versus the fitted values.

```
plotResiduals mdl, 'fitted'
```



The increase in the variance as the fitted values increase suggests possible heteroscedasticity.

## References

- [1] Atkinson, A. T. *Plots, Transformations, and Regression. An Introduction to Graphical Methods of Diagnostic Regression Analysis*. New York: Oxford Statistical Science Series, Oxford University Press, 1987.

[2] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*. IRWIN, The McGraw-Hill Companies, Inc., 1996.

[3] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics, Identifying Influential Data and Sources of Collinearity*. Wiley Series in Probability and Mathematical Statistics, John Wiley and Sons, Inc., 1980.

### See Also

LinearModel | fitlm | stepwiselmp | plotResiduals | dwtest |

### Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64



## Summary of Output and Diagnostic Statistics

Name	LinearModel	regstats
“Cook’s Distance” on page 9-72	CooksDistance and cookd	cookd
“Coefficient Confidence Intervals” on page 9-77	coefCI	N/A
“Coefficient Covariance and Standard Errors” on page 9-76	CoefficientCovariance	covb
“Coefficient of Determination (R-Squared)” on page 9-80	Rsquared: Ordinary, Adjusted	rsquare, adjrsquare
“Delete-1 Change in Covariance (covratio)” on page 9-83	CovRatio	covratio
“Delete-1 Scaled Difference in Coefficient Estimates (Dfbetas)” on page 9-86	Dfbetas	dfbetas
“Delete-1 Scaled Change in Fitted Values (Dffits)” on page 9-87	Dffits	dffits
“Delete-1 Variance (S2_i)” on page 9-90	S2_i	s2_i
“Durbin-Watson Test” on page 9-93	dwtest	dwstat
“F-statistic” on page 9-95	Fstat	fstat
“Hat Matrix” on page 9-102	HatMatrix	hatmat
“Leverage” on page 9-103	Leverage	leverage

<b>Name</b>	<b>LinearModel</b>	<b>regstats</b>
“Residuals” on page 9-107	Residuals: Raw, Pearson, Studentized, Standardized	r, studres, standres
“t-statistic” on page 9-98	tstats	tstat

**See Also**

[LinearModel](#) | [fitlm](#) | [stepwiselplotDiagnostics](#) | [plotResiduals](#) | [dwtest](#) |

**Related Examples**

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Interpret Linear Regression Results” on page 9-64

## Stepwise Regression

### In this section...

“Stepwise Regression to Select Appropriate Models” on page 9-119

“Compare large and small stepwise models” on page 9-119

### Stepwise Regression to Select Appropriate Models

`stepwiselm` creates a linear model and automatically adds to or trims the model. To create a small model, start from a constant model. To create a large model, start with a model containing many terms. A large model usually has lower error as measured by the fit to the original data, but might not have any advantage in predicting new data.

`stepwiselm` can use all the name-value options from `fitlm`, with additional options relating to the starting and bounding models. In particular:

- For a small model, start with the default lower bounding model: 'constant' (a model that has no predictor terms).
- The default upper bounding model has linear terms and interaction terms (products of pairs of predictors). For an upper bounding model that also includes squared terms, set the Upper name-value pair to 'quadratic'.

### Compare large and small stepwise models

This example shows how to compare models that `stepwiselm` returns starting from a constant model and starting from a full interaction model.

Load the `carbig` data and create a dataset array from some of the data.

```
load carbig
ds = dataset(Acceleration,Displacement,Horsepower,Weight,MPG);
```

Create a mileage model stepwise starting from the constant model.

```
mdl1 = stepwiselm(ds,'constant','ResponseVar','MPG')
```

1. Adding Weight, FStat = 888.8507, pValue = 2.9728e-103
2. Adding Horsepower, FStat = 3.8217, pValue = 0.00049608

3. Adding Horsepower:Weight, FStat = 64.8709, pValue = 9.93362e-15

mdl1 =

Linear regression model:

MPG ~ 1 + Horsepower\*Weight

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	63.558	2.3429	27.127	1.2343e-91
Horsepower	-0.25084	0.027279	-9.1952	2.3226e-18
Weight	-0.010772	0.00077381	-13.921	5.1372e-36
Horsepower:Weight	5.3554e-05	6.6491e-06	8.0542	9.9336e-15

Number of observations: 392, Error degrees of freedom: 388

Root Mean Squared Error: 3.93

R-squared: 0.748, Adjusted R-Squared 0.746

F-statistic vs. constant model: 385, p-value = 7.26e-116

Create a mileage model stepwise starting from the full interaction model.

`mdl2 = stepwiselm(ds,'interactions','ResponseVar','MPG')`

1. Removing Acceleration:Displacement, FStat = 0.024186, pValue = 0.8765

2. Removing Displacement:Weight, FStat = 0.33103, pValue = 0.56539

3. Removing Acceleration:Horsepower, FStat = 1.7334, pValue = 0.18876

4. Removing Acceleration:Weight, FStat = 0.93269, pValue = 0.33477

5. Removing Horsepower:Weight, FStat = 0.64486, pValue = 0.42245

mdl2 =

Linear regression model:

MPG ~ 1 + Acceleration + Weight + Displacement\*Horsepower

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	61.285	2.8052	21.847	1.8593e-69

Acceleration	-0.34401	0.11862	-2.9	0.0039445
Displacement	-0.081198	0.010071	-8.0623	9.5014e-15
Horsepower	-0.24313	0.026068	-9.3265	8.6556e-19
Weight	-0.0014367	0.00084041	-1.7095	0.088166
Displacement:Horsepower	0.00054236	5.7987e-05	9.3531	7.0527e-19

Number of observations: 392, Error degrees of freedom: 386

Root Mean Squared Error: 3.84

R-squared: 0.761, Adjusted R-Squared 0.758

F-statistic vs. constant model: 246, p-value = 1.32e-117

Notice that:

- `mdl1` has four coefficients (the Estimate column), and `mdl2` has six coefficients.
- The adjusted R-squared of `mdl1` is 0.746, which is slightly less (worse) than that of `mdl2`, 0.758.

Create a mileage model stepwise with a full quadratic model as the upper bound, starting from the full quadratic model:

```
mdl3 = stepwiselm(ds,'quadratic',...
    'ResponseVar','MPG','Upper','quadratic');
```

Compare the three model complexities by examining their formulas.

`mdl1`.Formula

```
ans =
MPG ~ 1 + Horsepower*Weight
```

`mdl2`.Formula

```
ans =
MPG ~ 1 + Acceleration + Weight + Displacement*Horsepower
```

`mdl3`.Formula

```
ans =
MPG ~ 1 + Weight + Acceleration*Displacement
```

```
+ Displacement*Horsepower + Acceleration^2
```

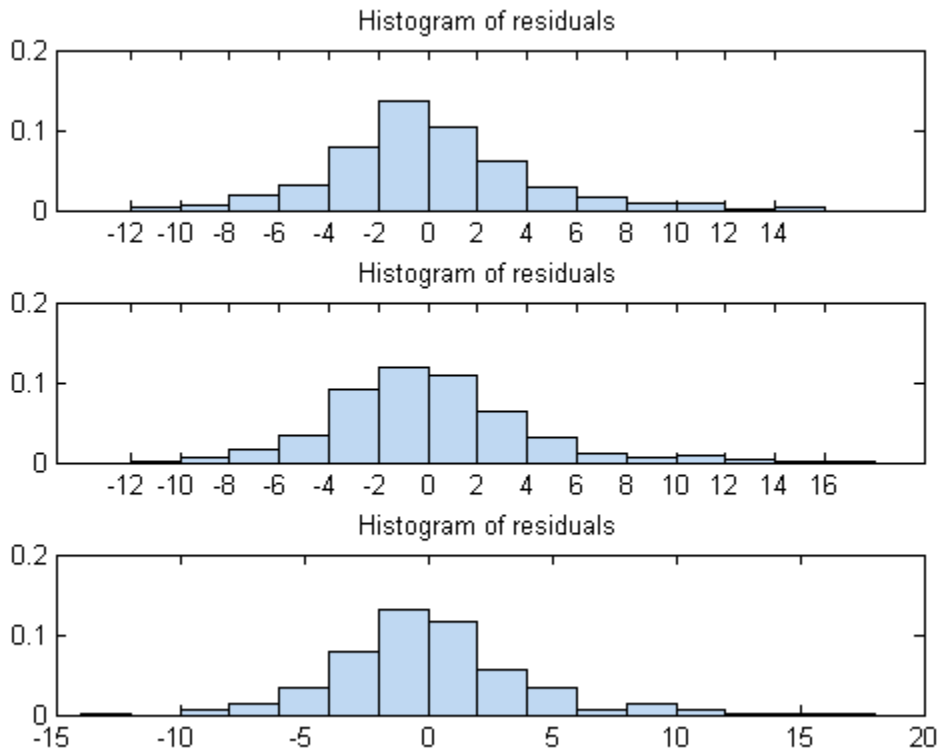
The adjusted  $R^2$  values improve slightly as the models become more complex:

```
RSquared = [mdl1.Rsquared.Adjusted, ...  
            mdl2.Rsquared.Adjusted, mdl3.Rsquared.Adjusted]
```

```
RSquared =  
    0.7465    0.7580    0.7599
```

Compare residual plots of the three models.

```
subplot(3,1,1)  
plotResiduals(mdl1)  
subplot(3,1,2)  
plotResiduals(mdl2)  
subplot(3,1,3)  
plotResiduals(mdl3)
```



The models have similar residuals. It is not clear which fits the data better.

Interestingly, the more complex models have larger maximum deviations of the residuals:

```

Range1 = [min(md11.Residuals.Raw),max(md11.Residuals.Raw)];
Range2 = [min(md12.Residuals.Raw),max(md12.Residuals.Raw)];
Range3 = [min(md13.Residuals.Raw),max(md13.Residuals.Raw)];
Ranges = [range1;range2;range3]

```

```

Ranges =
    -10.7725    14.7314
    -11.4407    16.7562

```

-12.2723    16.7927



## Robust Regression — Reduce Outlier Effects

### In this section...

“What Is Robust Regression?” on page 9-125

“Robust Regression versus Standard Least-Squares Fit” on page 9-125

### What Is Robust Regression?

The models described in “What Are Linear Regression Models?” on page 9-8 are based on certain assumptions, such as a normal distribution of errors in the observed responses. If the distribution of errors is asymmetric or prone to outliers, model assumptions are invalidated, and parameter estimates, confidence intervals, and other computed statistics become unreliable. Use `fitlm` with the `RobustOpts` name-value pair to create a model that is not much affected by outliers. The robust fitting method is less sensitive than ordinary least squares to large changes in small parts of the data.

Robust regression works by assigning a weight to each data point. Weighting is done automatically and iteratively using a process called *iteratively reweighted least squares*. In the first iteration, each point is assigned equal weight and model coefficients are estimated using ordinary least squares. At subsequent iterations, weights are recomputed so that points farther from model predictions in the previous iteration are given lower weight. Model coefficients are then recomputed using weighted least squares. The process continues until the values of the coefficient estimates converge within a specified tolerance.

### Robust Regression versus Standard Least-Squares Fit

This example shows how to use robust regression. It compares the results of a robust fit to a standard least-squares fit.

#### Step 1. Prepare data.

Load the moore data. The data is in the first five columns, and the response in the sixth.

```
load moore
X = [moore(:,1:5)];
```

```
y = moore(:,6);
```

**Step 2. Fit robust and nonrobust models.**

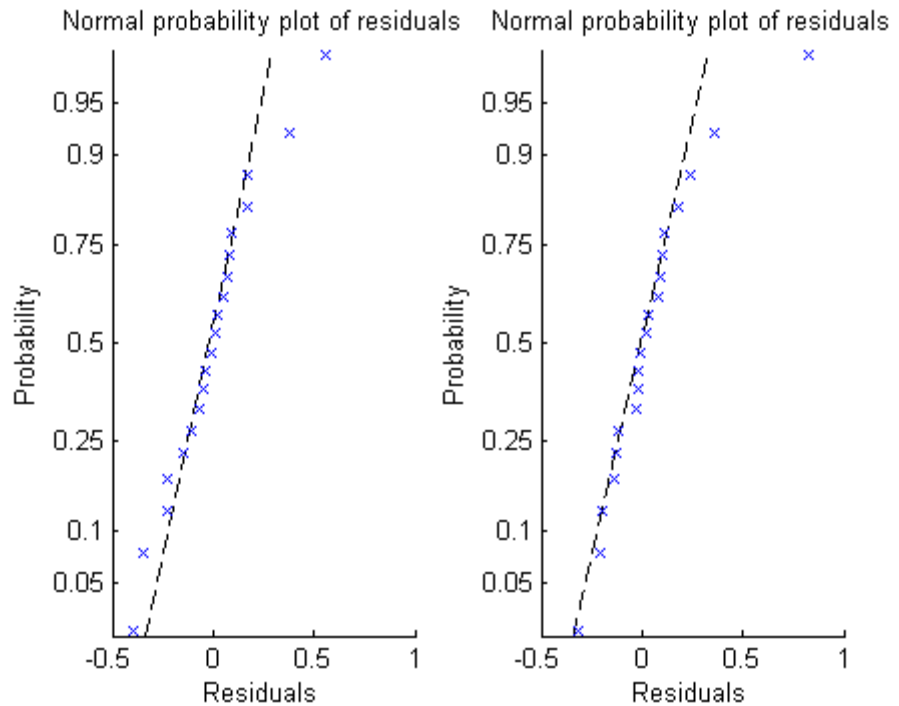
Fit two linear models to the data, one using robust fitting, one not.

```
mdl = fitlm(X,y); % not robust  
mdlr = fitlm(X,y,'RobustOpts','on');
```

**Step 3. Examine model residuals.**

Examine the residuals of the two models.

```
subplot(1,2,1);plotResiduals(mdl,'probability')  
subplot(1,2,2);plotResiduals(mdlr,'probability')
```



The residuals from the robust fit (right half of the plot) are nearly all closer to the straight line, except for the one obvious outlier.

#### 4. Remove the outlier from the standard model

Find the index of the outlier. Examine the weight of the outlier in the robust fit.

```
[~,outlier] = max mdlr.Residuals.Raw);  
mdlr.Robust.Weights(outlier)
```

```
ans =
```

```
0.0246
```

Check the median weight.

```
median(mdlr.Robust.Weights)
```

```
ans =
```

```
0.9718
```

This weight of the outlier in the robust fit is much less than a typical weight of an observation.

## Ridge Regression

### In this section...

“Introduction to Ridge Regression” on page 9-128

“Ridge Regression” on page 9-128

### Introduction to Ridge Regression

Coefficient estimates for the models described in “Linear Regression” on page 9-12 rely on the independence of the model terms. When terms are correlated and the columns of the design matrix  $X$  have an approximate linear dependence, the matrix  $(X^T X)^{-1}$  becomes close to singular. As a result, the least-squares estimate

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

becomes highly sensitive to random errors in the observed response  $y$ , producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

*Ridge regression* addresses the problem by estimating regression coefficients using

$$\hat{\beta} = (X^T X + kI)^{-1} X^T y$$

where  $k$  is the *ridge parameter* and  $I$  is the identity matrix. Small positive values of  $k$  improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of ridge estimates often result in a smaller mean square error when compared to least-squares estimates.

The Statistics Toolbox function `ridge` carries out ridge regression.

### Ridge Regression

For example, load the data in `acetylene.mat`, with observations of the predictor variables `x1`, `x2`, `x3`, and the response variable `y`:

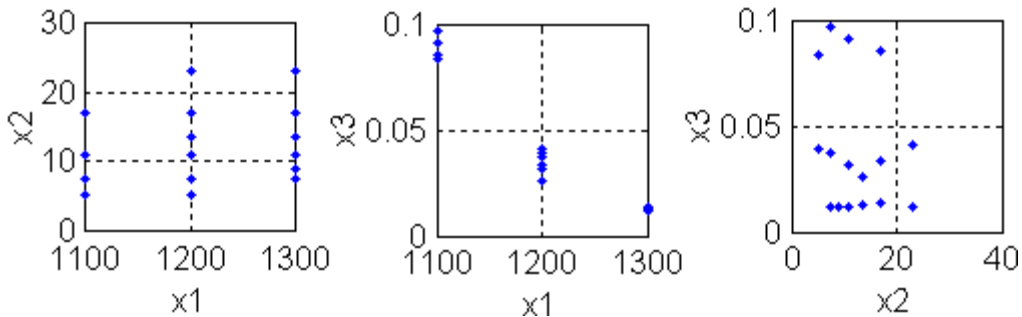
```
load acetylene
```

Plot the predictor variables against each other:

```
subplot(1,3,1)
plot(x1,x2,'. ')
xlabel('x1'); ylabel('x2'); grid on; axis square
```

```
subplot(1,3,2)
plot(x1,x3,'. ')
xlabel('x1'); ylabel('x3'); grid on; axis square
```

```
subplot(1,3,3)
plot(x2,x3,'. ')
xlabel('x2'); ylabel('x3'); grid on; axis square
```



Note the correlation between  $x_1$  and the other two predictor variables.

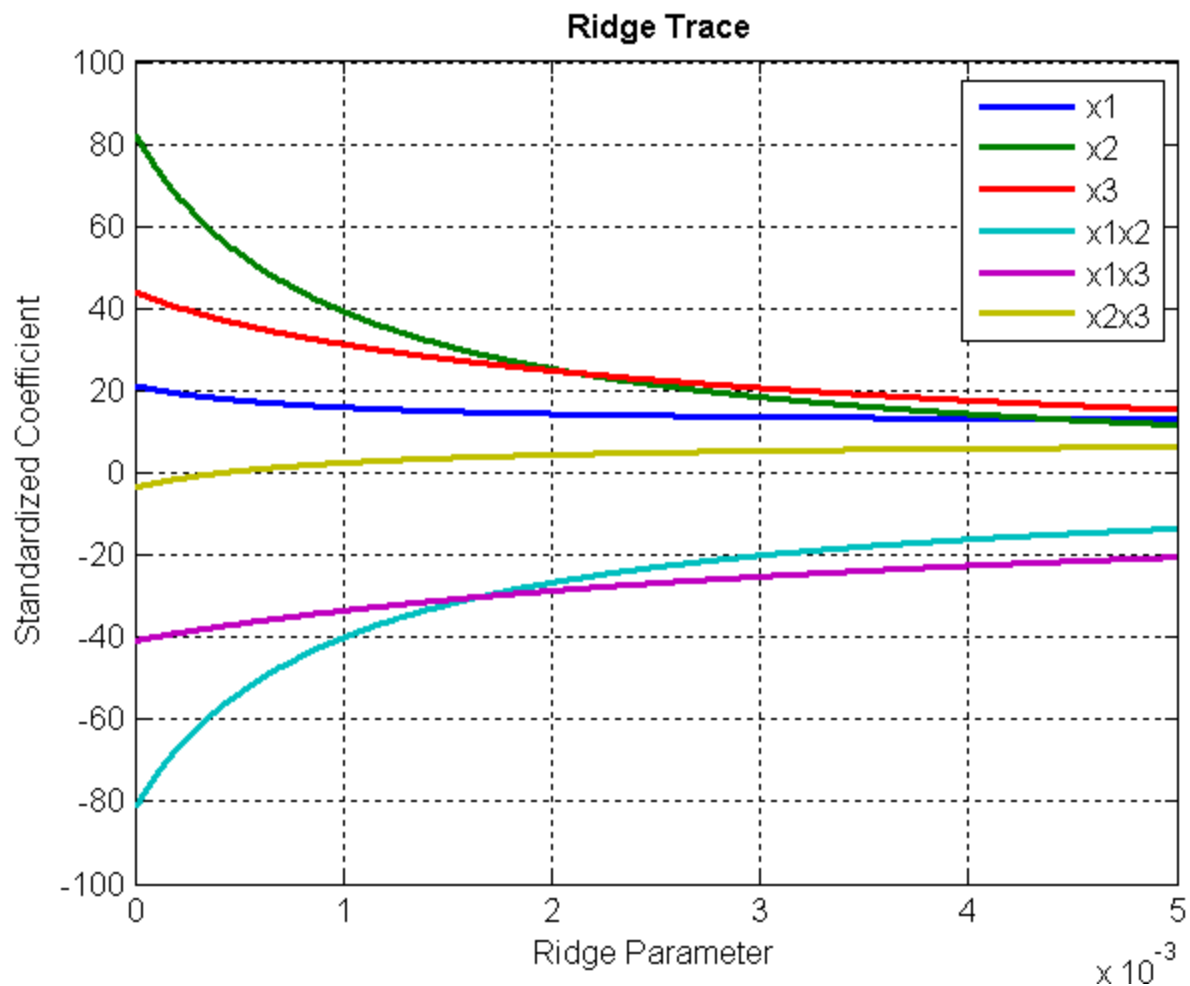
Use `ridge` and `x2fx` to compute coefficient estimates for a multilinear model with interaction terms, for a range of ridge parameters:

```
X = [x1 x2 x3];
D = x2fx(X,'interaction');
D(:,1) = []; % No constant term
k = 0:1e-5:5e-3;
betahat = ridge(y,D,k);
```

Plot the ridge trace:

```
figure
```

```
plot(k,betahat,'LineWidth',2)
ylim([-100 100])
grid on
xlabel('Ridge Parameter')
ylabel('Standardized Coefficient')
title('{\bf Ridge Trace}')
legend('x1','x2','x3','x1x2','x1x3','x2x3')
```



The estimates stabilize to the right of the plot. Note that the coefficient of the  $x_2x_3$  interaction term changes sign at a value of the ridge parameter  $\approx 5 \times 10^{-4}$ .

## Lasso and Elastic Net

### In this section...

“What Are Lasso and Elastic Net?” on page 9-132

“Lasso Regularization” on page 9-132

“Lasso and Elastic Net with Cross Validation” on page 9-135

“Wide Data via Lasso and Parallel Computing” on page 9-138

“Lasso and Elastic Net Details” on page 9-144

“References” on page 9-145

### What Are Lasso and Elastic Net?

Lasso is a regularization technique. Use lasso to:

- Reduce the number of predictors in a regression model.
- Identify important predictors.
- Select among redundant predictors.
- Produce shrinkage estimates with potentially lower predictive errors than ordinary least squares.

Elastic net is a related technique. Use elastic net when you have several highly correlated variables. `lasso` provides elastic net regularization when you set the Alpha name-value pair to a number strictly between 0 and 1.

See “Lasso and Elastic Net Details” on page 9-144.

For lasso regularization of regression ensembles, see `regularize`.

### Lasso Regularization

To see how `lasso` identifies and discards unnecessary predictors:

- 1 Generate 200 samples of five-dimensional artificial data  $X$  from exponential distributions with various means:



```

rng(3,'twister') % for reproducibility
X = zeros(200,5);
for ii = 1:5
    X(:,ii) = exprnd(ii,200,1);
end

```

- 2** Generate response data  $Y = X*r + \text{eps}$  where  $r$  has just two nonzero components, and the noise  $\text{eps}$  is normal with standard deviation 0.1:

```

r = [0;2;0;-3;0];
Y = X*r + randn(200,1)*.1;

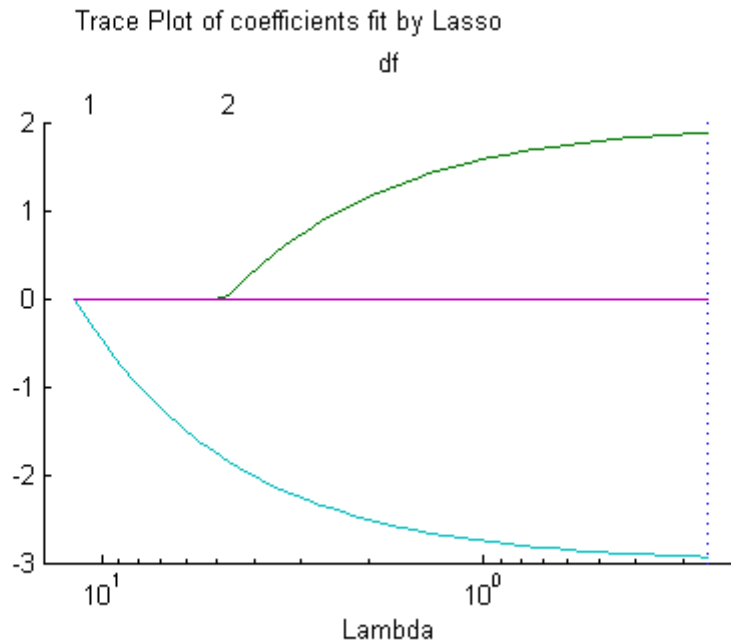
```

- 3** Fit a cross-validated sequence of models with `lasso`, and plot the result:

```

[b fitinfo] = lasso(X,Y,'CV',10);
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');

```



The plot shows the nonzero coefficients in the regression for various values of the Lambda regularization parameter. Larger values of Lambda appear on the left side of the graph, meaning more regularization, resulting in fewer nonzero regression coefficients.

The dashed vertical lines represent the Lambda value with minimal mean squared error (on the right), and the Lambda value with minimal mean squared error plus one standard deviation. This latter value is a recommended setting for Lambda. These lines appear only when you perform cross validation. Cross validate by setting the 'CV' name-value pair. This example uses 10-fold cross validation.

The upper part of the plot shows the degrees of freedom (df), meaning the number of nonzero coefficients in the regression, as a function of Lambda. On the left, the large value of Lambda causes all but one coefficient to be 0. On the right all five coefficients are nonzero, though the plot shows only two clearly. The other three coefficients are so small that you cannot visually distinguish them from 0.

For small values of Lambda (toward the right in the plot), the coefficient values are close to the least-squares estimate. See step 5 on page 9-135.

- 4 Find the Lambda value of the minimal cross-validated mean squared error plus one standard deviation. Examine the MSE and coefficients of the fit at that Lambda:

```
lam = fitinfo.Index1SE;  
fitinfo.MSE(lam)
```

```
ans =  
    0.1398
```

```
b(:,lam)
```

```
ans =  
     0  
    1.8855  
     0  
   -2.9367  
     0
```

lasso did a good job finding the coefficient vector  $\mathbf{r}$ .

- 5** For comparison, find the least-squares estimate of  $\mathbf{r}$ :

```
rhat = X\Y
```

```
rhat =
```

```
-0.0038
 1.9952
 0.0014
-2.9993
 0.0031
```

The estimate  $\mathbf{b}(:, \lambda)$  has slightly more mean squared error than the mean squared error of  $\mathbf{rhat}$ :

```
res = X*rhat - Y; % calculate residuals
MSEmin = res'*res/200 % b(:,lam) value is 0.1398
```

```
MSEmin =
 0.0088
```

But  $\mathbf{b}(:, \lambda)$  has only two nonzero components, and therefore can provide better predictive estimates on new data.

## Lasso and Elastic Net with Cross Validation

Consider predicting the mileage (MPG) of a car based on its weight, displacement, horsepower, and acceleration. The `carbig` data contains these measurements. The data seem likely to be correlated, making elastic net an attractive choice.

- 1** Load the data:

```
load carbig
```

- 2** Extract the continuous (noncategorical) predictors (lasso does not handle categorical predictors):

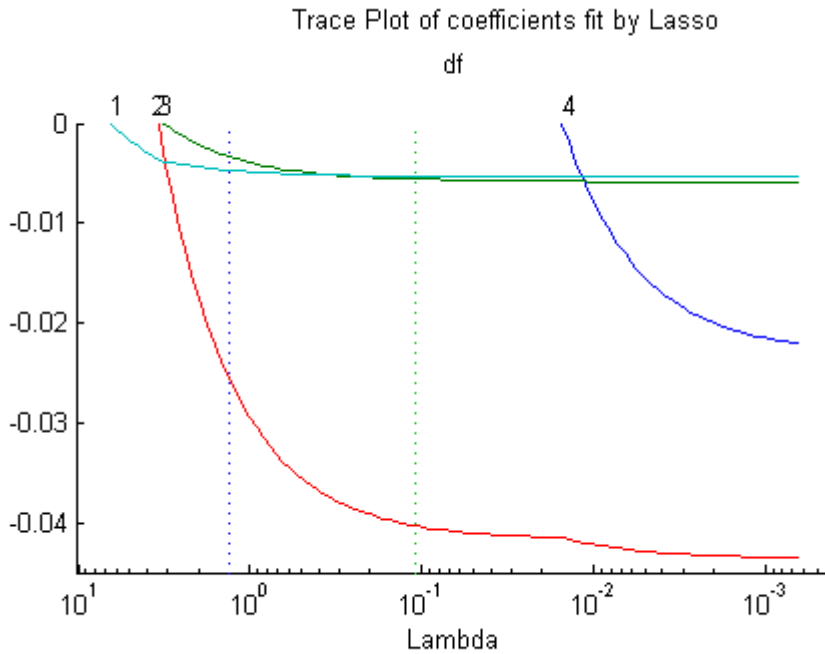
```
X = [Acceleration Displacement Horsepower Weight];
```

3 Perform a lasso fit with 10-fold cross validation:

```
[b fitinfo] = lasso(X,MPG,'CV',10);
```

4 Plot the result:

```
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');
```



5 Calculate the correlation of the predictors:

```
% Eliminate NaNs so corr runs
nonan = ~any(isnan([X MPG]),2);
Xnonan = X(nonan,:);
MPGnonan = MPG(nonan,:);
corr(Xnonan)

ans =
```

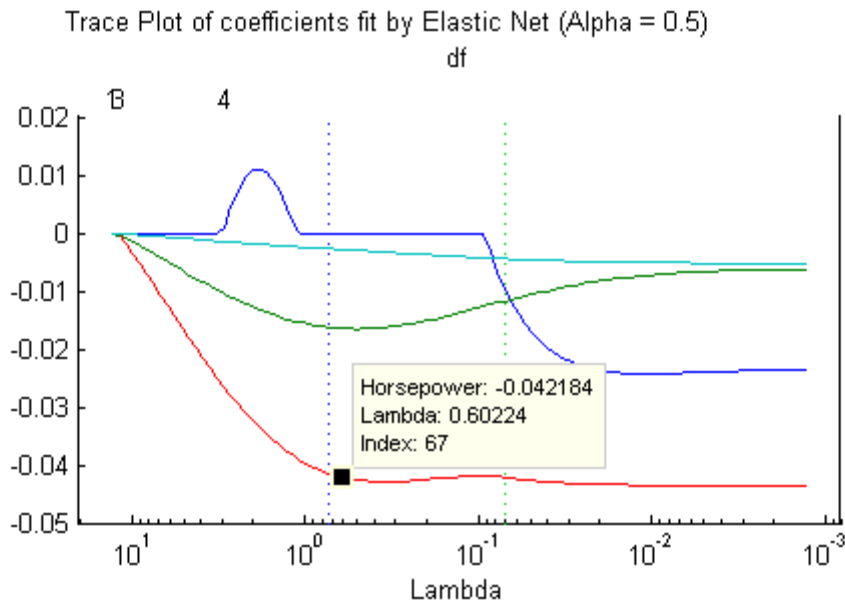
1.0000	-0.5438	-0.6892	-0.4168
-0.5438	1.0000	0.8973	0.9330
-0.6892	0.8973	1.0000	0.8645
-0.4168	0.9330	0.8645	1.0000

- 6** Because some predictors are highly correlated, perform elastic net fitting. Use Alpha = 0.5:

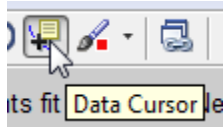
```
[ba fitinfoa] = lasso(X,MPG,'CV',10,'Alpha',.5);
```

- 7** Plot the result. Name each predictor so you can tell which curve is which:

```
pnames = {'Acceleration','Displacement',...
          'Horsepower','Weight'};
lassoPlot(ba,fitinfoa,'PlotType','Lambda',...
          'XScale','log','PredictorNames',pnames);
```



When you activate the data cursor



and click the plot, you see the name of the predictor, the coefficient, the value of  $\text{Lambda}$ , and the index of that point, meaning the column in  $\mathbf{b}$  associated with that fit.

Here, the elastic net and lasso results are not very similar. Also, the elastic net plot reflects a notable qualitative property of the elastic net technique. The elastic net retains three nonzero coefficients as  $\text{Lambda}$  increases (toward the left of the plot), and these three coefficients reach 0 at about the same  $\text{Lambda}$  value. In contrast, the lasso plot shows two of the three coefficients becoming 0 at the same value of  $\text{Lambda}$ , while another coefficient remains nonzero for higher values of  $\text{Lambda}$ .

This behavior exemplifies a general pattern. In general, elastic net tends to retain or drop groups of highly correlated predictors as  $\text{Lambda}$  increases. In contrast, lasso tends to drop smaller groups, or even individual predictors.

## Wide Data via Lasso and Parallel Computing

Lasso and elastic net are especially well suited to *wide* data, meaning data with more predictors than observations. Obviously, there are redundant predictors in this type of data. Use lasso along with cross validation to identify important predictors.

Cross validation can be slow. If you have a Parallel Computing Toolbox license, speed the computation using parallel computing.

### 1 Load the spectra data:

```
load spectra
Description

Description =

== Spectral and octane data of gasoline ==
```

NIR spectra and octane numbers of 60 gasoline samples

NIR: NIR spectra, measured in 2 nm intervals from 900 nm to 1700 nm

octane: octane numbers

spectra: a dataset array containing variables for NIR and octane

Reference:

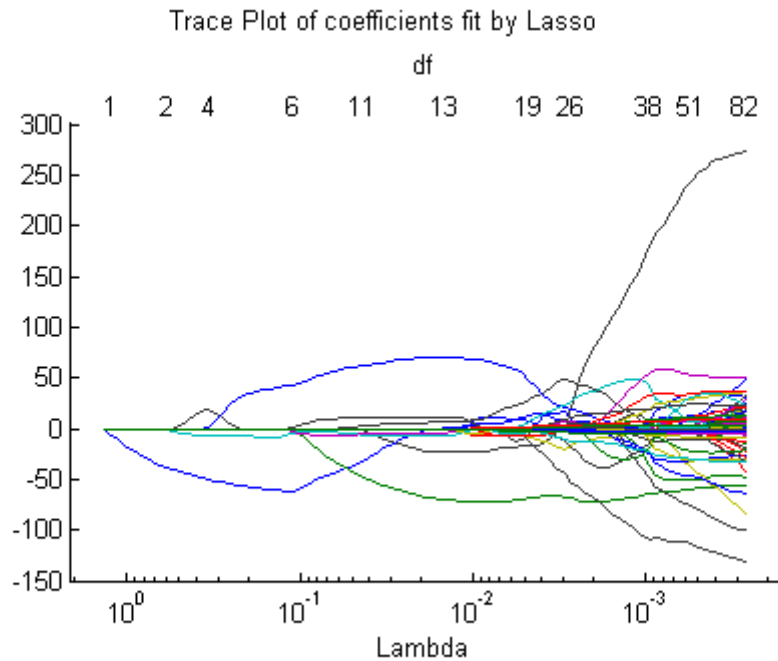
Kalivas, John H., "Two Data Sets of Near Infrared Spectra," *Chemometrics and Intelligent Laboratory Systems*, v.37 (1997) pp.255-259

**2** Compute the default lasso fit:

```
[b fitinfo] = lasso(NIR,octane);
```

**3** Plot the number of predictors in the fitted lasso regularization as a function of Lambda, using a logarithmic  $x$ -axis:

```
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');
```



- 4 It is difficult to tell which value of Lambda is appropriate. To determine a good value, try fitting with cross validation:

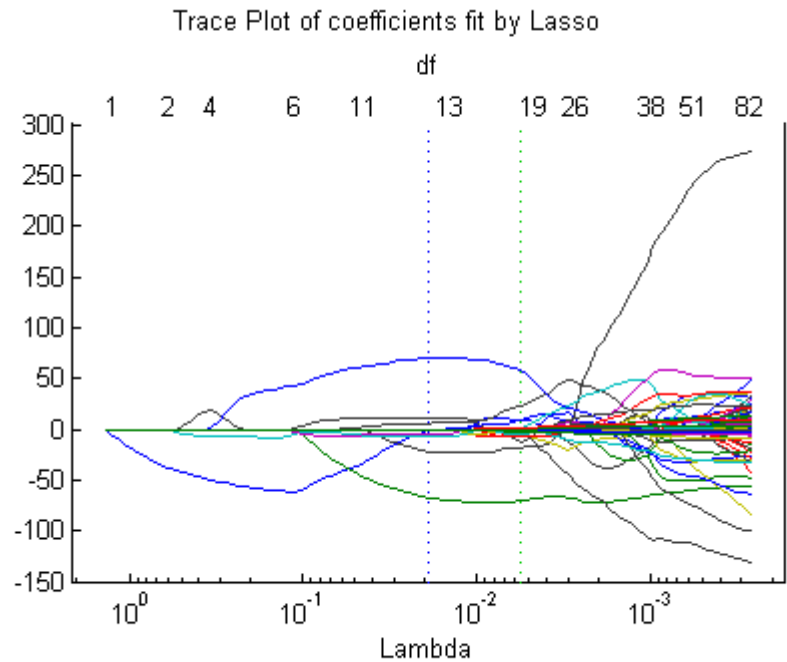
```
tic
[b fitinfo] = lasso(NIR,octane,'CV',10);
% A time-consuming operation
toc
```

Elapsed time is 226.876926 seconds.

- 5 Plot the result:

```
lassoPlot(b,fitinfo,'PlotType','Lambda','XScale','log');
```





You can see the suggested value of Lambda is over  $1e-2$ , and the Lambda with minimal MSE is under  $1e-2$ . These values are in the `fitinfo` structure:

```
fitinfo.LambdaMinMSE
```

```
ans =  
    0.0057
```

```
fitinfo.Lambda1SE
```

```
ans =  
    0.0190
```

- 6 Examine the quality of the fit for the suggested value of Lambda:

```
lambdaindex = fitinfo.Index1SE;  
fitinfo.MSE(lambdaindex)
```

```
ans =
    0.0532
```

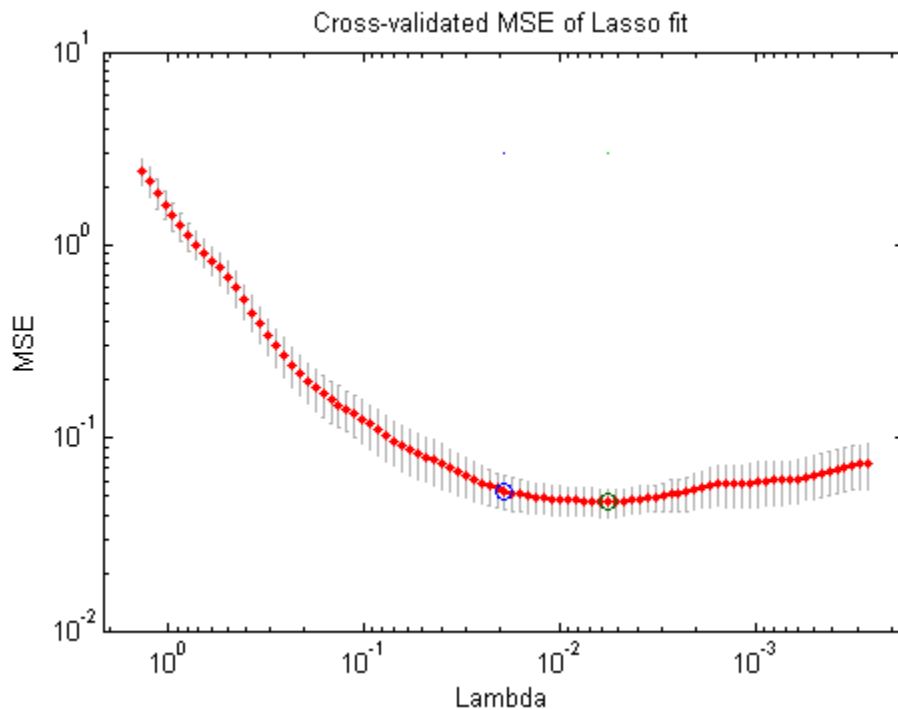
```
fitinfo.DF(lambdaindex)
```

```
ans =
    11
```

The fit uses just 11 of the 401 predictors, and achieves a cross-validated MSE of 0.0532.

**7** Examine the plot of cross-validated MSE:

```
lassoPlot(b,fitinfo,'PlotType','CV');
% Use a log scale for MSE to see small MSE values better
set(gca,'YScale','log');
```



As `Lambda` increases (toward the left), MSE increases rapidly. The coefficients are reduced too much and they do not adequately fit the responses.

As `Lambda` decreases, the models are larger (have more nonzero coefficients). The increasing MSE suggests that the models are overfitted.

The default set of `Lambda` values does not include values small enough to include all predictors. In this case, there does not appear to be a reason to look at smaller values. However, if you want smaller values than the default, use the `LambdaRatio` parameter, or supply a sequence of `Lambda` values using the `Lambda` parameter. For details, see the `lasso` reference page.

- 8 To compute the cross-validated lasso estimate faster, use parallel computing (available with a Parallel Computing Toolbox license):

```
parpool()
Starting parpool using the 'local' profile ... connected to 2 workers.

ans =

    Pool with properties:

        AttachedFiles: {0x1 cell}
           NumWorkers: 2
        IdleTimeout: 30
           Cluster: [1x1 parallel.cluster.Local]
    RequestQueue: [1x1 parallel.RequestQueue]
        SpmdEnabled: 1

opts = statset('UseParallel',true);
tic;
[b fitinfo] = lasso(NIR,octane,'CV',10,'Options',opts);
toc
Elapsed time is 114.712260 seconds.
```

Computing in parallel using two workers is faster on this problem.

## Lasso and Elastic Net Details

### Overview of Lasso and Elastic Net

Lasso is a regularization technique for performing linear regression. Lasso includes a penalty term that constrains the size of the estimated coefficients. Therefore, it resembles ridge regression. Lasso is a *shrinkage estimator*: it generates coefficient estimates that are biased to be small. Nevertheless, a lasso estimator can have smaller mean squared error than an ordinary least-squares estimator when you apply it to new data.

Unlike ridge regression, as the penalty term increases, lasso sets more coefficients to zero. This means that the lasso estimator is a smaller model, with fewer predictors. As such, lasso is an alternative to stepwise regression and other model selection and dimensionality reduction techniques.

Elastic net is a related technique. Elastic net is a hybrid of ridge regression and lasso regularization. Like lasso, elastic net can generate reduced models by generating zero-valued coefficients. Empirical studies have suggested that the elastic net technique can outperform lasso on data with highly correlated predictors.

### Definition of Lasso

The *lasso* technique solves this regularization problem. For a given value of  $\lambda$ , a nonnegative parameter, lasso solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda \sum_{j=1}^p |\beta_j| \right),$$

where

- $N$  is the number of observations.
- $y_i$  is the response at observation  $i$ .
- $x_i$  is data, a vector of  $p$  values at observation  $i$ .
- $\lambda$  is a positive regularization parameter corresponding to one value of Lambda.

- The parameters  $\beta_0$  and  $\beta$  are scalar and  $p$ -vector respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

### Definition of Elastic Net

The *elastic net* technique solves this regularization problem. For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right).$$

Elastic net is the same as lasso when  $\alpha = 1$ . As  $\alpha$  shrinks toward 0, elastic net approaches ridge regression. For other values of  $\alpha$ , the penalty term  $P_\alpha(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ .

### References

- [1] Tibshirani, R. *Regression shrinkage and selection via the lasso*. Journal of the Royal Statistical Society, Series B, Vol 58, No. 1, pp. 267–288, 1996.
- [2] Zou, H. and T. Hastie. *Regularization and variable selection via the elastic net*. Journal of the Royal Statistical Society, Series B, Vol. 67, No. 2, pp. 301–320, 2005.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. *Regularization paths for generalized linear models via coordinate descent*. Journal of Statistical Software, Vol 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>

[4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition. Springer, New York, 2008.

## Partial Least Squares

In this section...
“Introduction to Partial Least Squares” on page 9-147
“Partial Least Squares” on page 9-148

### Introduction to Partial Least Squares

*Partial least-squares (PLS)* regression is a technique used with data that contain correlated predictor variables. This technique constructs new predictor variables, known as *components*, as linear combinations of the original predictor variables. PLS constructs these components while considering the observed response values, leading to a parsimonious model with reliable predictive power.

The technique is something of a cross between multiple linear regression and principal component analysis:

- Multiple linear regression finds a combination of the predictors that best fit a response.
- Principal component analysis finds combinations of the predictors with large variance, reducing correlations. The technique makes no use of response values.
- PLS finds combinations of the predictors that have a large covariance with the response values.

PLS therefore combines information about the variances of both the predictors and the responses, while also considering the correlations among them.

PLS shares characteristics with other regression and feature transformation techniques. It is similar to ridge regression in that it is used in situations with correlated predictors. It is similar to stepwise regression (or more general feature selection techniques) in that it can be used to select a smaller set of model terms. PLS differs from these methods, however, by transforming the original predictor space into the new component space.

The Statistics Toolbox function `pplsregress` carries out PLS regression.

## Partial Least Squares

For example, consider the data on biochemical oxygen demand in `moore.mat`, padded with noisy versions of the predictors to introduce correlations:

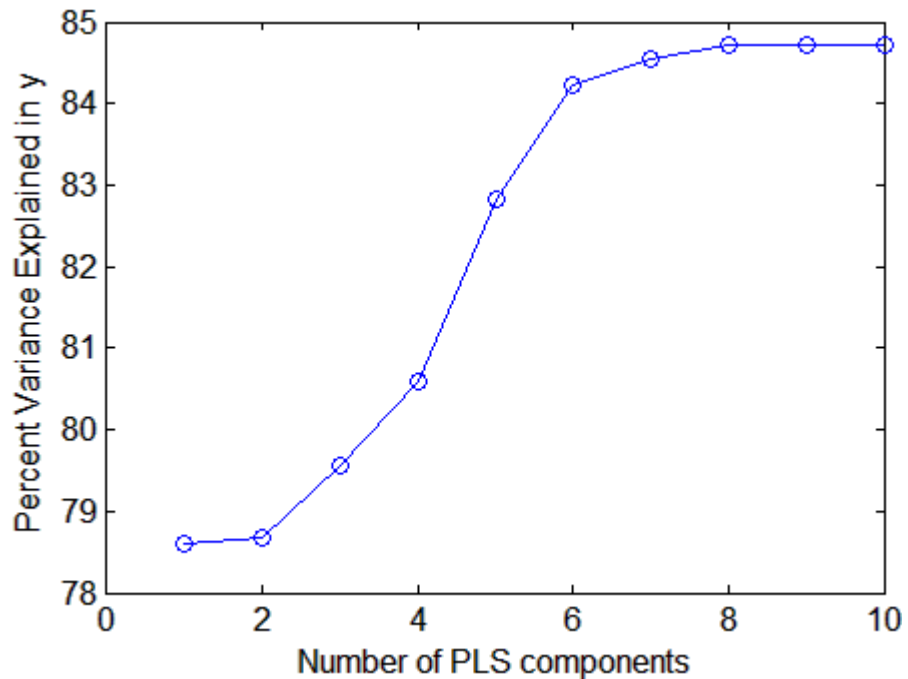
```
load moore
y = moore(:,6);           % Response
X0 = moore(:,1:5);       % Original predictors
X1 = X0+10*randn(size(X0)); % Correlated predictors
X = [X0,X1];
```

Use `plsregress` to perform PLS regression with the same number of components as predictors, then plot the percentage variance explained in the response as a function of the number of components:

```
[XL,y1,XS,YS,beta,PCTVAR] = plsregress(X,y,10);

plot(1:10,cumsum(100*PCTVAR(2,:)),'-bo');
xlabel('Number of PLS components');
ylabel('Percent Variance Explained in y');
```

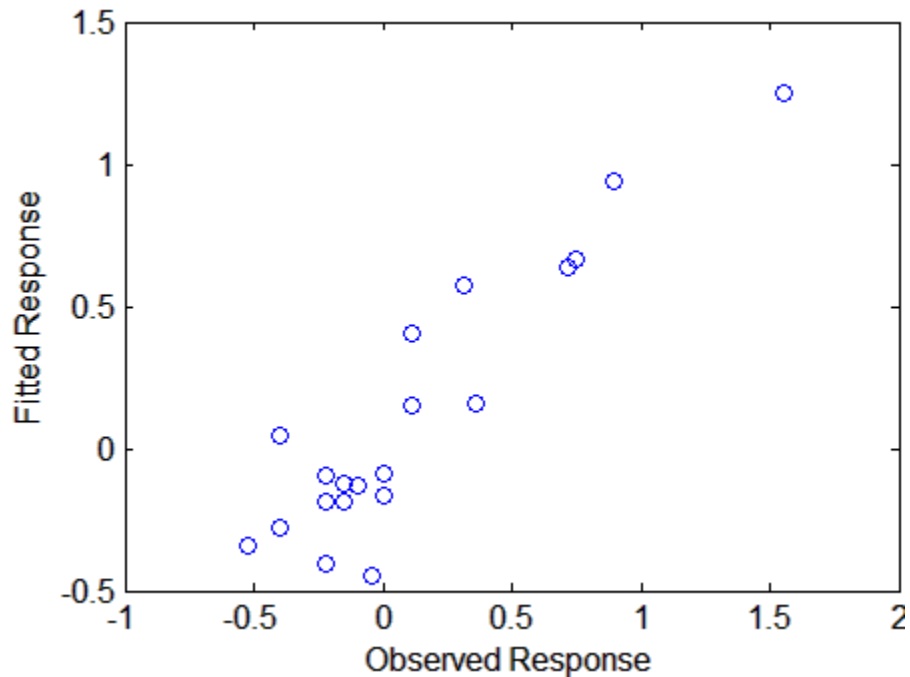




Choosing the number of components in a PLS model is a critical step. The plot gives a rough indication, showing nearly 80% of the variance in  $y$  explained by the first component, with as many as five additional components making significant contributions.

The following computes the six-component model:

```
[XL,y1,XS,YS,beta,PCTVAR,MSE,stats] = plsregress(X,y,6);  
yfit = [ones(size(X,1),1) X]*beta;  
  
plot(y,yfit,'o')
```

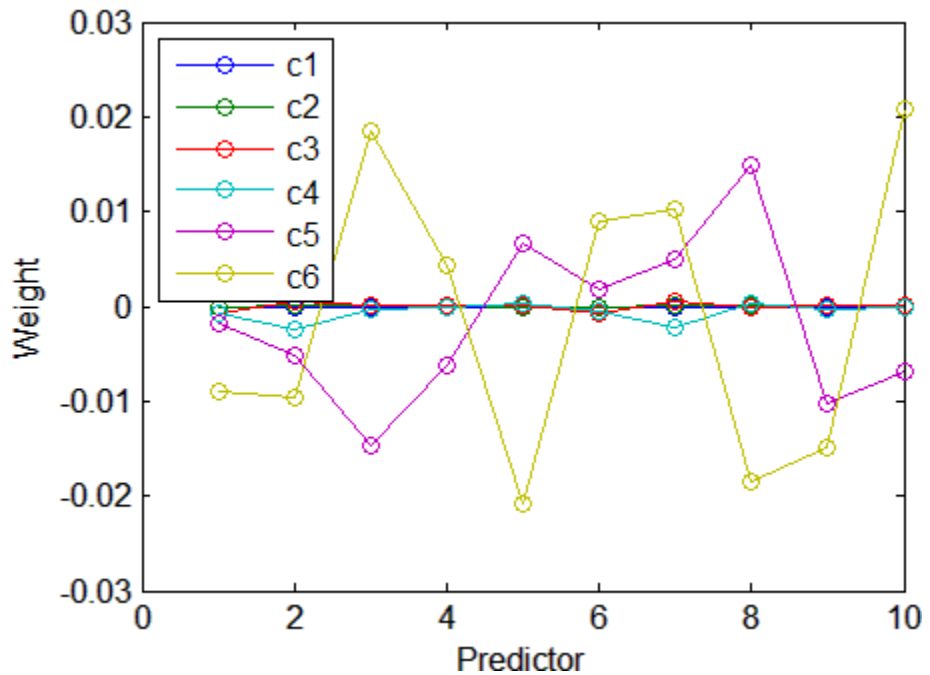


The scatter shows a reasonable correlation between fitted and observed responses, and this is confirmed by the  $R^2$  statistic:

```
TSS = sum((y-mean(y)).^2);  
RSS = sum((y-yfit).^2);  
Rsquared = 1 - RSS/TSS  
Rsquared =  
    0.8421
```

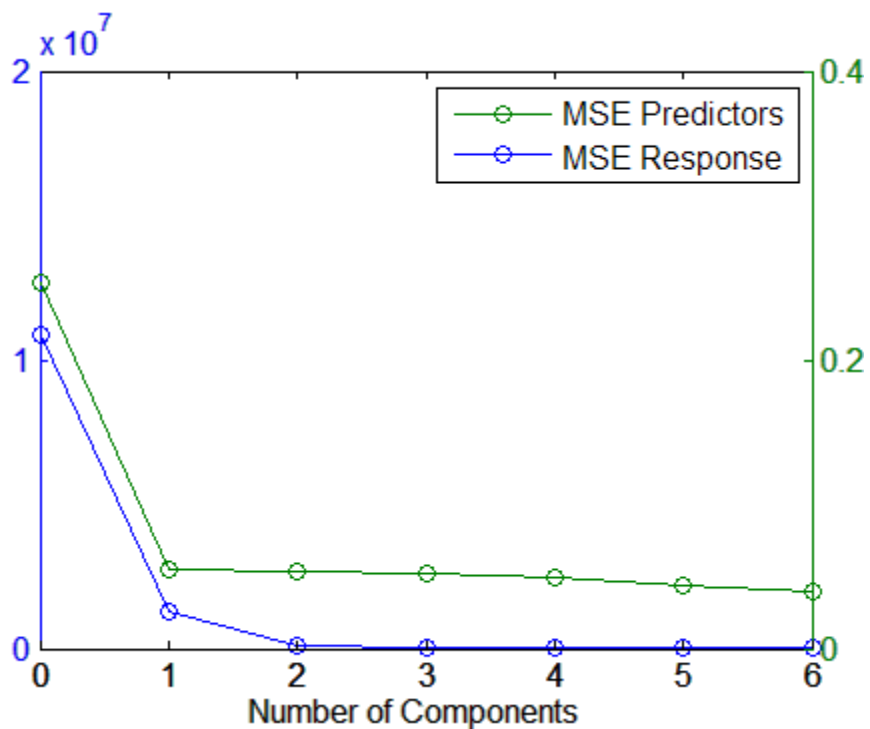
A plot of the weights of the ten predictors in each of the six components shows that two of the components (the last two computed) explain the majority of the variance in  $X$ :

```
plot(1:10,stats.W,'o-');  
legend({'c1','c2','c3','c4','c5','c6'},'Location','NW')  
xlabel('Predictor');  
ylabel('Weight');
```



A plot of the mean-squared errors suggests that as few as two components may provide an adequate model:

```
[axes,h1,h2] = plotyy(0:6,MSE(1,:),0:6,MSE(2,:));
set(h1,'Marker','o')
set(h2,'Marker','o')
legend('MSE Predictors','MSE Response')
xlabel('Number of Components')
```



The calculation of mean-squared errors by `plsregress` is controlled by optional parameter name/value pairs specifying cross-validation type and the number of Monte Carlo repetitions.

## Linear Mixed-Effects Models

Linear mixed-effects models are extensions of linear regression models for data that are collected and summarized in groups. These models describe the relationship between a response variable and independent variables, with coefficients that can vary with respect to one or more grouping variables. A mixed-effects model consists of two parts, fixed effects and random effects. Fixed-effects terms are usually the conventional linear regression part, and the random effects are associated with individual experimental units drawn at random from a population. The random effects have prior distributions whereas fixed effects do not. Mixed-effects models can represent the covariance structure related to the grouping of data by associating the common random effects to observations that have the same level of a grouping variable. The standard form of a linear mixed-effects model is

$$y = \underbrace{X\beta}_{\text{fixed}} + \underbrace{Zb}_{\text{random}} + \underbrace{\varepsilon}_{\text{error}},$$

where

- $y$  is the  $n$ -by-1 response vector, and  $n$  is the number of observations.
- $X$  is an  $n$ -by- $p$  fixed-effects design matrix.
- $\beta$  is a  $p$ -by-1 fixed-effects vector.
- $Z$  is an  $n$ -by- $q$  random-effects design matrix.
- $b$  is a  $q$ -by-1 random-effects vector.
- $\varepsilon$  is the  $n$ -by-1 observation error vector

The assumptions for the linear mixed-effects model are:

- Random-effects vector,  $b$ , and the error vector,  $\varepsilon$ , have the following prior distributions:

$$b \sim N(0, \sigma^2 D(\theta)),$$

$$\varepsilon \sim N(0, \sigma^2 I),$$

where  $D$  is a symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $I$  is an  $n$ -by- $n$  identity matrix, and  $\sigma^2$  is the error variance.

- Random-effects vector,  $b$ , and the error vector,  $\varepsilon$ , are independent from each other.

Mixed-effects models are also called *multilevel models* or *hierarchical models* depending on the context. Mixed-effects models is a more general term than the latter two. Mixed-effects models might include factors that are not necessarily multilevel or hierarchical, for example crossed factors. That is why mixed-effects is the terminology preferred here. Sometimes mixed-effects models are expressed as multilevel regression models (first level and grouping level models) that are fit simultaneously. For example, a varying or random intercept model, with one continuous predictor variable  $x$  and one grouping variable with  $M$  levels, can be expressed as

$$y_{im} = \beta_{0m} + \beta_1 x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \quad \varepsilon_{im} \sim N(0, \sigma^2),$$

$$\beta_{0m} = \beta_{00} + b_{0m}, \quad b_{0m} \sim N(0, \sigma_0^2),$$

where  $y_{im}$  corresponds to data for observation  $i$  and group  $m$ ,  $n$  is the total number of observations, and  $b_{0m}$  and  $\varepsilon_{im}$  are independent of each other. After substituting the group-level parameters in the first-level model, the model for the response vector becomes

$$y_{im} = \underbrace{\beta_{00} + \beta_1 x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m}}_{\text{random effects}} + \varepsilon_{im}.$$

A random intercept and slope model with one continuous predictor variable  $x$ , where both the intercept and slope vary independently by a grouping variable with  $M$  levels is

$$y_{im} = \beta_{0m} + \beta_{1m} x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \quad \varepsilon_{im} \sim N(0, \sigma^2),$$

$$\beta_{0m} = \beta_{00} + b_{0m}, \quad b_{0m} \sim N(0, \sigma_0^2),$$

$$\beta_{1m} = \beta_{10} + b_{1m}, \quad b_{1m} \sim N(0, \sigma_1^2),$$

or

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_0^2 & 0 \\ 0 & \sigma_1^2 \end{pmatrix} \right).$$

You might also have correlated random effects. In general, for a model with a random intercept and slope, the distribution of the random effects is

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \sigma^2 D(\theta) \right),$$

where  $D$  is a 2-by-2 symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ .

After substituting the group-level parameters in the first-level model, the model for the response vector is

$$y_{im} = \underbrace{\beta_{00} + \beta_{10}x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m} + b_{1m}x_{im}}_{\text{random effects}} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M.$$

If you express the group-level variable,  $x_{im}$ , in the random-effects term by  $z_{im}$ , this model is

$$y_{im} = \underbrace{\beta_{00} + \beta_{10}x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m} + b_{1m}z_{im}}_{\text{random effects}} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M.$$

In this case, the same terms appear in both the fixed-effects design matrix and random-effects design matrix.  $z$  includes as many columns as the levels of the grouping variable. Each column of  $z$  is formed by taking parts of  $x$  that correspond to a level of the grouping variable.

It is also possible to explain more of the group-level variations by adding more group-level predictor variables. A random-intercept and random-slope model with one continuous predictor variable  $x$ , where both the intercept and slope vary independently by a grouping variable with  $M$  levels, and one group-level predictor variable  $v_m$  is

$$\begin{aligned}
 y_{im} &= \beta_{0im} + \beta_{1im}x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \quad \varepsilon_{im} \sim N(0, \sigma^2), \\
 \beta_{0im} &= \beta_{00} + \beta_{01}v_{im} + b_{0m}, \quad b_{0m} \sim N(0, \sigma_0^2), \\
 \beta_{1im} &= \beta_{10} + \beta_{11}v_{im} + b_{1m}, \quad b_{1m} \sim N(0, \sigma_1^2).
 \end{aligned}$$

This model results in main effects of the group-level predictor and an interaction term between the first-level and group-level predictor variables in the model for the response variable as

$$\begin{aligned}
 y_{im} &= \beta_{00} + \beta_{01}v_{im} + b_{0m} + (\beta_{10} + \beta_{11}v_{im} + b_{1m})x_{im} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M, \\
 &= \underbrace{\beta_{00} + \beta_{10}x_{im} + \beta_{01}v_{im} + \beta_{11}v_{im}x_{im}}_{\text{fixed effects}} + \underbrace{b_{0m} + b_{1m}x_{im}}_{\text{random effects}} + \varepsilon_{im}.
 \end{aligned}$$

The term  $\beta_{11}v_{im}x_{im}$  is often called a cross-level interaction in many textbooks on multilevel models. The model for the response variable  $y$  can be expressed as

$$y_{im} = \begin{bmatrix} 1 & x_{1im} & v_{im} & v_{im}x_{1im} \end{bmatrix} \begin{bmatrix} \beta_{00} \\ \beta_{10} \\ \beta_{01} \\ \beta_{11} \end{bmatrix} + \begin{bmatrix} 1 & x_{1im} \end{bmatrix} \begin{bmatrix} b_{0m} \\ b_{1m} \end{bmatrix} + \varepsilon_{im}, \quad i = 1, 2, \dots, n, \quad m = 1, 2, \dots, M,$$

which corresponds to the standard form given earlier,

$$y = X\beta + Zb + \varepsilon.$$

In general, if there are  $R$  grouping variables, and  $m(r, i)$  shows the level of grouping variable  $r$ , for observation  $i$ , then the model for the response variable for observation  $i$  is

$$y_i = x_i^T \beta + \sum_{r=1}^R z_{ir} b_{m(r, i)}^{(r)} + \varepsilon_i, \quad i = 1, 2, \dots, n,$$



where  $\beta$  is a  $p$ -by-1 fixed-effects vector,  $b^{(r)}_{m(r,i)}$  is a  $q(r)$ -by-1 random-effects vector for the  $r$ th grouping variable and level  $m(r,i)$ , and  $\varepsilon_i$  is a 1-by-1 error term for observation  $i$ .

## References

- [1] Pinheiro, J. C., and D. M. Bates. *Mixed-Effects Models in S and S-PLUS*. Statistics and Computing Series, Springer, 2004.
- [2] Hariharan, S. and J. H. Rogers. “Estimation Procedures for Hierarchical Linear Models.” *Multilevel Modeling of Educational Data* (A. A. Connell and D. B. McCoach, eds.). Charlotte, NC: Information Age Publishing, Inc., 2008.
- [3] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc., 2002
- [4] Snijders, T. and R. Bosker. *Multilevel Analysis*. Thousand Oaks, CA: Sage Publications, 1999.
- [5] Gelman, A. and J. Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. New York, NY: Cambridge University Press, 2007.

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

## Concepts

- “Prepare Data for Linear Mixed-Effects Models” on page 9-158

## Prepare Data for Linear Mixed-Effects Models

### In this section...

“Dataset Arrays” on page 9-158

“Design Matrices” on page 9-160

“Relation of Matrix Form to Dataset Arrays” on page 9-162

### Dataset Arrays

To fit a linear-mixed effects model, you must store your data in a dataset array. In your dataset array, you must have a column for each variable including the response variable. More specifically, the dataset array, say  $ds$ , must contain the following:

- A response variable  $y$
- Predictive variables  $X_j$  which can be continuous or grouping variables
- Grouping variables  $g_1, g_2, \dots, g_R$ ,

where the grouping variables in  $X_j$  and  $g_r$  can be categorical, logical, character arrays, or a cell arrays of strings,  $r = 1, 2, \dots, R$ .

You must organize your data so that each row represents an observation. And each row should contain the value of variables and the levels of grouping variables corresponding to that observation. For example, if you have data from an experiment with four treatment options, on five different types of individuals chosen randomly from a population of individuals (blocks), the dataset array must look like this.

Block	Treatment	Response
1	1	y11
1	2	y12
1	3	y13
1	4	y14
...	...	...

Block	Treatment	Response
5	1	y51
5	2	y52
5	3	y53
5	4	y54

Now, consider a split-plot experiment, where the effect of four different types of fertilizers on the yield of tomato plants is studied. The soil where the tomato plants are planted is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five types of tomato plants, (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. Then, the tomato plants in the plots are divided into subplots, where each subplot is treated by one of the four fertilizers. The data from this experiment looks like:

Soil	Tomato	Fertilizer	Yield
'Sandy'	'Plum'	1	104
'Sandy'	'Plum'	2	136
'Sandy'	'Plum'	3	158
'Sandy'	'Plum'	4	174
'Sandy'	'Cherry'	1	57
'Sandy'	'Cherry'	2	86
...	...	...	...
'Sandy'	'Vine'	3	99
'Sandy'	'Vine'	4	117
'Silty'	'Plum'	1	120
'Silty'	'Plum'	2	115
...	...	...	...
'Loamy'	'Vine'	3	111
'Loamy'	'Vine'	4	105

You must specify the model you want to fit using the formula input argument to `fitlme`.

In general, a formula for model specification is a string of the term 'y ~ terms'. For linear mixed-effects models, this formula is in the form 'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)', where `fixed` contains the fixed-effects terms and `random1`, ..., `randomR` contain the random-effects terms. For example, for the previous fertilizer experiment, consider the following mixed-effects model

$$y_{imjk} = \beta_0 + \sum_{m=2}^4 \beta_{1m} I[F]_{im} + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + b_{0k} S_k + b_{0jk} (S * T)_{jk} + \varepsilon_{imjk},$$

where  $i = 1, 2, \dots, 60$ , the index  $m$  corresponds to the fertilizer types,  $j$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $I[F]_{im}$  is the dummy variable representing level  $m$  of the fertilizer. Similarly,  $I[T]_{ij}$  is the dummy variable representing the level  $j$  of the tomato type.

You can fit this model using the formula 'Yield ~ 1 + Fertilizer + Tomato + (1|Soil)+(1|Soil:Tomato)'

For detailed information on how to specify your model using formula, see "Relationship Between Formula and Design Matrices" on page 9-164.

## Design Matrices

If you cannot easily describe your model using a formula, you can create design matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X,y,Z,G)`. You must create your design matrices as follows.

Fixed-effects and random-effects design matrices X and Z:

- Enter a column of 1s for the intercept using `ones(n,1)`, where  $n$  is the total number of observations.
- If X1 is a continuous variable, then enter X1 as it is in a separate column.
- If X1 is a categorical variable with  $m$  levels, then there must be  $m - 1$  dummy variables for  $m - 1$  levels of X1 in X.

For example, consider an experiment where you want to study the impact of quality of raw materials from four different providers on the productivity of a production line. If you fit a linear mixed-effects model with intercept and provider as the fixed-effects terms, intercept is the random-effects term, and you use reference contrasts coding, then you must construct your fixed- and random-effects design matrices as follows.

```
D = dummyvar(provider); % Create dummy variables
X = [ones(n,1) D(:,2) D(:,3) D(:,4)];
Z = [ones(n,1)];
```

Because reference contrast coding uses the first provider as the reference, and the model has an intercept, you must use the dummy variables for only the last three providers.

- If there is an interaction term of predictor variables X1 and X2, then you must enter a column that you form by elementwise product of the vectors X1 and X2.

For example, if you want to fit a model, where there is an intercept, a continuous treatment factor, a continuous time factor, and their interaction as the fixed-effects in a longitudinal study, and time is the random-effects term, then your fixed- and random-effects design matrices should look like

```
X = [ones(n,1), treatment, time, treatment.*time];
y = response;
Z = [time];
```

Grouping variables G:

There is one column for each grouping variable and a column of elementwise product of the grouping variables in case of a nesting.

For example, if you want to group plots (plot) within blocks (block), then you must add a column of elementwise product of plot by block. More specifically, if you want to fit a model where there is intercept and a continuous treatment factor as the fixed-effects in a split-block experiment, and the intercept and treatment are grouped by the plots nested within blocks, then the design matrices should look like this.

```
X = [ones(n,1), treatment];
y = response;
```

```
Z = [ones(n,1),treatment];
G = [block.*plot];
```

Suppose in the earlier quality of raw materials example, the raw materials arrive in bulks, and the bulks are nested within providers. If you want to fit a linear mixed-effects model, where intercept is grouped by the bulks within providers, then your design matrices should look like this.

```
D = dummyvar(provider);
X = [ones(n,1) D(:,2) D(:,3) D(:,4)];
y = response;
Z = ones(n,1);
G = [provider.*bulks];
```

In the earlier longitudinal study example, if you want to add random effects for intercept and time grouped by subjects that participated in the study, then your design matrices should look like

```
X = [ones(n,1),treatment,time, treatment.*time];
y = response;
Z = [ones(n,1),time];
G = subject;
```

## Relation of Matrix Form to Dataset Arrays

`fitlme(ds,formula)` and `fitlmematrix(X,y,Z,G)` are equivalent in functionality, such that

- $y$  is the  $n$ -by-1 response vector.
- $X$  is an  $n$ -by- $p$  fixed-effects design matrix. `fitlme` constructs this from the expression `fixed` in `formula`.
- $Z$  is an  $R$ -by-1 cell array with  $Z\{r\}$  being an  $n$ -by- $q(r)$  random-effects design matrix constructed from the  $r$ th expression in `random` in `formula`,  $r = 1, 2, \dots, R$ .
- $G$  is an  $R$ -by-1 cell array with  $G\{r\}$  being an  $n$ -by-1 grouping variable,  $g_r$ , in `formula` with  $M(r)$  levels or groups.

For example, if `ds` is a dataset array containing the response variable `y`, the continuous variables `X1` and `X2`, and the grouping variable `g`, then to fit a linear mixed-effects model that corresponds to the formula expression `'y ~`

$X1 + X2 + (X1 * X2 | g)$  using `fitlmematrix(X,y,Z,G)` the input arguments must correspond to the following:

```
y = ds.y
X = [ones(n,1), ds.X1, ds.X2]
Z = [ones(n,1), ds.X1, ds.X2, ds.X1.*ds.X2]
G = ds.g
```

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

## Concepts

- “Linear Mixed-Effects Models” on page 9-153
- “Relationship Between Formula and Design Matrices” on page 9-164

## Relationship Between Formula and Design Matrices

### In this section...

“Formula” on page 9-164

“Design Matrices for Fixed and Random Effects” on page 9-166

“Grouping Variables” on page 9-169

### Formula

In general, a formula for model specification is a string of the form `'y ~ terms'`. For the linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` and `random` contain the fixed-effects and the random-effects terms. Suppose a dataset array `ds` contains the following:

- A response variable, `y`
- Predictor variables, `Xj`, which can be continuous or grouping variables
- Grouping variables, `g1, g2, ..., gR`,

where the grouping variables in `Xj` and `gr` can be categorical, logical, character arrays, or cell arrays of strings.

Then, in a formula of the form, `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`, the term `fixed` corresponds to a specification of the fixed-effects design matrix `X`, `random1` is a specification of the random-effects design matrix `Z1` corresponding to grouping variable `g1`, and similarly `randomR` is a specification of the random-effects design matrix `ZR` corresponding to grouping variable `gR`. You can express the `fixed` and `random` terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.



Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1.*X2$ (elementwise multiplication of $X1$ and $X2$ )
$X1:X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1:X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1:X2:X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`. Here are some examples for linear mixed-effects model specification.

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, $X1$ and $X2$ . This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	Fixed effects for $X1$ and $X2$ . The implicit intercept term is suppressed by including <code>-1</code> .
'y ~ 1 + (1   g1)'	Intercept plus random effect for each level of the grouping variable $g1$ .
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1)'	Random intercept and slope, with possible correlation between them.

Formula	Description
'y ~ X1 + (1   g1) + (-1 + X1   g1)'	Independent random intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

### Design Matrices for Fixed and Random Effects

fitlme converts the expressions in the fixed and random parts (not grouping variables) of a formula into design matrices as follows:

- Each term in a formula adds one or more columns to the corresponding design matrix.
- A term containing a single continuous variable adds one column to the design matrix.
- A fixed term containing a categorical variable X with  $k$  levels adds  $(k - 1)$  dummy variables to the design matrix.

For example, if the variable `Supplier` represents three different suppliers a manufacturer receives parts from, i.e. a categorical variable with three levels, and out of six batches of parts, the first two batches come from supplier 1 (level 1), the second two batches come from supplier 2 (level 2), and the last two batches come from supplier 3 (level 3), such as

Supplier =

1  
1  
2  
2  
3  
3

Then, adding `Supplier` to the formula as a fixed-effects or random-effects term adds the following two dummy variables to the corresponding design matrix, using the 'reference' contrast:

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

For more details on dummy variables, see “Dummy Indicator Variables” on page 2-55. For other contrast options, see the 'DummyVarCoding' name-value pair argument of `fitlme`.

- If  $X1$  and  $X2$  are continuous variables, the product term  $X1:X2$  adds one column obtained by elementwise multiplication of  $X1$  and  $X2$  to the design matrix.
- If  $X1$  is continuous and  $X2$  is categorical with  $k$  levels, the product term  $X1:X2$  multiplies elementwise  $X1$  with the  $(k - 1)$  dummy variables representing  $X2$ , and adds these  $(k - 1)$  columns to the design matrix.

For example, if `Drug` is the amount of a drug given to patients, a continuous treatment, and `Time` is three distinct points in time when the health measures are taken, a categorical variable with three levels, and out of nine observations, the first three are observed at time point 1, the second three are observed at time point 2, and the last three are observed at time point 3 so that

```
[Drug Time] =
0.1000 1.0000
0.2000 1.0000
0.5000 2.0000
0.6000 2.0000
0.3000 3.0000
0.8000 3.0000
```

Then, the product term `Drug:Time` adds the following two variables to the design matrix:

```

      0      0
      0      0
0.5000      0
0.6000      0
      0    0.3000
      0    0.8000
    
```

- If  $X_1$  and  $X_2$  are categorical variables with  $k$  and  $m$  levels respectively, the product term  $X_1:X_2$  adds  $(k - 1)*(m - 1)$  dummy variables to the design matrix formed by taking the elementwise product of each dummy variable representing  $X_1$  with each dummy variable representing  $X_2$ .

For example, in an experiment to determine the impact of the type of corn and the popping method on the yield, suppose there are three types of Corn and two types of Method as follows:

```

1   oil
1   oil
1   air
1   air
2   oil
2   oil
2   air
2   air
3   oil
3   oil
3   air
3   air
    
```

Then, the interaction term  $\text{Corn:Method}$  adds the following to the design matrix:

```

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

$$\begin{array}{cc} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{array}$$

- The term  $X1*X2$  adds the necessary number of columns for  $X1$ ,  $X2$ , and  $X1:X2$  to the design matrix.
- The term  $X1^2$  adds the necessary number of columns for  $X1$  and  $X1:X1$  to the design matrix.
- The symbol 1 (one) in the formula stands for a column of all 1s. By default a column of 1s is included in the design matrix. To exclude a column of ones from the design matrix, you must explicitly specify `1` as a term in the expression.

## Grouping Variables

`fitlme` handles the grouping variables in the `(. | group)` part of a formula as follows:

- If a grouping variable has  $k$  levels, then  $k$  dummy variables represent this grouping.

For example, suppose `District` is a categorical grouping variable with three levels, showing the three types of districts, and out of six schools, the first two are in district 1, the second two are in district 2, and the last two are in district 3, so that

District =

$$\begin{array}{c} 1 \\ 1 \\ 2 \\ 2 \\ 3 \\ 3 \end{array}$$

Then, the dummy variables that represent this grouping are:

$$\begin{array}{ccc} 1 & 0 & 0 \end{array}$$

1	0	0
0	1	0
0	1	0
0	0	1
0	0	1

- If  $X_1$  is a continuous random-effects variable and  $X_2$  is a grouping variable with  $k$  levels, then the random term  $(X_1 \quad 1 | X_2)$  multiplies elementwise  $X_1$  with the  $k$  dummy variables representing  $X_2$  and adds these  $k$  columns to the random-effects design matrix.

For example, suppose `Score` is a continuous variable showing the scores of students from a math exam in a school, and `Class` is a categorical variable with three levels, showing the three different classes in a school. Also, suppose out of nine observations first three correspond to the scores of students in the first class, the second three correspond to scores of students in the second class, and the last three correspond to the scores of students in the third class, such as

[Score Class] =

78.0000	1.0000
68.0000	1.0000
81.0000	2.0000
53.0000	2.0000
85.0000	3.0000
72.0000	3.0000

Then, the random term  $(\text{Score} \quad 1 | \text{Class})$  adds the following three columns to the random-effects design matrix:

78.0000	0	0
68.0000	0	0
0	81.0000	0
0	53.0000	0
0	0	85.0000
0	0	72.0000

- If  $X_1$  is a continuous predictor variable and  $X_2$  and  $X_3$  are grouping variables with  $k$  and  $m$  levels respectively, the term  $(X_1 | X_2 : X_3)$  represents this grouping of  $X_1$  with  $k*m$  dummy variables formed by taking the

elementwise product of each dummy variable representing X2 with each dummy variable representing X3.

For example, suppose `Treatment` is a continuous predictor variable, and there are three levels of `Block` and two levels of `Plot` nested within the block as follows:

0.1000	1	a
0.2000	1	b
0.5000	2	a
0.6000	2	b
0.3000	3	a
0.8000	3	b

Then, the random term (`Treatment`  $\times$  `1|Block:Plot`) adds the following to the random-effects design matrix:

0.1000	0	0	0	0	0
0	0.2000	0	0	0	0
0	0	0.5000	0	0	0
0	0	0	0.6000	0	0
0	0	0	0	0.3000	0
0	0	0	0	0	0.8000

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

## Concepts

- “Prepare Data for Linear Mixed-Effects Models” on page 9-158

## Estimating Parameters in Linear Mixed-Effects Models

### In this section...

“Maximum Likelihood (ML)” on page 9-173

“Restricted Maximum Likelihood (REML)” on page 9-174

A linear mixed-effects model is of the form

$$y = \underbrace{X\beta}_{\text{fixed}} + \underbrace{Zb}_{\text{random}} + \underbrace{\varepsilon}_{\text{error}},$$

where

- $y$  is the  $n$ -by-1 response vector, and  $n$  is the number of observations.
- $X$  is an  $n$ -by- $p$  fixed-effects design matrix.
- $\beta$  is a  $p$ -by-1 fixed-effects vector.
- $Z$  is an  $n$ -by- $q$  random-effects design matrix.
- $b$  is a  $q$ -by-1 random-effects vector.
- $\varepsilon$  is the  $n$ -by-1 observation error vector.

The random-effects vector,  $b$ , and the error vector,  $\varepsilon$ , are assumed to have the following independent prior distributions:

$$b \sim N(0, \sigma^2 D(\theta)),$$

$$\varepsilon \sim N(0, \sigma^2 I),$$

where  $D$  is a symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $I$  is an  $n$ -by- $n$  identity matrix, and  $\sigma^2$  is the error variance.

In this model, the parameters to estimate are the fixed-effects coefficients  $\beta$ , and the variance components  $\theta$  and  $\varepsilon$ . The two most commonly used approaches to parameter estimation in linear mixed-effects models are maximum likelihood and restricted maximum likelihood methods.



## Maximum Likelihood (ML)

The maximum likelihood estimation includes both regression coefficients and the variance components, that is, both fixed-effects and random-effects terms in the likelihood function.

For a linear mixed-effects model defined above, the conditional response of the response variable  $y$  given  $\beta$ ,  $b$ ,  $\theta$ , and  $\sigma^2$  is

$$y | b, \beta, \theta, \sigma^2 \sim N(X\beta + Zb, \sigma^2 I_n).$$

The likelihood of  $y$  given  $\beta$ ,  $\theta$ , and  $\sigma^2$  is

$$P(y | \beta, \theta, \sigma^2) = \int P(y | b, \beta, \theta, \sigma^2) P(b | \theta, \sigma^2) db,$$

where

$$P(b | \theta, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{q/2}} \frac{1}{|D(\theta)|^{1/2}} \exp\left\{-\frac{1}{2\sigma^2} b^T D^{-1} b\right\} \quad \text{and}$$

$$P(y | b, \beta, \theta, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{n/2}} \exp\left\{-\frac{1}{2\sigma^2} (y - X\beta - Zb)^T (y - X\beta - Zb)\right\}.$$

Suppose  $\Lambda(\theta)$  is the lower triangular Cholesky factor of  $D(\theta)$  and  $\Delta(\theta)$  is the inverse of  $\Lambda(\theta)$ . Then,

$$D(\theta)^{-1} = \Delta(\theta)^T \Delta(\theta).$$

Define

$$r^2(\beta, b, \theta) = b^T \Delta(\theta)^T \Delta(\theta) b + (y - X\beta - Zb)^T (y - X\beta - Zb),$$

and suppose  $b^*$  is the value of  $b$  that satisfies

$$\left. \frac{\partial r^2(\beta, b, \theta)}{\partial b} \right|_{b^*} = 0$$

for given  $\beta$  and  $\theta$ . Then, the likelihood function is

$$P(y | \beta, \theta, \sigma^2) = (2\pi\sigma^2)^{-n/2} |D(\theta)|^{-1/2} \exp\left\{-\frac{1}{2\sigma^2} r^2(\beta, b^*(\beta), \theta)\right\} \frac{1}{|\Delta^T \Delta + Z^T Z|^{1/2}}.$$

$P(y | \beta, \theta, \sigma^2)$  is first maximized with respect to  $\beta$  and  $\sigma^2$  for a given  $\theta$ .

Thus the optimized solutions  $\hat{\beta}(\theta)$  and  $\hat{\sigma}^2(\theta)$  are obtained as functions of  $\theta$ . Substituting these solutions into the likelihood function produces

$P(y | \beta(\theta), \theta, \sigma^2(\theta))$ . This expression is called a profiled likelihood where  $\beta$

and  $\sigma^2$  have been profiled out.  $P(y | \beta(\theta), \theta, \sigma^2(\theta))$  is a function of  $\theta$ , and the algorithm then optimizes it with respect to  $\theta$ . Once it finds the optimal estimate of  $\theta$ , the estimates of  $\beta$  and  $\sigma^2$  are given by  $\hat{\beta}(\theta)$ .

The ML method treats  $\beta$  as fixed but unknown quantities when the variance components are estimated, but does not take into account the degrees of freedom lost by estimating the fixed effects. This causes ML estimates to be biased with smaller variances. However, one advantage of ML over REML is that it is possible to compare two models in terms of their fixed- and random-effects terms. On the other hand, if you use REML to estimate the parameters, you can only compare two models, that are nested in their random-effects terms, with the same fixed-effects design.

### Restricted Maximum Likelihood (REML)

Restricted maximum likelihood estimation includes only the variance components, that is, the parameters that parameterize the random-effects terms in the linear mixed-effects model.  $\beta$  is estimated in a second step. Assuming a uniform improper prior distribution for  $\beta$  and integrating the likelihood  $P(y | \beta, \theta, \sigma^2)$  with respect to  $\beta$  results in the restricted likelihood  $P(y | \theta, \sigma^2)$ . That is,

$$P(y | \theta, \sigma^2) = \int P(y | \beta, \theta, \sigma^2) P(\beta) d\beta = \int P(y | \beta, \theta, \sigma^2) d\beta.$$

The algorithm first profiles out  $\hat{\sigma}_R^2$  and maximizes remaining objective function with respect to  $\theta$  to find  $\hat{\theta}_R$ . The restricted likelihood is then maximized with respect to  $\sigma^2$  to find  $\hat{\sigma}_R^2$ . Then, it estimates  $\beta$  by finding its expected value with respect to the posterior distribution

$$P(\beta | y, \theta_R, \sigma_R^2).$$

REML accounts for the degrees of freedom lost by estimating the fixed effects, and makes a less biased estimation of random effects variances. The estimates of  $\theta$  and  $\sigma^2$  are invariant to the value of  $\beta$  and less sensitive to outliers in the data compared to ML estimates. However, if you use REML to estimate the parameters, you can only compare two models that have the identical fixed-effects design matrices and are nested in their random-effects terms.

## References

- [1] Pinheiro, J. C., and D. M. Bates. *Mixed-Effects Models in S and S-PLUS*. Statistics and Computing Series, Springer, 2004.
- [2] Hariharan, S. and J. H. Rogers. "Estimation Procedures for Hierarchical Linear Models." *Multilevel Modeling of Educational Data* (A. A. Connell and D. B. McCoach, eds.). Charlotte, NC: Information Age Publishing, Inc., 2008.
- [3] Raudenbush, S. W. and A. S. Bryk. *Hierarchical Linear Models: Applications and Data Analysis Methods*, 2nd ed. Thousand Oaks, CA: Sage Publications, 2002.
- [4] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc, 2002.
- [5] Snijders, T. and R. Bosker. *Multilevel Analysis*. Thousand Oaks, CA: Sage Publications, 1999.

[6] McCulloch, C.E., R. S. Shayle, and J. M. Neuhaus. *Generalized, Linear, and Mixed Models*. Wiley, 2008.

**See Also**

`LinearMixedModel` | `fitlme` | `fitlmematrix`

**Concepts**

- “Linear Mixed-Effects Models” on page 9-153

# Generalized Linear Models

---

- “Multinomial Models for Nominal Responses” on page 10-2
- “Multinomial Models for Ordinal Responses” on page 10-5
- “Hierarchical Multinomial Models” on page 10-9
- “Generalized Linear Models” on page 10-12
- “Lasso Regularization of Generalized Linear Models” on page 10-48

## Multinomial Models for Nominal Responses

The outcome of a response variable might be one of a restricted set of possible values. If there are only two possible outcomes, such as a yes or no answer to a question, these responses are called binary responses. If there are multiple outcomes, then they are called polytomous responses. Some examples include the degree of a disease (mild, medium, severe), preferred districts to live in a city, and so on. When the response variable is *nominal*, there is no natural order among the response variable categories. Nominal response models explain and predict the probability that an observation is in each category of a categorical response variable.

A nominal response model is one of several natural extensions of the binary logit model and is also called a *multinomial logit* model. The multinomial logit model explains the relative risk of being in one category versus being in the reference category,  $k$ , using a linear combination of predictor variables. Consequently, the probability of each outcome is expressed as a nonlinear function of  $p$  predictor variables. The 'interactions', 'on' name-value pair argument in `mnrfit` corresponds to this multinomial model with separate intercept and slopes among categories. `mnrfit` uses the default logit link function for multinomial models. You cannot specify a different link function for multinomial responses.

The multinomial logit model is

$$\begin{aligned}\ln\left(\frac{\pi_1}{\pi_k}\right) &= \alpha_1 + \beta_{11}X_1 + \beta_{12}X_2 + \cdots + \beta_{1p}X_p, \\ \ln\left(\frac{\pi_2}{\pi_k}\right) &= \alpha_2 + \beta_{21}X_1 + \beta_{22}X_2 + \cdots + \beta_{2p}X_p, \\ &\vdots \\ \ln\left(\frac{\pi_{k-1}}{\pi_k}\right) &= \alpha_{(k-1)} + \beta_{(k-1)1}X_1 + \beta_{(k-1)2}X_2 + \cdots + \beta_{(k-1)p}X_p,\end{aligned}$$

where  $\pi_j = P(y = j)$  is the probability of an outcome being in category  $j$ ,  $k$  is the number of response categories, and  $p$  is the number of predictor variables. Theoretically, any category can be the reference category, but `mnrfit` chooses the last one,  $k$ , as the reference category. Thus, `mnrfit` assumes the coefficients of the  $k$ th category are zero. The total of  $j - 1$  equations are

solved simultaneously to estimate the coefficients. `mnrfit` uses the iteratively weighted least squares algorithm to find the maximum likelihood estimates.

The coefficients in the model express the effects of the predictor variables on the relative risk or the log odds of being in category  $j$  versus the reference category, here  $k$ . For example, the coefficient  $\beta_{23}$  indicates that the probability of the response variable being in category 2 compared to the probability of being in category  $k$  increases  $\exp(\beta_{23})$  times for each unit increase in  $X_3$ , given all else is held constant. Or it indicates that the relative log odds of the response variable being category 2 versus in category  $k$  increases  $\beta_{23}$  times with a one-unit increase in  $X_3$ , given all else equal.

Based on the nominal response model, and the assumption that the coefficients for the last category are zero, the probability of being in each category is

$$\pi_j = P(y = j) = \frac{e^{\sum_{l=1}^p \beta_{jl} x_l}}{1 + \sum_{j=1}^{k-1} e^{\sum_{l=1}^p \beta_{jl} x_l}}, \quad j = 1, \dots, k-1.$$

The probability of the  $k$ th category becomes

$$\pi_k = P(y = k) = \frac{1}{1 + \sum_{j=1}^{k-1} e^{\sum_{l=1}^p \beta_{jl} x_l}},$$

which is simply equal to  $1 - \pi_1 - \pi_2 - \dots - \pi_{k-1}$ .

After estimating the model coefficients using `mnrfit`, you can estimate the category probabilities or the number in each category using `mnrval` (the default name-value pair is 'type', 'category'). This function accepts the coefficient estimates and the model statistics `mnrfit` returns and estimates the categorical probabilities or the number in each category and their confidence bounds. You can also specify the cumulative or conditional probabilities or numbers to estimate using the 'type' name-value pair argument in `mnrval`.

## References

[1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

[2] Long, J. S. *Regression Models for Categorical and Limited Dependent Variables*. Sage Publications, 1997.

[3] Dobson, A. J., and A. G. Barnett. *An Introduction to Generalized Linear Models*. Chapman and Hall/CRC. Taylor & Francis Group, 2008.

## See Also

`fitglm` | `mnrfit` | `mnrval` | `glmfit` | `glmval`

## Concepts

- “Multinomial Models for Ordinal Responses” on page 10-5
- “Hierarchical Multinomial Models” on page 10-9



## Multinomial Models for Ordinal Responses

The outcome of a response variable might be one of a restricted set of possible values. If there are only two possible outcomes, such as male and female for gender, these responses are called binary responses. If there are multiple outcomes, then they are called polytomous responses. Some examples of polytomous responses include levels of a disease (mild, medium, severe), preferred districts to live in a city, the species for a certain flower type, and so on. Sometimes there might be a natural order among the response categories. These responses are called *ordinal responses*.

The ordering might be inherent in the category choices, such as an individual being not satisfied, satisfied, or very satisfied with an online customer service. The ordering might also be introduced by categorization of a latent (continuous) variable, such as in the case of an individual being in the low risk, medium risk, or high risk group for developing a certain disease, based on a quantitative medical measure such as blood pressure.

You can specify a multinomial regression model that uses the natural ordering among the response categories. This ordinal model describes the relationship between the cumulative probabilities of the categories and predictor variables.

Different link functions can describe this relationship with logit and probit being the most used.

- **Logit:** The default link function `mnrfit` uses for ordinal categories is the *logit* link function. This models the *log cumulative odds*. The `'link', 'logit'` name-value pair specifies this in `mnrfit`. Log cumulative odds is the logarithm of the ratio of the probability that a response belongs to a category with a value less than or equal to category  $j$ ,  $P(y \leq c_j)$ , and the probability that a response belongs to a category with a value greater than category  $j$ ,  $P(y > c_j)$ .

Ordinal models are usually based on the assumption that the effects of predictor variables are the same for all categories on the logarithmic scale. That is, the model has different intercepts but common slopes (coefficients) among categories. This model is called *parallel regression* or the *proportional odds* model. It is the default for ordinal responses, and the `'interactions', 'off'` name-value pair specifies this model in `mnrfit`.

The proportional odds model is

$$\begin{aligned}\ln\left(\frac{P(y \leq c_1)}{P(y > c_1)}\right) &= \ln\left(\frac{\pi_1}{\pi_2 + \dots + \pi_k}\right) = \alpha_1 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p, \\ \ln\left(\frac{P(y \leq c_2)}{P(y > c_2)}\right) &= \ln\left(\frac{\pi_1 + \pi_2}{\pi_3 + \dots + \pi_k}\right) = \alpha_2 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p, \\ &\vdots \\ \ln\left(\frac{P(y \leq c_{k-1})}{P(y > c_{k-1})}\right) &= \ln\left(\frac{\pi_1 + \pi_2 + \dots + \pi_{k-1}}{\pi_k}\right) = \alpha_{k-1} + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p,\end{aligned}$$

where  $\pi_j, j = 1, 2, \dots, k$ , are the category probabilities.

For example, for a response variable with three categories, there are  $3 - 1 = 2$  equations as follows:

$$\begin{aligned}\ln\left(\frac{\pi_1}{\pi_2 + \pi_3}\right) &= \alpha_1 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p, \\ \ln\left(\frac{\pi_1 + \pi_2}{\pi_3}\right) &= \alpha_2 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p.\end{aligned}$$

Under the proportional odds assumption, the partial effect of a predictor variable  $X$  is invariant to the choice of the response variable category,  $j$ . For example, if there are three categories, then the coefficients express the impact of a predictor variable on the relative risk or log odds of the response value being in category 1 versus categories 2 or 3, or in category 1 or 2 versus category 3.

Thus, a unit change in variable  $X_2$  would mean a change in the cumulative odds of the response value being in category 1 versus categories 2 or 3, or category 1 or 2 versus category 3 by a factor of  $\exp(\beta_2)$ , given all else equal.

You can alternatively fit a model with different intercept and slopes among the categories by using the 'interactions', 'on' name-value pair argument. However, using this option for ordinal models when the equal slopes model is true causes a loss of efficiency (you lose the advantage of estimating fewer parameters).

- **Probit:** The 'link', 'probit' name-value pair argument uses the *probit* link function which is based on a normally distributed latent variable assumption. For ordinal response variables this is also called an *ordered probit* model. Consider the regression model that describes the relationship

of a latent variable  $y^*$  of an ordinal process and a vector of predictor variables,  $X$ ,

$$y = \beta X + \varepsilon,$$

where the error term  $\varepsilon$  has a standard normal distribution. Suppose there is the following relationship between the latent variable  $y^*$  and the observed variable  $y$ :

$$\begin{aligned} y = c_1 & \text{ if } \alpha_0 < y^* \leq \alpha_1, \\ y = c_2 & \text{ if } \alpha_1 < y^* \leq \alpha_2, \\ & \vdots \\ y = c_k & \text{ if } \alpha_{k-1} < y^* \leq \alpha_k, \end{aligned}$$

where  $\alpha_0 = -\infty$  and  $\alpha_k = \infty$ . Then, the cumulative probability of  $y$  being in category  $j$  or one of earlier categories,  $P(y \leq c_j)$ , is equal to

$$P(y \leq c_j) = P(y^* < \alpha_j) = P(\beta X + \varepsilon < \alpha_j) = P(\varepsilon < \alpha_j - \beta X) = \Phi(\alpha_j - \beta X),$$

where  $\Phi$  is standard normal cumulative distribution function. Thus,

$$\Phi^{-1}(P(y \leq c_j)) = \alpha_j - \beta X,$$

where  $\alpha_j$  corresponds to the cut points of the latent variable and the intercept in the regression model. This only holds under the assumptions of a normal latent variable and parallel regression. More generally, for a response variable with  $k$  categories and multiple predictors, the ordered probit model is

$$\begin{aligned} \Phi^{-1}(P(y \leq c_1)) &= \alpha_1 + \beta_1 X_1 + \cdots + \beta_p X_p, \\ \Phi^{-1}(P(y \leq c_2)) &= \alpha_2 + \beta_1 X_1 + \cdots + \beta_p X_p, \\ & \vdots \\ \Phi^{-1}(P(y \leq c_{k-1})) &= \alpha_{k-1} + \beta_1 X_1 + \cdots + \beta_p X_p, \end{aligned}$$

where  $P(y \leq c_j) = \pi_1 + \pi_2 + \dots + \pi_j$ .

The coefficients indicate the impact of a unit change in the predictor variable on the likelihood of a state. A positive coefficient,  $\beta_1$ , for example, indicates an increase in the underlying latent variable with an increase in the corresponding predictor variable,  $X_1$ . Hence, it causes a decrease in  $P(y \leq c_1)$  and an increase in  $P(y \leq c_k)$ .

After estimating the model coefficients using `mnrfit`, you can estimate the cumulative probabilities or the cumulative number in each category using `mnrval` with the 'type', 'cumulative' name-value pair option. `mnrval` accepts the coefficient estimates and the model statistics `mnrfit` returns, and estimates the categorical probabilities or the number in each category and their confidence intervals. You can specify which category or conditional probabilities or numbers to estimate by changing the value of the 'type' name-value pair argument.

## References

- [1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] Long, J. S. *Regression Models for Categorical and Limited Dependent Variables*. Sage Publications, 1997.
- [3] Dobson, A. J., and A. G. Barnett. *An Introduction to Generalized Linear Models*. Chapman and Hall/CRC. Taylor & Francis Group, 2008.

## See Also

`fitglm` | `mnrfit` | `mnrval` | `glmfit` | `glmval`

## Concepts

- “Multinomial Models for Nominal Responses” on page 10-2
- “Hierarchical Multinomial Models” on page 10-9

## Hierarchical Multinomial Models

The outcome of a response variable might sometimes be one of a restricted set of possible values. If there are only two possible outcomes, such as male and female for gender, these responses are called binary responses. If there are multiple outcomes, then they are called polytomous responses. These responses are usually qualitative rather than quantitative, such as preferred districts to live in a city, the severity level of a disease, the species for a certain flower type, and so on. Polytomous responses might also have categories which are not independent of each other. Instead the response happens in a sequential manner, or one category is nested in the previous one. These types of responses are called *hierarchical*, or *sequential*, or *nested multinomial responses*.

For example, if the response is the number of cigarettes a person smokes in a given day, the first level is whether the person is a smoker or not. Given that he or she is a smoker, the number of cigarettes he or she smokes can be from one to five or more than five a day. Given that it is more than 5, this person might be smoking from 6 to 10 or more than 10 cigarettes a day, and so on. The risk group at each level changes accordingly. At level one, the risk group is all of the individuals of interest (smoker or not), say  $m$ . If out of  $m$  individuals,  $y_1$  of them are not smokers, then at level two, the risk group is the number of all smoking individuals,  $m - y_1$ . If  $y_2$  of these  $m - y_1$  individuals smoke from one to five cigarettes a day, then at level three, the risk group is  $m - y_1 - y_2$ . So, at each level, the number of people in that category becomes a conditional binomial observation.

The hierarchical multinomial regression models are extensions of binary regression models based on conditional binary observations. The default is a model with different intercept and slopes (coefficients) among categories, in which case `mnrfit` fits a sequence of conditional binomial models. The 'interactions', 'on' name-value pair specifies this in `mnrfit`. The default link function is logit and the 'link', 'logit' name-value pair specifies this model in `mnrfit`.

Suppose the probability that an individual is in category  $j$  given that he or she is not in the previous categories is  $\pi_j$ , and the cumulative probability that a response belongs to a category  $j$  or a previous category is  $P(y \leq c_j)$ . Then the hierarchical model with a logit link function and different slopes assumption is

$$\begin{aligned} \ln\left(\frac{\pi_1}{1 - P(y \leq c_1)}\right) &= \ln\left(\frac{\pi_1}{1 - \pi_1}\right) = \alpha_1 + \beta_{11}X_1 + \beta_{12}X_2 + \cdots + \beta_{1p}X_p, \\ \ln\left(\frac{\pi_2}{1 - P(y \leq c_2)}\right) &= \ln\left(\frac{\pi_2}{1 - (\pi_1 + \pi_2)}\right) = \alpha_2 + \beta_{21}X_1 + \beta_{22}X_2 + \cdots + \beta_{2p}X_p, \\ &\vdots \\ \ln\left(\frac{\pi_{k-1}}{1 - P(y \leq c_{k-1})}\right) &= \ln\left(\frac{\pi_{k-1}}{1 - (\pi_1 + \cdots + \pi_{k-1})}\right) = \alpha_{k-1} + \beta_{(k-1)1}X_1 + \beta_{(k-1)2}X_2 + \cdots + \beta_{(k-1)p}X_p. \end{aligned}$$

For example, for a response variable with four sequential categories, there are  $4 - 1 = 3$  equations as follows:

$$\begin{aligned} \ln\left(\frac{\pi_1}{\pi_2 + \pi_3 + \pi_4}\right) &= \alpha_1 + \beta_{11}X_1 + \beta_{12}X_2 + \cdots + \beta_{1p}X_p, \\ \ln\left(\frac{\pi_2}{\pi_3 + \pi_4}\right) &= \alpha_2 + \beta_{21}X_1 + \beta_{22}X_2 + \cdots + \beta_{2p}X_p, \\ \ln\left(\frac{\pi_3}{\pi_4}\right) &= \alpha_3 + \beta_{31}X_1 + \beta_{32}X_2 + \cdots + \beta_{3p}X_p. \end{aligned}$$

The coefficients  $\beta_{ij}$  are interpreted within each level. For example, for the previous smoking example,  $\beta_{12}$  shows the impact of  $X_2$  on the log odds of a person being a smoker versus a nonsmoker, provided that everything else is held constant. Alternatively,  $\beta_{22}$  shows the impact of  $X_2$  on the log odds of a person smoking one to five cigarettes versus more than five cigarettes a day, given that he or she is a smoker, provided that everything else is held constant. Similarly,  $\beta_{33}$  shows the effect of  $X_2$  on the log odds of a person smoking 6 to 10 cigarettes versus more than 10 cigarettes a day, given that he or she smokes more than 5 cigarettes a day, provided that everything else is held constant.

You can specify other link functions for hierarchical models. The 'link', 'probit' name-value pair argument uses the probit link function. With the separate slopes assumption, the model becomes

$$\begin{aligned}\Phi^{-1}(\pi_1) &= \alpha_1 + \beta_{11}X_1 + \cdots + \beta_{1p}X_p, \\ \Phi^{-1}(\pi_2) &= \alpha_2 + \beta_{21}X_1 + \cdots + \beta_{2p}X_p, \\ &\vdots \\ \Phi^{-1}(\pi_k) &= \alpha_k + \beta_{k1}X_1 + \cdots + \beta_{kp}X_p,\end{aligned}$$

where  $\pi_j$  is the conditional probability of being in category  $j$ , given that it is not in categories previous to category  $j$ . And  $\Phi^{-1}(\cdot)$  is the inverse of the standard normal cumulative distribution function.

After estimating the model coefficients using `mnrfit`, you can estimate the cumulative probabilities or the cumulative number in each category using `mnrval` with the 'type', 'conditional' name-value pair argument. The function `mnrval` accepts the coefficient estimates and the model statistics `mnrfit` returns, and estimates the categorical probabilities or the number in each category and their confidence bounds. You can specify which category or cumulative probabilities or numbers to estimate by changing the value of the 'type' name-value pair argument in `mnrval`.

## References

- [1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] Liao, T. F. *Interpreting Probability Models: Logit, Probit, and Other Generalized Linear Models* Series: Quantitative Applications in the Social Sciences. Sage Publications, 1994.

## See Also

`fitglm` | `mnrfit` | `mnrval` | `glmfit` | `glmval`

## Concepts

- “Multinomial Models for Nominal Responses” on page 10-2
- “Multinomial Models for Ordinal Responses” on page 10-5

## Generalized Linear Models

### In this section...

“What Are Generalized Linear Models?” on page 10-12

“Prepare Data” on page 10-13

“Choose Generalized Linear Model and Link Function” on page 10-15

“Choose Fitting Method and Model” on page 10-19

“Fit Model to Data” on page 10-24

“Examine Quality and Adjust the Fitted Model” on page 10-24

“Predict or Simulate Responses to New Data” on page 10-37

“Share Fitted Models” on page 10-40

“Generalized Linear Model Workflow” on page 10-42

### What Are Generalized Linear Models?

Linear regression models describe a linear relationship between a response and one or more predictive terms. Many times, however, a nonlinear relationship exists. “Nonlinear Regression” on page 11-2 describes general nonlinear models. A special class of nonlinear models, called *generalized linear models*, uses linear methods.

Recall that linear models have these characteristics:

- At each set of values for the predictors, the response has a normal distribution with mean  $\mu$ .
- A coefficient vector  $b$  defines a linear combination  $Xb$  of the predictors  $X$ .
- The model is  $\mu = Xb$ .

In generalized linear models, these characteristics are generalized as follows:

- At each set of values for the predictors, the response has a distribution that can be normal, binomial, Poisson, gamma, or inverse Gaussian, with parameters including a mean  $\mu$ .
- A coefficient vector  $b$  defines a linear combination  $Xb$  of the predictors  $X$ .



- A *link function*  $f$  defines the model as  $f(\mu) = Xb$ .

## Prepare Data

To begin fitting a regression, put your data into a form that fitting functions expect. All regression techniques begin with input data in an array  $X$  and response data in a separate vector  $y$ , or input data in a dataset array  $ds$  and response data as a column in  $ds$ . Each row of the input data represents one observation. Each column represents one predictor (variable).

For a dataset array  $ds$ , indicate the response variable with the 'ResponseVar' name-value pair:

```
mdl = fitlm(ds,'ResponseVar','BloodPressure');
% or
mdl = fitglm(ds,'ResponseVar','BloodPressure');
```

The response variable is the last column by default.

You can use numeric *categorical* predictors. A categorical predictor is one that takes values from a fixed set of possibilities.

- For a numeric array  $X$ , indicate the categorical predictors using the 'Categorical' name-value pair. For example, to indicate that predictors 2 and 3 out of six are categorical:

```
mdl = fitlm(X,y,'Categorical',[2,3]);
% or
mdl = fitglm(X,y,'Categorical',[2,3]);
% or equivalently
mdl = fitlm(X,y,'Categorical',logical([0 1 1 0 0 0]));
```

- For a dataset array  $ds$ , fitting functions assume that these data types are categorical:
  - Logical
  - Categorical (nominal or ordinal)
  - String or character array

If you want to indicate that a numeric predictor is categorical, use the 'Categorical' name-value pair.

Represent missing numeric data as NaN. To represent missing data for other data types, see “Missing Group Values” on page 2-53.

- For a 'binomial' model with data matrix  $X$ , the response  $y$  can be:
  - Binary column vector — Each entry represents success (1) or failure (0).
  - Two-column matrix of integers — The first column is the number of successes in each observation, the second column is the number of trials in that observation.
- For a 'binomial' model with dataset  $ds$ :
  - Use the `ResponseVar` name-value pair to specify the column of  $ds$  that gives the number of successes in each observation.
  - Use the `BinomialSize` name-value pair to specify the column of  $ds$  that gives the number of trials in each observation.

### **Dataset Array for Input and Response Data**

For example, to create a dataset array from an Excel spreadsheet:

```
ds = dataset('XLSFile','hospital.xls',...  
            'ReadObsNames',true);
```

To create a dataset array from workspace variables:

```
load carsmall  
ds = dataset(MPG,Weight);  
ds.Year = ordinal(Model_Year);
```

### **Numeric Matrix for Input Data, Numeric Vector for Response**

For example, to create numeric arrays from workspace variables:

```
load carsmall  
X = [Weight Horsepower Cylinders Model_Year];  
y = MPG;
```

To create numeric arrays from an Excel spreadsheet:

```
[X Xnames] = xlsread('hospital.xls');  
y = X(:,4); % response y is systolic pressure
```

```
X(:,4) = []; % remove y from the X matrix
```

Notice that the nonnumeric entries, such as `sex`, do not appear in `X`.

## Choose Generalized Linear Model and Link Function

Often, your data suggests the distribution type of the generalized linear model.

Response Data Type	Suggested Model Distribution Type
Any real number	'normal'
Any positive number	'gamma' or 'inverse gaussian'
Any nonnegative integer	'poisson'
Integer from 0 to $n$ , where $n$ is a fixed positive value	'binomial'

Set the model distribution type with the `Distribution` name-value pair. After selecting your model type, choose a link function to map between the mean  $\mu$  and the linear predictor  $Xb$ .

Value	Description
'comloglog'	$\log(-\log((1-\mu))) = Xb$
'identity', default for the distribution 'normal'	$\mu = Xb$
'log', default for the distribution 'poisson'	$\log(\mu) = Xb$
'logit', default for the distribution 'binomial'	$\log(\mu/(1-\mu)) = Xb$
'loglog'	$\log(-\log(\mu)) = Xb$
'probit'	$\Phi^{-1}(\mu) = Xb$ , where $\Phi$ is the normal (Gaussian) CDF function
'reciprocal', default for the distribution 'gamma'	$\mu^{-1} = Xb$

Value	Description
p (a number), default for the distribution 'inverse gaussian' (with $p = -2$ )	$\mu^p = Xb$
Cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI). Equivalently, can be a structure of function handles with field Link containing FL, field Derivative containing FD, and field Inverse containing FI.	User-specified link function (see “Custom Link Function” on page 10-16)

The nondefault link functions are mainly useful for binomial models. These nondefault link functions are 'comloglog', 'loglog', and 'probit'.

### Custom Link Function

The link function defines the relationship  $f(\mu) = Xb$  between the mean response  $\mu$  and the linear combination  $Xb = X*b$  of the predictors. You can choose one of the built-in link functions or define your own by specifying the link function FL, its derivative FD, and its inverse FI:

- The link function FL calculates  $f(\mu)$ .
- The derivative of the link function FD calculates  $df(\mu)/d\mu$ .
- The inverse function FI calculates  $g(Xb) = \mu$ .

You can specify a custom link function in either of two equivalent ways. Each way contains function handles that accept a single array of values representing  $\mu$  or  $Xb$ , and returns an array the same size. The function handles are either in a cell array or a structure:

- Cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI).
- Structure `s` with three fields, each containing a function handle created using @:
  - `s.Link` — Link function
  - `s.Derivative` — Derivative of the link function
  - `s.Inverse` — Inverse of the link function

For example, to fit a model using the 'probit' link function:

```
x = [2100 2300 2500 2700 2900 ...
     3100 3300 3500 3700 3900 4100 4300]';
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
g = fitglm(x,[y n],...
           'linear','distr','binomial','link','probit')
```

```
g =
```

Generalized Linear regression model:

```
probit(y) ~ 1 + x1
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-7.3628	0.66815	-11.02	3.0701e-28
x1	0.0023039	0.00021352	10.79	3.8274e-27

12 observations, 10 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 241, p-value = 2.25e-54

You can perform the same fit using a custom link function that performs identically to the 'probit' link function:

```
s = {@norminv,@(x)1./normpdf(norminv(x)),@normcdf};
g = fitglm(x,[y n],...
           'linear','distr','binomial','link',s)
```

```

g =

Generalized Linear regression model:
  link(y) ~ 1 + x1
  Distribution = Binomial

Estimated Coefficients:

```

	Estimate	SE	tStat	pValue
(Intercept)	-7.3628	0.66815	-11.02	3.0701e-28
x1	0.0023039	0.00021352	10.79	3.8274e-27

```

12 observations, 10 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 241, p-value = 2.25e-54

```

The two models are the same.

Equivalently, you can write `s` as a structure instead of a cell array of function handles:

```

s.Link = @norminv;
s.Derivative = @(x) 1./normpdf(norminv(x));
s.Inverse = @normcdf;
g = fitglm(x,[y n],...
  'linear','distr','binomial','link',s)

g =

Generalized Linear regression model:
  link(y) ~ 1 + x1
  Distribution = Binomial

Estimated Coefficients:

```

	Estimate	SE	tStat	pValue
(Intercept)	-7.3628	0.66815	-11.02	3.0701e-28
x1	0.0023039	0.00021352	10.79	3.8274e-27

```

12 observations, 10 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 241, p-value = 2.25e-54

```

## Choose Fitting Method and Model

There are two ways to create a fitted model.

- Use `fitglm` when you have a good idea of your generalized linear model, or when you want to adjust your model later to include or exclude certain terms.
- Use `stepwiseglm` when you want to fit your model using stepwise regression. `stepwiseglm` starts from one model, such as a constant, and adds or subtracts terms one at a time, choosing an optimal term each time in a greedy fashion, until it cannot improve further. Use stepwise fitting to find a good model, one that has only relevant terms.

The result depends on the starting model. Usually, starting with a constant model leads to a small model. Starting with more terms can lead to a more complex model, but one that has lower mean squared error.

In either case, provide a model to the fitting function (which is the starting model for `stepwiseglm`).

Specify a model using one of these methods.

- “Brief String” on page 10-19
- “Terms Matrix” on page 10-20
- “Formula” on page 10-23

### Brief String

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.

String	Model Type
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.



```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

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

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```

0    0    0
1    0    0
0    1    0
1    1    0
0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

```
y ~ 1 + x1*x2 + x2^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

mdl =

Linear regression model:

$$y \sim 1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of fitlm in the previous step.

## Formula

A formula for a model specification is a string of the form

`'Y ~ terms'`,

- Y is the response name.
- *terms* contains
  - Variable names
  - + to include the next variable
  - - to exclude the next variable
  - : to define an interaction, a product of terms
  - \* to define an interaction and all lower-order terms
  - ^ to raise the predictor to a power, exactly as in \* repeated, so ^ includes lower order terms as well
  - () to group terms

---

**Tip** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

Examples:

`'Y ~ A + B + C'` is a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a  $B^2$  term.

`'Y ~ A + B^2 + C'` is the same as the previous example, since  $B^2$  includes a  $B$  term.

`'Y ~ A + B + C + A:B'` includes an  $A*B$  term.

`'Y ~ A*B + C'` is the same as the previous example, since  $A*B = A + B + A:B$ .

`'Y ~ A*B*C - A:B:C'` has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

`'Y ~ A*(B + C + D)'` has all linear terms, plus products of  $A$  with each of the other variables.

## Fit Model to Data

Create a fitted model using `fitglm` or `stepwiseglm`. Choose between them as in “Choose Fitting Method and Model” on page 10-19. For generalized linear models other than those with a normal distribution, give a `Distribution` name-value pair as in “Choose Generalized Linear Model and Link Function” on page 10-15. For example,

```
mdl = fitglm(X,y,'linear','Distribution','poisson')
% or
mdl = fitglm(X,y,'quadratic',...
            'Distribution','binomial')
```

## Examine Quality and Adjust the Fitted Model

After fitting a model, examine the result.

- “Model Display” on page 10-25
- “Diagnostic Plots” on page 10-26

- “Residuals — Model Quality for Training Data” on page 10-29
- “Plots to Understand Predictor Effects and How to Modify a Model” on page 10-32

## Model Display

A linear regression model shows several diagnostics when you enter its name or enter `disp mdl`. This display gives some of the basic information to check whether the fitted model represents the data adequately.

For example, fit a Poisson model to data constructed with two out of five predictors not affecting the response, and with no intercept term:

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:,[1 4 5])*[.4;.2;.3]);
y = poissrnd(mu);
mdl = fitglm(X,y,...
    'linear','Distribution','poisson')
```

```
mdl =
```

Generalized Linear regression model:

```
log(y) ~ 1 + x1 + x2 + x3 + x4 + x5
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.039829	0.10793	0.36901	0.71212
x1	0.38551	0.076116	5.0647	4.0895e-07
x2	-0.034905	0.086685	-0.40266	0.6872
x3	-0.17826	0.093552	-1.9054	0.056722
x4	0.21929	0.09357	2.3436	0.019097
x5	0.28918	0.1094	2.6432	0.0082126

100 observations, 94 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 44.9, p-value = 1.55e-08

Notice that:

- The display contains the estimated values of each coefficient in the Estimate column. These values are reasonably near the true values [0; .4; 0; 0; .2; .3], except possibly the coefficient of `x3` is not terribly near 0.
- There is a standard error column for the coefficient estimates.
- The reported `pValue` (which are derived from the  $t$  statistics under the assumption of normal errors) for predictors 1, 4, and 5 are small. These are the three predictors that were used to create the response data `y`.
- The `pValue` for (Intercept), `x2` and `x3` are larger than 0.01. These three predictors were not used to create the response data `y`. The `pValue` for `x3` is just over .05, so might be regarded as possibly significant.
- The display contains the Chi-square statistic.

### Diagnostic Plots

Diagnostic plots help you identify outliers, and see other problems in your model or fit. To illustrate these plots, consider binomial regression with a logistic link function.

The *logistic model* is useful for proportion data. It defines the relationship between the proportion  $p$  and the weight  $w$  by:

$$\log[p/(1 - p)] = b_1 + b_2w$$

This example fits a binomial model to data. The data are derived from `carbig.mat`, which contains measurements of large cars of various weights. Each weight in `w` has a corresponding number of cars in `total` and a corresponding number of poor-mileage cars in `poor`.

It is reasonable to assume that the values of `poor` follow binomial distributions, with the number of trials given by `total` and the percentage of successes depending on `w`. This distribution can be accounted for in the context of a logistic model by using a generalized linear model with link function  $\log(\mu/(1 - \mu)) = Xb$ . This link function is called 'logit'.

`w = [2100 2300 2500 2700 2900 3100 ...`

```
3300 3500 3700 3900 4100 4300]';  
total = [48 42 31 34 31 21 23 23 21 16 17 21]';  
poor = [1 2 0 3 8 8 14 17 19 15 17 21]';  
mdl = fitglm(w,[poor total],...  
            'linear','Distribution','binomial','link','logit')
```

```
mdl =
```

Generalized Linear regression model:

logit(y) ~ 1 + x1

Distribution = Binomial

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-13.38	1.394	-9.5986	8.1019e-22
x1	0.0041812	0.00044258	9.4474	3.4739e-21

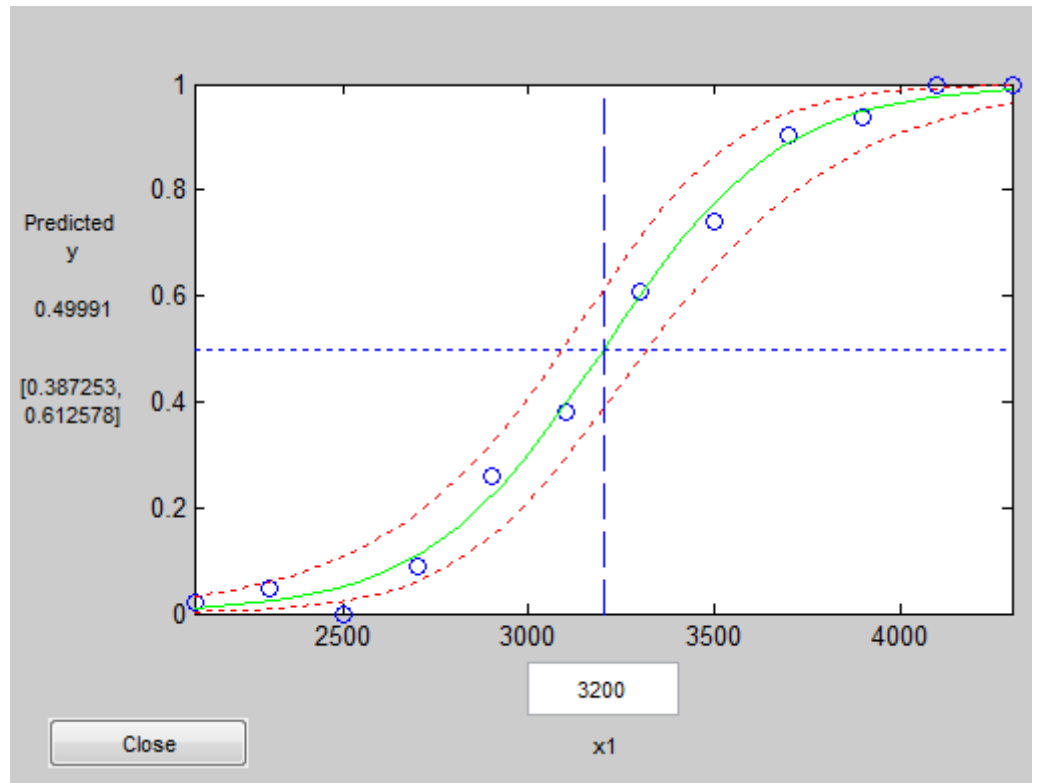
12 observations, 10 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 242, p-value = 1.3e-54

See how well the model fits the data.

```
plotSlice(mdl)
```

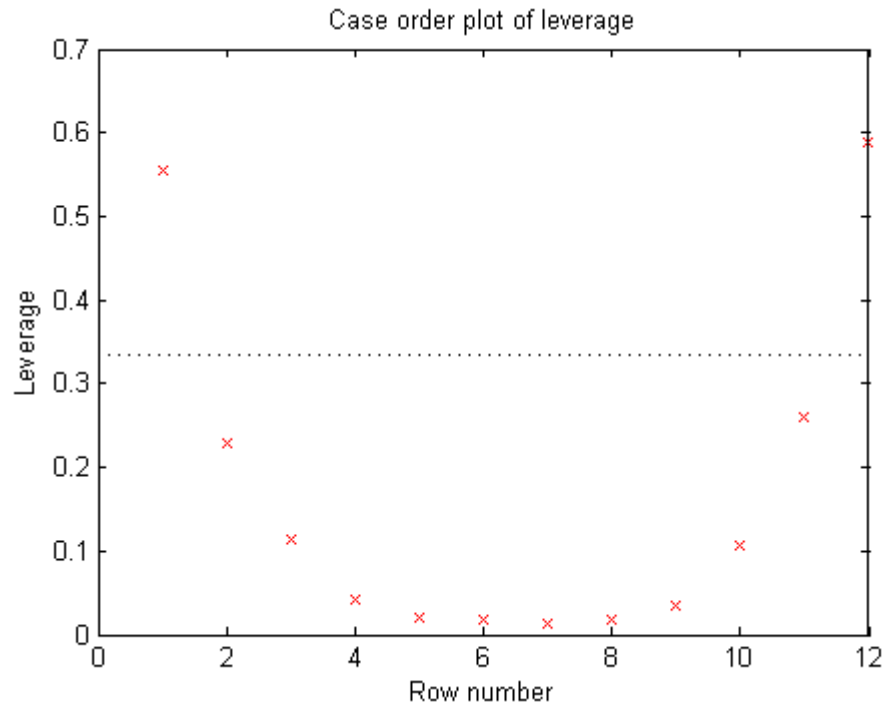


The fit looks reasonably good, with fairly wide confidence bounds.

To examine further details, create a leverage plot.

```
plotDiagnostics(md1)
```





This is typical of a regression with points ordered by the predictor variable. The leverage of each point on the fit is higher for points with relatively extreme predictor values (in either direction) and low for points with average predictor values. In examples with multiple predictors and with points not ordered by predictor value, this plot can help you identify which observations have high leverage because they are outliers as measured by their predictor values.

### **Residuals – Model Quality for Training Data**

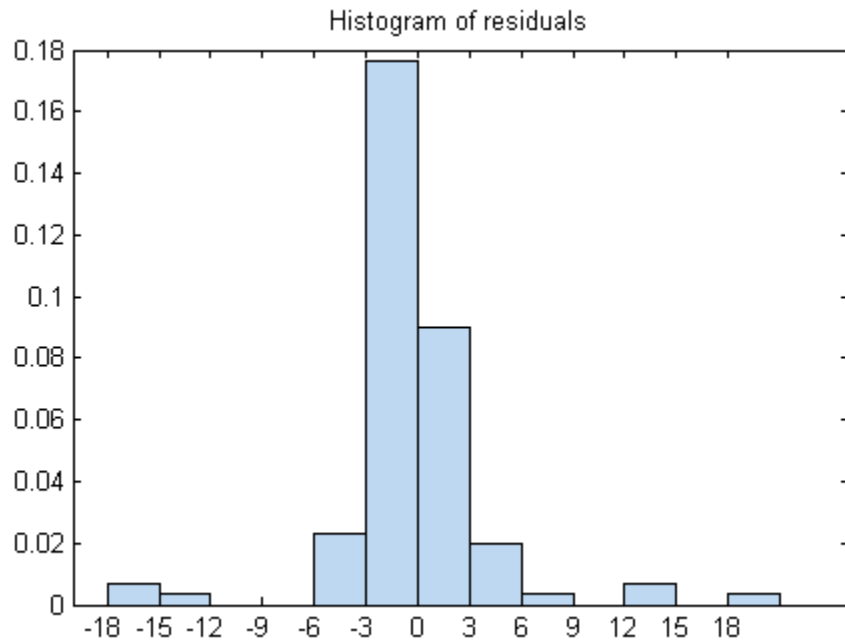
There are several residual plots to help you discover errors, outliers, or correlations in the model or data. The simplest residual plots are the default histogram plot, which shows the range of the residuals and their frequencies, and the probability plot, which shows how the distribution of the residuals compares to a normal distribution with matched variance.

This example shows residual plots for a fitted Poisson model. The data construction has two out of five predictors not affecting the response, and no intercept term:

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:,[1 4 5])*[2;1;.5]);
y = poissrnd(mu);
mdl = fitglm(X,y,...
            'linear','Distribution','poisson');
```

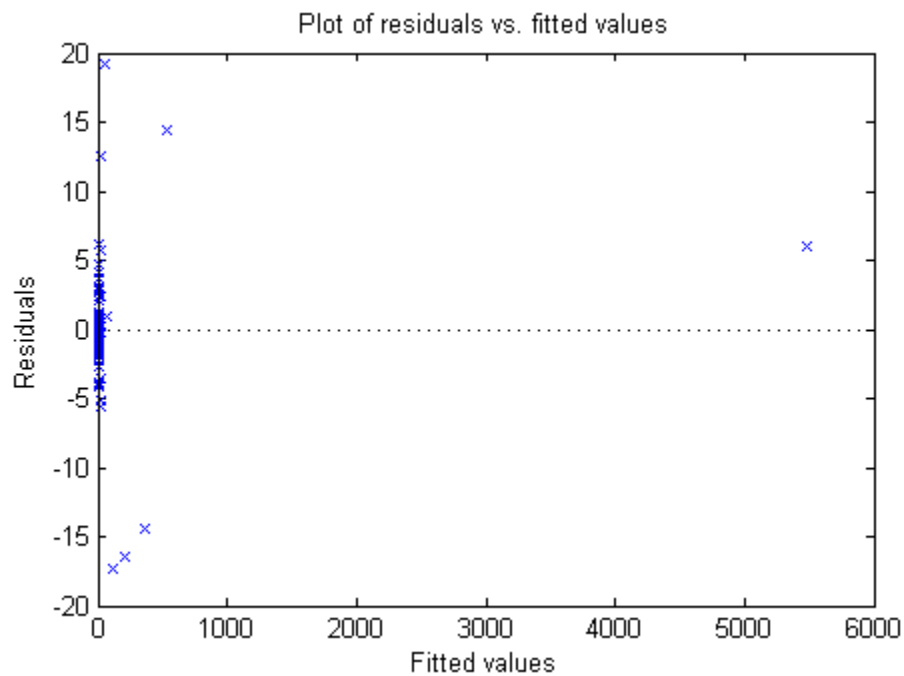
Examine the residuals:

```
plotResiduals(mdl)
```



While most residuals cluster near 0, there are several near  $\pm 18$ . So examine a different residuals plot.

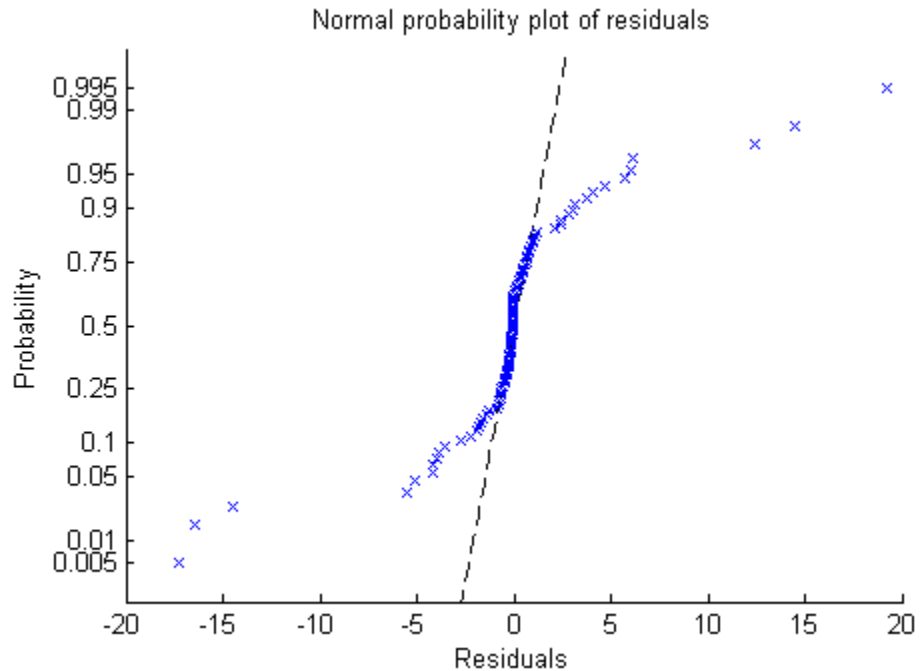
```
plotResiduals mdl, 'fitted'
```



The large residuals don't seem to have much to do with the sizes of the fitted values.

Perhaps a probability plot is more informative.

```
plotResiduals mdl, 'probability'
```



Now it is clear. The residuals do not follow a normal distribution. Instead, they have fatter tails, much as an underlying Poisson distribution.

### Plots to Understand Predictor Effects and How to Modify a Model

This example shows how to understand the effect each predictor has on a regression model, and how to modify the model to remove unnecessary terms.

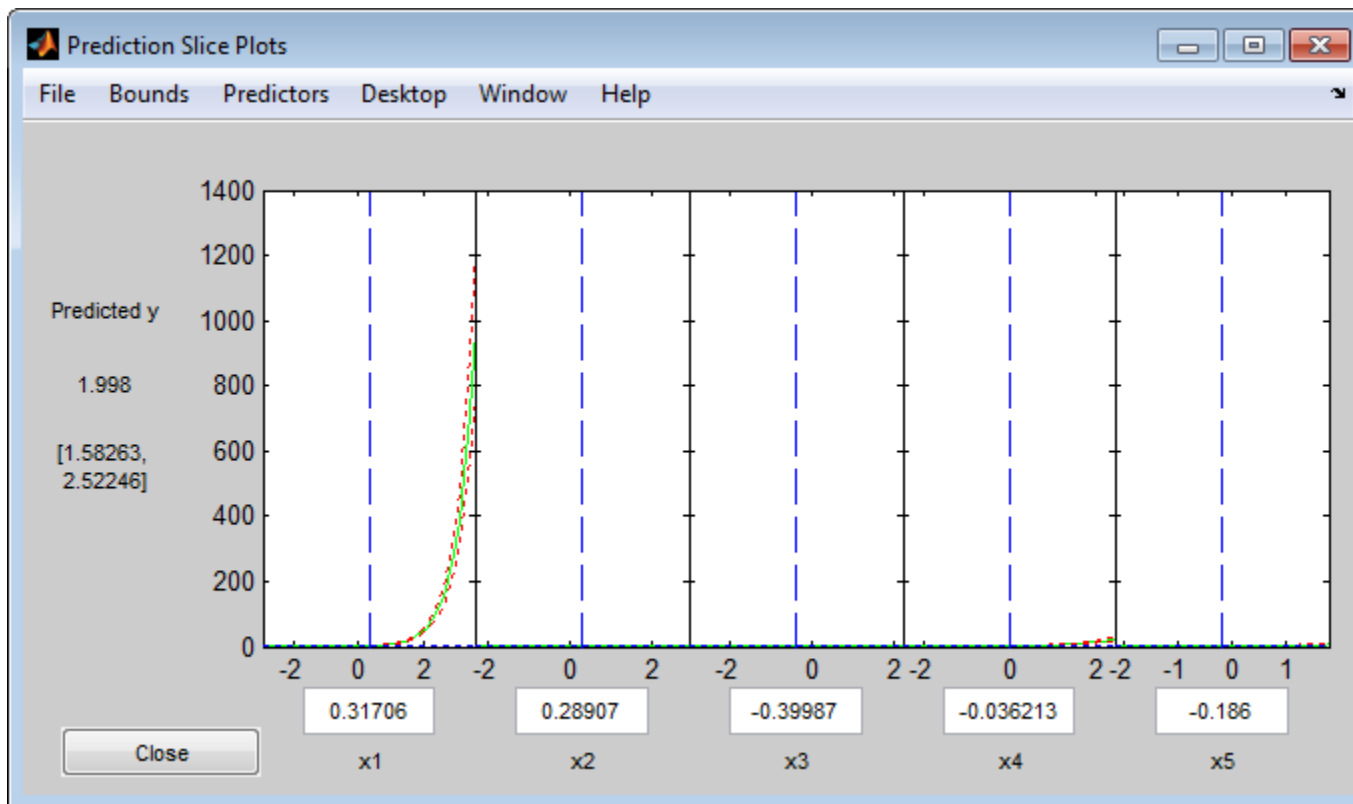
- 1 Create a model from some predictors in artificial data. The data do not use the second and third columns in  $X$ . So you expect the model not to show much dependence on those predictors.

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:,[1 4 5])*[2;1;.5]);
y = poissrnd(mu);
mdl = fitglm(X,y,...
```

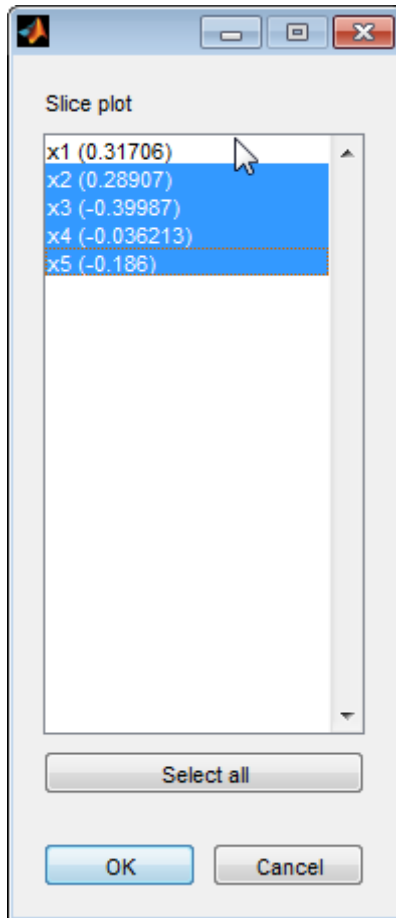
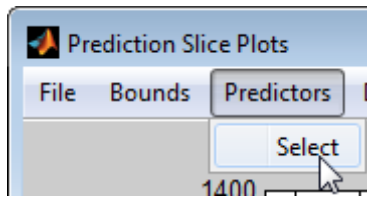
```
'linear', 'Distribution', 'poisson');
```

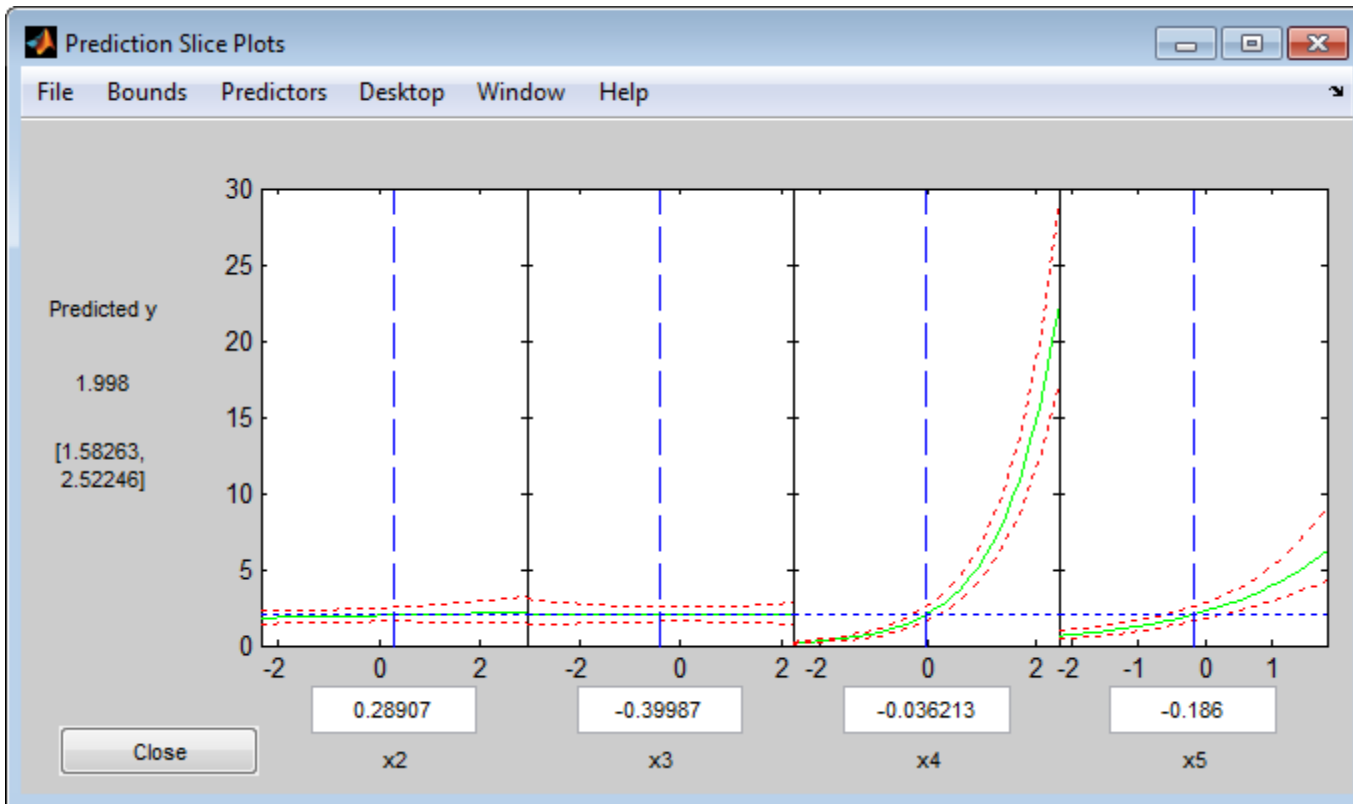
- 2 Examine a slice plot of the responses. This displays the effect of each predictor separately.

```
plotSlice mdl
```



The scale of the first predictor is overwhelming the plot. Disable it using the **Predictors** menu.





Now it is clear that predictors 2 and 3 have little to no effect.

You can drag the individual predictor values, which are represented by dashed blue vertical lines. You can also choose between simultaneous and non-simultaneous confidence bounds, which are represented by dashed red curves. Dragging the predictor lines confirms that predictors 2 and 3 have little to no effect.

- Remove the unnecessary predictors using either `removeTerms` or `step`. Using `step` can be safer, in case there is an unexpected importance to a term that becomes apparent after removing another term. However, sometimes `removeTerms` can be effective when `step` does not proceed. In this case, the two give identical results.

```
mdl1 = removeTerms(mdl, 'x2 + x3')
```

```
mdl1 =
```

Generalized Linear regression model:

$\log(y) \sim 1 + x1 + x4 + x5$

Distribution = Poisson

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.17604	0.062215	2.8295	0.004662
x1	1.9122	0.024638	77.614	0
x4	0.98521	0.026393	37.328	5.6696e-305
x5	0.61321	0.038435	15.955	2.6473e-57

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 4.97e+04, p-value = 0

```
mdl1 = step(mdl, 'NSteps', 5, 'Upper', 'linear')
```

1. Removing x3, Deviance = 93.856, Chi2Stat = 0.00075551, PValue = 0.97807

2. Removing x2, Deviance = 96.333, Chi2Stat = 2.4769, PValue = 0.11553

```
mdl1 =
```

Generalized Linear regression model:

$\log(y) \sim 1 + x1 + x4 + x5$

Distribution = Poisson

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.17604	0.062215	2.8295	0.004662
x1	1.9122	0.024638	77.614	0
x4	0.98521	0.026393	37.328	5.6696e-305
x5	0.61321	0.038435	15.955	2.6473e-57



```

100 observations, 96 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 4.97e+04, p-value = 0

```

## Predict or Simulate Responses to New Data

There are three ways to use a linear model to predict the response to new data:

- “predict” on page 10-37
- “feval” on page 10-38
- “random” on page 10-39

### predict

The `predict` method gives a prediction of the mean responses and, if requested, confidence bounds.

This example shows how to predict and obtain confidence intervals on the predictions using the `predict` method.

- 1 Create a model from some predictors in artificial data. The data do not use the second and third columns in  $X$ . So you expect the model not to show much dependence on these predictors. Construct the model stepwise to include the relevant predictors automatically.

```

rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:,[1 4 5])*[2;1;.5]);
y = poissrnd(mu);
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson');

```

```

1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52

```

- 2 Generate some new data, and evaluate the predictions from the data.

```

Xnew = randn(3,5) + repmat([1 2 3 4 5],[3,1]); % new data
[ynew,ynewci] = predict(mdl,Xnew)

```

```

ynew =

    1.0e+04 *

    0.1130
    1.7375
    3.7471

ynewci =

    1.0e+04 *

    0.0821    0.1555
    1.2167    2.4811
    2.8419    4.9407

```

## feval

When you construct a model from a dataset array, `feval` is often more convenient for predicting mean responses than `predict`. However, `feval` does not provide confidence bounds.

This example shows how to predict mean responses using the `feval` method.

- 1 Create a model from some predictors in artificial data. The data do not use the second and third columns in  $X$ . So you expect the model not to show much dependence on these predictors. Construct the model stepwise to include the relevant predictors automatically.

```

rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:,[1 4 5])*[2;1;.5]);
y = poissrnd(mu);
ds = dataset({X,'x1','x2','x3','x4','x5'},{y,'y'});
mdl = stepwiseglm(ds,...
    'constant','upper','linear','Distribution','poisson');

```

1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52

**2** Generate some new data, and evaluate the predictions from the data.

```
Xnew = randn(3,5) + repmat([1 2 3 4 5],[3,1]); % new data
ynew = feval mdl,Xnew(:,1),Xnew(:,4),Xnew(:,5)) % only need predictors 1,
```

```
ynew =

    1.0e+04 *

    0.1130
    1.7375
    3.7471
```

Equivalently,

```
ynew = feval(mdl,Xnew(:,[1 4 5])) % only need predictors 1,4,5
```

```
ynew =

    1.0e+04 *

    0.1130
    1.7375
    3.7471
```

## random

The `random` method generates new random response values for specified predictor values. The distribution of the response values is the distribution used in the model. `random` calculates the mean of the distribution from the predictors, estimated coefficients, and link function. For distributions such as normal, the model also provides an estimate of the variance of the response. For the binomial and Poisson distributions, the variance of the response is determined by the mean; `random` does not use a separate “dispersion” estimate.

This example shows how to simulate responses using the `random` method.

**1** Create a model from some predictors in artificial data. The data do not use the second and third columns in `X`. So you expect the model not to show

much dependence on these predictors. Construct the model stepwise to include the relevant predictors automatically.

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:,[1 4 5])*[2;1;.5]);
y = poissrnd(mu);
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson');
```

1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52

## 2 Generate some new data, and evaluate the predictions from the data.

```
Xnew = randn(3,5) + repmat([1 2 3 4 5],[3,1]); % new data
ysim = random(mdl,Xnew)
```

```
ysim =

    1111
    17121
    37457
```

The predictions from random are Poisson samples, so are integers.

## 3 Evaluate the random method again, the result changes.

```
ysim = random(mdl,Xnew)
```

```
ysim =

    1175
    17320
    37126
```

## Share Fitted Models

The model display contains enough information to enable someone else to recreate the model in a theoretical sense. For example,

```
rng('default') % for reproducibility
```

```
X = randn(100,5);
mu = exp(X(:, [1 4 5])*[2;1;.5]);
y = poissrnd(mu);
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson')
```

1. Adding x1, Deviance = 2515.02869, Chi2Stat = 47242.9622, PValue = 0
2. Adding x4, Deviance = 328.39679, Chi2Stat = 2186.6319, PValue = 0
3. Adding x5, Deviance = 96.3326, Chi2Stat = 232.0642, PValue = 2.114384e-52

```
mdl =
```

Generalized Linear regression model:

```
log(y) ~ 1 + x1 + x4 + x5
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	0.17604	0.062215	2.8295	0.004662
x1	1.9122	0.024638	77.614	0
x4	0.98521	0.026393	37.328	5.6696e-305
x5	0.61321	0.038435	15.955	2.6473e-57

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 4.97e+04, p-value = 0

You can access the model description programmatically, too. For example,

```
mdl.Coefficients.Estimate
```

```
ans =
```

```
0.1760
1.9122
0.9852
0.6132
```

```
mdl.Formula
```

```
ans =

log(y) ~ 1 + x1 + x4 + x5
```

## Generalized Linear Model Workflow

This example shows how to fit a generalized linear model and analyze the results. A typical workflow involves the following: import data, fit a generalized linear model, test its quality, modify it to improve the quality, and make predictions based on the model. It computes the probability that a flower is in one of two classes, based on the Fisher iris data.

### Step 1. Load the data.

Load the Fisher iris data. Extract the rows that have classification `versicolor` or `virginica`. These are rows 51 to 150. Create logical response variables that are true for `versicolor` flowers.

```
load fisheriris
X = meas(51:end,:); % versicolor and virginica
y = strcmp('versicolor',species(51:end));
```

### Step 2. Fit a generalized linear model.

Fit a binomial generalized linear model to the data.

```
mdl = fitglm(X,y,'linear',...
            'distr','binomial')
```

```
mdl =
```

```
Generalized Linear regression model:
logit(y) ~ 1 + x1 + x2 + x3 + x4
Distribution = Binomial
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	42.638	25.708	1.6586	0.097204
x1	2.4652	2.3943	1.0296	0.30319
x2	6.6809	4.4796	1.4914	0.13585
x3	-9.4294	4.7372	-1.9905	0.046537
x4	-18.286	9.7426	-1.8769	0.060529

```

100 observations, 95 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 127, p-value = 1.95e-26

```

### Step 3. Examine the result, consider alternative models.

Some  $p$ -values in the `pValue` column are not very small. Perhaps the model can be simplified.

See if some 95% confidence intervals for the coefficients include 0. If so, perhaps these model terms could be removed.

```
confint = coefCI mdl
```

```

confint =
   -8.3984   93.6740
   -2.2881    7.2185
   -2.2122   15.5739
  -18.8339   -0.0248
  -37.6277    1.0554

```

Only two of the predictors have coefficients whose confidence intervals do not include 0.

The coefficients of 'x1' and 'x2' have the largest  $p$ -values. Test whether both coefficients could be zero.

```

M = [0 1 0 0 0      % picks out coefficient for column 1
      0 0 1 0 0];  % picks out coefficient for column 2
p = coefTest(mdl,M)

```

```
p =
```

```
0.1442
```

The  $p$ -value of about 0.14 is not very small. Drop those terms from the model.

```
mdl1 = removeTerms(mdl, 'x1 + x2')
```

```
mdl1 =
```

Generalized Linear regression model:

logit(y) ~ 1 + x3 + x4  
 Distribution = Binomial

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	45.272	13.612	3.326	0.00088103
x3	-5.7545	2.3059	-2.4956	0.012576
x4	-10.447	3.7557	-2.7816	0.0054092

100 observations, 97 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 118, p-value = 2.3e-26

Perhaps it would have been better to have `stepwiseglm` identify the model initially.

```
mdl2 = stepwiseglm(X,y,...
    'constant','Distribution','binomial','upper','linear')
```

1. Adding x4, Deviance = 33.4208, Chi2Stat = 105.2086, PValue = 1.099298e-24
2. Adding x3, Deviance = 20.5635, Chi2Stat = 12.8573, PValue = 0.000336166
3. Adding x2, Deviance = 13.2658, Chi2Stat = 7.29767, PValue = 0.00690441

mdl2 =

Generalized Linear regression model:

logit(y) ~ 1 + x2 + x3 + x4  
 Distribution = Binomial

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	50.527	23.995	2.1057	0.035227
x2	8.3761	4.7612	1.7592	0.078536
x3	-7.8745	3.8407	-2.0503	0.040334
x4	-21.43	10.707	-2.0014	0.04535

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 125, p-value = 5.4e-27

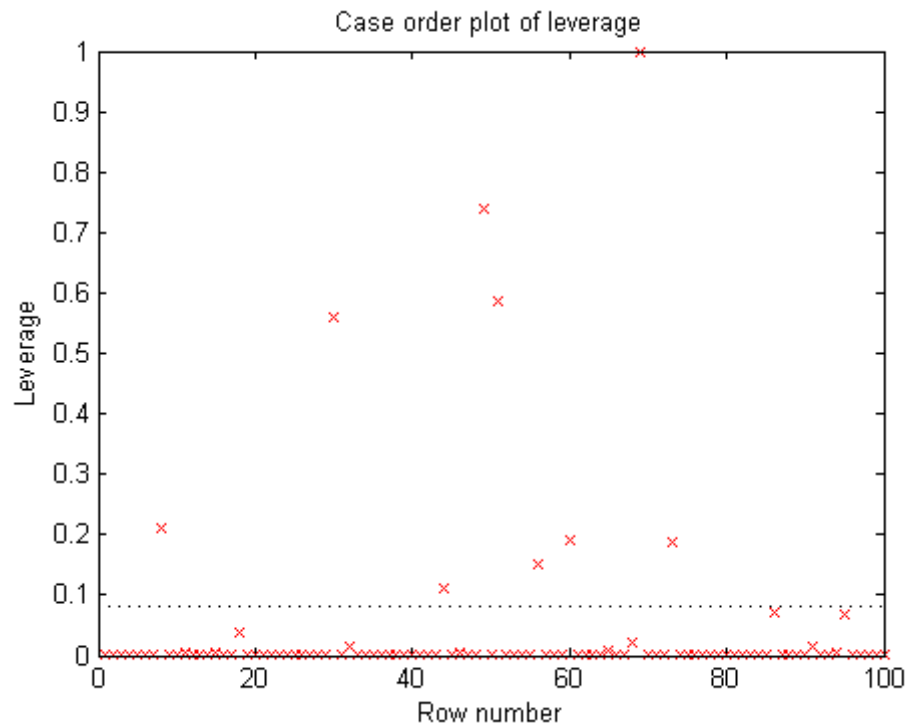


stepwiseglm included 'x2' in the model, because it neither adds nor removes terms with  $p$ -values between 0.05 and 0.10.

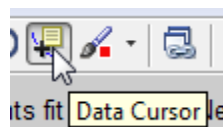
**Step 4. Look for outliers and exclude them.**

Examine a leverage plot to look for influential outliers.

```
plotDiagnostics(md12, 'leverage')
```



There is one observation with a leverage close to one. Using the Data Cursor, click the point, and find it has index 69.



See if the model coefficients change when you fit a model excluding this point.

```
oldCoeffs = mdl2.Coefficients.Estimate;  
mdl3 = fitglm(X,y,'linear',...  
    'distr','binomial','pred',2:4,'exclude',69);  
newCoeffs = mdl3.Coefficients.Estimate;  
disp([oldCoeffs newCoeffs])
```

```
50.5268    50.5268  
8.3761     8.3761  
-7.8745    -7.8745  
-21.4296   -21.4296
```

The model coefficients do not change, suggesting that the response at the high-leverage point is consistent with the predicted value from the reduced model.

#### **Step 5. Predict the probability that a new flower is versicolor.**

Use `mdl2` to predict the probability that a flower with average measurements is versicolor. Generate confidence intervals for your prediction.

```
[newf newc] = predict(mdl2,mean(X))
```

```
newf =  
0.5086
```

```
newc =  
0.1863    0.8239
```

The model gives almost a 50% probability that the average flower is versicolor, with a wide confidence interval about this estimate.

## References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.

[3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

[4] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.

## Lasso Regularization of Generalized Linear Models

### In this section...

“What is Generalized Linear Model Lasso Regularization?” on page 10-48

“Regularize Poisson Regression” on page 10-48

“Regularize Logistic Regression” on page 10-52

“Regularize Wide Data in Parallel” on page 10-59

“Generalized Linear Model Lasso and Elastic Net” on page 10-65

“References” on page 10-67

### What is Generalized Linear Model Lasso Regularization?

Lasso is a regularization technique. Use `lasso1m` to:

- Reduce the number of predictors in a generalized linear model.
- Identify important predictors.
- Select among redundant predictors.
- Produce shrinkage estimates with potentially lower predictive errors than ordinary least squares.

Elastic net is a related technique. Use it when you have several highly correlated variables. `lasso1m` provides elastic net regularization when you set the `Alpha` name-value pair to a number strictly between 0 and 1.

For details about lasso and elastic net computations and algorithms, see “Generalized Linear Model Lasso and Elastic Net” on page 10-65. For a discussion of generalized linear models, see “What Are Generalized Linear Models?” on page 10-12.

### Regularize Poisson Regression

This example shows how to identify and remove redundant predictors from a generalized linear model.

Create data with 20 predictors, and Poisson responses using just three of the predictors, plus a constant.

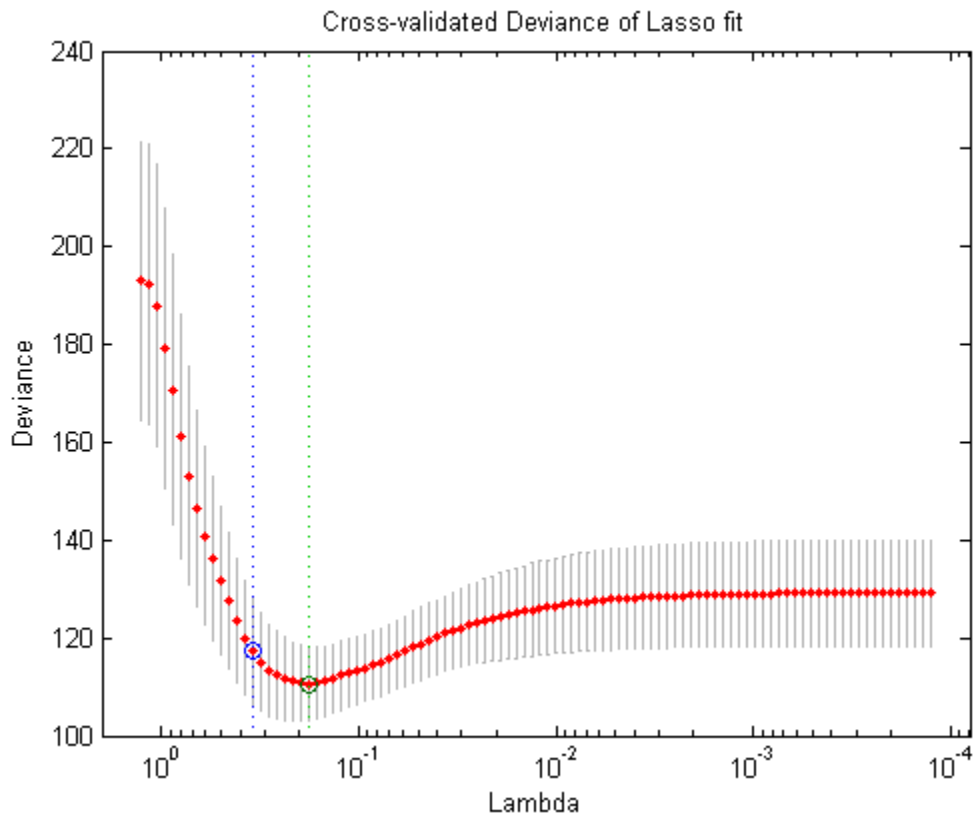
```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15]) * [.4; .2; .3] + 1);
y = poissrnd(mu);
```

Construct a cross-validated lasso regularization of a Poisson regression model of the data.

```
[B FitInfo] = lassoglm(X,y,'poisson','CV',10);
```

Examine the cross-validation plot to see the effect of the Lambda regularization parameter.

```
lassoPlot(B,FitInfo,'plottype','CV');
```



The green circle and dashed line locate the Lambda with minimal cross-validation error. The blue circle and dashed line locate the point with minimal cross-validation error plus one standard deviation.

Find the nonzero model coefficients corresponding to the two identified points.

```
minpts = find(B(:,FitInfo.IndexMinDeviance))
```

```
minpts =
     3
     5
     6
```

```

10
11
15
16

```

```
min1pts = find(B(:,FitInfo.Index1SE))
```

```
min1pts =
```

```

5
10
15

```

The coefficients from the minimal plus one standard error point are exactly those coefficients used to create the data.

Find the values of the model coefficients at the minimal plus one standard error point.

```
B(min1pts,FitInfo.Index1SE)
```

```
ans =
```

```

0.2903
0.0789
0.2081

```

The values of the coefficients are, as expected, smaller than the original  $[0.4, 0.2, 0.3]$ . Lasso works by “shrinkage,” which biases predictor coefficients toward zero.

The constant term is in the `FitInfo.Intercept` vector.

```
FitInfo.Intercept(FitInfo.Index1SE)
```

```
ans =
```

```
1.0879
```

The constant term is near 1, which is the value used to generate the data.

## Regularize Logistic Regression

This example shows how to regularize binomial regression. The default (canonical) link function for binomial regression is the logistic function.

### Step 1. Prepare the data.

Load the `ionosphere` data. The response `Y` is a cell array of 'g' or 'b' strings. Convert the cells to logical values, with `true` representing 'g'. Remove the first two columns of `X` because they have some awkward statistical properties, which are beyond the scope of this discussion.

```
load ionosphere
Ybool = strcmp(Y, 'g');
X = X(:,3:end);
```

### Step 2. Create a cross-validated fit.

Construct a regularized binomial regression using 25 Lambda values and 10-fold cross validation. This process can take a few minutes.

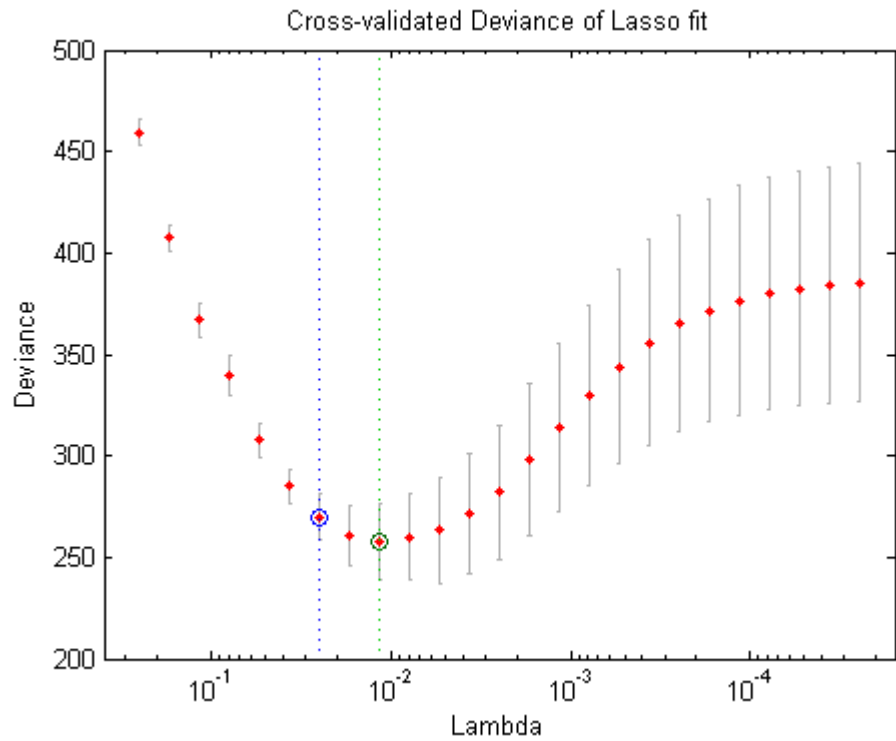
```
rng('default') % for reproducibility
[B,FitInfo] = lassoglm(X,Ybool,'binomial',...
    'NumLambda',25,'CV',10);
```

### Step 3. Examine plots to find appropriate regularization.

`lassoPlot` can give both a standard trace plot and a cross-validated deviance plot. Examine both plots.

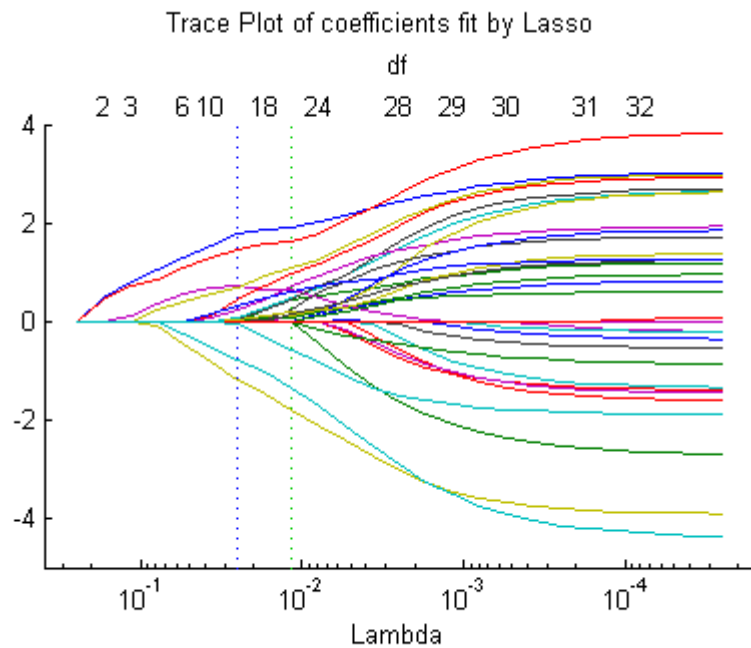
```
lassoPlot(B,FitInfo,'PlotType','CV');
```





The plot identifies the minimum-deviance point with a green circle and dashed line as a function of the regularization parameter  $\lambda$ . The blue circled point has minimum deviance plus no more than one standard deviation.

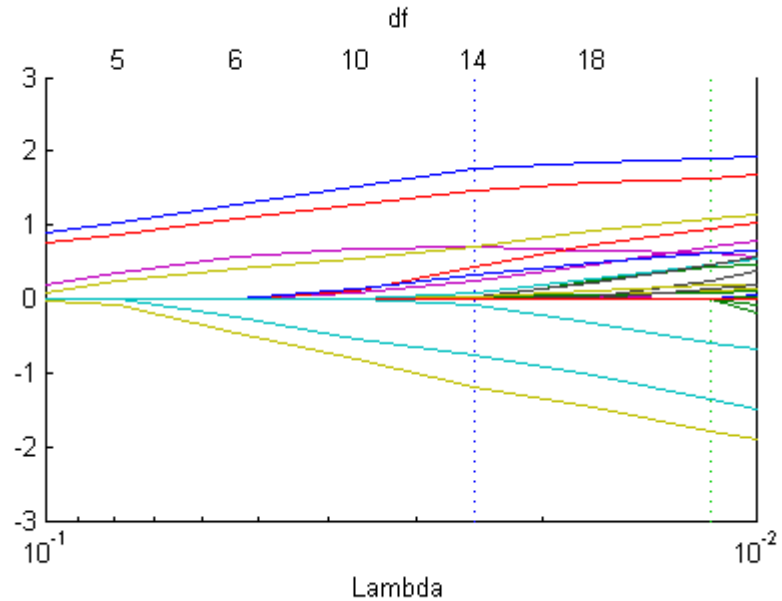
```
lassoPlot(B,FitInfo,'PlotType','Lambda','XScale','log');
```



The trace plot shows nonzero model coefficients as a function of the regularization parameter  $\text{Lambda}$ . Because there are 32 predictors and a linear model, there are 32 curves. As  $\text{Lambda}$  increases to the left, `lasso1m` sets various coefficients to zero, removing them from the model.

The trace plot is somewhat compressed. Zoom in to see more detail.

```
xlim([.01 .1])
ylim([-3 3])
```



As Lambda increases toward the left side of the plot, fewer nonzero coefficients remain.

Find the number of nonzero model coefficients at the Lambda value with minimum deviance plus one standard deviation point. The regularized model coefficients are in column `FitInfo.Index1SE` of the B matrix.

```
indx = FitInfo.Index1SE;
B0 = B(:,indx);
nonzeros = sum(B0 ~= 0)
```

```
nonzeros =
```

```
14
```

When you set `Lambda` to `FitInfo.Index1SE`, `lassoglm` removes over half of the 32 original predictors.

**Step 4. Create a regularized model.**

The constant term is in the `FitInfo.Index1SE` entry of the `FitInfo.Intercept` vector. Call that value `cnst`.

The model is  $\text{logit}(\mu) = \log(\mu/(1 - \mu))X*B0 + \text{cnst}$ . Therefore, for predictions,  $\mu = \exp(X*B0 + \text{cnst}) / (1 + \exp(x*B0 + \text{cnst}))$ .

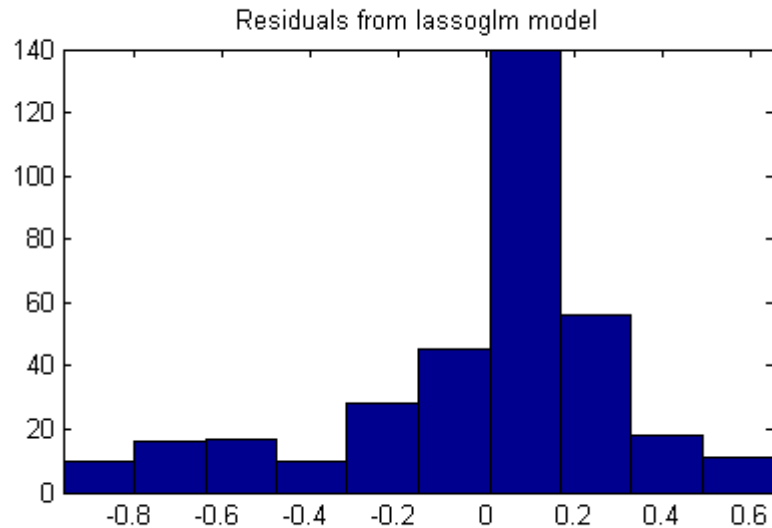
The `glmval` function evaluates model predictions. It assumes that the first model coefficient relates to the constant term. Therefore, create a coefficient vector with the constant term first.

```
cnst = FitInfo.Intercept(indx);  
B1 = [cnst;B0];
```

**Step 5. Examine residuals.**

Plot the training data against the model predictions for the regularized `lassoglm` model.

```
preds = glmval(B1,X,'logit');  
hist(Ybool - preds) % plot residuals  
title('Residuals from lassoglm model')
```



**Step 6. Alternative: Use identified predictors in a least-squares generalized linear model.**

Instead of using the biased predictions from the model, you can make an unbiased model using just the identified predictors.

```
predictors = find(B0); % indices of nonzero predictors
mdl = fitglm(X,Ybool,'linear',...
    'Distribution','binomial','PredictorVars',predictors)

mdl =
```

Generalized Linear regression model:

```
y ~ [Linear formula with 15 terms in 14 predictors]
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-2.9367	0.50926	-5.7666	8.0893e-09
x1	2.492	0.60795	4.099	4.1502e-05
x3	2.5501	0.63304	4.0284	5.616e-05
x4	0.48816	0.50336	0.9698	0.33215
x5	0.6158	0.62192	0.99015	0.3221
x6	2.294	0.5421	4.2317	2.3198e-05
x7	0.77842	0.57765	1.3476	0.1778
x12	1.7808	0.54316	3.2786	0.0010432
x16	-0.070993	0.50515	-0.14054	0.88823
x20	-2.7767	0.55131	-5.0365	4.7402e-07
x24	2.0212	0.57639	3.5067	0.00045372
x25	-2.3796	0.58274	-4.0835	4.4363e-05
x27	0.79564	0.55904	1.4232	0.15467
x29	1.2689	0.55468	2.2876	0.022162
x32	-1.5681	0.54336	-2.8859	0.0039035

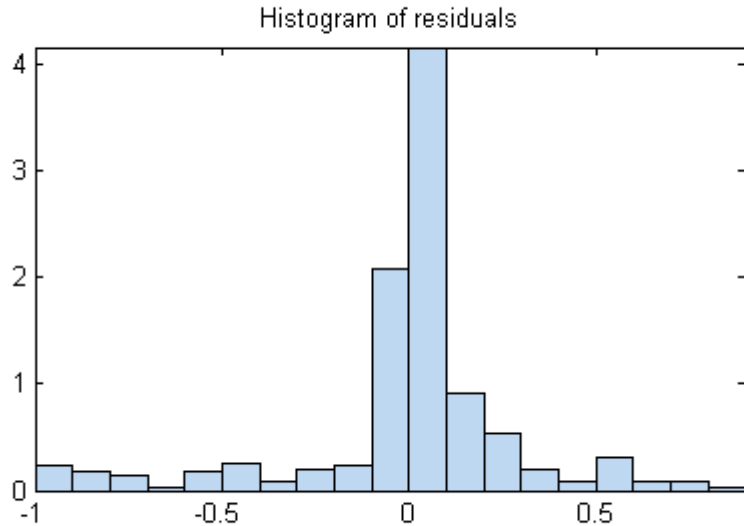
351 observations, 336 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 262, p-value = 1e-47

Plot the residuals of the model.

`plotResiduals(md1)`



As expected, residuals from the least-squares model are slightly smaller than those of the regularized model. However, this does not mean that `mdl` is a better predictor for new data.

## Regularize Wide Data in Parallel

This example shows how to regularize a model with many more predictors than observations. *Wide data* is data with more predictors than observations. Typically, with wide data you want to identify important predictors. Use `lasso glm` as an exploratory or screening tool to select a smaller set of variables to prioritize your modeling and research. Use parallel computing to speed up cross validation.

Load the `ovariancancer` data. This data has 216 observations and 4000 predictors in the `obs` workspace variable. The responses are binary, either

'Cancer' or 'Normal', in the grp workspace variable. Convert the responses to binary for use in `lassoglm`.

```
load ovariancancer
y = strcmp(grp,'Cancer');
```

Set options to use parallel computing. Prepare to compute in parallel using `parpool`.

```
opt = statset('UseParallel',true);
parpool()
```

Starting `parpool` using the 'local' profile ... connected to 2 workers.

```
ans =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}
NumWorkers: 2
IdleTimeout: 30
Cluster: [1x1 parallel.cluster.Local]
RequestQueue: [1x1 parallel.RequestQueue]
SpmEnabled: 1
```

Fit a cross-validated set of regularized models. Use the Alpha parameter to favor retaining groups of highly correlated predictors, as opposed to eliminating all but one member of the group. Commonly, you use a relatively large value of Alpha.

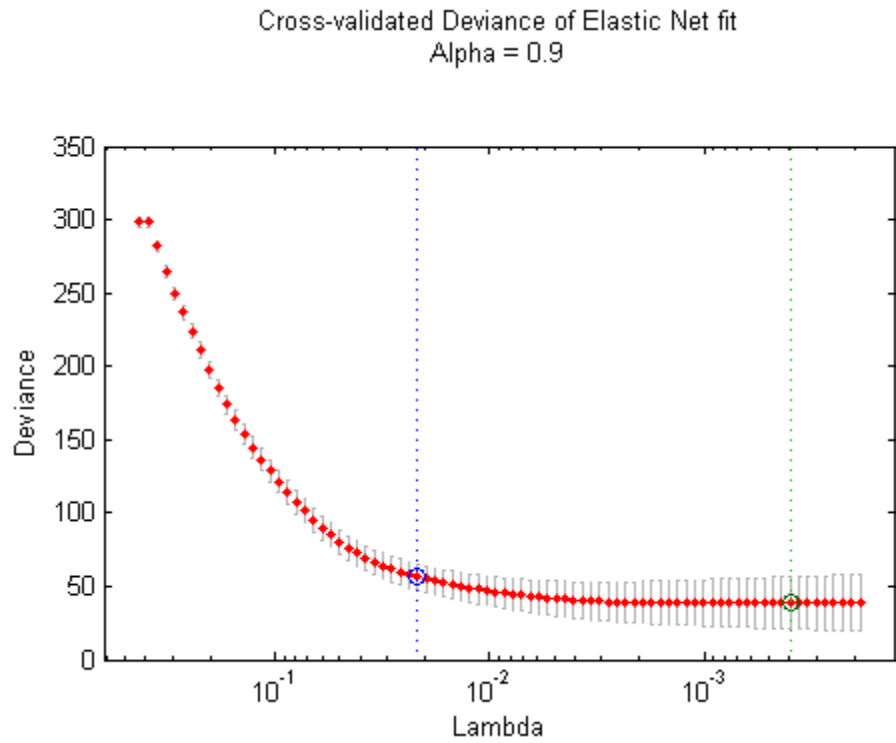
```
rng('default') % for reproducibility
tic
[B,S] = lassoglm(obs,y,'binomial','NumLambda',100, ...
'Alpha',0.9,'LambdaRatio',1e-4,'CV',10,'Options',opt);
toc
```

```
Elapsed time is 398.635386 seconds.
```

Examine cross-validation plot.

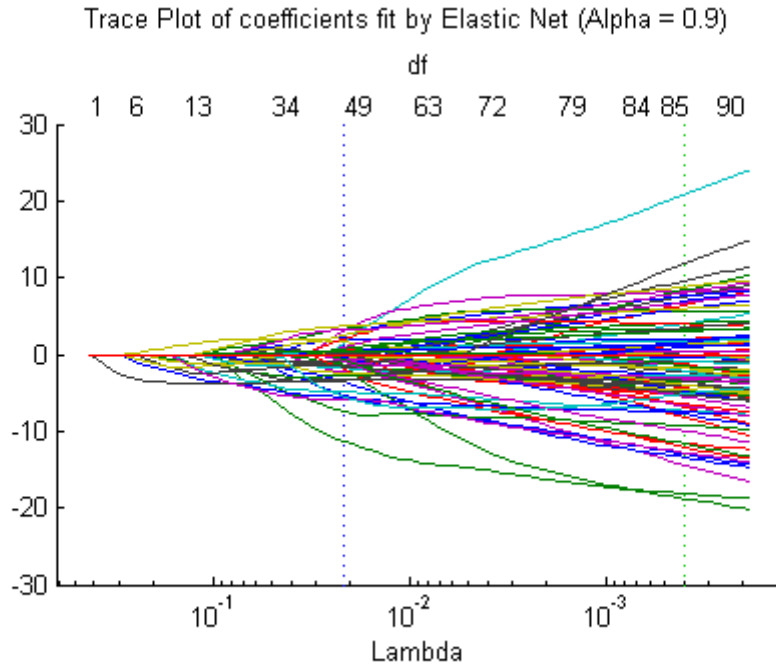
```
lassoPlot(B,S,'PlotType','CV');
```





Examine trace plot.

```
lassoPlot(B,S,'PlotType','Lambda','XScale','log')
```



The right (green) vertical dashed line represents the Lambda providing the smallest cross-validated deviance. The left (blue) dashed line has the minimal deviance plus no more than one standard deviation. This blue line has many fewer predictors:

```
[S.DF(S.Index1SE) S.DF(S.IndexMinDeviance)]
```

```
ans =
```

```
50 86
```

You asked `lasso1m` to fit using 100 different Lambda values. How many did it use?

```
size(B)
```

```
ans =
      4000      84
```

`lassoglm` stopped after 84 values because the deviance was too small for small Lambda values. To avoid overfitting, `lassoglm` halts when the deviance of the fitted model is too small compared to the deviance in the binary responses, ignoring the predictor variables.

You can force `lassoglm` to include more terms by explicitly providing a set of Lambda values.

```
minLambda = min(S.Lambda);
explicitLambda = [minLambda*.1 .01 .001] S.Lambda];
[B2,S2] = lassoglm(obs,y,'binomial','Lambda',explicitLambda,...
    'LambdaRatio',1e-4, 'CV',10,'Options',opt);
length(S2.Lambda)
```

```
ans =
      87
```

`lassoglm` used the three smaller values in fitting.

To save time, you can use:

- Fewer Lambda, meaning fewer fits
- Fewer cross-validation folds
- A larger value for LambdaRatio

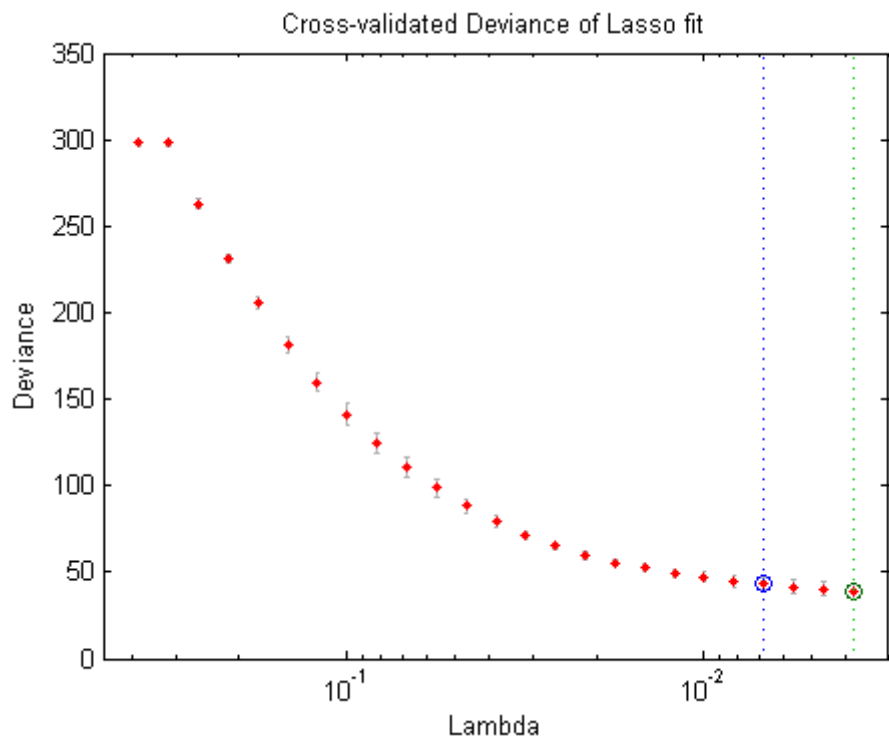
Use serial computation and all three of these time-saving methods:

```
tic
[Bquick,Squick] = lassoglm(obs,y,'binomial','NumLambda',25,...
    'LambdaRatio',1e-2,'CV',5);
toc
```

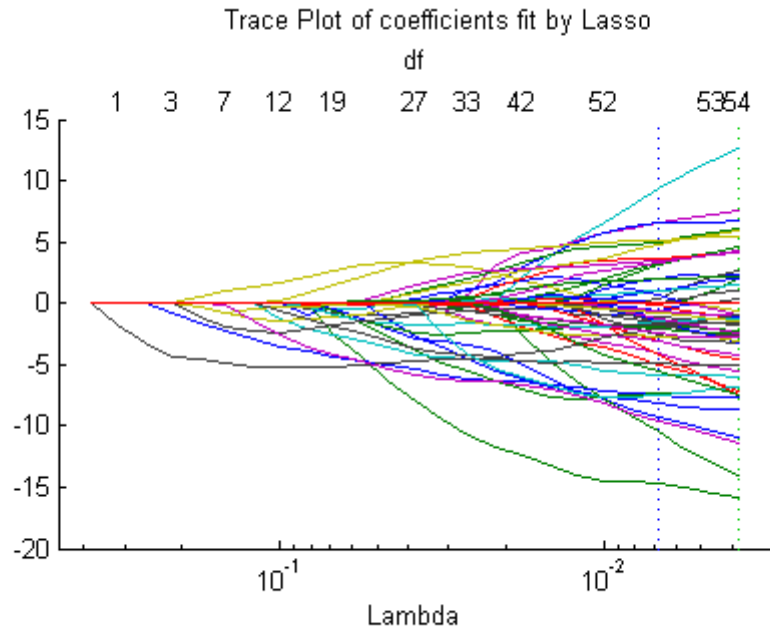
Elapsed time is 51.708074 seconds.

Graphically compare the new results to the first results.

```
lassoPlot(Bquick,Squick,'PlotType','CV');
```



```
lassoPlot(Bquick,Squick,'PlotType','Lambda','XScale','log')
```



The number of nonzero coefficients in the lowest plus one standard deviation model is around 50, similar to the first computation.

## Generalized Linear Model Lasso and Elastic Net

### Overview of Lasso and Elastic Net

*Lasso* is a regularization technique for estimating generalized linear models. Lasso includes a penalty term that constrains the size of the estimated coefficients. Therefore, it resembles ridge regression. Lasso is a *shrinkage estimator*: it generates coefficient estimates that are biased to be small. Nevertheless, a lasso estimator can have smaller error than an ordinary maximum likelihood estimator when you apply it to new data.

Unlike ridge regression, as the penalty term increases, the lasso technique sets more coefficients to zero. This means that the lasso estimator is a smaller

model, with fewer predictors. As such, lasso is an alternative to stepwise regression and other model selection and dimensionality reduction techniques.

*Elastic net* is a related technique. Elastic net is akin to a hybrid of ridge regression and lasso regularization. Like lasso, elastic net can generate reduced models by generating zero-valued coefficients. Empirical studies suggest that the elastic net technique can outperform lasso on data with highly correlated predictors.

### Definition of Lasso for Generalized Linear Models

For a nonnegative value of  $\lambda$ , lasso solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda \sum_{j=1}^p |\beta_j| \right),$$

where

- Deviance is the deviance of the model fit to the responses using intercept  $\beta_0$  and predictor coefficients  $\beta$ . The formula for Deviance depends on the `distr` parameter you supply to `lasso glm`. Minimizing the  $\lambda$ -penalized deviance is equivalent to maximizing the  $\lambda$ -penalized log likelihood.
- $N$  is the number of observations.
- $\lambda$  is a nonnegative regularization parameter corresponding to one value of `Lambda`.
- Parameters  $\beta_0$  and  $\beta$  are scalar and  $p$ -vector respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

### Definition of Elastic Net for Generalized Linear Models

For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right).$$

Elastic net is the same as lasso when  $\alpha = 1$ . For other values of  $\alpha$ , the penalty term  $P_\alpha(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ . As  $\alpha$  shrinks toward 0, elastic net approaches ridge regression.

## References

- [1] Tibshirani, R. *Regression Shrinkage and Selection via the Lasso*. Journal of the Royal Statistical Society, Series B, Vol. 58, No. 1, pp. 267–288, 1996.
- [2] Zou, H. and T. Hastie. *Regularization and Variable Selection via the Elastic Net*. Journal of the Royal Statistical Society, Series B, Vol. 67, No. 2, pp. 301–320, 2005.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. *Regularization Paths for Generalized Linear Models via Coordinate Descent*. Journal of Statistical Software, Vol. 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>
- [4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition. Springer, New York, 2008.
- [5] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*, 2nd edition. Chapman & Hall/CRC Press, 1989.





# Nonlinear Regression

---

- “Nonlinear Regression” on page 11-2
- “Mixed-Effects Models” on page 11-23
- “Pitfalls in Fitting Nonlinear Models by Transforming to Linearity” on page 11-61

## Nonlinear Regression

### In this section...

“What Are Parametric Nonlinear Regression Models?” on page 11-2

“Prepare Data” on page 11-3

“Represent the Nonlinear Model” on page 11-4

“Choose Initial Vector  $\beta_0$ ” on page 11-7

“Fit Nonlinear Model to Data” on page 11-7

“Examine Quality and Adjust the Fitted Model” on page 11-8

“Predict or Simulate Responses to New Data” on page 11-12

“Nonlinear Regression Workflow” on page 11-16

### What Are Parametric Nonlinear Regression Models?

Parametric nonlinear models represent the relationship between a continuous response variable and one or more continuous predictor variables in the form

$$y = f(X, \beta) + \varepsilon,$$

where

- $y$  is an  $n$ -by-1 vector of observations of the response variable.
- $f$  is any function of  $X$  and  $\beta$  that evaluates each row of  $X$  along with the vector  $\beta$  to compute the prediction for the corresponding row of  $y$ .
- $X$  is an  $n$ -by- $p$  matrix of predictors, with one row for each observation, and one column for each predictor.
- $\beta$  is a  $p$ -by-1 vector of unknown parameters to be estimated.
- $\varepsilon$  is an  $n$ -by-1 vector of independent, identically distributed random disturbances.

In contrast, nonparametric models do not attempt to characterize the relationship between predictors and response with model parameters. Descriptions are often graphical, as in the case of “Classification Trees and Regression Trees” on page 16-31.

`fitnlm` attempts to find values of the parameters  $\beta$  that minimize the mean squared differences between the observed responses  $y$  and the predictions of the model  $f(X,\beta)$ . To do so, it needs a starting value `beta0` before iteratively modifying the vector  $\beta$  to a vector with minimal mean squared error.

## Prepare Data

To begin fitting a regression, put your data into a form that fitting functions expect. All regression techniques begin with input data in an array  $X$  and response data in a separate vector  $y$ , or input data in a dataset array `ds` and response data as a column in `ds`. Each row of the input data represents one observation. Each column represents one predictor (variable).

For a dataset array `ds`, indicate the response variable with the 'ResponseVar' name-value pair:

```
mdl = fitlm(ds,'ResponseVar','BloodPressure');
```

The response variable is the last column by default.

You cannot use numeric *categorical* predictors for nonlinear regression. A categorical predictor is one that takes values from a fixed set of possibilities.

Represent missing data as NaN for both input data and response data.

## Dataset Array for Input and Response Data

For example, to create a dataset array from an Excel spreadsheet:

```
ds = dataset('XLSFile','hospital.xls',...
            'ReadObsNames',true);
```

To create a dataset array from workspace variables:

```
load carsmall
ds = dataset(Weight,Model_Year,MPG);
```

## Numeric Matrix for Input Data and Numeric Vector for Response

For example, to create numeric arrays from workspace variables:

```
load carsmall
X = [Weight Horsepower Cylinders Model_Year];
y = MPG;
```

To create numeric arrays from an Excel spreadsheet:

```
[X Xnames] = xlsread('hospital.xls');
y = X(:,4); % response y is systolic pressure
X(:,4) = []; % remove y from the X matrix
```

Notice that the nonnumeric entries, such as `sex`, do not appear in `X`.

## Represent the Nonlinear Model

There are several ways to represent a nonlinear model. Use whichever is most convenient.

The nonlinear model is a required input to `fitnlm`, in the `modelfun` input.

`fitnlm` assumes that the response function  $f(X, \beta)$  is smooth in the parameters  $\beta$ . If your function is not smooth, `fitnlm` can fail to provide optimal parameter estimates.

- “Function Handle to Anonymous Function or Function File” on page 11-4
- “String Representation of Formula” on page 11-5

### Function Handle to Anonymous Function or Function File

The function handle `@modelfun(b, x)` accepts a vector `b` and matrix or dataset array `x`. The function handle should return a vector `f` with the same number of rows as `x`. For example, the function file `hougen.m` computes

$$\text{hougen}(b, x) = \frac{b(1)x(2) - x(3) / b(5)}{1 + b(2)x(1) + b(3)x(2) + b(4)x(3)}.$$

Examine the function by entering type `hougen` at the MATLAB command line.

```
function yhat = hougen(beta, x)
%HOUGEN Hougen-Watson model for reaction kinetics.
```

```

%   YHAT = HOUGEN(BETA,X) gives the predicted values of the
%   reaction rate, YHAT, as a function of the vector of
%   parameters, BETA, and the matrix of data, X.
%   BETA must have 5 elements and X must have three
%   columns.
%
%   The model form is:
%    $y = (b1*x2 - x3/b5) ./ (1+b2*x1+b3*x2+b4*x3)$ 
%
%   Reference:
%   [1] Bates, Douglas, and Watts, Donald, "Nonlinear
%   Regression Analysis and Its Applications", Wiley
%   1988 p. 271-272.
%
%   Copyright 1993-2004 The MathWorks, Inc.
%   $Revision: 1.1.6.4.2.1 $ $Date: 2013/07/08 16:23:10 $
%   B.A. Jones 1-06-95.

```

```

b1 = beta(1);
b2 = beta(2);
b3 = beta(3);
b4 = beta(4);
b5 = beta(5);

```

```

x1 = x(:,1);
x2 = x(:,2);
x3 = x(:,3);

```

```

yhat = (b1*x2 - x3/b5) ./ (1+b2*x1+b3*x2+b4*x3);

```

You can write an anonymous function that performs the same calculation as `hougen.m`.

```

modelfun = @(b,x)(b(1)*x(:,2) - x(:,3)/b(5))./...
(1 + b(2)*x(:,1) + b(3)*x(:,2) + b(4)*x(:,3));

```

### String Representation of Formula

For data in a matrix  $X$  and response in a vector  $y$ :

- Represent the formula using 'x1' as the first predictor (column) in X, 'x2' as the second predictor, etc.
- Represent the vector of parameters to optimize as 'b1', 'b2', etc.
- Write the formula as 'y ~ (mathematical expressions)'.

For example, to represent the response to the reaction data:

```
modelfun = 'y ~ (b1*x2 - x3/b5)/(1 + b2*x1 + b3*x2 + b4*x3)';
```

For data in a dataset array, you can use formulas represented as strings with the variable names from the dataset array. Put the response variable name at the left of the formula, followed by a ~, followed by a string representing the response formula.

This example shows how to create a string to represent the response to the reaction data that is in a dataset array.

- 1 Load the reaction data.

```
load reaction
```

- 2 Put the data into a dataset array, where each variable has a name given in the xn or yn strings.

```
ds = dataset({reactants,xn(1,:),xn(2,:),xn(3,:)},...
            {rate,yn});
```

- 3 Examine the first row of the dataset array.

```
ds(1,:)
```

```
ans =
```

Hydrogen	n_Pentane	Isopentane	ReactionRate
470	300	10	8.55

- 4 Write the hougén formula using names in the dataset array.

```
modelfun = ['ReactionRate ~ (b1*n_Pentane - Isopentane/b5) /'...
            '(1 + Hydrogen*b2 + n_Pentane*b3 + Isopentane*b4)']
```

```

modelfun =
ReactionRate ~ (b1*n_Pentane - Isopentane/b5) / ...
              (1 + Hydrogen*b2 + n_Pentane*b3 + Isopentane*b4)

```

## Choose Initial Vector beta0

The initial vector for the fitting iterations, `beta0`, can greatly influence the quality of the resulting fitted model. `beta0` gives the dimensionality of the problem, meaning it needs the correct length. A good choice of `beta0` leads to a quick, reliable model, while a poor choice can lead to a long computation, or to an inadequate model.

It is difficult to give advice on choosing a good `beta0`. If you believe certain components of the vector should be positive or negative, set your `beta0` to have those characteristics. If you know the approximate value of other components, include them in `beta0`. However, if you don't know good values, try a random vector, such as

```

beta0 = randn(nVars,1);
% or
beta0 = 10*rand(nVars,1);

```

## Fit Nonlinear Model to Data

The syntax for fitting a nonlinear regression model using a dataset array `ds` is

```
mdl = fitnlm(ds,modelfun,beta0)
```

The syntax for fitting a nonlinear regression model using a numeric array `X` and numeric response vector `y` is

```
mdl = fitnlm(X,y,modelfun,beta0)
```

For information on representing the input parameters, see “Prepare Data” on page 11-3, “Represent the Nonlinear Model” on page 11-4, and “Choose Initial Vector `beta0`” on page 11-7.

`fitnlm` assumes that the response variable in a dataset array `ds` is the last column. To change this, use the `ResponseVar` name-value pair to name the response column.

## Examine Quality and Adjust the Fitted Model

There are diagnostic plots to help you examine the quality of a model.

- `plotDiagnostics mdl` gives a variety of plots, including leverage and Cook's distance plots.
- `plotResiduals mdl` gives the difference between the fitted model and the data.

There are also properties of `mdl` that relate to the model quality.

- `mdl.RMSE` gives the root mean square error between the data and the fitted model.
- `mdl.Residuals.Raw` gives the raw residuals.
- `mdl.Diagnostics` contains several fields, such as `Leverage` and `CooksDistance`, that can help you identify particularly interesting observations.

This example shows how to examine a fitted nonlinear model using diagnostic, residual, and slice plots.

- 1 Load the reaction data.

```
load reaction
```

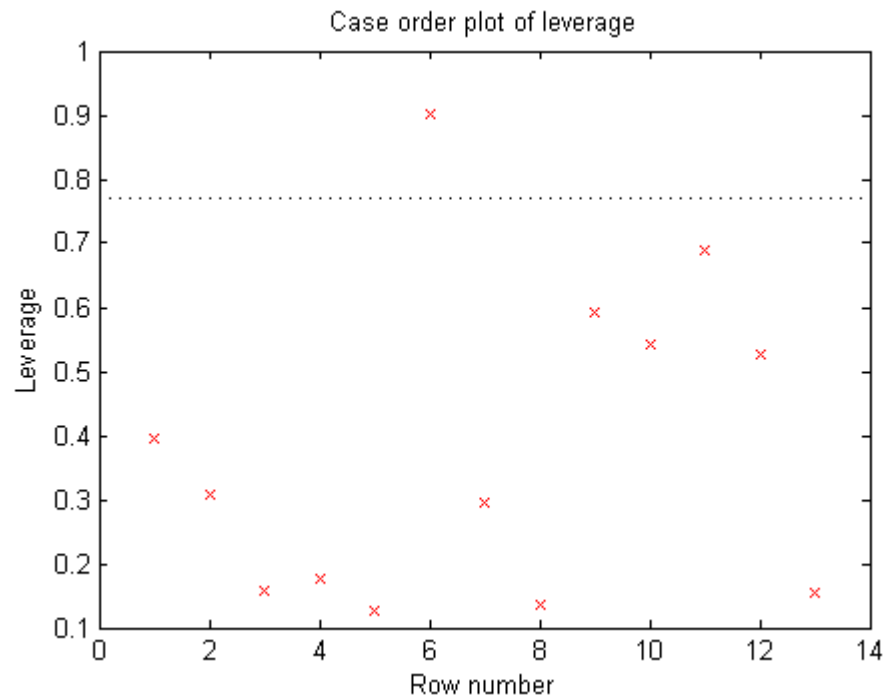
- 2 Create a nonlinear model of `rate` as a function of `reactants` using the `hougen.m` function, starting from `beta0 = ones(5,1);`.

```
beta0 = ones(5,1);  
mdl = fitnlm(reactants,...  
            rate,@hougen,beta0);
```

- 3 Make a leverage plot of the data and model.

```
plotDiagnostics(mdl)
```





**4** There is one point that has high leverage. Locate the point:

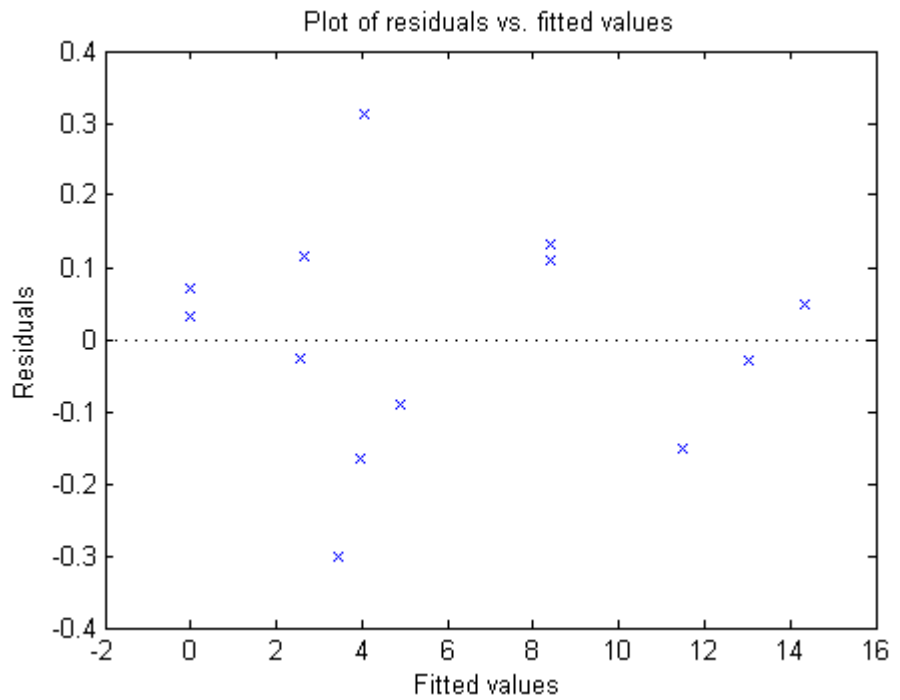
```
[-,max1] = max mdl.Diagnostics.Leverage)
```

```
max1 =
```

```
6
```

**5** Examine a residuals plot.

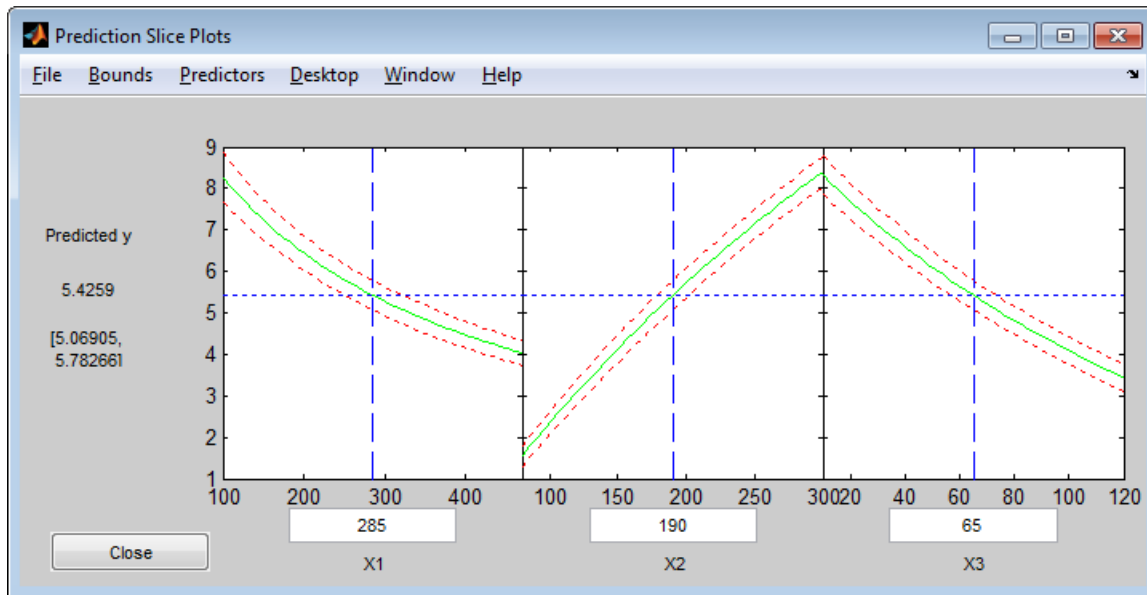
```
plotResiduals(mdl, 'fitted')
```



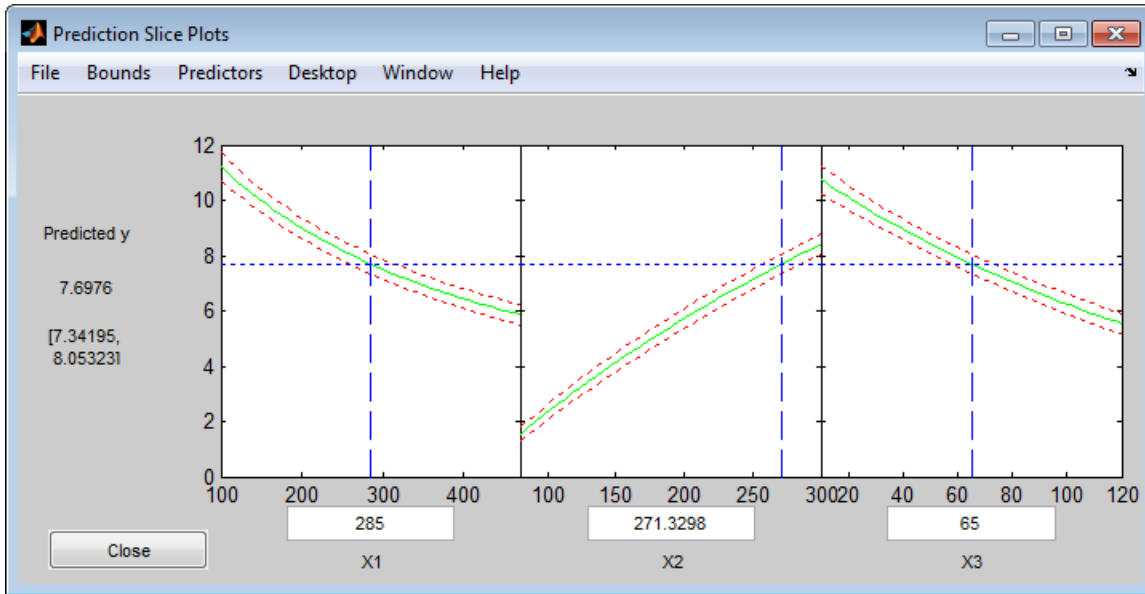
Nothing stands out as an outlier.

- 6 Use a slice plot to show the effect of each predictor on the model.

```
plotSlice mdl)
```



You can drag the vertical dashed blue lines to see the effect of a change in one predictor on the response. For example, drag the X2 line to the right, and notice that the slope of the X3 line changes.



## Predict or Simulate Responses to New Data

There are three ways to use a linear model to predict or simulate the response to new data:

- “predict” on page 11-14
- “feval” on page 11-15
- “random” on page 11-15

The prediction method examples use the following synthetic data and model. The problem is to fit a nonlinear curve of known form, with unknown center and scale parameters.

- 1 Generate synthetic data from a Cauchy distribution.

```
rng('default')
X = rand(100,1);
X = tan(pi*X - pi/2);
```

- 2 Generate the response according to the model

$$y = b_1 * (\pi/2 + \text{atan}((x - b_2) / b_3)).$$

Add noise to the response.

```
modelfun = @(b,x) b(1) * ...
    (pi/2 + atan((x - b(2))/b(3)));
y = modelfun([12 5 10],X) + randn(100,1);
```

**3** Fit a model starting from the arbitrary parameters  $b = [1,1,1]$ .

```
beta0 = [1 1 1]; % an arbitrary guess
mdl = fitnlm(X,y,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
y ~ b1*(pi/2 + atan((x - b2)/b3))
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	12.082	0.80028	15.097	3.3151e-27
b2	5.0603	1.0825	4.6747	9.5063e-06
b3	9.64	0.46499	20.732	2.0382e-37

```
Number of observations: 100, Error degrees of freedom: 97
```

```
Root Mean Squared Error: 1.02
```

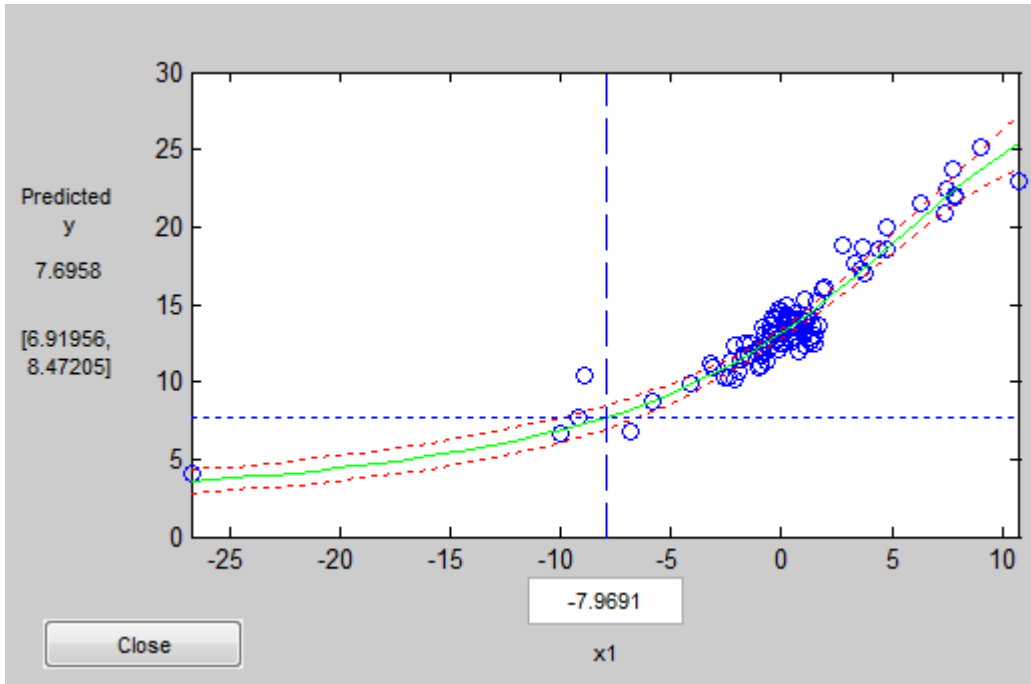
```
R-Squared: 0.92, Adjusted R-Squared 0.918
```

```
F-statistic vs. constant model: 558, p-value = 6.11e-54
```

The fitted values are within a few percent of the parameters [12,5,10].

**4** Examine the fit:

```
plotSlice(mdl)
```



## predict

The predict method predicts the mean responses and, if requested, gives confidence bounds. For example, to find the predicted response values and predicted confidence intervals about the response at X values [-15;5;12]:

```
Xnew = [-15;5;12];
[ynew,ynewci] = predict mdl,Xnew
```

```
ynew =
```

```
5.4122
18.9022
26.5161
```

```
ynewci =
```

```

    4.8233    6.0010
   18.4555   19.3490
   25.0170   28.0151

```

The confidence intervals are reflected in the slice plot.

## feval

The `feval` method predicts the mean responses. `feval` is often more convenient to use than `predict` when you construct a model from a dataset array. For example,

- 1 Create the nonlinear model from a dataset array.

```

ds = dataset({X,'X'},{y,'y'});
mdl2 = fitnlm(ds,modelfun,beta0);

```

- 2 Find the predicted model responses (CDF) at X values [-15;5;12].

```

Xnew = [-15;5;12];
ynew = feval(mdl2,Xnew)

```

```

ynew =

    5.4122
   18.9022
   26.5161

```

## random

The `random` method simulates new random response values, equal to the mean prediction plus a random disturbance with the same variance as the training data. For example,

```

Xnew = [-15;5;12];
ysim = random(mdl,Xnew)

```

```

ysim =

    6.0505
   19.0893
   25.4647

```

Rerun the random method. The results change.

```
ysim = random mdl, Xnew)
```

```
ysim =
```

```
6.3813
```

```
19.2157
```

```
26.6541
```

## Nonlinear Regression Workflow

This example shows how to do a typical nonlinear regression workflow: import data, fit a nonlinear regression, test its quality, modify it to improve the quality, and make predictions based on the model.

### Step 1. Prepare the data.

Load the reaction data

```
load reaction
```

Examine the data in the workspace. `reactants` is a matrix with 13 rows and 3 columns. Each row corresponds to one observation, and each column corresponds to one variable. The variable names are in `xn`:

```
xn
```

```
xn =
```

```
Hydrogen
```

```
n-Pentane
```

```
Isopentane
```

Similarly, `rate` is a vector of 13 responses, with the variable name in `yn`:

```
yn
```

```
yn =
```

```
Reaction Rate
```



The `hougen.m` file contains a nonlinear model of reaction rate as a function of the three predictor variables. For a 5-D vector  $b$  and 3-D vector  $x$ ,

$$\text{hougen}(b,x) = \frac{b(1)x(2) - x(3) / b(5)}{1 + b(2)x(1) + b(3)x(2) + b(4)x(3)}.$$

As a start point for the solution, take  $b$  as a vector of ones.

```
beta0 = ones(5,1);
```

### Step 2. Fit a nonlinear model to the data.

```
mdl = fitnlm(reactants,...
            rate,@hougen,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
y ~ hougen(b,X)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.2526	0.86702	1.4447	0.18654
b2	0.062776	0.043562	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075158	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

```
Number of observations: 13, Error degrees of freedom: 8
```

```
Root Mean Squared Error: 0.193
```

```
R-Squared: 0.999, Adjusted R-Squared 0.998
```

```
F-statistic vs. constant model: 1.81e+03, p-value = 7.36e-12
```

### Step 3. Examine the quality of the model.

The root mean squared error is fairly low compared to the range of observed values.

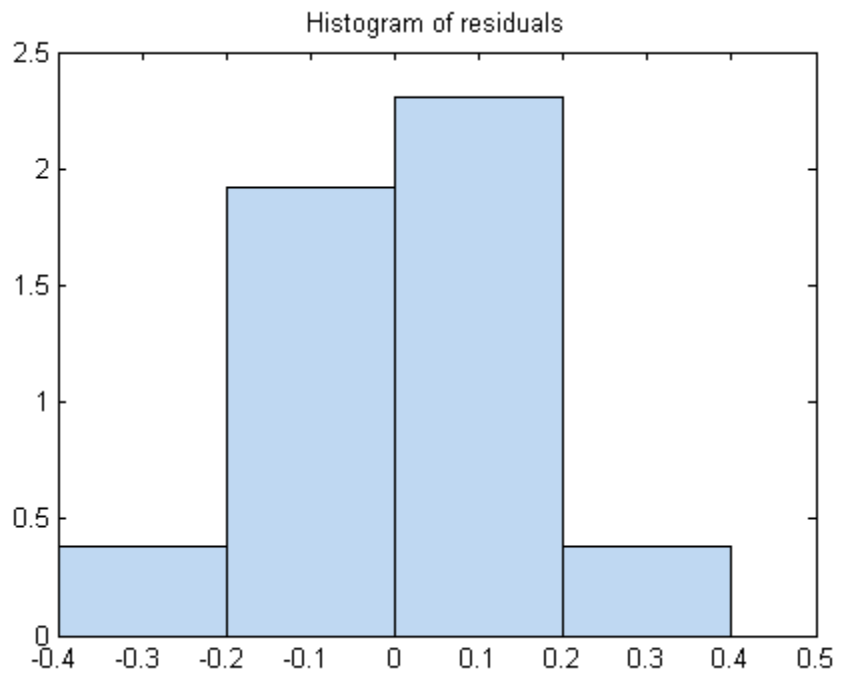
```
[mdl.RMSE min(rate) max(rate)]
```

```
ans =
```

0.1933 0.0200 14.3900

Examine a residuals plot.

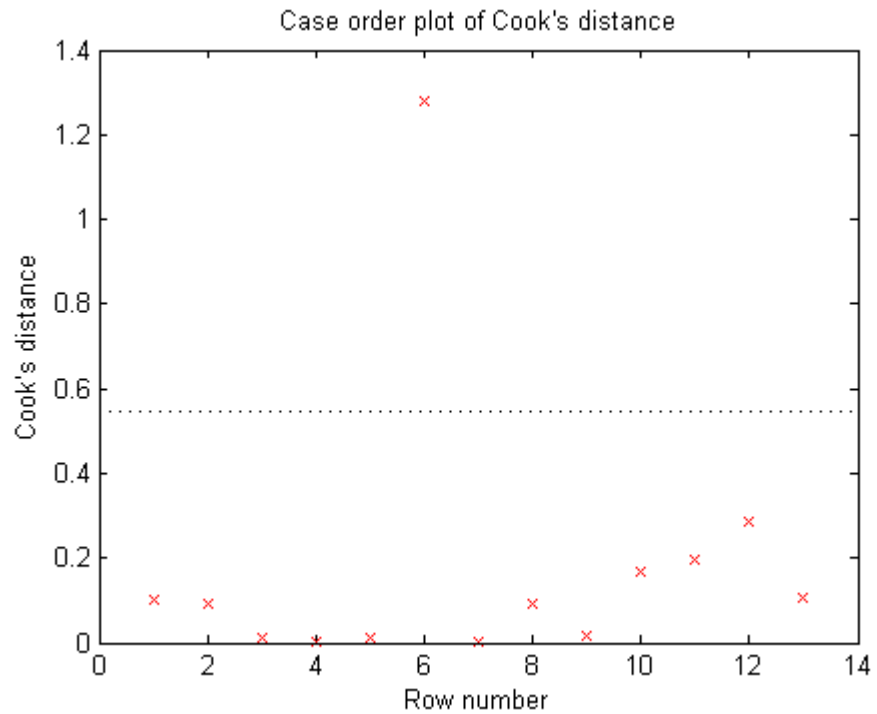
```
plotResiduals(md1)
```



The model seems adequate for the data.

Examine a diagnostic plot to look for outliers.

```
plotDiagnostics(md1, 'cookd')
```



Observation 6 seems out of line.

**Step 4. Remove the outlier.**

Remove the outlier from the fit using the `Exclude` name-value pair.

```
mdl1 = fitnlm(reactants,...
    rate,@hougen,ones(5,1),'Exclude',6)
```

```
mdl1 =
```

```
Nonlinear regression model:
y ~ hougen(b,X)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	0.619	0.4552	1.3598	0.21605

b2	0.030377	0.023061	1.3172	0.22924
b3	0.018927	0.01574	1.2024	0.26828
b4	0.053411	0.041084	1.3	0.23476
b5	2.4125	1.7903	1.3475	0.2198

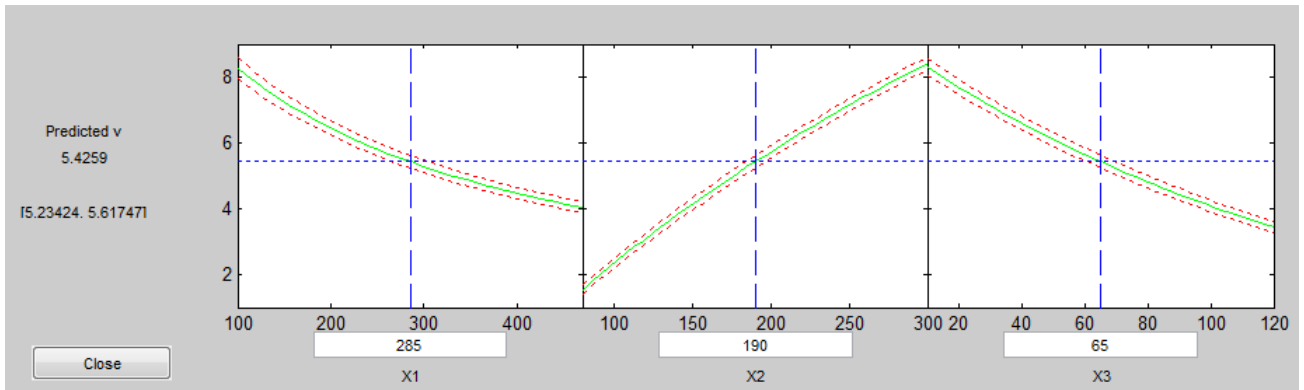
Number of observations: 12, Error degrees of freedom: 7  
 Root Mean Squared Error: 0.198  
 R-Squared: 0.999, Adjusted R-Squared 0.998  
 F-statistic vs. constant model: 1.24e+03, p-value = 4.73e-10

The model coefficients changed quite a bit from those in mdl.

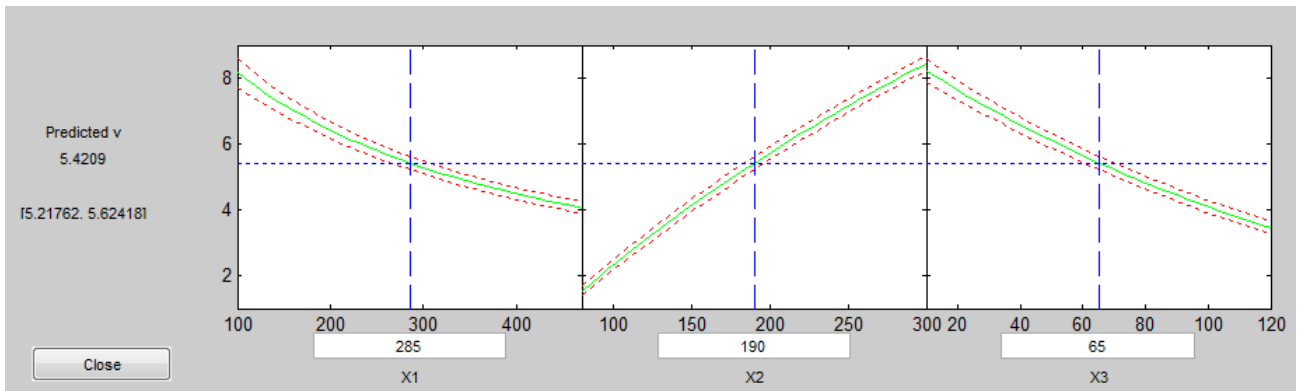
### Step 5. Examine slice plots of both models.

To see the effect of each predictor on the response, make a slice plot using `plotSlice(mdl)`.

```
plotSlice(mdl)
```



```
plotSlice(mdl1)
```



The plots look very similar, with slightly wider confidence bounds for mdl1. This difference is understandable, since there is one less data point in the fit, representing over 7% fewer observations.

#### Step 6. Predict for new data.

Create some new data and predict the response from both models.

```
Xnew = [200,200,200;100,200,100;500,50,5];
[ypred yci] = predict(mdl,Xnew)
```

```
ypred =
```

```
1.8762
6.2793
1.6718
```

```
yci =
```

```
1.6283    2.1242
5.9789    6.5797
1.5589    1.7846
```

```
[ypred1 yci1] = predict(mdl1,Xnew)
```

```
ypred1 =
```

1.8984  
6.2555  
1.6594

yci1 =

1.6260 2.1708  
5.9323 6.5787  
1.5345 1.7843

Even though the model coefficients are dissimilar, the predictions are nearly identical.

## Mixed-Effects Models

### In this section...

“Introduction to Mixed-Effects Models” on page 11-23

“Mixed-Effects Model Hierarchy” on page 11-24

“Specifying Mixed-Effects Models” on page 11-25

“Specifying Covariate Models” on page 11-28

“Choosing nlmeFit or nlmeFitsa” on page 11-30

“Using Output Functions with Mixed-Effects Models” on page 11-33

“Mixed-Effects Models Using nlmeFit and nlmeFitsa” on page 11-38

“Examining Residuals for Model Verification” on page 11-55

### Introduction to Mixed-Effects Models

In statistics, an *effect* is anything that influences the value of a response variable at a particular setting of the predictor variables. Effects are translated into model parameters. In linear models, effects become coefficients, representing the proportional contributions of model terms. In nonlinear models, effects often have specific physical interpretations, and appear in more general nonlinear combinations.

*Fixed effects* represent population parameters, assumed to be the same each time data is collected. Estimating fixed effects is the traditional domain of regression modeling. *Random effects*, by comparison, are sample-dependent random variables. In modeling, random effects act like additional error terms, and their distributions and covariances must be specified.

For example, consider a model of the elimination of a drug from the bloodstream. The model uses time  $t$  as a predictor and the concentration of the drug  $C$  as the response. The nonlinear model term  $C_0 e^{-rt}$  combines parameters  $C_0$  and  $r$ , representing, respectively, an initial concentration and an elimination rate. If data is collected across multiple individuals, it is reasonable to assume that the elimination rate is a random variable  $r_i$  depending on individual  $i$ , varying around a population mean  $\bar{r}$ . The term  $C_0 e^{-rt}$  becomes

$$C_0 e^{-[\bar{r} + (r_i - \bar{r})]t} = C_0 e^{-(\beta + b_i)t},$$

where  $\beta = \bar{r}$  is a fixed effect and  $b_i = r_i - \bar{r}$  is a random effect.

Random effects are useful when data falls into natural groups. In the drug elimination model, the groups are simply the individuals under study. More sophisticated models might group data by an individual's age, weight, diet, etc. Although the groups are not the focus of the study, adding random effects to a model extends the reliability of inferences beyond the specific sample of individuals.

*Mixed-effects models* account for both fixed and random effects. As with all regression models, their purpose is to describe a response variable as a function of the predictor variables. Mixed-effects models, however, recognize correlations within sample subgroups. In this way, they provide a compromise between ignoring data groups entirely and fitting each group with a separate model.

## Mixed-Effects Model Hierarchy

Suppose data for a nonlinear regression model falls into one of  $m$  distinct groups  $i = 1, \dots, m$ . To account for the groups in a model, write response  $j$  in group  $i$  as:

$$y_{ij} = f(\varphi, x_{ij}) + \varepsilon_{ij}$$

$y_{ij}$  is the response,  $x_{ij}$  is a vector of predictors,  $\varphi$  is a vector of model parameters, and  $\varepsilon_{ij}$  is the measurement or process error. The index  $j$  ranges from 1 to  $n_i$ , where  $n_i$  is the number of observations in group  $i$ . The function  $f$  specifies the form of the model. Often,  $x_{ij}$  is simply an observation time  $t_{ij}$ . The errors are usually assumed to be independent and identically, normally distributed, with constant variance.

Estimates of the parameters in  $\varphi$  describe the population, assuming those estimates are the same for all groups. If, however, the estimates vary by group, the model becomes

$$y_{ij} = f(\varphi_i, x_{ij}) + \varepsilon_{ij}$$



In a mixed-effects model,  $\varphi_i$  may be a combination of a fixed and a random effect:

$$\varphi_i = \beta + b_i$$

The random effects  $b_i$  are usually described as multivariate normally distributed, with mean zero and covariance  $\Psi$ . Estimating the fixed effects  $\beta$  and the covariance of the random effects  $\Psi$  provides a description of the population that does not assume the parameters  $\varphi_i$  are the same across groups. Estimating the random effects  $b_i$  also gives a description of specific groups within the data.

Model parameters do not have to be identified with individual effects. In general, *design matrices*  $A$  and  $B$  are used to identify parameters with linear combinations of fixed and random effects:

$$\varphi_i = A\beta + Bb_i$$

If the design matrices differ among groups, the model becomes

$$\varphi_i = A_i\beta + B_i b_i$$

If the design matrices also differ among observations, the model becomes

$$\begin{aligned}\varphi_{ij} &= A_{ij}\beta + B_{ij}b_i \\ y_{ij} &= f(\varphi_{ij}, x_{ij}) + \varepsilon_{ij}\end{aligned}$$

Some of the group-specific predictors in  $x_{ij}$  may not change with observation  $j$ . Calling those  $v_i$ , the model becomes

$$y_{ij} = f(\varphi_{ij}, x_{ij}, v_i) + \varepsilon_{ij}$$

## Specifying Mixed-Effects Models

Suppose data for a nonlinear regression model falls into one of  $m$  distinct groups  $i = 1, \dots, m$ . (Specifically, suppose that the groups are not nested.) To specify a general nonlinear mixed-effects model for this data:

- 1** Define group-specific model parameters  $\varphi_i$  as linear combinations of fixed effects  $\beta$  and random effects  $b_i$ .
- 2** Define response values  $y_i$  as a nonlinear function  $f$  of the parameters and group-specific predictor variables  $X_i$ .

The model is:

$$\begin{aligned}\varphi_i &= A_i\beta + B_ib_i \\ y_i &= f(\varphi_i, X_i) + \varepsilon_i \\ b_i &\square N(0, \Psi) \\ \varepsilon_i &\square N(0, \sigma^2)\end{aligned}$$

This formulation of the nonlinear mixed-effects model uses the following notation:

$\varphi_i$	A vector of group-specific model parameters
$\beta$	A vector of fixed effects, modeling population parameters
$b_i$	A vector of multivariate normally distributed group-specific random effects
$A_i$	A group-specific design matrix for combining fixed effects
$B_i$	A group-specific design matrix for combining random effects
$X_i$	A data matrix of group-specific predictor values
$y_i$	A data vector of group-specific response values
$f$	A general, real-valued function of $\varphi_i$ and $X_i$
$\varepsilon_i$	A vector of group-specific errors, assumed to be independent, identically, normally distributed, and independent of $b_i$
$\Psi$	A covariance matrix for the random effects
$\sigma^2$	The error variance, assumed to be constant across observations

For example, consider a model of the elimination of a drug from the bloodstream. The model incorporates two overlapping phases:

- An initial phase  $p$  during which drug concentrations reach equilibrium with surrounding tissues
- A second phase  $q$  during which the drug is eliminated from the bloodstream

For data on multiple individuals  $i$ , the model is

$$y_{ij} = C_{pi}e^{-r_{pi}t_{ij}} + C_{qi}e^{-r_{qi}t_{ij}} + \varepsilon_{ij},$$

where  $y_{ij}$  is the observed concentration in individual  $i$  at time  $t_{ij}$ . The model allows for different sampling times and different numbers of observations for different individuals.

The elimination rates  $r_{pi}$  and  $r_{qi}$  must be positive to be physically meaningful. Enforce this by introducing the log rates  $R_{pi} = \log(r_{pi})$  and  $R_{qi} = \log(r_{qi})$  and reparametrizing the model:

$$y_{ij} = C_{pi}e^{-\exp(R_{pi})t_{ij}} + C_{qi}e^{-\exp(R_{qi})t_{ij}} + \varepsilon_{ij}$$

Choosing which parameters to model with random effects is an important consideration when building a mixed-effects model. One technique is to add random effects to all parameters, and use estimates of their variances to determine their significance in the model. An alternative is to fit the model separately to each group, without random effects, and look at the variation of the parameter estimates. If an estimate varies widely across groups, or if confidence intervals for each group have minimal overlap, the parameter is a good candidate for a random effect.

To introduce fixed effects  $\beta$  and random effects  $b_i$  for all model parameters, reexpress the model as follows:

$$\begin{aligned} y_{ij} &= [\bar{C}_p + (C_{pi} - \bar{C}_p)]e^{-\exp[\bar{R}_p + (R_{pi} - \bar{R}_p)]t_{ij}} + \\ &\quad [\bar{C}_q + (C_{qi} - \bar{C}_q)]e^{-\exp[\bar{R}_q + (R_{qi} - \bar{R}_q)]t_{ij}} + \varepsilon_{ij} \\ &= (\beta_1 + b_{1i})e^{-\exp(\beta_2 + b_{2i})t_{ij}} + \\ &\quad (\beta_3 + b_{3i})e^{-\exp(\beta_4 + b_{4i})t_{ij}} + \varepsilon_{ij} \end{aligned}$$

In the notation of the general model:

$$\beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_4 \end{pmatrix}, b_i = \begin{pmatrix} b_{i1} \\ \vdots \\ b_{i4} \end{pmatrix}, y_i = \begin{pmatrix} y_{i1} \\ \vdots \\ y_{in_i} \end{pmatrix}, X_i = \begin{pmatrix} t_{i1} \\ \vdots \\ t_{in_i} \end{pmatrix},$$

where  $n_i$  is the number of observations of individual  $i$ . In this case, the design matrices  $A_i$  and  $B_i$  are, at least initially, 4-by-4 identity matrices. Design matrices may be altered, as necessary, to introduce weighting of individual effects, or time dependency.

Fitting the model and estimating the covariance matrix  $\Psi$  often leads to further refinements. A relatively small estimate for the variance of a random effect suggests that it can be removed from the model. Likewise, relatively small estimates for covariances among certain random effects suggests that a full covariance matrix is unnecessary. Since random effects are unobserved,  $\Psi$  must be estimated indirectly. Specifying a diagonal or block-diagonal covariance pattern for  $\Psi$  can improve convergence and efficiency of the fitting algorithm.

Statistics Toolbox functions `nlmefit` and `nlmefitsa` fit the general nonlinear mixed-effects model to data, estimating the fixed and random effects. The functions also estimate the covariance matrix  $\Psi$  for the random effects. Additional diagnostic outputs allow you to assess tradeoffs between the number of model parameters and the goodness of fit.

## Specifying Covariate Models

If the model in “Specifying Mixed-Effects Models” on page 11-25 assumes a group-dependent covariate such as weight ( $w$ ) the model becomes:

$$\begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & w_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Thus, the parameter  $\varphi_i$  for any individual in the  $i$ th group is:

$$\begin{pmatrix} \varphi_{1_i} \\ \varphi_{2_i} \\ \varphi_{3_i} \end{pmatrix} = \begin{pmatrix} \beta_1 + \beta_4 * w_i \\ \beta_2 \\ \beta_3 \end{pmatrix} + \begin{pmatrix} b_{1_i} \\ b_{2_i} \\ b_{3_i} \end{pmatrix}$$

To specify a covariate model, use the 'FEGroupDesign' option.

'FEGroupDesign' is a p-by-q-by-m array specifying a different p-by-q fixed-effects design matrix for each of the m groups. Using the previous example, the array resembles the following:

$$\begin{bmatrix} \begin{bmatrix} 1 & 0 & 0 & W_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 & 0 & W_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 & 0 & W_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\ \dots & \dots \\ \begin{bmatrix} 1 & 0 & 0 & W_m \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{bmatrix}$$

**1** Create the array.

```
% Number of parameters in the model (Phi)
num_params = 3;
% Number of covariates
num_cov = 1;
% Assuming number of groups in the data set is 7
num_groups = 7;
% Array of covariate values
covariates = [75; 52; 66; 55; 70; 58; 62 ];
A = repmat(eye(num_params, num_params+num_cov), ...
[1,1,num_groups]);
A(1,num_params+1,1:num_groups) = covariates(:,1)
```

**2** Create a struct with the specified design matrix.

```
options.FEGroupDesign = A;
```

- 3 Specify the arguments for `nlmefit` (or `nlmefitsa`) as shown in “Mixed-Effects Models Using `nlmefit` and `nlmefitsa`” on page 11-38.

## Choosing `nlmefit` or `nlmefitsa`

Statistics Toolbox provides two functions, `nlmefit` and `nlmefitsa` for fitting nonlinear mixed-effects models. Each function provides different capabilities, which may help you decide which to use.

- “Approximation Methods” on page 11-30
- “Parameters Specific to `nlmefitsa`” on page 11-31
- “Model and Data Requirements” on page 11-32

## Approximation Methods

`nlmefit` provides the following four approximation methods for fitting nonlinear mixed-effects models:

- 'LME' — Use the likelihood for the linear mixed-effects model at the current conditional estimates of `beta` and `B`. This is the default.
- 'RELME' — Use the restricted likelihood for the linear mixed-effects model at the current conditional estimates of `beta` and `B`.
- 'FO' — First-order Laplacian approximation without random effects.
- 'FOCE' — First-order Laplacian approximation at the conditional estimates of `B`.

`nlmefitsa` provides an additional approximation method, Stochastic Approximation Expectation-Maximization (SAEM) [24] with three steps :

- 1 Simulation: Generate simulated values of the random effects  $b$  from the posterior density  $p(b | \Sigma)$  given the current parameter estimates.
- 2 Stochastic approximation: Update the expected value of the log likelihood function by taking its value from the previous step, and moving part way toward the average value of the log likelihood calculated from the simulated random effects.

- 3 Maximization step: Choose new parameter estimates to maximize the log likelihood function given the simulated values of the random effects.

Both `nlmefit` and `nlmefitsa` attempt to find parameter estimates to maximize a likelihood function, which is difficult to compute. `nlmefit` deals with the problem by approximating the likelihood function in various ways, and maximizing the approximate function. It uses traditional optimization techniques that depend on things like convergence criteria and iteration limits.

`nlmefitsa`, on the other hand, simulates random values of the parameters in such a way that in the long run they converge to the values that maximize the exact likelihood function. The results are random, and traditional convergence tests don't apply. Therefore `nlmefitsa` provides options to plot the results as the simulation progresses, and to restart the simulation multiple times. You can use these features to judge whether the results have converged to the accuracy you desire.

### Parameters Specific to `nlmefitsa`

The following parameters are specific to `nlmefitsa`. Most control the stochastic algorithm.

- `Cov0` — Initial value for the covariance matrix `PSI`. Must be an  $r$ -by- $r$  positive definite matrix. If empty, the default value depends on the values of `BETA0`.
- `ComputeStdErrors` — `true` to compute standard errors for the coefficient estimates and store them in the output `STATS` structure, or `false` (default) to omit this computation.
- `LogLikMethod` — Specifies the method for approximating the log likelihood.
- `NBurnIn` — Number of initial burn-in iterations during which the parameter estimates are not recomputed. Default is 5.
- `NIterations` — Controls how many iterations are performed for each of three phases of the algorithm.
- `NMCMCIterations` — Number of Markov Chain Monte Carlo (MCMC) iterations.

## Model and Data Requirements

There are some differences in the capabilities of `nlmefit` and `nlmefitsa`. Therefore some data and models are usable with either function, but some may require you to choose just one of them.

- **Error models** — `nlmefitsa` supports a variety of error models. For example, the standard deviation of the response can be constant, proportional to the function value, or a combination of the two. `nlmefit` fits models under the assumption that the standard deviation of the response is constant. One of the error models, 'exponential', specifies that the log of the response has a constant standard deviation. You can fit such models using `nlmefit` by providing the log response as input, and by rewriting the model function to produce the log of the nonlinear function value.
- **Random effects** — Both functions fit data to a nonlinear function with parameters, and the parameters may be simple scalar values or linear functions of covariates. `nlmefit` allows any coefficients of the linear functions to have both fixed and random effects. `nlmefitsa` supports random effects only for the constant (intercept) coefficient of the linear functions, but not for slope coefficients. So in the example in “Specifying Covariate Models” on page 11-28, `nlmefitsa` can treat only the first three beta values as random effects.
- **Model form** — `nlmefit` supports a very general model specification, with few restrictions on the design matrices that relate the fixed coefficients and the random effects to the model parameters. `nlmefitsa` is more restrictive:
  - The fixed effect design must be constant in every group (for every individual), so an observation-dependent design is not supported.
  - The random effect design must be constant for the entire data set, so neither an observation-dependent design nor a group-dependent design is supported.
  - As mentioned under **Random Effects**, the random effect design must not specify random effects for slope coefficients. This implies that the design must consist of zeros and ones.
  - The random effect design must not use the same random effect for multiple coefficients, and cannot use more than one random effect for any single coefficient.



- The fixed effect design must not use the same coefficient for multiple parameters. This implies that it can have at most one nonzero value in each column.

If you want to use `nlmefitsa` for data in which the covariate effects are random, include the covariates directly in the nonlinear model expression. Don't include the covariates in the fixed or random effect design matrices.

- **Convergence** — As described in the **Model form**, `nlmefit` and `nlmefitsa` have different approaches to measuring convergence. `nlmefit` uses traditional optimization measures, and `nlmefitsa` provides diagnostics to help you judge the convergence of a random simulation.

In practice, `nlmefitsa` tends to be more robust, and less likely to fail on difficult problems. However, `nlmefit` may converge faster on problems where it converges at all. Some problems may benefit from a combined strategy, for example by running `nlmefitsa` for a while to get reasonable parameter estimates, and using those as a starting point for additional iterations using `nlmefit`.

## Using Output Functions with Mixed-Effects Models

The `OutputFcn` field of the options structure specifies one or more functions that the solver calls after each iteration. Typically, you might use an output function to plot points at each iteration or to display optimization quantities from the algorithm. To set up an output function:

- 1 Write the output function as a MATLAB file function or local function.
- 2 Use `statset` to set the value of `OutputFcn` to be a function handle, that is, the name of the function preceded by the `@` sign. For example, if the output function is `outfun.m`, the command

```
options = statset('OutputFcn', @outfun);
```

specifies `OutputFcn` to be the handle to `outfun`. To specify multiple output functions, use the syntax:

```
options = statset('OutputFcn',{@outfun, @outfun2});
```

- 3 Call the optimization function with options as an input argument.

For an example of an output function, see “Sample Output Function” on page 11-38.

## Structure of the Output Function

The function definition line of the output function has the following form:

```
stop = outfun(beta,status,state)
```

where

- *beta* is the current fixed effects.
- *status* is a structure containing data from the current iteration. “Fields in status” on page 11-34 describes the structure in detail.
- *state* is the current state of the algorithm. “States of the Algorithm” on page 11-35 lists the possible values.
- *stop* is a flag that is true or false depending on whether the optimization routine should quit or continue. See “Stop Flag” on page 11-36 for more information.

The solver passes the values of the input arguments to `outfun` at each iteration.

## Fields in status

The following table lists the fields of the `status` structure:

Field	Description
procedure	<ul style="list-style-type: none"> <li>• 'ALT' — alternating algorithm for the optimization of the linear mixed effects or restricted linear mixed effects approximations</li> <li>• 'LAP' — optimization of the Laplacian approximation for first order or first order conditional estimation</li> </ul>
iteration	An integer starting from 0.

Field	Description
inner	<p>A structure describing the status of the inner iterations within the ALT and LAP procedures, with the fields:</p> <ul style="list-style-type: none"> <li>• procedure — When procedure is 'ALT': <ul style="list-style-type: none"> <li>▪ 'PNLS' (penalized nonlinear least squares)</li> <li>▪ 'LME' (linear mixed-effects estimation)</li> <li>▪ 'none'</li> </ul> When procedure is 'LAP', <ul style="list-style-type: none"> <li>▪ 'PNLS' (penalized nonlinear least squares)</li> <li>▪ 'PLM' (profiled likelihood maximization)</li> <li>▪ 'none'</li> </ul> </li> <li>• state — one of the following: <ul style="list-style-type: none"> <li>▪ 'init'</li> <li>▪ 'iter'</li> <li>▪ 'done'</li> <li>▪ 'none'</li> </ul> </li> <li>• iteration — an integer starting from 0, or NaN. For nlme fits with burn-in iterations, the output function is called after each of those iterations with a negative value for STATUS.iteration.</li> </ul>
fval	The current log likelihood
Psi	The current random-effects covariance matrix
theta	The current parameterization of Psi
mse	The current error variance

### States of the Algorithm

The following table lists the possible values for state:

state	Description
'init'	The algorithm is in the initial state before the first iteration.
'iter'	The algorithm is at the end of an iteration.
'done'	The algorithm is in the final state after the last iteration.

The following code illustrates how the output function might use the value of `state` to decide which tasks to perform at the current iteration:

```
switch state
    case 'iter'
        % Make updates to plot or guis as needed
    case 'init'
        % Setup for plots or guis
    case 'done'
        % Cleanup of plots, guis, or final plot
otherwise
end
```

### Stop Flag

The output argument `stop` is a flag that is true or false. The flag tells the solver whether it should quit or continue. The following examples show typical ways to use the stop flag.

**Stopping an Optimization Based on Intermediate Results.** The output function can stop the estimation at any iteration based on the values of arguments passed into it. For example, the following code sets `stop` to true based on the value of the log likelihood stored in the `'fval'` field of the status structure:

```
stop = outfun(beta,status,state)
stop = false;
% Check if loglikelihood is more than 132.
if status.fval > -132
    stop = true;
end
```

**Stopping an Iteration Based on GUI Input.** If you design a GUI to perform `nlmefit` iterations, you can make the output function stop when a user clicks a **Stop** button on the GUI. For example, the following code implements a dialog to cancel calculations:

```
function retval = stop_outfcn(beta,str,status)
persistent h stop;
if isequal(str.inner.state,'none')
    switch(status)
        case 'init'
            % Initialize dialog
            stop = false;
            h = msgbox('Press STOP to cancel calculations.',...
                'NLMEFIT: Iteration 0 ');
            button = findobj(h,'type','uicontrol');
            set(button,'String','STOP','Callback',@stopper)
            pos = get(h,'Position');
            pos(3) = 1.1 * pos(3);
            set(h,'Position',pos)
            drawnow
        case 'iter'
            % Display iteration number in the dialog title
            set(h,'Name',sprintf('NLMEFIT: Iteration %d',...
                str.iteration))
            drawnow;
        case 'done'
            % Delete dialog
            delete(h);
    end
end
if stop
    % Stop if the dialog button has been pressed
    delete(h)
end
retval = stop;

function stopper(varargin)
    % Set flag to stop when button is pressed
    stop = true;
    disp('Calculation stopped.')
```

```
        end
    end
```

### Sample Output Function

`nlmefitoutputfcn` is the sample Statistics Toolbox output function for `nlmefit` and `nlmefitsa`. It initializes or updates a plot with the fixed-effects (BETA) and variance of the random effects (`diag(STATUS.Psi)`). For `nlmefit`, the plot also includes the log-likelihood (`STATUS.fval`).

`nlmefitoutputfcn` is the default output function for `nlmefitsa`. To use it with `nlmefit`, specify a function handle for it in the options structure:

```
opt = statset('OutputFcn', @nlmefitoutputfcn, )
beta = nlmefit( , 'Options', opt, )
```

To prevent `nlmefitsa` from using of this function, specify an empty value for the output function:

```
opt = statset('OutputFcn', [], )
beta = nlmefitsa( , 'Options', opt, )
```

`nlmefitoutputfcn` stops `nlmefit` or `nlmefitsa` if you close the figure that it produces.

## Mixed-Effects Models Using `nlmefit` and `nlmefitsa`

Load the sample data.

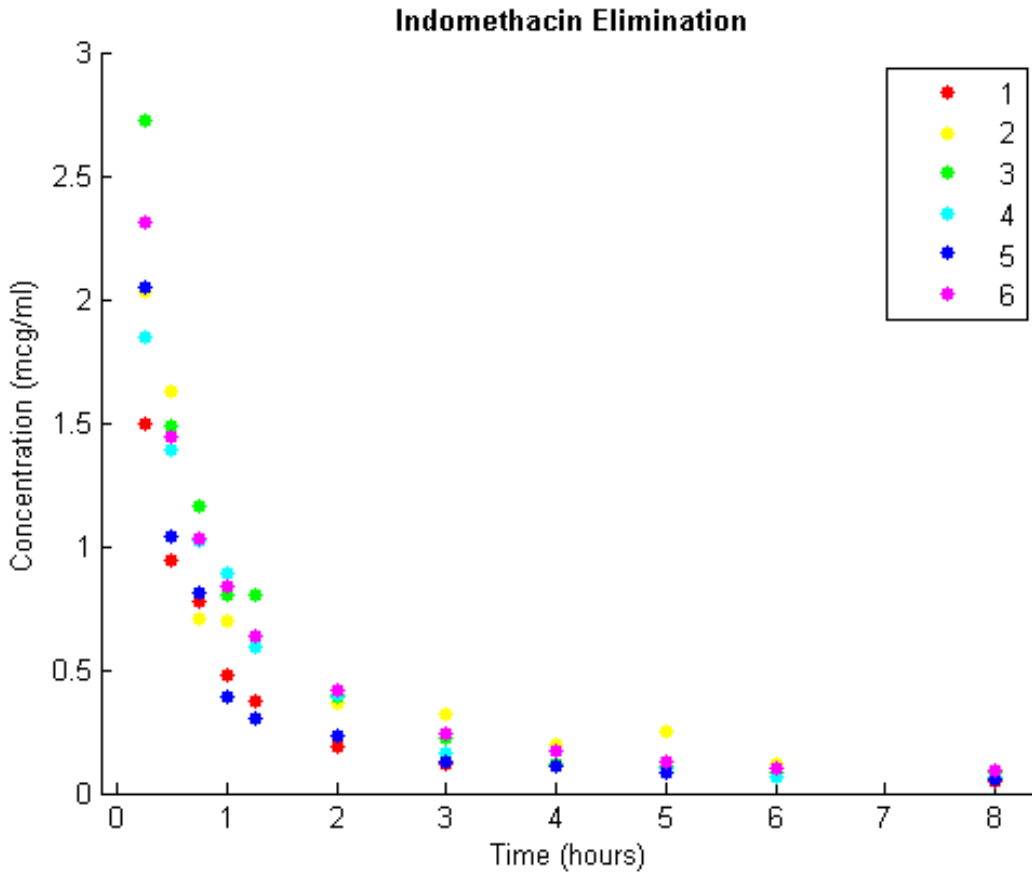
```
load indomethacin
```

The data in `indomethacin.mat` records concentrations of the drug indomethacin in the bloodstream of six subjects over eight hours.

Plot the scatter plot of indomethacin in the bloodstream grouped by subject.

```
gscatter(time, concentration, subject)
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('{\bf Indomethacin Elimination}')
```

hold on



Specifying Mixed-Effects Models page discusses a useful model for this type of data.

Construct the model via an anonymous function.

```
model = @(phi,t)(phi(1)*exp(-exp(phi(2))*t) + ...
              phi(3)*exp(-exp(phi(4))*t));
```

Use the `nlinfit` function to fit the model to all of the data, ignoring subject-specific effects.

```
phi0 = [1 2 1 1];  
[phi,res] = nlinfit(time,concentration,model,phi0);
```

Compute the mean squared error.

```
numObs = length(time);  
numParams = 4;  
df = numObs-numParams;  
mse = (res'*res)/df
```

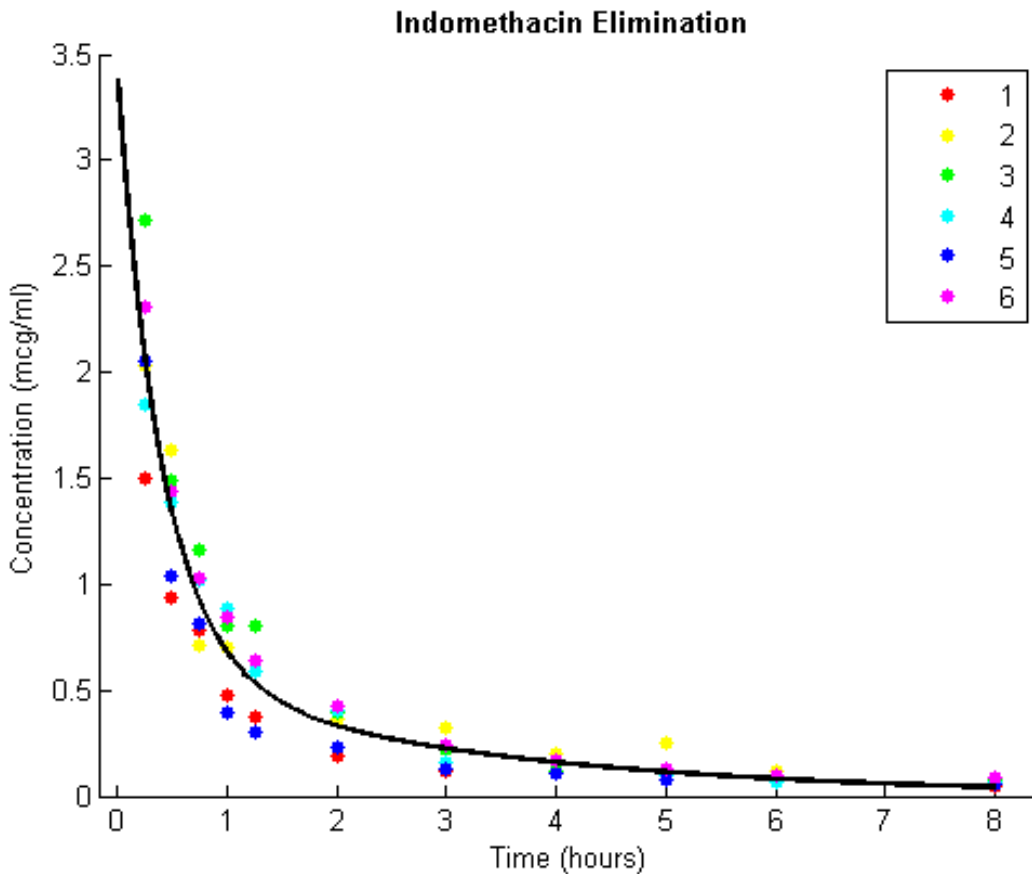
```
mse =
```

```
0.0304
```

Super impose the model on the scatter plot of data.

```
tplot = 0:0.01:8;  
plot(tplot,model(phi,tplot),'k','LineWidth',2)  
hold off
```



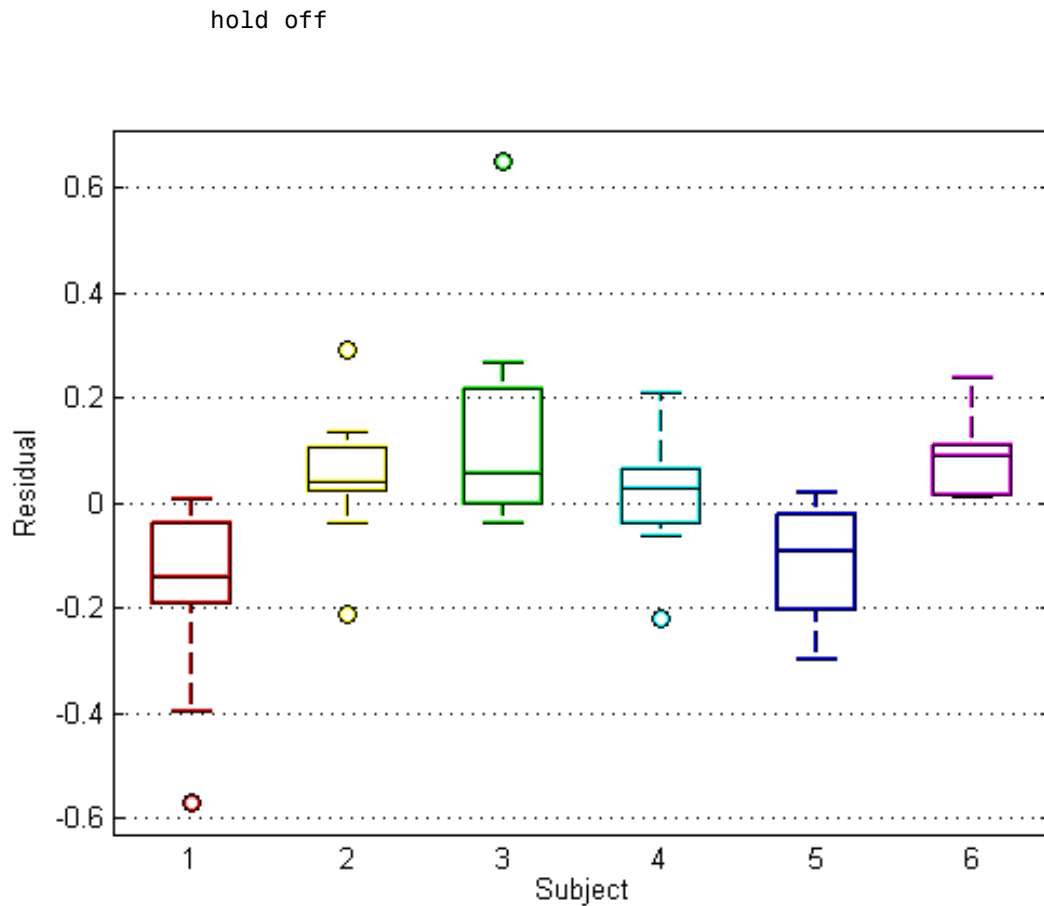


Draw the box-plot of residuals by subject.

```

colors = 'rygcbm';
h = boxplot(res,subject,'colors',colors,'symbol','o');
set(h(~isnan(h)),'LineWidth',2)
hold on
boxplot(res,subject,'colors','k','symbol','ko')
grid on
xlabel('Subject')
ylabel('Residual')

```



The box plot of residuals by subject shows that the boxes are mostly above or below zero, indicating that the model has failed to account for subject-specific effects.

To account for subject-specific effects, fit the model separately to the data for each subject.

```
phi0 = [1 2 1 1];  
PHI = zeros(4,6);
```

```

RES = zeros(11,6);
for I = 1:6
    tI = time(subject == I);
    cI = concentration(subject == I);
    [PHI(:,I),RES(:,I)] = nlinfit(tI,cI,model,phi0);
end
PHI

```

PHI =

2.0293	2.8277	5.4683	2.1981	3.5661	3.0023
0.5794	0.8013	1.7498	0.2423	1.0408	1.0882
0.1915	0.4989	1.6757	0.2545	0.2915	0.9685
-1.7878	-1.6354	-0.4122	-1.6026	-1.5069	-0.8731

Compute the mean squared error.

```

numParams = 24;
df = numObs-numParams;
mse = (RES(:)'*RES(:))/df

```

mse =

0.0057

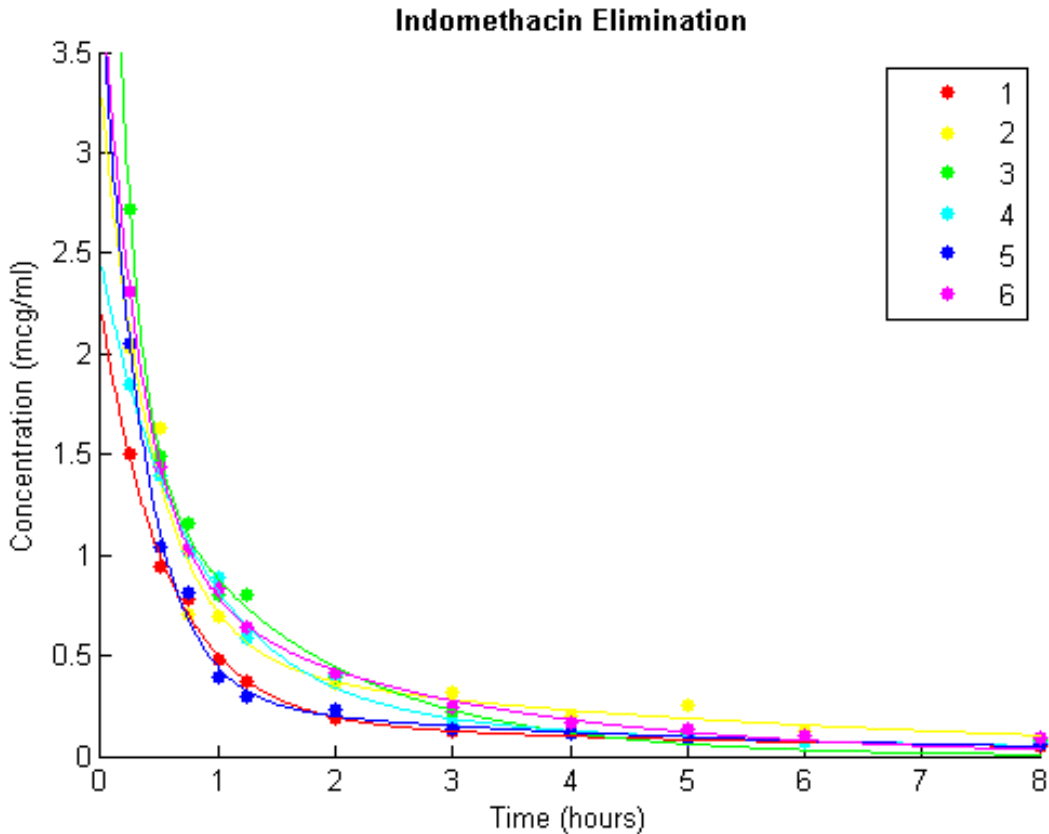
Plot the scatter plot of the data and superimpose the model for each subject.

```

gscatter(time,concentration,subject)
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('{\bf Indomethacin Elimination}')
hold on
for I = 1:6
    plot(tplot,model(PHI(:,I),tplot),'Color',colors(I))
end
axis([0 8 0 3.5])

```

hold off

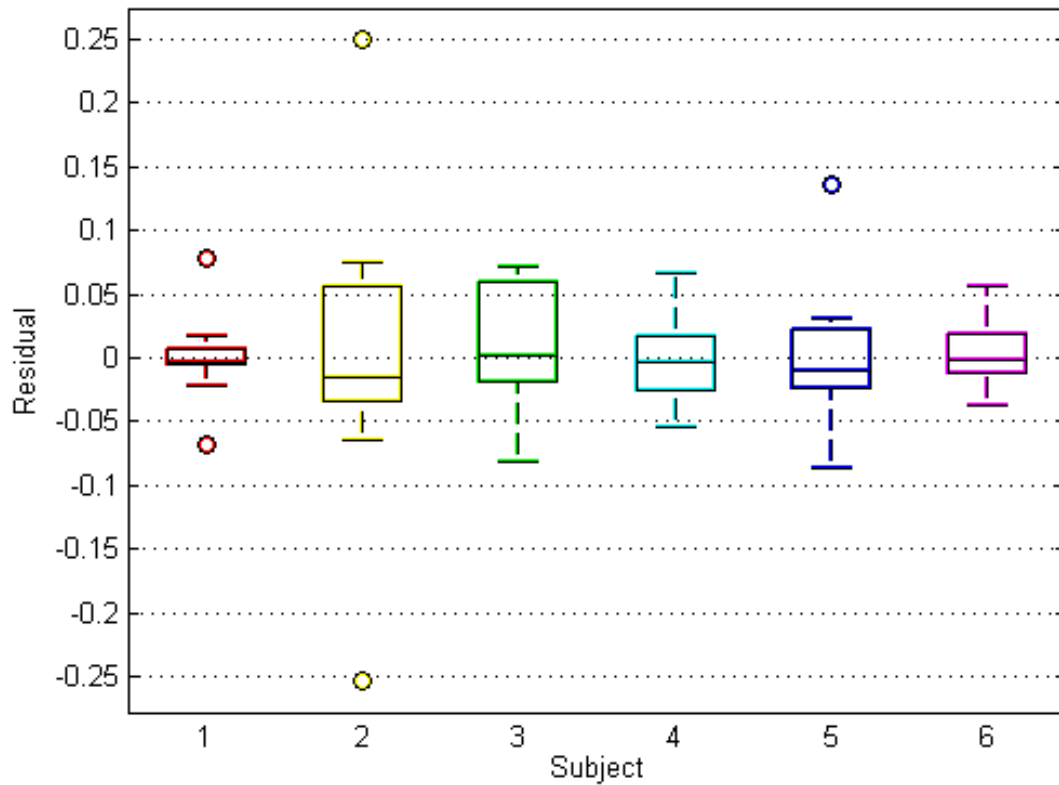


PHI gives estimates of the four model parameters for each of the six subjects. The estimates vary considerably, but taken as a 24-parameter model of the data, the mean-squared error of 0.0057 is a significant reduction from 0.0304 in the original four-parameter model.

Draw the box plot of residuals by subject.

```
h = boxplot(RES,'colors',colors,'symbol','o');
set(h(~isnan(h)),'LineWidth',2)
```

```
hold on
boxplot(RES, 'colors', 'k', 'symbol', 'ko')
grid on
xlabel('Subject')
ylabel('Residual')
hold off
```



Now the box plot shows that the larger model accounts for most of the subject-specific effects. The spread of the residuals (the vertical scale of the

box plot) is much smaller than in the previous box plot, and the boxes are now mostly centered on zero.

While the 24-parameter model successfully accounts for variations due to the specific subjects in the study, it does not consider the subjects as representatives of a larger population. The sampling distribution from which the subjects are drawn is likely more interesting than the sample itself. The purpose of mixed-effects models is to account for subject-specific variations more broadly, as random effects varying around population means.

Use the `nlmefit` function to fit a mixed-effects model to the data. You can also use `nlmefitsa` in place of `nlmefit`.

The following anonymous function, `nlme_model`, adapts the four-parameter model used by `nlmefit` to the calling syntax of `nlmefit` by allowing separate parameters for each individual. By default, `nlmefit` assigns random effects to all the model parameters. Also by default, `nlmefit` assumes a diagonal covariance matrix (no covariance among the random effects) to avoid overparametrization and related convergence issues.

```
nlme_model = @(PHI,t)(PHI(:,1).*exp(-exp(PHI(:,2)).*t) + ...
                    PHI(:,3).*exp(-exp(PHI(:,4)).*t));
```

```
phi0 = [1 2 1 1];
```

```
[phi,PSI,stats] = nlmefit(time,concentration,subject, ...
                        [],nlme_model,phi0)
```

```
phi =
```

```
2.8277
0.7729
0.4606
-1.3459
```

```
PSI =
```

```
0.3264    0    0    0
    0    0.0250    0    0
    0    0    0.0124    0
```

```

0          0          0    0.0000

stats =

    dfe: 57
    logl: 54.5882
    mse: 0.0066
    rmse: 0.0787
    errorparam: 0.0815
    aic: -91.1765
    bic: -93.0506
    covb: [4x4 double]
    sebeta: [0.2558 0.1066 0.1092 0.2244]
    ires: [66x1 double]
    pres: [66x1 double]
    iwres: [66x1 double]
    pwres: [66x1 double]
    cwres: [66x1 double]

```

The mean-squared error of 0.0066 is comparable to the 0.0057 of the 24-parameter model without random effects, and significantly better than the 0.0304 of the four-parameter model without random effects.

The estimated covariance matrix PSI shows that the variance of the fourth random effect is essentially zero, suggesting that you can remove it to simplify the model. To do this, use the 'REParamsSelect' name-value pair to specify the indices of the parameters to be modeled with random effects in `nlmefit`.

```

[phi,PSI,stats] = nlmefit(time,concentration,subject, ...
                        [],nlme_model,phi0, ...
                        'REParamsSelect',[1 2 3])

```

```

phi =

    2.8277
    0.7728
    0.4605

```

```
-1.3460
```

```
PSI =
```

```
0.3270      0      0
      0 0.0250      0
      0      0 0.0124
```

```
stats =
```

```
    dfe: 58
    logl: 54.5875
    mse: 0.0066
    rmse: 0.0780
errorparam: 0.0815
    aic: -93.1750
    bic: -94.8410
    covb: [4x4 double]
sebeta: [0.2560 0.1066 0.1092 0.2244]
    ires: [66x1 double]
    pres: [66x1 double]
    iwres: [66x1 double]
    pwres: [66x1 double]
    cwres: [66x1 double]
```

The log-likelihood `logl` is almost identical to what it was with random effects for all of the parameters, the Akaike information criterion `aic` is reduced from -91.1765 to -93.1750, and the Bayesian information criterion `bic` is reduced from -93.0506 to -94.8410. These measures support the decision to drop the fourth random effect.

Refitting the simplified model with a full covariance matrix allows for identification of correlations among the random effects. To do this, use the `CovPattern` parameter to specify the pattern of nonzero elements in the covariance matrix.

```
[phi,PSI,stats] = nlmefit(time,concentration,subject, ...
```



```
[ ],nlme_model,phi0, ...
'REParamsSelect',[1 2 3], ...
'CovPattern',ones(3))

phi =

    2.8149
    0.8292
    0.5613
   -1.1409

PSI =

    0.4767    0.1151    0.0499
    0.1151    0.0321    0.0032
    0.0499    0.0032    0.0236

stats =

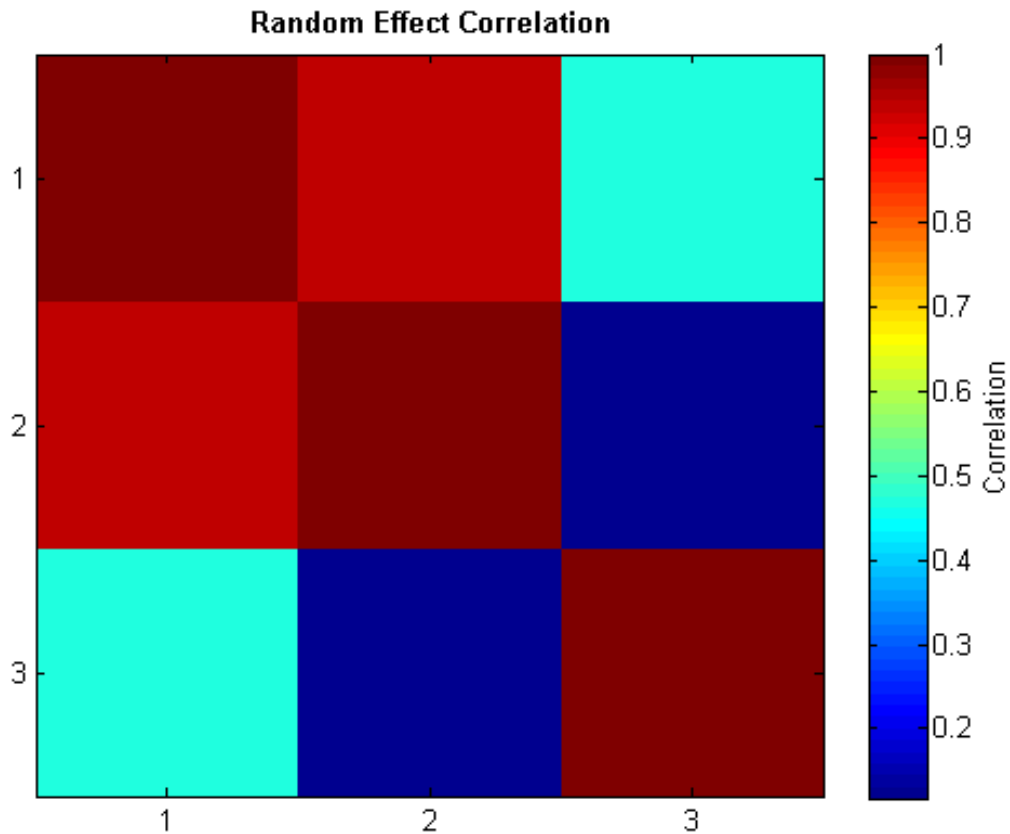
    dfe: 55
    logl: 58.4726
    mse: 0.0061
    rmse: 0.0782
    errorparam: 0.0781
    aic: -94.9451
    bic: -97.2358
    covb: [4x4 double]
    sebeta: [0.3028 0.1103 0.1179 0.1662]
    ires: [66x1 double]
    pres: [66x1 double]
    iwres: [66x1 double]
    pwres: [66x1 double]
    cwres: [66x1 double]
```

The estimated covariance matrix PSI shows that the random effects on the first two parameters have a relatively strong correlation, and both have a

relatively weak correlation with the last random effect. This structure in the covariance matrix is more apparent if you convert PSI to a correlation matrix using `corrcoef`.

```
RHO = corrcoef(PSI)
clf;
imagesc(RHO)
set(gca,'XTick',[1 2 3],'YTick',[1 2 3])
title('\bf Random Effect Correlation')
h = colorbar;
set(get(h,'YLabel'),'String','Correlation');
```

```
RHO =
    1.0000    0.9315    0.4706
    0.9315    1.0000    0.1175
    0.4706    0.1175    1.0000
```



Incorporate this structure into the model by changing the specification of the covariance pattern to block-diagonal.

```
P = [1 1 0;1 1 0;0 0 1] % Covariance pattern
[phi,PSI,stats,b] = nlmeFit(time,concentration,subject, ...
    [],nlme_model,phi0, ...
    'REParamsSelect',[1 2 3], ...
    'CovPattern',P)
```

P =

1	1	0
1	1	0
0	0	1

phi =

2.7830  
0.8981  
0.6581  
-1.0000

PSI =

0.5180	0.1069	0
0.1069	0.0221	0
0	0	0.0454

stats =

dfc: 57  
logl: 58.0804  
mse: 0.0061  
rmse: 0.0768  
errorparam: 0.0782  
aic: -98.1608  
bic: -100.0350  
covb: [4x4 double]  
sebeta: [0.3171 0.1073 0.1384 0.1453]  
ires: [66x1 double]  
pres: [66x1 double]  
iwres: [66x1 double]  
pwres: [66x1 double]  
cwres: [66x1 double]

b =

-0.8507	-0.1563	1.0427	-0.7559	0.5652	0.1550
-0.1756	-0.0323	0.2152	-0.1560	0.1167	0.0320
-0.2756	0.0519	0.2620	0.1064	-0.2835	0.1389

The block-diagonal covariance structure reduces `aic` from -94.9462 to -98.1608 and `bic` from -97.2368 to -100.0350 without significantly affecting the log-likelihood. These measures support the covariance structure used in the final model. The output `b` gives predictions of the three random effects for each of the six subjects. These are combined with the estimates of the fixed effects in `phi` to produce the mixed-effects model.

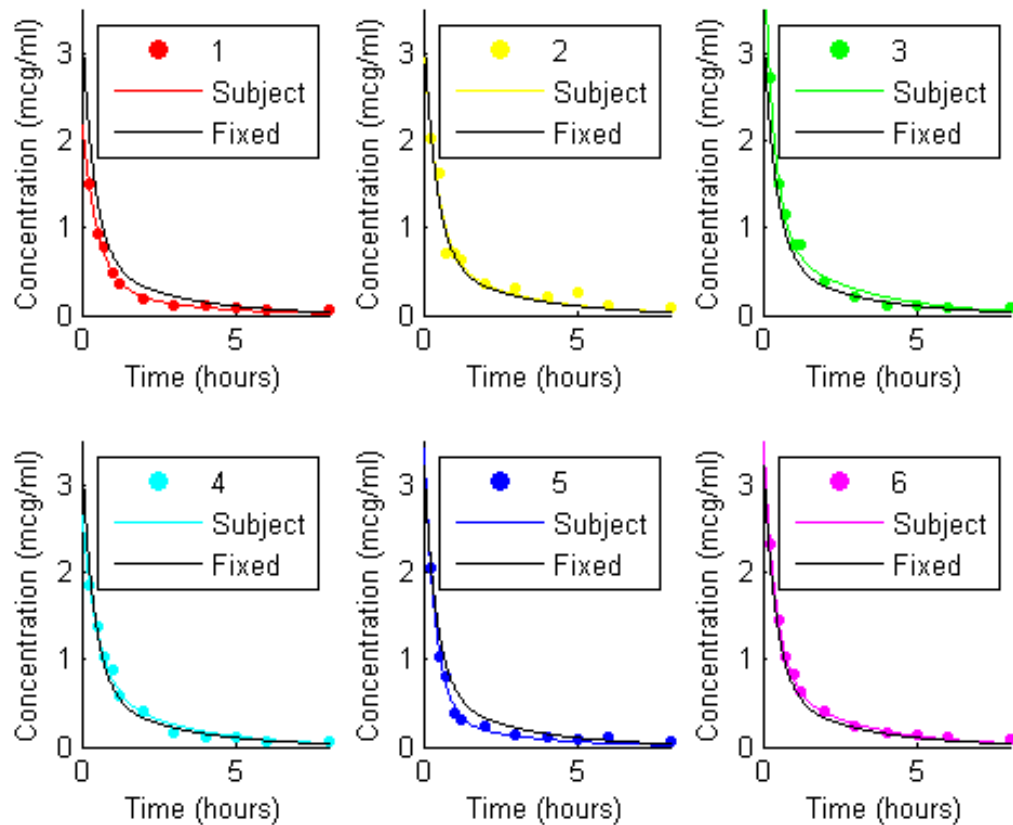
Plot the mixed-effects model for each of the six subjects. For comparison, the model without random effects is also shown.

```

PHI = repmat(phi,1,6) + ...           % Fixed effects
      [b(1,:);b(2,:);b(3,:);zeros(1,6)]; % Random effects
RES = zeros(11,6); % Residuals
colors = 'rygcbm';
for I = 1:6
    fitted_model = @(t)(PHI(1,I)*exp(-exp(PHI(2,I))*t) + ...
                        PHI(3,I)*exp(-exp(PHI(4,I))*t));
    tI = time(subject == I);
    cI = concentration(subject == I);
    RES(:,I) = cI - fitted_model(tI);

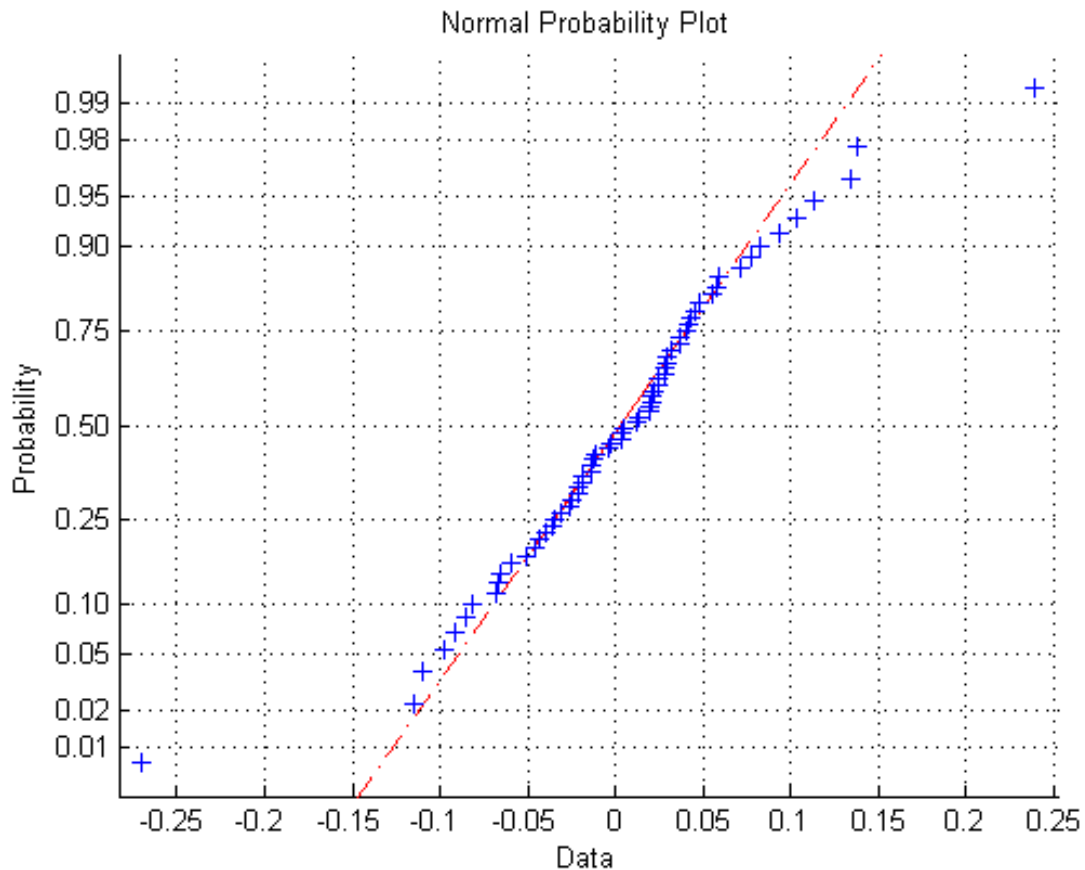
    subplot(2,3,I)
    scatter(tI,cI,20,colors(I),'filled')
    hold on
    plot(tplot,fitted_model(tplot),'Color',colors(I))
    plot(tplot,model(phi,tplot),'k')
    axis([0 8 0 3.5])
    xlabel('Time (hours)')
    ylabel('Concentration (mcg/ml)')
    legend(num2str(I),'Subject','Fixed')
end

```



If obvious outliers in the data (visible in previous box plots) are ignored, a normal probability plot of the residuals shows reasonable agreement with model assumptions on the errors.

```
clf; normplot(RES(:))
```



## Examining Residuals for Model Verification

You can examine the `stats` structure, which is returned by both `nlmeFit` and `nlmeFitSA`, to determine the quality of your model. The `stats` structure contains fields with conditional weighted residuals (`cwres` field) and individual weighted residuals (`iwres` field). Since the model assumes that residuals are normally distributed, you can examine the residuals to see how well this assumption holds.

This example generates synthetic data using normal distributions. It shows how the fit statistics look:

- Good when testing against the same type of model as generates the data
- Poor when tested against incorrect data models

**1** Initialize a 2-D model with 100 individuals:

```
nGroups = 100; % 100 Individuals
nlmefun = @(PHI,t)(PHI(:,1)*5 + PHI(:,2)^2.*t); % Regression fcn
REParamSelect = [1 2]; % Both Parameters have random effect
errorParam = .03;
beta0 = [ 1.5 5]; % Parameter means
psi = [ 0.35 0; ... % Covariance Matrix
       0 0.51 ];
time =[0.25;0.5;0.75;1;1.25;2;3;4;5;6];
nParameters = 2;
rng(0,'twister') % for reproducibility
```

**2** Generate the data for fitting with a proportional error model:

```
b_i = mvnrnd(zeros(1, numel(REParamSelect)), psi, nGroups);
individualParameters = zeros(nGroups,nParameters);
individualParameters(:, REParamSelect) = ...
    bsxfun(@plus,beta0(REParamSelect), b_i);

groups = repmat(1:nGroups,numel(time),1);
groups = vertcat(groups(:));

y = zeros(numel(time)*nGroups,1);
x = zeros(numel(time)*nGroups,1);
for i = 1:nGroups
    idx = groups == i;
    f = nlmefun(individualParameters(i,:), time);
    % Make a proportional error model for y:
    y(idx) = f + errorParam*f.*randn(numel(f),1);
    x(idx) = time;
end

P = [ 1 0 ; 0 1 ];
```



- 3** Fit the data using the same regression function and error model as the model generator:

```
[~,~,stats] = nlmefit(x,y,groups, ...
    [],nlmefun,[1 1], 'REParamsSelect', REParamSelect, ...
    'ErrorModel', 'Proportional', 'CovPattern', P);
```

- 4** Create a plotting routine by copying the following function definition, and creating a file `plotResiduals.m` on your MATLAB path:

```
function plotResiduals(stats)
pwres = stats.pwres;
iwres = stats.iwres;
cwres = stats.cwres;
figure
subplot(2,3,1);
normplot(pwres); title('PWRES')
subplot(2,3,4);
createhistplot(pwres);

subplot(2,3,2);
normplot(cwres); title('CWRES')
subplot(2,3,5);
createhistplot(cwres);

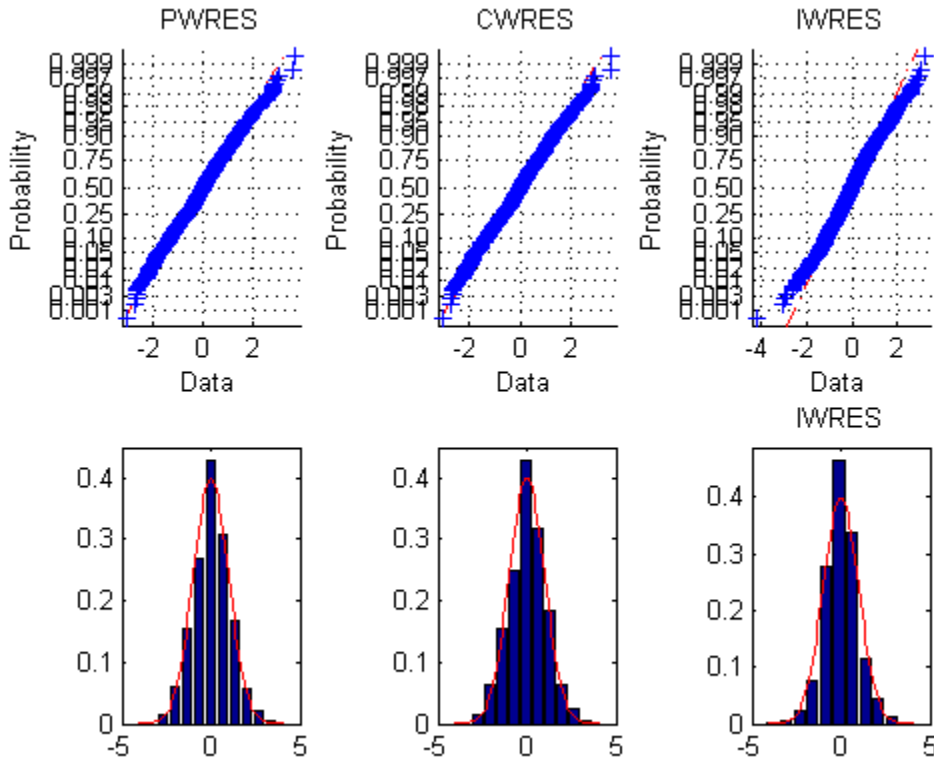
subplot(2,3,3);
normplot(iwres); title('IWRES')
subplot(2,3,6);
createhistplot(iwres); title('IWRES')

function createhistplot(pwres)
[x, n] = hist(pwres);
d = n(2)- n(1);
x = x/sum(x*d);
bar(n,x);
ylim([0 max(x)*1.05]);
hold on;
x2 = -4:0.1:4;
f2 = normpdf(x2,0,1);
plot(x2,f2, 'r');
end
```

end

5 Plot the residuals using the plotResiduals function:

```
plotResiduals(stats);
```



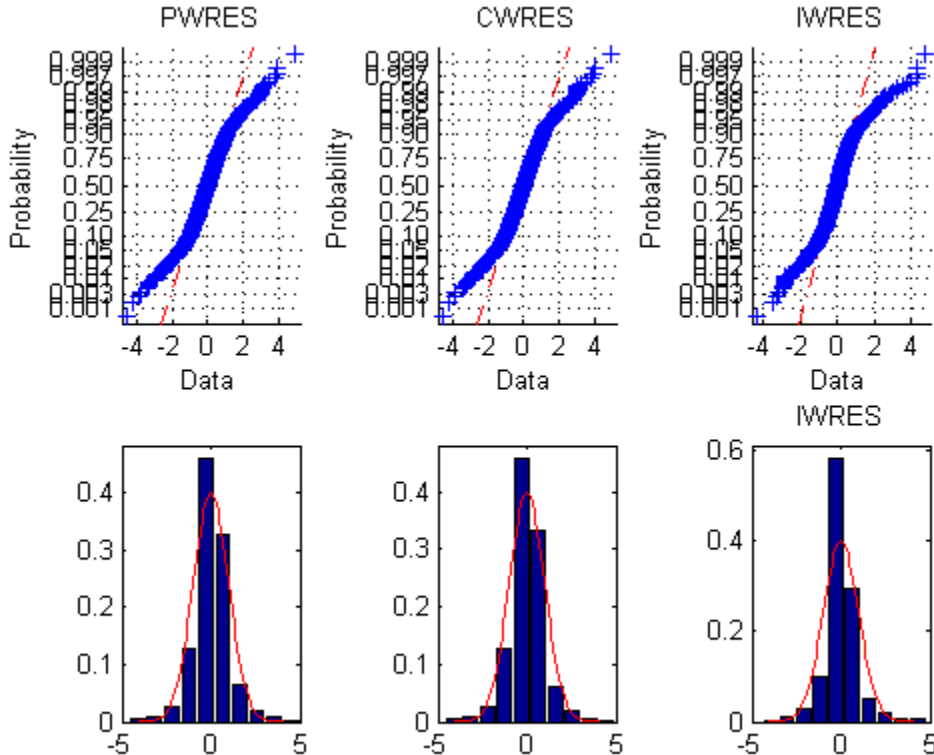
The upper probability plots look straight, meaning the residuals are normally distributed. The bottom histogram plots match the superimposed normal density plot. So you can conclude that the error model matches the data.

6 For comparison, fit the data using a constant error model, instead of the proportional model that created the data:

```

[~,~,stats] = nlmeFit(x,y,groups, ...
    [],nlmeFun,[0 0], 'REParamsSelect',REParamSelect,...
    'ErrorModel', 'Constant', 'CovPattern',P);
plotResiduals(stats);

```

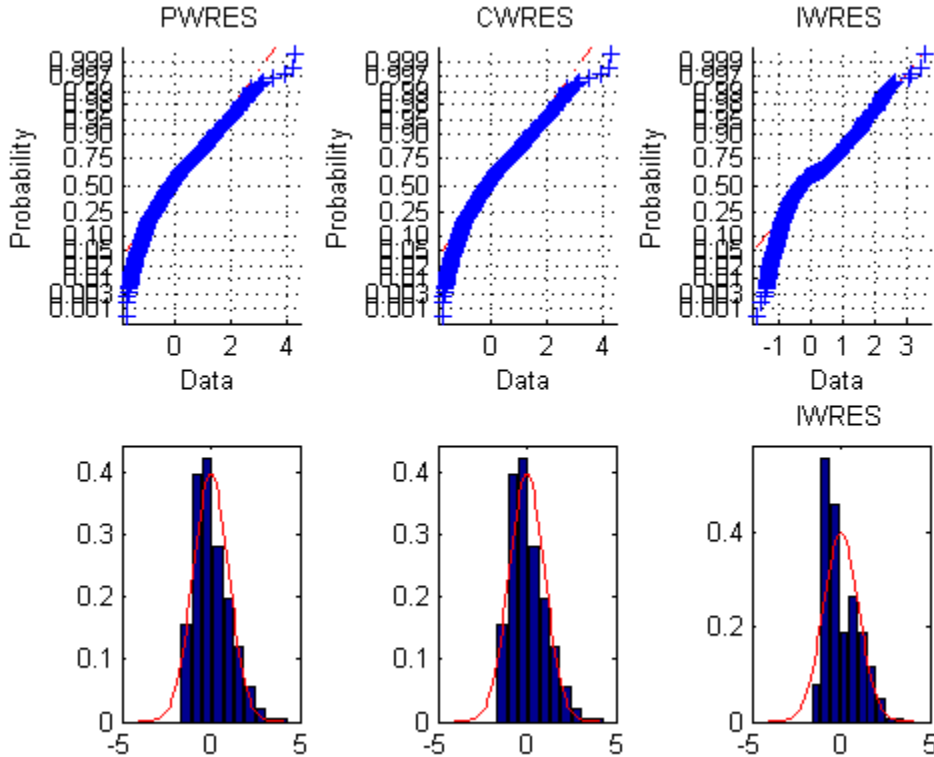


The upper probability plots are not straight, indicating the residuals are not normally distributed. The bottom histogram plots are fairly close to the superimposed normal density plots.

- 7 For another comparison, fit the data to a different structural model than created the data:

```
nlmeFun2 = @(PHI,t)(PHI(:,1)*5 + PHI(:,2).*t.^4);
```

```
[~,~,stats] = nlmefit(x,y,groups, ...
    [],nlmefun2,[0 0], 'REParamsSelect',REParamSelect,...
    'ErrorModel','constant', 'CovPattern',P);
plotResiduals(stats);
```



Not only are the upper probability plots not straight, but the histogram plot is quite skewed compared to the superimposed normal density. These residuals are not normally distributed, and do not match the model.

## Pitfalls in Fitting Nonlinear Models by Transforming to Linearity

This example shows pitfalls that can occur when fitting a nonlinear model by transforming to linearity. Imagine that we have collected measurements on two variables,  $x$  and  $y$ , and we want to model  $y$  as a function of  $x$ . Assume that  $x$  is measured exactly, while measurements of  $y$  are affected by additive, symmetric, zero-mean errors.

```
x = [5.72 4.22 5.72 3.59 5.04 2.66 5.02 3.11 0.13 2.26 ...
      5.39 2.57 1.20 1.82 3.23 5.46 3.15 1.84 0.21 4.29 ...
      4.61 0.36 3.76 1.59 1.87 3.14 2.45 5.36 3.44 3.41]';
y = [2.66 2.91 0.94 4.28 1.76 4.08 1.11 4.33 8.94 5.25 ...
      0.02 3.88 6.43 4.08 4.90 1.33 3.63 5.49 7.23 0.88 ...
      3.08 8.12 1.22 4.24 6.21 5.48 4.89 2.30 4.13 2.17]';
```

Let's also assume that theory tells us that these data should follow a model of exponential decay,  $y = p_1 \exp(p_2 x)$ , where  $p_1$  is positive and  $p_2$  is negative. To fit this model, we could use nonlinear least squares.

```
modelFun = @(p,x) p(1)*exp(p(2)*x);
```

But the nonlinear model can also be transformed to a linear one by taking the log on both sides, to get  $\log(y) = \log(p_1) + p_2 x$ . That's tempting, because we can fit that linear model by ordinary linear least squares. The coefficients we'd get from a linear least squares would be  $\log(p_1)$  and  $p_2$ .

```
paramEstsLin = [ones(size(x)), x] \ log(y);
paramEstsLin(1) = exp(paramEstsLin(1))
```

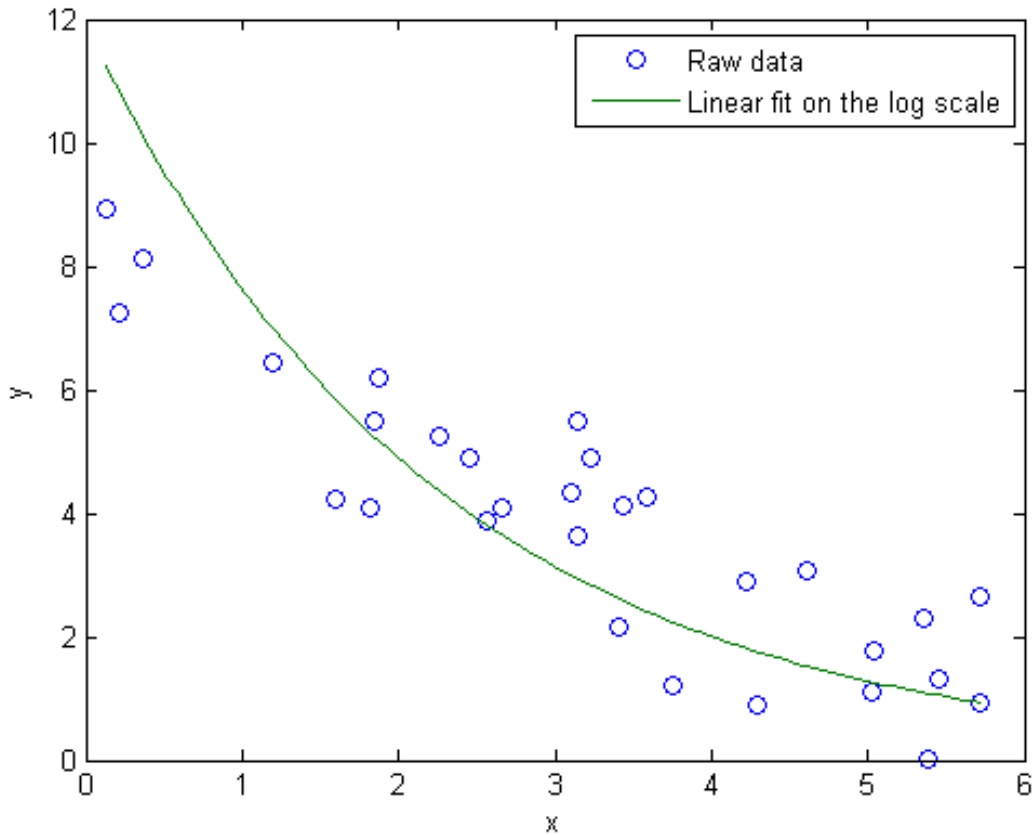
```
paramEstsLin =
```

```
11.9312
-0.4462
```

How did we do? We can superimpose the fit on the data to find out.

```
xx = linspace(min(x), max(x));
```

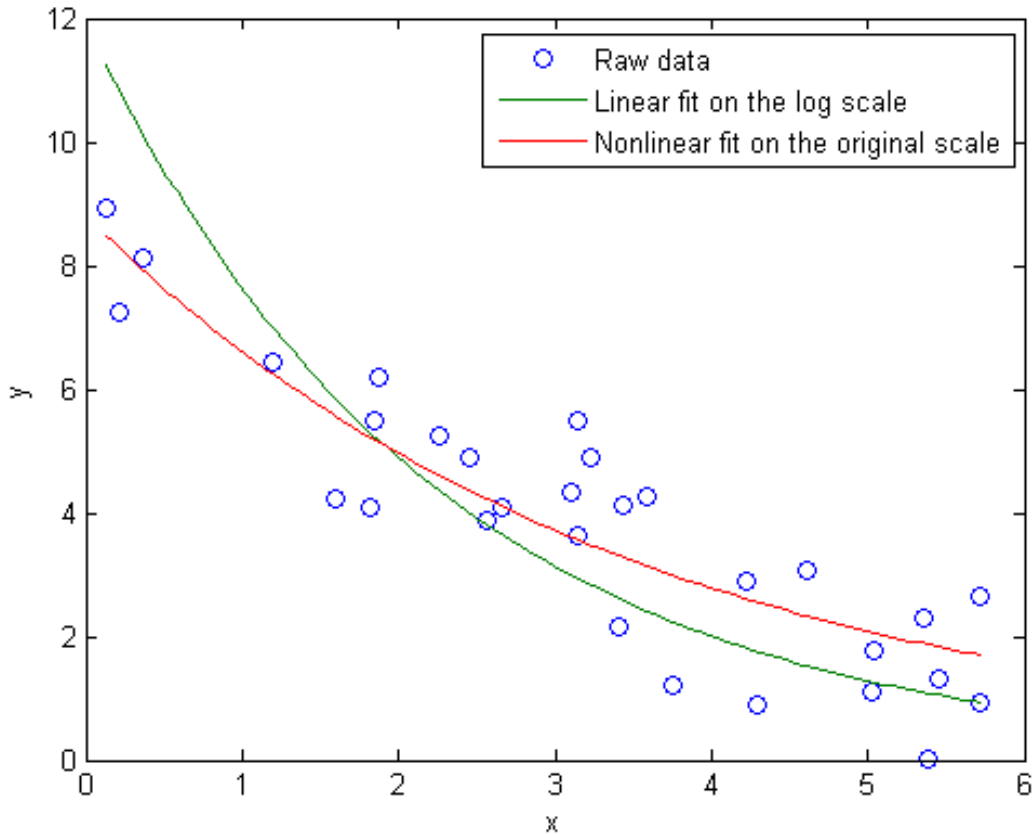
```
yyLin = modelFun(paramEstsLin, xx);  
plot(x,y,'o', xx,yyLin,'-');  
xlabel('x'); ylabel('y');  
legend({'Raw data','Linear fit on the log scale'},'location','NE');
```



Something seems to have gone wrong, because the fit doesn't really follow the trend that we can see in the raw data. What kind of fit would we get if we used `nlinfit` to do nonlinear least squares instead? We'll use the previous fit as a rough starting point, even though it's not a great fit.

```
paramEsts = nlinfit(x, y, modelFun, paramEstsLin)
```

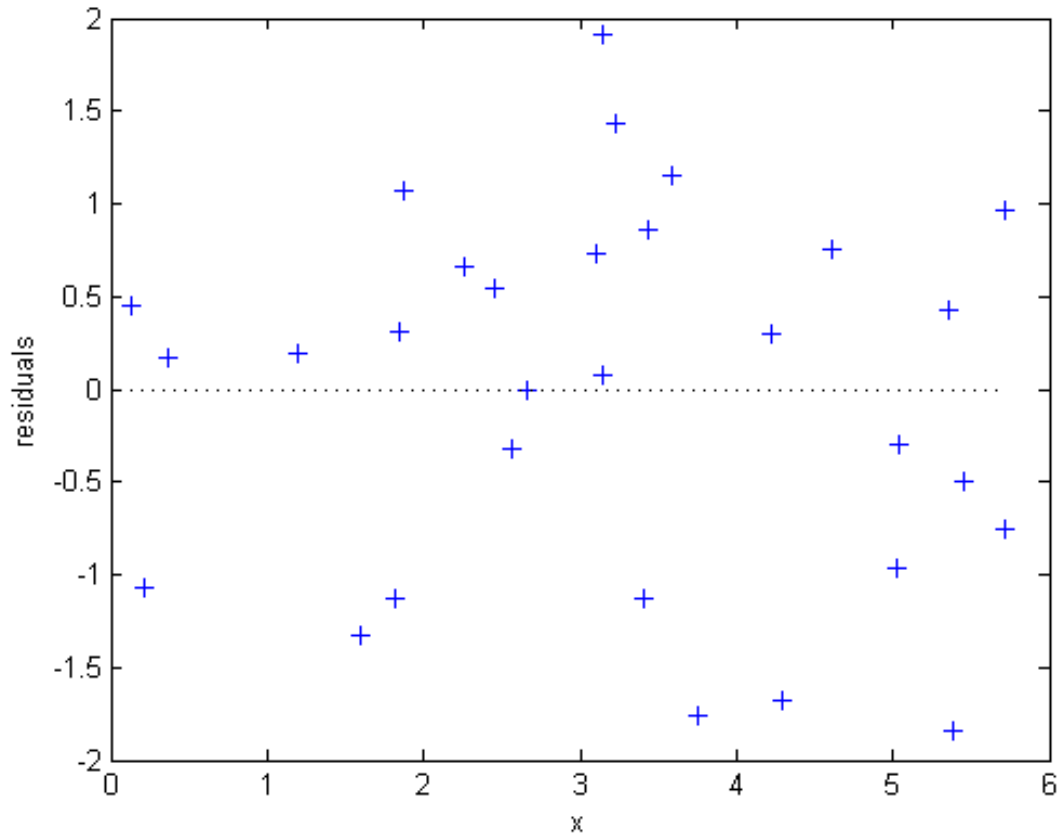
```
paramEsts =  
  
    8.8145  
   -0.2885  
  
yy = modelFun(paramEsts,xx);  
plot(x,y,'o', xx,yyLin,'-', xx,yy,'-');  
xlabel('x'); ylabel('y');  
legend({'Raw data','Linear fit on the log scale', ...  
       'Nonlinear fit on the original scale'},'location','NE');
```



The fit using `nlinfit` more or less passes through the center of the data point scatter. A residual plot shows something approximately like an even scatter about zero.

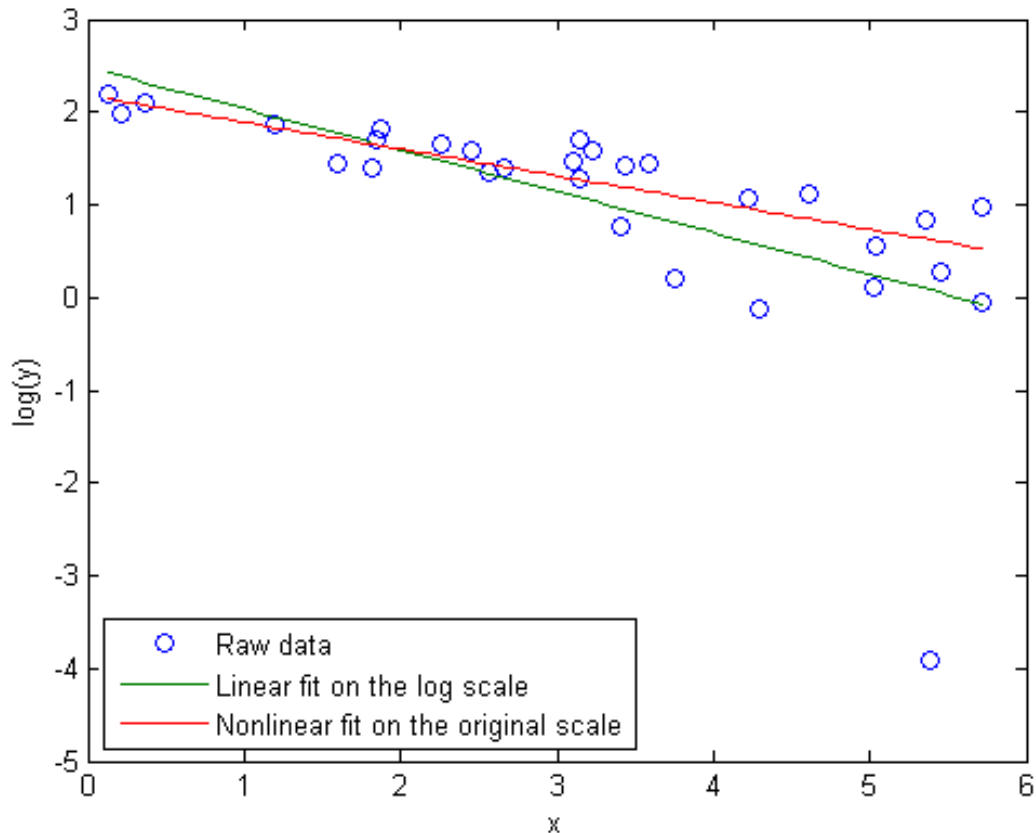
```
r = y-modelFun(paramEsts,x);
plot(x,r,'+', [min(x) max(x)], [0 0], 'k:');
xlabel('x'); ylabel('residuals');
```





So what went wrong with the linear fit? The problem is in log transform. If we plot the data and the two fits on the log scale, we can see that there's an extreme outlier.

```
plot(x,log(y),'o', xx,log(yyLin),'- ', xx,log(yy),'- ');
xlabel('x'); ylabel('log(y)');
ylim([-5,3]);
legend({'Raw data', 'Linear fit on the log scale', ...
       'Nonlinear fit on the original scale'},'location','SW');
```



That observation is not an outlier in the original data, so what happened to make it one on the log scale? The log transform is exactly the right thing to straighten out the trend line. But the log is a very nonlinear transform, and so symmetric measurement errors on the original scale have become asymmetric on the log scale. Notice that the outlier had the smallest y value on the original scale -- close to zero. The log transform has "stretched out" that smallest y value more than its neighbors. We made the linear fit on the log scale, and so it is very much affected by that outlier.

Had the measurement at that one point been slightly different, the two fits might have been much more similar. For example,

```
y(11) = 1;
paramEsts = nlinfit(x, y, modelFun, [10;-.3])

paramEsts =

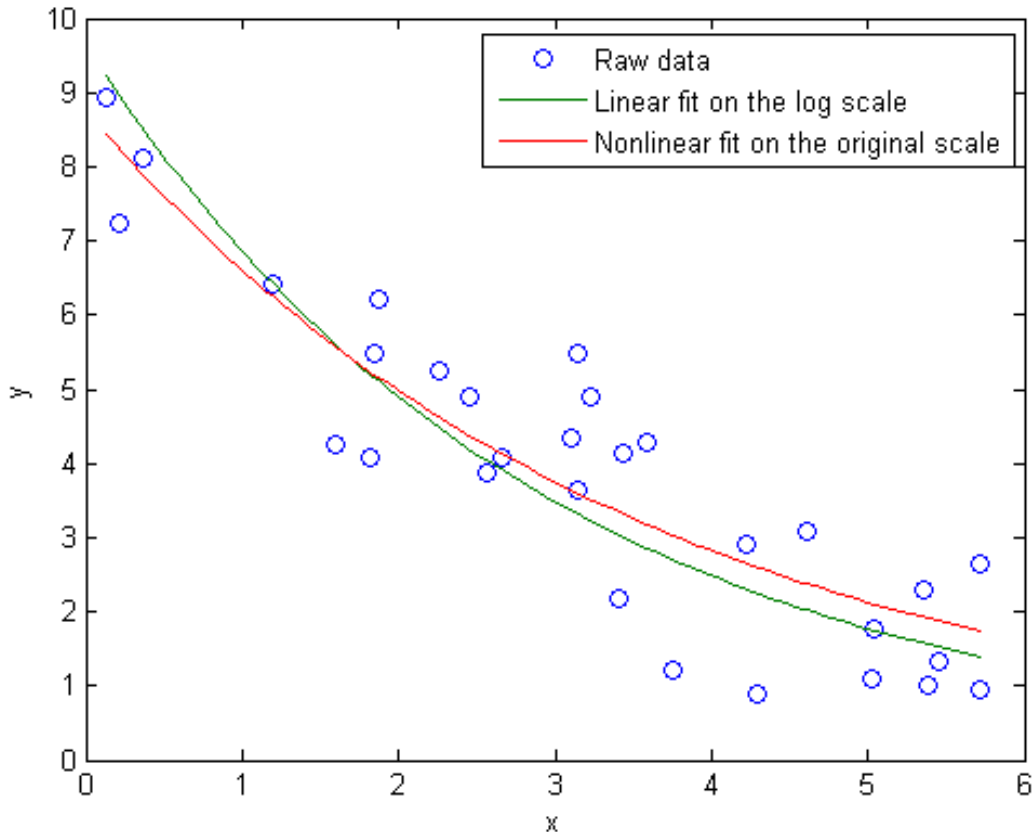
    8.7618
   -0.2833

paramEstsLin = [ones(size(x)), x] \ log(y);
paramEstsLin(1) = exp(paramEstsLin(1))

paramEstsLin =

    9.6357
   -0.3394

yy = modelFun(paramEsts,xx);
yyLin = modelFun(paramEstsLin, xx);
plot(x,y,'o', xx,yyLin,'-', xx,yy,'-');
xlabel('x'); ylabel('y');
legend({'Raw data', 'Linear fit on the log scale', ...
       'Nonlinear fit on the original scale'},'location','NE');
```



Still, the two fits are different. Which one is "right"? To answer that, suppose that instead of additive measurement errors, measurements of  $y$  were affected by multiplicative errors. These errors would not be symmetric, and least squares on the original scale would not be appropriate. On the other hand, the log transform would make the errors symmetric on the log scale, and the linear least squares fit on that scale is appropriate.

So, which method is "right" depends on what assumptions you are willing to make about your data. In practice, when the noise term is small relative to the trend, the log transform is "locally linear" in the sense that  $y$  values

near the same  $x$  value will not be stretched out too asymmetrically. In that case, the two methods lead to essentially the same fit. But when the noise term is not small, you should consider what assumptions are realistic, and choose an appropriate fitting method.



# Survival Analysis

---

- “What Is Survival Analysis?” on page 12-2
- “Kaplan-Meier Method” on page 12-11
- “Hazard and Survivor Functions for Different Groups” on page 12-19
- “Survivor Functions for Two Groups” on page 12-26
- “Cox Proportional Hazards Regression” on page 12-31
- “Cox Proportional Hazards Model for Censored Data” on page 12-34

## What Is Survival Analysis?

In this section...
“Introduction” on page 12-2
“Censoring” on page 12-2
“Data” on page 12-3
“Survivor Function” on page 12-5
“Hazard Function” on page 12-7

### Introduction

Survival analysis is time-to-event analysis, that is, when the outcome of interest is the time until an event occurs. Examples of time-to-events are the time until infection, reoccurrence of a disease, or recovery in health sciences, duration of unemployment in economics, time until the failure of a machine part or lifetime of light bulbs in engineering, and so on. Survival analysis is a part of reliability studies in engineering. In this case, it is usually used to study the lifetime of industrial components. In reliability analyses, survival times are usually called failure times as the variable of interest is how much time a component functions properly before it fails.

Survival analysis consists of parametric, semiparametric, and nonparametric methods. You can use these to estimate the most commonly used measures in survival studies, survivor and hazard functions, compare them for different groups, and assess the relationship of predictor variables to survival time. Some statistical probability distributions describe survival times well. Commonly used distributions are exponential, Weibull, lognormal, Burr, and Birnbaum-Saunders distributions. Statistics Toolbox functions `ecdf` and `ksdensity` compute the empirical and kernel density estimates of the cdf, cumulative hazard, and survivor functions. `coxphfit` fits the Cox proportional hazards model to the data.

### Censoring

One important concept in survival analysis is censoring. The survival times of some individuals might not be fully observed due to different reasons. In life sciences, this might happen when the survival study (e.g., the clinical trial)



stops before the full survival times of all individuals can be observed, or a person drops out of a study, or for long-term studies, when the patient is lost to follow up. In the industrial context, not all components might have failed before the end of the reliability study. In such cases, the individual survives beyond the time of the study, and the exact survival time is unknown. This is called right censoring.

During a survival study either the individual is observed to fail at time  $T$ , or the observation on that individual ceases at time  $c$ . Then the observation is  $\min(T, c)$  and an indicator variable  $I_c$  shows if the individual is censored or not. The calculations for hazard and survivor functions must be adjusted to account for censoring. Statistics Toolbox functions such as `ecdf`, `ksdensity`, `coxphfit`, `mle` account for censoring.

## Data

Survival data usually consists of the time until an event of interest occurs and the censoring information for each individual or component. The following table shows the fictitious unemployment time of individuals in a 6-month study. Two individuals are right censored (indicated by a censoring value of 1). One individual was still unemployed after the 24th week, when the study ended. Contact with the other censored individual was lost at the end of the 21st week.

Unemployment Time (Weeks)	Censoring
14	0
23	0
7	0
21	1
19	0
16	0
24	1
8	0

Survival data might also include the number of failures at a certain time (the number of times a particular survival or failure time was observed). The

following table shows the simulated time until a light-emitting diodes drops to 70% of its full light output level, in hours, in an accelerated life test.

<b>Failure Time (hrs)</b>	<b>Frequency</b>
8600	6
15300	19
22000	11
28600	20
35300	17
42000	14
48700	8
55400	2
62100	0
68800	2

Data might also have information on the predictor variables, to use in semi-parametric regression-like methods such as Cox proportional hazards regression.

<b>Time Until Recovery (weeks)</b>	<b>Censoring</b>	<b>Gender</b>	<b>Systolic Blood Pressure</b>	<b>Diastolic Blood Pressure</b>
12	1	Male	124	93
20	0	Female	109	77
7	0	Female	125	83
13	0	Male	117	75
9	1	Male	122	80
15	0	Female	121	70
17	1	Male	130	88
8	0	Female	115	82
14	0	Male	118	86

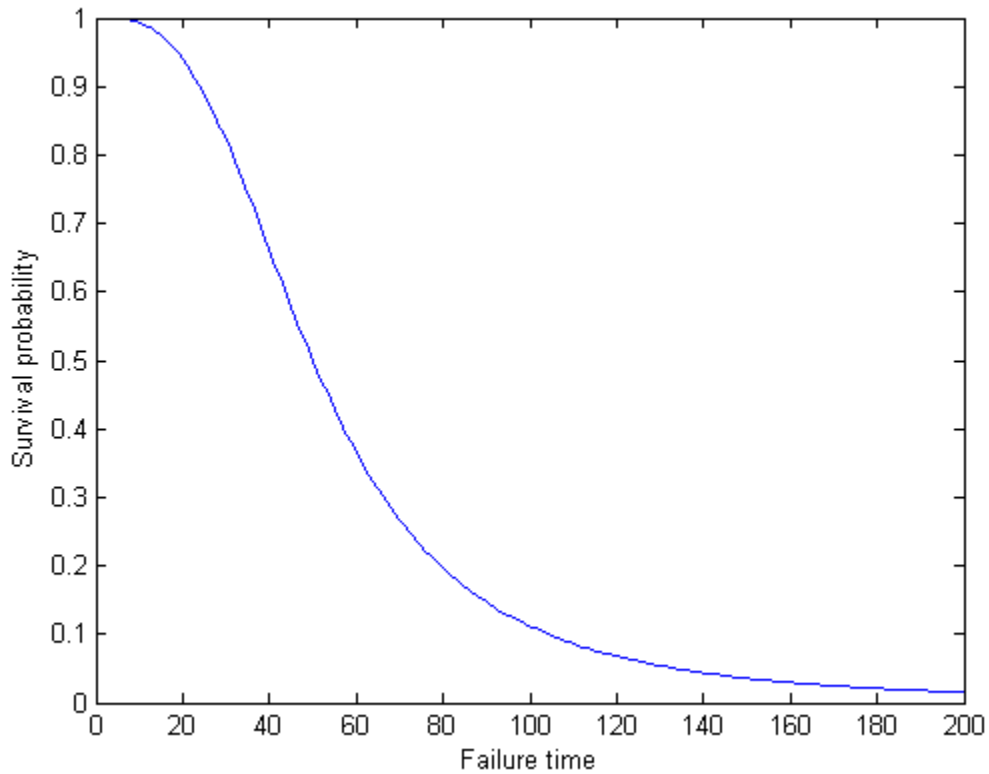
## Survivor Function

The survivor function is the probability of survival as a function of time. It is also called the survival function. It gives the probability that the survival time of an individual exceeds a certain value. Since the cumulative distribution function,  $F(t)$ , is the probability that the survival time is less than or equal to a given point in time, the survival function for a continuous distribution,  $S(t)$ , is the complement of the cumulative distribution function:

$$S(t) = 1 - F(t).$$

For example, for data coming from a Burr distribution with parameters 50, 3, and 1, you can calculate and plot the survivor function.

```
x = 0:0.1:200;
figure()
plot(x,1-cdf('Burr',x,50,3,1))
xlabel('Failure time');
ylabel('Survival probability');
```



The survivor function is also related to the hazard function. If the data has the hazard function,  $h(t)$ , then the survivor function is

$$S(t) = \exp\left(-\int_0^t h(u) du\right),$$

which corresponds to

$$S(t) = \exp(-H(t)),$$

where  $H(t)$  is the cumulative hazard function.

## Hazard Function

The hazard function gives the instantaneous failure rate of an individual conditioned on the fact that the individual survived until a given time. That is,

$$h(t) = \lim_{\Delta t \rightarrow 0} \frac{P(t \leq T < t + \Delta t \mid T \geq t)}{\Delta t},$$

where  $\Delta t$  is a very small time interval. The hazard rate, therefore, is sometimes called the conditional failure rate. The hazard function always takes a positive value. However, these values do not correspond to probabilities and might be greater than 1.

The hazard function is related to the probability density function,  $f(t)$ , cumulative distribution function,  $F(t)$ , and survivor function,  $S(t)$ , as follows:

$$h(t) = \frac{f(t)}{S(t)} = \frac{f(t)}{1 - F(t)},$$

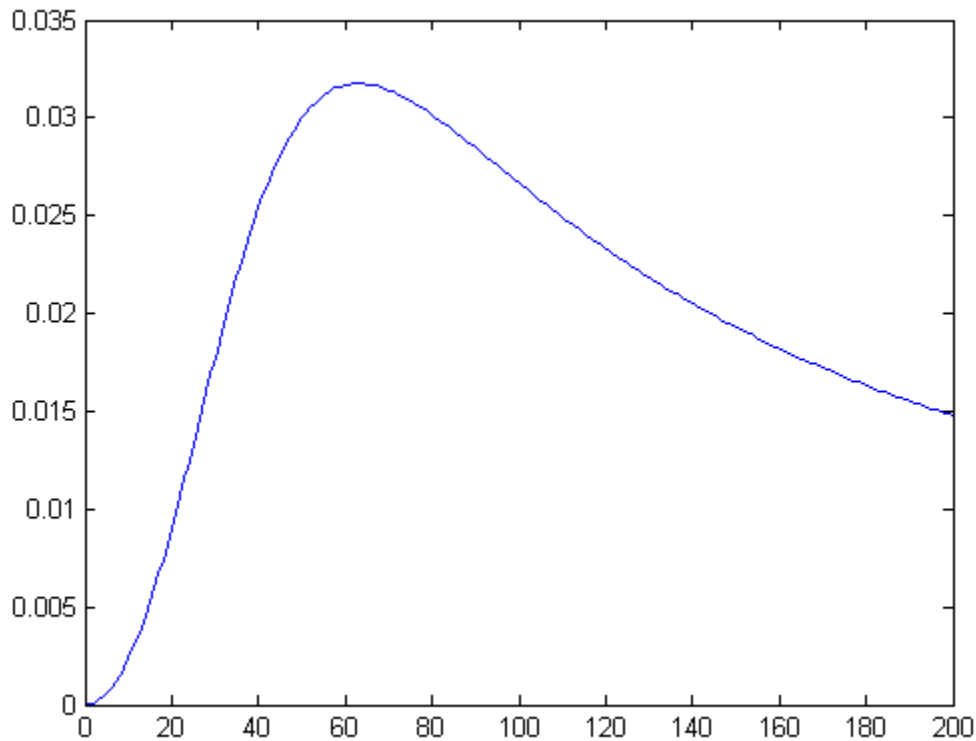
which is also equivalent to

$$h(t) = -\frac{d}{dt} \ln S(t).$$

So, if you know the shape of the survival function, you can also derive the corresponding hazard function.

For example, for data coming from a Burr distribution with parameters 50, 3, and 1, you can calculate and plot the hazard function.

```
x = 0:1:200;
Burrhazard = pdf('Burr',x,50,3,1) ./ (1-cdf('Burr',x,50,3,1));
figure()
plot(x,Burrhazard)
xlabel('Failure time');
ylabel('Hazard rate');
```



There are different types of hazard functions. The previous figure shows a situation when the hazard rate increases for the early time periods and then gradually decreases. The hazard rate might also be monotonically decreasing, increasing, or constant over time. The following figure shows examples of different types of hazard functions for data coming from different Weibull distributions.

```
ax1 = subplot(3,1,1);
x1 = 0:0.5:30;
hazard1 = pdf('wbl',x1,3,0.6)./(1-cdf('wbl',x1,3,0.6));
plot(x1,hazard1)

ax2 = subplot(3,1,2);
x2 = 0:0.05:2;
```

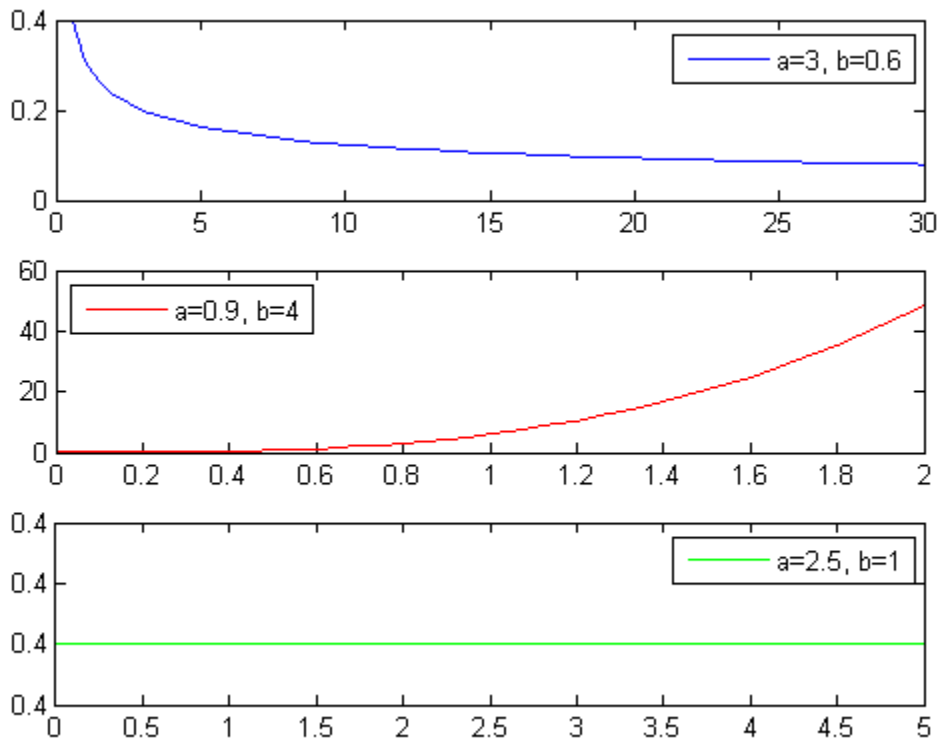
```

hazard2 = pdf('wbl',x2,0.9,4)./(1-cdf('wbl',x2,0.9,4));
plot(x2,hazard2,'color','r')

ax3 = subplot(3,1,3);
x3 = 0:0.05:5;
hazard3 = pdf('wbl',x3,2.5,1)./(1-cdf('wbl',x3,2.5,1));
plot(x3,hazard3)

set(ax1,'Ylim',[0 0.4]);
legend(ax1,'a=3, b=0.6');
legend(ax2,'a=0.9, b=4','location','northwest');
legend(ax3,'a=2.5, b=1');

```



In the third case, the Weibull distribution has a shape parameter value of 1, which corresponds to the exponential distribution. The exponential distribution always has a constant hazard rate over time.

## References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.

## See Also

`ecdf` | `coxphfit` | `ksdensity`

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 12-19
- “Survivor Functions for Two Groups” on page 12-26
- “Cox Proportional Hazards Model for Censored Data” on page 12-34

## Concepts

- “Kaplan-Meier Method” on page 12-11
- “Cox Proportional Hazards Regression” on page 12-31



## Kaplan-Meier Method

Use the Kaplan-Meier nonparametric method to estimate the empirical hazard, survivor, and cumulative distribution functions. The Statistics Toolbox function `ecdf` produces the empirical cumulative hazard, survivor, and cumulative distribution functions. The Kaplan-Meier estimator for the survivor function is also called the *product-limit estimator*.

The Kaplan-Meier method uses survival data summarized in life tables. Life tables order data according to ascending failure times, but you don't have to enter the failure/survival times in an ordered manner to use `ecdf`.

A life table usually consists of:

- Failure times
- Number of items failed at a time/time period
- Number of items censored at a time/time period
- Number of items at risk at the beginning of a time/time period

The number at risk is the total number of survivors at the beginning of each period. The number at risk at the beginning of the first period is all individuals in the lifetime study. At the beginning of each remaining period, the number at risk is reduced by the number of failures plus individuals censored at the end of the previous period.

This life table shows fictitious survival data. At the beginning of the first failure time, there are seven items at risk. At time 4, three fail. So at the beginning of time 7, there are four items at risk. Only one fails at time 7, so the number at risk at the beginning of time 11 is three. Two fail at time 11, so at the beginning of time 12, the number at risk is one. The remaining item fails at time 12.

Failure Time (t)	Number Failed	Number at Risk
4	3	7
7	1	4

Failure Time ( $t$ )	Number Failed	Number at Risk
11	2	3
12	1	1

You can estimate the hazard, cumulative hazard, survival, and cumulative distribution functions using the life tables as described next.

**Cumulative Hazard Rate (Failure Rate)**

The hazard rate at each period is the number of failures in the given period divided by the number of surviving individuals at the beginning of the period (number at risk).

Failure Time ( $t$ )	Hazard Rate ( $h(t)$ )	Cumulative Hazard Rate
0	0	0
$t_1$	$d_1/r_1$	$d_1/r_1$
$t_2$	$d_2/r_2$	$h(t_1) + d_2/r_2$
...	...	...
$t_n$	$d_n/r_n$	$h(t_{n-1}) + d_n/r_n$

**Survival Probability**

For each period, the survival probability is the product of the complement of hazard rates. The initial survival probability at the beginning of the first time period is 1. If the hazard rate for the each period is  $h(t_i)$ , then the survivor probability is as shown.

Time ( $t$ )	Survival Probability ( $S(t)$ )
0	1
$t_1$	$1*(1 - h(t_1))$
$t_2$	$S(t_1)*(1 - h(t_2))$

Time ( $t$ )	Survival Probability ( $S(t)$ )
...	...
$t_n$	$S(t_{n-1}) * (1 - h(t_n))$

### Cumulative Distribution Function

Because the cumulative distribution function (cdf) and the survivor function are complements of each other, you can find the cdf from the life tables using  $F(t) = 1 - S(t)$ .

You can compute the cumulative hazard rate, survival rate, and cumulative distribution function for the simulated data in the first table on this page as follows.

$t$	Number Failed ( $d$ )	Number at Risk ( $r$ )	Hazard Rate	Survival Probability	Cumulative Distribution Function
4	3	7	3/7	$1 - 3/7 = 4/7 = 0.5714$	0.4286
7	1	4	1/4	$4/7 * (1 - 1/4) = 3/7 = .4286$	0.5714
11	2	3	2/3	$3/7 * (1 - 2/3) = 1/7 = 0.1429$	0.8571
12	1	1	1/1	$1/7 * (1 - 1) = 0$	1

This rates in this example are based on the discrete failure times, and hence the calculations do not necessarily follow the derivative-based definition in “What Is Survival Analysis?” on page 12-2

Here is how you can enter the data and calculate these measures using `ecdf`. The data does not necessarily have to be in ascending order. Suppose the failure times are stored in an array `y`.

```
y = [4 7 11 12];
freq = [3 1 2 1];
[f,x] = ecdf(y, 'frequency', freq)
```

```
f =
```

```
0
0.4286
0.5714
0.8571
1.0000
```

```
x =
```

```
4
4
7
11
12
```

When you have censored data, the life table might look like the following:

Time (t)	Number failed (d)	Censoring	Number at Risk (r)	Hazard Rate	Survival Probability	Cumulative Distribution Function
4	2	1	7	2/7	$1 - 2/7 = 0.7143$	0.2857
7	1	0	4	1/4	$0.7143 * (1 - 1/4) = 0.5357$	0.4643
11	1	1	3	2/3	$0.5357 * (1 - 1/3) = 0.3571$	0.6429
12	1	0	1	1/1	$0.3571 * (1 - 1) = 0$	1.0000

At any given time, the censored items are also considered in the total of number at risk, and the hazard rate formula is based on the number failed and the total number at risk. While updating the number at risk at the beginning of each period, the total number failed and censored in the previous period is reduced from the number at risk at the beginning of that period.

While using `ecdf`, you must also enter the censoring information using an array of binary variables. Enter 1 for censored data, and enter 0 for exact failure time.

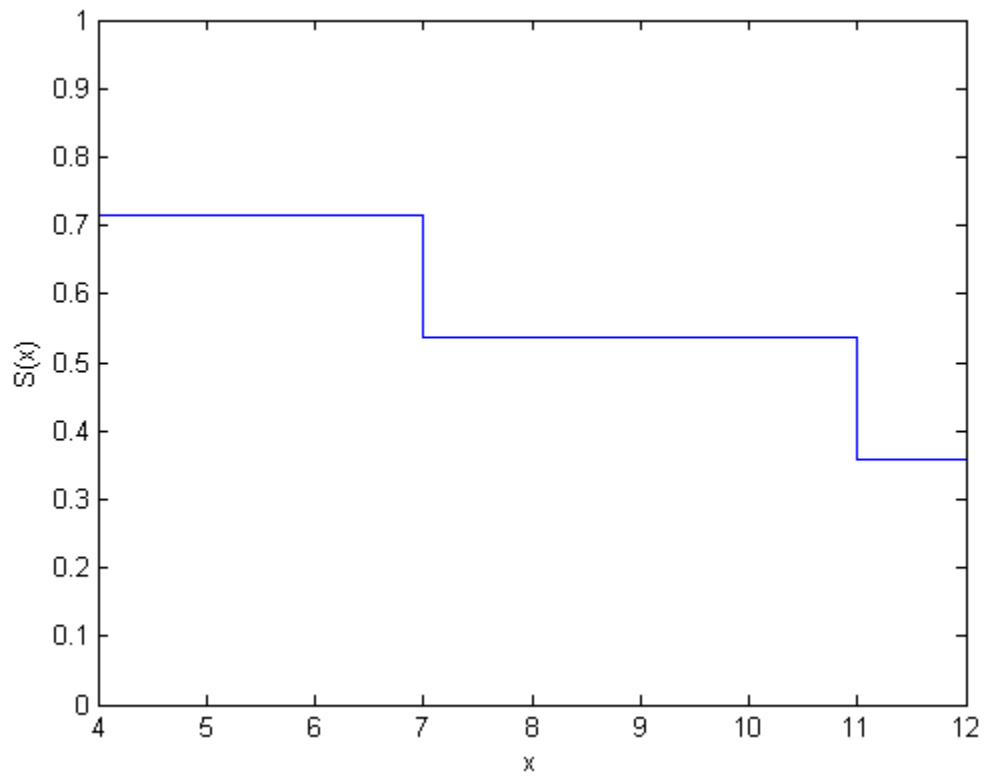
```
y = [4 4 4 7 11 11 12];  
cens = [0 1 0 0 1 0 0];  
[f,x] = ecdf(y,'censoring',cens)
```

```
f =  
  
      0  
0.2857  
0.4643  
0.6429  
1.0000
```

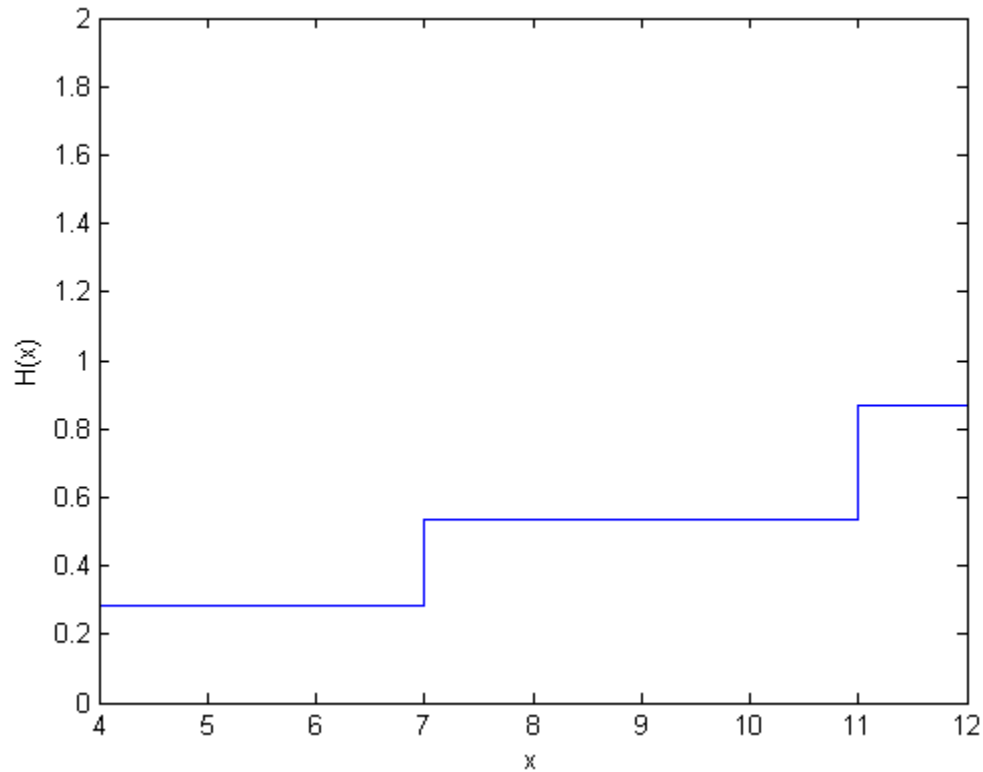
```
x =  
  
      4  
      4  
      7  
     11  
     12
```

`ecdf`, by default, produces the cumulative distribution function values. You have to specify the survivor function or the hazard function using optional name-value pair arguments. You can also plot the results as follows.

```
figure()  
ecdf(y,'censoring',cens,'function','survivor');
```



```
figure()  
ecdf(y,'censoring',cens,'function','cumulative hazard');
```



## References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.

### **See Also**

`ecdf` | `coxphfit` | `ksdensity`

### **Related Examples**

- “Hazard and Survivor Functions for Different Groups” on page 12-19
- “Survivor Functions for Two Groups” on page 12-26
- “Cox Proportional Hazards Model for Censored Data” on page 12-34

### **Concepts**

- “What Is Survival Analysis?” on page 12-2
- “Cox Proportional Hazards Regression” on page 12-31



## Hazard and Survivor Functions for Different Groups

This example shows how to estimate and plot the cumulative hazard and survivor functions for different groups.

### Step 1. Load and organize sample data.

Navigate to a folder containing sample data.

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

Load the sample data.

```
load readmissiontimes
```

The data has readmission times of patients with information on their gender, age, weight, smoking status, and censorship. This is simulated data.

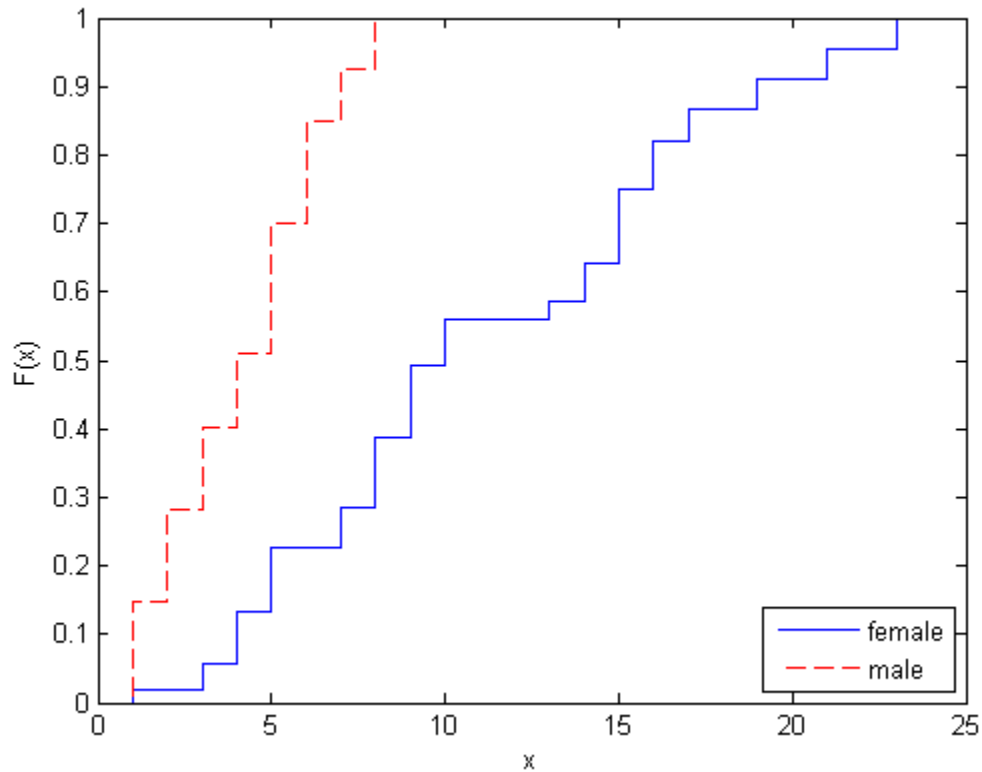
Create a matrix of readmission times and censoring for each gender.

```
female = [ReadmissionTime(Sex==1),Censored(Sex==1)];
male = [ReadmissionTime(Sex==0),Censored(Sex==0)];
```

### Step 2. Estimate and plot cumulative distribution function for each gender.

Plot the Kaplan-Meier estimate of the cumulative distribution function for female and male patients.

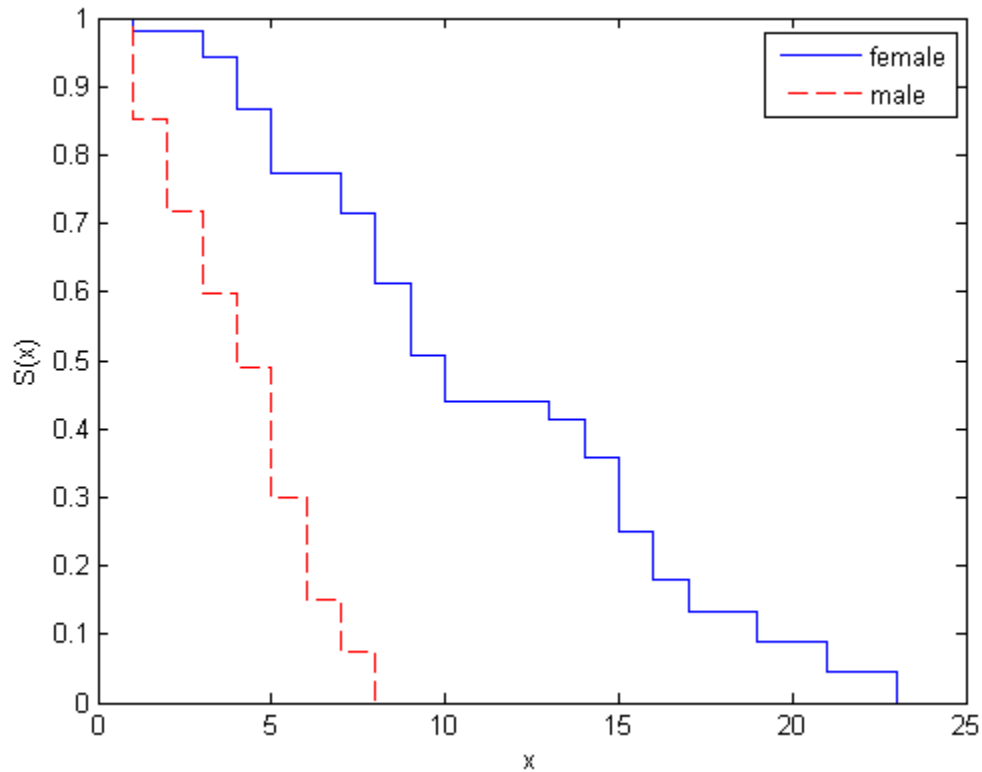
```
figure()
ecdf(gca,female(:,1),'Censoring',female(:,2));
hold on
[f,x] = ecdf(male(:,1),'Censoring',male(:,2));
stairs(x,f,'--r')
hold off
legend('female','male','Location','SouthEast')
```



### Step 3. Plot survivor functions.

Compare the survivor functions for female and male patients.

```
figure()
ax1 = gca;
ecdf(ax1,female(:,1),'Censoring',female(:,2),'function','survivor');
hold on
[f,x] = ecdf(male(:,1),'Censoring',male(:,2),'function','survivor');
stairs(x,f,'--r')
legend('female','male')
```



This figure shows that readmission times are shorter for male patients than female patients.

**Step 4. Fit Weibull survivor functions.**

Fit Weibull distributions to readmission times of female and male patients.

```
pd = fitdist(female(:,1),'wbl','Censoring',female(:,2))
```

```
pd =
```

```
weibull distribution
```

```
a = 12.5593
b = 1.99834

pd2 = fitdist(male(:,1), 'wbl', 'Censoring', male(:,2))

pd2 =

weibull distribution

a = 4.63991
b = 1.94422

pd2 = fitdist(male(:,1), 'wbl', 'Censoring', male(:,2))

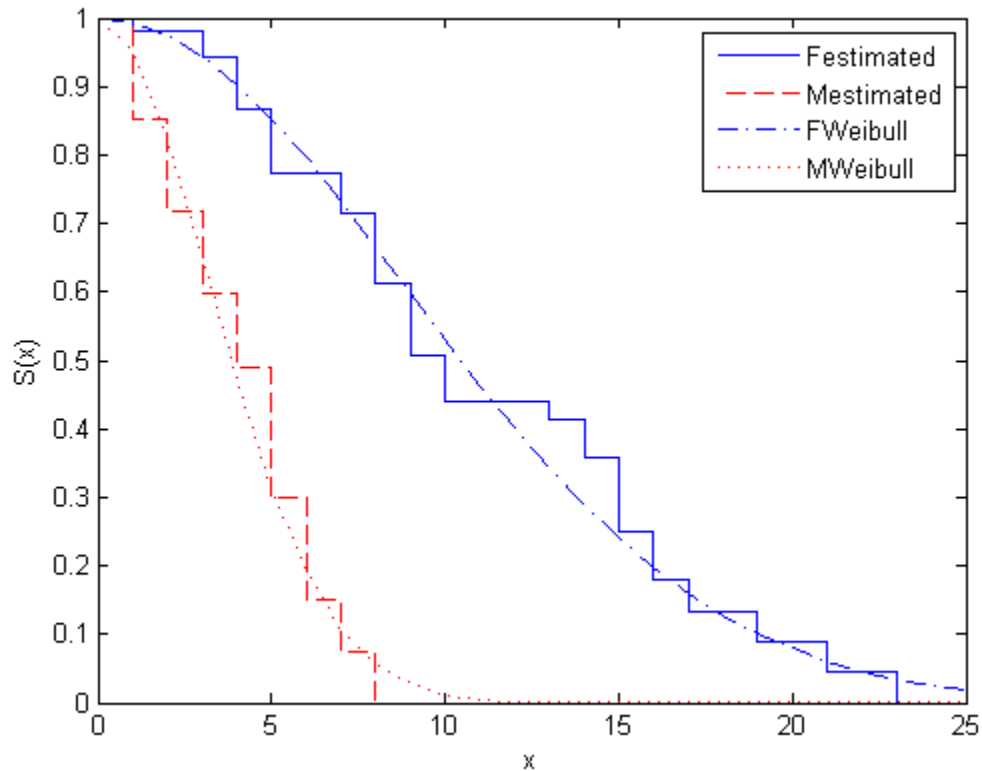
pd2 =

weibull distribution

a = 4.63991
b = 1.94422
```

Plot the Weibull survivor functions for female and male patients on estimated survivor functions.

```
plot(0:1:25, 1-cdf('wbl', 0:1:25, 12.5593, 1.99834), '-. ')
plot(0:1:25, 1-cdf('wbl', 0:1:25, 4.63991, 1.94422), ':r')
hold off
legend('Festimated', 'Mestimated', 'FWeibull', 'MWeibull')
```



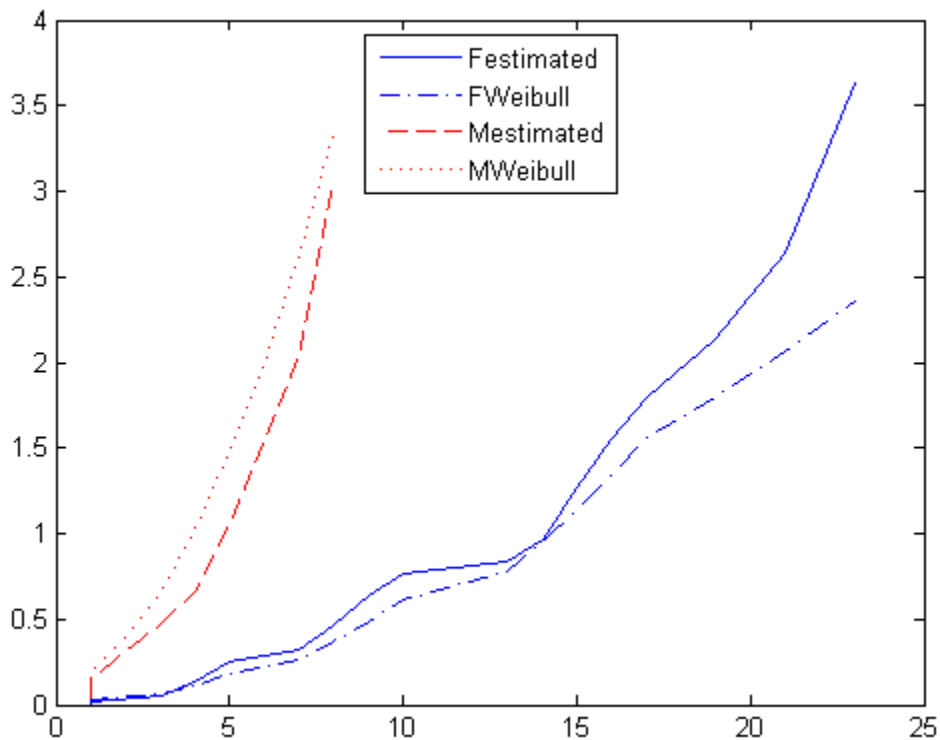
Weibull distribution provides a good fit for the data.

**Step 5. Estimate cumulative hazard and fit Weibull cumulative hazard functions.**

Estimate the cumulative hazard function for the genders and fit Weibull cumulative hazard functions.

```
figure()
[f,x] = ecdf(female(:,1),'Censoring',female(:,2),...
'function','cumhazard');
plot(x,f)
hold on
plot(x,cumsum(pdf(pd,x)./(1-cdf(pd,x))),'-.-')
```

```
[f,x] = ecdf(male(:,1),'Censoring',male(:,2),...
'function','cumhazard');
plot(x,f,'--r')
plot(x,cumsum(pdf(pd2,x)./(1-cdf(pd2,x))),':r')
legend('Festimated','FWeibull','Mestimated','MWeibull',...
'Location','North')
```



## See Also

[ecdf](#) | [coxphfit](#) | [ksdensity](#)

## Related Examples

- “Survivor Functions for Two Groups” on page 12-26
- “Cox Proportional Hazards Model for Censored Data” on page 12-34

## **Concepts**

- “What Is Survival Analysis?” on page 12-2
- “Kaplan-Meier Method” on page 12-11
- “Cox Proportional Hazards Regression” on page 12-31

## Survivor Functions for Two Groups

This example shows how to find the empirical survivor functions and the parametric survivor functions using the Burr type XII distribution fit to data for two groups.

### Step 1. Load and prepare sample data.

Navigate to a folder containing sample data.

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

Load the sample data

```
load lightbulb
```

The first column of the data has the lifetime (in hours) of two types of light bulbs. The second column has information about the type of light bulb. 1 indicates fluorescent bulbs whereas 0 indicates the incandescent bulb. The third column has censoring information. 1 indicates censored data, and 0 indicates the exact failure time. This is simulated data.

Create a variable for each light bulb type and also include the censorship information.

```
fluo = [lightbulb(lightbulb(:,2)==0,1),...
        lightbulb(lightbulb(:,2)==0,3)];
insec = [lightbulb(lightbulb(:,2)==1,1),...
         lightbulb(lightbulb(:,2)==1,3)];
```

### Step 2. Plot estimated survivor functions.

Plot the estimated survivor functions for the two different types of light bulbs.

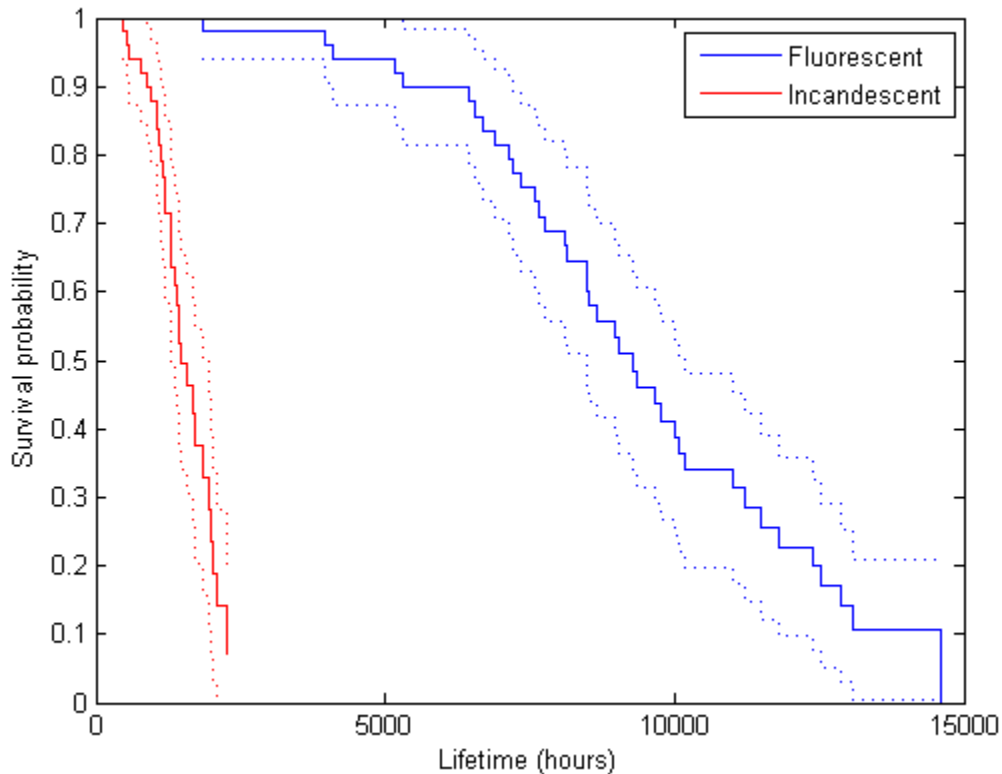
```
figure()
[f,x,flow,fup] = ecdf(fluc(:,1),'censoring',fluc(:,2),...
    'function','survivor');
ax1 = stairs(x,f);
hold on
stairs(x,flow,':')
```



```

stairs(x,fup,':')
[f,x,flow,fup] = ecdf(incsc(:,1),'censoring',incsc(:,2),...
    'function','survivor');
ax2 = stairs(x,f,'color','r');
stairs(x,flow,':r')
stairs(x,fup,':r')
legend([ax1,ax2],{'Fluorescent','Incandescent'})
xlabel('Lifetime (hours)')
ylabel('Survival probability')

```



You can see that the survival probability of incandescent light bulbs is much smaller than that of fluorescent light bulbs.

**Step 3. Fit Burr Type XII distribution.**

Fit Burr distribution to the lifetime data of fluorescent and incandescent type bulbs.

```
pd = fitdist(fluo(:,1), 'burr', 'Censoring', fluo(:,2))
```

```
pd =
```

```
burr distribution
```

```
alpha = 29143.5
```

```
c = 3.44582
```

```
k = 33.704
```

```
pd2 = fitdist(incsc(:,1), 'burr', 'Censoring', incsc(:,2))
```

```
pd2 =
```

```
burr distribution
```

```
alpha = 2650.76
```

```
c = 3.41898
```

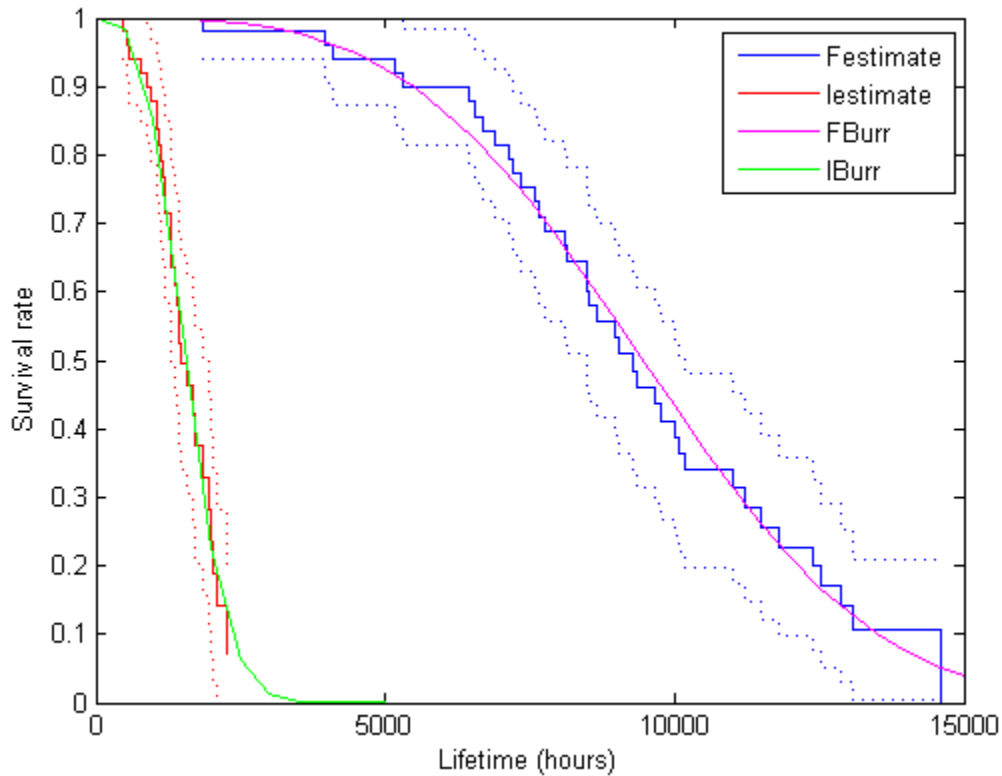
```
k = 4.5891
```

Superimpose Burr type XII survivor functions.

```
ax3 = plot(0:500:15000, 1-cdf('burr', 0:500:15000, 29143.5, ...  
3.44582, 33.704), 'm');
```

```
ax4 = plot(0:500:5000, 1-cdf('burr', 0:500:5000, 2650.76, ...  
3.41898, 4.5891), 'g');
```

```
legend([ax1;ax2;ax3;ax4], 'Festimate', 'Iestimate', 'FBurr', 'IBurr')
```



Burr distribution provides a good fit for the lifetime of light bulbs in this example.

#### Step 4. Fit a Cox proportional hazards model.

Fit a Cox proportional hazards regression where the type of the bulb is the explanatory variable.

```
[b,logl,H,stats] = coxphfit(lightbulb(:,2),lightbulb(:,1),...
    'Censoring',lightbulb(:,3));
stats
```

```
stats =
```

```
covb: 1.0757  
beta: 4.7262  
se: 1.0372  
z: 4.5568  
p: 5.1936e-06
```

The  $p$ -value,  $p$ , indicates that the type of light bulb is statistically significant. The estimate of the hazard ratio is  $\exp(b) = 112.8646$ . This means that the hazard for the incandescent bulbs is 112.86 times the hazard for the fluorescent bulbs.

### See Also

`ecdf` | `coxphfit` | `ksdensity`

### Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 12-19
- “Cox Proportional Hazards Model for Censored Data” on page 12-34

### Concepts

- “What Is Survival Analysis?” on page 12-2
- “Kaplan-Meier Method” on page 12-11
- “Cox Proportional Hazards Regression” on page 12-31

## Cox Proportional Hazards Regression

Cox proportional hazards regression is a semiparametric method for adjusting survival rate estimates to quantify the effect of predictor variables. The method represents the effects of explanatory variables as a multiplier of a common baseline hazard function,  $h_0(t)$ . The hazard function is the nonparametric part of the Cox proportional hazards regression function, whereas the impact of the predictor variables is a loglinear regression. For a baseline relative to 0, this model corresponds to

$$h_X(t) = h_0(t)e^{\sum_i X_i b_i},$$

where  $h_X(t)$  is the hazard rate at  $X$  and  $h_0(t)$  is the baseline hazard rate function.

The Cox proportional hazards model relates the hazard rate for individuals or items at the value  $X$ , to the hazard rate for individuals or items at the baseline value. It produces an estimate for the hazard ratio,  $HR = h_X(t)/h_0(t)$ . The model is based on the assumption that the baseline hazard function depends on time,  $t$ , but the predictor variables do not. This is also called the proportional hazards assumption, which states that the hazard rate does not change over time for any individual. The hazard ratio represents the relative risk of instant failure for individuals or items having the predictive variable value  $X$  compared to the ones having the baseline values. For example, if the predictive variable is smoking status, where nonsmoking is the baseline category, the hazard ratio shows the relative instant failure rate of smokers compared to the baseline category, that is, nonsmokers.

For a baseline relative to  $X^*$  and the predictor variable value  $X$ , the hazard ratio is

$$HR = \frac{h_X(t)}{h_{X^*}(t)} = \exp\left[\sum_i (X_i - X_i^*) b_i\right].$$

For example, if the baseline is the mean values of the predictor variables ( $\text{mean}(X)$ ), then the hazard rate model becomes

$$h_X(t) = h_{\bar{X}}(t) \exp \left[ \sum_i (X_i - \bar{X}) b_i \right].$$

Hazard rates are related to survival rates, such that the survival rate at time  $t$  for an individual with the explanatory variable value  $x$  is

$$S_X(t) = S_0(t)^{HR_x(t)},$$

where  $S_0(t)$  is the survivor function with the baseline hazard rate function  $h_0(t)$ , and  $HR_x(t)$  is the hazard ratio of the predictor variable value  $x$  relative to the baseline value.

A point estimate of the effect of each explanatory variable, that is, the estimated hazard ratio for the effect of each explanatory variable is  $\exp(b)$ , given all other variables are held constant, where  $b$  is the coefficient estimate for that variable. The coefficient estimates are found by maximizing the likelihood function of the model. The likelihood function for the proportional hazards regression model is based on the observed order of events. It is the product of likelihood of a failure estimated for each failure time. If there are  $n$  failures at  $n$  distinct failure times, then the likelihood is

$$L = \left[ \frac{h(t_1)}{\sum_{i=1}^n h(t_i)} \right] \times \left[ \frac{h(t_2)}{\sum_{i=2}^n h(t_i)} \right] \times \dots \times \left[ \frac{h(t_n)}{h(t_n)} \right].$$

You can use a likelihood ratio test to assess the significance of adding a term or terms in a model. Consider the two models where the first model has  $p$  predictive variables and the second model has  $p + r$  predictive variables. Then, comparing the two models,  $-2 \ln(L_1/L_2)$  has a chi-square distribution with  $r$  degrees of freedom (the number of terms being tested).

## References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.

[2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.

[3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.

## See Also

`ecdf` | `coxphfit` | `ksdensity`

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 12-19
- “Survivor Functions for Two Groups” on page 12-26
- “Cox Proportional Hazards Model for Censored Data” on page 12-34

## Concepts

- “What Is Survival Analysis?” on page 12-2
- “Kaplan-Meier Method” on page 12-11

## Cox Proportional Hazards Model for Censored Data

This example shows how to construct a Cox proportional hazards model, and assess the significance of the predictor variables.

### Step 1. Load sample data.

Navigate to a folder containing sample data.

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

Load the sample data.

```
load readmissiontimes
```

The response variable is `Readmission Time`, which shows the readmission times for 100 patients. The predictor variables are `Age`, `Sex`, `Weight`, and the smoking status of each patient, `Smoker`. 1 indicates the patient is a smoker, and 0 indicates that the patient does not smoke. The column vector `Censored` has the censorship information for each patient, where 1 indicates censored data, and 0 indicates the exact readmission times are observed. This is simulated data.

### Step 2. Fit Cox proportional hazards function.

Fit a Cox proportional hazard function with the variable `Sex` as the predictor variable, taking the censoring into account.

```
X = Sex;
[b,logl,H,stats] = coxphfit(X,ReadmissionTime,...
'censoring',Censored);
```

Assess the statistical significance of the term `Sex`.

```
stats

stats =

    covb: 0.1016
    beta: -1.7642
```



```

se: 0.3188
z: -5.5335
p: 3.1392e-08

```

The  $p$ -value,  $p$ , indicates that the term Sex is statistically significant.

Save the loglikelihood value with a different name. You will use this to assess the significance of the extended models.

```

loglSex = logl
loglSex =
-262.1365

```

### Step 3. Add Age and Weight to the model.

Fit a Cox proportional hazards model with the variables Sex, Age, and Weight.

```

X = [Sex Age Weight];
[b,logl,H,stats] = coxphfit(X,ReadmissionTime,...
'censoring',Censored);

```

Assess the significance of the terms.

```

stats.beta
ans =
-0.5441
0.0143
0.0250

```

```

stats.p
ans =
0.4953
0.3842
0.0960

```

None of the terms, adjusted for others, is statistically significant.

Assess the significance of the terms using the log likelihood ratio. You can assess the significance of the new model using the likelihood ratio statistic. First find the difference between the log-likelihood statistic of the model without the terms Age and Weight and the log-likelihood of the model with Sex, Age, and Weight.

```
-2*[loglSex - logl]
```

```
ans =
```

```
3.6705
```

Now, compute the  $p$ -value for the likelihood ratio statistic. The likelihood ratio statistic has a Chi-square distribution with a degrees of freedom equal to the number of predictor variables being assessed. In this case, the degrees of freedom is 2.

```
p = 1 - cdf('chi2',3.6705,2)
```

```
p =
```

```
0.1596
```

The  $p$ -value of 0.1596 indicates that the terms Age and Weight are not statistically significant, given the term Sex in the model.

#### **Step 4. Add Smoker to the model.**

Fit a Cox proportional hazards model with the variables Sex and Smoker.

```
X = [Sex Smoker];  
[b,logl,H,stats] = coxphfit(X,ReadmissionTime,...  
'censoring',Censored);
```

Assess the significance of the terms in the model.

```
stats.p
```

```
ans =
```

```
0.0000
```

```
0.0148
```

Compare this model to the first model where Sex is the only term.

```
-2*[loglSex - logl]
```

```
ans =
```

```
5.5789
```

Compute the  $p$ -value for the likelihood ratio statistic. The likelihood ratio statistic has a Chi-square distribution with a degree of freedom of 1.

```
p = 1 - cdf('chi2',5.5789,1)
```

```
p =
```

```
0.0182
```

The  $p$ -value of 0.0182 indicates that Sex and Smoker are statistically significant given the other is in the model. The model with Sex and Smoker is a better fit compared to the model with only Sex.

Request the coefficient estimates.

```
stats.beta
```

```
ans =
```

```
-1.7165
```

```
0.6338
```

The default baseline is the mean of  $X$ , so the final model for the hazard ratio is

$$HR = \frac{h_X(t)}{h_{\bar{X}}(t)} = \exp\left[\beta_s (X_s - \bar{X}_s) + \beta_a (X_a - \bar{X}_a)\right].$$

Fit a Cox ph model with a baseline of 0.

```
X = [Sex Smoker];
```

```
[b,logl,H,stats] = coxphfit(X,ReadmissionTime,...
```

```
'censoring',Censored,'baseline',0);
```

The model for the hazard ratio is

$$HR = \frac{h_X(t)}{h_0(t)} = \exp[\beta_s X_s + \beta_a X_a].$$

Request the coefficient estimates.

```
stats.beta
```

```
ans =
```

```
-1.7165  
0.6338
```

The coefficients are not affected, but the hazard rate differs from when the baseline is the mean of  $X$ .

## See Also

```
ecdf | coxphfit | ksdensity
```

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 12-19
- “Survivor Functions for Two Groups” on page 12-26

## Concepts

- “What Is Survival Analysis?” on page 12-2
- “Kaplan-Meier Method” on page 12-11
- “Cox Proportional Hazards Regression” on page 12-31

# Multivariate Methods

---

- “Introduction to Multivariate Methods” on page 13-2
- “Multivariate Linear Regression” on page 13-3
- “Estimation of Multivariate Regression Models” on page 13-6
- “Set Up Multivariate Regression Problems” on page 13-16
- “Multivariate General Linear Model” on page 13-30
- “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35
- “Longitudinal Analysis” on page 13-43
- “Multidimensional Scaling” on page 13-50
- “Procrustes Analysis” on page 13-62
- “Feature Selection” on page 13-71
- “Feature Transformation” on page 13-76
- “Partial Least Squares Regression and Principal Components Regression” on page 13-106

## Introduction to Multivariate Methods

Large, high-dimensional data sets are common in the modern era of computer-based instrumentation and electronic data storage. High-dimensional data present many challenges for statistical visualization, analysis, and modeling.

Data visualization, of course, is impossible beyond a few dimensions. As a result, pattern recognition, data preprocessing, and model selection must rely heavily on numerical methods.

A fundamental challenge in high-dimensional data analysis is the so-called *curse of dimensionality*. Observations in a high-dimensional space are necessarily sparser and less representative than those in a low-dimensional space. In higher dimensions, data over-represent the edges of a sampling distribution, because regions of higher-dimensional space contain the majority of their volume near the surface. (A  $d$ -dimensional spherical shell has a volume, relative to the total volume of the sphere, that approaches 1 as  $d$  approaches infinity.) In high dimensions, typical data points at the interior of a distribution are sampled less frequently.

Often, many of the dimensions in a data set—the measured features—are not useful in producing a model. Features may be irrelevant or redundant. Regression and classification algorithms may require large amounts of storage and computation time to process raw data, and even if the algorithms are successful the resulting models may contain an incomprehensible number of terms.

Because of these challenges, multivariate statistical methods often begin with some type of *dimension reduction*, in which data are approximated by points in a lower-dimensional space. Dimension reduction is the goal of the methods presented in this chapter. Dimension reduction often leads to simpler models and fewer measured variables, with consequent benefits when measurements are expensive and visualization is important.

## Multivariate Linear Regression

### In this section...

“Multivariate Linear Regression Model” on page 13-3

“Solving Multivariate Regression Problems” on page 13-4

### Multivariate Linear Regression Model

The multivariate linear regression model expresses a  $d$ -dimensional continuous response vector as a linear combination of predictor terms plus a vector of error terms with a multivariate normal distribution. Let

$\mathbf{y}_i = (y_{i1}, \dots, y_{id})'$  denote the response vector for observation  $i$ ,  $i = 1, \dots, n$ . In the most general case, given the  $d$ -by- $K$  design matrix  $\mathbf{X}_i$  and the  $K$ -by-1 vector of coefficients  $\beta$ , the multivariate linear regression model is

$$\mathbf{y}_i = \mathbf{X}_i \beta + \varepsilon_i,$$

where the  $d$ -dimensional vector of error terms follows a multivariate normal distribution,

$$\varepsilon_i \square MVN_d(\mathbf{0}, \Sigma).$$

The model assumes independence between observations, meaning the error variance-covariance matrix for the  $n$  stacked  $d$ -dimensional response vectors is

$$\mathbf{I}_n \otimes \Sigma = \begin{pmatrix} \Sigma & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \Sigma \end{pmatrix}.$$

If  $\mathbf{Y}$  denotes the  $nd$ -by-1 vector of stacked  $d$ -dimensional responses, and  $\mathbf{X}$  denotes the  $nd$ -by- $K$  matrix of stacked design matrices, then the distribution of the response vector is

$$\mathbf{y} \square MVN_{nd}(\mathbf{X}\beta, \mathbf{I}_n \otimes \Sigma).$$

## Solving Multivariate Regression Problems

To fit multivariate linear regression models of the form

$$\mathbf{y}_i = \mathbf{X}_i\boldsymbol{\beta} + \boldsymbol{\varepsilon}_i, \boldsymbol{\varepsilon}_i \sim MVN_d(\mathbf{0}, \boldsymbol{\Sigma})$$

in Statistics Toolbox, use `mvregress`. This function fits multivariate regression models with a diagonal (heteroscedastic) or unstructured (heteroscedastic and correlated) error variance-covariance matrix,  $\boldsymbol{\Sigma}$ , using least squares or maximum likelihood estimation.

Many variations of multivariate regression might not initially appear to be of the form supported by `mvregress`, such as:

- Multivariate general linear model
- Multivariate analysis of variance (MANOVA)
- Longitudinal analysis
- Panel data analysis
- Seemingly unrelated regression (SUR)
- Vector autoregressive (VAR) model

In many cases, you can frame these problems in the form used by `mvregress` (but `mvregress` does not support parameterized error variance-covariance matrices). For the special case of one-way MANOVA, you can alternatively use `manova1`. Econometrics Toolbox™ has functions for VAR estimation.

---

**Note** The multivariate linear regression model is distinct from the multiple linear regression model, which models a *univariate* continuous response as a linear combination of exogenous terms plus an independent and identically distributed error term. To fit a multiple linear regression model, use `LinearModel.fit`.

---

### See Also

`manova1` | `mvregress` | `mvregresslike` | `LinearModel.fit` |



**Related  
Examples**

- “Set Up Multivariate Regression Problems” on page 13-16
- “Multivariate General Linear Model” on page 13-30
- “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35
- “Longitudinal Analysis” on page 13-43

**Concepts**

- “Estimation of Multivariate Regression Models” on page 13-6

## Estimation of Multivariate Regression Models

### In this section...

- “Least Squares Estimation” on page 13-6
- “Maximum Likelihood Estimation” on page 13-10
- “Missing Response Data” on page 13-12

### Least Squares Estimation

- “Ordinary Least Squares” on page 13-6
- “Covariance-Weighted Least Squares” on page 13-7
- “Error Covariance Estimation” on page 13-8
- “Feasible Generalized Least Squares” on page 13-9
- “Panel Corrected Standard Errors” on page 13-10

### Ordinary Least Squares

When you fit multivariate linear regression models using `mvregress`, you can use the optional name-value pair `'algorithm'`, `'cwlsl'` to choose least squares estimation. In this case, by default, `mvregress` returns ordinary least squares (OLS) estimates using  $\Sigma = \mathbf{I}_d$ . Alternatively, if you specify a covariance matrix for weighting, you can return covariance-weighted least squares (CWLS) estimates. If you combine OLS and CWLS, you can get feasible generalized least squares (FGLS) estimates.

The OLS estimate for the coefficient vector is the vector  $\mathbf{b}$  that minimizes

$$\sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i \mathbf{b})' (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}).$$

Let  $\mathbf{y}$  denote the  $nd$ -by-1 vector of stacked  $d$ -dimensional responses, and  $\mathbf{X}$  denote the  $nd$ -by- $K$  matrix of stacked design matrices. The  $K$ -by-1 vector of OLS regression coefficient estimates is

$$\mathbf{b}_{OLS} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}.$$

This is the first `mvregress` output.

Given  $\Sigma = \mathbf{I}_d$  (the `mvregress` OLS default), the variance-covariance matrix of the OLS estimates is

$$V(\mathbf{b}_{OLS}) = (\mathbf{X}'\mathbf{X})^{-1}.$$

This is the fourth `mvregress` output. The standard errors of the OLS regression coefficients are the square root of the diagonal of this variance-covariance matrix.

If your data is not scaled such that  $\Sigma = \sigma^2\mathbf{I}_d$ , then you can multiply the `mvregress` variance-covariance matrix by the mean squared error (MSE), an unbiased estimate of  $\sigma^2$ . To compute the MSE, return the  $n$ -by- $d$  matrix of residuals,  $\mathbf{E}$  (the third `mvregress` output). Then,

$$\text{MSE} = \frac{\sum_{i=1}^n \mathbf{e}_i \mathbf{e}_i'}{n - K},$$

where  $\mathbf{e}_i = (\mathbf{y}_i - \mathbf{X}_i\boldsymbol{\beta})'$  is the  $i$ th row of  $\mathbf{E}$ .

### Covariance-Weighted Least Squares

For most multivariate problems, an identity error covariance matrix is insufficient, and leads to inefficient or biased standard error estimates. You can specify a matrix for CWLS estimation using the optional name-value pair argument `covar0`, for example, an invertible  $d$ -by- $d$  matrix named  $\mathbf{C}_0$ .

Usually,  $\mathbf{C}_0$  is a diagonal matrix such that the inverse matrix  $\mathbf{C}_0^{-1}$  contains weights for each dimension to model heteroscedasticity. However,  $\mathbf{C}_0$  can also be a nondiagonal matrix that models correlation.

Given  $\mathbf{C}_0$ , the CWLS solution is the vector  $\mathbf{b}$  that minimizes

$$\sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i \mathbf{b})' \mathbf{C}_0 (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}).$$

In this case, the  $K$ -by-1 vector of CWLS regression coefficient estimates is

$$\mathbf{b}_{CWLS} = \left( \mathbf{X}' (\mathbf{I}_n \otimes \mathbf{C}_0)^{-1} \mathbf{X} \right)^{-1} \mathbf{X}' (\mathbf{I}_n \otimes \mathbf{C}_0)^{-1} \mathbf{y}.$$

This is the first `mvregress` output.

If  $\Sigma = \mathbf{C}_0$ , this is the generalized least squares (GLS) solution. The corresponding variance-covariance matrix of the CWLS estimates is

$$V(\mathbf{b}_{CWLS}) = \left( \mathbf{X}' (\mathbf{I}_n \otimes \mathbf{C}_0)^{-1} \mathbf{X} \right)^{-1}.$$

This is the fourth `mvregress` output. The standard errors of the CWLS regression coefficients are the square root of the diagonal of this variance-covariance matrix.

If you only know the error covariance matrix up to a proportion, that is,

$\Sigma = \sigma^2 \mathbf{C}_0$ , you can multiply the `mvregress` variance-covariance matrix by the MSE, as described in “Ordinary Least Squares” on page 13-6.

### Error Covariance Estimation

Regardless of which least squares method you use, the estimate for the error variance-covariance matrix is

$$\hat{\Sigma} = \begin{pmatrix} \hat{\sigma}_1^2 & \hat{\sigma}_{12} & \cdots & \hat{\sigma}_{1d} \\ \hat{\sigma}_{12} & \hat{\sigma}_2^2 & \cdots & \hat{\sigma}_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\sigma}_{1d} & \hat{\sigma}_{2d} & \cdots & \hat{\sigma}_d^2 \end{pmatrix} = \frac{\mathbf{E}' \mathbf{E}}{n},$$

where  $\mathbf{E}$  is the  $n$ -by- $d$  matrix of residuals. The  $i$ th row of  $\mathbf{E}$  is

$$\mathbf{e}_i = (\mathbf{y}_i - \mathbf{X}_i \mathbf{b})'.$$

The error covariance estimate,  $\hat{\Sigma}$ , is the second `mvregress` output, and the matrix of residuals,  $\mathbf{E}$ , is the third output. If you specify the optional name-value pair 'covtype', 'diagonal', then `mvregress` returns  $\hat{\Sigma}$  with zeros in the off-diagonal entries,

$$\hat{\Sigma} = \begin{pmatrix} \hat{\sigma}_1^2 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \hat{\sigma}_d^2 \end{pmatrix}.$$

### Feasible Generalized Least Squares

The generalized least squares estimate is the CWLS estimate with a known covariance matrix. That is, given  $\Sigma$  is known, the GLS solution is

$$\mathbf{b}_{GLS} = \left( \mathbf{X}'(\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{X} \right)^{-1} \mathbf{X}'(\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{y},$$

with variance-covariance matrix

$$V(\mathbf{b}_{GLS}) = \left( \mathbf{X}'(\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{X} \right)^{-1}.$$

In most cases, the error covariance is unknown. The feasible generalized least squares (FGLS) estimate uses  $\hat{\Sigma}$  in place of  $\Sigma$ . You can obtain two-step FGLS estimates as follows:

- 1 Perform OLS regression, and return an estimate  $\hat{\Sigma}$ .
- 2 Perform CWLS regression, using  $\mathbf{C}_0 = \hat{\Sigma}$ .

You can also iterate between these two steps until convergence is reached.

For some data, the OLS estimate  $\hat{\Sigma}$  is positive semidefinite, and has no unique inverse. In this case, you cannot get the FGLS estimate using `mvregress`. As an alternative, you can use `lscov`, which uses a generalized inverse to return weighted least squares solutions for positive semidefinite covariance matrices.

### Panel Corrected Standard Errors

An alternative to FGLS is to use OLS coefficient estimates (which are consistent) and make a standard error correction to improve efficiency. One such standard error adjustment—which does not require inversion of the covariance matrix—is panel corrected standard errors (PCSE) [1]. The panel corrected variance-covariance matrix for OLS estimates is

$$V_{pcse}(\mathbf{b}_{OLS}) = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'(\mathbf{I}_n \otimes \Sigma)\mathbf{X}(\mathbf{X}'\mathbf{X}).$$

The PCSE are the square root of the diagonal of this variance-covariance matrix. “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35 illustrates PCSE computation.

### Maximum Likelihood Estimation

- “Maximum Likelihood Estimates” on page 13-10
- “Standard Errors” on page 13-11

### Maximum Likelihood Estimates

The default estimation algorithm used by `mvregress` is maximum likelihood estimation (MLE). The loglikelihood function for the multivariate linear regression model is

$$\begin{aligned} \log L(\beta, \Sigma | \mathbf{y}, \mathbf{X}) &= \frac{1}{2}nd \log(2\pi) + \frac{1}{2}n \log(\det(\Sigma)) \\ &+ \frac{1}{2} \sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i\beta)' \Sigma^{-1} (\mathbf{y}_i - \mathbf{X}_i\beta). \end{aligned}$$

The MLEs for  $\beta$  and  $\Sigma$  are the values that maximize the loglikelihood objective function.

`mvregress` finds the MLEs using an iterative two-stage algorithm. At iteration  $m + 1$ , the estimates are

$$\mathbf{b}_{MLE}^{(m+1)} = \left( \mathbf{X}' (\mathbf{I}_n \otimes \Sigma^{(m)})^{-1} \mathbf{X} \right)^{-1} \mathbf{X}' (\mathbf{I}_n \otimes \Sigma^{(m)})^{-1} \mathbf{y}$$

and

$$\hat{\Sigma}^{(m+1)} = \frac{1}{n} \sum_{i=1}^n (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}_{MLE}^{(m+1)}) (\mathbf{y}_i - \mathbf{X}_i \mathbf{b}_{MLE}^{(m+1)})'.$$

The algorithm terminates when the changes in the coefficient estimates and loglikelihood objective function are less than a specified tolerance, or when the specified maximum number of iterations is reached. The optional name-value pair arguments for changing these convergence criteria are `tolbeta`, `tolobj`, and `maxiter`, respectively.

### Standard Errors

The variance-covariance matrix of the MLEs is an optional `mvregress` output. By default, `mvregress` returns the variance-covariance matrix for only the regression coefficients, but you can also get the variance-covariance matrix of  $\hat{\Sigma}$  using the optional name-value pair `'vartype', 'full'`. In this case, `mvregress` returns the variance-covariance matrix for all  $K$  regression coefficients, and  $d$  or  $d(d+1)/2$  covariance terms (depending on whether the error covariance is diagonal or full).

By default, the variance-covariance matrix is the inverse of the observed Fisher information matrix (the `'hessian'` option). You can request the expected Fisher information matrix using the optional name-value pair `'vartype', 'fisher'`. Provided there is no missing response data, the observed and expected Fisher information matrices are the same. If response data is missing, the observed Fisher information accounts for the added uncertainty due to the missing values, whereas the expected Fisher information matrix does not.

The variance-covariance matrix for the regression coefficient MLEs is

$$V(\mathbf{b}_{MLE}) = \left( \mathbf{X}' (\mathbf{I}_n \otimes \Sigma)^{-1} \mathbf{X} \right)^{-1},$$

evaluated at the MLE of the error covariance matrix. This is the fourth `mvregress` output. The standard errors of the MLEs are the square root of the diagonal of this variance-covariance matrix.

For  $\hat{\Sigma}$ , let  $\theta$  denote the vector of parameters in the estimated error variance-covariance matrix. For example, if  $d = 2$ , then:

- If the estimated covariance matrix is diagonal, then  $\theta = (\sigma_1^2, \sigma_2^2)$ .
- If the estimated covariance matrix is full, then  $\theta = (\sigma_1^2, \sigma_{12}, \sigma_2^2)$ .

The Fisher information matrix for  $\theta$ ,  $I(\theta)$ , has elements

$$I(\theta)_{u,v} = \frac{1}{2} \text{tr} \left( \Sigma^{-1} \frac{\partial \Sigma}{\partial \theta_u} \Sigma^{-1} \frac{\partial \Sigma}{\partial \theta_v} \right), \quad u, v = 1, \dots, n_\theta,$$

where  $n_\theta$  is the length of  $\theta$  (either  $d$  or  $d(d + 1)/2$ ). The resulting variance-covariance matrix is

$$V(\theta) = I(\theta)^{-1}.$$

When you request the full variance-covariance matrix, `mvregress` returns (as the fourth output) the block diagonal matrix

$$\begin{pmatrix} V(\mathbf{b}_{MLE}) & \mathbf{0} \\ \mathbf{0} & V(\theta) \end{pmatrix}.$$

## Missing Response Data

- “Expectation/Conditional Maximization” on page 13-12
- “Observed Information Matrix” on page 13-14

### Expectation/Conditional Maximization

If any response values are missing, indicated by NaN, `mvregress` uses an expectation/conditional maximization (ECM) algorithm for estimation (if



enough data is available). In this case, the algorithm is iterative for both least squares and maximum likelihood estimation. During each iteration, `mvregress` imputes missing response values using their conditional expectation.

Consider organizing the data so that the joint distribution of the missing and observed responses, denoted  $\tilde{\mathbf{y}}$  and  $\mathbf{y}$  respectively, can be written as

$$\begin{pmatrix} \tilde{\mathbf{y}} \\ \mathbf{y} \end{pmatrix} \square MVN \left\{ \begin{pmatrix} \tilde{\mathbf{X}}\beta \\ \mathbf{X}\beta \end{pmatrix}, \begin{pmatrix} \Sigma_{\tilde{y}} & \Sigma_{\tilde{y}y} \\ \Sigma_{y\tilde{y}} & \Sigma_y \end{pmatrix} \right\}.$$

Using properties of the multivariate normal distribution, the conditional expectation of the missing responses given the observed responses is

$$E(\tilde{\mathbf{y}}|\mathbf{y}) = \tilde{\mathbf{X}}\beta + \Sigma_{\tilde{y}y}\Sigma_y^{-1}(\mathbf{y} - \mathbf{X}\beta).$$

Also, the variance-covariance matrix of the conditional distribution is

$$\text{COV}(\tilde{\mathbf{y}}|\mathbf{y}) = \Sigma_{\tilde{y}} - \Sigma_{\tilde{y}y}\Sigma_y^{-1}\Sigma_{y\tilde{y}}.$$

At each iteration of the ECM algorithm, `mvregress` uses the parameter values from the previous iteration to:

- Update the regression coefficients using the combined vector of observed responses and conditional expectations of missing responses.
- Update the variance-covariance matrix, adjusting for missing responses using the variance-covariance matrix of the conditional distribution.

Finally, the residuals that `mvregress` returns for missing responses are the difference between the conditional expectation and the fitted value, both evaluated at the final parameter estimates.

If you prefer to ignore any observations that have missing response values, use the name-value pair `'algorithm', 'mvm'`. Note that `mvregress` always ignores observations that have missing predictor values.

### Observed Information Matrix

By default, `mvregress` uses the observed Fisher information matrix (the 'hessian' option) to compute the variance-covariance matrix of the regression parameters. This accounts for the additional uncertainty due to missing response values.

The observed information matrix includes contributions from only the observed responses. That is, the observed Fisher information matrix for the parameters in the error variance-covariance matrix has elements

$$I(\theta)_{u,v} = \frac{1}{2} \sum_{i=1}^n \text{tr} \left( \Sigma_i^{-1} \frac{\partial \Sigma_i}{\partial \theta_u} \Sigma_i^{-1} \frac{\partial \Sigma_i}{\partial \theta_v} \right), \quad u, v = 1, \dots, n_\theta,$$

where  $\hat{\Sigma}_i$  is the subset of  $\hat{\Sigma}$  corresponding to the observed responses in  $\mathbf{Y}_i$ .

For example, if  $d = 3$ , but  $y_{i2}$  is missing, then

$$\hat{\Sigma}_i = \begin{pmatrix} \hat{\sigma}_1^2 & \hat{\sigma}_{13} \\ \hat{\sigma}_{13} & \hat{\sigma}_3^2 \end{pmatrix}.$$

The observed Fisher information for the regression coefficients has similar contributions from the design and covariance matrices.

## References

[1] Beck, N. and J. N. Katz. *What to Do (and Not to Do) with Time-Series-Cross-Section Data in Comparative Politics*. American Political Science Review, Vol. 89, No. 3, pp. 634–647, 1995.

### See Also

`mvregress` | `mvregresslike`

### Related Examples

- “Set Up Multivariate Regression Problems” on page 13-16
- “Multivariate General Linear Model” on page 13-30
- “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35
- “Longitudinal Analysis” on page 13-43

**Concepts**

- “Multivariate Linear Regression” on page 13-3

## Set Up Multivariate Regression Problems

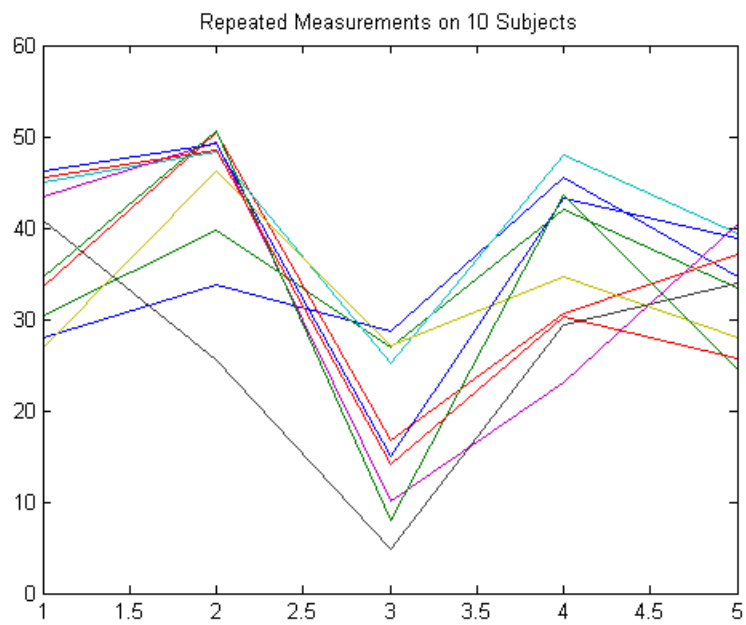
In this section...
“Response Matrix” on page 13-16
“Design Matrices” on page 13-21
“Common Multivariate Regression Problems” on page 13-22

### Response Matrix

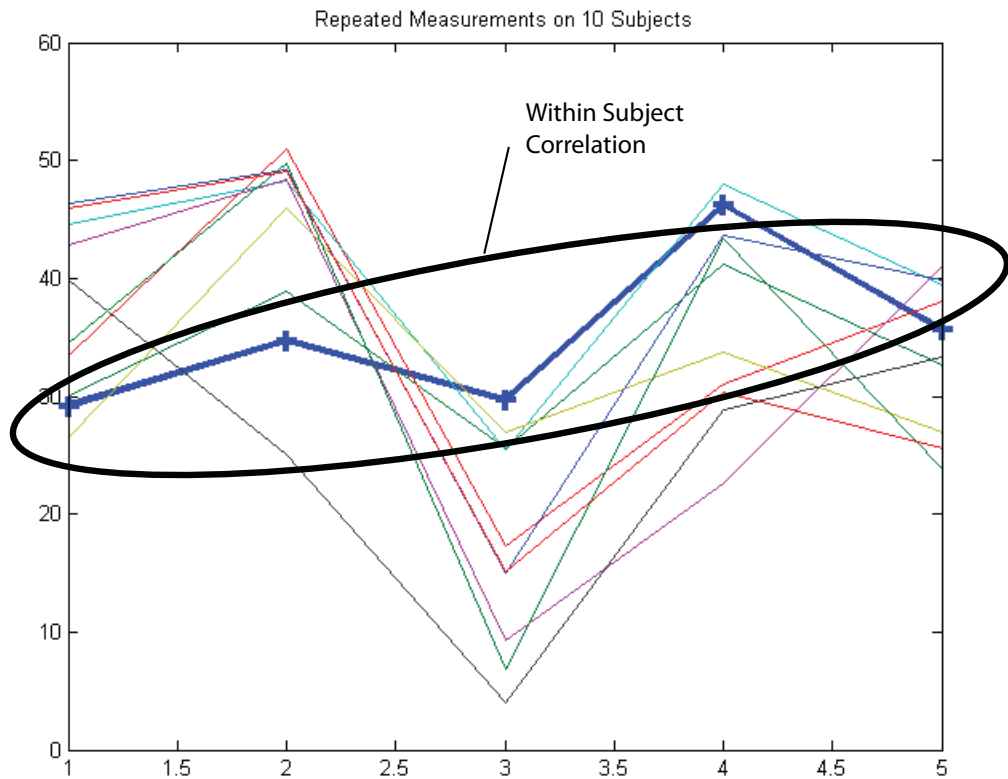
To fit a multivariate linear regression model using `mvregress`, you must set up your response matrix and design matrices in a particular way. Given properly formatted inputs, `mvregress` can handle a variety of multivariate regression problems.

`mvregress` expects the  $n$  observations of potentially correlated  $d$ -dimensional responses to be in an  $n$ -by- $d$  matrix, named  $Y$ , for example. That is, set up your responses so that the dependency structure is between observations in the same *row*. If you specify  $Y$  as a vector of length  $n$  (either a row or column vector), then `mvregress` assumes that  $d = 1$ , and treats the elements as  $n$  independent observations. It does *not* model the vector as one realization of a correlated series (such as a time series).

To illustrate how to set up a response matrix, suppose that your multivariate responses are repeated measurements made on subjects at multiple time points, as in the following figure.



Suppose that observations within a subject are correlated.

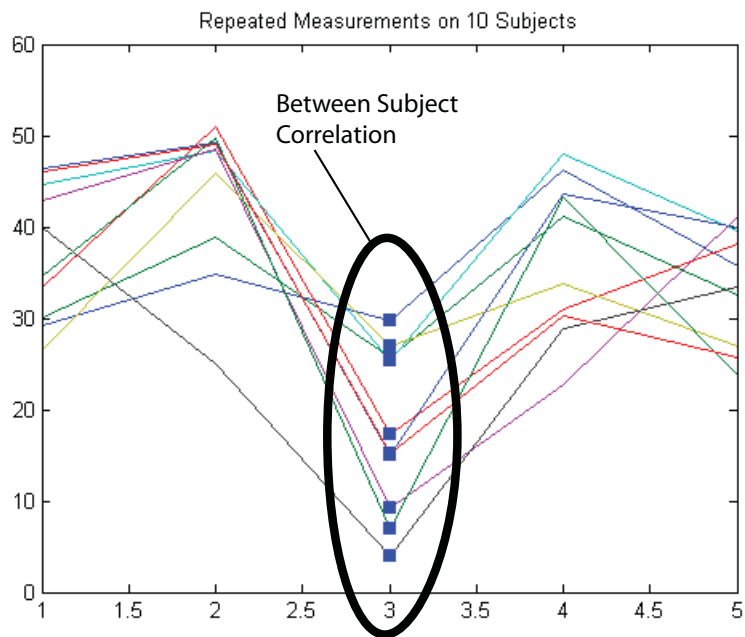


In this case, set up the response matrix  $Y$  such that each row corresponds to a subject, and each column corresponds to a time point.

$d = \text{Number of Time Points}$  $n = \text{Number of Subjects}$ 

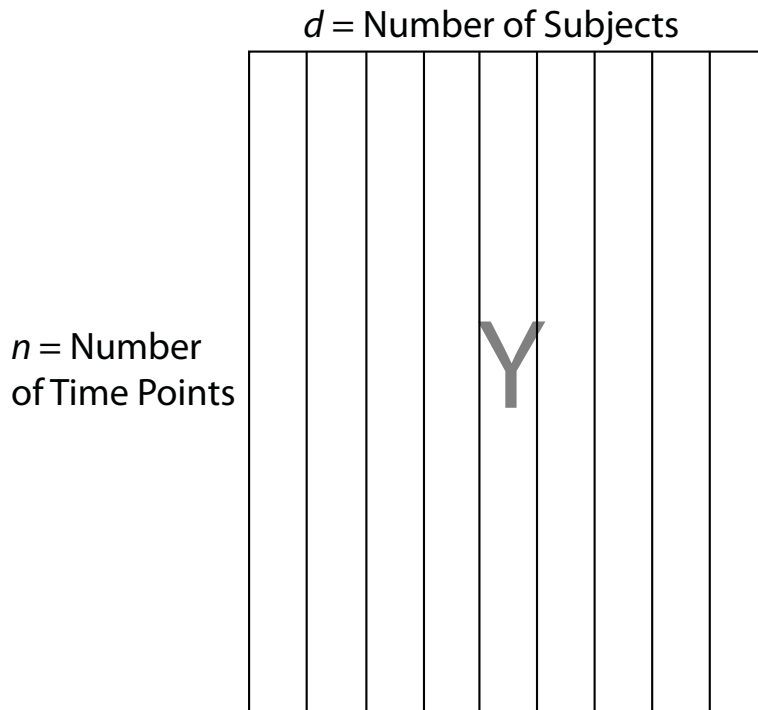
	Y

Then again, suppose that observations made on subjects at the same time are correlated (concurrent correlation).



In this case, set up the response matrix  $Y$  such that each row corresponds to a time point, and each column corresponds to a subject.





## Design Matrices

In the multivariate linear regression model, each  $d$ -dimensional response has a corresponding design matrix. Depending on the model, the design matrix might be comprised of exogenous predictor variables, dummy variables, lagged responses, or a combination of these and other covariate terms.

- If  $d > 1$  and all  $d$  dimensions have the same design matrix, then specify one  $n$ -by- $p$  design matrix, where  $p$  is the number of predictor variables. To determine an intercept for each dimension, add a column of ones to the design matrix. In this case, `mvregress` applies the design matrix to all  $d$  dimensions.
- If  $d > 1$  and all  $d$  dimensions do not have the same design matrix, then specify the design matrices using a length- $n$  cell array of  $d$ -by- $K$  arrays, named  $X$ , for example.  $K$  is the total number of regression coefficients in

the model. Note that the rows of the arrays in  $X$  correspond to the columns of the response matrix,  $Y$ .

$$X = \left\{ \begin{array}{c} \overleftarrow{K} \\ \boxed{\mathbf{X}_1} \\ \overrightarrow{d} \end{array} , \begin{array}{c} \overleftarrow{K} \\ \boxed{\mathbf{X}_2} \\ \overrightarrow{d} \end{array} , \dots , \begin{array}{c} \overleftarrow{K} \\ \boxed{\mathbf{X}_n} \\ \overrightarrow{d} \end{array} \right\}$$

If all  $n$  observations have the same design matrix, you can specify a cell array containing one  $d$ -by- $K$  design matrix. In this case, `mvregress` applies the design matrix to all  $n$  observations. For example, this situation might arise if the predictors are functions of time, and all observations were measured at the same time points.

- In the special case that  $d = 1$ , you can specify one  $n$ -by- $K$  design matrix (not in a cell array). However, you should consider using `fitlm` to fit regression models to univariate, continuous responses.

The following sections illustrate how to set up the some common multivariate regression problems for estimation using `mvregress`.

## Common Multivariate Regression Problems

- “Multivariate General Linear Model” on page 13-22
- “Longitudinal Analysis” on page 13-25
- “Panel Analysis” on page 13-26
- “Seemingly Unrelated Regression” on page 13-27
- “Vector Autoregressive Model” on page 13-28

### Multivariate General Linear Model

The multivariate general linear model is of the form

$$\mathbf{Y}_{n \times d} = \mathbf{X}_{n \times (p+1)} \mathbf{B}_{(p+1) \times d} + \mathbf{E}_{n \times d}.$$

In expanded form,

$$\begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1d} \\ y_{21} & y_{22} & \cdots & y_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nd} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix} \begin{bmatrix} \beta_{01} & \beta_{02} & \cdots & \beta_{0d} \\ \beta_{11} & \beta_{12} & \cdots & \beta_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{p1} & \beta_{p2} & \cdots & \beta_{pd} \end{bmatrix} + \begin{bmatrix} \varepsilon_{11} & \varepsilon_{12} & \cdots & \varepsilon_{1d} \\ \varepsilon_{21} & \varepsilon_{22} & \cdots & \varepsilon_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ \varepsilon_{n1} & \varepsilon_{n2} & \cdots & \varepsilon_{nd} \end{bmatrix}.$$

That is, each  $d$ -dimensional response has an intercept and  $p$  predictor variables, and each dimension has its own set of regression coefficients. In this form, the least squares solution is  $\mathbf{B} = \mathbf{X} \backslash \mathbf{Y}$ . To estimate this model using `mvregress`, use the  $n$ -by- $d$  matrix of responses, as above.

If all  $d$  dimensions have the same design matrix, use the  $n$ -by- $(p+1)$  design matrix, as above. Adding a column of ones to the  $p$  predictor variables computes the intercept for each dimension.

If all  $d$  dimensions do not have the same design matrix, reformat the  $n$ -by- $(p+1)$  design matrix into a length- $n$  cell array of  $d$ -by- $K$  matrices. Here,  $K = (p+1)d$  for an intercept and slopes for each dimension.

For example, suppose  $n = 4$ ,  $d = 3$ , and  $p = 2$  (two predictor terms in addition to an intercept). This figure shows how to format the  $i$ th element in the cell array.

$$\begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \\ y_{41} & y_{42} & y_{43} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ 1 & x_{31} & x_{32} \\ 1 & x_{41} & x_{42} \end{bmatrix} \begin{bmatrix} \beta_{01} & \beta_{02} & \beta_{03} \\ \beta_{11} & \beta_{12} & \beta_{13} \\ \beta_{21} & \beta_{22} & \beta_{23} \end{bmatrix} + \begin{bmatrix} \varepsilon_{11} & \varepsilon_{12} & \varepsilon_{13} \\ \varepsilon_{21} & \varepsilon_{22} & \varepsilon_{23} \\ \varepsilon_{31} & \varepsilon_{32} & \varepsilon_{33} \\ \varepsilon_{41} & \varepsilon_{42} & \varepsilon_{43} \end{bmatrix}$$



$$\begin{bmatrix} 1 & 0 & 0 & x_{i1} & 0 & 0 & x_{i2} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{i1} & 0 & 0 & x_{i2} & 0 \\ 0 & 0 & 1 & 0 & 0 & x_{i1} & 0 & 0 & x_{i2} \end{bmatrix} \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{03} \\ \beta_{11} \\ \beta_{12} \\ \beta_{13} \\ \beta_{21} \\ \beta_{22} \\ \beta_{23} \end{bmatrix}$$

$X\{i\}$

If you prefer, you can reshape the  $K$ -by-1 vector of coefficients back into a  $(p + 1)$ -by- $d$  matrix after estimation.

To put constraints on the model parameters, adjust the design matrix accordingly. For example, suppose that the three dimensions in the previous example have a common slope. That is,  $\beta_{11} = \beta_{12} = \beta_{13} = \beta_1$  and  $\beta_{21} = \beta_{22} = \beta_{23} = \beta_2$ . In this case, each design matrix is 3-by-5, as shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & x_{i1} & x_{i2} \\ 0 & 1 & 0 & x_{i1} & x_{i2} \\ 0 & 0 & 1 & x_{i1} & x_{i2} \end{bmatrix}}_{X\{i\}} \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{03} \\ \beta_1 \\ \beta_2 \end{bmatrix}$$

### Longitudinal Analysis

In a longitudinal analysis, you might measure responses on  $n$  subjects at  $d$  time points, with correlation between observations made on the same subject. For example, suppose that you measure responses  $y_{ij}$  at times  $t_{ij}$ ,  $i = 1, \dots, n$  and  $j = 1, \dots, d$ . In addition, suppose that each subject is in one of two groups (such as male or female), specified by the indicator variable  $G_i$ . You could model  $y_{ij}$  as a function of  $G_i$  and  $t_{ij}$ , with group-specific intercepts and slopes, as follows:

$$y_{ij} = \beta_0 + \beta_1 G_i + \beta_2 t_{ij} + \beta_3 G_i \times t_{ij} + \varepsilon_{ij}, \quad i = 1, \dots, n; j = 1, \dots, d,$$

where

$$\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \square MVN(\mathbf{0}, \Sigma).$$

Most longitudinal models include time as an explicit predictor.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, where  $n$  is the number of subjects and  $d$  is the number of time points. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = 4$  for the four regression coefficients.

For example, suppose  $d = 5$  (five observations per subject). The  $i$ th design matrix and corresponding parameter vector for the specified model are shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & G_i & t_{i1} & G_i * t_{i1} \\ 1 & G_i & t_{i2} & G_i * t_{i2} \\ 1 & G_i & t_{i3} & G_i * t_{i3} \\ 1 & G_i & t_{i4} & G_i * t_{i4} \\ 1 & G_i & t_{i5} & G_i * t_{i5} \end{bmatrix}}_{X\{i\}} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

### Panel Analysis

In a panel analysis, you might measure responses and covariates on  $d$  subjects (such as individuals or countries) at  $n$  time points. For example, suppose you measure responses  $y_{tj}$  and covariates  $x_{tj}$  on subjects  $j = 1, \dots, d$  at times  $t = 1, \dots, n$ . A fixed effects panel model, with subject-specific fixed effects, and concurrent correlation might look like:

$$y_{tj} = \alpha_j + \beta x_{tj} + \varepsilon_{tj},$$

where

$$\varepsilon_t = (\varepsilon_{t1}, \dots, \varepsilon_{td})' \square MVN(\mathbf{0}, \Sigma).$$

In contrast to longitudinal models, the panel analysis model typically includes covariates measured at each time point, instead of using time as an explicit predictor.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, such that each column corresponds to a subject. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = d + 1$  for the  $d$  intercepts and a slope term.

For example, suppose  $d = 4$  (four subjects). The  $t$ th design matrix and corresponding parameter vector are shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & x_{t1} \\ 0 & 1 & 0 & 0 & x_{t2} \\ 0 & 0 & 1 & 0 & x_{t3} \\ 0 & 0 & 0 & 1 & x_{t4} \end{bmatrix}}_{X\{t\}} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \beta \end{bmatrix}$$

### Seemingly Unrelated Regression

In a seemingly unrelated regression (SUR), you model  $d$  separate regressions, each with its own intercept and slope, but a common error variance-covariance matrix. For example, suppose you measure responses  $y_{ij}$  and covariates  $x_{ij}$  for regression models  $j = 1, \dots, d$ , with  $i = 1, \dots, n$  observations to fit each regression. The SUR model might look like:

$$y_{ij} = \beta_{0j} + \beta_j x_{ij} + \varepsilon_{ij},$$

where

$$\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \square MVN(\mathbf{0}, \Sigma).$$

This model is very similar to the multivariate general linear model, except that it has different covariates for each dimension.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, such that each column has the data for the  $j$ th regression model. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = 2d$  for  $d$  intercepts and  $d$  slopes.

For example, suppose  $d = 3$  (three regressions). The  $i$ th design matrix and corresponding parameter vector are shown in the following figure.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & x_{i1} & 0 & 0 \\ 0 & 1 & 0 & 0 & x_{i2} & 0 \\ 0 & 0 & 1 & 0 & 0 & x_{i3} \end{bmatrix}}_{X\{i\}} \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{03} \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

### Vector Autoregressive Model

The VAR( $p$ ) vector autoregressive model expresses  $d$ -dimensional time series responses as a linear function of  $p$  lagged  $d$ -dimensional responses from previous times. For example, suppose you measure responses  $y_{ij}$  for time series  $j = 1, \dots, d$  at times  $t = 1, \dots, n$ . The VAR( $p$ ) model might look like:

$$\begin{bmatrix} y_{t1} \\ y_{t2} \\ \vdots \\ y_{td} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_d \end{bmatrix} + \begin{bmatrix} \varphi_{11}^{(1)} & \varphi_{12}^{(1)} & \dots & \varphi_{1d}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_{d1}^{(1)} & \varphi_{d2}^{(1)} & \dots & \varphi_{dd}^{(1)} \end{bmatrix} \begin{bmatrix} y_{t-1,1} \\ y_{t-1,2} \\ \vdots \\ y_{t-1,d} \end{bmatrix} + \dots + \begin{bmatrix} \varphi_{11}^{(p)} & \varphi_{12}^{(p)} & \dots & \varphi_{1d}^{(p)} \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_{d1}^{(p)} & \varphi_{d2}^{(p)} & \dots & \varphi_{dd}^{(p)} \end{bmatrix} \begin{bmatrix} y_{t-p,1} \\ y_{t-p,2} \\ \vdots \\ y_{t-p,d} \end{bmatrix} + \begin{bmatrix} \varepsilon_{t1} \\ \varepsilon_{t2} \\ \vdots \\ \varepsilon_{td} \end{bmatrix},$$

where

$$\varepsilon_t = (\varepsilon_{t1}, \dots, \varepsilon_{td})' \square MVN(\mathbf{0}, \Sigma).$$

When estimating vector autoregressive models, you typically need to use the first  $p$  observations to initiate the model, or provide some other presample response values.

To fit this model using `mvregress`, arrange the responses in an  $n$ -by- $d$  matrix, such that each column corresponds to a time series. Specify the design matrices in an  $n$ -length cell array of  $d$ -by- $K$  matrices, where here  $K = d + pd^2$ .

For example, suppose  $d = 2$  (two time series) and  $p = 1$  (one lag). The  $t$ th design matrix and corresponding parameter vector are shown in the following figure.



$$\begin{bmatrix} 1 & 0 & y_{t-1,1} & 0 & y_{t-1,2} & 0 \\ 0 & 1 & 0 & y_{t-1,1} & 0 & y_{t-1,2} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \phi_{11}^{(1)} \\ \phi_{21}^{(1)} \\ \phi_{12}^{(1)} \\ \phi_{22}^{(1)} \end{bmatrix}$$

$\underbrace{\hspace{10em}}_{X\{t\}}$

Alternatively, Econometrics Toolbox has functions for fitting and forecasting VAR( $p$ ) models, including the option to specify exogenous predictor variables.

## See Also

`mvregress` | `mvregresslike`

## Related Examples

- “Multivariate General Linear Model” on page 13-30
- “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35
- “Longitudinal Analysis” on page 13-43

## Concepts

- “Multivariate Linear Regression” on page 13-3
- “Estimation of Multivariate Regression Models” on page 13-6

## Multivariate General Linear Model

This example shows how to set up a multivariate general linear model for estimation using `mvregress`.

### **Load sample data.**

This data contains measurements on a sample of 205 auto imports from 1985.

Here, model the bivariate response of city and highway MPG (columns 14 and 15).

For predictors, use wheel base (column 3), curb weight (column 7), and fuel type (column 18). The first two predictors are continuous, and for this example are centered and scaled. Fuel type is a categorical variable with two categories (11 and 20), so a dummy indicator variable is needed for the regression.

```
load('imports-85')
Y = X(:,14:15);
[n,d] = size(Y);

X1 = zscore(X(:,3));
X2 = zscore(X(:,7));
X3 = X(:,18)==20;

Xmat = [ones(n,1) X1 X2 X3];
```

The variable `X3` is coded to have value 1 for the fuel type 20, and value 0 otherwise.

For convenience, the three predictors (wheel base, curb weight, and fuel type indicator) are combined into one design matrix, with an added intercept term.

### **Set up design matrices.**

Given these predictors, the multivariate general linear model for the bivariate MPG response is

$$\begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \\ \vdots & \vdots \\ y_{n1} & y_{n2} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} \\ 1 & x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & x_{n3} \end{bmatrix} \begin{bmatrix} \beta_{01} & \beta_{02} \\ \beta_{11} & \beta_{12} \\ \beta_{21} & \beta_{22} \\ \beta_{31} & \beta_{32} \end{bmatrix} + \begin{bmatrix} \varepsilon_{11} & \varepsilon_{12} \\ \varepsilon_{21} & \varepsilon_{22} \\ \vdots & \vdots \\ \varepsilon_{n1} & \varepsilon_{n2} \end{bmatrix},$$

where  $\varepsilon_i = (\varepsilon_{i1}, \varepsilon_{i2})' \square MVN(\mathbf{0}, \Sigma)$ . There are  $K = 8$  regression coefficients in total.

Create a length  $n = 205$  cell array of 2-by-8 ( $d$ -by- $K$ ) matrices for use with `mvregress`. The  $i$ th matrix in the cell array is

$$X\{i\} = \begin{bmatrix} 1 & 0 & x_{i1} & 0 & x_{i2} & 0 & x_{i3} & 0 \\ 0 & 1 & 0 & x_{i1} & 0 & x_{i2} & 0 & x_{i3} \end{bmatrix}.$$

```
Xcell = cell(1,n);
for i = 1:n
    Xcell{i} = [kron([Xmat(i,:)],eye(d))];
end
```

Given this specification of the design matrices, the corresponding parameter vector is

$$\beta = \begin{bmatrix} \beta_{01} \\ \beta_{02} \\ \beta_{11} \\ \beta_{12} \\ \beta_{21} \\ \beta_{22} \\ \beta_{31} \\ \beta_{32} \end{bmatrix}.$$

**Estimate regression coefficients.**

Fit the model using maximum likelihood estimation.

```
[beta,sigma,E,V] = mvregress(Xcell,Y);
```

```
beta
beta =
  33.5476
  38.5720
   0.9723
   0.3950
  -6.3064
  -6.3584
  -9.2284
  -8.6663
```

These coefficient estimates show:

- The expected city and highway MPG for cars of average wheel base, curb weight, and fuel type 11 are 33.5 and 38.6, respectively. For fuel type 20, the expected city and highway MPG are  $33.5476 - 9.2284 = 24.3192$  and  $38.5720 - 8.6663 = 29.9057$ .
- An increase of one standard deviation in curb weight has almost the same effect on expected city and highway MPG. Given all else is equal, the expected MPG decreases by about 6.3 with each one standard deviation increase in curb weight, for both city and highway MPG.
- For each one standard deviation increase in wheel base, the expected city MPG increases 0.972, while the expected highway MPG increases by only 0.395, given all else is equal.

**Compute standard errors.**

The standard errors for the regression coefficients are the square root of the diagonal of the variance-covariance matrix,  $V$ .

```
se = sqrt(diag(V))
se =
  0.7365
  0.7599
  0.3589
  0.3702
```

```

0.3497
0.3608
0.7790
0.8037

```

### Reshape coefficient matrix.

You can easily reshape the regression coefficients into the original 4-by-2 matrix.

```
B = reshape(beta,2,4)'
```

```
B =
```

```

33.5476  38.5720
 0.9723   0.3950
-6.3064 -6.3584
-9.2284 -8.6663

```

### Check model assumptions.

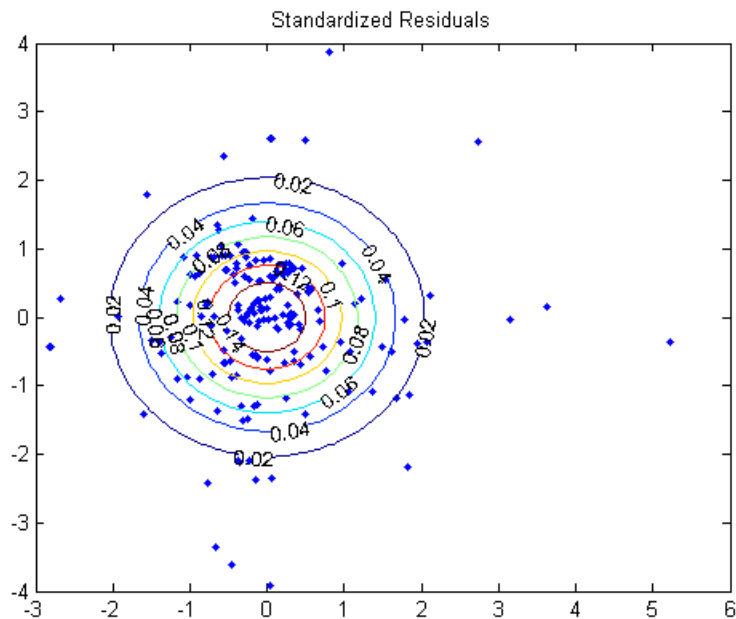
Under the model assumptions,  $\mathbf{z} = \mathbf{E}\Sigma^{-1/2}$  should be independent, with a bivariate standard normal distribution. In this 2-D case, you can assess the validity of this assumption using a scatter plot.

```

z = E/chol(sigma);
figure()
plot(z(:,1),z(:,2),'.')
title('Standardized Residuals')
hold on

% Overlay standard normal contours
z1 = linspace(-5,5);
z2 = linspace(-5,5);
[zx,zy] = meshgrid(z1,z2);
zgrid = [reshape(zx,100^2,1),reshape(zy,100^2,1)];
zn = reshape(mvnpdf(zgrid),100,100);
[c,h] = contour(zx,zy,zn);
clabel(c,h)

```



Several residuals are larger than expected, but overall, there is little evidence against the multivariate normality assumption.

**See Also**

`mvregress` | `mvregresslike`

**Related Examples**

- “Set Up Multivariate Regression Problems” on page 13-16
- “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35
- “Longitudinal Analysis” on page 13-43

**Concepts**

- “Multivariate Linear Regression” on page 13-3
- “Estimation of Multivariate Regression Models” on page 13-6

## Fixed Effects Panel Model with Concurrent Correlation

This example shows how to perform panel data analysis using `mvregress`. First, a fixed effects model with concurrent correlation is fit by ordinary least squares (OLS) to some panel data. Then, the estimated error covariance matrix is used to get panel corrected standard errors for the regression coefficients.

### Load sample data.

Navigate to the folder containing sample data. Load the sample panel data.

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

The dataset array, `panelData`, contains yearly observations on eight cities for 6 years. This is simulated data.

### Define variables.

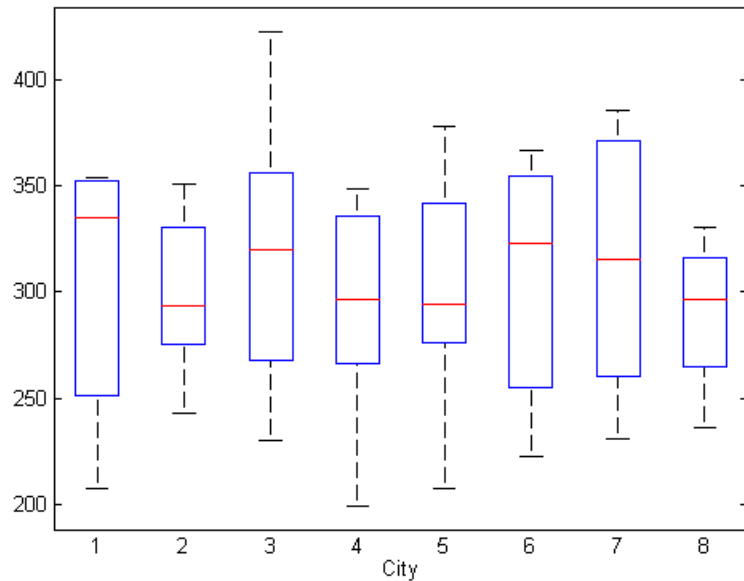
The first variable, `Growth`, measures economic growth (the response variable). The second and third variables are `city` and `year` indicators, respectively. The last variable, `Employ`, measures employment (the predictor variable).

```
y = panelData.Growth;
city = panelData.City;
year = panelData.Year;
x = panelData.Employ;
```

### Plot data grouped by category.

To look for potential city-specific fixed effects, create a box plot of the response grouped by city.

```
figure()
boxplot(y,city)
xlabel('City')
```



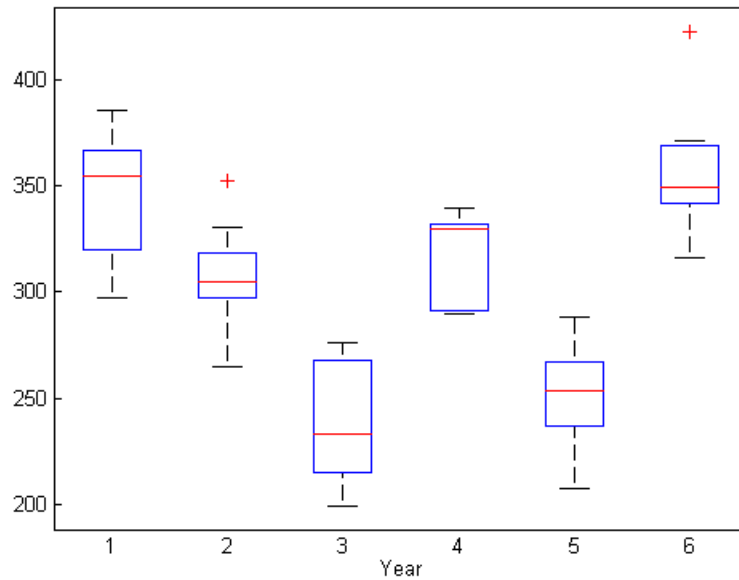
There does not appear to be any systematic differences in the mean response among cities.

**Plot data grouped by a different category.**

To look for potential year-specific fixed effects, create a box plot of the response grouped by year.

```
figure()  
boxplot(y,year)  
xlabel('Year')
```





Some evidence of systematic differences in the mean response between years seems to exist.

#### Format response data.

Let  $y_{ij}$  denote the response for city  $j = 1, \dots, d$ , in year  $i = 1, \dots, n$ . Similarly,  $x_{ij}$  is the corresponding value of the predictor variable. In this example,  $n = 6$  and  $d = 8$ .

Consider fitting a year-specific fixed effects model with a constant slope and concurrent correlation among cities in the same year,

$$y_{ij} = \alpha_i + \beta_1 x_{ij} + \varepsilon_{ij}, \quad i = 1, \dots, n, j = 1, \dots, d,$$

where  $\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \square MVN(\mathbf{0}, \Sigma)$ . The concurrent correlation accounts for any unmeasured, time-static factors that might impact growth similarly for some cities. For example, cities with close spatial proximity might be more likely to have similar economic growth.

To fit this model using `mvregress`, reshape the response data into an  $n$ -by- $d$  matrix.

```
n = 6; d = 8;
Y = reshape(y,n,d);
```

**Format design matrices.**

Create a length- $n$  cell array of  $d$ -by- $K$  design matrices. For this model, there are  $K = 7$  parameters ( $d = 6$  intercept terms and a slope).

Suppose the vector of parameters is arranged as

$$\beta = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_6 \\ \beta_1 \end{pmatrix}.$$

In this case, the first design matrix for year 1 looks like

$$X\{1\} = \begin{pmatrix} 1 & 0 & \cdots & 0 & x_{11} \\ 1 & 0 & \cdots & 0 & x_{12} \\ \vdots & \vdots & \cdots & 0 & \vdots \\ 1 & 0 & \cdots & 0 & x_{18} \end{pmatrix},$$

and the second design matrix for year 2 looks like

$$X\{2\} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & x_{21} \\ 0 & 1 & 0 & \cdots & 0 & x_{22} \\ \vdots & \vdots & 0 & \cdots & 0 & \vdots \\ 0 & 1 & 0 & \cdots & 0 & x_{28} \end{pmatrix}.$$

The design matrices for the remaining 4 years are similar.

```
K = 7; N = n*d;
X = cell(n,1);
for i = 1:n
    x0 = zeros(d,K-1);
```

```

    x0(:,i) = 1;
    X{i} = [x0,x(i:n:N)];
end

```

### Fit the model.

Fit the model using ordinary least squares (OLS).

```

[b,sig,E,V] = mvregress(X,Y,'algorithm','cqls');
b

```

b =

```

    41.6878
    26.1864
   -64.5107
    11.0924
   -59.1872
    71.3313
     4.9525

```

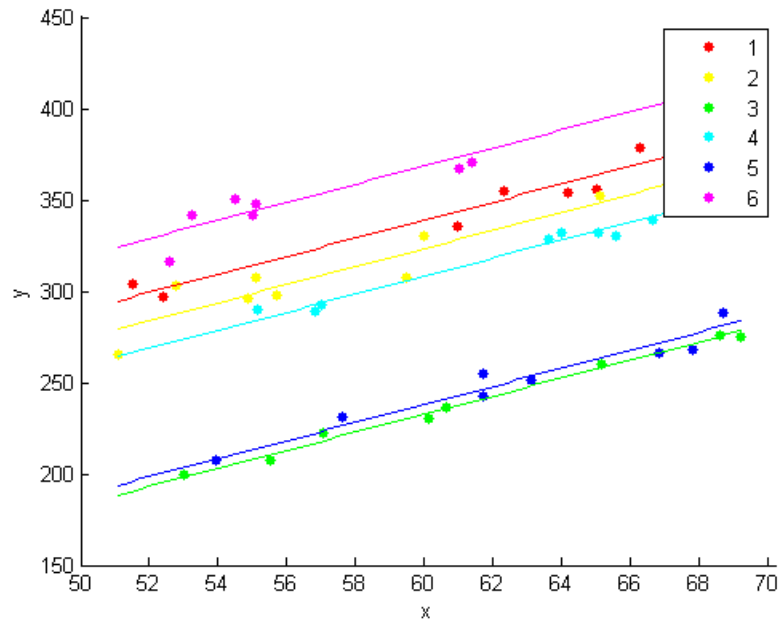
### Plot fitted model.

```

xx = linspace(min(x),max(x));
axx = repmat(b(1:K-1),1,length(xx));
bxx = repmat(b(K)*xx,n,1);
yhat = axx + bxx;

figure()
hPoints = gscatter(x,y,year);
hold on
hLines = plot(xx,yhat);
for i=1:n
    set(hLines(i),'color',get(hPoints(i),'color'));
end
hold off

```

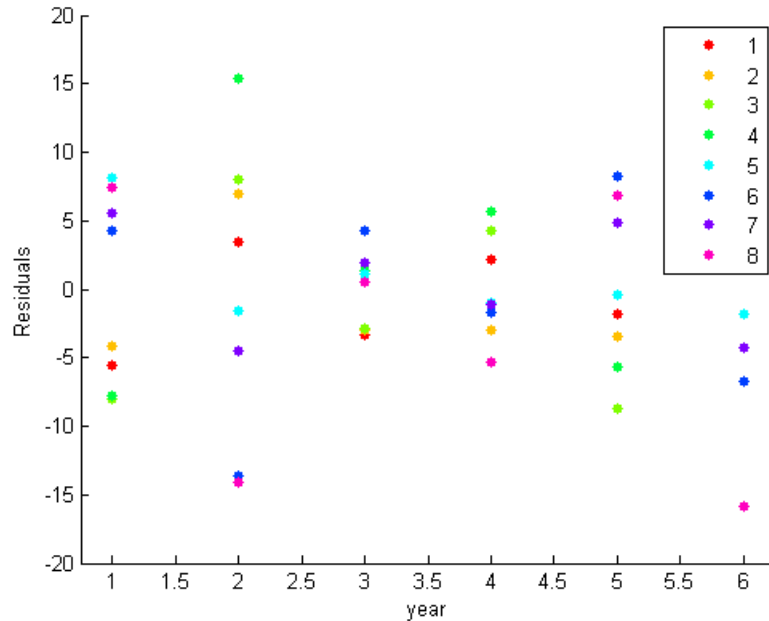


The model with year-specific intercepts and common slope appears to fit the data quite well.

**Residual correlation.**

Plot the residuals, grouped by year.

```
figure()  
gscatter(year,E(:),city)  
ylabel('Residuals')
```



The residual plot suggests concurrent correlation is present. For examples, cities 1, 2, 3, and 4 are consistently above or below average as a group in any given year. The same is true for the collection of cities 5, 6, 7, and 8. As seen in the exploratory plots, there are no systematic city-specific effects.

### Panel corrected standard errors.

Use the estimated error variance-covariance matrix to compute panel corrected standard errors for the regression coefficients.

```
XX = cell2mat(X);
S = kron(eye(n), sig);
Vpcse = inv(XX'*XX)*XX'*S*XX*inv(XX'*XX);
se = sqrt(diag(Vpcse))
```

se =

9.3750

8.6698  
9.3406  
9.4286  
9.5729  
8.8207  
0.1527

### See Also

`mvregress` | `mvregresslike`

### Related Examples

- “Set Up Multivariate Regression Problems” on page 13-16
- “Multivariate General Linear Model” on page 13-30
- “Longitudinal Analysis” on page 13-43

### Concepts

- “Multivariate Linear Regression” on page 13-3
- “Estimation of Multivariate Regression Models” on page 13-6

## Longitudinal Analysis

This example shows how to perform longitudinal analysis using `mvregress`.

### Load sample data.

Navigate to the folder containing sample data. Load the sample longitudinal data.

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

The matrix `Y` contains response data for 16 individuals. The response is the blood level of a drug measured at five time points ( $t = 0, 2, 4, 6,$  and  $8$ ). Each row of `Y` corresponds to an individual, and each column corresponds to a time point. The first eight subjects are female, and the second eight subjects are male. This is simulated data.

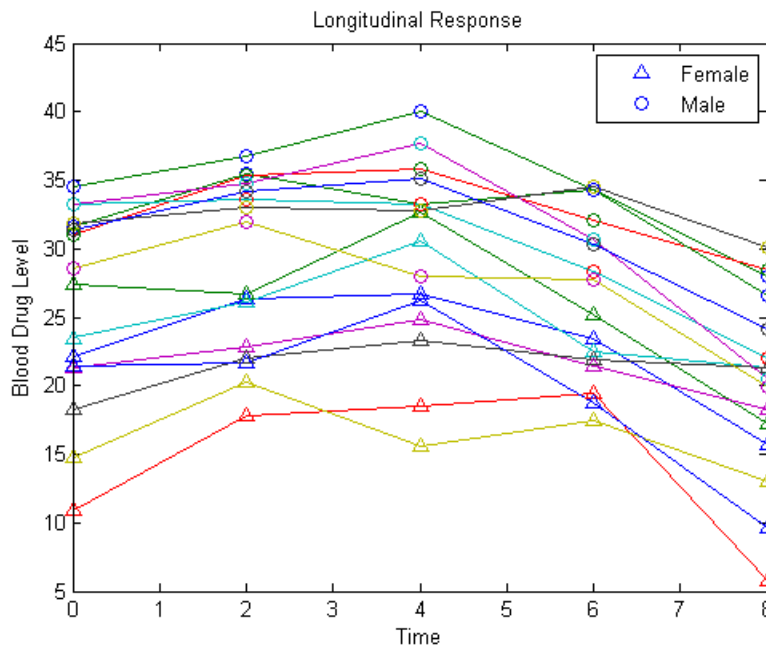
### Plot data.

Plot the data for all 16 subjects.

```
figure()
t = [0,2,4,6,8];
plot(t,Y)
hold on

hf = plot(t,Y(1:8,:), '^');
hm = plot(t,Y(9:16,:), 'o');
legend([hf(1),hm(1)], 'Female', 'Male', 'Location', 'NorthEast')

title('Longitudinal Response')
ylabel('Blood Drug Level')
xlabel('Time')
hold off
```



**Define design matrices.**

Let  $y_{ij}$  denote the response for individual  $i = 1, \dots, n$  measured at times  $t_{ij}$ ,  $j = 1, \dots, d$ . In this example,  $n = 16$  and  $d = 5$ . Let  $G_i$  denote the gender of individual  $i$ , where  $G_i = 1$  for males and 0 for females.

Consider fitting a quadratic longitudinal model, with a separate slope and intercept for each gender,

$$y_{ij} = \beta_0 + \beta_1 G_i + \beta_2 t_{ij} + \beta_3 t_{ij}^2 + \beta_4 G_i \times t_{ij} + \beta_5 G_i \times t_{ij}^2 + \varepsilon_{ij},$$

where  $\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \square MVN(\mathbf{0}, \Sigma)$ . The error correlation accounts for clustering within an individual.

To fit this model using `mvregress`, the response data should be in an  $n$ -by- $d$  matrix.  $Y$  is already in the proper format.



Next, create a length- $n$  cell array of  $d$ -by- $K$  design matrices. For this model, there are  $K = 6$  parameters.

For individual  $i$ , the 5-by-6 design matrix is

$$X\{i\} = \begin{pmatrix} 1 & G_i & t_{i1} & t_{i1}^2 & G_i \times t_{i1} & G_i \times t_{i1}^2 \\ 1 & G_i & t_{i2} & t_{i2}^2 & G_i \times t_{i2} & G_i \times t_{i2}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & G_i & t_{i5} & t_{i5}^2 & G_i \times t_{i5} & G_i \times t_{i5}^2 \end{pmatrix},$$

corresponding to the parameter vector

$$\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_5 \end{pmatrix}.$$

The matrix X1 has the design matrix for a female, and X2 has the design matrix for a male.

Create a cell array of design matrices. The first eight individuals are females, and the second eight are males.

```
X = cell(8,1);
X(1:8) = {X1};
X(9:16) = {X2};
```

**Fit the model.**

Fit the model using maximum likelihood estimation. Display the estimated coefficients and standard errors.

```
[b,sig,E,V,loglikF] = mvregress(X,Y);
[b sqrt(diag(V))]
```

ans =

```
18.8619    0.7432
13.0942    1.0511
```

```
2.5968    0.2845
-0.3771    0.0398
-0.5929    0.4023
0.0290    0.0563
```

The coefficients on the interaction terms (in the last two rows of  $b$ ) do not appear significant. You can use the value of the loglikelihood objective function for this fit, `loglikF`, to compare this model to one without the interaction terms using a likelihood ratio test.

**Plot fitted model.**

Plot the fitted lines for females and males.

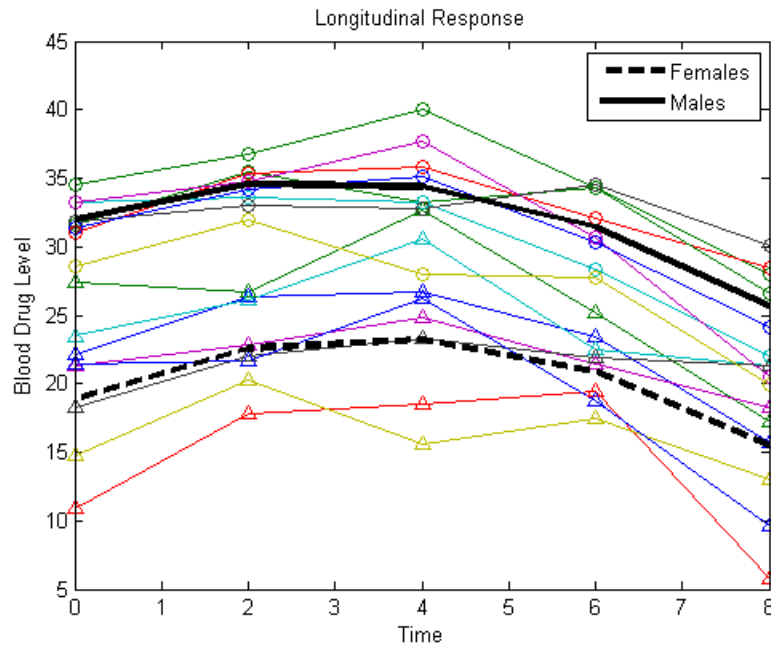
```
Yhatf = X1*b;
Yhatm = X2*b;

figure()
plot(t,Y)
hold on

plot(t,Y(1:8,:), '^', t, Y(9:16,:), 'o')

hf = plot(t, Yhatf, 'k--', 'LineWidth', 3);
hm = plot(t, Yhatm, 'k', 'LineWidth', 3);
legend([hf, hm], 'Females', 'Males', 'Location', 'NorthEast')

title('Longitudinal Response')
ylabel('Blood Drug Level')
xlabel('Time')
hold off
```



**Define a reduced model.**

Fit the model without interaction terms,

$$y_{ij} = \beta_0 + \beta_1 G_i + \beta_2 t_{ij} + \beta_3 t_{ij}^2 + \varepsilon_{ij},$$

where  $\varepsilon_i = (\varepsilon_{i1}, \dots, \varepsilon_{id})' \square MVN(\mathbf{0}, \Sigma)$ .

This model has four coefficients, which correspond to the first four columns of the design matrices X1 and X2 (for females and males, respectively).

```
X1R = X1(:, 1:4);
```

```
X2R = X2(:, 1:4);
```

```
XR = cell(8, 1);
```

```
XR(1:8) = {X1R};
```

```
XR(9:16) = {X2R};
```

**Fit the reduced model.**

Fit this model using maximum likelihood estimation. Display the estimated coefficients and their standard errors.

```
[bR,sigR,ER,VR,loglikR] = mvregress(XR,Y);
[bR,sqrt(diag(VR))]
```

```
ans =
```

```
    19.3765    0.6898
    12.0936    0.8591
     2.2919    0.2139
    -0.3623    0.0283
```

**Conduct a likelihood ratio test.**

Compare the two models using a likelihood ratio test. The null hypothesis is that the reduced model is sufficient. The alternative is that the reduced model is inadequate (compared to the full model with the interaction terms).

The likelihood ratio test statistic is compared to a chi-squared distribution with two degrees of freedom (for the two coefficients being dropped).

```
LR = 2*(loglikF-loglikR);
pval = 1 - chi2cdf(LR,2)
```

```
pval =
```

```
    0.0803
```

The  $p$ -value 0.0803 indicates that the null hypothesis is not rejected at the 5% significance level. Therefore, there is insufficient evidence that the extra terms improve the fit.

**See Also**

mvregress | mvregresslike

**Related Examples**

- “Set Up Multivariate Regression Problems” on page 13-16
- “Multivariate General Linear Model” on page 13-30
- “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35

**Concepts**

- “Multivariate Linear Regression” on page 13-3
- “Estimation of Multivariate Regression Models” on page 13-6

## Multidimensional Scaling

### In this section...

“Introduction to Multidimensional Scaling” on page 13-50

“Classical Multidimensional Scaling” on page 13-50

“Nonclassical Multidimensional Scaling” on page 13-56

“Nonmetric Multidimensional Scaling” on page 13-58

### Introduction to Multidimensional Scaling

One of the most important goals in visualizing data is to get a sense of how near or far points are from each other. Often, you can do this with a scatter plot. However, for some analyses, the data that you have might not be in the form of points at all, but rather in the form of pairwise similarities or dissimilarities between cases, observations, or subjects. There are no points to plot.

Even if your data are in the form of points rather than pairwise distances, a scatter plot of those data might not be useful. For some kinds of data, the relevant way to measure how near two points are might not be their Euclidean distance. While scatter plots of the raw data make it easy to compare Euclidean distances, they are not always useful when comparing other kinds of inter-point distances, city block distance for example, or even more general dissimilarities. Also, with a large number of variables, it is very difficult to visualize distances unless the data can be represented in a small number of dimensions. Some sort of dimension reduction is usually necessary.

Multidimensional scaling (MDS) is a set of methods that address all these problems. MDS allows you to visualize how near points are to each other for many kinds of distance or dissimilarity metrics and can produce a representation of your data in a small number of dimensions. MDS does not require raw data, but only a matrix of pairwise distances or dissimilarities.

### Classical Multidimensional Scaling

- “Introduction to Classical Multidimensional Scaling” on page 13-51

- “Example: Multidimensional Scaling” on page 13-53

## Introduction to Classical Multidimensional Scaling

This example shows how to use `cmdscale` to perform classical (metric) multidimensional scaling, also known as principal coordinates analysis.

`cmdscale` takes as an input a matrix of inter-point distances and creates a configuration of points. Ideally, those points are in two or three dimensions, and the Euclidean distances between them reproduce the original distance matrix. Thus, a scatter plot of the points created by `cmdscale` provides a visual representation of the original distances.

As a very simple example, you can reconstruct a set of points from only their inter-point distances. First, create some four dimensional points with a small component in their fourth coordinate, and reduce them to distances.

```
rng('default'); % For reproducibility
X = [normrnd(0,1,10,3),normrnd(0,.1,10,1)];
D = pdist(X,'euclidean');
```

Next, use `cmdscale` to find a configuration with those inter-point distances. `cmdscale` accepts distances as either a square matrix, or, as in this example, in the vector upper-triangular form produced by `pdist`.

```
[Y,eigvals] = cmdscale(D);
```

`cmdscale` produces two outputs. The first output, `Y`, is a matrix containing the reconstructed points. The second output, `eigvals`, is a vector containing the sorted eigenvalues of what is often referred to as the "scalar product matrix," which, in the simplest case, is equal to  $Y*Y'$ . The relative magnitudes of those eigenvalues indicate the relative contribution of the corresponding columns of `Y` in reproducing the original distance matrix `D` with the reconstructed points.

```
format short g
[eigvals eigvals/max(abs(eigvals))]
```

```
ans =
```

```
35.41
```

```
1
```

```

11.158      0.31511
 1.6894     0.04771
 0.1436     0.0040553
7.5802e-15  2.1407e-16
 6.299e-15  1.7789e-16
 2.1747e-15  6.1415e-17
-4.8934e-17 -1.3819e-18
-5.9631e-16 -1.684e-17
-2.8339e-15 -8.0032e-17

```

If `eigvals` contains only positive and zero (within round-off error) eigenvalues, the columns of `Y` corresponding to the positive eigenvalues provide an exact reconstruction of `D`, in the sense that their inter-point Euclidean distances, computed using `pdist`, for example, are identical (within round-off) to the values in `D`.

```
maxerr4 = max(abs(D - pdist(Y))) % Exact reconstruction
```

```
maxerr4 =
```

```
5.9952e-15
```

If two or three of the eigenvalues in `eigvals` are much larger than the rest, then the distance matrix based on the corresponding columns of `Y` nearly reproduces the original distance matrix `D`. In this sense, those columns form a lower-dimensional representation that adequately describes the data. However it is not always possible to find a good low-dimensional reconstruction.

```
maxerr3 = max(abs(D - pdist(Y(:,1:3)))) % Good reconstruction in 3D
```

```
maxerr2 = max(abs(D - pdist(Y(:,1:2)))) % Poor reconstruction in 2D
```

```
maxerr3 =
```

```
0.043142
```



```
maxerr2 =
    0.98315
```

The reconstruction in three dimensions reproduces  $D$  very well, but the reconstruction in two dimensions has errors that are of the same order of magnitude as the largest values in  $D$ .

```
max(max(D))
```

```
ans =
    5.8974
```

Often, `eigvals` contains some negative eigenvalues, indicating that the distances in  $D$  cannot be reproduced exactly. That is, there might not be any configuration of points whose inter-point Euclidean distances are given by  $D$ . If the largest negative eigenvalue is small in magnitude relative to the largest positive eigenvalues, then the configuration returned by `cmdscale` might still reproduce  $D$  well.

### Example: Multidimensional Scaling

This example shows how to construct a map of 10 US cities based on the distances between those cities, using `cmdscale`.

First, create the distance matrix and pass it to `cmdscale`. In this example,  $D$  is a full distance matrix: it is square and symmetric, has positive entries off the diagonal, and has zeros on the diagonal.

```
cities = ...
{'Atl', 'Chi', 'Den', 'Hou', 'LA', 'Mia', 'NYC', 'SF', 'Sea', 'WDC'};
D = [
    0  587 1212  701 1936  604  748 2139 2182  543;
    587  0  920  940 1745 1188  713 1858 1737  597;
    1212 920  0  879  831 1726 1631  949 1021 1494;
    701  940  879  0 1374  968 1420 1645 1891 1220;
```

```

1936 1745 831 1374 0 2339 2451 347 959 2300;
604 1188 1726 968 2339 0 1092 2594 2734 923;
748 713 1631 1420 2451 1092 0 2571 2408 205;
2139 1858 949 1645 347 2594 2571 0 678 2442;
2182 1737 1021 1891 959 2734 2408 678 0 2329;
543 597 1494 1220 2300 923 205 2442 2329 0];
[Y,eigvals] = cmdscale(D);

```

Next, look at the eigenvalues returned by `cmdscale`. Some of these are negative, indicating that the original distances are not Euclidean. This is because of the curvature of the earth.

```

format short g
[eigvals eigvals/max(abs(eigvals))]

```

```
ans =
```

```

9.5821e+06      1
1.6868e+06      0.17604
8157.3          0.0008513
1432.9          0.00014954
508.67         5.3085e-05
25.143         2.624e-06
4.0575e-10     4.2344e-17
-897.7         -9.3685e-05
-5467.6        -0.0005706
-35479         -0.0037026

```

However, in this case, the two largest positive eigenvalues are much larger in magnitude than the remaining eigenvalues. So, despite the negative eigenvalues, the first two coordinates of `Y` are sufficient for a reasonable reproduction of `D`.

```

Dtriu = D(find(tril(ones(10),-1)))';
maxrelerr = max(abs(Dtriu-pdist(Y(:,1:2))))./max(Dtriu)

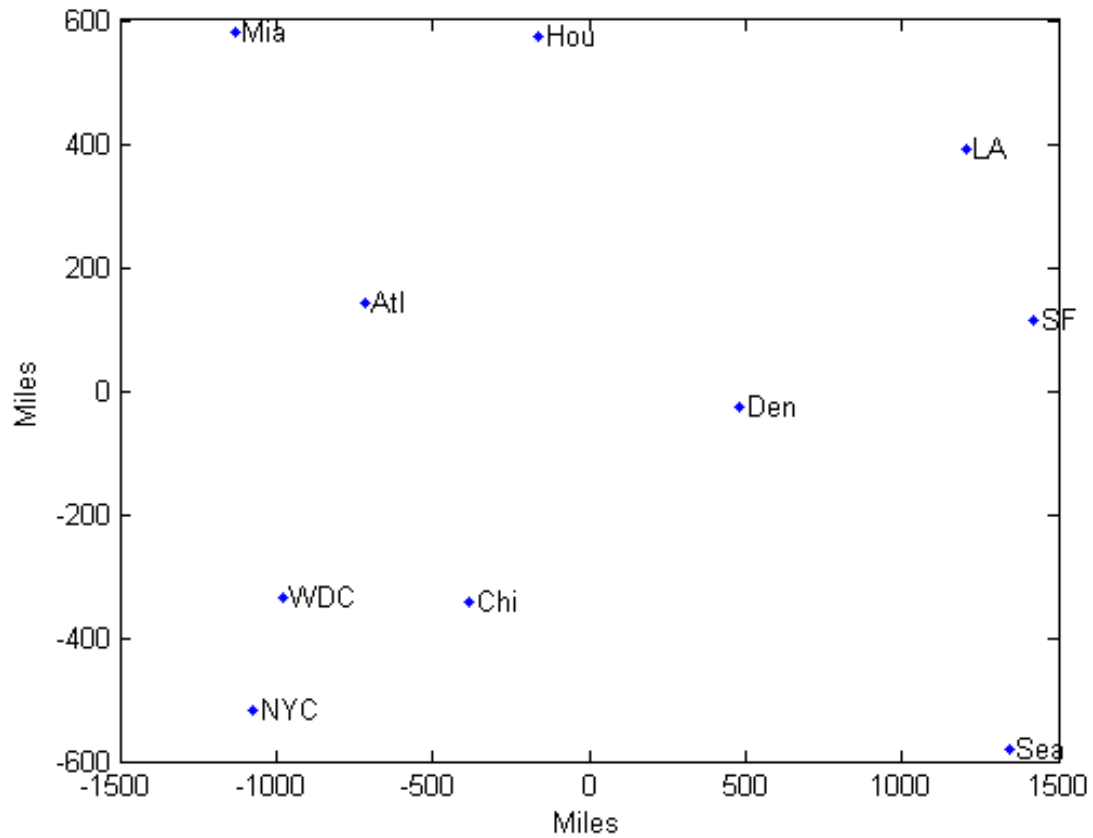
```

```
maxrelerr =
```

0.0075371

Here is a plot of the reconstructed city locations as a map. The orientation of the reconstruction is arbitrary.

```
plot(Y(:,1),Y(:,2),'.')
text(Y(:,1)+25,Y(:,2),cities)
xlabel('Miles')
ylabel('Miles')
```



## Nonclassical Multidimensional Scaling

The function `mdscale` performs nonclassical multidimensional scaling. As with `cmdscale`, you use `mdscale` either to visualize dissimilarity data for which no “locations” exist, or to visualize high-dimensional data by reducing its dimensionality. Both functions take a matrix of dissimilarities as an input and produce a configuration of points. However, `mdscale` offers a choice of different criteria to construct the configuration, and allows missing data and weights.

For example, the cereal data include measurements on 10 variables describing breakfast cereals. You can use `mdscale` to visualize these data in two dimensions. First, load the data. For clarity, this example code selects a subset of 22 of the observations.

```
load cereal.mat
X = [Calories Protein Fat Sodium Fiber ...
     Carbo Sugars Shelf Potass Vitamins];
% Take a subset from a single manufacturer
mfg1 = strcmp('G',cellstr(Mfg));
X = X(mfg1,:);
size(X)
ans =
    22 10
```

Then use `pdist` to transform the 10-dimensional data into dissimilarities. The output from `pdist` is a symmetric dissimilarity matrix, stored as a vector containing only the  $(23 \times 22/2)$  elements in its upper triangle.

```
dissimilarities = pdist(zscore(X),'cityblock');
size(dissimilarities)
ans =
    1 231
```

This example code first standardizes the cereal data, and then uses city block distance as a dissimilarity. The choice of transformation to dissimilarities is application-dependent, and the choice here is only for simplicity. In some applications, the original data are already in the form of dissimilarities.

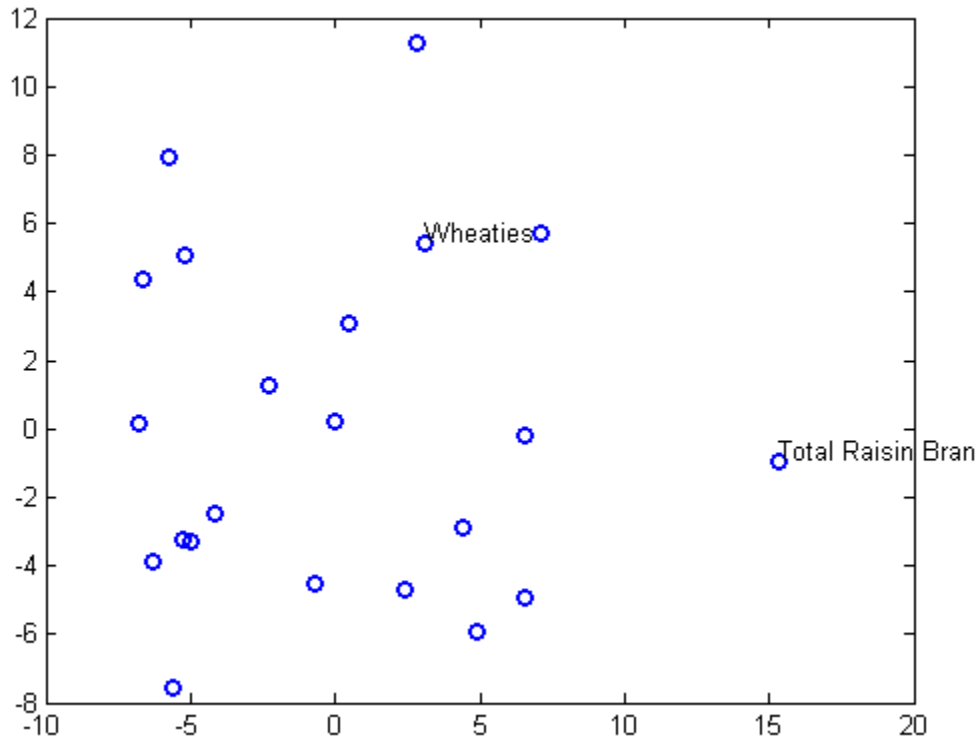
Next, use `mdscale` to perform metric MDS. Unlike `cmdscale`, you must specify the desired number of dimensions, and the method to use to construct

the output configuration. For this example, use two dimensions. The metric STRESS criterion is a common method for computing the output; for other choices, see the `mdscale` reference page in the online documentation. The second output from `mdscale` is the value of that criterion evaluated for the output configuration. It measures the how well the inter-point distances of the output configuration approximate the original input dissimilarities:

```
[Y, stress] = ...
mdscale(dissimilarities, 2, 'criterion', 'metricstress');
stress
stress =
    0.1856
```

A scatterplot of the output from `mdscale` represents the original 10-dimensional data in two dimensions, and you can use the `gname` function to label selected points:

```
plot(Y(:,1), Y(:,2), 'o', 'LineWidth', 2);
gname(Name(mfg1))
```



### Nonmetric Multidimensional Scaling

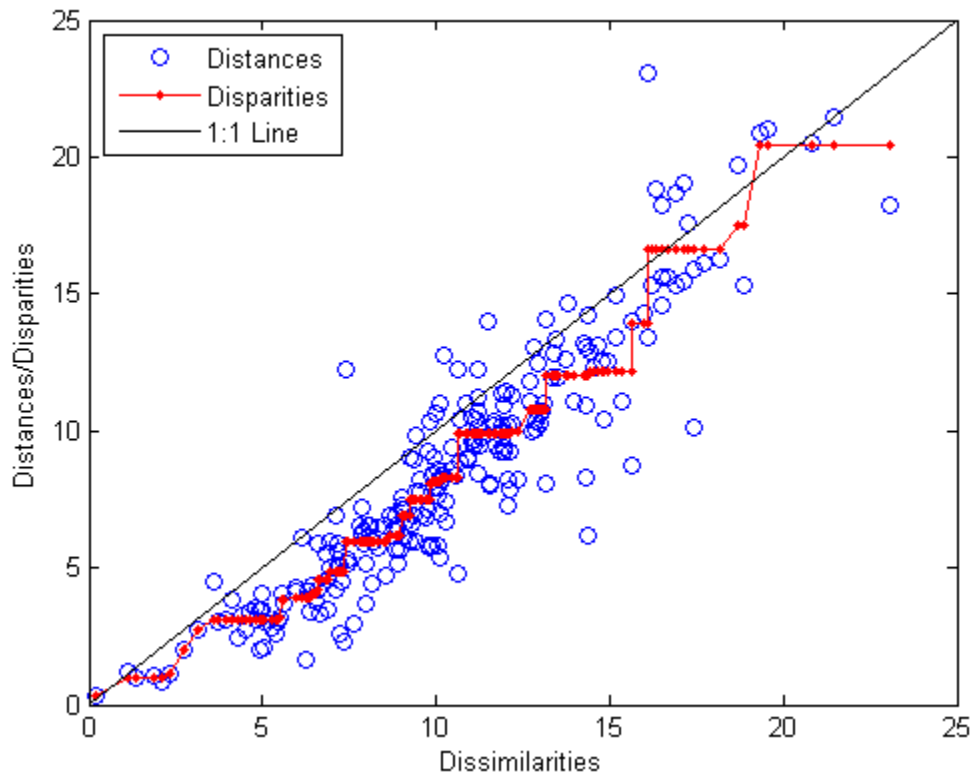
Metric multidimensional scaling creates a configuration of points whose inter-point distances approximate the given dissimilarities. This is sometimes too strict a requirement, and non-metric scaling is designed to relax it a bit. Instead of trying to approximate the dissimilarities themselves, non-metric scaling approximates a nonlinear, but monotonic, transformation of them. Because of the monotonicity, larger or smaller distances on a plot of the output will correspond to larger or smaller dissimilarities, respectively. However, the nonlinearity implies that `mdscale` only attempts to preserve the ordering of dissimilarities. Thus, there may be contractions or expansions of distances at different scales.

You use `mdscale` to perform nonmetric MDS in much the same way as for metric scaling. The nonmetric STRESS criterion is a common method for computing the output; for more choices, see the `mdscale` reference page in the online documentation. As with metric scaling, the second output from `mdscale` is the value of that criterion evaluated for the output configuration. For nonmetric scaling, however, it measures the how well the inter-point distances of the output configuration approximate the disparities. The disparities are returned in the third output. They are the transformed values of the original dissimilarities:

```
[Y, stress, disparities] = ...
mdscale(dissimilarities, 2, 'criterion', 'stress');
stress
stress =
    0.1562
```

To check the fit of the output configuration to the dissimilarities, and to understand the disparities, it helps to make a Shepard plot:

```
distances = pdist(Y);
[dum, ord] = sortrows([disparities(:) dissimilarities(:)]);
plot(dissimilarities, distances, 'bo', ...
     dissimilarities(ord), disparities(ord), 'r.-', ...
     [0 25], [0 25], 'k-')
xlabel('Dissimilarities')
ylabel('Distances/Disparities')
legend({'Distances' 'Disparities' '1:1 Line'}, ...
     'Location', 'NorthWest');
```



This plot shows that `mdscale` has found a configuration of points in two dimensions whose inter-point distances approximates the disparities, which in turn are a nonlinear transformation of the original dissimilarities. The concave shape of the disparities as a function of the dissimilarities indicates that fit tends to contract small distances relative to the corresponding dissimilarities. This may be perfectly acceptable in practice.

`mdscale` uses an iterative algorithm to find the output configuration, and the results can often depend on the starting point. By default, `mdscale` uses `cmdscale` to construct an initial configuration, and this choice often leads to a globally best solution. However, it is possible for `mdscale` to stop at a configuration that is a local minimum of the criterion. Such



cases can be diagnosed and often overcome by running `mdscale` multiple times with different starting points. You can do this using the `'start'` and `'replicates'` parameters. The following code runs five replicates of MDS, each starting at a different randomly-chosen initial configuration. The criterion value is printed out for each replication; `mdscale` returns the configuration with the best fit.

```
opts = statset('Display','final');  
[Y, stress] = ...  
mdscale(dissimilarities,2,'criterion','stress',...  
'start','random','replicates',5,'Options',opts);
```

```
35 iterations, Final stress criterion = 0.156209  
31 iterations, Final stress criterion = 0.156209  
48 iterations, Final stress criterion = 0.171209  
33 iterations, Final stress criterion = 0.175341  
32 iterations, Final stress criterion = 0.185881
```

Notice that `mdscale` finds several different local solutions, some of which do not have as low a stress value as the solution found with the `cmdscale` starting point.

## Procrustes Analysis

### In this section...

“Compare Landmark Data” on page 13-62

“Data Input” on page 13-62

“Preprocess Data for Accurate Results” on page 13-63

“Compare Handwritten Shapes” on page 13-64

### Compare Landmark Data

The `procrustes` function analyzes the distribution of a set of shapes using Procrustes analysis. This analysis method matches landmark data (geometric locations representing significant features in a given shape) to calculate the best shape-preserving Euclidian transformations. These transformations minimize the differences in location between compared landmark data.

Procrustes analysis is also useful in conjunction with multidimensional scaling. In “Example: Multidimensional Scaling” on page 13-53 there is an observation that the orientation of the reconstructed points is arbitrary. Two different applications of multidimensional scaling could produce reconstructed points that are very similar in principle, but that look different because they have different orientations. The `procrustes` function transforms one set of points to make them more comparable to the other.

### Data Input

The `procrustes` function takes two matrices as input:

- The target shape matrix  $X$  has dimension  $n \times p$ , where  $n$  is the number of landmarks in the shape and  $p$  is the number of measurements per landmark.
- The comparison shape matrix  $Y$  has dimension  $n \times q$  with  $q \leq p$ . If there are fewer measurements per landmark for the comparison shape than the target shape ( $q < p$ ), the function adds columns of zeros to  $Y$ , yielding an  $n \times p$  matrix.

The equation to obtain the transformed shape,  $Z$ , is

$$Z = bYT + c \quad (13-1)$$

where:

- $b$  is a scaling factor that stretches ( $b > 1$ ) or shrinks ( $b < 1$ ) the points.
- $T$  is the orthogonal rotation and reflection matrix.
- $c$  is a matrix with constant values in each column, used to shift the points.

The `procrustes` function chooses  $b$ ,  $T$ , and  $c$  to minimize the distance between the target shape  $X$  and the transformed shape  $Z$  as measured by the least squares criterion:

$$\sum_{i=1}^n \sum_{j=1}^p (X_{ij} - Z_{ij})^2$$

## Preprocess Data for Accurate Results

Procrustes analysis is appropriate when all  $p$  measurement dimensions have similar scales. The analysis would be inaccurate, for example, if the columns of  $Z$  had different scales:

- The first column is measured in milliliters ranging from 2,000 to 6,000.
- The second column is measured in degrees Celsius ranging from 10 to 25.
- The third column is measured in kilograms ranging from 50 to 230.

In such cases, standardize your variables by:

- 1 Subtracting the sample mean from each variable.
- 2 Dividing each resultant variable by its sample standard deviation.

Use the `zscore` function to perform this standardization.

## Compare Handwritten Shapes

In this example, use Procrustes analysis to compare two handwritten number threes. Visually and analytically explore the effects of forcing size and reflection changes as follows:

- “Step 1: Load and Display the Original Data” on page 13-64
- “Step 2: Calculate the Best Transformation” on page 13-65
- “Step 3: Examine the Similarity of the Two Shapes” on page 13-66
- “Step 4: Restrict the Form of the Transformations” on page 13-68

### Step 1: Load and Display the Original Data

Input landmark data for two handwritten number threes:

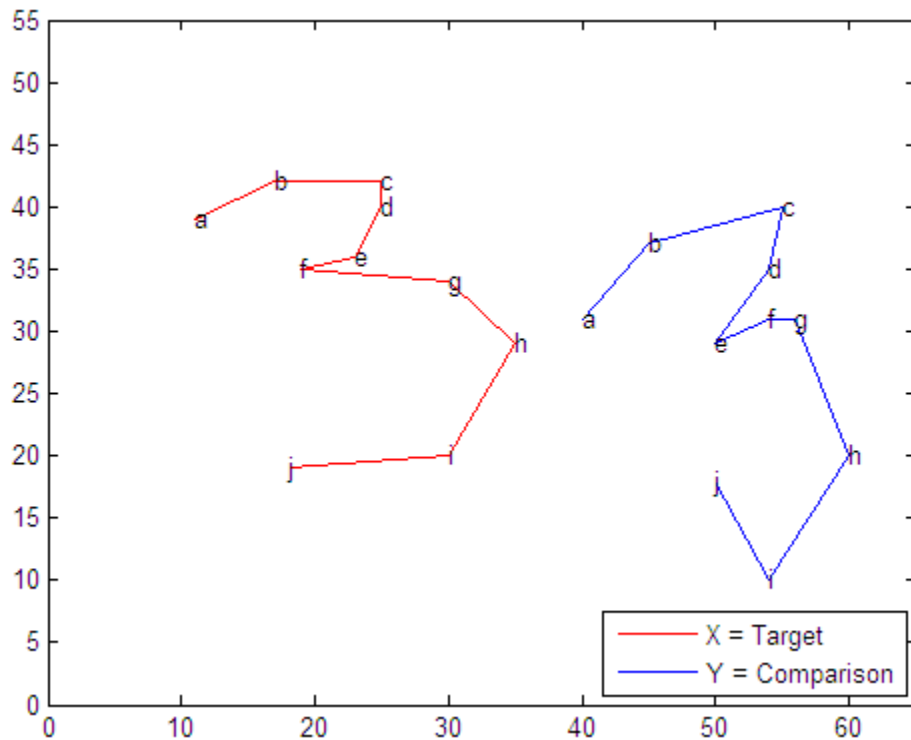
```
A = [11 39;17 42;25 42;25 40;23 36;19 35;30 34;35 29;...
30 20;18 19];
B = [15 31;20 37;30 40;29 35;25 29;29 31;31 31;35 20;...
29 10;25 18];
```

Create X and Y from A and B, moving B to the side to make each shape more visible:

```
X = A;
Y = B + repmat([25 0], 10,1);
```

Plot the shapes, using letters to designate the landmark points. Lines in the figure join the points to indicate the drawing path of each shape.

```
plot(X(:,1), X(:,2), 'r-', Y(:,1), Y(:,2), 'b-');
text(X(:,1), X(:,2), ('abcdefghij'))
text(Y(:,1), Y(:,2), ('abcdefghij'))
legend('X = Target', 'Y = Comparison', 'location', 'SE')
set(gca, 'YLim', [0 55], 'XLim', [0 65]);
```



## Step 2: Calculate the Best Transformation

Use Procrustes analysis to find the transformation that minimizes distances between landmark data points.

Call `procrustes` as follows:

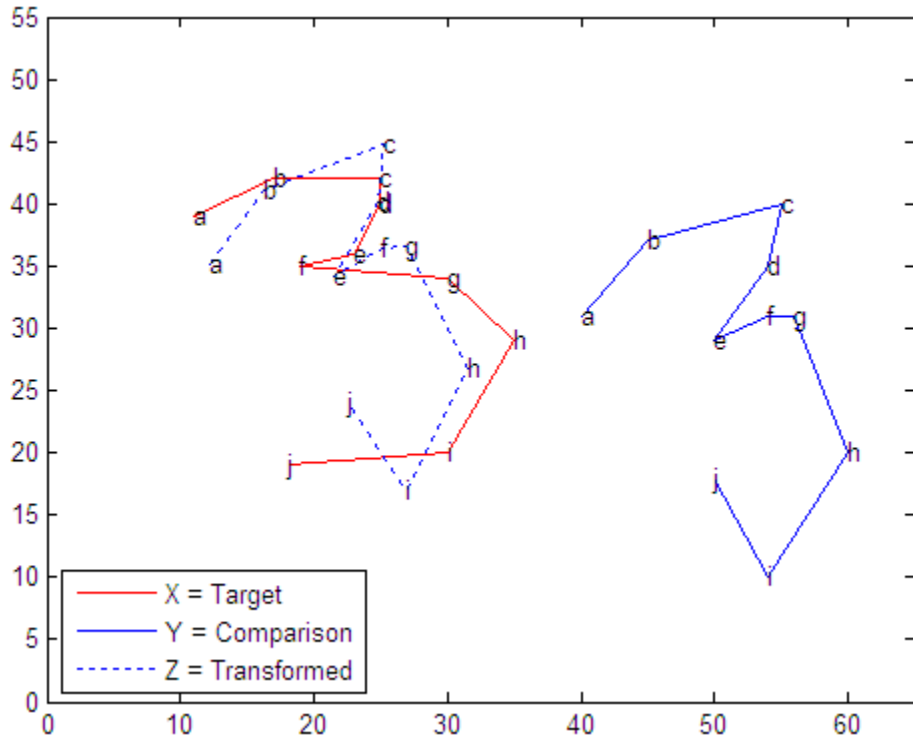
```
[d, Z, tr] = procrustes(X,Y);
```

The outputs of the function are:

- `d` – A standardized dissimilarity measure.)
- `Z` – A matrix of the transformed landmarks.
- `tr` – A structure array of the computed transformation with fields `T`, `b`, and `c` which correspond to the transformation equation, Equation 13-1.

Visualize the transformed shape, Z, using a dashed blue line:

```
plot(X(:,1), X(:,2), 'r-', Y(:,1), Y(:,2), 'b-', ...
     Z(:,1), Z(:,2), 'b:');
text(X(:,1), X(:,2), ('abcdefghij'))
text(Y(:,1), Y(:,2), ('abcdefghij'))
text(Z(:,1), Z(:,2), ('abcdefghij'))
legend('X = Target', 'Y = Comparison', ...
       'Z = Transformed', 'location', 'SW')
set(gca, 'YLim', [0 55], 'XLim', [0 65]);
```



**Step 3: Examine the Similarity of the Two Shapes**

Use two different numerical values to assess the similarity of the target shape and the transformed shape.

**Dissimilarity Measure d.** The dissimilarity measure  $d$  gives a number between 0 and 1 describing the difference between the target shape and the transformed shape. Values near 0 imply more similar shapes, while values near 1 imply dissimilarity. For this example:

```
d =
    0.1502
```

The small value of  $d$  in this case shows that the two shapes are similar.

`procrustes` calculates  $d$  by comparing the sum of squared deviations between the set of points with the sum of squared deviations of the original points from their column means:

```
numerator = sum(sum((X-Z).^2))
numerator =

    166.5321

denominator = sum(sum(bsxfun(@minus,X,mean(X)).^2))
denominator =

    1.1085e+003

ratio = numerator/denominator
ratio =

    0.1502
```

---

**Note** The resulting measure  $d$  is independent of the scale of the size of the shapes and takes into account only the similarity of landmark data. “Examine the Scaling Measure  $b$ ” on page 13-67 shows how to examine the size similarity of the shapes.

---

**Examine the Scaling Measure  $b$ .** The target and comparison threes in the previous figure visually show that the two numbers are of a similar size. The closeness of calculated value of the scaling factor  $b$  to 1 supports this observation as well:

```
tr.b
ans =
    0.9291
```

The sizes of the target and comparison shapes appear similar. This visual impression is reinforced by the value of  $b = 0.93$ , which implies that the best transformation results in shrinking the comparison shape by a factor .93 (only 7%).

#### Step 4: Restrict the Form of the Transformations

Explore the effects of manually adjusting the scaling and reflection coefficients.

**Fix the Scaling Factor  $b = 1$ .** Force  $b$  to equal 1 (set 'Scaling' to false) to examine the amount of dissimilarity in size of the target and transformed figures:

```
ds = procrustes(X,Y,'Scaling',false)
ds =
    0.1552
```

In this case, setting 'Scaling' to false increases the calculated value of  $d$  only 0.0049, which further supports the similarity in the size of the two number threes. A larger increase in  $d$  would have indicated a greater size discrepancy.

**Force a Reflection in the Transformation.** This example requires only a rotation, not a reflection, to align the shapes. You can show this by observing that the determinant of the matrix  $T$  is 1 in this analysis:

```
det(tr.T)
ans =
    1.0000
```

If you need a reflection in the transformation, the determinant of  $T$  is -1. You can force a reflection into the transformation as follows:

```
[dr,Zr,trr] = procrustes(X,Y,'Reflection',true);
dr
dr =
```

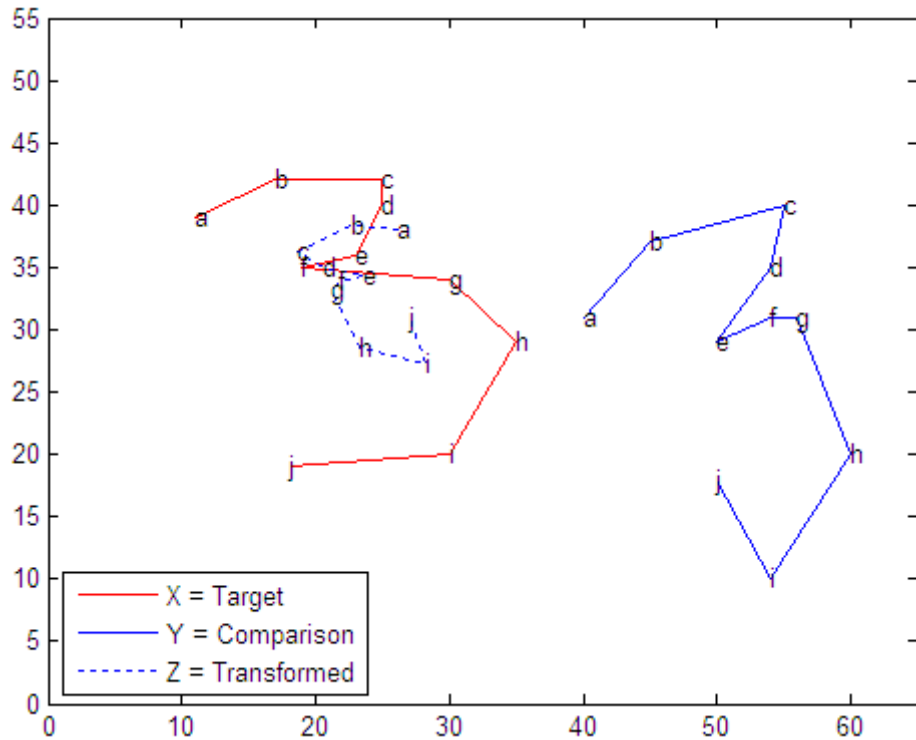


0.8130

The  $d$  value increases dramatically, indicating that a forced reflection leads to a poor transformation of the landmark points. A plot of the transformed shape shows a similar result:

- The landmark data points are now further away from their target counterparts.
- The transformed three is now an undesirable mirror image of the target three.

```
plot(X(:,1), X(:,2), 'r-', Y(:,1), Y(:,2), 'b-', ...  
Zr(:,1), Zr(:,2), 'b:');  
text(X(:,1), X(:,2), ('abcdefghij'))  
text(Y(:,1), Y(:,2), ('abcdefghij'))  
text(Zr(:,1), Zr(:,2), ('abcdefghij'))  
legend('X = Target', 'Y = Comparison', ...  
'Z = Transformed', 'location', 'SW')  
set(gca, 'YLim', [0 55], 'XLim', [0 65]);
```



It appears that the shapes might be better matched if you flipped the transformed shape upside down. Flipping the shapes would make the transformation even worse, however, because the landmark data points would be further away from their target counterparts. From this example, it is clear that manually adjusting the scaling and reflection parameters is generally not optimal.

## Feature Selection

In this section...
“Introduction to Feature Selection” on page 13-71
“Sequential Feature Selection” on page 13-71

### Introduction to Feature Selection

*Feature selection* reduces the dimensionality of data by selecting only a subset of measured features (predictor variables) to create a model. Selection criteria usually involve the minimization of a specific measure of predictive error for models fit to different subsets. Algorithms search for a subset of predictors that optimally model measured responses, subject to constraints such as required or excluded features and the size of the subset.

Feature selection is preferable to feature transformation when the original units and meaning of features are important and the modeling goal is to identify an influential subset. When categorical features are present, and numerical transformations are inappropriate, feature selection becomes the primary means of dimension reduction.

### Sequential Feature Selection

- “Introduction to Sequential Feature Selection” on page 13-71
- “Example: Sequential Feature Selection” on page 13-72

### Introduction to Sequential Feature Selection

A common method of feature selection is *sequential feature selection*. This method has two components:

- An objective function, called the *criterion*, which the method seeks to minimize over all feasible feature subsets. Common criteria are mean squared error (for regression models) and misclassification rate (for classification models).
- A sequential search algorithm, which adds or removes features from a candidate subset while evaluating the criterion. Since an exhaustive

comparison of the criterion value at all  $2^n$  subsets of an  $n$ -feature data set is typically infeasible (depending on the size of  $n$  and the cost of objective calls), sequential searches move in only one direction, always growing or always shrinking the candidate set.

The method has two variants:

- *Sequential forward selection (SFS)*, in which features are sequentially added to an empty candidate set until the addition of further features does not decrease the criterion.
- *Sequential backward selection (SBS)*, in which features are sequentially removed from a full candidate set until the removal of further features increase the criterion.

Stepwise regression is a sequential feature selection technique designed specifically for least-squares fitting. The functions `stepwise` and `stepwisefit` make use of optimizations that are only possible with least-squares criteria. Unlike generalized sequential feature selection, stepwise regression may remove features that have been added or add features that have been removed.

The Statistics Toolbox function `sequentialfs` carries out sequential feature selection. Input arguments include predictor and response data and a function handle to a file implementing the criterion function. Optional inputs allow you to specify SFS or SBS, required or excluded features, and the size of the feature subset. The function calls `cvpartition` and `crossval` to evaluate the criterion at different candidate sets.

### **Example: Sequential Feature Selection**

For example, consider a data set with 100 observations of 10 predictors. The following generates random data from a logistic model, with a binomial distribution of responses at each set of values for the predictors. Some coefficients are set to zero so that not all of the predictors affect the response:

```
n = 100;  
m = 10;  
X = rand(n,m);  
b = [1 0 0 2 .5 0 0 0.1 0 1];  
Xb = X*b';  
p = 1./(1+exp(-Xb));
```

```
N = 50;  
y = binornd(N,p);
```

The `glmfit` function fits a logistic model to the data:

```
Y = [y N*ones(size(y))];  
[b0,dev0,stats0] = glmfit(X,Y,'binomial');
```

% Display coefficient estimates and their standard errors:

```
model0 = [b0 stats0.se]  
model0 =
```

0.3115	0.2596
0.9614	0.1656
-0.1100	0.1651
-0.2165	0.1683
1.9519	0.1809
0.5683	0.2018
-0.0062	0.1740
0.0651	0.1641
-0.1034	0.1685
0.0017	0.1815
0.7979	0.1806

% Display the deviance of the fit:

```
dev0  
dev0 =  
101.2594
```

This is the full model, using all of the features (and an initial constant term). Sequential feature selection searches for a subset of the features in the full model with comparative predictive power.

First, you must specify a criterion for selecting the features. The following function, which calls `glmfit` and returns the deviance of the fit (a generalization of the residual sum of squares) is a useful criterion in this case:

```
function dev = critfun(X,Y)  
  
[b,dev] = glmfit(X,Y,'binomial');
```

You should create this function as a file on the MATLAB path.

The function `sequentialfs` performs feature selection, calling the criterion function via a function handle:

```
maxdev = chi2inv(.95,1);
opt = statset('display','iter',...
             'TolFun',maxdev,...
             'TolTypeFun','abs');

inmodel = sequentialfs(@critfun,X,Y,...
                       'cv','none',...
                       'nullmodel',true,...
                       'options',opt,...
                       'direction','forward');
```

Start forward sequential feature selection:

```
Initial columns included: none
Columns that can not be included: none
Step 1, used initial columns, criterion value 309.118
Step 2, added column 4, criterion value 180.732
Step 3, added column 1, criterion value 138.862
Step 4, added column 10, criterion value 114.238
Step 5, added column 5, criterion value 103.503
Final columns included: 1 4 5 10
```

The iterative display shows a decrease in the criterion value as each new feature is added to the model. The final result is a reduced model with only four of the original ten features: columns 1, 4, 5, and 10 of  $X$ . These features are indicated in the logical vector `inmodel` returned by `sequentialfs`.

The deviance of the reduced model is higher than for the full model, but the addition of any other single feature would not decrease the criterion by more than the absolute tolerance, `maxdev`, set in the options structure. Adding a feature with no effect reduces the deviance by an amount that has a chi-square distribution with one degree of freedom. Adding a significant feature results in a larger change. By setting `maxdev` to `chi2inv(.95,1)`, you instruct `sequentialfs` to continue adding features so long as the change in deviance is more than would be expected by random chance.

The reduced model (also with an initial constant term) is:

```
[b,dev,stats] = glmfit(X(:,inmodel),Y,'binomial');
```

```
% Display coefficient estimates and their standard errors:  
model = [b stats.se]  
model =  
    0.0784    0.1642  
    1.0040    0.1592  
    1.9459    0.1789  
    0.6134    0.1872  
    0.8245    0.1730
```

## Feature Transformation

### In this section...

“Introduction to Feature Transformation” on page 13-76

“Nonnegative Matrix Factorization” on page 13-76

“Principal Component Analysis (PCA)” on page 13-80

“Quality of Life in U.S. Cities” on page 13-81

“Factor Analysis” on page 13-94

### Introduction to Feature Transformation

*Feature transformation* is a group of methods that create new features (predictor variables). The methods are useful for dimension reduction when the transformed features have a descriptive power that is more easily ordered than the original features. In this case, less descriptive features can be dropped from consideration when building models.

Feature transformation methods are contrasted with the methods presented in “Feature Selection” on page 13-71, where dimension reduction is achieved by computing an optimal subset of predictive features measured in the original data.

The methods presented in this section share some common methodology. Their goals, however, are essentially different:

- Nonnegative matrix factorization is used when model terms must represent nonnegative quantities, such as physical quantities.
- Principal component analysis is used to summarize data in fewer dimensions, for example, to visualize it.
- Factor analysis is used to build explanatory models of data correlations.

### Nonnegative Matrix Factorization

- “Introduction to Nonnegative Matrix Factorization” on page 13-77
- “Example: Nonnegative Matrix Factorization” on page 13-77



## Introduction to Nonnegative Matrix Factorization

*Nonnegative matrix factorization (NMF)* is a dimension-reduction technique based on a low-rank approximation of the feature space. Besides providing a reduction in the number of features, NMF guarantees that the features are nonnegative, producing additive models that respect, for example, the nonnegativity of physical quantities.

Given a nonnegative  $m$ -by- $n$  matrix  $X$  and a positive integer  $k < \min(m, n)$ , NMF finds nonnegative  $m$ -by- $k$  and  $k$ -by- $n$  matrices  $W$  and  $H$ , respectively, that minimize the norm of the difference  $X - WH$ .  $W$  and  $H$  are thus approximate nonnegative factors of  $X$ .

The  $k$  columns of  $W$  represent transformations of the variables in  $X$ ; the  $k$  rows of  $H$  represent the coefficients of the linear combinations of the original  $n$  variables in  $X$  that produce the transformed variables in  $W$ . Since  $k$  is generally smaller than the rank of  $X$ , the product  $WH$  provides a compressed approximation of the data in  $X$ . A range of possible values for  $k$  is often suggested by the modeling context.

The Statistics Toolbox function `nnmf` carries out nonnegative matrix factorization. `nnmf` uses one of two iterative algorithms that begin with random initial values for  $W$  and  $H$ . Because the norm of the residual  $X - WH$  may have local minima, repeated calls to `nnmf` may yield different factorizations. Sometimes the algorithm converges to a solution of lower rank than  $k$ , which may indicate that the result is not optimal.

### Example: Nonnegative Matrix Factorization

For example, consider the five predictors of biochemical oxygen demand in the data set `moore.mat`.

```
load moore
X = moore(:,1:5);
rng('default'); % For reproducibility
```

The following uses `nnmf` to compute a rank-two approximation of  $X$  with a multiplicative update algorithm that begins from five random initial values for  $W$  and  $H$ .

```
opt = statset('MaxIter',10,'Display','final');
```

```
[W0,H0] = nnmf(X,2,'replicates',5,'options',opt,'algorithm','mult');
```

```

      rep   iteration   rms resid   |delta x|
      1      10      358.296   0.00190554
      2      10      78.3556   0.000351747
      3      10      230.962   0.0172839
      4      10      326.347   0.00739552
      5      10      361.547   0.00705539

```

```
Final root mean square residual = 78.3556
```

The 'mult' algorithm is sensitive to initial values, which makes it a good choice when using 'replicates' to find W and H from multiple random starting values.

Now perform the factorization using an alternating least-squares algorithm, which converges faster and more consistently. Run 100 times more iterations, beginning from the initial W0 and H0 identified above.

```
opt = statset('Maxiter',1000,'Display','final');
[W,H] = nnmf(X,2,'w0',W0,'h0',H0,'options',opt,'algorithm','als');
```

```

      rep   iteration   rms resid   |delta x|
      1      2      77.5315   0.000830334

```

```
Final root mean square residual = 77.5315
```

The two columns of W are the transformed predictors. The two rows of H give the relative contributions of each of the five predictors in X to the predictors in W.

H

H =

```

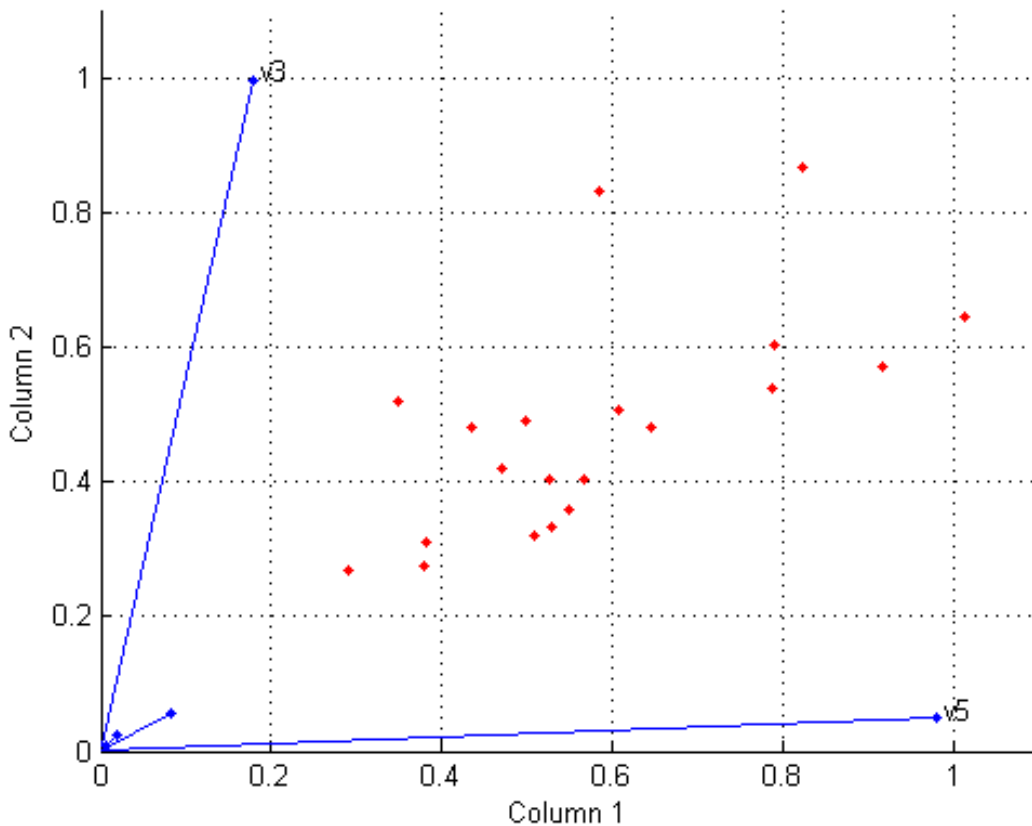
      0.0835   0.0190   0.1782   0.0072   0.9802
      0.0559   0.0250   0.9969   0.0085   0.0497

```

The fifth predictor in  $X$  (weight 0.9802) strongly influences the first predictor in  $W$ . The third predictor in  $X$  (weight 0.9969) strongly influences the second predictor in  $W$ .

Visualize the relative contributions of the predictors in  $X$  with a biplot, showing the data and original variables in the column space of  $W$ .

```
biplot(H', 'scores', W, 'varlabels', {'', '', 'v3', '', 'v5'});  
axis([0 1.1 0 1.1])  
xlabel('Column 1')  
ylabel('Column 2')
```



## Principal Component Analysis (PCA)

One of the difficulties inherent in multivariate statistics is the problem of visualizing data that has many variables. The MATLAB function `plot` displays a graph of the relationship between two variables. The `plot3` and `surf` commands display different three-dimensional views. But when there are more than three variables, it is more difficult to visualize their relationships.

Fortunately, in data sets with many variables, groups of variables often move together. One reason for this is that more than one variable might be measuring the same driving principle governing the behavior of the system. In many systems there are only a few such driving forces. But an abundance of instrumentation enables you to measure dozens of system variables. When this happens, you can take advantage of this redundancy of information. You can simplify the problem by replacing a group of variables with a single new variable.

Principal component analysis is a quantitatively rigorous method for achieving this simplification. The method generates a new set of variables, called *principal components*. Each principal component is a linear combination of the original variables. All the principal components are orthogonal to each other, so there is no redundant information. The principal components as a whole form an orthogonal basis for the space of the data.

There are an infinite number of ways to construct an orthogonal basis for several columns of data. What is so special about the principal component basis?

The first principal component is a single axis in space. When you project each observation on that axis, the resulting values form a new variable. And the variance of this variable is the maximum among all possible choices of the first axis.

The second principal component is another axis in space, perpendicular to the first. Projecting the observations on this axis generates another new variable. The variance of this variable is the maximum among all possible choices of this second axis.

The full set of principal components is as large as the original set of variables. But it is commonplace for the sum of the variances of the first few principal

components to exceed 80% of the total variance of the original data. By examining plots of these few new variables, researchers often develop a deeper understanding of the driving forces that generated the original data.

You can use the function `pca` to find the principal components. To use `pca`, you need to have the actual measured data you want to analyze. However, if you lack the actual data, but have the sample covariance or correlation matrix for the data, you can still use the function `pcacov` to perform a principal components analysis. See the reference page for `pcacov` for a description of its inputs and outputs.

## Quality of Life in U.S. Cities

This example shows how to perform a weighted principal components analysis and interpret the results.

### Load sample data.

Load the sample data. The data includes ratings for 9 different indicators of the quality of life in 329 U.S. cities. These are climate, housing, health, crime, transportation, education, arts, recreation, and economics. For each category, a higher rating is better. For example, a higher rating for crime means a lower crime rate.

Display the `categories` variable.

```
load cities
categories

categories =
  climate
  housing
  health
  crime
  transportation
  education
  arts
  recreation
  economics
```

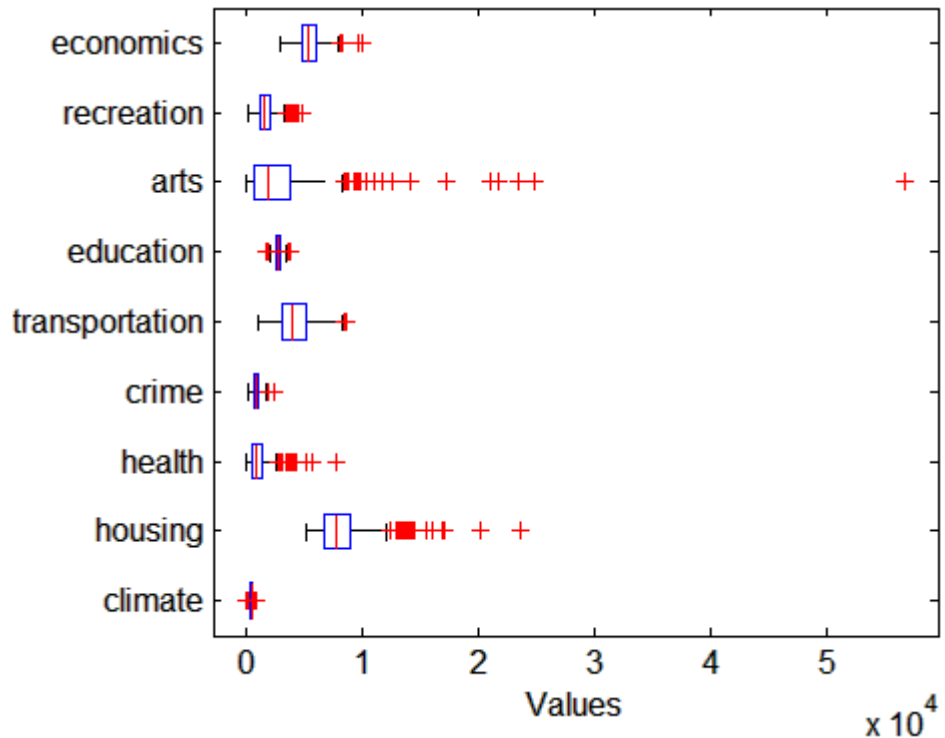
In total, the `cities` data set contains three variables:

- categories, a string matrix containing the names of the indices
- names, a string matrix containing the 329 city names
- ratings, the data matrix with 329 rows and 9 columns

**Plot data.**

Make a boxplot to look at the distribution of the ratings data.

```
figure()  
boxplot(ratings, 'orientation', 'horizontal', 'labels', categories)
```



There is more variability in the ratings of the arts and housing than in the ratings of crime and climate.

**Check pairwise correlation.**

Check the pairwise correlation between the variables.

```
C = corr(ratings,ratings);
```

The correlation among some variables is as high as 0.85. Principal components analysis constructs independent new variables which are linear combinations of the original variables.

**Compute principal components.**

When all variables are in the same unit, it is appropriate to compute principal components for raw data. When the variables are in different units or the difference in the variance of different columns is substantial (as in this case), scaling of the data or use of weights is often preferable.

Perform the principal component analysis by using the inverse variances of the ratings as weights.

```
w = 1./var(ratings);
[wcoeff,score,latent,tsquared,explained] = pca(ratings,...
'VariableWeights',w);
```

Or equivalently:

```
[wcoeff,score,latent,tsquared,explained] = pca(ratings,...
'VariableWeights','variance');
```

The following sections explain the five outputs of `pca`.

**Component coefficients.**

The first output, `wcoeff`, contains the coefficients of the principal components.

The first three principal component coefficient vectors are:

```
c3 = wcoeff(:,1:3)
```

```
c3 = wcoeff(:,1:3)
c3 =
```

```
1.0e+03 *  
0.0249 -0.0263 -0.0834  
0.8504 -0.5978 -0.4965  
0.4616 0.3004 -0.0073  
0.1005 -0.1269 0.0661  
0.5096 0.2606 0.2124  
0.0883 0.1551 0.0737  
2.1496 0.9043 -0.1229  
0.2649 -0.3106 -0.0411  
0.1469 -0.5111 0.6586
```

These coefficients are weighted, hence the coefficient matrix is not orthonormal.

#### **Transform coefficients.**

Transform the coefficients so that they are orthonormal.

```
coefforth = inv(diag(std(ratings)))*wcoeff;
```

Note that if you use a weights vector,  $w$ , while conducting the pca, then

```
coefforth = diag(sqrt(w))*wcoeff;
```

#### **Check coefficients are orthonormal.**

The transformed coefficients are now orthonormal.

```
I = c3'*c3
```

```
I =  
1.0000 -0.0000 -0.0000  
-0.0000 1.0000 -0.0000  
-0.0000 -0.0000 1.0000
```

#### **Component scores.**

The second output, `score`, contains the coordinates of the original data in the new coordinate system defined by the principal components. The `score`



matrix is the same size as the input data matrix. You can also obtain the component scores using the orthonormal coefficients and the standardized ratings as follows.

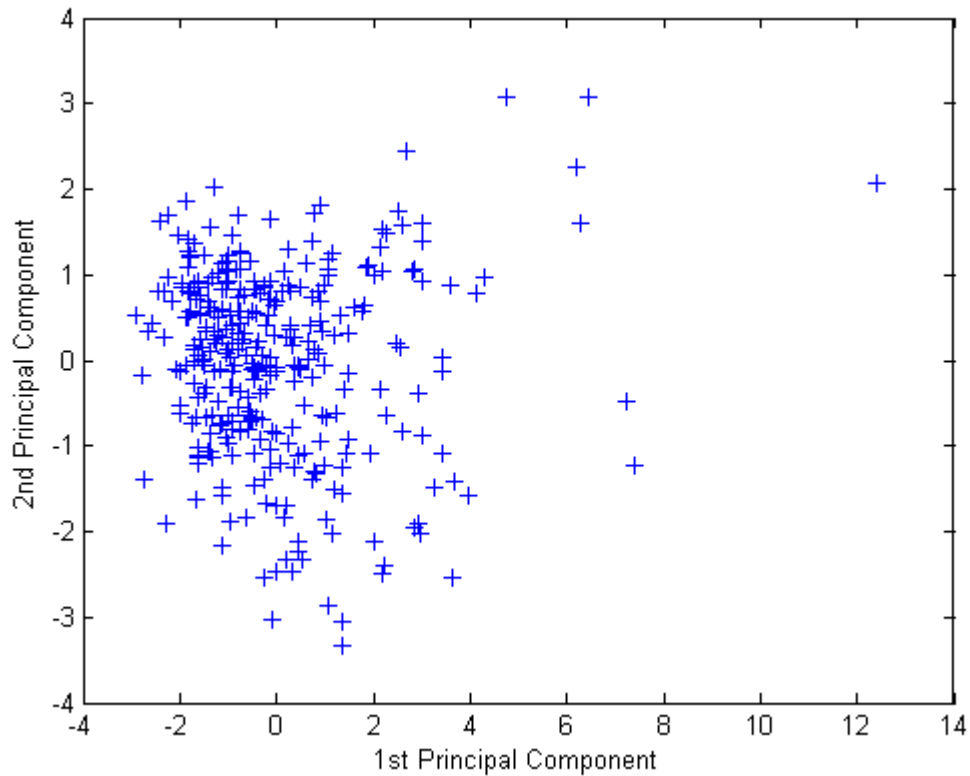
```
cscores = zscore(ratings)*coefforth;
```

cscores and score are identical matrices.

**Plot component scores.**

Create a plot of the first two columns of score.

```
figure()  
plot(score(:,1),score(:,2),'+')  
xlabel('1st Principal Component')  
ylabel('2nd Principal Component')
```



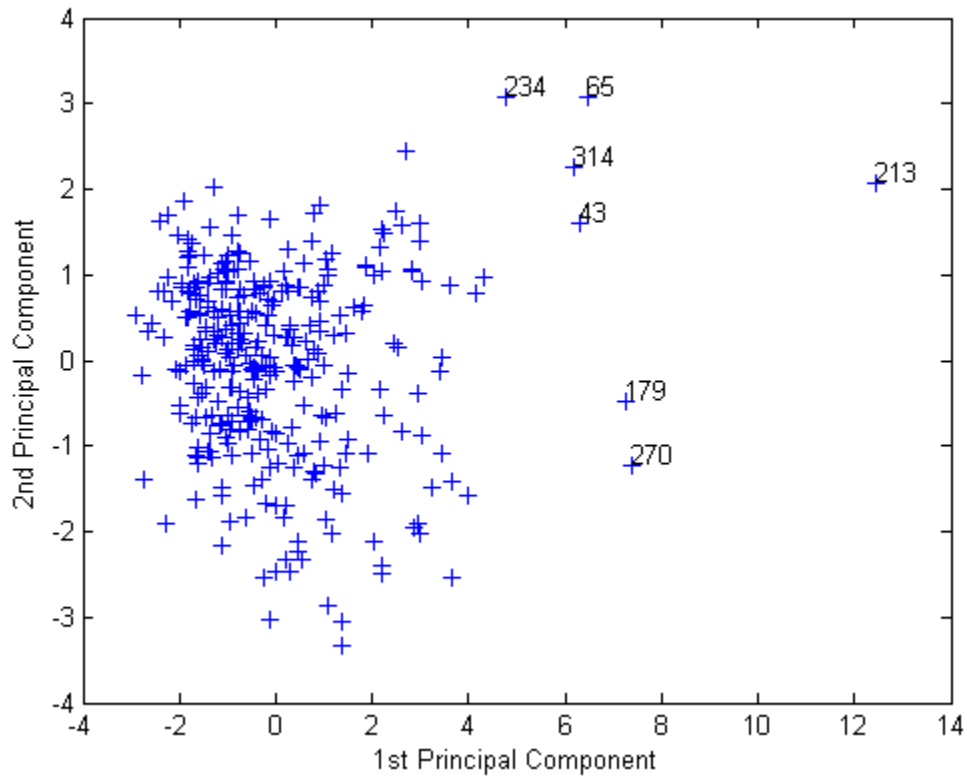
This plot shows the centered and scaled ratings data projected onto the first two principal components. `pca` computes the scores to have mean zero.

**Explore plot interactively.**

Note the outlying points in the right half of the plot. You can graphically identify these points as follows.

```
gname
```

Move your cursor over the plot and click once near the rightmost seven points. This labels the points by their row numbers as in the following figure.



After labeling points, press **Return**.

**Extract observation names.**

Create an index variable containing the row numbers of all the cities you chose and get the names of the cities.

```
metro = [43 65 179 213 234 270 314];
names(metro,:)
```

```
ans =
    Boston, MA
    Chicago, IL
```

```
Los Angeles, Long Beach, CA
New York, NY
Philadelphia, PA-NJ
San Francisco, CA
Washington, DC-MD-VA
```

These labeled cities are some of the biggest population centers in the United States and they appear more extreme than the remainder of the data.

**Component variances.**

The third output, `latent`, is a vector containing the variance explained by the corresponding principal component. Each column of `score` has a sample variance equal to the corresponding row of `latent`.

```
latent
```

```
latent =
```

```
3.4083
1.2140
1.1415
0.9209
0.7533
0.6306
0.4930
0.3180
0.1204
```

**Percent variance explained.**

The fifth output, `explained`, is a vector containing the percent variance explained by the corresponding principal component.

```
explained
```

```
explained =
```

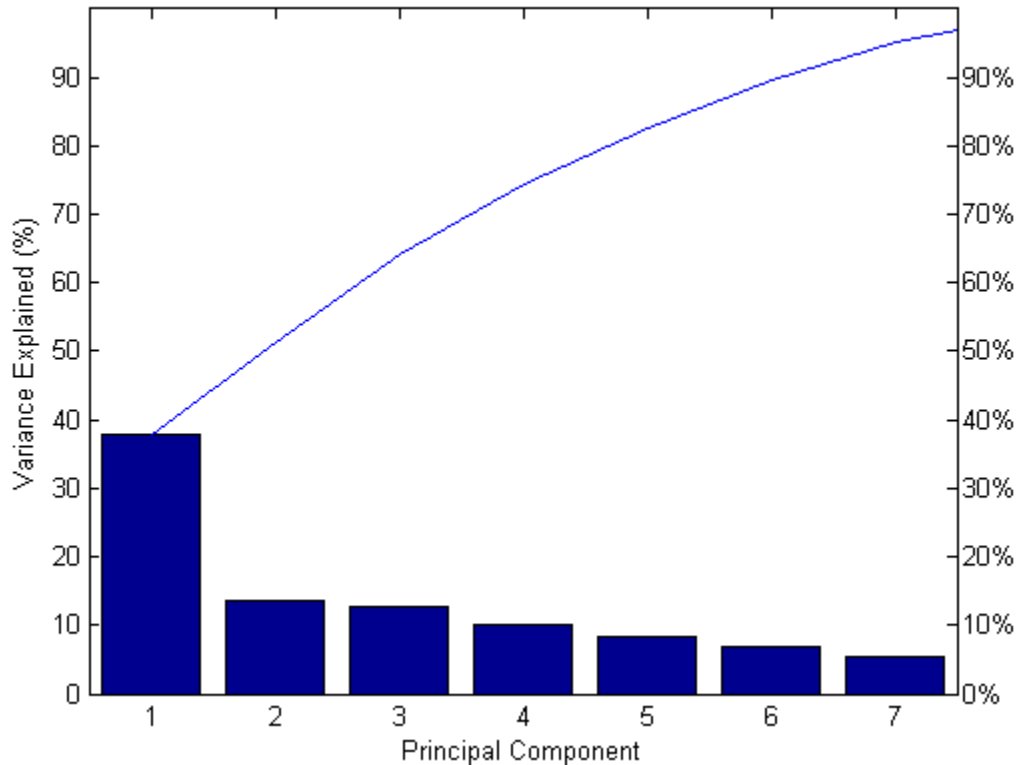
```
37.8699
13.4886
12.6831
```

```
10.2324
 8.3698
 7.0062
 5.4783
 3.5338
 1.3378
```

**Create scree plot.**

Make a scree plot of the percent variability explained by each principal component.

```
figure()
pareto(explained)
xlabel('Principal Component')
ylabel('Variance Explained (%)')
```



This scree plot only shows the first seven (instead of the total nine) components that explain 95% of the total variance. The only clear break in the amount of variance accounted for by each component is between the first and second components. However, the first component by itself explains less than 40% of the variance, so more components might be needed. You can see that the first three principal components explain roughly two-thirds of the total variability in the standardized ratings, so that might be a reasonable way to reduce the dimensions.

**Hotelling's T-squared statistic.**

The last output from `pca` is `tsquared`, which is Hotelling's  $T^2$ , a statistical measure of the multivariate distance of each observation from the center of the data set. This is an analytical way to find the most extreme points in the data.

```
[st2,index] = sort(tsquared,'descend'); % sort in descending order
extreme = index(1);
names(extreme,:)
```

```
ans =
```

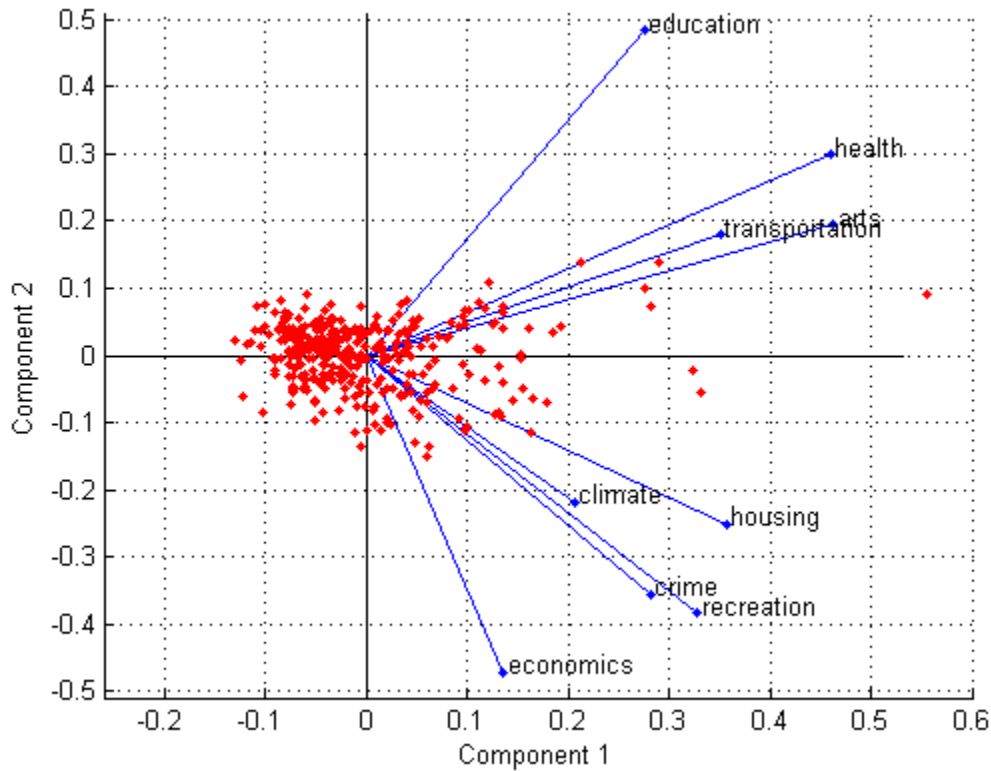
```
New York, NY
```

The ratings for New York are the furthest from the average U.S. city.

### **Visualize the results.**

Visualize both the orthonormal principal component coefficients for each variable and the principal component scores for each observation in a single plot.

```
biplot(coefforth(:,1:2), 'scores', score(:,1:2), 'varlabels', categories);
axis([-0.26 0.6 -0.51 0.51]);
```



All nine variables are represented in this bi-plot by a vector, and the direction and length of the vector indicate how each variable contributes to the two principal components in the plot. For example, the first principal component, on the horizontal axis, has positive coefficients for all nine variables. That is why the nine vectors are directed into the right half of the plot. The largest coefficients in the first principal component are the third and seventh elements, corresponding to the variables health and arts.

The second principal component, on the vertical axis, has positive coefficients for the variables education, health, arts, and transportation, and negative coefficients for the remaining five variables. This indicates that the



second component distinguishes among cities that have high values for the first set of variables and low for the second, and cities that have the opposite.

The variable labels in this figure are somewhat crowded. You can either exclude the `VarLabels` parameter when making the plot, or select and drag some of the labels to better positions using the `Edit Plot` tool from the figure window toolbar.

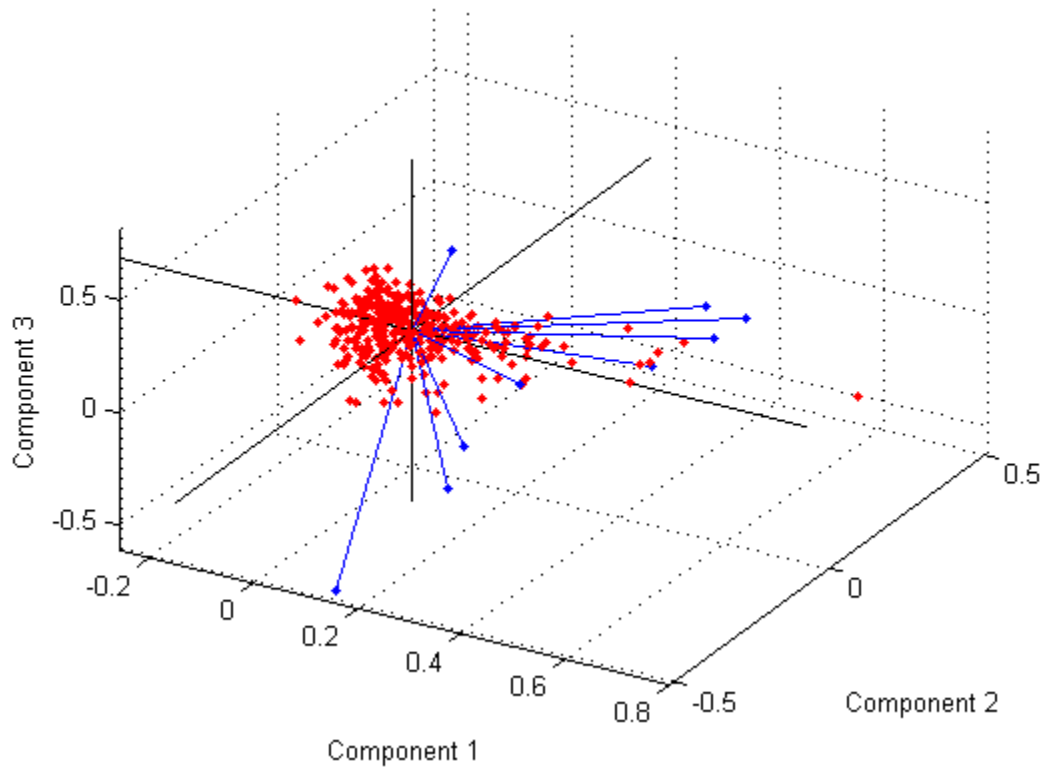
This 2-D bi-plot also includes a point for each of the 329 observations, with coordinates indicating the score of each observation for the two principal components in the plot. For example, points near the left edge of this plot have the lowest scores for the first principal component. The points are scaled with respect to the maximum score value and maximum coefficient length, so only their relative locations can be determined from the plot.

You can identify items in the plot by selecting **Tools>Data Cursor** from the figure window. By clicking a variable (vector), you can read that variable's coefficients for each principal component. By clicking an observation (point), you can read that observation's scores for each principal component.

### **Create a three-dimensional bi-plot.**

You can also make a bi-plot in three dimensions.

```
figure()  
biplot(coefforth(:,1:3), 'scores', score(:,1:3), 'obslabels', names);  
axis([-0.26 0.8 -0.51 0.51 -0.61 0.81]);  
view([30 40]);
```



This graph is useful if the first two principal coordinates do not explain enough of the variance in your data. You can also rotate the figure to see it from different angles by selecting the **Tools**> Rotate 3D.

## Factor Analysis

- “Introduction to Factor Analysis” on page 13-95
- “Example: Factor Analysis” on page 13-95

## Introduction to Factor Analysis

Multivariate data often includes a large number of measured variables, and sometimes those variables overlap, in the sense that groups of them might be dependent. For example, in a decathlon, each athlete competes in 10 events, but several of them can be thought of as speed events, while others can be thought of as strength events, etc. Thus, you can think of a competitor's 10 event scores as largely dependent on a smaller set of three or four types of athletic ability.

Factor analysis is a way to fit a model to multivariate data to estimate just this sort of interdependence. In a factor analysis model, the measured variables depend on a smaller number of unobserved (latent) factors. Because each factor might affect several variables in common, they are known as *common factors*. Each variable is assumed to be dependent on a linear combination of the common factors, and the coefficients are known as *loadings*. Each measured variable also includes a component due to independent random variability, known as *specific variance* because it is specific to one variable.

Specifically, factor analysis assumes that the covariance matrix of your data is of the form

$$\Sigma_x = \Lambda\Lambda^T + \Psi$$

where  $\Lambda$  is the matrix of loadings, and the elements of the diagonal matrix  $\Psi$  are the specific variances. The function `factoran` fits the Factor Analysis model using maximum likelihood.

### Example: Factor Analysis

- “Factor Loadings” on page 13-96
- “Factor Rotation” on page 13-98
- “Factor Scores” on page 13-101
- “Visualize the Results” on page 13-103

**Factor Loadings.** Over the course of 100 weeks, the percent change in stock prices for ten companies has been recorded. Of the ten companies, the first four can be classified as primarily technology, the next three as financial, and the last three as retail. It seems reasonable that the stock prices for companies that are in the same sector might vary together as economic conditions change. Factor Analysis can provide quantitative evidence that companies within each sector do experience similar week-to-week changes in stock price.

In this example, you first load the data, and then call `factoran`, specifying a model fit with three common factors. By default, `factoran` computes rotated estimates of the loadings to try and make their interpretation simpler. But in this example, you specify an unrotated solution.

```
load stockreturns
```

```
[Loadings,specificVar,T,stats] = factoran(stocks,3,'rotate','none');
```

The first two `factoran` return arguments are the estimated loadings and the estimated specific variances. Each row of the loadings matrix represents one of the ten stocks, and each column corresponds to a common factor. With unrotated estimates, interpretation of the factors in this fit is difficult because most of the stocks contain fairly large coefficients for two or more factors.

Loadings

```
Loadings =
```

0.8885	0.2367	-0.2354
0.7126	0.3862	0.0034
0.3351	0.2784	-0.0211
0.3088	0.1113	-0.1905
0.6277	-0.6643	0.1478
0.4726	-0.6383	0.0133
0.1133	-0.5416	0.0322
0.6403	0.1669	0.4960
0.2363	0.5293	0.5770
0.1105	0.1680	0.5524

---

**Note** “Factor Rotation” on page 13-98 helps to simplify the structure in the Loadings matrix, to make it easier to assign meaningful interpretations to the factors.

---

From the estimated specific variances, you can see that the model indicates that a particular stock price varies quite a lot beyond the variation due to the common factors.

```
specificVar
```

```
specificVar =
```

```
0.0991
0.3431
0.8097
0.8559
0.1429
0.3691
0.6928
0.3162
0.3311
0.6544
```

A specific variance of 1 would indicate that there is *no* common factor component in that variable, while a specific variance of 0 would indicate that the variable is *entirely* determined by common factors. These data seem to fall somewhere in between.

The  $p$  value returned in the `stats` structure fails to reject the null hypothesis of three common factors, suggesting that this model provides a satisfactory explanation of the covariation in these data.

```
stats.p
```

```
ans =
```

```
0.8144
```

To determine whether fewer than three factors can provide an acceptable fit, you can try a model with two common factors. The  $p$  value for this second fit is highly significant, and rejects the hypothesis of two factors, indicating that the simpler model is not sufficient to explain the pattern in these data.

```
[Loadings2,specificVar2,T2,stats2] = factoran(stocks, 2, 'rotate', 'none');
```

```
stats2.p
```

```
ans =
```

```
3.5610e-06
```

**Factor Rotation.** As the results illustrate, the estimated loadings from an unrotated factor analysis fit can have a complicated structure. The goal of factor rotation is to find a parameterization in which each variable has only a small number of large loadings. That is, each variable is affected by a small number of factors, preferably only one. This can often make it easier to interpret what the factors represent.

If you think of each row of the loadings matrix as coordinates of a point in  $M$ -dimensional space, then each factor corresponds to a coordinate axis. Factor rotation is equivalent to rotating those axes and computing new loadings in the rotated coordinate system. There are various ways to do this. Some methods leave the axes orthogonal, while others are oblique methods that change the angles between them. For this example, you can rotate the estimated loadings by using the promax criterion, a common oblique method.

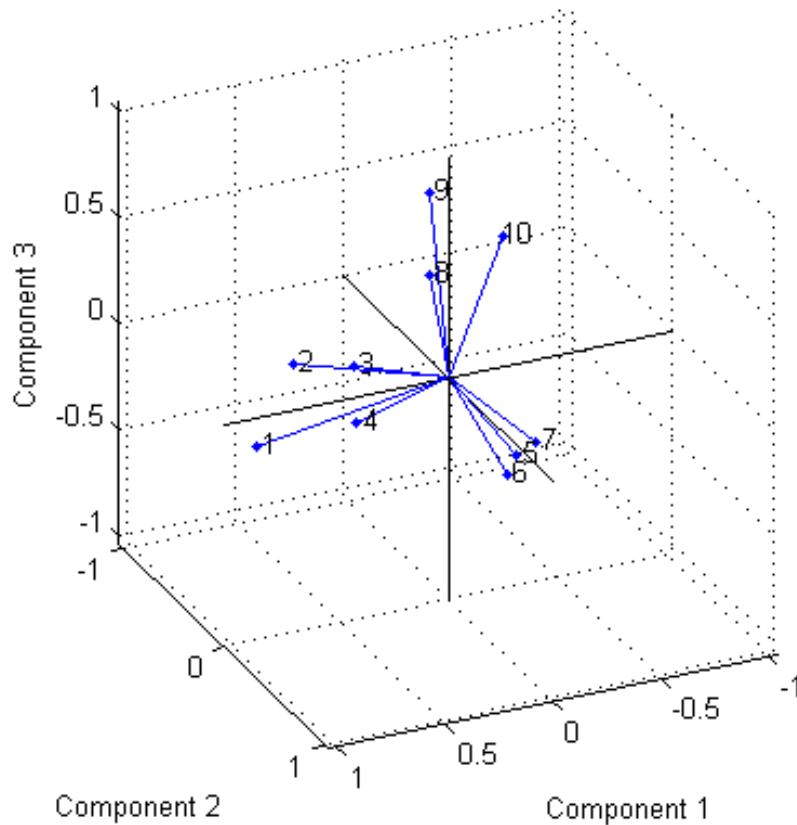
```
[LoadingsPM,specVarPM] = factoran(stocks,3,'rotate','promax');  
LoadingsPM
```

```
LoadingsPM =
```

0.9452	0.1214	-0.0617
0.7064	-0.0178	0.2058
0.3885	-0.0994	0.0975
0.4162	-0.0148	-0.1298
0.1021	0.9019	0.0768
0.0873	0.7709	-0.0821
-0.1616	0.5320	-0.0888
0.2169	0.2844	0.6635
0.0016	-0.1881	0.7849
-0.2289	0.0636	0.6475

Promax rotation creates a simpler structure in the loadings, one in which most of the stocks have a large loading on only one factor. To see this structure more clearly, you can use the `biplot` function to plot each stock using its factor loadings as coordinates.

```
biplot(LoadingsPM, 'varlabels', num2str((1:10)'));  
axis square  
view(155,27);
```



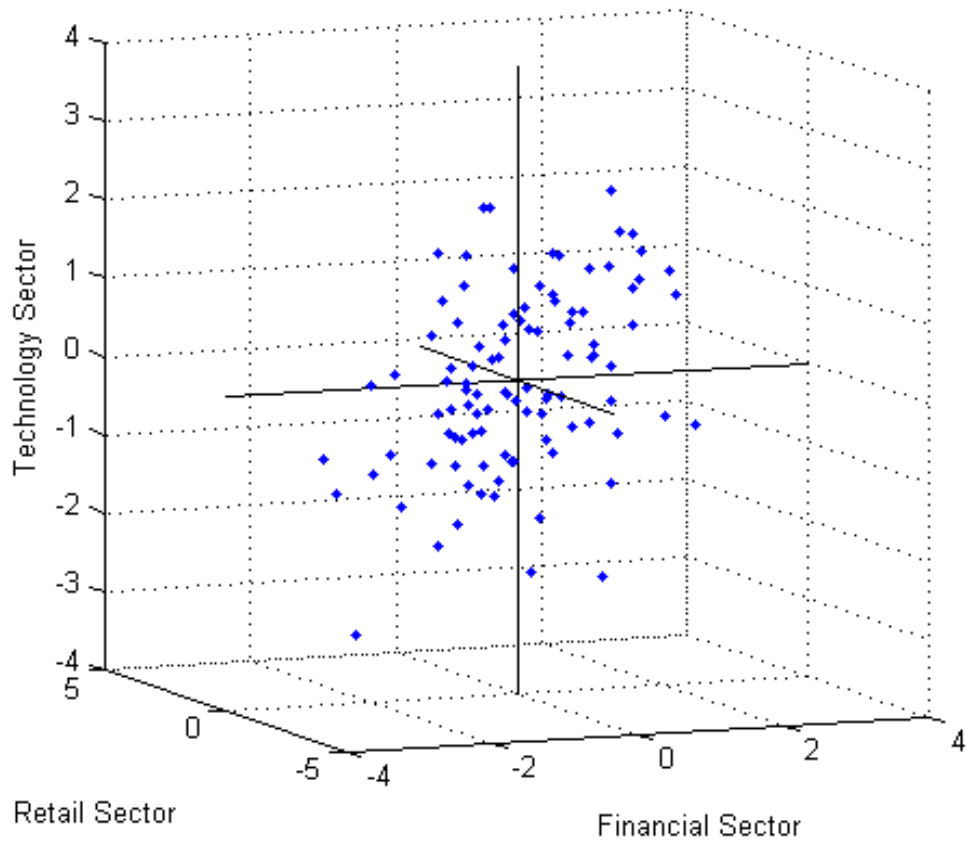
This plot shows that promax has rotated the factor loadings to a simpler structure. Each stock depends primarily on only one factor, and it is possible to describe each factor in terms of the stocks that it affects. Based on which companies are near which axes, you could reasonably conclude that the first factor axis represents the financial sector, the second retail, and the third technology. The original conjecture, that stocks vary primarily within sector, is apparently supported by the data.



**Factor Scores.** Sometimes, it is useful to be able to classify an observation based on its factor scores. For example, if you accepted the three-factor model and the interpretation of the rotated factors, you might want to categorize each week in terms of how favorable it was for each of the three stock sectors, based on the data from the 10 observed stocks.

Because the data in this example are the raw stock price changes, and not just their correlation matrix, you can have `factoran` return estimates of the value of each of the three rotated common factors for each week. You can then plot the estimated scores to see how the different stock sectors were affected during each week.

```
[LoadingsPM,specVarPM,TPM,stats,F] = factoran(stocks, 3, 'rotate', 'promax');  
  
plot3(F(:,1),F(:,2),F(:,3), 'b. ')  
line([-4 4 NaN 0 0 NaN 0 0], [0 0 NaN -4 4 NaN 0 0],[0 0 NaN 0 0 NaN -4 4],  
xlabel('Financial Sector')  
ylabel('Retail Sector')  
zlabel('Technology Sector')  
grid on  
axis square  
view(-22.5, 8)
```



Oblique rotation often creates factors that are correlated. This plot shows some evidence of correlation between the first and third factors, and you can investigate further by computing the estimated factor correlation matrix.

```
inv(TPM' * TPM)
```

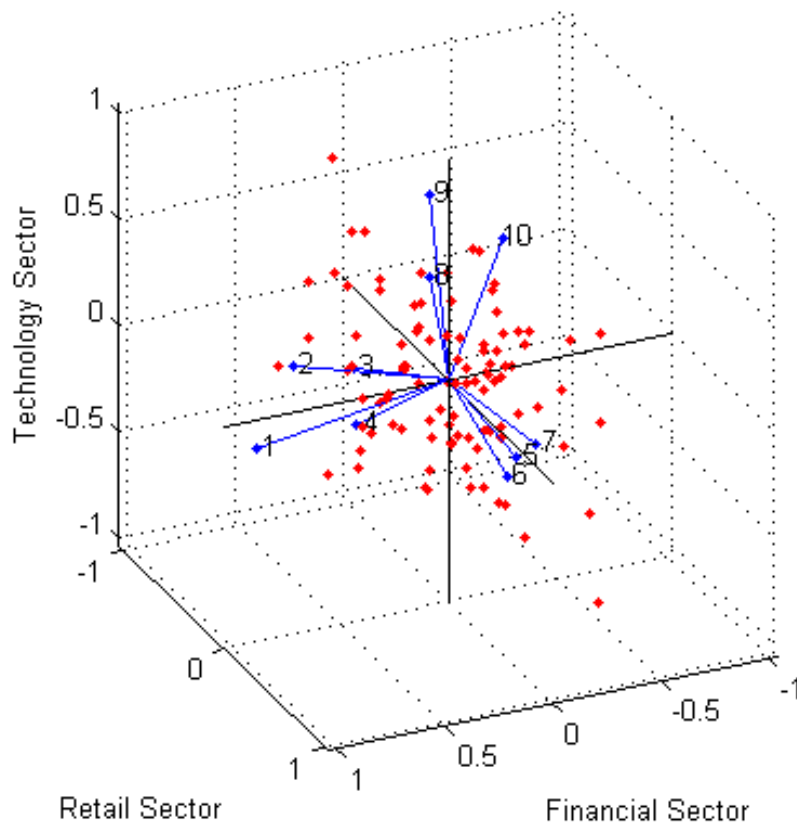
```
ans =
```

```
1.0000    0.1559    0.4082
```

```
0.1559    1.0000   -0.0559
0.4082   -0.0559    1.0000
```

**Visualize the Results.** You can use the `biplot` function to help visualize both the factor loadings for each variable and the factor scores for each observation in a single plot. For example, the following command plots the results from the factor analysis on the stock data and labels each of the 10 stocks.

```
biplot(LoadingsPM,'scores',F,'varlabels',num2str((1:10)))
xlabel('Financial Sector')
ylabel('Retail Sector')
zlabel('Technology Sector')
axis square
view(155,27)
```



In this case, the factor analysis includes three factors, and so the biplot is three-dimensional. Each of the 10 stocks is represented in this plot by a vector, and the direction and length of the vector indicates how each stock depends on the underlying factors. For example, you have seen that after promax rotation, the first four stocks have positive loadings on the first factor, and unimportant loadings on the other two factors. That first factor, interpreted as a financial sector effect, is represented in this biplot as one of the horizontal axes. The dependence of those four stocks on that factor corresponds to the four vectors directed approximately along that axis. Similarly, the dependence

of stocks 5, 6, and 7 primarily on the second factor, interpreted as a retail sector effect, is represented by vectors directed approximately along that axis.

Each of the 100 observations is represented in this plot by a point, and their locations indicate the score of each observation for the three factors. For example, points near the top of this plot have the highest scores for the technology sector factor. The points are scaled to fit within the unit square, so only their relative locations can be determined from the plot.

You can use the **Data Cursor** tool from the **Tools** menu in the figure window to identify the items in this plot. By clicking a stock (vector), you can read off that stock's loadings for each factor. By clicking an observation (point), you can read off that observation's scores for each factor.

## Partial Least Squares Regression and Principal Components Regression

This example shows how to apply Partial Least Squares Regression (PLSR) and Principal Components Regression (PCR), and discusses the effectiveness of the two methods. PLSR and PCR are both methods to model a response variable when there are a large number of predictor variables, and those predictors are highly correlated or even collinear. Both methods construct new predictor variables, known as components, as linear combinations of the original predictor variables, but they construct those components in different ways. PCR creates components to explain the observed variability in the predictor variables, without considering the response variable at all. On the other hand, PLSR does take the response variable into account, and therefore often leads to models that are able to fit the response variable with fewer components. Whether or not that ultimately translates into a more parsimonious model, in terms of its practical use, depends on the context.

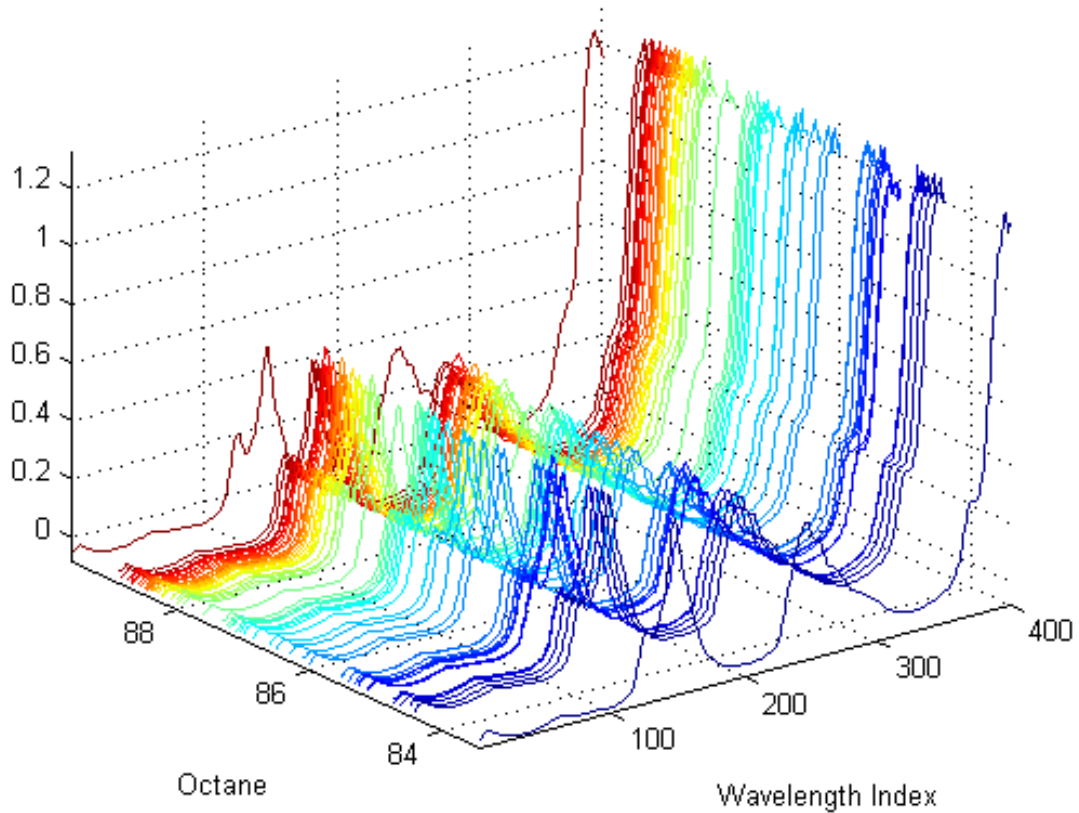
### Loading the Data

Load a data set comprising spectral intensities of 60 samples of gasoline at 401 wavelengths, and their octane ratings. These data are described in Kalivas, John H., "Two Data Sets of Near Infrared Spectra," *Chemometrics and Intelligent Laboratory Systems*, v.37 (1997) pp.255-259.

```
load spectra
whos NIR octane
```

Name	Size	Bytes	Class	Attributes
NIR	60x401	192480	double	
octane	60x1	480	double	

```
[dummy,h] = sort(octane);
oldorder = get(gcf,'DefaultAxesColorOrder');
set(gcf,'DefaultAxesColorOrder',jet(60));
plot3(repmat(1:401,60,1)',repmat(octane(h),1,401)',NIR(h,:));
set(gcf,'DefaultAxesColorOrder',oldorder);
xlabel('Wavelength Index'); ylabel('Octane'); axis('tight');
grid on
```



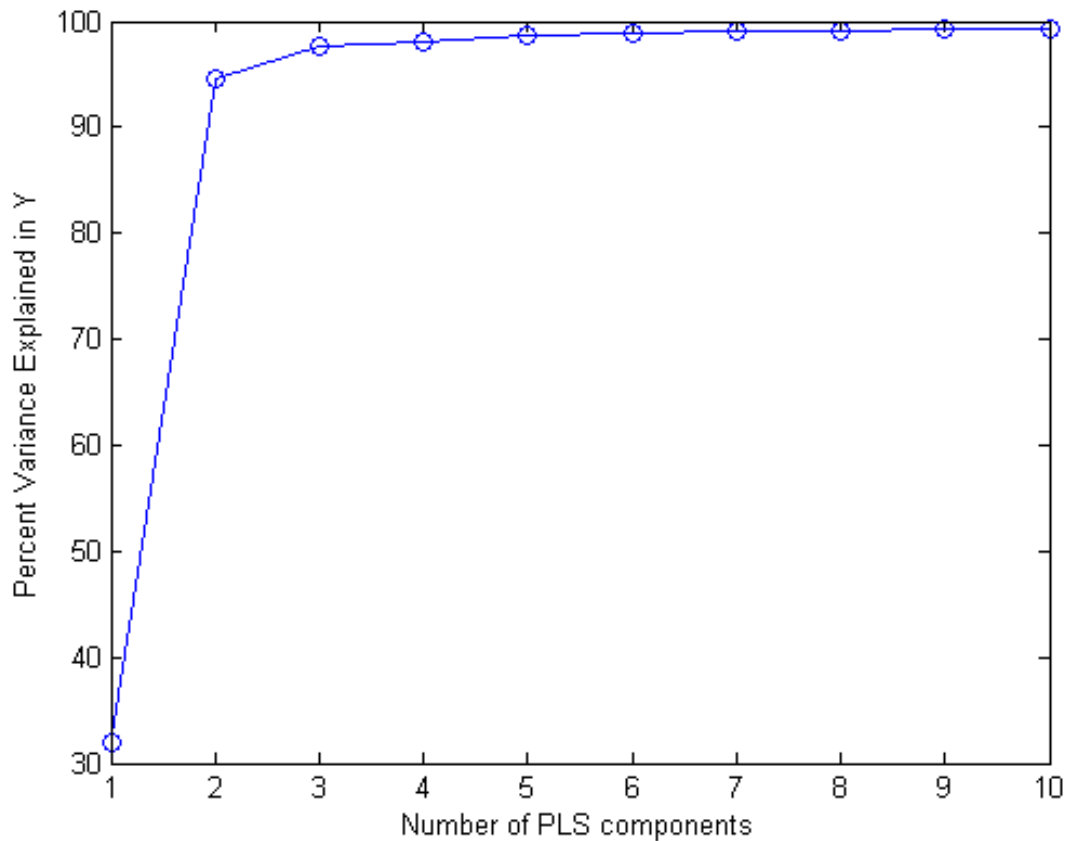
### Fitting the Data with Two Components

Use the `plsregress` function to fit a PLSR model with ten PLS components and one response.

```
X = NIR;  
y = octane;  
[n,p] = size(X);  
[Xloadings,Yloadings,Xscores,Yscores,betaPLS10,PLSPctVar] = plsregress(...  
X,y,10);
```

Ten components may be more than will be needed to adequately fit the data, but diagnostics from this fit can be used to make a choice of a simpler model with fewer components. For example, one quick way to choose the number of components is to plot the percent of variance explained in the response variable as a function of the number of components.

```
plot(1:10,cumsum(100*PLSPctVar(2,:)),'-bo');  
xlabel('Number of PLS components');  
ylabel('Percent Variance Explained in Y');
```





In practice, more care would probably be advisable in choosing the number of components. Cross-validation, for instance, is a widely-used method that will be illustrated later in this example. For now, the above plot suggests that PLSR with two components explains most of the variance in the observed  $y$ . Compute the fitted response values for the two-component model.

```
[Xloadings,Yloadings,Xscores,Yscores,betaPLS] = plsregress(X,y,2);
yfitPLS = [ones(n,1) X]*betaPLS;
```

Next, fit a PCR model with two principal components. The first step is to perform Principal Components Analysis on  $X$ , using the `pca` function, and retaining two principal components. PCR is then just a linear regression of the response variable on those two components. It often makes sense to normalize each variable first by its standard deviation when the variables have very different amounts of variability, however, that is not done here.

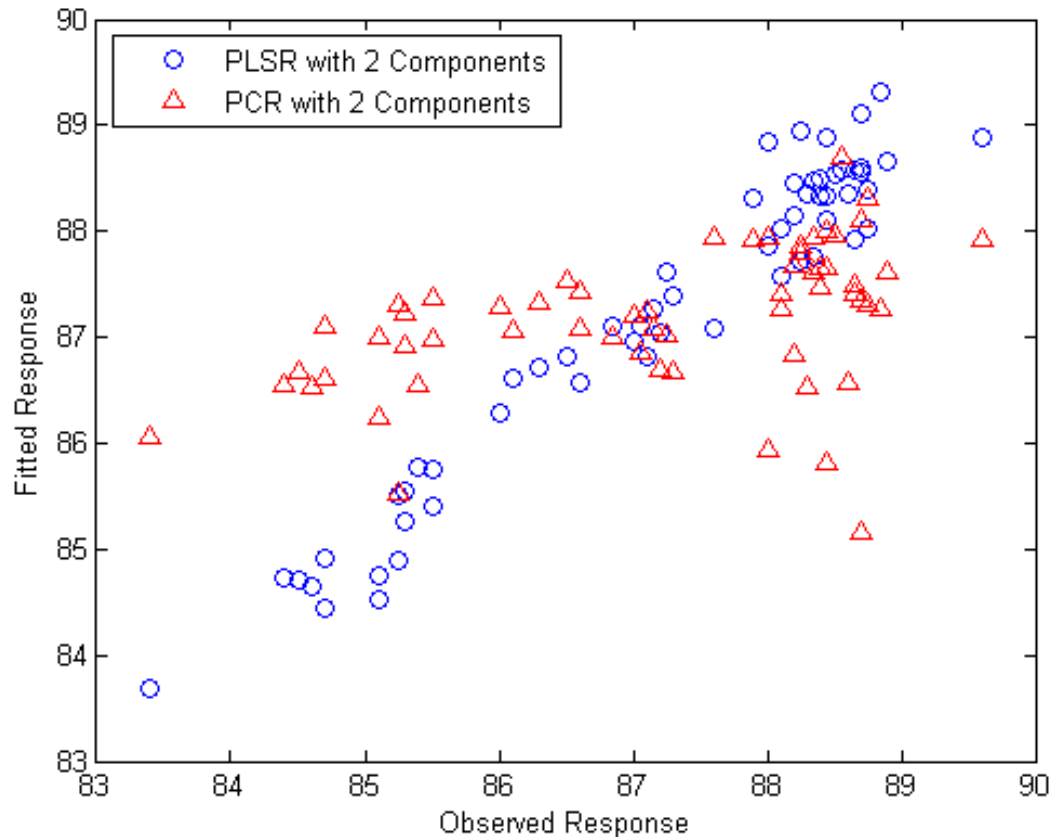
```
[PCALoadings,PCAScores,PCAVar] = pca(X,'Economy',false);
betaPCR = regress(y-mean(y), PCAScores(:,1:2));
```

To make the PCR results easier to interpret in terms of the original spectral data, transform to regression coefficients for the original, uncentered variables.

```
betaPCR = PCALoadings(:,1:2)*betaPCR;
betaPCR = [mean(y) - mean(X)*betaPCR; betaPCR];
yfitPCR = [ones(n,1) X]*betaPCR;
```

Plot fitted vs. observed response for the PLSR and PCR fits.

```
plot(y,yfitPLS,'bo',y,yfitPCR,'r^');
xlabel('Observed Response');
ylabel('Fitted Response');
legend({'PLSR with 2 Components' 'PCR with 2 Components'}, ...
      'location','NW');
```



In a sense, the comparison in the plot above is not a fair one -- the number of components (two) was chosen by looking at how well a two-component PLSR model predicted the response, and there's no reason why the PCR model should be restricted to that same number of components. With the same number of components, however, PLSR does a much better job at fitting  $y$ . In fact, looking at the horizontal scatter of fitted values in the plot above, PCR with two components is hardly better than using a constant model. The  $r$ -squared values from the two regressions confirm that.

$$\text{TSS} = \text{sum}((y - \text{mean}(y)).^2);$$

```
RSS_PLS = sum((y-yfitPLS).^2);  
rsquaredPLS = 1 - RSS_PLS/TSS
```

```
rsquaredPLS =
```

```
0.9466
```

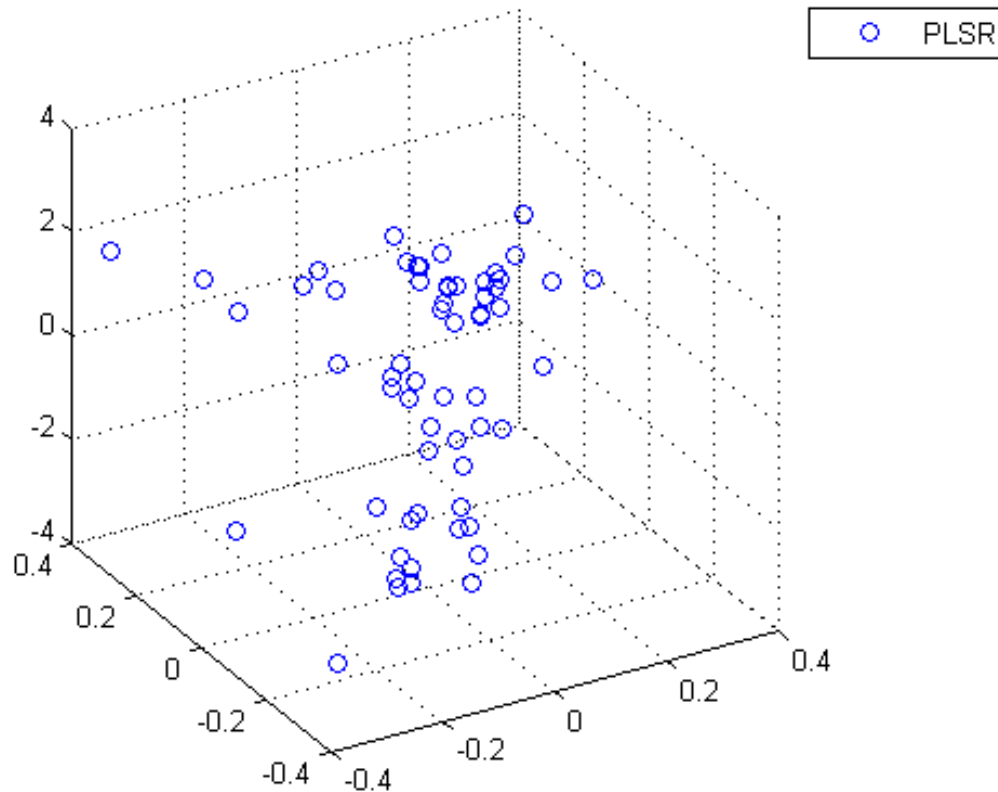
```
RSS_PCR = sum((y-yfitPCR).^2);  
rsquaredPCR = 1 - RSS_PCR/TSS
```

```
rsquaredPCR =
```

```
0.1962
```

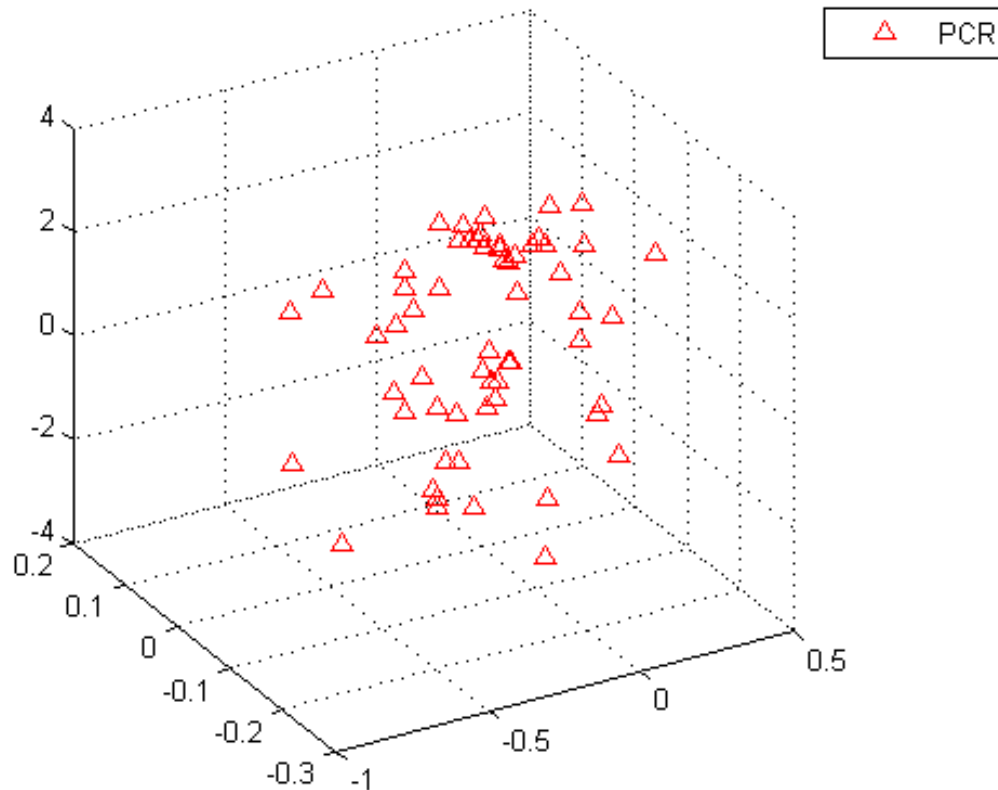
Another way to compare the predictive power of the two models is to plot the response variable against the two predictors in both cases.

```
plot3(Xscores(:,1),Xscores(:,2),y-mean(y),'bo');  
legend('PLSR');  
grid on; view(-30,30);
```



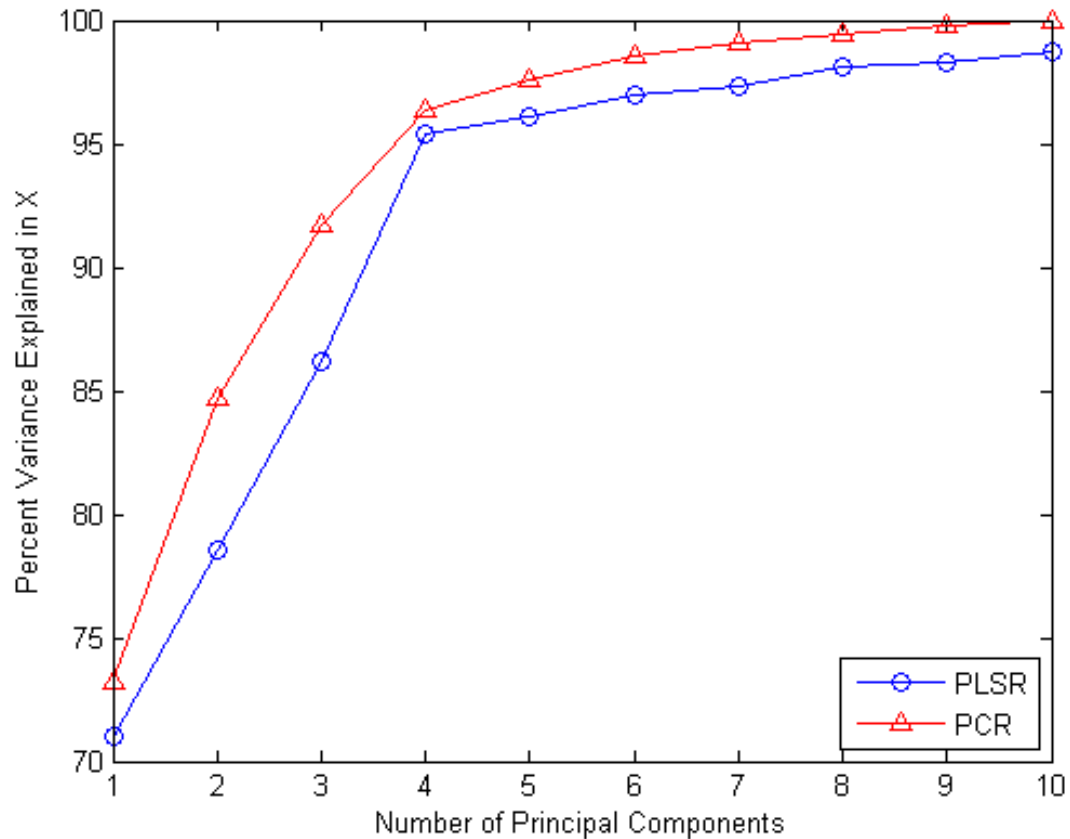
It's a little hard to see without being able to interactively rotate the figure, but the PLSR plot above shows points closely scattered about a plane. On the other hand, the PCR plot below shows a cloud of points with little indication of a linear relationship.

```
plot3(PCAScores(:,1),PCAScores(:,2),y-mean(y),'r^');  
legend('PCR');  
grid on; view(-30,30);
```



Notice that while the two PLS components are much better predictors of the observed  $y$ , the following figure shows that they explain somewhat less variance in the observed  $X$  than the first two principal components used in the PCR.

```
plot(1:10,100*cumsum(PLSPctVar(1,:)), 'b-o', 1:10, ...
     100*cumsum(PCAVar(1:10))/sum(PCAVar(1:10)), 'r-^');
xlabel('Number of Principal Components');
ylabel('Percent Variance Explained in X');
legend({'PLSR' 'PCR'}, 'location', 'SE');
```



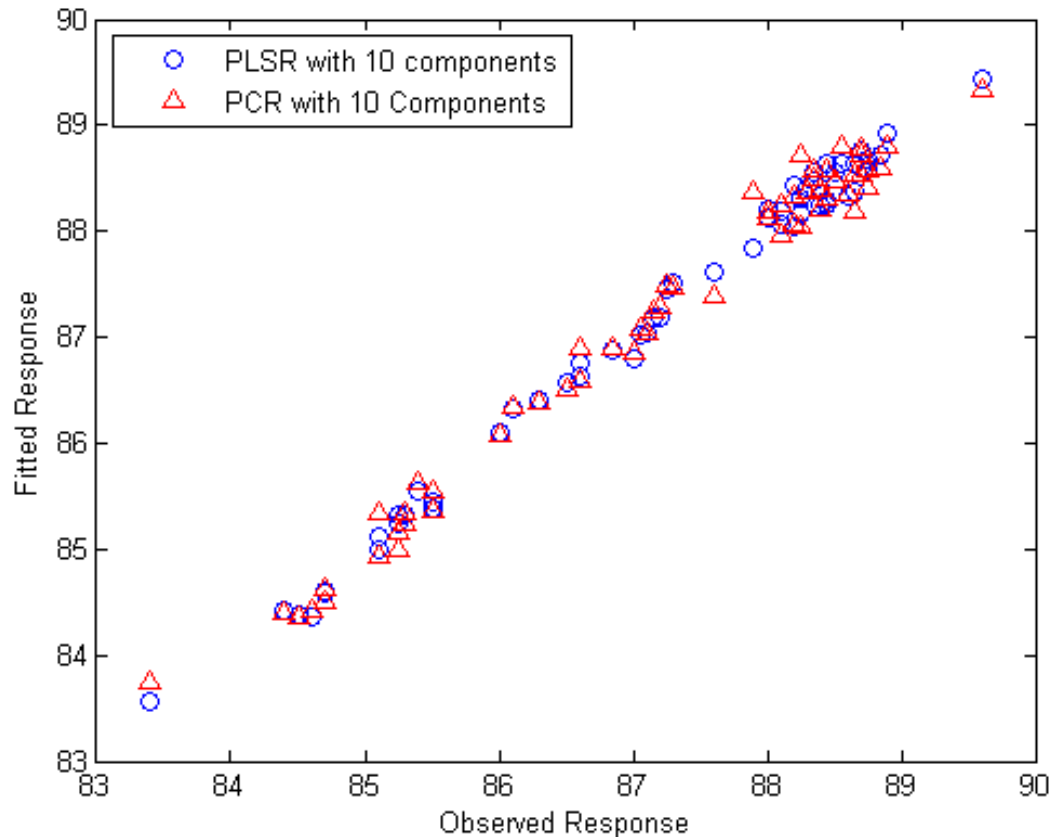
The fact that the PCR curve is uniformly higher suggests why PCR with two components does such a poor job, relative to PLSR, in fitting  $y$ . PCR constructs components to best explain  $X$ , and as a result, those first two components ignore the information in the data that is important in fitting the observed  $y$ .

#### **Fitting with More Components**

As more components are added in PCR, it will necessarily do a better job of fitting the original data  $y$ , simply because at some point most of the important

predictive information in  $X$  will be present in the principal components. For example, the following figure shows that the difference in residuals for the two methods is much less dramatic when using ten components than it was for two components.

```
yfitPLS10 = [ones(n,1) X]*betaPLS10;
betaPCR10 = regress(y-mean(y), PCAScores(:,1:10));
betaPCR10 = PCALoadings(:,1:10)*betaPCR10;
betaPCR10 = [mean(y) - mean(X)*betaPCR10; betaPCR10];
yfitPCR10 = [ones(n,1) X]*betaPCR10;
plot(y,yfitPLS10,'bo',y,yfitPCR10,'r^');
xlabel('Observed Response');
ylabel('Fitted Response');
legend({'PLSR with 10 components' 'PCR with 10 Components'}, ...
       'location','NW');
```



Both models fit  $y$  fairly accurately, although PLSR still makes a slightly more accurate fit. However, ten components is still an arbitrarily-chosen number for either model.

#### **Choosing the Number of Components with Cross-Validation**

It's often useful to choose the number of components to minimize the expected error when predicting the response from future observations on the predictor variables. Simply using a large number of components will do a good job in fitting the current observed data, but is a strategy that leads to overfitting.



Fitting the current data too well results in a model that does not generalize well to other data, and gives an overly-optimistic estimate of the expected error.

Cross-validation is a more statistically sound method for choosing the number of components in either PLSR or PCR. It avoids overfitting data by not reusing the same data to both fit a model and to estimate prediction error. Thus, the estimate of prediction error is not optimistically biased downwards.

`plsregress` has an option to estimate the mean squared prediction error (MSEP) by cross-validation, in this case using 10-fold C-V.

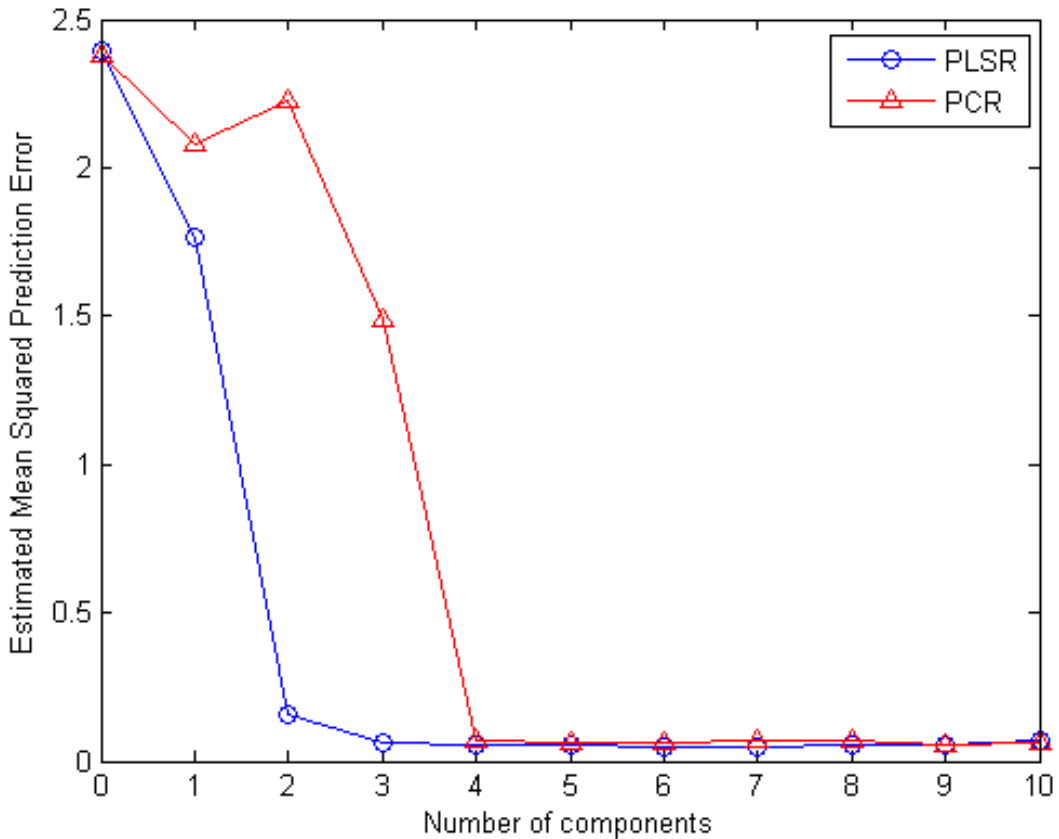
```
[X1,Y1,Xs,Ys,beta,pctVar,PLSmsep] = plsregress(X,y,10,'CV',10);
```

For PCR, `crossval` combined with a simple function to compute the sum of squared errors for PCR, can estimate the MSEP, again using 10-fold cross-validation.

```
PCRmsep = sum(crossval(@pcrsse,X,y,'KFold',10),1) / n;
```

The MSEP curve for PLSR indicates that two or three components does about as good a job as possible. On the other hand, PCR needs four components to get the same prediction accuracy.

```
plot(0:10,PLSmsep(2,:), 'b-o',0:10,PCRmsep, 'r-^');  
xlabel('Number of components');  
ylabel('Estimated Mean Squared Prediction Error');  
legend({'PLSR' 'PCR'}, 'location', 'NE');
```



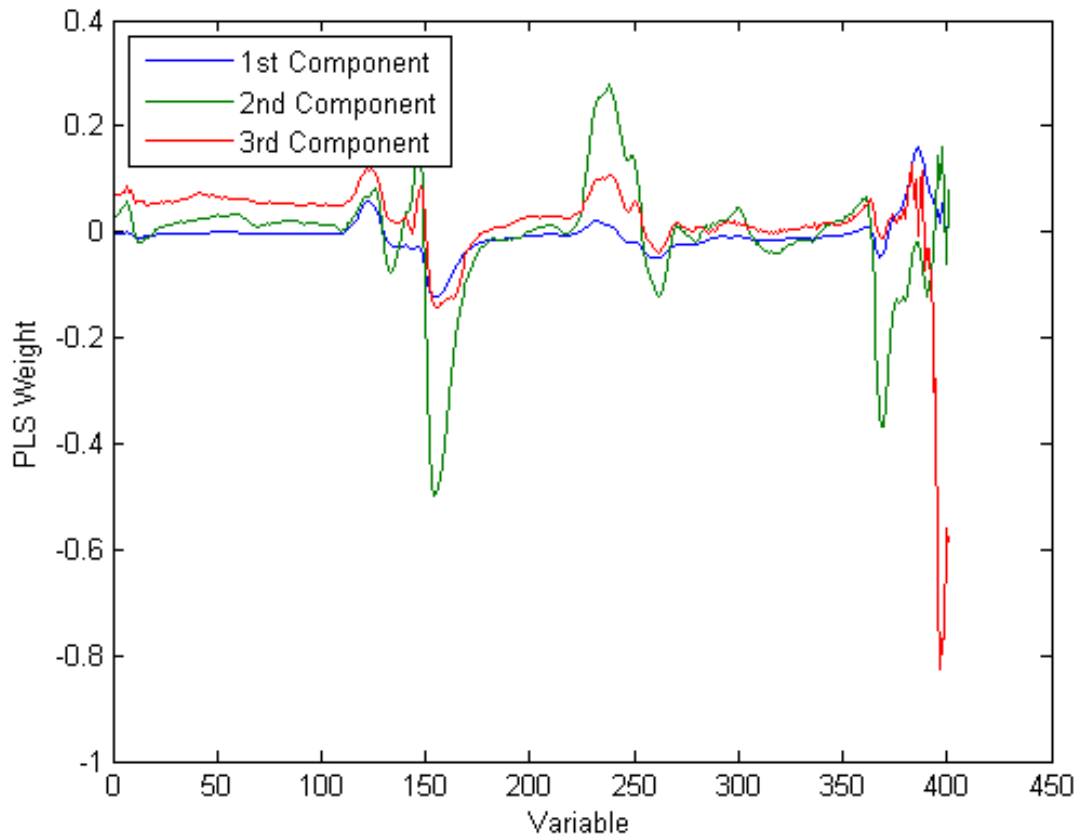
In fact, the second component in PCR *increases* the prediction error of the model, suggesting that the combination of predictor variables contained in that component is not strongly correlated with  $y$ . Again, that's because PCR constructs components to explain variation in  $X$ , not  $y$ .

### Model Parsimony

So if PCR requires four components to get the same prediction accuracy as PLSR with three components, is the PLSR model more parsimonious? That depends on what aspect of the model you consider.

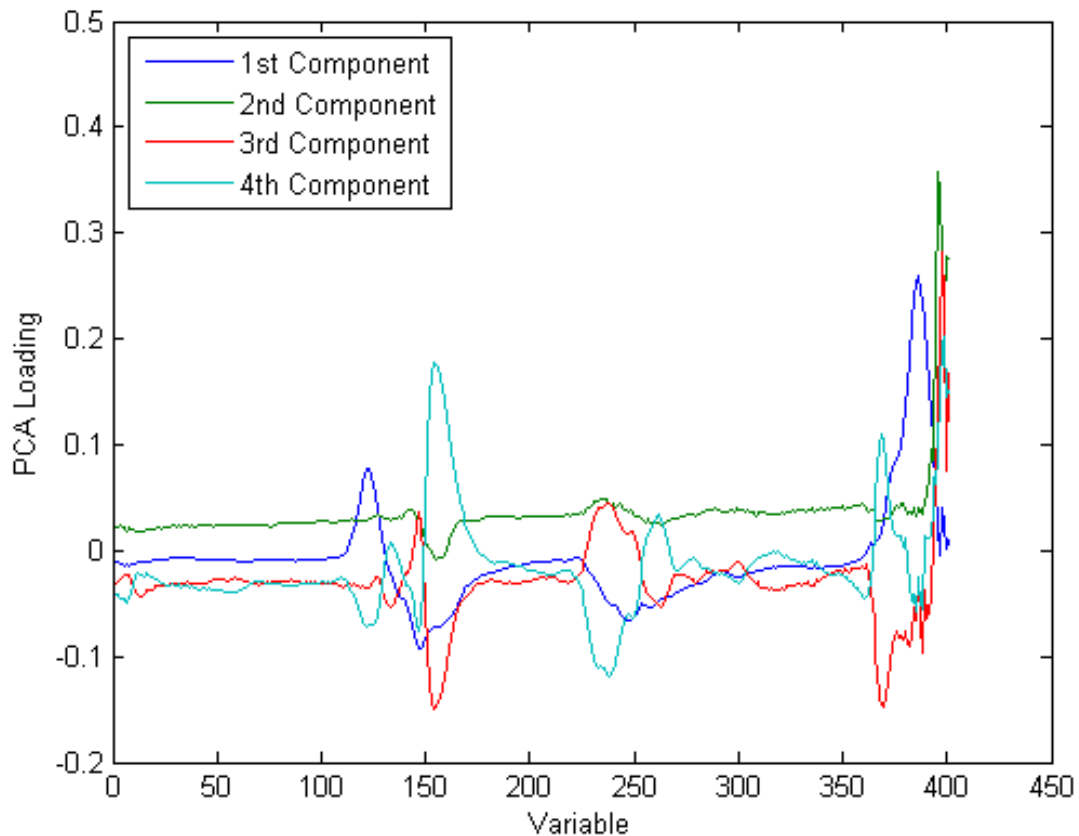
The PLS weights are the linear combinations of the original variables that define the PLS components, i.e., they describe how strongly each component in the PLSR depends on the original variables, and in what direction.

```
[X1,Y1,Xs,Ys,beta,pctVar,mse,stats] = plsregress(X,y,3);  
plot(1:401,stats.W,'-');  
xlabel('Variable');  
ylabel('PLS Weight');  
legend({'1st Component' '2nd Component' '3rd Component'}, ...  
       'location','NW');
```



Similarly, the PCA loadings describe how strongly each component in the PCR depends on the original variables.

```
plot(1:401,PCALoadings(:,1:4),'-');  
xlabel('Variable');  
ylabel('PCA Loading');  
legend({'1st Component' '2nd Component' '3rd Component' ...  
      '4th Component'},'location','NW');
```



For either PLSR or PCR, it may be that each component can be given a physically meaningful interpretation by inspecting which variables it weights

most heavily. For instance, with these spectral data it may be possible to interpret intensity peaks in terms of compounds present in the gasoline, and then to observe that weights for a particular component pick out a small number of those compounds. From that perspective, fewer components are simpler to interpret, and because PLSR often requires fewer components to predict the response adequately, it leads to more parsimonious models.

On the other hand, both PLSR and PCR result in one regression coefficient for each of the original predictor variables, plus an intercept. In that sense, neither is more parsimonious, because regardless of how many components are used, both models depend on all predictors. More concretely, for these data, both models need 401 spectral intensity values in order to make a prediction.

However, the ultimate goal may be to reduce the original set of variables to a smaller subset still able to predict the response accurately. For example, it may be possible to use the PLS weights or the PCA loadings to select only those variables that contribute most to each component. As shown earlier, some components from a PCR model fit may serve primarily to describe the variation in the predictor variables, and may include large weights for variables that are not strongly correlated with the response. Thus, PCR can lead to retaining variables that are unnecessary for prediction.

For the data used in this example, the difference in the number of components needed by PLSR and PCR for accurate prediction is not great, and the PLS weights and PCA loadings seem to pick out the same variables. That may not be true for other data.



# Cluster Analysis

---

- “Introduction to Cluster Analysis” on page 14-2
- “Hierarchical Clustering” on page 14-3
- “k-Means Clustering” on page 14-22
- “Gaussian Mixture Models” on page 14-30

## Introduction to Cluster Analysis

*Cluster analysis*, also called *segmentation analysis* or *taxonomy analysis*, creates groups, or *clusters*, of data. Clusters are formed in such a way that objects in the same cluster are very similar and objects in different clusters are very distinct. Measures of similarity depend on the application.

“Hierarchical Clustering” on page 14-3 groups data over a variety of scales by creating a cluster tree or *dendrogram*. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics Toolbox function `clusterdata` performs all of the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which may be used separately for more detailed analysis. The `dendrogram` function plots the cluster tree.

“k-Means Clustering” on page 14-22 is a partitioning method. The function `kmeans` partitions data into  $k$  mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering, k-means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that k-means clustering is often more suitable than hierarchical clustering for large amounts of data.

“Gaussian Mixture Models” on page 14-30 form clusters by representing the probability density function of observed variables as a mixture of multivariate normal densities. Mixture models of the `gmdistribution` class use an expectation maximization (EM) algorithm to fit data, which assigns posterior probabilities to each component density with respect to each observation. Clusters are assigned by selecting the component that maximizes the posterior probability. Clustering using Gaussian mixture models is sometimes considered a soft clustering method. The posterior probabilities for each point indicate that each data point has some probability of belonging to each cluster. Like k-means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than k-means clustering when clusters have different sizes and correlation within them.



# Hierarchical Clustering

## In this section...

“Introduction to Hierarchical Clustering” on page 14-3

“Algorithm Description” on page 14-3

“Similarity Measures” on page 14-4

“Linkages” on page 14-6

“Dendrograms” on page 14-8

“Verify the Cluster Tree” on page 14-10

“Create Clusters” on page 14-16

## Introduction to Hierarchical Clustering

Hierarchical clustering groups data over a variety of scales by creating a cluster tree or *dendrogram*. The tree is not a single set of clusters, but rather a multilevel hierarchy, where clusters at one level are joined as clusters at the next level. This allows you to decide the level or scale of clustering that is most appropriate for your application. The Statistics Toolbox function `clusterdata` supports agglomerative clustering and performs all of the necessary steps for you. It incorporates the `pdist`, `linkage`, and `cluster` functions, which you can use separately for more detailed analysis. The `dendrogram` function plots the cluster tree.

## Algorithm Description

To perform agglomerative hierarchical cluster analysis on a data set using Statistics Toolbox functions, follow this procedure:

- 1 Find the similarity or dissimilarity between every pair of objects in the data set.** In this step, you calculate the *distance* between objects using the `pdist` function. The `pdist` function supports many different ways to compute this measurement. See “Similarity Measures” on page 14-4 for more information.
- 2 Group the objects into a binary, hierarchical cluster tree.** In this step, you link pairs of objects that are in close proximity using the `linkage`

function. The `linkage` function uses the distance information generated in step 1 to determine the proximity of objects to each other. As objects are paired into binary clusters, the newly formed clusters are grouped into larger clusters until a hierarchical tree is formed. See “Linkages” on page 14-6 for more information.

**3 Determine where to cut the hierarchical tree into clusters.** In this step, you use the `cluster` function to prune branches off the bottom of the hierarchical tree, and assign all the objects below each cut to a single cluster. This creates a partition of the data. The `cluster` function can create these clusters by detecting natural groupings in the hierarchical tree or by cutting off the hierarchical tree at an arbitrary point.

The following sections provide more information about each of these steps.

---

**Note** The Statistics Toolbox function `clusterdata` performs all of the necessary steps for you. You do not need to execute the `pdist`, `linkage`, or `cluster` functions separately.

---

## Similarity Measures

You use the `pdist` function to calculate the distance between every pair of objects in a data set. For a data set made up of  $m$  objects, there are  $m*(m - 1)/2$  pairs in the data set. The result of this computation is commonly known as a distance or dissimilarity matrix.

There are many ways to calculate this distance information. By default, the `pdist` function calculates the Euclidean distance between objects; however, you can specify one of several other options. See `pdist` for more information.

---

**Note** You can optionally normalize the values in the data set before calculating the distance information. In a real world data set, variables can be measured against different scales. For example, one variable can measure Intelligence Quotient (IQ) test scores and another variable can measure head circumference. These discrepancies can distort the proximity calculations. Using the `zscore` function, you can convert all the values in the data set to use the same proportional scale. See `zscore` for more information.

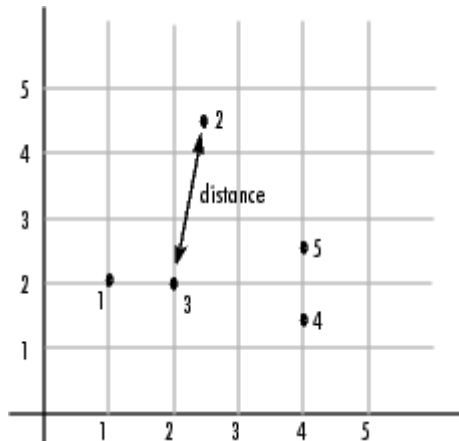
---

For example, consider a data set,  $X$ , made up of five objects where each object is a set of  $x,y$  coordinates.

- **Object 1:** 1, 2
- **Object 2:** 2.5, 4.5
- **Object 3:** 2, 2
- **Object 4:** 4, 1.5
- **Object 5:** 4, 2.5

You can define this data set as a matrix

and pass it to `pdist`. The `pdist` function calculates the distance between object 1 and object 2, object 1 and object 3, and so on until the distances between all the pairs have been calculated. The following figure plots these objects in a graph. The Euclidean distance between object 2 and object 3 is shown to illustrate one interpretation of distance.



### Distance Information

The `pdist` function returns this distance information in a vector,  $Y$ , where each element contains the distance between a pair of objects.

$Y = \text{pdist}(X)$

```

Y =

Columns 1 through 7

    2.9155    1.0000    3.0414    3.0414    2.5495    3.3541    2.5000

Columns 8 through 10

    2.0616    2.0616    1.0000

```

To make it easier to see the relationship between the distance information generated by `pdist` and the objects in the original data set, you can reformat the distance vector into a matrix using the `squareform` function. In this matrix, element  $i,j$  corresponds to the distance between object  $i$  and object  $j$  in the original data set. In the following example, element 1,1 represents the distance between object 1 and itself (which is zero). Element 1,2 represents the distance between object 1 and object 2, and so on.

```
squareform(Y)
```

```

ans =

    0    2.9155    1.0000    3.0414    3.0414
  2.9155    0    2.5495    3.3541    2.5000
  1.0000    2.5495    0    2.0616    2.0616
  3.0414    3.3541    2.0616    0    1.0000
  3.0414    2.5000    2.0616    1.0000    0

```

## Linkages

Once the proximity between objects in the data set has been computed, you can determine how objects in the data set should be grouped into clusters, using the `linkage` function. The `linkage` function takes the distance information generated by `pdist` and links pairs of objects that are close together into binary clusters (clusters made up of two objects). The `linkage` function then links these newly formed clusters to each other and to other

objects to create bigger clusters until all the objects in the original data set are linked together in a hierarchical tree.

For example, given the distance vector `Y` generated by `pdist` from the sample data set of  $x$ - and  $y$ -coordinates, the `linkage` function generates a hierarchical cluster tree, returning the linkage information in a matrix, `Z`.

```
Z = linkage(Y)
```

```
Z =
```

```

4.0000    5.0000    1.0000
1.0000    3.0000    1.0000
6.0000    7.0000    2.0616
2.0000    8.0000    2.5000
```

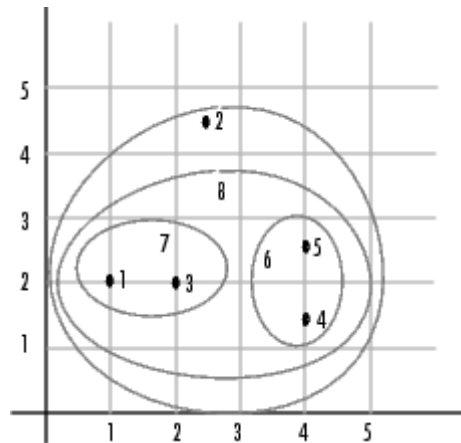
In this output, each row identifies a link between objects or clusters. The first two columns identify the objects that have been linked. The third column contains the distance between these objects. For the sample data set of  $x$ - and  $y$ -coordinates, the `linkage` function begins by grouping objects 4 and 5, which have the closest proximity (distance value = 1.0000). The `linkage` function continues by grouping objects 1 and 3, which also have a distance value of 1.0000.

The third row indicates that the `linkage` function grouped objects 6 and 7. If the original sample data set contained only five objects, what are objects 6 and 7? Object 6 is the newly formed binary cluster created by the grouping of objects 4 and 5. When the `linkage` function groups two objects into a new cluster, it must assign the cluster a unique index value, starting with the value  $m+1$ , where  $m$  is the number of objects in the original data set. (Values 1 through  $m$  are already used by the original data set.) Similarly, object 7 is the cluster formed by grouping objects 1 and 3.

`linkage` uses distances to determine the order in which it clusters objects. The distance vector `Y` contains the distances between the original objects 1 through 5. But `linkage` must also be able to determine distances involving clusters that it creates, such as objects 6 and 7. By default, `linkage` uses a

method known as single linkage. However, there are a number of different methods available. See the `linkage` reference page for more information.

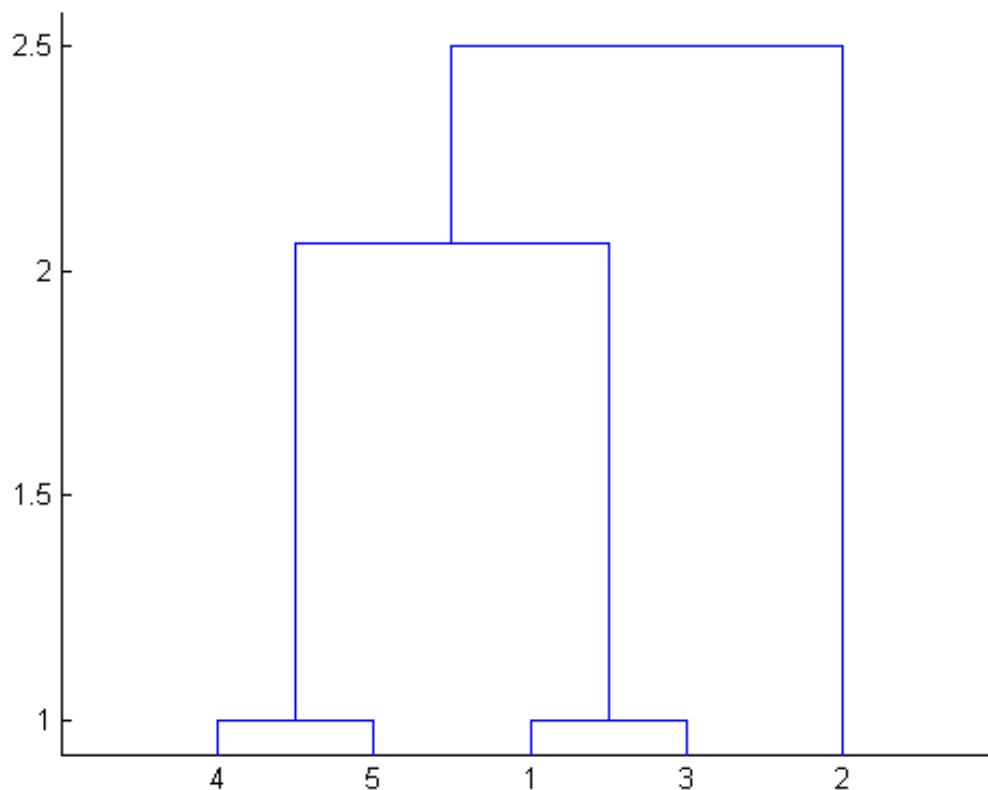
As the final cluster, the `linkage` function grouped object 8, the newly formed cluster made up of objects 6 and 7, with object 2 from the original data set. The following figure graphically illustrates the way `linkage` groups the objects into a hierarchy of clusters.



## Dendrograms

The hierarchical, binary cluster tree created by the `linkage` function is most easily understood when viewed graphically. The Statistics Toolbox function `dendrogram` plots the tree as follows.

```
dendrogram(Z)
```



In the figure, the numbers along the horizontal axis represent the indices of the objects in the original data set. The links between objects are represented as upside-down U-shaped lines. The height of the U indicates the distance between the objects. For example, the link representing the cluster containing objects 1 and 3 has a height of 1. The link representing the cluster that groups object 2 together with objects 1, 3, 4, and 5, (which are already clustered as object 8) has a height of 2.5. The height represents the distance linkage computes between objects 2 and 8. For more information about creating a dendrogram diagram, see the `dendrogram` reference page.

## Verify the Cluster Tree

After linking the objects in a data set into a hierarchical cluster tree, you might want to verify that the distances (that is, heights) in the tree reflect the original distances accurately. In addition, you might want to investigate natural divisions that exist among links between objects. Statistics Toolbox functions are available for both of these tasks, as described in the following sections.

- “Verify Dissimilarity” on page 14-10
- “Verify Consistency” on page 14-11

### Verify Dissimilarity

In a hierarchical cluster tree, any two objects in the original data set are eventually linked together at some level. The height of the link represents the distance between the two clusters that contain those two objects. This height is known as the *cophenetic distance* between the two objects. One way to measure how well the cluster tree generated by the `linkage` function reflects your data is to compare the cophenetic distances with the original distance data generated by the `pdist` function. If the clustering is valid, the linking of objects in the cluster tree should have a strong correlation with the distances between objects in the distance vector. The `cophenet` function compares these two sets of values and computes their correlation, returning a value called the *cophenetic correlation coefficient*. The closer the value of the cophenetic correlation coefficient is to 1, the more accurately the clustering solution reflects your data.

You can use the cophenetic correlation coefficient to compare the results of clustering the same data set using different distance calculation methods or clustering algorithms. For example, you can use the `cophenet` function to evaluate the clusters created for the sample data set.

```
c = cophenet(Z,Y)
```

```
c =
```

```
0.8615
```



Z is the matrix output by the linkage function and Y is the distance vector output by the pdist function.

Execute pdist again on the same data set, this time specifying the city block metric. After running the linkage function on this new pdist output using the average linkage method, call cophenet to evaluate the clustering solution.

```
Y = pdist(X, 'cityblock');  
Z = linkage(Y, 'average');  
c = cophenet(Z,Y)
```

```
c =  
  
    0.9047
```

The cophenetic correlation coefficient shows that using a different distance and linkage method creates a tree that represents the original distances slightly better.

### **Verify Consistency**

One way to determine the natural cluster divisions in a data set is to compare the height of each link in a cluster tree with the heights of neighboring links below it in the tree.

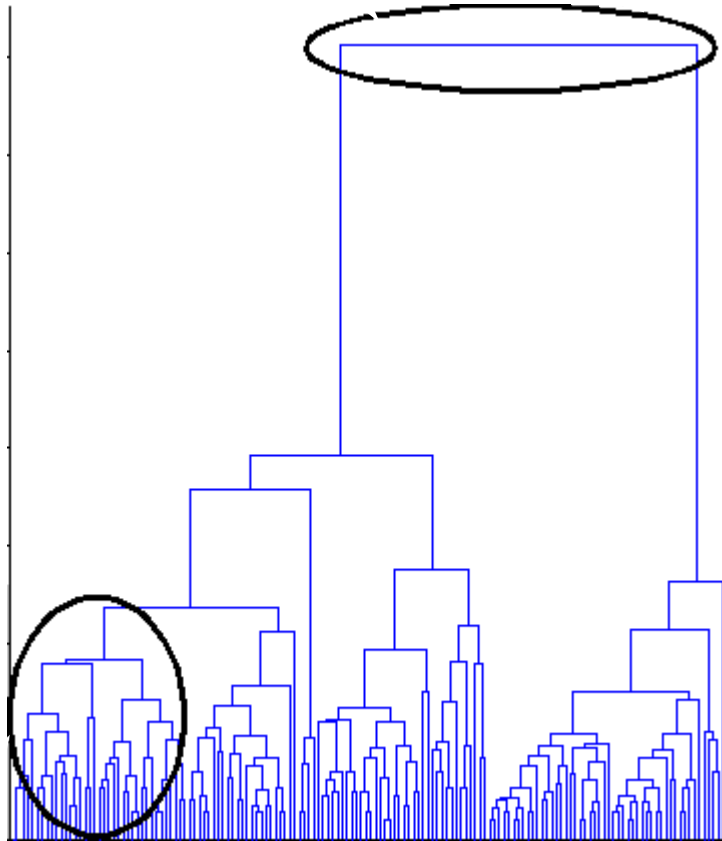
A link that is approximately the same height as the links below it indicates that there are no distinct divisions between the objects joined at this level of the hierarchy. These links are said to exhibit a high level of consistency, because the distance between the objects being joined is approximately the same as the distances between the objects they contain.

On the other hand, a link whose height differs noticeably from the height of the links below it indicates that the objects joined at this level in the cluster tree are much farther apart from each other than their components were when they were joined. This link is said to be inconsistent with the links below it.

In cluster analysis, inconsistent links can indicate the border of a natural division in a data set. The `cluster` function uses a quantitative measure of inconsistency to determine where to partition your data set into clusters.

The following dendrogram illustrates inconsistent links. Note how the objects in the dendrogram fall into two groups that are connected by links at a much higher level in the tree. These links are inconsistent when compared with the links below them in the hierarchy.

These links show inconsistency when compared to the links below them.



These links show consistency.

The relative consistency of each link in a hierarchical cluster tree can be quantified and expressed as the *inconsistency coefficient*. This value compares the height of a link in a cluster hierarchy with the average height of links below it. Links that join distinct clusters have a high inconsistency coefficient; links that join indistinct clusters have a low inconsistency coefficient.

To generate a listing of the inconsistency coefficient for each link in the cluster tree, use the `inconsistent` function. By default, the `inconsistent` function compares each link in the cluster hierarchy with adjacent links that are less than two levels below it in the cluster hierarchy. This is called the *depth* of the comparison. You can also specify other depths. The objects at the bottom of the cluster tree, called leaf nodes, that have no further objects below them, have an inconsistency coefficient of zero. Clusters that join two leaves also have a zero inconsistency coefficient.

For example, you can use the `inconsistent` function to calculate the inconsistency values for the links created by the `linkage` function in “Linkages” on page 14-6.

First, recompute the distance and linkage values using the default settings.

```
Y = pdist(X);
Z = linkage(Y);
```

Next, use `inconsistent` to calculate the inconsistency values.

```
I = inconsistent(Z)
```

```
I =
```

1.0000	0	1.0000	0
1.0000	0	1.0000	0
1.3539	0.6129	3.0000	1.1547
2.2808	0.3100	2.0000	0.7071

The `inconsistent` function returns data about the links in an  $(m-1)$ -by-4 matrix, whose columns are described in the following table.

Column	Description
1	Mean of the heights of all the links included in the calculation
2	Standard deviation of all the links included in the calculation
3	Number of links included in the calculation
4	Inconsistency coefficient

In the sample output, the first row represents the link between objects 4 and 5. This cluster is assigned the index 6 by the `linkage` function. Because both 4 and 5 are leaf nodes, the inconsistency coefficient for the cluster is zero. The second row represents the link between objects 1 and 3, both of which are also leaf nodes. This cluster is assigned the index 7 by the `linkage` function.

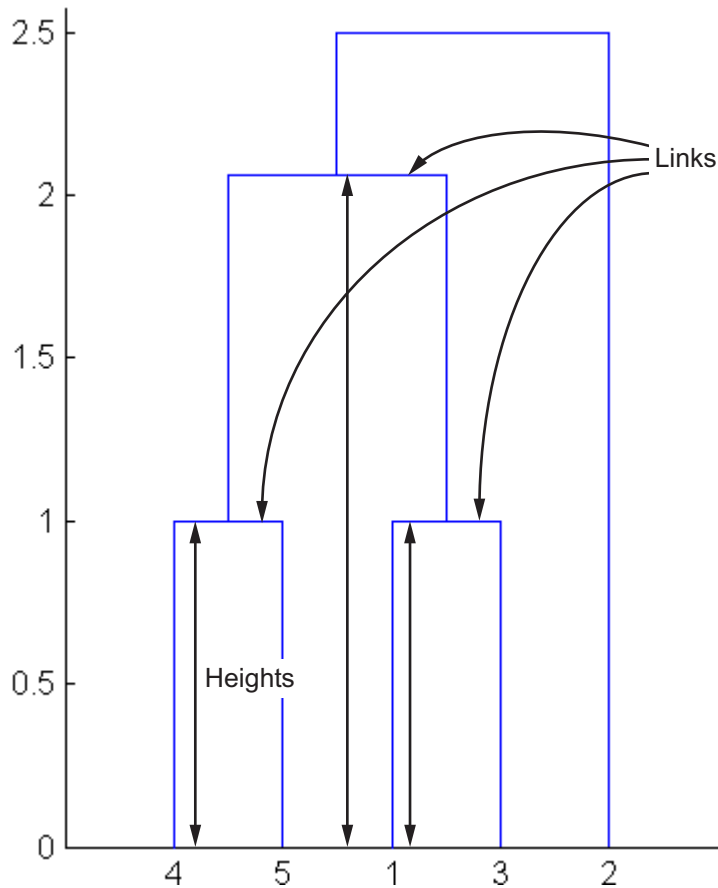
The third row evaluates the link that connects these two clusters, objects 6 and 7. (This new cluster is assigned index 8 in the `linkage` output). Column 3 indicates that three links are considered in the calculation: the link itself and the two links directly below it in the hierarchy. Column 1 represents the mean of the heights of these links. The `inconsistent` function uses the height information output by the `linkage` function to calculate the mean. Column 2 represents the standard deviation between the links. The last column contains the inconsistency value for these links, 1.1547. It is the difference between the current link height and the mean, normalized by the standard deviation.

$$(2.0616 - 1.3539) / .6129$$

ans =

$$1.1547$$

The following figure illustrates the links and heights included in this calculation.



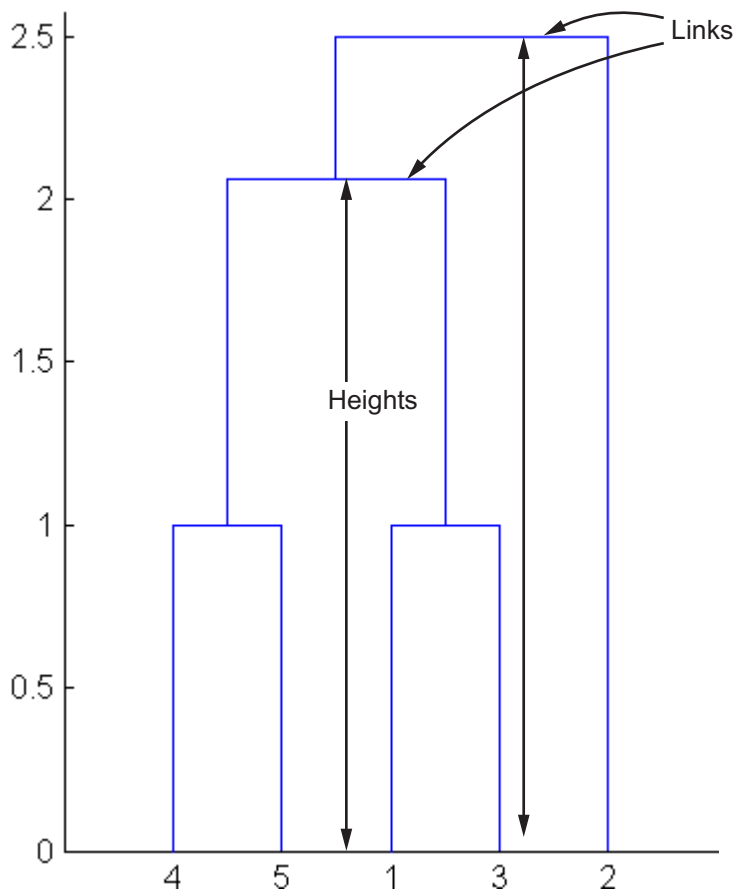

---

**Note** In the preceding figure, the lower limit on the y-axis is set to 0 to show the heights of the links. To set the lower limit to 0, select **Axes Properties** from the **Edit** menu, click the **Y Axis** tab, and enter 0 in the field immediately to the right of **Y Limits**.

---

Row 4 in the output matrix describes the link between object 8 and object 2. Column 3 indicates that two links are included in this calculation: the link itself and the link directly below it in the hierarchy. The inconsistency coefficient for this link is 0.7071.

The following figure illustrates the links and heights included in this calculation.



### Create Clusters

After you create the hierarchical tree of binary clusters, you can prune the tree to partition your data into clusters using the `cluster` function. The `cluster` function lets you create clusters in two ways, as discussed in the following sections:

- “Find Natural Divisions in Data” on page 14-17

- “Specify Arbitrary Clusters” on page 14-18

### Find Natural Divisions in Data

The hierarchical cluster tree may naturally divide the data into distinct, well-separated clusters. This can be particularly evident in a dendrogram diagram created from data where groups of objects are densely packed in certain areas and not in others. The inconsistency coefficient of the links in the cluster tree can identify these divisions where the similarities between objects change abruptly. (See “Verify the Cluster Tree” on page 14-10 for more information about the inconsistency coefficient.) You can use this value to determine where the `cluster` function creates cluster boundaries.

For example, if you use the `cluster` function to group the sample data set into clusters, specifying an inconsistency coefficient threshold of 1.2 as the value of the `cutoff` argument, the `cluster` function groups all the objects in the sample data set into one cluster. In this case, none of the links in the cluster hierarchy had an inconsistency coefficient greater than 1.2.

```
T = cluster(Z, 'cutoff', 1.2)
```

```
T =
```

```

1
1
1
1
1
```

The `cluster` function outputs a vector, `T`, that is the same size as the original data set. Each element in this vector contains the number of the cluster into which the corresponding object from the original data set was placed.

If you lower the inconsistency coefficient threshold to 0.8, the `cluster` function divides the sample data set into three separate clusters.

```
T = cluster(Z, 'cutoff', 0.8)
```

```
T =  
  
    3  
    2  
    3  
    1  
    1
```

This output indicates that objects 1 and 3 were placed in cluster 1, objects 4 and 5 were placed in cluster 2, and object 2 was placed in cluster 3.

When clusters are formed in this way, the cutoff value is applied to the inconsistency coefficient. These clusters may, but do not necessarily, correspond to a horizontal slice across the dendrogram at a certain height. If you want clusters corresponding to a horizontal slice of the dendrogram, you can either use the `criterion` option to specify that the cutoff should be based on distance rather than inconsistency, or you can specify the number of clusters directly as described in the following section.

### Specify Arbitrary Clusters

Instead of letting the `cluster` function create clusters determined by the natural divisions in the data set, you can specify the number of clusters you want created.

For example, you can specify that you want the `cluster` function to partition the sample data set into two clusters. In this case, the `cluster` function creates one cluster containing objects 1, 3, 4, and 5 and another cluster containing object 2.

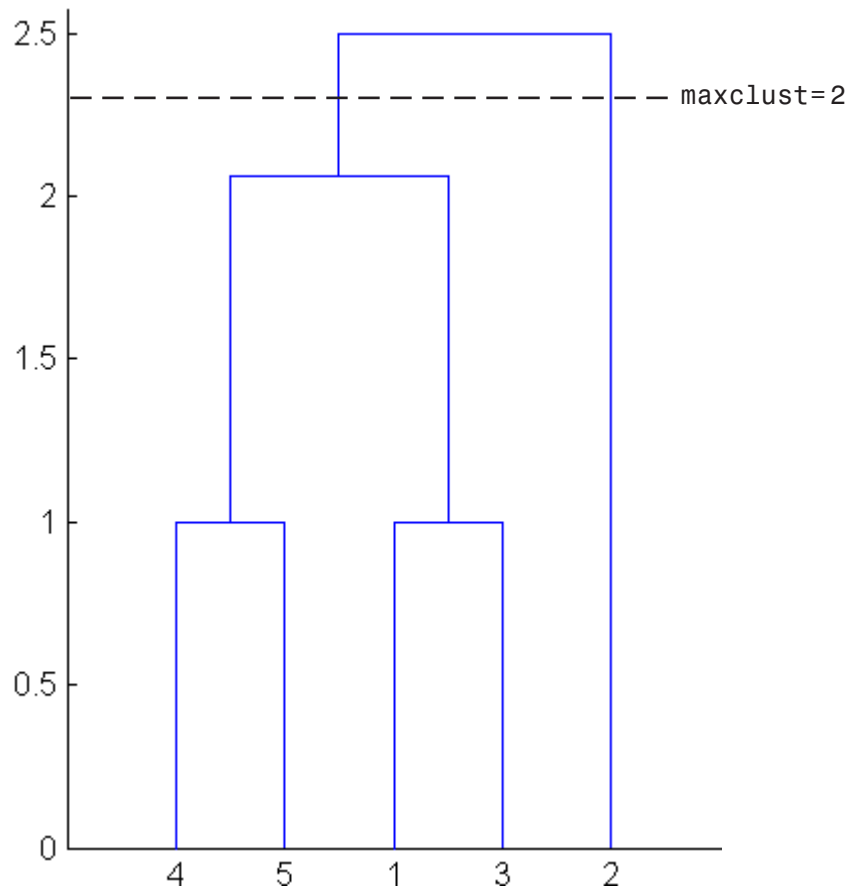
```
T = cluster(Z, 'maxclust', 2)
```

```
T =  
  
    2  
    1  
    2
```



2  
2

To help you visualize how the `cluster` function determines these clusters, the following figure shows the dendrogram of the hierarchical cluster tree. The horizontal dashed line intersects two lines of the dendrogram, corresponding to setting `'maxclust'` to 2. These two lines partition the objects into two clusters: the objects below the left-hand line, namely 1, 3, 4, and 5, belong to one cluster, while the object below the right-hand line, namely 2, belongs to the other cluster.



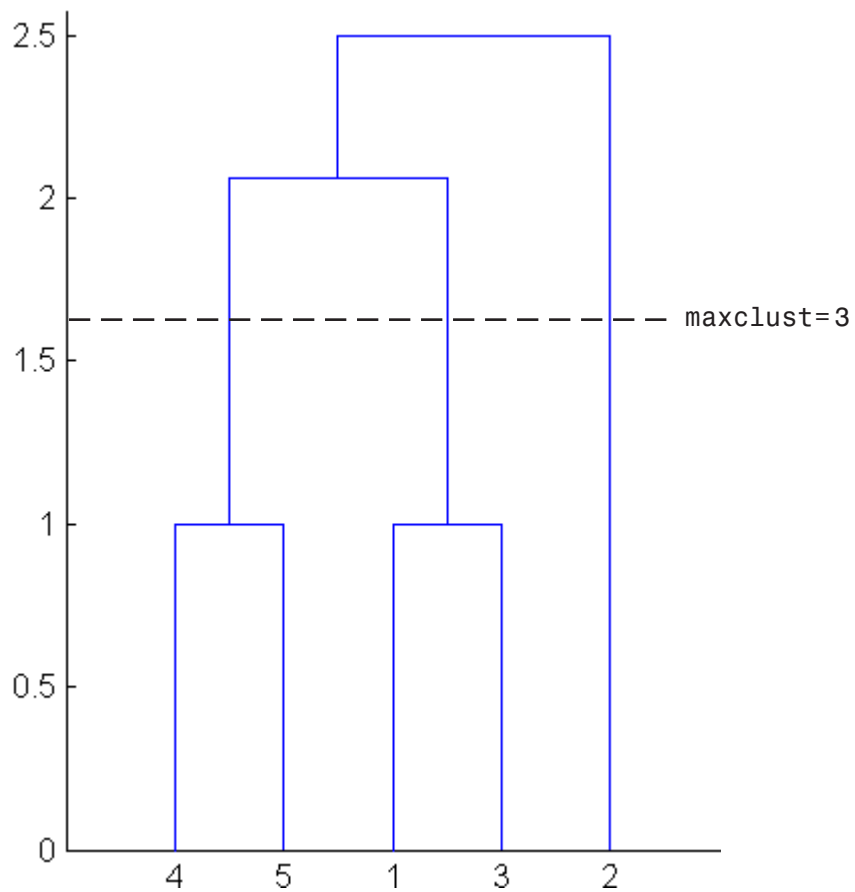
On the other hand, if you set 'maxclust' to 3, the cluster function groups objects 4 and 5 in one cluster, objects 1 and 3 in a second cluster, and object 2 in a third cluster. The following command illustrates this.

```
T = cluster(Z, 'maxclust', 3)
```

```
T =
```

```
2  
3  
2  
1  
1
```

This time, the cluster function cuts off the hierarchy at a lower point, corresponding to the horizontal line that intersects three lines of the dendrogram in the following figure.



## k-Means Clustering

### In this section...

“Introduction to k-Means Clustering” on page 14-22

“Create Clusters and Determine Separation” on page 14-23

“Determine the Correct Number of Clusters” on page 14-25

“Avoid Local Minima” on page 14-28

### Introduction to k-Means Clustering

k-means clustering is a partitioning method. The function `kmeans` partitions data into  $k$  mutually exclusive clusters, and returns the index of the cluster to which it has assigned each observation. Unlike hierarchical clustering, k-means clustering operates on actual observations (rather than the larger set of dissimilarity measures), and creates a single level of clusters. The distinctions mean that k-means clustering is often more suitable than hierarchical clustering for large amounts of data.

`kmeans` treats each observation in your data as an object having a location in space. It finds a partition in which objects within each cluster are as close to each other as possible, and as far from objects in other clusters as possible. You can choose from five different distance measures, depending on the kind of data you are clustering.

Each cluster in the partition is defined by its member objects and by its centroid, or center. The centroid for each cluster is the point to which the sum of distances from all objects in that cluster is minimized. `kmeans` computes cluster centroids differently for each distance measure, to minimize the sum with respect to the measure that you specify.

`kmeans` uses an iterative algorithm that minimizes the sum of distances from each object to its cluster centroid, over all clusters. This algorithm moves objects between clusters until the sum cannot be decreased further. The result is a set of clusters that are as compact and well-separated as possible. You can control the details of the minimization using several optional input parameters to `kmeans`, including ones for the initial values of the cluster centroids, and for the maximum number of iterations.

## Create Clusters and Determine Separation

The following example explores possible clustering in four-dimensional data by analyzing the results of partitioning the points into three, four, and five clusters.

---

**Note** Because each part of this example generates random numbers sequentially, i.e., without setting a new state, you must perform all steps in sequence to duplicate the results shown. If you perform the steps out of sequence, the answers will be essentially the same, but the intermediate results, number of iterations, or ordering of the silhouette plots may differ.

---

First, load some data:

```
rng('default'); % For reproducibility
load kmeansdata;
size(X)
```

```
ans =
```

```
560    4
```

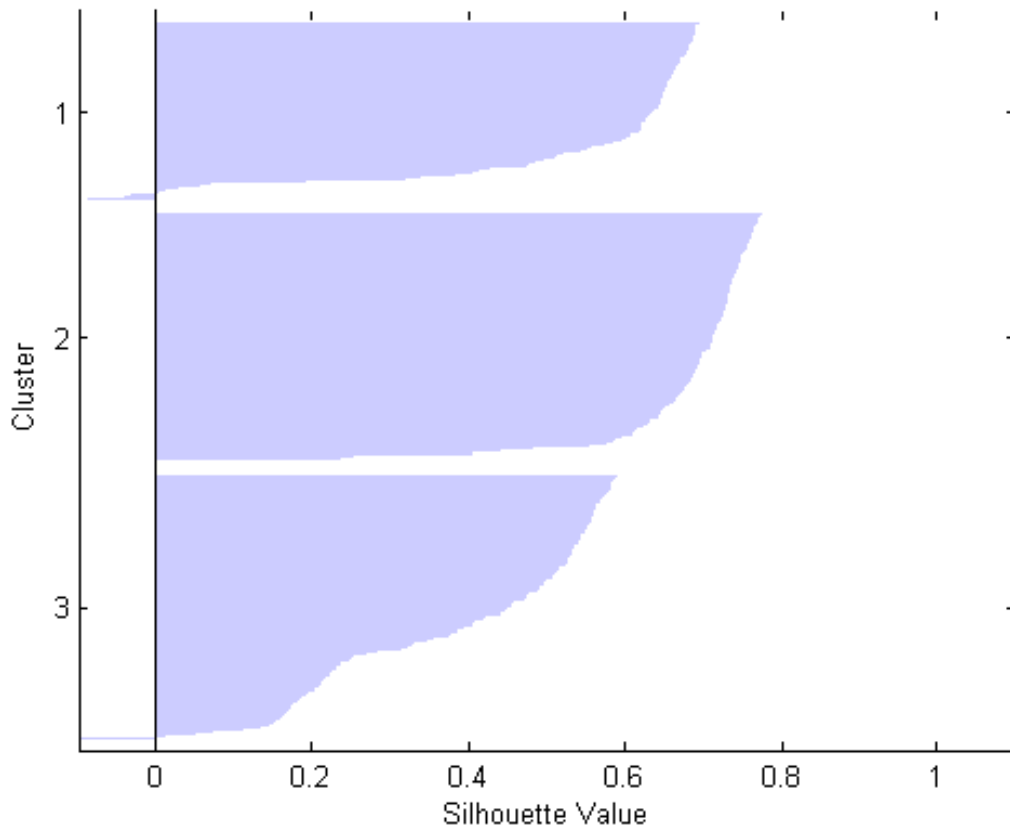
Even though these data are four-dimensional, and cannot be easily visualized, `kmeans` enables you to investigate whether a group structure exists in them. Call `kmeans` with `k`, the desired number of clusters, equal to 3. For this example, specify the city block distance measure, and use the default starting method of initializing centroids from randomly selected data points.

```
idx3 = kmeans(X,3,'distance','city');
```

To get an idea of how well-separated the resulting clusters are, you can make a silhouette plot using the cluster indices output from `kmeans`. The silhouette plot displays a measure of how close each point in one cluster is to points in the neighboring clusters. This measure ranges from +1, indicating points that are very distant from neighboring clusters, through 0, indicating points that are not distinctly in one cluster or another, to -1, indicating points that are

probably assigned to the wrong cluster. `silhouette` returns these values in its first output.

```
[silh3,h] = silhouette(X,idx3,'city');  
set(get(gca,'Children'),'FaceColor',[.8 .8 1])  
xlabel('Silhouette Value')  
ylabel('Cluster')
```



From the silhouette plot, you can see that most points in the second cluster have a large silhouette value, greater than 0.6, indicating that the cluster is somewhat separated from neighboring clusters. However, the third cluster

contains many points with low silhouette values, and the first contains a few points with negative values, indicating that those two clusters are not well separated.

## Determine the Correct Number of Clusters

Increase the number of clusters to see if `kmeans` can find a better grouping of the data. This time, use the optional `'display'` parameter to print information about each iteration.

```
idx4 = kmeans(X,4, 'dist','city', 'display','iter');
```

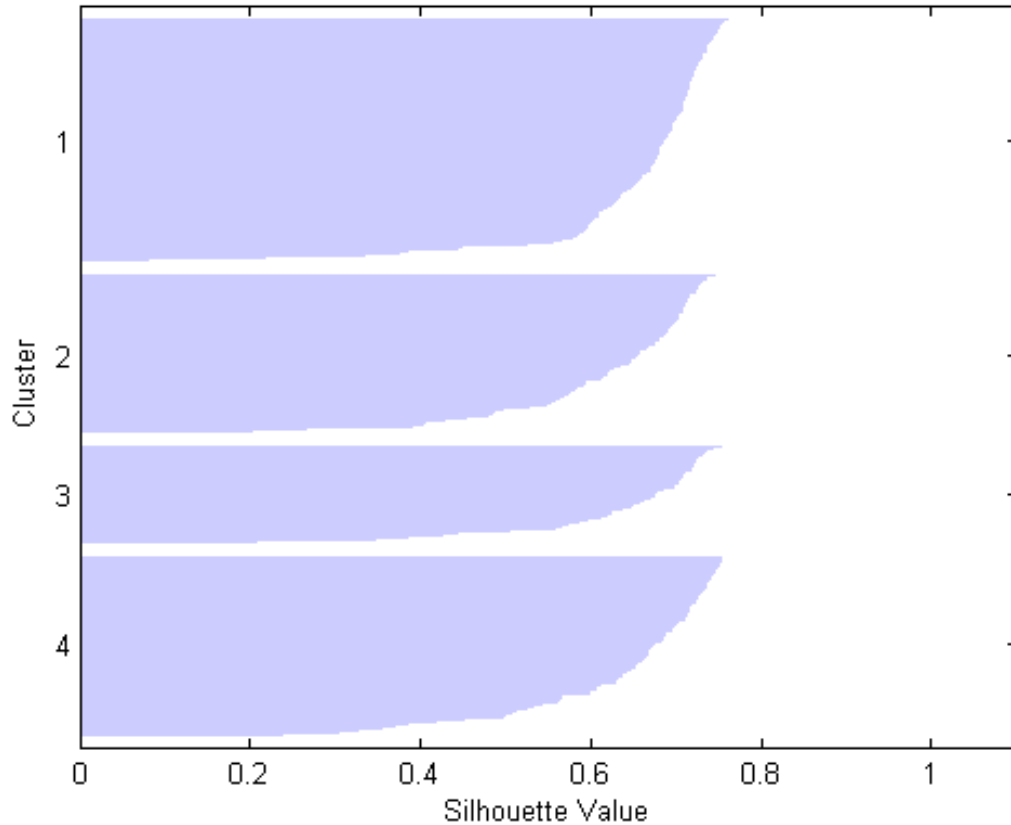
```

iter phase      num      sum
   1     1      560    2077.43
   2     1       51    1778.64
   3     1        3    1771.1
   4     2         0    1771.1
Best total sum of distances = 1771.1
```

Notice that the total sum of distances decreases at each iteration as `kmeans` reassigns points between clusters and recomputes cluster centroids. In this case, the second phase of the algorithm did not make any reassignments, indicating that the first phase reached a minimum after five iterations. In some problems, the first phase might not reach a minimum, but the second phase always will.

A silhouette plot for this solution indicates that these four clusters are better separated than the three in the previous solution.

```
[silh4,h] = silhouette(X,idx4,'city');
set(get(gca,'Children'),'FaceColor',[.8 .8 1])
xlabel('Silhouette Value')
ylabel('Cluster')
```



A more quantitative way to compare the two solutions is to look at the average silhouette values for the two cases.

```
cluster3 = mean(silh3)
cluster4 = mean(silh4)
```

```
cluster3 =
    0.5352
```



```
cluster4 =
```

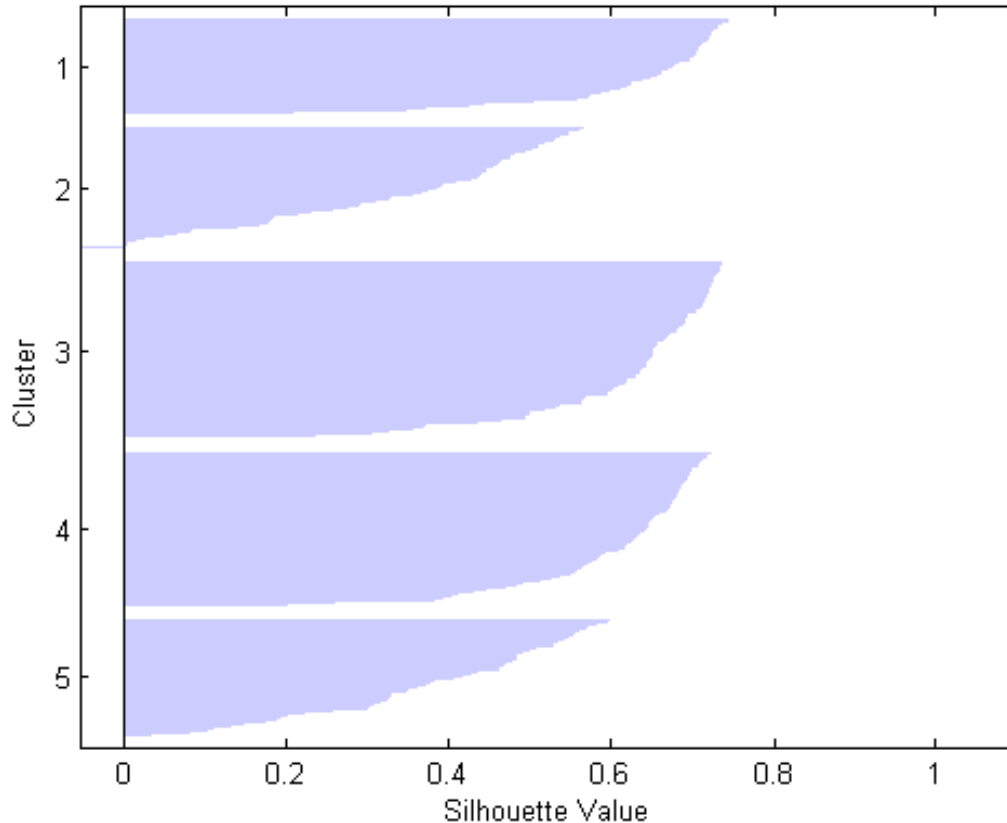
```
    0.6400
```

Finally, try clustering the data using five clusters.

```
idx5 = kmeans(X,5,'dist','city','replicates',5);  
[silh5,h] = silhouette(X,idx5,'city');  
set(get(gca,'Children'),'FaceColor',[.8 .8 1])  
xlabel('Silhouette Value')  
ylabel('Cluster')  
mean(silh5)
```

```
ans =
```

```
    0.5266
```



This silhouette plot indicates that this is probably not the right number of clusters, since two of the clusters contain points with mostly low silhouette values. Without some knowledge of how many clusters are really in the data, it is a good idea to experiment with a range of values for  $k$ .

### **Avoid Local Minima**

Like many other types of numerical minimizations, the solution that  $k$ means reaches often depends on the starting points. It is possible for  $k$ means to reach a local minimum, where reassigning any one point to a new cluster

would increase the total sum of point-to-centroid distances, but where a better solution does exist. However, you can use the optional 'replicates' parameter to overcome that problem.

For four clusters, specify five replicates, and use the 'display' parameter to print out the final sum of distances for each of the solutions.

```
[idx4,cent4,sumdist] = kmeans(X,4,'dist','city',...  
                             'display','final','replicates',5);
```

```
Replicate 1, 4 iterations, total sum of distances = 1771.1.  
Replicate 2, 7 iterations, total sum of distances = 1771.1.  
Replicate 3, 8 iterations, total sum of distances = 1771.1.  
Replicate 4, 5 iterations, total sum of distances = 1771.1.  
Replicate 5, 6 iterations, total sum of distances = 1771.1.  
Best total sum of distances = 1771.1
```

In this example, `kmeans` found the same minimum in all five replications. However, even for relatively simple problems, nonglobal minima do exist. Each of these five replicates began from a different randomly selected set of initial centroids, so sometimes `kmeans` finds more than one local minimum. However, the final solution that `kmeans` returns is the one with the lowest total sum of distances, over all replicates.

```
sum(sumdist)
```

```
ans =
```

```
1.7711e+03
```

## Gaussian Mixture Models

### In this section...

“Introduction to Gaussian Mixture Models” on page 14-30

“Cluster with Gaussian Mixtures” on page 14-30

### Introduction to Gaussian Mixture Models

Gaussian mixture models are formed by combining multivariate normal density components.

In Statistics Toolbox software, use the `gmdistribution` class to fit data using an expectation maximization (EM) algorithm, which assigns posterior probabilities to each component density with respect to each observation.

Gaussian mixture models are often used for data clustering. Clusters are assigned by selecting the component that maximizes the posterior probability. Like k-means clustering, Gaussian mixture modeling uses an iterative algorithm that converges to a local optimum. Gaussian mixture modeling may be more appropriate than k-means clustering when clusters have different sizes and correlation within them. Clustering using Gaussian mixture models is sometimes considered a soft clustering method. The posterior probabilities for each point indicate that each data point has some probability of belonging to each cluster.

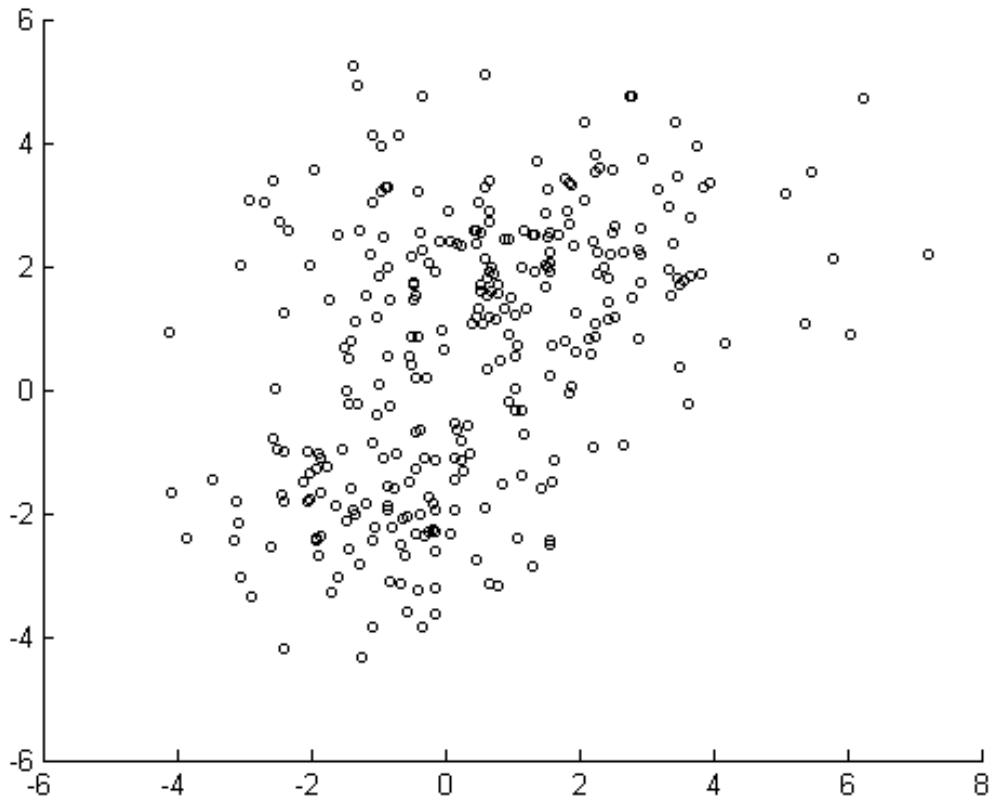
### Cluster with Gaussian Mixtures

Gaussian mixture distributions can be used for clustering data, by realizing that the multivariate normal components of the fitted model can represent clusters.

- 1 To demonstrate the process, first generate some simulated data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function.

```
rng('default') % For reproducibility
mu1 = [1 2];
sigma1 = [3 .2; .2 2];
mu2 = [-1 -2];
```

```
sigma2 = [2 0; 0 1];  
X = [mvnrnd(mu1,sigma1,200);mvnrnd(mu2,sigma2,100)];  
  
scatter(X(:,1),X(:,2),10,'ko')
```



- 2** Fit a two-component Gaussian mixture distribution. Here, you know the correct number of components to use. In practice, with real data, this decision would require comparing models with different numbers of components.

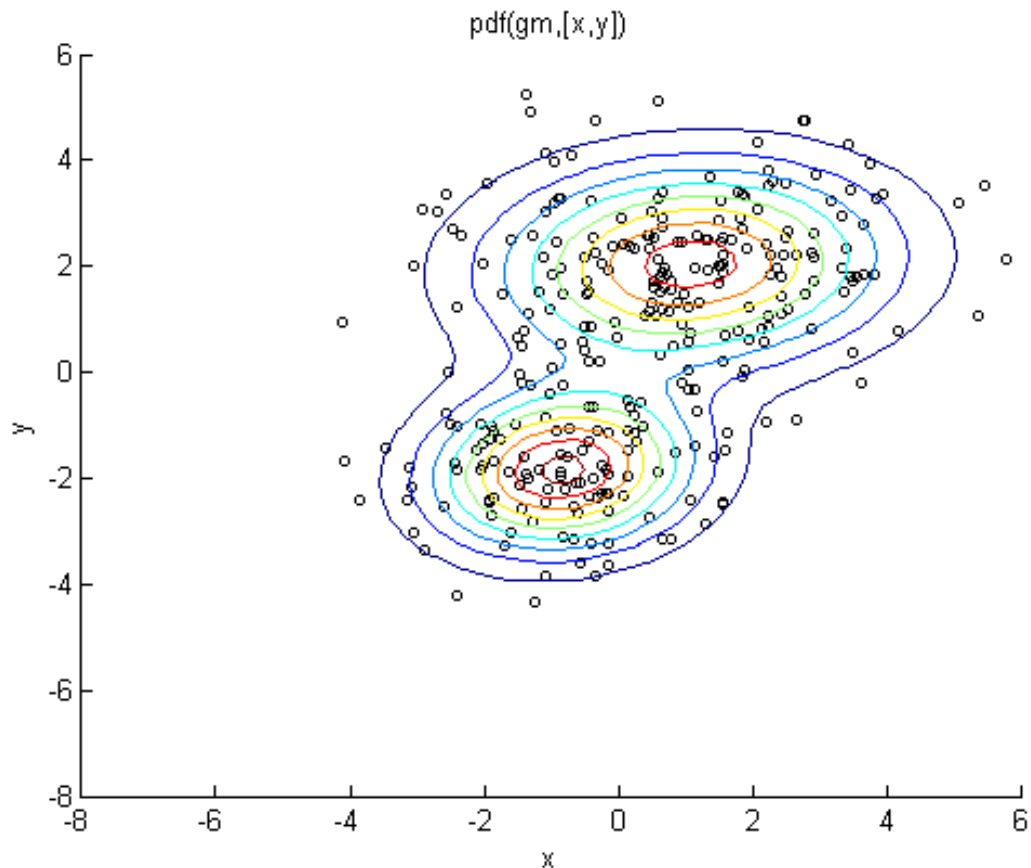
```
options = statset('Display','final');
```

```
gm = gmdistribution.fit(X,2,'Options',options);
```

```
33 iterations, log-likelihood = -1210.59
```

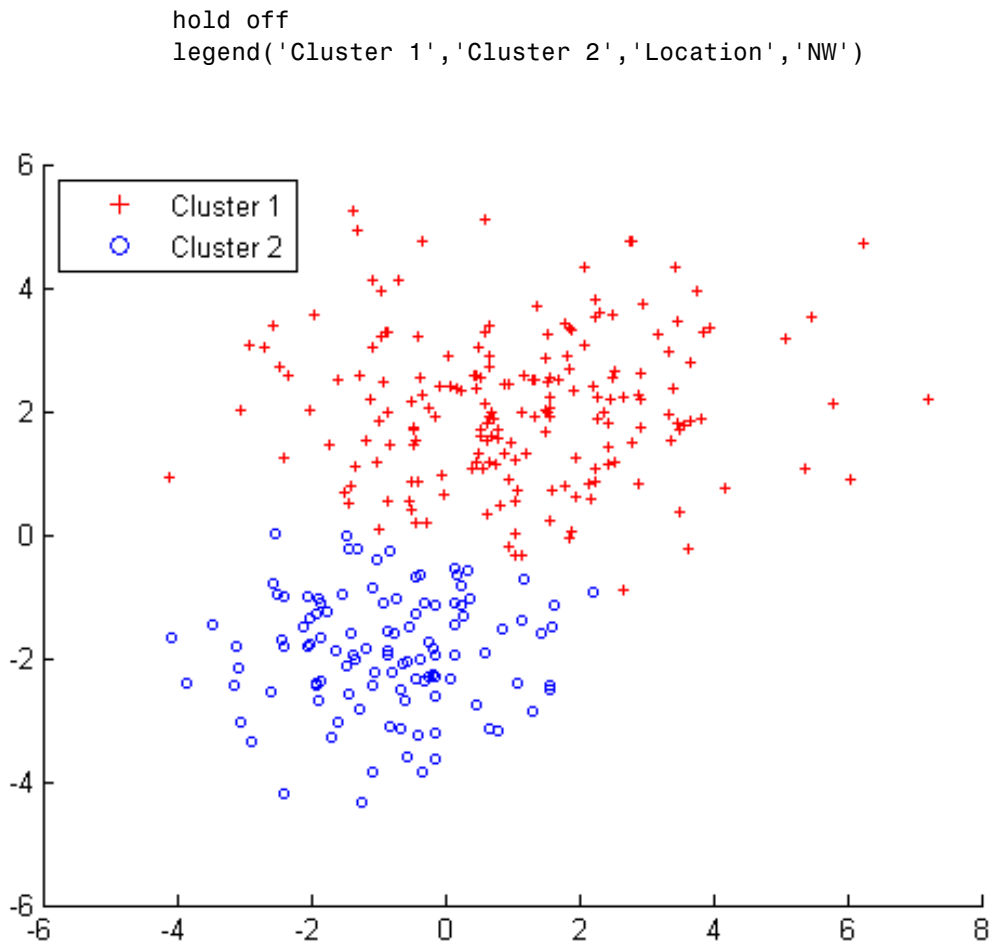
- 3** Plot the estimated probability density contours for the two-component mixture distribution. The two bivariate normal components overlap, but their peaks are distinct. This suggests that the data could reasonably be divided into two clusters.

```
hold on  
ezcontour(@(x,y)pdf(gm,[x y]),[-8 6],[-8 6]);  
hold off
```



- 4** Partition the data into clusters using the `cluster` method for the fitted mixture distribution. The `cluster` method assigns each point to one of the two components in the mixture distribution.

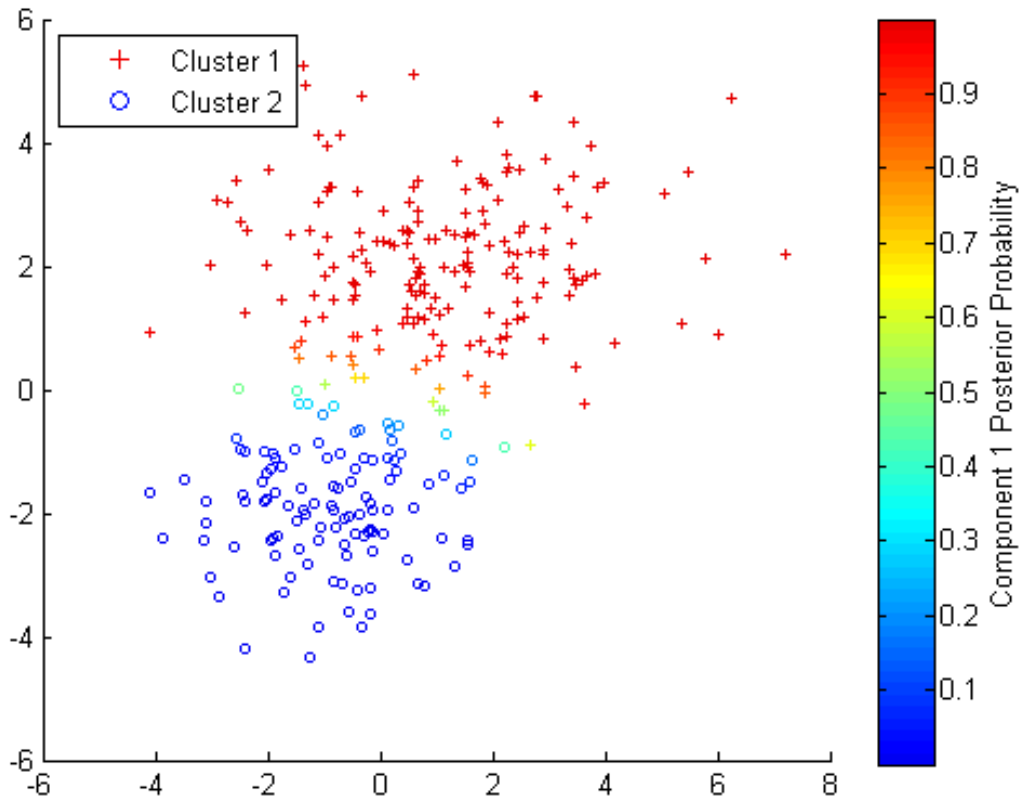
```
idx = cluster(gm,X);  
cluster1 = (idx == 1);  
cluster2 = (idx == 2);  
  
scatter(X(cluster1,1),X(cluster1,2),10,'r+');  
hold on  
scatter(X(cluster2,1),X(cluster2,2),10,'bo');
```



Each cluster corresponds to one of the bivariate normal components in the mixture distribution. `cluster` assigns points to clusters based on the estimated posterior probability that a point came from a component; each point is assigned to the cluster corresponding to the highest posterior probability. The posterior method returns those posterior probabilities. For example, plot the posterior probability of the first component for each point.



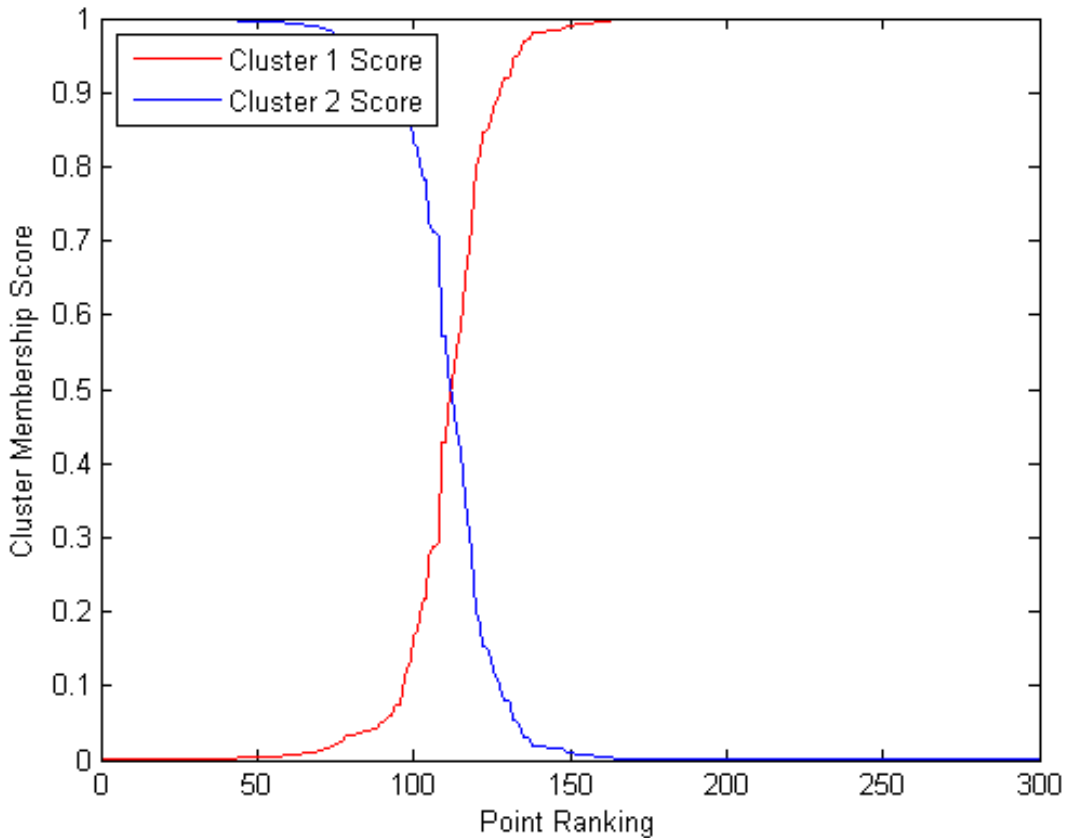
```
P = posterior(gm,X);  
  
scatter(X(cluster1,1),X(cluster1,2),10,P(cluster1,1),'+')  
hold on  
scatter(X(cluster2,1),X(cluster2,2),10,P(cluster2,1),'o')  
hold off  
legend('Cluster 1','Cluster 2','Location','NW')  
clrmmap = jet(80); colormap(clrmmap(9:72,:))  
ylabel(colorbar,'Component 1 Posterior Probability')
```



### **Soft Clustering Using Gaussian Mixture Distributions**

An alternative to the previous example is to use the posterior probabilities for "soft clustering". Each point is assigned a membership score to each cluster. Membership scores are simply the posterior probabilities, and describe how similar each point is to each cluster's archetype, i.e., the mean of the corresponding component. The points can be ranked by their membership score in a given cluster.

```
[~,order] = sort(P(:,1));  
plot(1:size(X,1),P(order,1),'r-',1:size(X,1),P(order,2),'b-');  
legend({'Cluster 1 Score' 'Cluster 2 Score'},'location','NW');  
ylabel('Cluster Membership Score');  
xlabel('Point Ranking');
```



Although a clear separation of the data is hard to see in a scatter plot of the data, plotting the membership scores indicates that the fitted distribution does a good job of separating the data into groups. Very few points have scores close to 0.5.

Soft clustering using a Gaussian mixture distribution is similar to fuzzy k-means clustering, which also assigns each point to each cluster with a membership score. The fuzzy k-means algorithm assumes that clusters are roughly spherical in shape, and all of roughly equal size. This is comparable to a Gaussian mixture distribution with a single covariance matrix that is

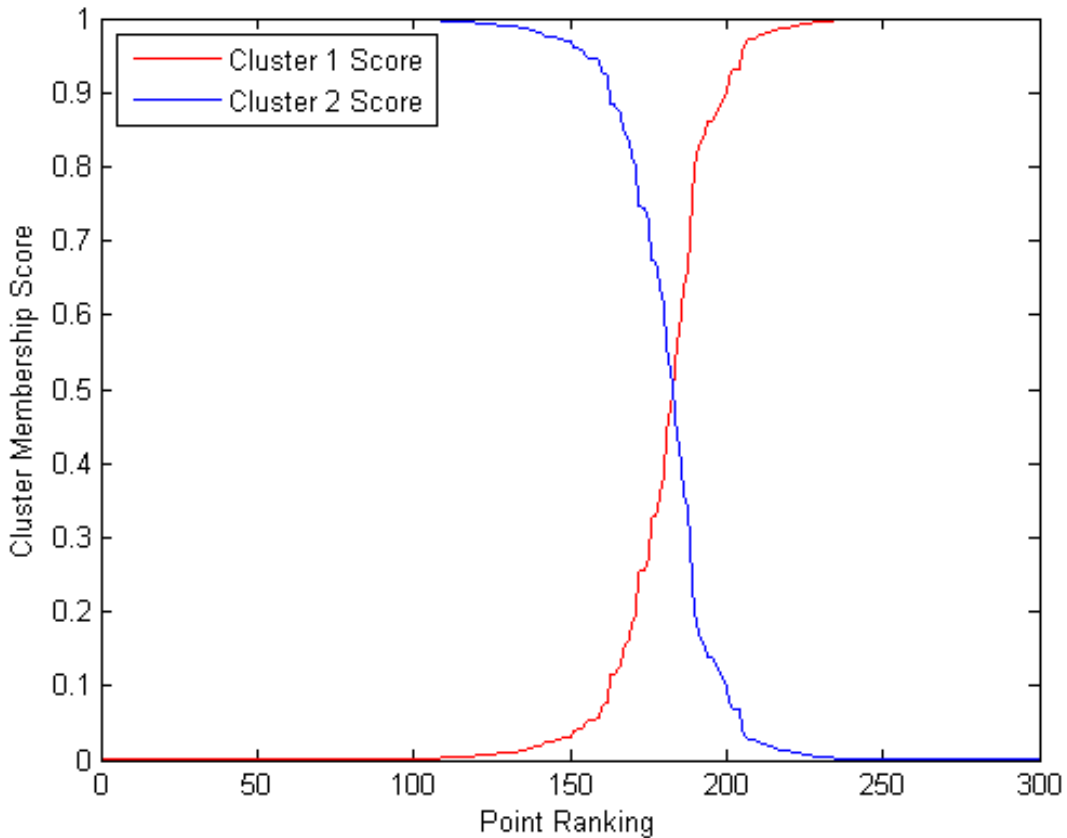
shared across all components, and is a multiple of the identity matrix. In contrast, `gmdistribution` allows you to specify different covariance options. The default is to estimate a separate, unconstrained covariance matrix for each component. A more restricted option, closer to k-means, would be to estimate a shared, diagonal covariance matrix.

```
gm2 = gmdistribution.fit(X,2,'CovType','Diagonal',...  
    'SharedCov',true);
```

This covariance option is similar to fuzzy k-means clustering, but provides more flexibility by allowing unequal variances for different variables.

You can compute the soft cluster membership scores without computing hard cluster assignments, using `posterior`, or as part of hard clustering, as the second output from `cluster`.

```
P2 = posterior(gm2,X); % equivalently [idx,P2] = cluster(gm2,X)  
[~,order] = sort(P2(:,1));  
plot(1:size(X,1),P2(order,1),'r-',1:size(X,1),P2(order,2),'b-');  
legend({'Cluster 1 Score' 'Cluster 2 Score'},'location','NW');  
ylabel('Cluster Membership Score');  
xlabel('Point Ranking');
```



### Assign New Data to Clusters

In the previous example, fitting the mixture distribution to data using `fit`, and clustering those data using `cluster`, are separate steps. However, the same data are used in both steps. You can also use the `cluster` method to assign new data points to the clusters (mixture components) found in the original data.

- 1 Given a data set  $X$ , first fit a Gaussian mixture distribution. The previous code has already done that.

```
gm
```

```
gm =
```

```
Gaussian mixture distribution with 2 components in 2 dimensions
```

```
Component 1:
```

```
Mixing proportion: 0.629379
```

```
Mean:    1.0758    2.0426
```

```
Component 2:
```

```
Mixing proportion: 0.370621
```

```
Mean:   -0.8292   -1.8482
```

- 2** You can then use `cluster` to assign each point in a new data set, `Y`, to one of the clusters defined for the original data.

```
Y = [mvnrnd(mu1,sigma1,50);mvnrnd(mu2,sigma2,25)];
```

```
idx = cluster(gm,Y);
```

```
cluster1 = (idx == 1);
```

```
cluster2 = (idx == 2);
```

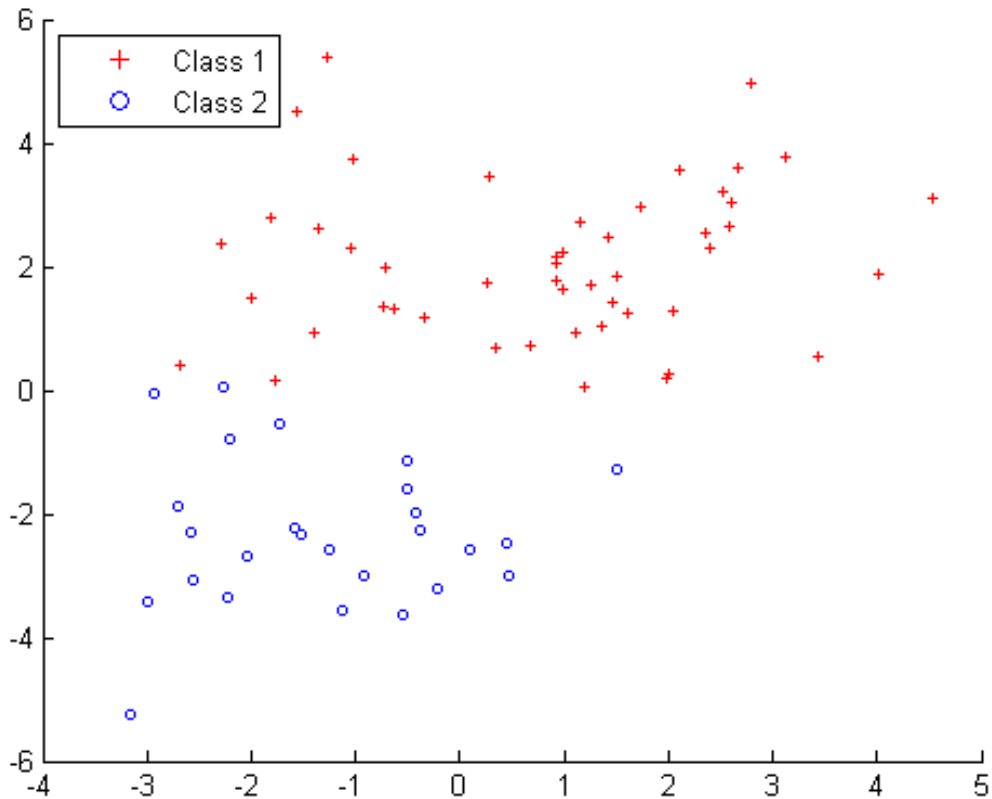
```
scatter(Y(cluster1,1),Y(cluster1,2),10,'r+');
```

```
hold on
```

```
scatter(Y(cluster2,1),Y(cluster2,2),10,'bo');
```

```
hold off
```

```
legend('Class 1','Class 2','Location','NW')
```



As with the previous example, the posterior probabilities for each point can be treated as membership scores rather than determining "hard" cluster assignments.

For cluster to provide meaningful results with new data,  $Y$  should come from the same population as  $X$ , the original data used to create the mixture distribution. In particular, the estimated mixing probabilities for the Gaussian mixture distribution fitted to  $X$  are used when computing the posterior probabilities for  $Y$ .





# Parametric Classification

---

- “Parametric Classification” on page 15-2
- “Discriminant Analysis” on page 15-3
- “Naive Bayes Classification” on page 15-38
- “Performance Curves” on page 15-41

## Parametric Classification

Models of data with a categorical response are called *classifiers*. A classifier is built from *training data*, for which classifications are known. The classifier assigns new *test data* to one of the categorical levels of the response.

Parametric methods, like “Discriminant Analysis” on page 15-3, fit a parametric model to the training data and interpolate to classify test data.

Nonparametric methods, like classification and regression trees, use other means to determine classifications.

## Discriminant Analysis

### In this section...

“What Is Discriminant Analysis?” on page 15-3

“Create Discriminant Analysis Classifiers” on page 15-3

“Creating a Classifier Using ClassificationDiscriminant.fit” on page 15-4

“How the predict Method Classifies” on page 15-6

“Create and Visualize Discriminant Analysis Classifier” on page 15-9

“Improve a Discriminant Analysis Classifier” on page 15-15

“Regularize a Discriminant Analysis Classifier” on page 15-23

“Examine the Gaussian Mixture Assumption” on page 15-29

“Bibliography” on page 15-37

### What Is Discriminant Analysis?

Discriminant analysis is a classification method. It assumes that different classes generate data based on different Gaussian distributions.

- To train (create) a classifier, the fitting function estimates the parameters of a Gaussian distribution for each class (see “Creating a Classifier Using ClassificationDiscriminant.fit” on page 15-4).
- To predict the classes of new data, the trained classifier finds the class with the smallest misclassification cost (see “How the predict Method Classifies” on page 15-6).

Linear discriminant analysis is also known as the Fisher discriminant, named for its inventor, Sir R. A. Fisher [2].

### Create Discriminant Analysis Classifiers

To create the basic types of discriminant analysis classifiers for the Fisher iris data:

- 1 Load the data:

```
load fisheriris;
```

- 2 Create a default (linear) discriminant analysis classifier:

```
linclass = ClassificationDiscriminant.fit(meas,species);
```

To visualize the classification boundaries of a 2-D linear classification of the data, see “Create and Visualize Discriminant Analysis Classifier” on page 15-9.

- 3 Classify an iris with average measurements:

```
meanmeas = mean(meas);  
meanclass = predict(linclass,meanmeas)
```

```
meanclass =  
    'versicolor'
```

- 4 Create a quadratic classifier:

```
quadclass = ClassificationDiscriminant.fit(meas,species,...  
    'discrimType','quadratic');
```

To visualize the classification boundaries of a 2-D quadratic classification of the data, see “Create and Visualize Discriminant Analysis Classifier” on page 15-9.

- 5 Classify an iris with average measurements using the quadratic classifier:

```
meanclass2 = predict(quadclass,meanmeas)
```

```
meanclass2 =  
    'versicolor'
```

## **Creating a Classifier Using ClassificationDiscriminant.fit**

The model for discriminant analysis is:

- Each class (Y) generates data (X) using a multivariate normal distribution. In other words, the model assumes X has a Gaussian mixture distribution (gmdistribution).

- For linear discriminant analysis, the model has the same covariance matrix for each class; only the means vary.
- For quadratic discriminant analysis, both means and covariances of each class vary.

Under this modeling assumption, `ClassificationDiscriminant.fit` infers the mean and covariance parameters of each class.

- For linear discriminant analysis, it computes the sample mean of each class. Then it computes the sample covariance by first subtracting the sample mean of each class from the observations of that class, and taking the empirical covariance matrix of the result.
- For quadratic discriminant analysis, it computes the sample mean of each class. Then it computes the sample covariances by first subtracting the sample mean of each class from the observations of that class, and taking the empirical covariance matrix of each class.

The `fit` method does not use prior probabilities or costs for fitting.

### Weighted Observations

The `fit` method constructs weighted classifiers using the following scheme. Suppose  $M$  is an  $N$ -by- $K$  class membership matrix:

$$\begin{aligned} M_{nk} &= 1 \text{ if observation } n \text{ is from class } k \\ M_{nk} &= 0 \text{ otherwise.} \end{aligned}$$

The estimate of the class mean for unweighted data is

$$\hat{\mu}_k = \frac{\sum_{n=1}^N M_{nk} x_n}{\sum_{n=1}^N M_{nk}}.$$

For weighted data with positive weights  $w_n$ , the natural generalization is

$$\hat{\mu}_k = \frac{\sum_{n=1}^N M_{nk} w_n x_n}{\sum_{n=1}^N M_{nk} w_n}.$$

The unbiased estimate of the pooled-in covariance matrix for unweighted data is

$$\hat{\Sigma} = \frac{\sum_{n=1}^N \sum_{k=1}^K M_{nk} (x_n - \hat{\mu}_k)(x_n - \hat{\mu}_k)^T}{N - K}.$$

For quadratic discriminant analysis, the `fit` method uses  $K = 1$ .

For weighted data, assuming the weights sum to 1, the unbiased estimate of the pooled-in covariance matrix is

$$\hat{\Sigma} = \frac{\sum_{n=1}^N \sum_{k=1}^K M_{nk} w_n (x_n - \hat{\mu}_k)(x_n - \hat{\mu}_k)^T}{1 - \sum_{k=1}^K \frac{W_k^{(2)}}{W_k}},$$

where

- $W_k = \sum_{n=1}^N M_{nk} w_n$  is the sum of the weights for class  $k$ .
- $W_k^{(2)} = \sum_{n=1}^N M_{nk} w_n^2$  is the sum of squared weights per class.

## How the predict Method Classifies

There are three elements in the `predict` classification algorithm:

- “Posterior Probability” on page 15-7
- “Prior Probability” on page 15-8
- “Cost” on page 15-8

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

The space of  $X$  values divides into regions where a classification  $Y$  is a particular value. The regions are separated by straight lines for linear discriminant analysis, and by conic sections (ellipses, hyperbolas, or parabolas) for quadratic discriminant analysis. For a visualization of these regions, see “Create and Visualize Discriminant Analysis Classifier” on page 15-9.

### Posterior Probability

The posterior probability that a point  $x$  belongs to class  $k$  is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_k$  and covariance  $\Sigma_k$  at a point  $x$  is

$$P(x | k) = \frac{1}{(2\pi |\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k | x) = \frac{P(x | k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, namely, the sum over  $k$  of  $P(x|k)P(k)$ .

### Prior Probability

The prior probability is one of three choices:

- 'uniform' — The prior probability of class  $k$  is 1 over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.
- A numeric vector — The prior probability of class  $k$  is the  $j$ th element of the prior vector. See `ClassificationDiscriminant.fit`.

After creating a classifier `obj`, you can set the prior using dot addressing:

```
obj.Prior = v;
```

where  $v$  is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

### Cost

There are two costs associated with discriminant analysis classification: the true misclassification cost per class, and the expected misclassification cost per observation.

**True Misclassification Cost per Class.**  $\text{Cost}(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $\text{Cost}(i, j)=1$  if  $i \neq j$ , and  $\text{Cost}(i, j)=0$  if  $i=j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

You can set any cost matrix you like when creating a classifier. Pass the cost matrix in the `Cost` name-value pair in `ClassificationDiscriminant.fit`.

After you create a classifier `obj`, you can set a custom cost using dot addressing:

```
obj.Cost = B;
```



$B$  is a square matrix of size  $K$ -by- $K$  when there are  $K$  classes. You do not need to retrain the classifier when you set a new cost.

**Expected Misclassification Cost per Observation.** Suppose you have  $N_{\text{obs}}$  observations that you want to classify with a trained discriminant analysis classifier `obj`. Suppose you have  $K$  classes. You place the observations into a matrix  $X_{\text{new}}$  with one observation per row. The command

```
[label,score,cost] = predict(obj,Xnew)
```

returns, among other outputs, a cost matrix of size  $N_{\text{obs}}$ -by- $K$ . Each row of the cost matrix contains the expected (average) cost of classifying the observation into each of the  $K$  classes. `cost(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{\text{new}}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{\text{new}}(n))$  is the posterior probability of class  $i$  for observation  $X_{\text{new}}(n)$ .
- $C(k | i)$  is the cost of classifying an observation as  $k$  when its true class is  $i$ .

## Create and Visualize Discriminant Analysis Classifier

This example shows how to perform linear and quadratic classification of Fisher iris data.

Load the sample data.

```
load fisheriris
```

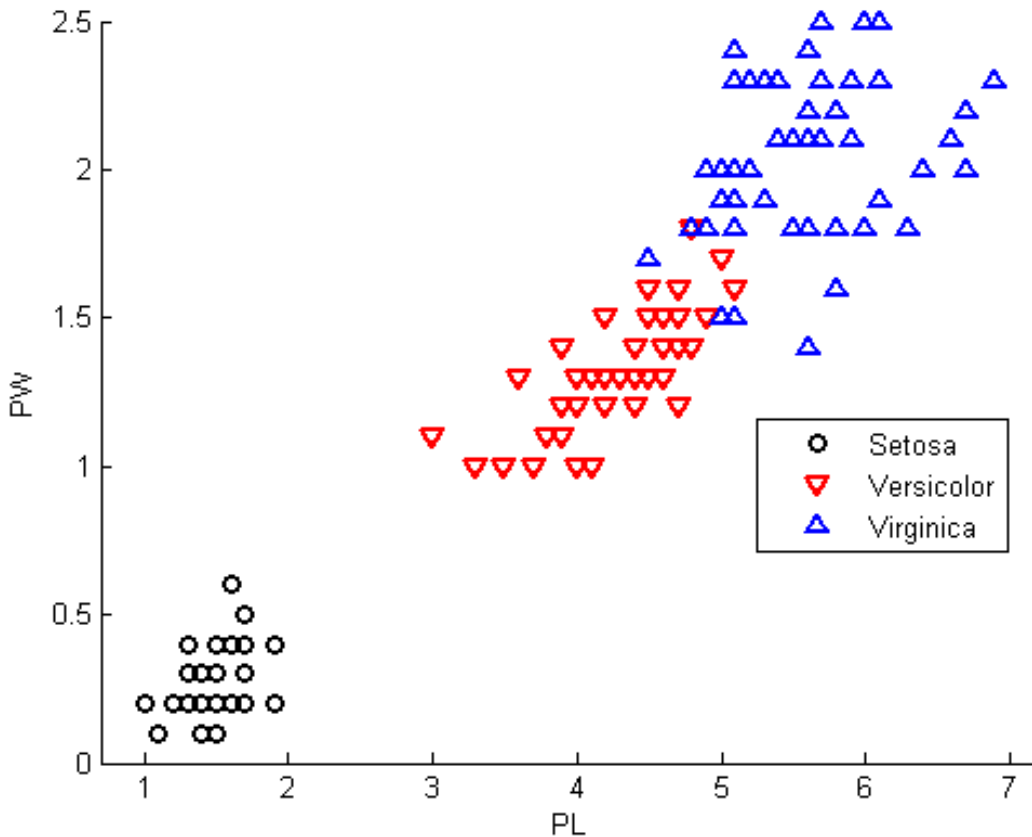
The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Use petal length (third column in `meas`) and petal width (fourth column in `meas`) measurements. Save these as variables `PL` and `PW`, respectively.

```
PL = meas(:,3);  
PW = meas(:,4);
```

Plot the data, showing the classification, that is, create a scatter plot of the measurements, grouped by species.

```
h1 = gscatter(PL,PW,species,'krb','ov^',[],'off');  
set(h1,'LineWidth',2)  
legend('Setosa','Versicolor','Virginica','Location','best')  
hold on
```



Create a linear classifier.

```
X = [PL,PW];
cls = ClassificationDiscriminant.fit(X,species);
```

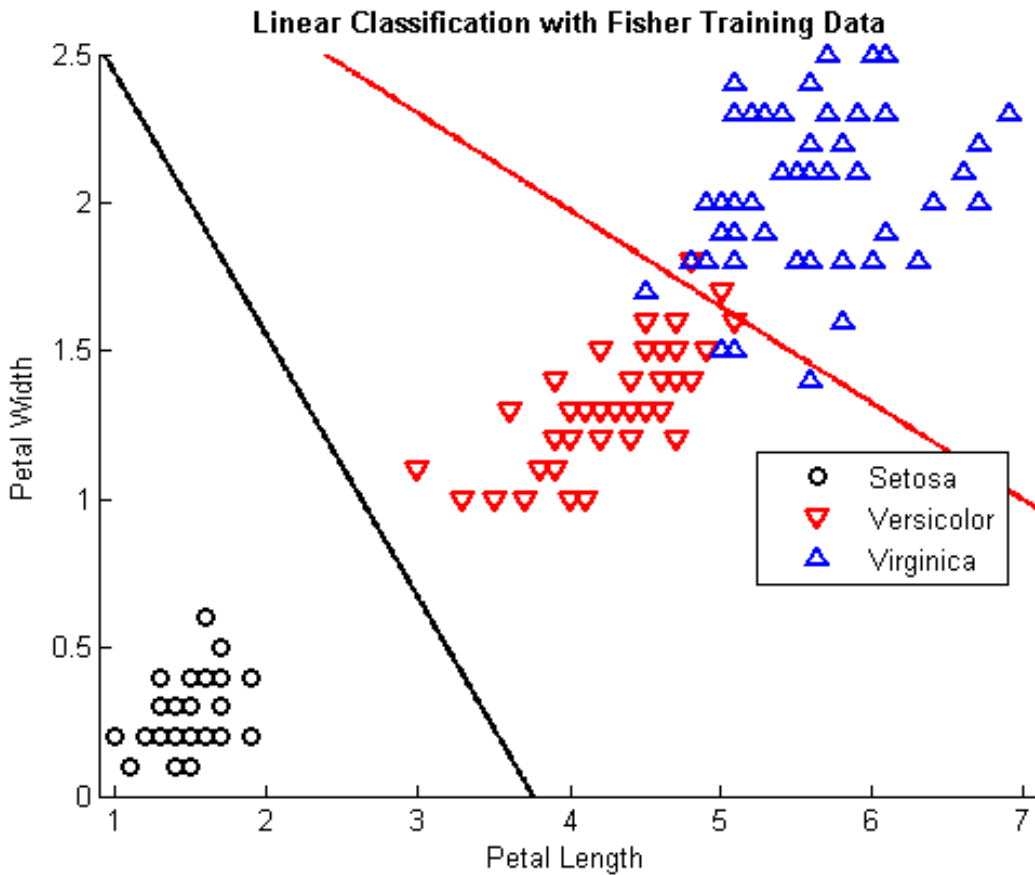
Plot the classification boundaries.

```
K = cls.Coeffs(2,3).Const; % First retrieve the coefficients for the linear
L = cls.Coeffs(2,3).Linear;% boundary between the second and third classes
% (versicolor and virginica).
```

```
% Plot the curve  $K + [x,y]*L = 0$ .
f = @(x1,x2) K + L(1)*x1 + L(2)*x2;
h2 = ezplot(f,[.9 7.1 0 2.5]);
set(h2,'Color','r','LineWidth',2)

% Now, retrieve the coefficients for the linear boundary between the first
% and second classes (setosa and versicolor).
K = cls.Coeffs(1,2).Const;
L = cls.Coeffs(1,2).Linear;

% Plot the curve  $K + [x1,x2]*L = 0$ :
f = @(x1,x2) K + L(1)*x1 + L(2)*x2;
h3 = ezplot(f,[.9 7.1 0 2.5]);
set(h3,'Color','k','LineWidth',2)
axis([.9 7.1 0 2.5])
xlabel('Petal Length')
ylabel('Petal Width')
title('\bf Linear Classification with Fisher Training Data')
```



Create a quadratic discriminant classifier.

```
cqs = ClassificationDiscriminant.fit(X,species,...
    'DiscrimType','quadratic');
```

Plot the classification boundaries similarly.

```
delete(h2); delete(h3) % First, remove the linear boundaries from the plot.
```

```
% Now, retrieve the coefficients for the quadratic boundary between the
% second and third classes (versicolor and virginica).
```

```

K = cqs.Coeffs(2,3).Const;
L = cqs.Coeffs(2,3).Linear;
Q = cqs.Coeffs(2,3).Quadratic;

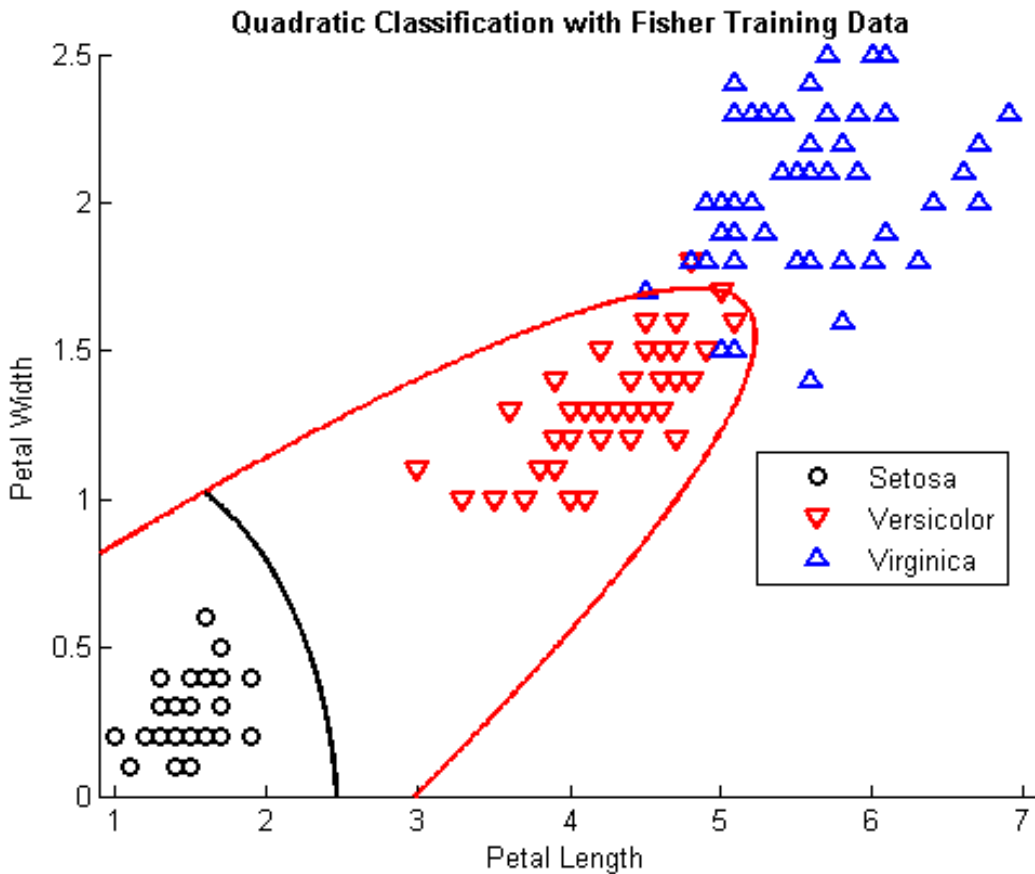
% Plot the curve  $K + [x_1, x_2]*L + [x_1, x_2]*Q*[x_1, x_2]' = 0$ .
f = @(x1,x2) K + L(1)*x1 + L(2)*x2 + Q(1,1)*x1.^2 + ...
    (Q(1,2)+Q(2,1))*x1.*x2 + Q(2,2)*x2.^2;
h2 = ezplot(f,[.9 7.1 0 2.5]);
set(h2,'Color','r','LineWidth',2)

% Now, retrieve the coefficients for the quadratic boundary between the
% first and second classes (setosa and versicolor).
K = cqs.Coeffs(1,2).Const;
L = cqs.Coeffs(1,2).Linear;
Q = cqs.Coeffs(1,2).Quadratic;

% Plot the curve  $K + [x_1, x_2]*L + [x_1, x_2]*Q*[x_1, x_2]' = 0$ :
f = @(x1,x2) K + L(1)*x1 + L(2)*x2 + Q(1,1)*x1.^2 + ...
    (Q(1,2)+Q(2,1))*x1.*x2 + Q(2,2)*x2.^2;
h3 = ezplot(f,[.9 7.1 0 1.02]); % Plot the relevant portion of the curve.

set(h3,'Color','k','LineWidth',2)
axis([.9 7.1 0 2.5])
xlabel('Petal Length')
ylabel('Petal Width')
title('{\bf Quadratic Classification with Fisher Training Data}')
hold off

```



## Improve a Discriminant Analysis Classifier

- “Deal with Singular Data” on page 15-16
- “Choose a Discriminant Type” on page 15-17
- “Examine the Resubstitution Error and Confusion Matrix” on page 15-18
- “Cross Validation” on page 15-19
- “Change Costs and Priors” on page 15-21

## Deal with Singular Data

Discriminant analysis needs data sufficient to fit Gaussian models with invertible covariance matrices. If your data is not sufficient to fit such a model uniquely, `ClassificationDiscriminant.fit` fails. This section shows methods for handling failures.

---

**Tip** To obtain a discriminant analysis classifier without failure, set the `DiscrimType` name-value pair to `'pseudoLinear'` or `'pseudoQuadratic'` in `ClassificationDiscriminant.fit`.

“Pseudo” discriminants never fail, because they use the pseudoinverse of the covariance matrix  $\Sigma_k$  (see `pinv`).

---

**Example: Singular Covariance Matrix.** When the covariance matrix of the fitted classifier is singular, `ClassificationDiscriminant.fit` can fail:

```
load popcorn
X = popcorn(:,[1 2]);
X(:,3) = 0; % a zero-variance column
Y = popcorn(:,3);
ppcrn = ClassificationDiscriminant.fit(X,Y);

Error using ClassificationDiscriminant (line 635)
Predictor x3 has zero variance. Either exclude this predictor or set 'discrimType' to
'pseudoLinear' or 'diagLinear'.

Error in classreg.learning.FitTemplate/fit (line 243)
    obj = this.MakeFitObject(X,Y,W,this.ModelParams,fitArgs{:});

Error in ClassificationDiscriminant.fit (line 296)
    this = fit(temp,X,Y);
```

To proceed with linear discriminant analysis, use a `pseudoLinear` or `diagLinear` discriminant type:

```
ppcrn = ClassificationDiscriminant.fit(X,Y,...
    'discrimType','pseudoLinear');
meanpredict = predict(ppcrn,mean(X))
```



```
meanpredict =  
    3.5000
```

## Choose a Discriminant Type

There are six types of discriminant analysis classifiers: linear and quadratic, with *diagonal* and *pseudo* variants of each type.

---

**Tip** To see if your covariance matrix is singular, set `discrimType` to 'linear' or 'quadratic'. If the matrix is singular, the `ClassificationDiscriminant.fit` method fails for 'quadratic', and the `Gamma` property is nonzero for 'linear'.

To obtain a quadratic classifier even when your covariance matrix is singular, set `discrimType` to 'pseudoQuadratic' or 'diagQuadratic'.

```
obj = ClassificationDiscriminant.fit(X,Y,...  
    'discrimType','pseudoQuadratic') % or 'diagQuadratic'
```

---

Choose a classifier type by setting the `discrimType` name-value pair to one of:

- 'linear' (default) — Estimate one covariance matrix for all classes.
- 'quadratic' — Estimate one covariance matrix for each class.
- 'diagLinear' — Use the diagonal of the 'linear' covariance matrix, and use its pseudoinverse if necessary.
- 'diagQuadratic' — Use the diagonals of the 'quadratic' covariance matrices, and use their pseudoinverses if necessary.
- 'pseudoLinear' — Use the pseudoinverse of the 'linear' covariance matrix if necessary.
- 'pseudoQuadratic' — Use the pseudoinverses of the 'quadratic' covariance matrices if necessary.

`ClassificationDiscriminant.fit` can fail for the 'linear' and 'quadratic' classifiers. When it fails, it returns an explanation, as shown in “Deal with Singular Data” on page 15-16.

`ClassificationDiscriminant.fit` always succeeds with the diagonal and pseudo variants. For information about pseudoinverses, see `pinv`.

You can set the discriminant type using dot addressing after constructing a classifier:

```
obj.DiscrimType = 'discrimType'
```

You can change between linear types or between quadratic types, but cannot change between a linear and a quadratic type.

### **Examine the Resubstitution Error and Confusion Matrix**

The *resubstitution error* is the difference between the response training data and the predictions the classifier makes of the response based on the input training data. If the resubstitution error is high, you cannot expect the predictions of the classifier to be good. However, having low resubstitution error does not guarantee good predictions for new data. Resubstitution error is often an overly optimistic estimate of the predictive error on new data.

The *confusion matrix* shows how many errors, and which types, arise in resubstitution. When there are  $K$  classes, the confusion matrix  $R$  is a  $K$ -by- $K$  matrix with

$R(i, j)$  = the number of observations of class  $i$  that the classifier predicts to be of class  $j$ .

#### **Example: Resubstitution Error of a Discriminant Analysis Classifier.**

Examine the resubstitution error of the default discriminant analysis classifier for the Fisher iris data:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
resuberror = resubLoss(obj)

resuberror =
    0.0200
```

The resubstitution error is very low, meaning `obj` classifies nearly all the Fisher iris data correctly. The total number of misclassifications is:

```
resuberror * obj.NObservations  
  
ans =  
    3.0000
```

To see the details of the three misclassifications, examine the confusion matrix:

```
R = confusionmat(obj.Y,resubPredict(obj))  
  
R =  
    50     0     0  
     0    48     2  
     0     1    49
```

```
obj.ClassNames
```

```
ans =  
    'setosa'  
    'versicolor'  
    'virginica'
```

- $R(1,:) = [50 \ 0 \ 0]$  means `obj` classifies all 50 setosa irises correctly.
- $R(2,:) = [0 \ 48 \ 2]$  means `obj` classifies 48 versicolor irises correctly, and misclassifies two versicolor irises as virginica.
- $R(3,:) = [0 \ 1 \ 49]$  means `obj` classifies 49 virginica irises correctly, and misclassifies one virginica iris as versicolor.

## Cross Validation

Typically, discriminant analysis classifiers are robust and do not exhibit overtraining when the number of predictors is much less than the number of observations. Nevertheless, it is good practice to cross validate your classifier to ensure its stability.

### **Cross Validating a Discriminant Analysis Classifier.**

This example shows how to perform five-fold cross validation of a quadratic discriminant analysis classifier.

Load the sample data.

```
load fisheriris
```

Create a quadratic discriminant analysis classifier for the data.

```
quadisc = ClassificationDiscriminant.fit(meas,species,'DiscrimType','quadra
```

Find the resubstitution error of the classifier.

```
qerror = resubLoss(quadisc)
```

```
qerror =
```

```
    0.0200
```

The classifier does an excellent job. Nevertheless, resubstitution error can be an optimistic estimate of the error when classifying new data. So proceed to cross validation.

Create a cross-validation model.

```
cvmodel = crossval(quadisc,'kfold',5);
```

Find the cross-validation loss for the model, meaning the error of the out-of-fold observations.

```
cverror = kfoldLoss(cvmodel)
```

```
cverror =
```

```
    0.0200
```

The cross-validated loss is as low as the original resubstitution loss. Therefore, you can have confidence that the classifier is reasonably accurate.

### Change Costs and Priors

Sometimes you want to avoid certain misclassification errors more than others. For example, it might be better to have oversensitive cancer detection instead of undersensitive cancer detection. Oversensitive detection gives more false positives (unnecessary testing or treatment). Undersensitive detection gives more false negatives (preventable illnesses or deaths). The consequences of underdetection can be high. Therefore, you might want to set costs to reflect the consequences.

Similarly, the training data  $Y$  can have a distribution of classes that does not represent their true frequency. If you have a better estimate of the true frequency, you can include this knowledge in the classification prior property.

**Example: Setting Custom Misclassification Costs.** Consider the Fisher iris data. Suppose that the cost of classifying a versicolor iris as virginica is 10 times as large as making any other classification error. Create a classifier from the data, then incorporate this cost and then view the resulting classifier.

- 1 Load the Fisher iris data and create a default (linear) classifier as in “Example: Resubstitution Error of a Discriminant Analysis Classifier” on page 15-18:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
resuberror = resubLoss(obj)

resuberror =
    0.0200

R = confusionmat(obj.Y,resubPredict(obj))

R =
    50     0     0
     0    48     2
     0     1    49
```

```
obj.ClassNames
```

```
ans =  
    'setosa'  
    'versicolor'  
    'virginica'
```

`R(2,:) = [0 48 2]` means `obj` classifies 48 versicolor irises correctly, and misclassifies two versicolor irises as virginica.

- 2 Change the cost matrix to make fewer mistakes in classifying versicolor irises as virginica:

```
obj.Cost(2,3) = 10;  
R2 = confusionmat(obj.Y,resubPredict(obj))
```

```
R2 =  
    50     0     0  
     0    50     0  
     0     7    43
```

`obj` now classifies all versicolor irises correctly, at the expense of increasing the number of misclassifications of virginica irises from 1 to 7.

**Example: Setting Alternative Priors.** Consider the Fisher iris data. There are 50 irises of each kind in the data. Suppose that, in a particular region, you have historical data that shows virginica are five times as prevalent as the other kinds. Create a classifier that incorporates this information.

- 1 Load the Fisher iris data and make a default (linear) classifier as in “Example: Resubstitution Error of a Discriminant Analysis Classifier” on page 15-18:

```
load fisheriris  
obj = ClassificationDiscriminant.fit(meas,species);  
resuberror = resubLoss(obj)  
  
resuberror =  
    0.0200  
  
R = confusionmat(obj.Y,resubPredict(obj))
```

```
R =
  50    0    0
   0   48    2
   0    1   49
```

```
obj.ClassNames
```

```
ans =
  'setosa'
  'versicolor'
  'virginica'
```

`R(3,:) = [0 1 49]` means `obj` classifies 49 virginica irises correctly, and misclassifies one virginica iris as versicolor.

- 2 Change the prior to match your historical data, and examine the confusion matrix of the new classifier:

```
obj.Prior = [1 1 5];
R2 = confusionmat(obj.Y, resubPredict(obj))
```

```
R2 =
  50    0    0
   0   46    4
   0    0   50
```

The new classifier classifies all virginica irises correctly, at the expense of increasing the number of misclassifications of versicolor irises from 2 to 4.

## Regularize a Discriminant Analysis Classifier

This example shows how to make a more robust and simpler model by trying to remove predictors without hurting the model's predictive power. This is especially important when you have many predictors in your data. Linear discriminant analysis uses the two regularization parameters, `Gamma` and `Delta`, to identify and remove redundant predictors. The `cvshrink` method helps you identify appropriate settings for these parameters.

1. Load data and create a classifier.

Create a linear discriminant analysis classifier for the ovariancancer data. Set the `SaveMemory` and `FillCoeffs` options to keep the resulting model reasonably small.

```
load ovariancancer
obj = ClassificationDiscriminant.fit(obs,grp,...
    'SaveMemory','on','FillCoeffs','off');
```

### **2. Cross validate the classifier.**

Use 30 levels of `Gamma` and 30 levels of `Delta` to search for good parameters. This search is time consuming. Set `Verbose` to 1 to view the progress.

```
rng(8000,'twister') % for reproducibility
[err,gamma,delta,numpred] = cvshrink(obj,...
    'NumGamma',29,'NumDelta',29,'Verbose',1);
```

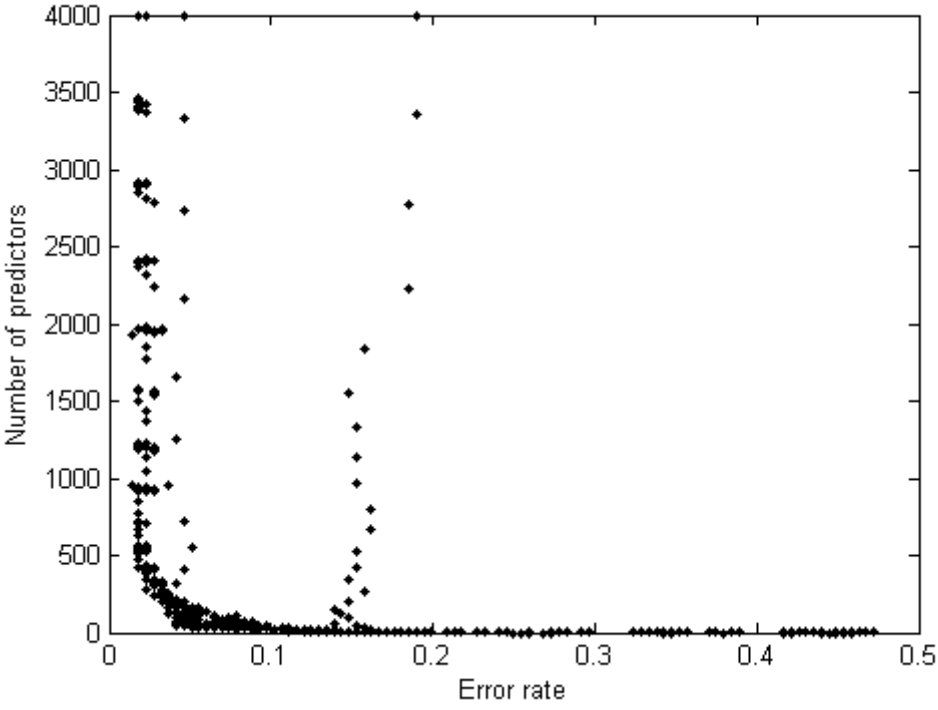
```
Done building cross-validated model.
Processing Gamma step 1 out of 30.
Processing Gamma step 2 out of 30.
Processing Gamma step 3 out of 30.
%%% (many lines removed) %%%
Processing Gamma step 28 out of 30.
Processing Gamma step 29 out of 30.
Processing Gamma step 30 out of 30.
```

### **3. Examine the quality of the regularized classifiers.**

Plot the number of predictors against the error.

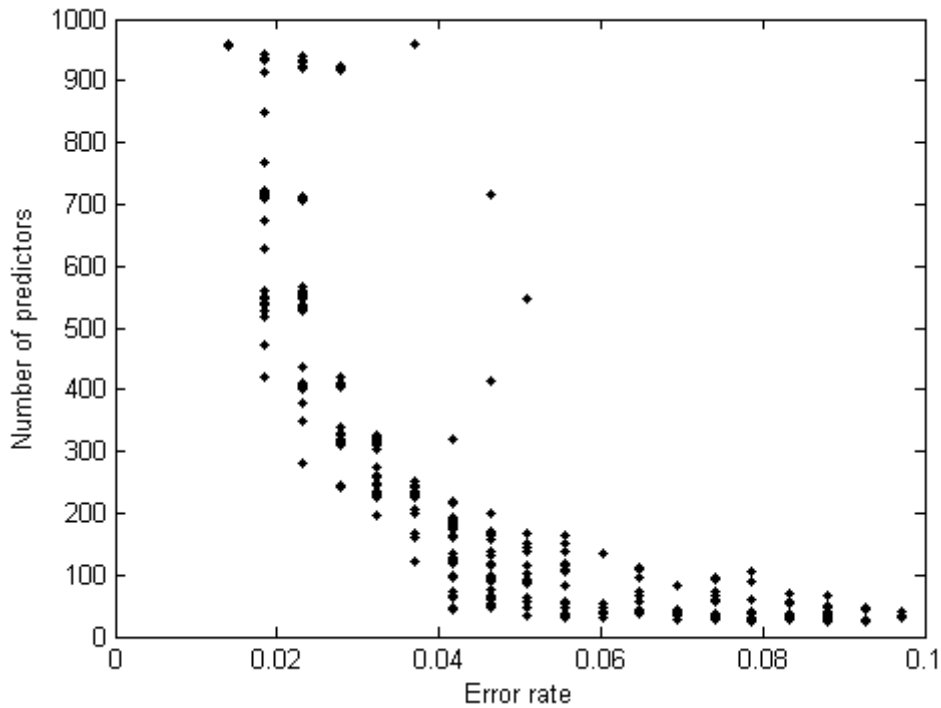
```
figure;
plot(err,numpred,'k.')
xlabel('Error rate');
ylabel('Number of predictors');
```





Examine the lower-left part of the plot more closely.

```
axis([0 .1 0 1000])
```



There is a clear tradeoff between lower number of predictors and lower error.

**4. Choose an optimal tradeoff between model size and accuracy.**

Find the values of Gamma and Delta that give minimal error.

```
minerr = min(min(err));
[p,q] = find(err == minerr)
```

```
p =
    24
    25
```

```
q =
     8
     8
```

Two points have the same minimal error: [24,8] and [25,8], which correspond to

```
[gamma(p(1)),delta(p(1),q(1)); gamma(p(2)),delta(p(2),q(2))]
```

```
ans =
    0.8436    0.1463
    0.8697    0.1425
```

These points correspond to about a quarter of the total predictors having nonzero coefficients in the model.

```
[numpred(p(1),q(1)); numpred(p(2),q(2))]
```

```
ans =
    957
    960
```

To further lower the number of predictors, you must accept larger error rates. For example, to choose the Gamma and Delta that give the lowest error rate with 250 or fewer predictors:

```
low250 = min(min(err(numpred <= 250)));
lownum = min(min(numpred(err == low250)));
[low250 lownum]
```

```
ans =

    [0.0278 243]
```

You need 243 predictors to achieve an error rate of 0.0278, and this is the lowest error rate among those that have 250 predictors or fewer.

The Gamma and Delta that achieve this error/number of predictors:

```
[r,s] = find((err == low250) & (numpred == lownum));
[gamma(r); delta(r,s)]
```

```
ans =

    0.7133
    0.2960
```

**5. Set the regularization parameters.**

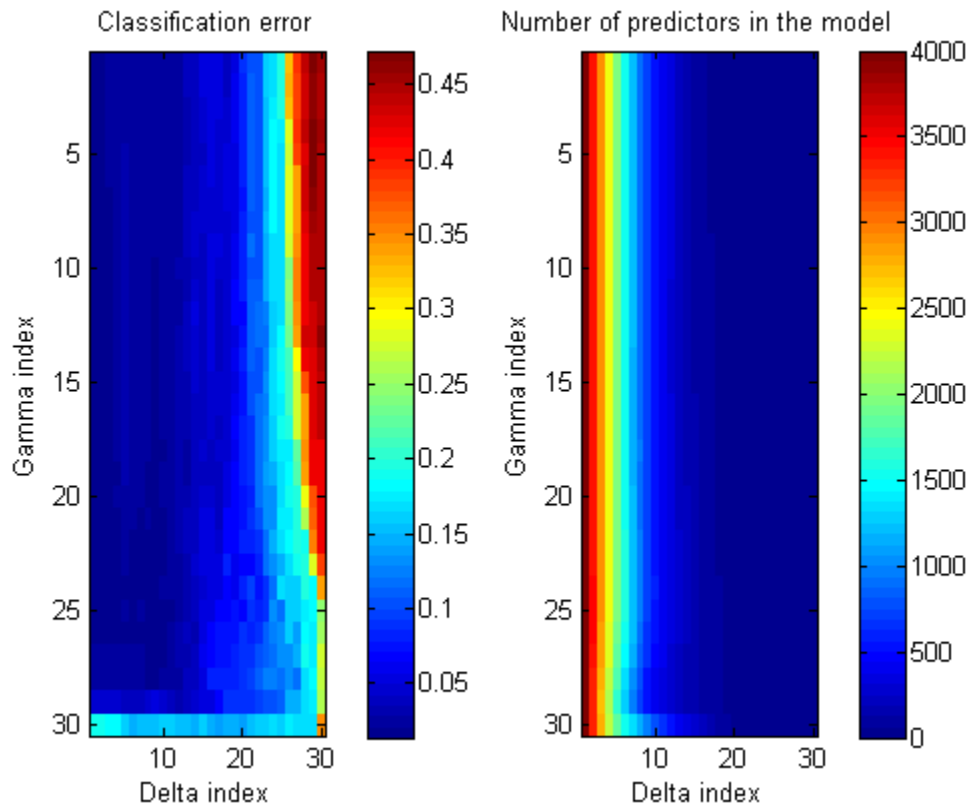
To set the classifier with these values of Gamma and Delta, use dot addressing.

```
obj.Gamma = gamma(r);  
obj.Delta = delta(r,s);
```

**6. Heat map plot.**

To compare the `cvshrink` calculation to that in Guo, Hastie, and Tibshirani [3], plot heat maps of error and number of predictors against Gamma and the index of the Delta parameter. (The Delta parameter range depends on the value of the Gamma parameter. So to get a rectangular plot, use the Delta index, not the parameter itself.)

```
% First create the Delta index matrix  
indx = repmat(1:size(delta,2),size(delta,1),1);  
figure  
subplot(1,2,1)  
imagesc(err);  
colorbar;  
title('Classification error');  
xlabel('Delta index');  
ylabel('Gamma index');  
  
subplot(1,2,2)  
imagesc(numpred);  
colorbar;  
title('Number of predictors in the model');  
xlabel('Delta index');  
ylabel('Gamma index');
```



You see the best classification error when Delta is small, but fewest predictors when Delta is large.

## Examine the Gaussian Mixture Assumption

Discriminant analysis assumes that the data comes from a Gaussian mixture model (see “Creating a Classifier Using ClassificationDiscriminant.fit” on page 15-4). If the data appears to come from a Gaussian mixture model, you can expect discriminant analysis to be a good classifier. Furthermore, the default linear discriminant analysis assumes that all class covariance matrices are equal. This section shows methods to check these assumptions:

- “Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 15-30
- “Q-Q Plot” on page 15-31
- “Mardia Kurtosis Test of Multivariate Normality” on page 15-35

### **Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis**

The Bartlett test (see Box [1]) checks equality of the covariance matrices of the various classes. If the covariance matrices are equal, the test indicates that linear discriminant analysis is appropriate. If not, consider using quadratic discriminant analysis, setting the `DiscrimType` name-value pair to 'quadratic' in `ClassificationDiscriminant.fit`.

The Bartlett test assumes normal (Gaussian) samples, where neither the means nor covariance matrices are known. To determine whether the covariances are equal, compute the following quantities:

- Sample covariance matrices per class  $\sigma_i$ ,  $1 \leq i \leq k$ , where  $k$  is the number of classes.
- Pooled-in covariance matrix  $\sigma$ .
- Test statistic  $V$ :

$$V = (n - k) \log(|\Sigma|) - \sum_{i=1}^k (n_i - 1) \log(|\Sigma_i|)$$

where  $n$  is the total number of observations, and  $n_i$  is the number of observations in class  $i$ , and  $|\Sigma|$  means the determinant of the matrix  $\Sigma$ .

- Asymptotically, as the number of observations in each class  $n_i$  become large,  $V$  is distributed approximately  $\chi^2$  with  $kd(d + 1)/2$  degrees of freedom, where  $d$  is the number of predictors (number of dimensions in the data).

The Bartlett test is to check whether  $V$  exceeds a given percentile of the  $\chi^2$  distribution with  $kd(d + 1)/2$  degrees of freedom. If it does, then reject the hypothesis that the covariances are equal.

**Example: Bartlett Test for Equal Covariance Matrices.** Check whether the Fisher iris data is well modeled by a single Gaussian covariance, or whether it would be better to model it as a Gaussian mixture.

```
load fisheriris;
prednames = {'SepalLength','SepalWidth','PetalLength','PetalWidth'};
L = ClassificationDiscriminant.fit(meas,species,'PredictorNames',prednames)
Q = ClassificationDiscriminant.fit(meas,species,'PredictorNames',prednames,
D = 4; % Number of dimensions of X
Nclass = [50 50 50];
N = L.NObservations;
K = numel(L.ClassNames);
SigmaQ = Q.Sigma;
SigmaL = L.Sigma;
logV = (N-K)*log(det(SigmaL));
for k=1:K
    logV = logV - (Nclass(k)-1)*log(det(SigmaQ(:,:,k)));
end
nu = (K-1)*D*(D+1)/2;
pval = 1 - chi2cdf(logV,nu)

pval =

    0
```

The Bartlett test emphatically rejects the hypothesis of equal covariance matrices. If `pval` had been greater than 0.05, the test would not have rejected the hypothesis. The result indicates to use quadratic discriminant analysis, as opposed to linear discriminant analysis.

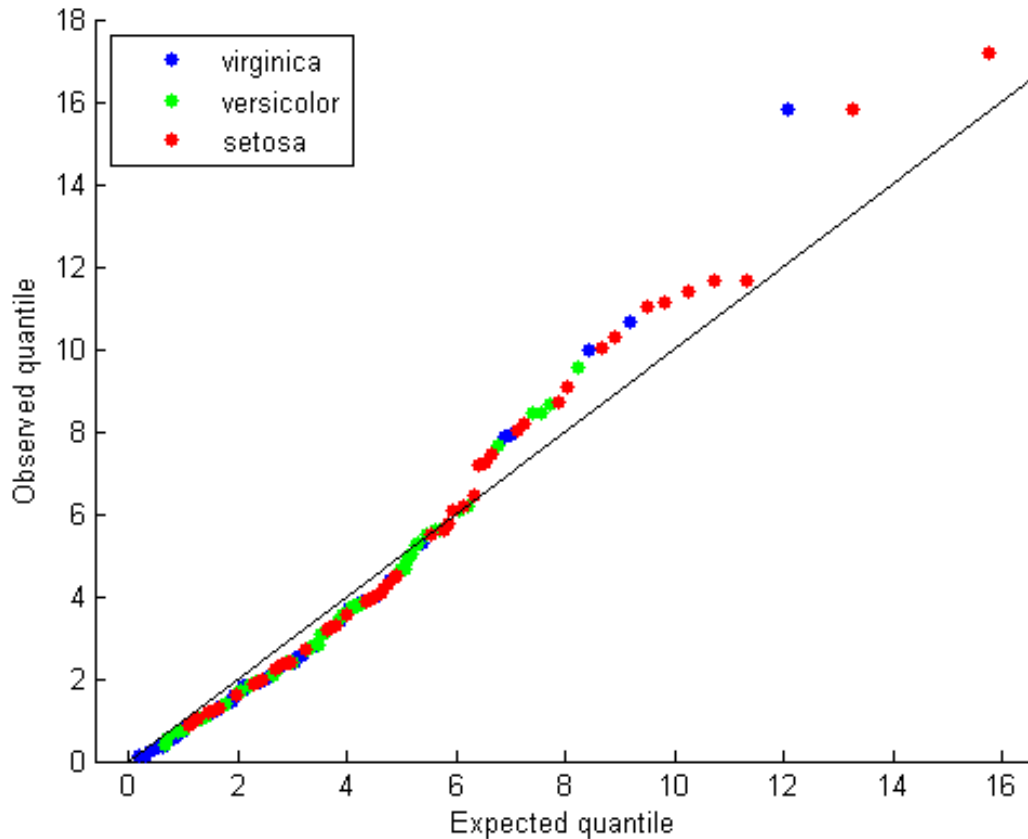
### Q-Q Plot

A Q-Q plot graphically shows whether an empirical distribution is close to a theoretical distribution. If the two are equal, the Q-Q plot lies on a 45° line. If not, the Q-Q plot strays from the 45° line.

**Check Q-Q Plots for Linear and Quadratic Discriminants.** For linear discriminant analysis, use a single covariance matrix for all classes.

```
load fisheriris;
prednames = {'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'};
L = ClassificationDiscriminant.fit(meas, species, 'PredictorNames', prednames);
N = L.NObservations;
K = numel(L.ClassNames);
mahL = mahal(L, L.X, 'ClassLabels', L.Y);
D = 4;
expQ = chi2inv(((1:N)-0.5)/N, D); % expected quantiles
[mahL, sorted] = sort(mahL); % sorted observed quantiles
figure;
gscatter(expQ, mahL, L.Y(sorted), 'bgr', [], [], 'off');
legend('virginica', 'versicolor', 'setosa', 'Location', 'NW');
xlabel('Expected quantile');
ylabel('Observed quantile');
line([0 20], [0 20], 'color', 'k');
```

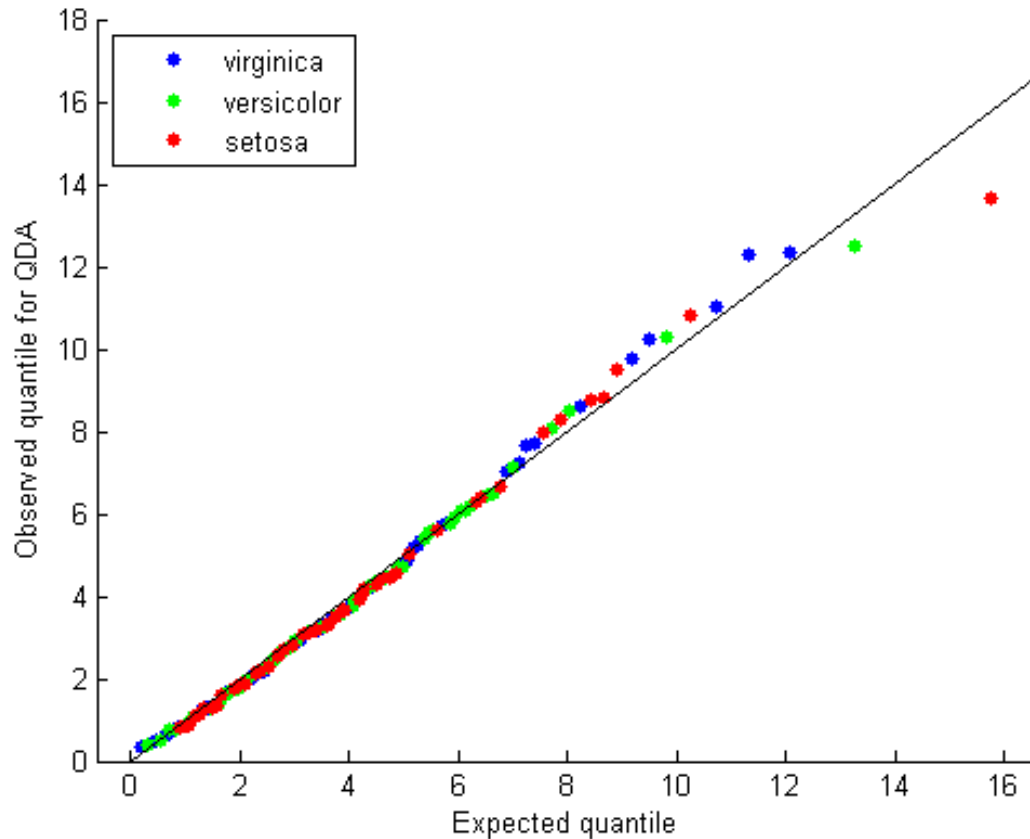




Overall, the agreement between the expected and observed quantiles is good. Look at the right half of the plot. The deviation of the plot from the 45° line upward indicates that the data has tails heavier than a normal distribution. There are three possible outliers on the right: two observations from class 'setosa' and one observation from class 'virginica'.

As shown in “Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 15-30, the data does not match a single covariance matrix. Redo the calculations for a quadratic discriminant.

```
load fisheriris;
prednames = {'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'};
Q = ClassificationDiscriminant.fit(meas, species, 'PredictorNames', prednames,
Nclass = [50 50 50];
N = L.NObservations;
K = numel(L.ClassNames);
mahQ = mahal(Q, Q.X, 'ClassLabels', Q.Y);
expQ = chi2inv((1:N)-0.5)/N, D);
[mahQ, sorted] = sort(mahQ);
figure;
gscatter(expQ, mahQ, Q.Y(sorted), 'bgr', [], [], 'off');
legend('virginica', 'versicolor', 'setosa', 'Location', 'NW');
xlabel('Expected quantile');
ylabel('Observed quantile for QDA');
line([0 20], [0 20], 'color', 'k');
```



The Q-Q plot shows a better agreement between the observed and expected quantiles. There is only one outlier candidate, from class 'setosa'.

### **Mardia Kurtosis Test of Multivariate Normality**

The Mardia kurtosis test (see Mardia [4]) is an alternative to examining a Q-Q plot. It gives a numeric approach to deciding if data matches a Gaussian mixture model.

In the Mardia kurtosis test you compute  $M$ , the mean of the fourth power of the Mahalanobis distance of the data from the class means. If the data is normally distributed with constant covariance matrix (and is thus suitable for linear discriminant analysis),  $M$  is asymptotically distributed as normal with mean  $d(d + 2)$  and variance  $8d(d + 2)/n$ , where

- $d$  is the number of predictors (number of dimensions in the data).
- $n$  is the total number of observations.

The Mardia test is two sided: check whether  $M$  is close enough to  $d(d + 2)$  with respect to a normal distribution of variance  $8d(d + 2)/n$ .

**Example: Mardia Kurtosis Test for Linear and Quadratic Discriminants.**

Check whether the Fisher iris data is approximately normally distributed for both linear and quadratic discriminant analysis. According to “Bartlett Test of Equal Covariance Matrices for Linear Discriminant Analysis” on page 15-30, the data is not normal for linear discriminant analysis (the covariance matrices are different). “Check Q-Q Plots for Linear and Quadratic Discriminants” on page 15-31 indicates that the data is well modeled by a Gaussian mixture model with different covariances per class. Check these conclusions with the Mardia kurtosis test:

```
load fisheriris;
prednames = {'SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth'};
L = ClassificationDiscriminant.fit(meas, species, 'PredictorNames', prednames)
mahL = mahal(L, L.X, 'ClassLabels', L.Y);
D = 4;
N = L.NObservations;
obsKurt = mean(mahL.^2);
expKurt = D*(D+2);
varKurt = 8*D*(D+2)/N;
[~, pval] = ztest(obsKurt, expKurt, sqrt(varKurt))
```

```
pval =
    0.0208
```

The Mardia test indicates to reject the hypothesis that the data is normally distributed.

Continuing the example with quadratic discriminant analysis:

```
Q = ClassificationDiscriminant.fit(meas,species,'PredictorNames',prednames,
mahQ = mahal(Q,Q.X,'ClassLabels',Q.Y);
obsKurt = mean(mahQ.^2);
[~,pval] = ztest(obsKurt,expKurt,sqrt(varKurt))
```

```
pval =
```

```
0.7230
```

Because `pval` is high, you conclude the data are consistent with the multivariate normal distribution.

## Bibliography

[1] Box, G. E. P. *A General Distribution Theory for a Class of Likelihood Criteria*. *Biometrika* 36(3), pp. 317–346, 1949.

[2] Fisher, R. A. *The Use of Multiple Measurements in Taxonomic Problems*. *Annals of Eugenics*, Vol. 7, pp. 179–188, 1936. Available at <http://digital.library.adelaide.edu.au/dspace/handle/2440/15227>.

[3] Guo, Y., T. Hastie, and R. Tibshirani. *Regularized Discriminant Analysis and Its Application in Microarray*. *Biostatistics*, Vol. 8, No. 1, pp. 86–100, 2007.

[4] Mardia, K. V. *Measures of multivariate skewness and kurtosis with applications*. *Biometrika* 57 (3), pp. 519–530, 1970.

## Naive Bayes Classification

The Naive Bayes classifier is designed for use when features are independent of one another within each class, but it appears to work well in practice even when that independence assumption is not valid. It classifies data in two steps:

- 1 Training step: Using the training samples, the method estimates the parameters of a probability distribution, assuming features are conditionally independent given the class.
- 2 Prediction step: For any unseen test sample, the method computes the posterior probability of that sample belonging to each class. The method then classifies the test sample according to the largest posterior probability.

The class-conditional independence assumption greatly simplifies the training step since you can estimate the one-dimensional class-conditional density for each feature individually. While the class-conditional independence between features is not true in general, research shows that this optimistic assumption works well in practice. This assumption of class independence allows the Naive Bayes classifier to better estimate the parameters required for accurate classification while using less training data than many other classifiers. This makes it particularly effective for datasets containing many predictors or features.

### Supported Distributions

Naive Bayes classification is based on estimating  $P(X|Y)$ , the probability or probability density of features  $X$  given class  $Y$ . The Naive Bayes classification object `NaiveBayes` provides support for normal (Gaussian), kernel, multinomial, and multivariate multinomial distributions. It is possible to use different distributions for different features.

### Normal (Gaussian) Distribution

The 'normal' distribution is appropriate for features that have normal distributions in each class. For each feature you model with a normal distribution, the Naive Bayes classifier estimates a separate normal

distribution for each class by computing the mean and standard deviation of the training data in that class.

### **Kernel Distribution**

The 'kernel' distribution is appropriate for features that have a continuous distribution. It does not require a strong assumption such as a normal distribution and you can use it in cases where the distribution of a feature may be skewed or have multiple peaks or modes. It requires more computing time and more memory than the normal distribution. For each feature you model with a kernel distribution, the Naive Bayes classifier computes a separate kernel density estimate for each class based on the training data for that class. By default the kernel is the normal kernel, and the classifier selects a width automatically for each class and feature. It is possible to specify different kernels for each feature, and different widths for each feature or class.

### **Multinomial Distribution**

The multinomial distribution (specify with the 'mn' keyword) is appropriate when all features represent counts of a set of words or tokens. This is sometimes called the "bag of words" model. For example, an email spam classifier might be based on features that count the number of occurrences of various tokens in an email. One feature might count the number of exclamation points, another might count the number of times the word "money" appears, and another might count the number of times the recipient's name appears. This is a Naive Bayes model under the further assumption that the total number of tokens (or the total document length) is independent of response class.

For the multinomial option, each feature represents the count of one token. The classifier counts the set of relative token probabilities separately for each class. The classifier defines the multinomial distribution for each row by the vector of probabilities for the corresponding class, and by  $N$ , the total token count for that row.

Classification is based on the relative frequencies of the tokens. For a row in which no token appears,  $N$  is 0 and no classification is possible. This classifier is not appropriate when the total number of tokens provides information about the response class.

### **Multivariate Multinomial Distribution**

The multivariate multinomial distribution (specify with the 'mvmn' keyword) is appropriate for categorical features. For example, you could fit a feature describing the weather in categories such as rain/sun/snow/clouds using the multivariate multinomial model. The feature categories are sometimes called the feature levels, and differ from the class levels for the response variable.

For each feature you model with a multivariate multinomial distribution, the Naive Bayes classifier computes a separate set of probabilities for the set of feature levels for each class.



## Performance Curves

### In this section...

“Introduction to Performance Curves” on page 15-41

“What are ROC Curves?” on page 15-41

“Evaluate Classifier Performance Using `perfcurve`” on page 15-41

### Introduction to Performance Curves

After a classification algorithm such as `NaiveBayes` or `TreeBagger` has trained on data, you may want to examine the performance of the algorithm on a specific test dataset. One common way of doing this would be to compute a gross measure of performance such as quadratic loss or accuracy, averaged over the entire test dataset.

### What are ROC Curves?

You may want to inspect the classifier performance more closely, for example, by plotting a Receiver Operating Characteristic (ROC) curve. By definition, a ROC curve [1,2] shows true positive rate versus false positive rate (equivalently, sensitivity versus  $1 - \text{specificity}$ ) for different thresholds of the classifier output. You can use it, for example, to find the threshold that maximizes the classification accuracy or to assess, in more broad terms, how the classifier performs in the regions of high sensitivity and high specificity.

### Evaluate Classifier Performance Using `perfcurve`

`perfcurve` computes measures for a plot of classifier performance. You can use this utility to evaluate classifier performance on test data after you train the classifier. Various measures such as mean squared error, classification error, or exponential loss can summarize the predictive power of a classifier in a single number. However, a performance curve offers more information as it lets you explore the classifier performance across a range of thresholds on its output.

You can use `perfcurve` with any classifier or, more broadly, with any method that returns a numeric score for an instance of input data. By convention adopted here,

- A high score returned by a classifier for any given instance signifies that the instance is likely from the positive class.
- A low score signifies that the instance is likely from the negative classes.

For some classifiers, you can interpret the score as the posterior probability of observing an instance of the positive class at point  $X$ . An example of such a score is the fraction of positive observations in a leaf of a decision tree. In this case, scores fall into the range from 0 to 1 and scores from positive and negative classes add up to unity. Other methods can return scores ranging between minus and plus infinity, without any obvious mapping from the score to the posterior class probability.

`perfcurve` does not impose any requirements on the input score range. Because of this lack of normalization, you can use `perfcurve` to process scores returned by any classification, regression, or fit method. `perfcurve` does not make any assumptions about the nature of input scores or relationships between the scores for different classes. As an example, consider a problem with three classes, A, B, and C, and assume that the scores returned by some classifier for two instances are as follows:

	A	B	C
instance 1	0.4	0.5	0.1
instance 2	0.4	0.1	0.5

If you want to compute a performance curve for separation of classes A and B, with C ignored, you need to address the ambiguity in selecting A over B. You could opt to use the score ratio,  $s(A)/s(B)$ , or score difference,  $s(A) - s(B)$ ; this choice could depend on the nature of these scores and their normalization. `perfcurve` always takes one score per instance. If you only supply scores for class A, `perfcurve` does not distinguish between observations 1 and 2. The performance curve in this case may not be optimal.

`perfcurve` is intended for use with classifiers that return scores, not those that return only predicted classes. As a counter-example, consider a decision tree that returns only hard classification labels, 0 or 1, for data with two classes. In this case, the performance curve reduces to a single point because classified instances can be split into positive and negative categories in one way only.

For input, `perfcurve` takes true class labels for some data and scores assigned by a classifier to these data. By default, this utility computes a Receiver Operating Characteristic (ROC) curve and returns values of 1-specificity, or false positive rate, for  $X$  and sensitivity, or true positive rate, for  $Y$ . You can choose other criteria for  $X$  and  $Y$  by selecting one out of several provided criteria or specifying an arbitrary criterion through an anonymous function. You can display the computed performance curve using `plot(X, Y)`.

`perfcurve` can compute values for various criteria to plot either on the  $x$ - or the  $y$ -axis. All such criteria are described by a 2-by-2 confusion matrix, a 2-by-2 cost matrix, and a 2-by-1 vector of scales applied to class counts.

The confusion matrix,  $C$ , is defined as

$$\begin{pmatrix} TP & FN \\ FP & TN \end{pmatrix}$$

where

- $P$  stands for "positive".
- $N$  stands for "negative".
- $T$  stands for "true".
- $F$  stands for "false".

For example, the first row of the confusion matrix defines how the classifier identifies instances of the positive class:  $C(1, 1)$  is the count of correctly identified positive instances and  $C(1, 2)$  is the count of positive instances misidentified as negative.

The cost matrix defines the cost of misclassification for each category:

$$\begin{pmatrix} Cost(P | P) & Cost(N | P) \\ Cost(P | N) & Cost(N | N) \end{pmatrix}$$

where  $Cost(I | J)$  is the cost of assigning an instance of class  $J$  to class  $I$ . Usually  $Cost(I | J) = 0$  for  $I = J$ . For flexibility, `perfcurve` allows you to specify nonzero costs for correct classification as well.

The two scales include prior information about class probabilities. `perfcurve` computes these scales by taking  $\text{scale}(P) = \text{prior}(P) * N$  and  $\text{scale}(N) = \text{prior}(N) * P$  and normalizing the sum  $\text{scale}(P) + \text{scale}(N)$  to 1.  $P = TP + FN$  and  $N = TN + FP$  are the total instance counts in the positive and negative class, respectively. The function then applies the scales as multiplicative factors to the counts from the corresponding class: `perfcurve` multiplies counts from the positive class by  $\text{scale}(P)$  and counts from the negative class by  $\text{scale}(N)$ . Consider, for example, computation of positive predictive value,  $PPV = TP / (TP + FP)$ . TP counts come from the positive class and FP counts come from the negative class. Therefore, you need to scale TP by  $\text{scale}(P)$  and FP by  $\text{scale}(N)$ , and the modified formula for PPV with prior probabilities taken into account is now:

$$PPV = \frac{\text{scale}(P) * TP}{\text{scale}(P) * TP + \text{scale}(N) * FP}$$

If all scores in the data are above a certain threshold, `perfcurve` classifies all instances as 'positive'. This means that TP is the total number of instances in the positive class and FP is the total number of instances in the negative class. In this case, PPV is simply given by the prior:

$$PPV = \frac{\text{prior}(P)}{\text{prior}(P) + \text{prior}(N)}$$

The `perfcurve` function returns two vectors, X and Y, of performance measures. Each measure is some function of `confusion`, `cost`, and `scale` values. You can request specific measures by name or provide a function handle to compute a custom measure. The function you provide should take `confusion`, `cost`, and `scale` as its three inputs and return a vector of output values.

The criterion for X must be a monotone function of the positive classification count, or equivalently, threshold for the supplied scores. If `perfcurve` cannot perform a one-to-one mapping between values of the X criterion and score thresholds, it exits with an error message.

By default, `perfcurve` computes values of the X and Y criteria for all possible score thresholds. Alternatively, it can compute a reduced number of specific X values supplied as an input argument. In either case, for M requested values, `perfcurve` computes M+1 values for X and Y. The first value out of these M+1 values is special. `perfcurve` computes it by setting the TP instance count

to zero and setting TN to the total count in the negative class. This value corresponds to the 'reject all' threshold. On a standard ROC curve, this translates into an extra point placed at (0,0).

If there are NaN values among input scores, `perfcurve` can process them in either of two ways:

- It can discard rows with NaN scores.
- It can add them to false classification counts in the respective class.

That is, for any threshold, instances with NaN scores from the positive class are counted as false negative (FN), and instances with NaN scores from the negative class are counted as false positive (FP). In this case, the first value of X or Y is computed by setting TP to zero and setting TN to the total count minus the NaN count in the negative class. For illustration, consider an example with two rows in the positive and two rows in the negative class, each pair having a NaN score:

<b>Class</b>	<b>Score</b>
Negative	0.2
Negative	NaN
Positive	0.7
Positive	NaN

If you discard rows with NaN scores, then as the score cutoff varies, `perfcurve` computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where rows 1 and 3 are classified correctly, and rows 2 and 4 are omitted.

TP	FN	FP	TN
0	1	0	1
1	0	0	1
1	0	1	0

If you add rows with NaN scores to the false category in their respective classes, `perfcurve` computes performance measures as in the following table. For example, a cutoff of 0.5 corresponds to the middle row where now rows

2 and 4 are counted as incorrectly classified. Notice that only the FN and FP columns differ between these two tables.

TP	FN	FP	TN
0	2	1	1
1	1	1	1
1	1	2	0

For data with three or more classes, `perfcurve` takes one positive class and a list of negative classes for input. The function computes the X and Y values using counts in the positive class to estimate TP and FN, and using counts in all negative classes to estimate TN and FP. `perfcurve` can optionally compute Y values for each negative class separately and, in addition to Y, return a matrix of size M-by-C, where M is the number of elements in X or Y and C is the number of negative classes. You can use this functionality to monitor components of the negative class contribution. For example, you can plot TP counts on the X-axis and FP counts on the Y-axis. In this case, the returned matrix shows how the FP component is split across negative classes.

You can also use `perfcurve` to estimate confidence intervals. `perfcurve` computes confidence bounds using either cross-validation or bootstrap. If you supply cell arrays for `labels` and `scores`, `perfcurve` uses cross-validation and treats elements in the cell arrays as cross-validation folds. If you set input parameter `NBoot` to a positive integer, `perfcurve` generates `nboot` bootstrap replicas to compute pointwise confidence bounds.

`perfcurve` estimates the confidence bounds using one of two methods:

- Vertical averaging (VA) — estimate confidence bounds on Y and T at fixed values of X. Use the `XVals` input parameter to use this method for computing confidence bounds.
- Threshold averaging (TA) — estimate confidence bounds for X and Y at fixed thresholds for the positive class score. Use the `TVals` input parameter to use this method for computing confidence bounds.

To use observation weights instead of observation counts, you can use the `'Weights'` parameter in your call to `perfcurve`. When you use this parameter, to compute X, Y and T or to compute confidence bounds by cross-validation, `perfcurve` uses your supplied observation weights instead of

observation counts. To compute confidence bounds by bootstrap, perfcurve samples  $N$  out of  $N$  with replacement using your weights as multinomial sampling probabilities.





# Nonparametric Supervised Learning

---

- “Supervised Learning (Machine Learning) Workflow and Algorithms” on page 16-2
- “Classification Using Nearest Neighbors” on page 16-9
- “Classification Trees and Regression Trees” on page 16-31
- “Splitting Categorical Predictors” on page 16-56
- “Ensemble Methods” on page 16-59
- “Support Vector Machines (SVM)” on page 16-162
- “Bibliography” on page 16-184

## Supervised Learning (Machine Learning) Workflow and Algorithms

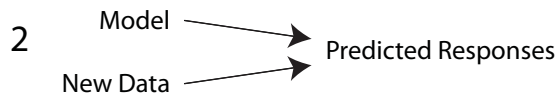
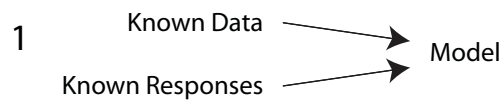
### In this section...

“Steps in Supervised Learning (Machine Learning)” on page 16-2

“Characteristics of Algorithms” on page 16-7

### Steps in Supervised Learning (Machine Learning)

Supervised learning (machine learning) takes a known set of input data and known responses to the data, and seeks to build a predictor model that generates reasonable predictions for the response to new data.



Suppose you want to predict if someone will have a heart attack within a year. You have a set of data on previous people, including age, weight, height, blood pressure, etc. You know if the previous people had heart attacks within a year of their data measurements. So the problem is combining all the existing data into a model that can predict whether a new person will have a heart attack within a year.

Supervised learning splits into two broad categories:

- Classification for responses that can have just a few known values, such as 'true' or 'false'. Classification algorithms apply to nominal, not ordinal response values.

- Regression for responses that are a real number, such as miles per gallon for a particular car.

You can have trouble deciding whether you have a classification problem or a regression problem. In that case, create a regression model first, because they are often more computationally efficient.

While there are many Statistics Toolbox algorithms for supervised learning, most use the same basic workflow for obtaining a predictor model. (Detailed instruction on the steps for ensemble learning is in “Framework for Ensemble Learning” on page 16-59.) The steps for supervised learning are:

- 1 “Prepare Data” on page 16-3
- 2 “Choose an Algorithm” on page 16-4
- 3 “Fit a Model” on page 16-4
- 4 “Choose a Validation Method” on page 16-5
- 5 “Examine Fit and Update Until Satisfied” on page 16-6
- 6 “Use Fitted Model for Predictions” on page 16-6

### **Prepare Data**

All supervised learning methods start with an input data matrix, usually called  $X$  here. Each row of  $X$  represents one observation. Each column of  $X$  represents one variable, or predictor. Represent missing entries with NaN values in  $X$ . Statistics Toolbox supervised learning algorithms can handle NaN values, either by ignoring them or by ignoring any row with a NaN value.

You can use various data types for response data  $Y$ . Each element in  $Y$  represents the response to the corresponding row of  $X$ . Observations with missing  $Y$  data are ignored.

- For regression,  $Y$  must be a numeric vector with the same number of elements as the number of rows of  $X$ .
- For classification,  $Y$  can be any of these data types. This table also contains the method of including missing entries.

Data Type	Missing Entry
Numeric vector	NaN
Categorical vector	<undefined>
Character array	Row of spaces
Cell array of strings	' '
Logical vector	(Cannot represent)

## Choose an Algorithm

There are tradeoffs between several characteristics of algorithms, such as:

- Speed of training
- Memory usage
- Predictive accuracy on new data
- Transparency or interpretability, meaning how easily you can understand the reasons an algorithm makes its predictions

Details of the algorithms appear in “Characteristics of Algorithms” on page 16-7. More detail about ensemble algorithms is in “Choose an Applicable Ensemble Method” on page 16-62.

## Fit a Model

The fitting function you use depends on the algorithm you choose.

Algorithm	Fitting Function
Classification Trees	<code>ClassificationTree.fit</code>
Regression Trees	<code>RegressionTree.fit</code>
Discriminant Analysis (classification)	<code>ClassificationDiscriminant.fit</code>
<i>K</i> -Nearest Neighbors (classification)	<code>ClassificationKNN.fit</code>
Naive Bayes (classification)	<code>NaiveBayes.fit</code>

Algorithm	Fitting Function
Classification or Regression Ensembles	fitensemble
Classification or Regression Ensembles in Parallel	TreeBagger

### Choose a Validation Method

The three main methods to examine the accuracy of the resulting fitted model are:

- Examine the resubstitution error. For examples, see:
  - “Example: Resubstitution Error of a Classification Tree” on page 16-39
  - “Example: Cross Validating a Regression Tree” on page 16-40
  - “Example: Test Ensemble Quality” on page 16-72
  - “Example: Resubstitution Error of a Discriminant Analysis Classifier” on page 15-18
- Examine the cross-validation error. For examples, see:
  - “Example: Cross Validating a Regression Tree” on page 16-40
  - “Example: Test Ensemble Quality” on page 16-72
  - “Classification with Many Categorical Levels” on page 16-91
  - “Cross Validating a Discriminant Analysis Classifier” on page 15-20
- Examine the out-of-bag error for bagged decision trees. For examples, see:
  - “Example: Test Ensemble Quality” on page 16-72
  - “Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger” on page 16-125
  - “Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger” on page 16-134

## Examine Fit and Update Until Satisfied

After validating the model, you might want to change it for better accuracy, better speed, or to use less memory.

- Change fitting parameters to try to get a more accurate model. For examples, see:
  - “Tuning RobustBoost” on page 16-115
  - “Example: Unequal Classification Costs” on page 16-85
  - “Improve a Discriminant Analysis Classifier” on page 15-15
- Change fitting parameters to try to get a smaller model. This sometimes gives a model with more accuracy. For examples, see:
  - “Example: Selecting Appropriate Tree Depth” on page 16-41
  - “Example: Pruning a Classification Tree” on page 16-44
  - “Surrogate Splits” on page 16-95
  - “Example: Regularizing a Regression Ensemble” on page 16-105
  - “Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger” on page 16-125
  - “Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger” on page 16-134
- Try a different algorithm. For applicable choices, see:
  - “Characteristics of Algorithms” on page 16-7
  - “Choose an Applicable Ensemble Method” on page 16-62

When satisfied with a model of some types, you can trim it using the appropriate `compact` method (`compact` for classification trees, `compact` for classification ensembles, `compact` for regression trees, `compact` for regression ensembles, `compact` for discriminant analysis). `compact` removes training data and pruning information, so the model uses less memory.

## Use Fitted Model for Predictions

To predict classification or regression response for most fitted models, use the `predict` method:

```
Ypredicted = predict(obj,Xnew)
```

- obj is the fitted model object.
- Xnew is the new input data.
- Ypredicted is the predicted response, either classification or regression.

For classtree, use the eval method instead of predict.

## Characteristics of Algorithms

This table shows typical characteristics of the various supervised learning algorithms. The characteristics in any particular case can vary from the listed ones. Use the table as a guide for your initial choice of algorithms, but be aware that the table can be inaccurate for some problems.

### Characteristics of Supervised Learning Algorithms

Algorithm	Predictive Accuracy	Fitting Speed	Prediction Speed	Memory Usage	Easy to Interpret	Handles Categorical Predictors
Trees	Low	Fast	Fast	Low	Yes	Yes
SVM	High	Medium	*	*	*	No
Naive Bayes	Low	**	**	**	Yes	Yes
Nearest Neighbor	***	Fast***	Medium	High	No	Yes***
Discriminant Analysis	****	Fast	Fast	Low	Yes	No
Ensembles	See “Suggestions for Choosing an Appropriate Ensemble Algorithm” on page 16-64 and “General Characteristics of Ensemble Algorithms” on page 16-65					

\* — SVM prediction speed and memory usage are good if there are few support vectors, but can be poor if there are many support vectors. When you use a kernel function, it can be difficult to interpret how SVM classifies data, though the default linear scheme is easy to interpret.

\*\* — Naive Bayes speed and memory usage are good for simple distributions, but can be poor for kernel distributions and large data sets.

\*\*\* — Nearest Neighbor usually has good predictions in low dimensions, but can have poor predictions in high dimensions. For linear search, Nearest Neighbor does not perform any fitting. For  $k$ d-trees, Nearest Neighbor does perform fitting. Nearest Neighbor can have either continuous or categorical predictors, but not both.

\*\*\*\* — Discriminant Analysis is accurate when the modeling assumptions are satisfied (multivariate normal by class). Otherwise, the predictive accuracy varies.



## Classification Using Nearest Neighbors

### In this section...

“Pairwise Distance” on page 16-9

“ $k$ -Nearest Neighbor Search and Radius Search” on page 16-12

“ $K$ -Nearest Neighbor Classification for Supervised Learning” on page 16-26

“Construct a KNN Classifier” on page 16-27

“Examine the Quality of a KNN Classifier” on page 16-27

“Predict Classification Based on a KNN Classifier” on page 16-28

“Modify a KNN Classifier” on page 16-29

### Pairwise Distance

Categorizing query points based on their distance to points in a training dataset can be a simple yet effective way of classifying new points. You can use various metrics to determine the distance, described next. Use `pdist2` to find the distance between a set of data and query points.

### Distance Metrics

Given an  $m \times n$  data matrix  $X$ , which is treated as  $m \times (1 \text{-by-} n)$  row vectors  $x_1, x_2, \dots, x_m$ , and  $m \times n$  data matrix  $Y$ , which is treated as  $m \times (1 \text{-by-} n)$  row vectors  $y_1, y_2, \dots, y_m$ , the various distances between the vector  $x_s$  and  $y_t$  are defined as follows:

- Euclidean distance

$$d_{st}^2 = (x_s - y_t)(x_s - y_t)'$$

The Euclidean distance is a special case of the Minkowski metric, where  $p = 2$ .

- Standardized Euclidean distance

$$d_{st}^2 = (x_s - y_t)V^{-1}(x_s - y_t)'$$

where  $V$  is the  $n$ -by- $n$  diagonal matrix whose  $j$ th diagonal element is  $S(j)^2$ , where  $S$  is the vector containing the inverse weights.

- Mahalanobis distance

$$d_{st}^2 = (x_s - y_t)C^{-1}(x_s - y_t)'$$

where  $C$  is the covariance matrix.

- City block metric

$$d_{st} = \sum_{j=1}^n |x_{sj} - y_{tj}|$$

The city block distance is a special case of the Minkowski metric, where  $p = 1$ .

- Minkowski metric

$$d_{st} = \sqrt[p]{\sum_{j=1}^n |x_{sj} - y_{tj}|^p}$$

For the special case of  $p = 1$ , the Minkowski metric gives the city block metric, for the special case of  $p = 2$ , the Minkowski metric gives the Euclidean distance, and for the special case of  $p = \infty$ , the Minkowski metric gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j \{|x_{sj} - y_{tj}|\}$$

The Chebychev distance is a special case of the Minkowski metric, where  $p = \infty$ .

- Cosine distance

$$d_{st} = \left( 1 - \frac{x_s y_t'}{\sqrt{(x_s x_s')(y_t y_t')}} \right)$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \bar{x}_s)(y_t - \bar{y}_t)'}{\sqrt{(x_s - \bar{x}_s)(x_s - \bar{x}_s)' (y_t - \bar{y}_t)(y_t - \bar{y}_t)'}}$$

where

$$\bar{x}_s = \frac{1}{n} \sum_j x_{sj}$$

and

$$\bar{y}_t = \frac{1}{n} \sum_j y_{tj}$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq y_{tj}) / n)$$

- Jaccard distance

$$d_{st} = \frac{\#[(x_{sj} \neq y_{tj}) \cap ((x_{sj} \neq 0) \cup (y_{tj} \neq 0))]}{\#[(x_{sj} \neq 0) \cup (y_{tj} \neq 0)]}$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \bar{r}_s)(r_t - \bar{r}_t)'}{\sqrt{(r_s - \bar{r}_s)(r_s - \bar{r}_s)' (r_t - \bar{r}_t)(r_t - \bar{r}_t)'}}$$

where

- $r_{sj}$  is the rank of  $x_{sj}$  taken over  $x_{1j}, x_{2j}, \dots, x_{mj}$ , as computed by tiedrank.
- $r_{tj}$  is the rank of  $y_{tj}$  taken over  $y_{1j}, y_{2j}, \dots, y_{mj}$ , as computed by tiedrank.
- $r_s$  and  $r_t$  are the coordinate-wise rank vectors of  $x_s$  and  $y_t$ , i.e.,  $r_s = (r_{s1}, r_{s2}, \dots, r_{sn})$  and  $r_t = (r_{t1}, r_{t2}, \dots, r_{tn})$ .

$$\bar{r}_s = \frac{1}{n} \sum_j r_{sj} = \frac{(n+1)}{2}.$$

$$\bar{r}_t = \frac{1}{n} \sum_j r_{tj} = \frac{(n+1)}{2}.$$

### ***k*-Nearest Neighbor Search and Radius Search**

Given a set  $X$  of  $n$  points and a distance function,  $k$ -nearest neighbor ( $k$ NN) search lets you find the  $k$  closest points in  $X$  to a query point or set of points  $Y$ . The  $k$ NN search technique and  $k$ NN-based algorithms are widely used as benchmark learning rules. The relative simplicity of the  $k$ NN search technique makes it easy to compare the results from other classification techniques to  $k$ NN results. The technique has been used in various areas such as:

- bioinformatics
- image processing and data compression
- document retrieval
- computer vision
- multimedia database
- marketing data analysis

You can use  $k$ NN search for other machine learning algorithms, such as:

- $k$ NN classification
- local weighted regression
- missing data imputation and interpolation
- density estimation

You can also use  $k$ NN search with many distance-based learning functions, such as K-means clustering.

In contrast, for a positive real value  $r$ , `rangesearch` finds all points in  $X$  that are within a distance  $r$  of each point in  $Y$ . This fixed-radius search is closely

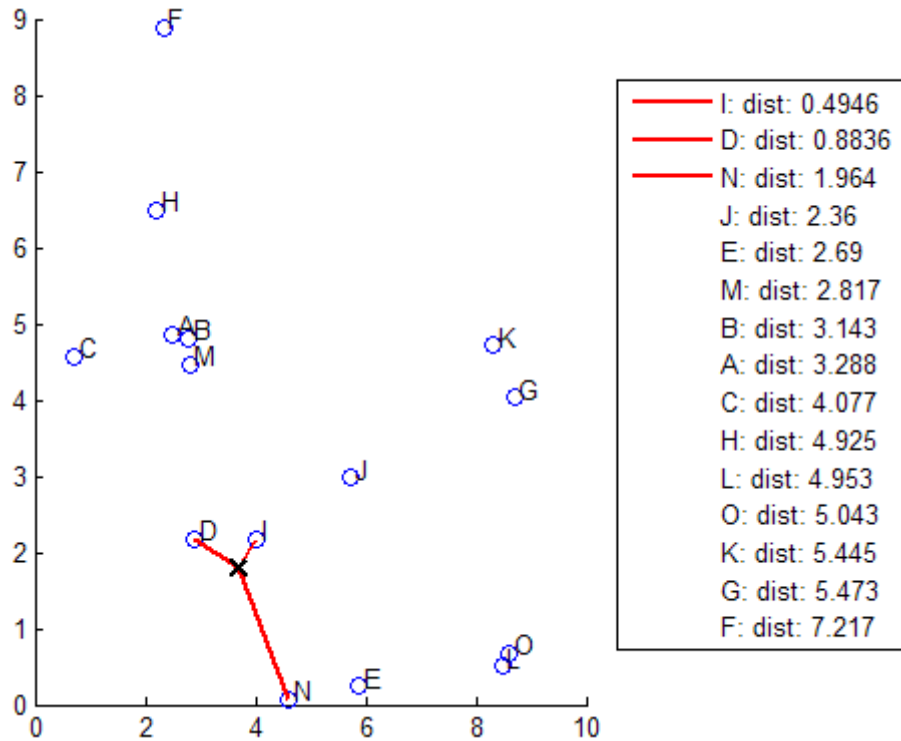
related to  $k$ NN search, as it supports the same distance metrics and search classes, and uses the same search algorithms.

### ***k*-Nearest Neighbor Search Using Exhaustive Search**

When your input data meets any of the following criteria, `knnsearch` uses the exhaustive search method by default to find the  $k$ -nearest neighbors:

- The number of columns of  $X$  is more than 10.
- $X$  is sparse.
- The distance measure is either:
  - 'seuclidean'
  - 'mahalanobis'
  - 'cosine'
  - 'correlation'
  - 'spearman'
  - 'hamming'
  - 'jaccard'
  - A custom distance function

`knnsearch` also uses the exhaustive search method if your search object is an `ExhaustiveSearcher` object. The exhaustive search method finds the distance from each query point to every point in  $X$ , ranks them in ascending order, and returns the  $k$  points with the smallest distances. For example, this diagram shows the  $k = 3$  nearest neighbors.



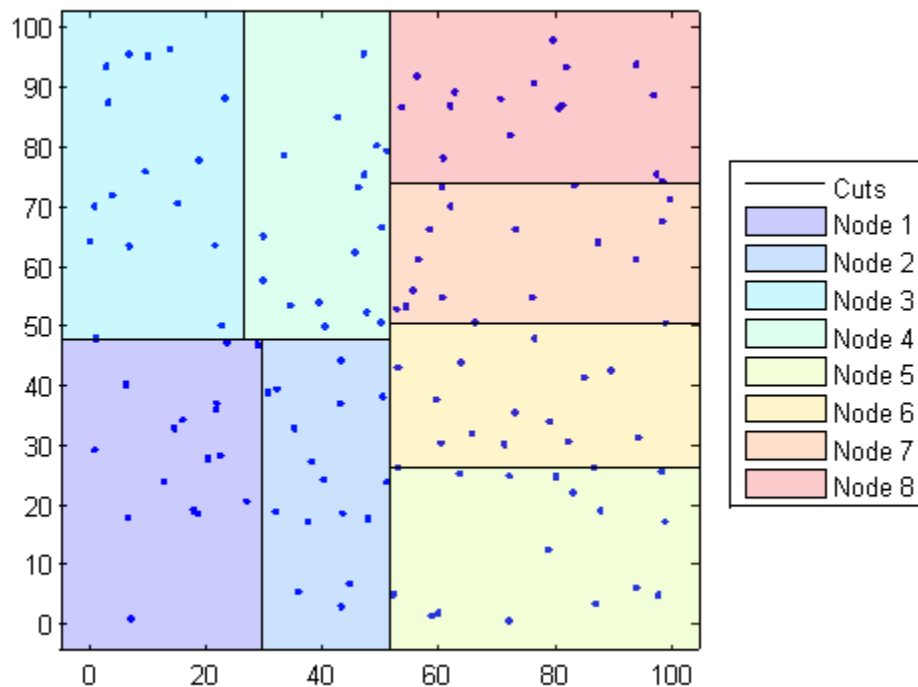
### ***k*-Nearest Neighbor Search Using a *kd*-Tree**

When your input data meets all of the following criteria, `knnsearch` creates a *kd*-tree by default to find the *k*-nearest neighbors:

- The number of columns of *X* is less than 10.
- *X* is not sparse.
- The distance measure is either:
  - 'euclidean' (default)
  - 'cityblock'
  - 'minkowski'
  - 'chebychev'

`knnsearch` also uses a *kd*-tree if your search object is a `KDTreeSearcher` object.

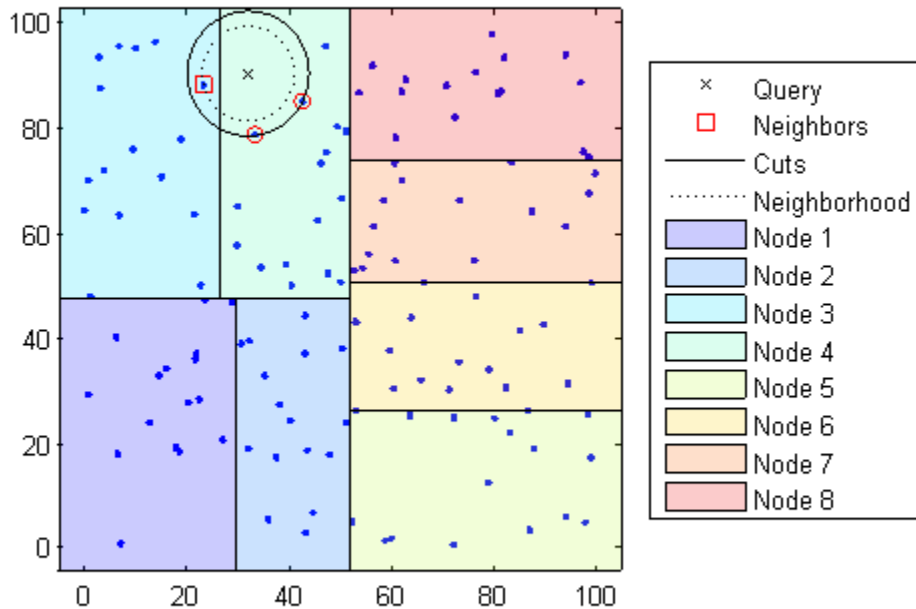
*kd*-trees divide your data into nodes with at most `BucketSize` (default is 50) points per node, based on coordinates (as opposed to categories). The following diagrams illustrate this concept using patch objects to color code the different “buckets.”



When you want to find the  $k$ -nearest neighbors to a given query point, `knnsearch` does the following:

- 1 Determines the node to which the query point belongs. In the following example, the query point (32,90) belongs to Node 4.

- 2 Finds the closest  $k$  points within that node and its distance to the query point. In the following example, the points in red circles are equidistant from the query point, and are the closest points to the query point within Node 4.
- 3 Chooses all other nodes having any area that is within the same distance, in any direction, from the query point to the  $k$ th closest point. In this example, only Node 3 overlaps the solid black circle centered at the query point with radius equal to the distance to the closest points within Node 4.
- 4 Searches nodes within that range for any points closer to the query point. In the following example, the point in a red square is slightly closer to the query point than those within Node 4.





Using a *kd*-tree for large datasets with fewer than 10 dimensions (columns) can be much more efficient than using the exhaustive search method, as `knnsearch` needs to calculate only a subset of the distances. To maximize the efficiency of *kd*-trees, use a `KDTreeSearcher` object.

## What Are Search Objects?

Basically, objects are a convenient way of storing information. Classes of related objects (for example, all search objects) have the same properties with values and types relevant to a specified search method. In addition to storing information within objects, you can perform certain actions (called *methods*) on objects.

All search objects have a `knnsearch` method specific to that class. This lets you efficiently perform a *k*-nearest neighbors search on your object for that specific object type. In addition, there is a generic `knnsearch` function that searches without creating or using an object.

To determine which type of object and search method is best for your data, consider the following:

- Does your data have many columns, say more than 10? The `ExhaustiveSearcher` object may perform better.
- Is your data sparse? Use the `ExhaustiveSearcher` object.
- Do you want to use one of these distance measures to find the nearest neighbors? Use the `ExhaustiveSearcher` object.
  - `'seuclidean'`
  - `'mahalanobis'`
  - `'cosine'`
  - `'correlation'`
  - `'spearman'`
  - `'hamming'`
  - `'jaccard'`
  - A custom distance function

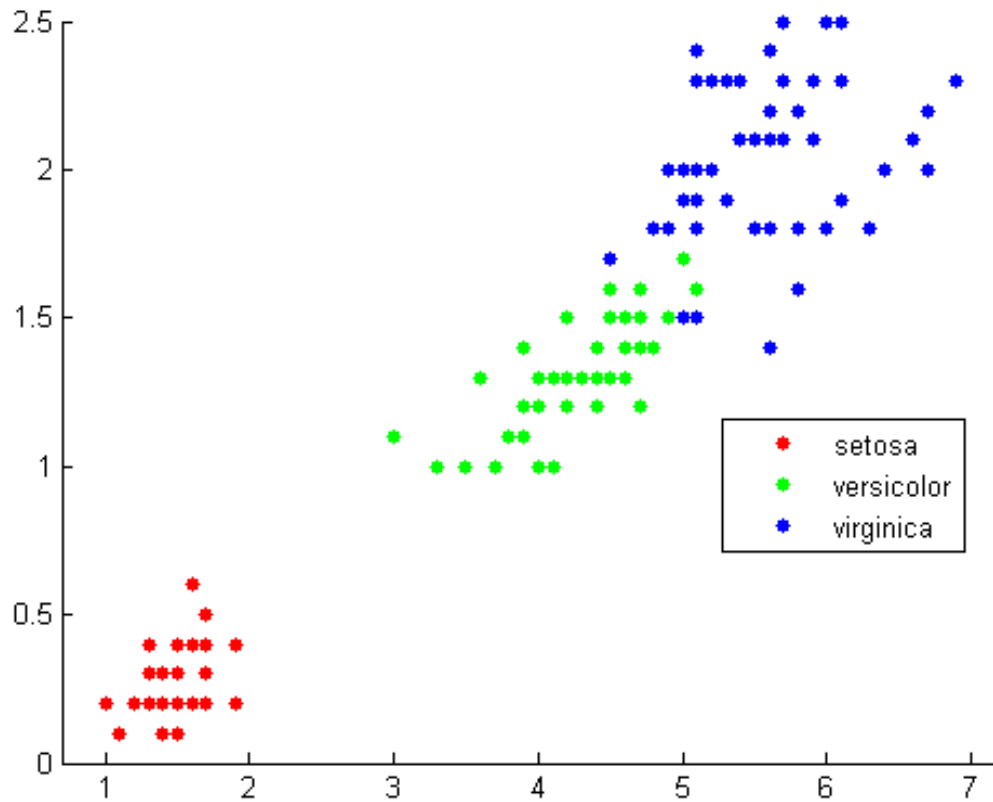
- Is your dataset huge (but with fewer than 10 columns)? Use the `KDTreeSearcher` object.
- Are you searching for the nearest neighbors for a large number of query points? Use the `KDTreeSearcher` object.

### **Classifying Query Data Using `knnsearch`**

This example shows how to classify query data using `knnsearch` .

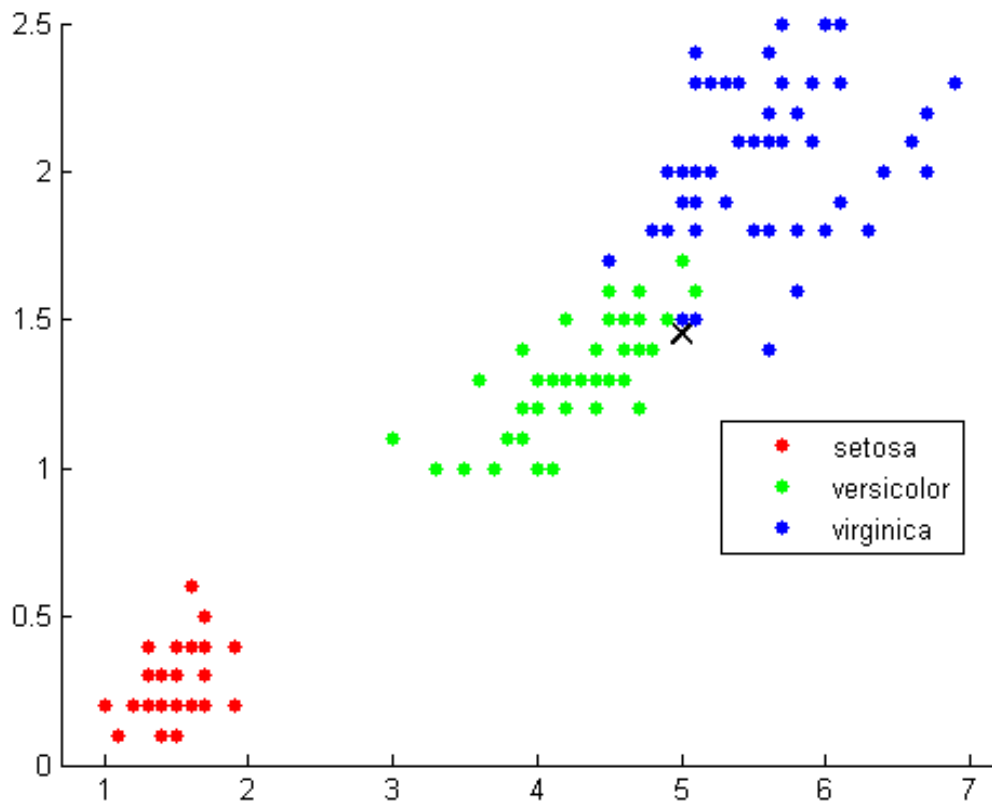
Classify a new point based on the last two columns of the Fisher iris data. Using only the last two columns makes it easier to plot.

```
load fisheriris
x = meas(:,3:4);
gscatter(x(:,1),x(:,2),species)
set(legend,'location','best')
```



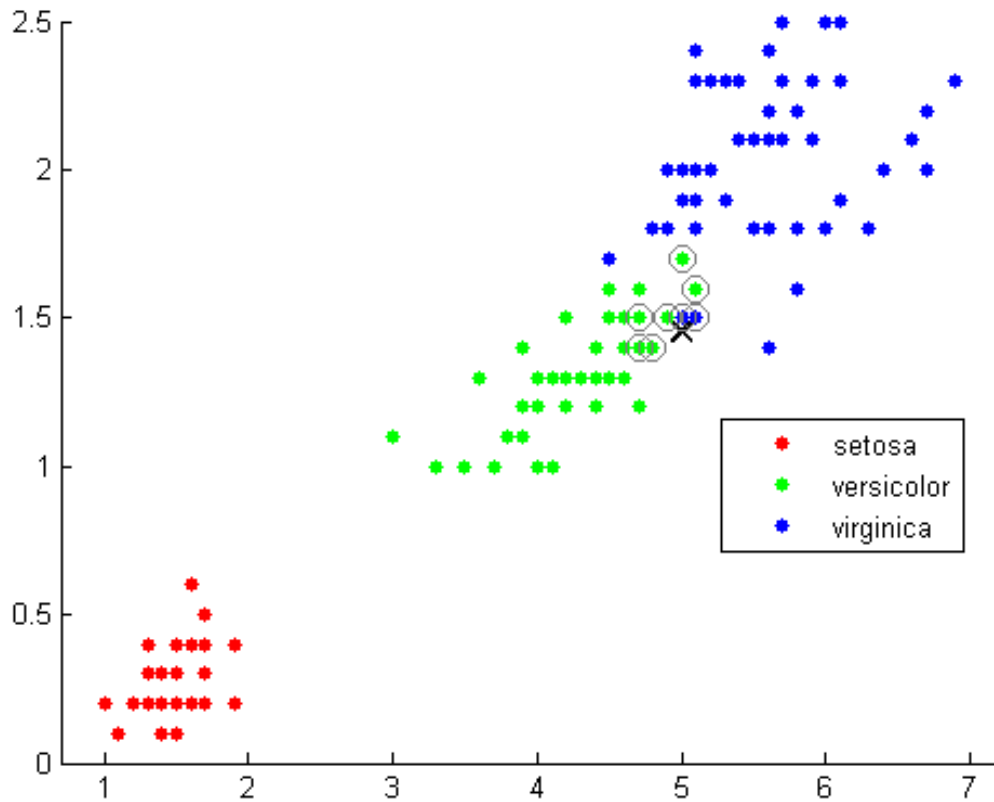
Plot the new point.

```
newpoint = [5 1.45];  
line(newpoint(1),newpoint(2),'marker','x','color','k',...  
      'markersize',10,'linewidth',2)
```



Find the 10 sample points closest to the new point.

```
[n,d] = knnsearch(x,newpoint,'k',10);  
line(x(n,1),x(n,2),'color',[.5 .5 .5],'marker','o',...  
      'linestyle','none','markersize',10)
```



It appears that `knnsearch` has found only the nearest eight neighbors. In fact, this particular dataset contains duplicate values.

```
x(n, :)
```

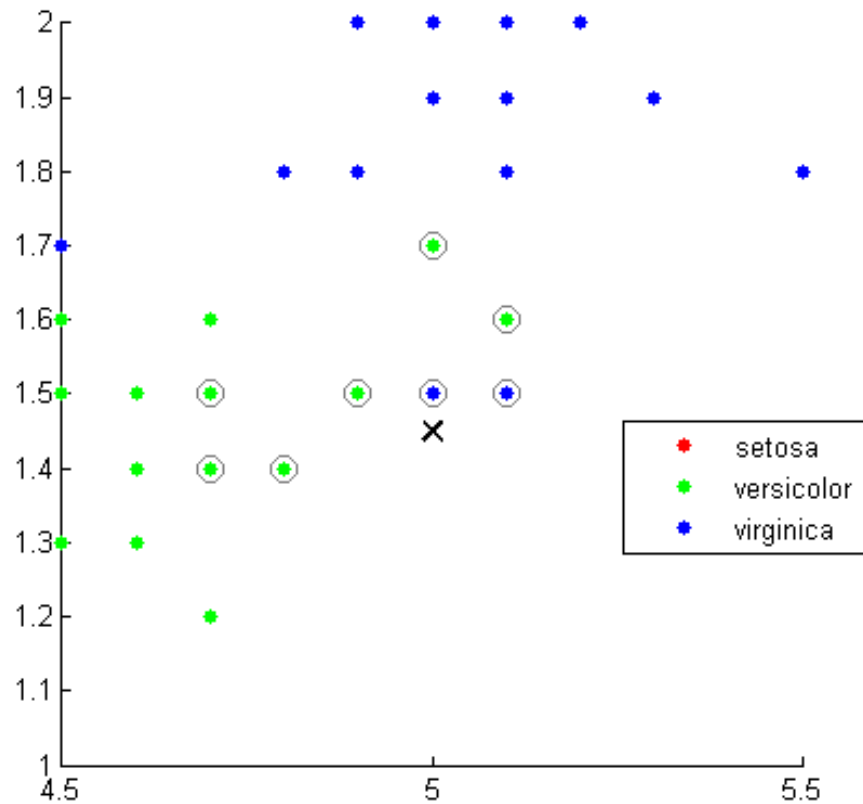
```
ans =
```

```
5.0000    1.5000
4.9000    1.5000
```

4.9000	1.5000
5.1000	1.5000
5.1000	1.6000
4.8000	1.4000
5.0000	1.7000
4.7000	1.4000
4.7000	1.4000
4.7000	1.5000

Make the axes equal so the calculated distances correspond to the apparent distances on the plot axis equal and zoom in to see the neighbors better.

```
set(gca,'xlim',[4.5 5.5],'ylim',[1 2]); axis square
```



Find the species of the 10 neighbors.

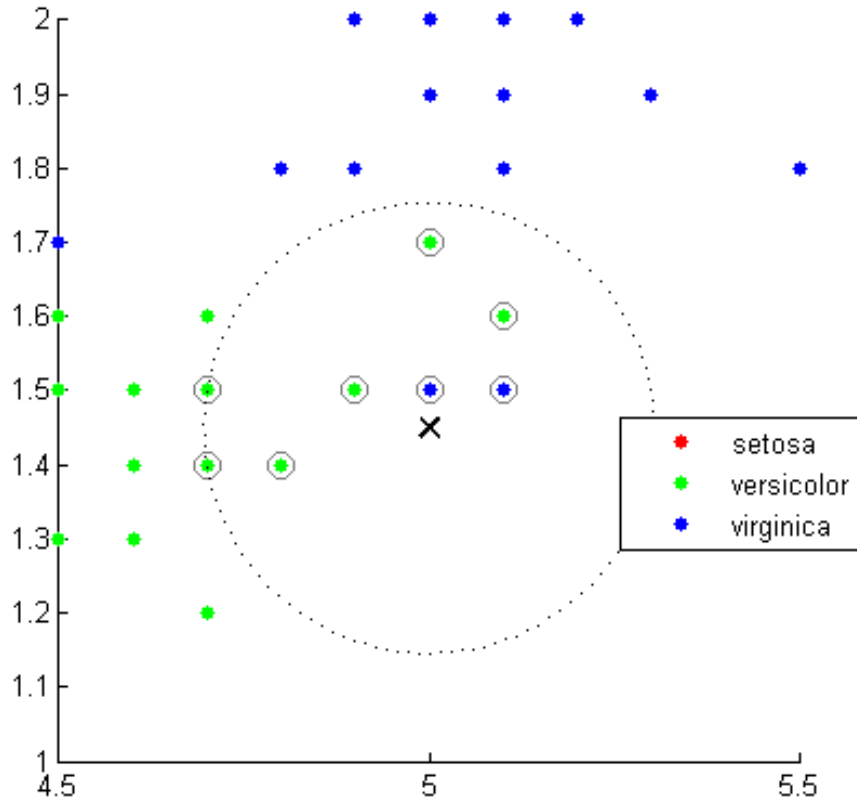
```
tabulate(species(n))
```

Value	Count	Percent
virginica	2	20.00%
versicolor	8	80.00%

Using a rule based on the majority vote of the 10 nearest neighbors, you can classify this new point as a versicolor.

Visually identify the neighbors by drawing a circle around the group of them. Define the center and diameter of a circle, based on the location of the new point.

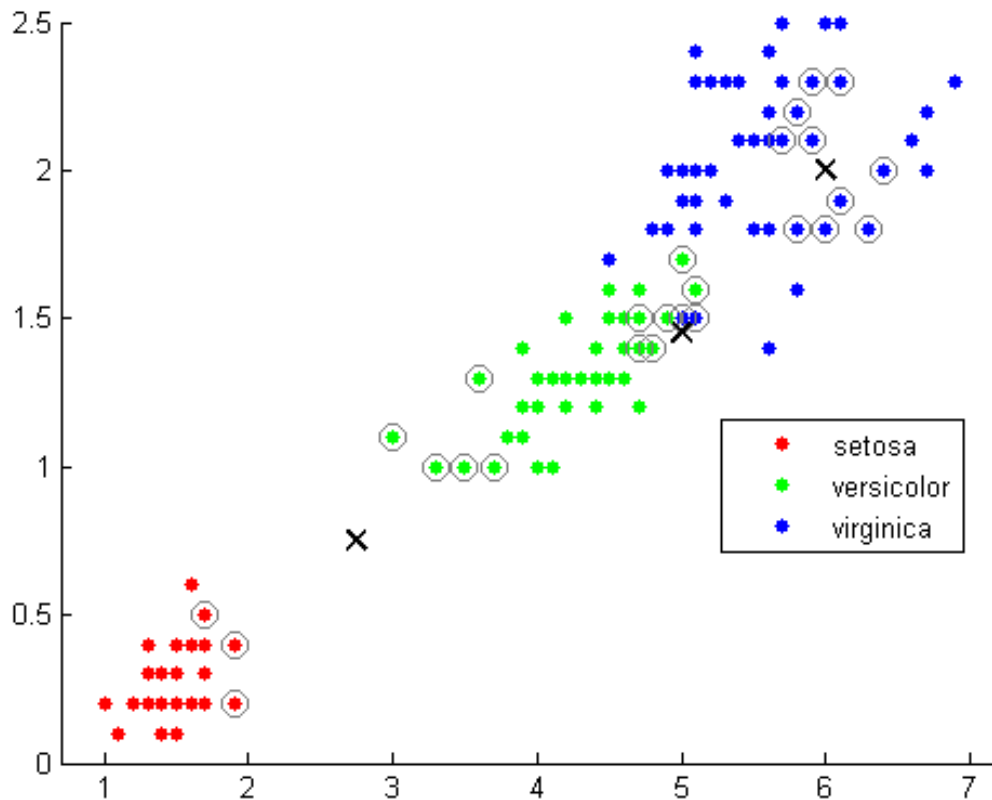
```
ctr = newpoint - d(end);
diameter = 2*d(end);
% Draw a circle around the 10 nearest neighbors.
h = rectangle('position',[ctr,diameter,diameter],...
    'curvature',[1 1]);
set(h,'linestyle',':')
```





Using the same dataset, find the 10 nearest neighbors to three new points.

```
figure
newpoint2 = [5 1.45;6 2;2.75 .75];
gscatter(x(:,1),x(:,2),species)
legend('location','best')
[n2,d2] = knnsearch(x,newpoint2,'k',10);
line(x(n2,1),x(n2,2),'color',[.5 .5 .5],'marker','o',...
     'linestyle','none','markersize',10)
line(newpoint2(:,1),newpoint2(:,2),'marker','x','color','k',...
     'markersize',10,'linewidth',2,'linestyle','none')
```



Find the species of the 10 nearest neighbors for each new point.

```
tabulate(species(n2(1, :)))
```

Value	Count	Percent
virginica	2	20.00%
versicolor	8	80.00%

```
tabulate(species(n2(2, :)))
```

Value	Count	Percent
virginica	10	100.00%

```
tabulate(species(n2(3, :)))
```

Value	Count	Percent
versicolor	7	70.00%
setosa	3	30.00%

For more examples using `knnsearch` methods and function, see the individual reference pages.

## **K-Nearest Neighbor Classification for Supervised Learning**

The `ClassificationKNN` class lets you:

- “Construct a KNN Classifier” on page 16-27
- “Examine the Quality of a KNN Classifier” on page 16-27
- “Predict Classification Based on a KNN Classifier” on page 16-28
- “Modify a KNN Classifier” on page 16-29

Work with the classifier as you would with `ClassificationTree` or `ClassificationDiscriminant`. In particular, prepare your data for classification according to the procedure in “Steps in Supervised Learning (Machine Learning)” on page 16-2. Then construct the classifier using `ClassificationKNN.fit`.

## Construct a KNN Classifier

This example shows how to construct a  $k$ -nearest neighbor classifier for the Fisher iris data.

Load the Fisher iris data.

```
load fisheriris
X = meas; % use all data for fitting
Y = species; % response data
```

Construct the classifier using `ClassificationKNN.fit`.

```
mdl = ClassificationKNN.fit(X,Y)
```

```
mdl =
```

```
ClassificationKNN
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}
  ResponseName: 'Y'
  ClassNames: {'setosa' 'versicolor' 'virginica'}
  ScoreTransform: 'none'
  NObservations: 150
  Distance: 'euclidean'
  NumNeighbors: 1
```

Properties, Methods

A default  $k$ -nearest neighbor classifier uses just the single nearest neighbor. Often, a classifier is more robust with more neighbors than that. Change the neighborhood size of `mdl` to 4, meaning `mdl` classifies using the four nearest neighbors:

```
mdl.NumNeighbors = 4;
```

## Examine the Quality of a KNN Classifier

This example shows how to examine the quality of a  $k$ -nearest neighbor classifier using resubstitution and cross validation.

Construct a KNN classifier for the Fisher iris data as in “Construct a KNN Classifier” on page 16-27.

```
load fisheriris
X = meas; % use all data for fitting
Y = species; % response data
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',4);
```

Examine the resubstitution loss, which, by default, is the fraction of misclassifications from the predictions of mdl. (For nondefault cost, weights, or priors, see `ClassificationKNN.loss`.)

```
rloss = resubLoss(mdl)
```

```
rloss =
```

```
0.0400
```

The classifier predicts incorrectly for 4% of the training data.

Construct a cross-validated classifier from the model.

```
cvmdl = crossval(mdl);
```

Examine the cross-validation loss, which is the average loss of each cross-validation model when predicting on data that is not used for training.

```
kloss = kfoldLoss(cvmdl)
```

```
kloss =
```

```
0.0600
```

The cross-validated classification accuracy resembles the resubstitution accuracy. Therefore, you can expect mdl to misclassify approximately 5% of new data, assuming that the new data has about the same distribution as the training data.

## **Predict Classification Based on a KNN Classifier**

This example shows how to predict classification for a  $k$ -nearest neighbor classifier.

Construct a default KNN classifier for the Fisher iris data as in “Construct a KNN Classifier” on page 16-27.

```
load fisheriris
X = meas; % use all data for fitting
Y = species; % response data
mdl = ClassificationKNN.fit(X,Y);
```

Predict the classification of an average flower.

```
flwr = mean(X); % an average flower
flwrClass = predict(mdl,flwr)
```

```
flwrClass =
    'versicolor'
```

## Modify a KNN Classifier

This example shows how to modify a  $k$ -nearest neighbor classifier.

Construct a default KNN classifier for the Fisher iris data as in “Construct a KNN Classifier” on page 16-27.

```
load fisheriris
X = meas; % use all data for fitting
Y = species; % response data
mdl = ClassificationKNN.fit(X,Y);
```

Modify the model to use the three nearest neighbors, rather than the default one nearest neighbor.

```
mdl.NumNeighbors = 3;
```

Compare the resubstitution predictions and cross-validation loss with the new number of neighbors.

```
rloss = resubLoss(mdl)
```

```
rloss =
    0.0400
```

```
rng('default')
cvmdl = crossval mdl, 'kfold', 5);
kloss = kfoldLoss(cvmdl)
```

```
kloss =

    0.0333
```

The model with three neighbors has lower cross-validated loss than a model with four neighbors (see “Examine the Quality of a KNN Classifier” on page 16-27).

Modify the model to use cosine distance instead of the default, and examine the loss. To use cosine distance, you must recreate the model using the exhaustive search method.

```
cmd1 = ClassificationKNN.fit(X,Y,'NSMethod','exhaustive',...
    'Distance','cosine');
cmd1.NumNeighbors = 3;
closs = resubLoss(cmd1)
```

```
closs =

    0.0200
```

The classifier now has lower resubstitution error than before.

Check the quality of a cross-validated version of the new model.

```
cvcmd1 = crossval(cmd1);
kcloss = kfoldLoss(cvcmd1)
```

```
kcloss =

    0.0333
```

The cross-validated loss is the same as before. The lesson is that improving the resubstitution error does not necessarily produce a model with better predictions.

## Classification Trees and Regression Trees

### In this section...

“What Are Classification Trees and Regression Trees?” on page 16-31

“Creating a Classification Tree” on page 16-32

“Creating a Regression Tree” on page 16-32

“Viewing a Classification or Regression Tree” on page 16-33

“How the Fit Methods Create Trees” on page 16-36

“Predicting Responses With Classification and Regression Trees” on page 16-38

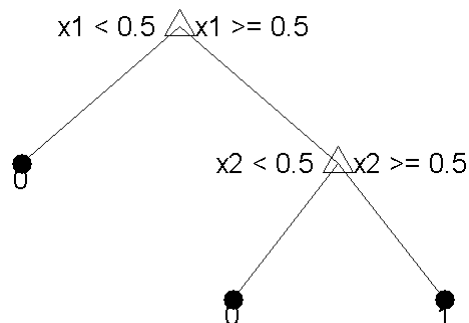
“Improving Classification Trees and Regression Trees” on page 16-39

“Alternative: classregtree” on page 16-48

### What Are Classification Trees and Regression Trees?

Classification trees and regression trees predict responses to data. To predict a response, follow the decisions in the tree from the root (beginning) node down to a leaf node. The leaf node contains the response. Classification trees give responses that are nominal, such as 'true' or 'false'. Regression trees give numeric responses.

Statistics Toolbox trees are binary. Each step in a prediction involves checking the value of one predictor (variable). For example, here is a simple classification tree:



This tree predicts classifications based on two predictors,  $x_1$  and  $x_2$ . To predict, start at the top node, represented by a triangle ( $\Delta$ ). The first decision is whether  $x_1$  is smaller than 0.5. If so, follow the left branch, and see that the tree classifies the data as type 0.

If, however,  $x_1$  exceeds 0.5, then follow the right branch to the lower-right triangle node. Here the tree asks if  $x_2$  is smaller than 0.5. If so, then follow the left branch to see that the tree classifies the data as type 0. If not, then follow the right branch to see that the that the tree classifies the data as type 1.

To learn how to prepare your data for classification or regression using decision trees, see “Steps in Supervised Learning (Machine Learning)” on page 16-2.

## Creating a Classification Tree

To create a classification tree for the ionosphere data:

```
load ionosphere % contains X and Y variables
ctree = ClassificationTree.fit(X,Y)
```

```
ctree =
```

```
ClassificationTree
    PredictorNames: {1x34 cell}
    ResponseName: 'Y'
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    CategoricalPredictors: []
    NObservations: 351
```

```
Properties, Methods
```

## Creating a Regression Tree

To create a regression tree for the carsmall data based on the Horsepower and Weight vectors for data, and MPG vector for response:

```
load carsmall % contains Horsepower, Weight, MPG
```



```
X = [Horsepower Weight];
rtree = RegressionTree.fit(X,MPG)

rtree =

    RegressionTree
      PredictorNames: {'x1' 'x2'}
      ResponseName: 'Y'
      ResponseTransform: 'none'
      CategoricalPredictors: []
      NObservations: 94
```

Properties, Methods

## Viewing a Classification or Regression Tree

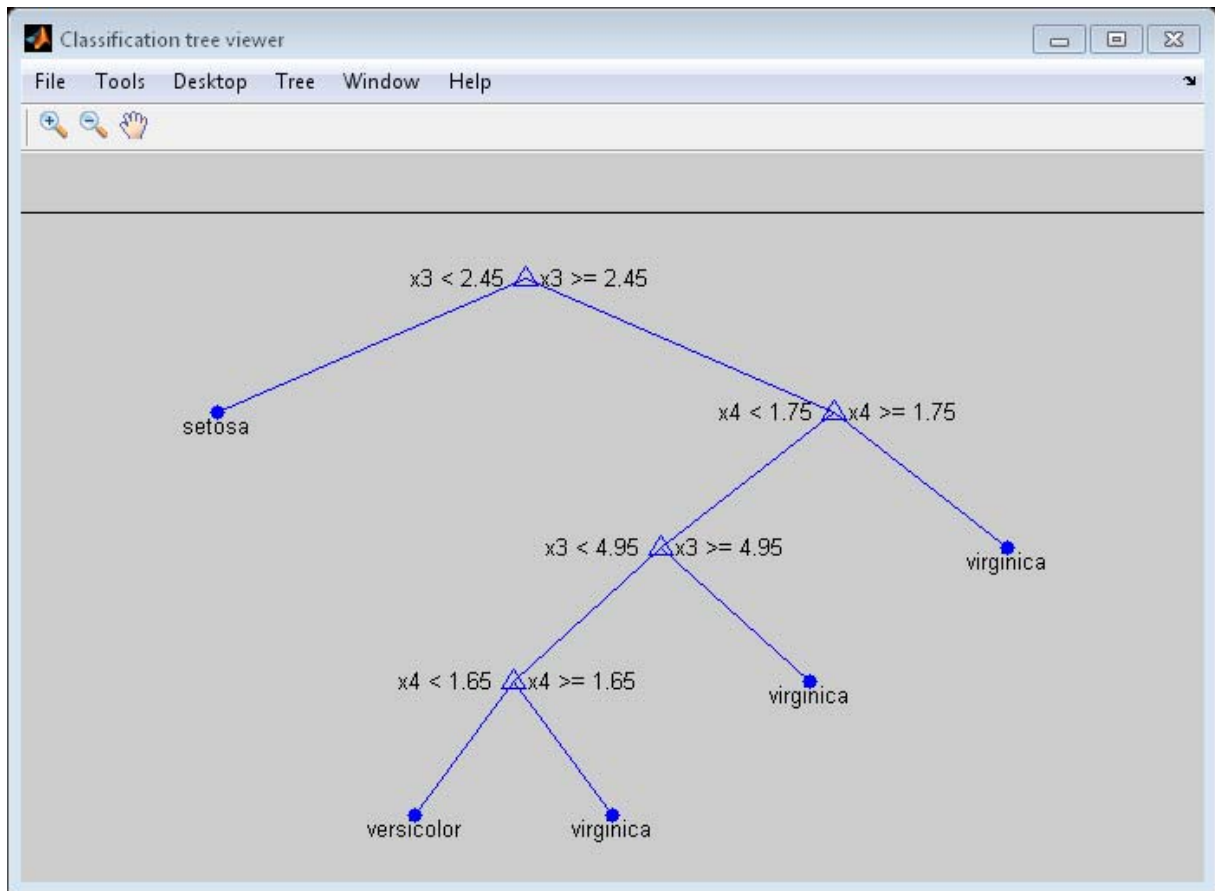
This example shows how to view a classification or a regression tree. There are two ways to view a tree: `view(tree)` returns a text description and `view(tree, 'mode', 'graph')` returns a graphic description of the tree.

Create and view a classification tree.

```
load fisheriris % load the sample data
ctree = ClassificationTree.fit(meas,species); % create classification tree
view(ctree) % text description
```

```
Decision tree for classification
1  if x3<2.45 then node 2 elseif x3>=2.45 then node 3 else setosa
2  class = setosa
3  if x4<1.75 then node 4 elseif x4>=1.75 then node 5 else versicolor
4  if x3<4.95 then node 6 elseif x3>=4.95 then node 7 else versicolor
5  class = virginica
6  if x4<1.65 then node 8 elseif x4>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

```
view(ctree, 'mode', 'graph') % graphic description
```



Now, create and view a regression tree.

```

load carsmall % load the sample data, contains Horsepower, Weight, MPG
X = [Horsepower Weight];
rtree = RegressionTree.fit(X,MPG,'MinParent',30); % create classification t
view(rtree) % text description
  
```

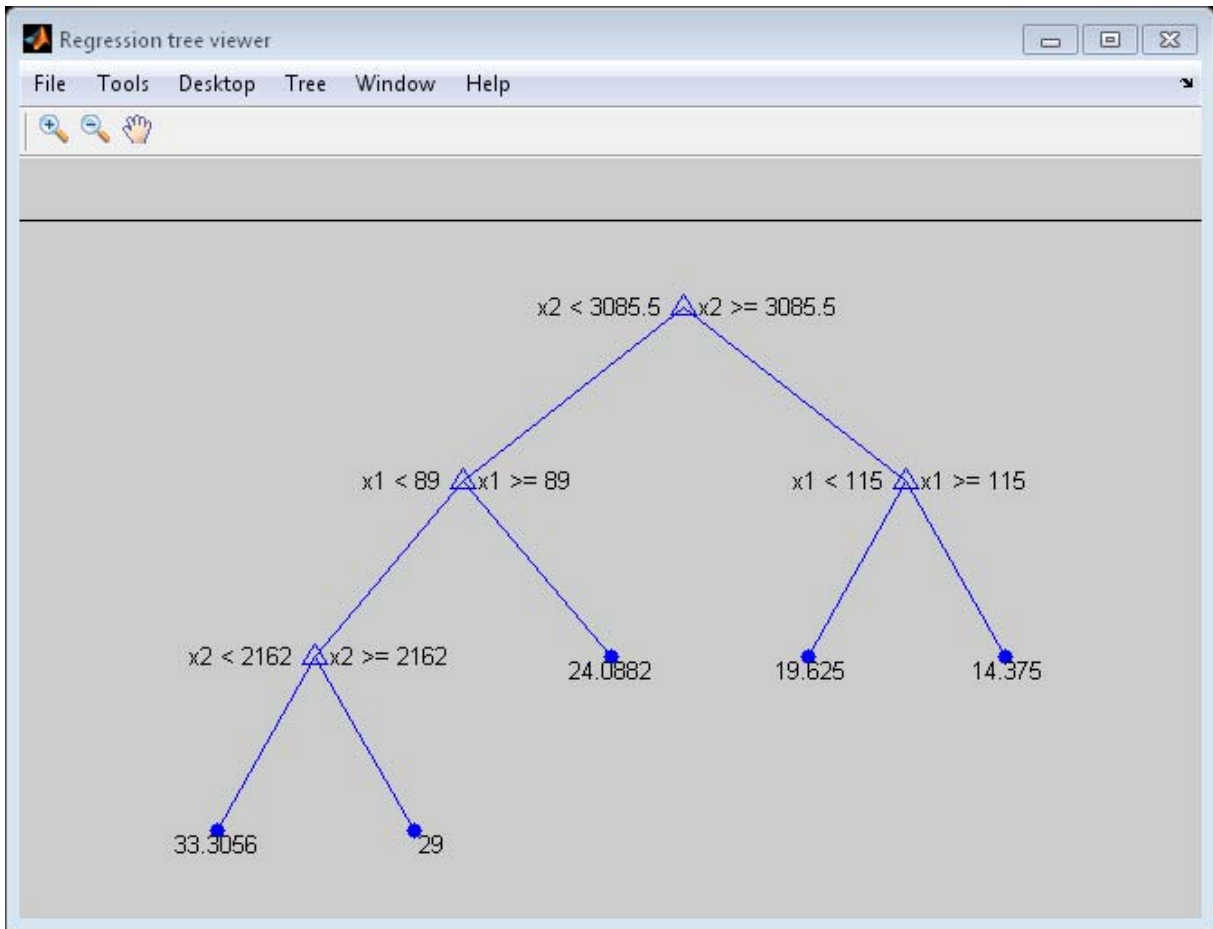
Decision tree for regression

```

1 if x2<3085.5 then node 2 elseif x2>=3085.5 then node 3 else 23.7181
  
```

```
2 if x1<89 then node 4 elseif x1>=89 then node 5 else 28.7931
3 if x1<115 then node 6 elseif x1>=115 then node 7 else 15.5417
4 if x2<2162 then node 8 elseif x2>=2162 then node 9 else 30.9375
5 fit = 24.0882
6 fit = 19.625
7 fit = 14.375
8 fit = 33.3056
9 fit = 29
```

```
view(rtree,'mode','graph') % graphic description
```



### How the Fit Methods Create Trees

The `ClassificationTree.fit` and `RegressionTree.fit` methods perform the following steps to create decision trees:

- 1 Start with all input data, and examine all possible binary splits on every predictor.
- 2 Select a split with best optimization criterion.

- If the split leads to a child node having too few observations (less than the `MinLeaf` parameter), select a split with the best optimization criterion subject to the `MinLeaf` constraint.

**3** Impose the split.

**4** Repeat recursively for the two child nodes.

The explanation requires two more items: description of the optimization criterion, and stopping rule.

**Stopping rule:** Stop splitting when any of the following hold:

- The node is *pure*.
  - For classification, a node is pure if it contains only observations of one class.
  - For regression, a node is pure if the mean squared error (MSE) for the observed response in this node drops below the MSE for the observed response in the entire data multiplied by the tolerance on quadratic error per node (`qetoler` parameter).
- There are fewer than `MinParent` observations in this node.
- Any split imposed on this node would produce children with fewer than `MinLeaf` observations.

**Optimization criterion:**

- Regression: mean-squared error (MSE). Choose a split to minimize the MSE of predictions compared to the training data.
- Classification: One of three measures, depending on the setting of the `SplitCriterion` name-value pair:
  - `'gdi'` (Gini's diversity index, the default)
  - `'twoing'`
  - `'deviance'`

For details, see `ClassificationTree` “Definitions” on page 21-326.

For a continuous predictor, a tree can split halfway between any two adjacent unique values found for this predictor. For a categorical predictor with  $L$  levels, a classification tree needs to consider  $2^{L-1}-1$  splits to find the optimal split. Alternatively, you can choose a heuristic algorithm to find a good split, as described in “Splitting Categorical Predictors” on page 16-56.

## Predicting Responses With Classification and Regression Trees

After creating a tree, you can easily predict responses for new data. Suppose  $X_{\text{new}}$  is new data that has the same number of columns as the original data  $X$ . To predict the classification or regression based on the tree and the new data, enter

```
Ynew = predict(tree,Xnew);
```

For each row of data in  $X_{\text{new}}$ , `predict` runs through the decisions in `tree` and gives the resulting prediction in the corresponding element of  $Y_{\text{new}}$ . For more information for classification, see the classification `predict` reference page; for regression, see the regression `predict` reference page.

For example, to find the predicted classification of a point at the mean of the ionosphere data:

```
load ionosphere % contains X and Y variables
ctree = ClassificationTree.fit(X,Y);
Ynew = predict(ctree,mean(X))
```

```
Ynew =
```

```
    'g'
```

To find the predicted MPG of a point at the mean of the `carsmall` data:

```
load carsmall % contains Horsepower, Weight, MPG
X = [Horsepower Weight];
rtree = RegressionTree.fit(X,MPG);
Ynew = predict(rtree,mean(X))
```

```
Ynew =
```

28.7931

## Improving Classification Trees and Regression Trees

You can tune trees by setting name-value pairs in `ClassificationTree.fit` and `RegressionTree.fit`. The remainder of this section describes how to determine the quality of a tree, how to decide which name-value pairs to set, and how to control the size of a tree:

- “Examining Resubstitution Error” on page 16-39
- “Cross Validation” on page 16-39
- “Control Depth or “Leafiness”” on page 16-40
- “Pruning” on page 16-44

### Examining Resubstitution Error

*Resubstitution error* is the difference between the response training data and the predictions the tree makes of the response based on the input training data. If the resubstitution error is high, you cannot expect the predictions of the tree to be good. However, having low resubstitution error does not guarantee good predictions for new data. Resubstitution error is often an overly optimistic estimate of the predictive error on new data.

**Example: Resubstitution Error of a Classification Tree.** Examine the resubstitution error of a default classification tree for the Fisher iris data:

```
load fisheriris
ctree = ClassificationTree.fit(meas,species);
resuberror = resubLoss(ctree)

resuberror =

    0.0200
```

The tree classifies nearly all the Fisher iris data correctly.

### Cross Validation

To get a better sense of the predictive accuracy of your tree for new data, cross validate the tree. By default, cross validation splits the training data

into 10 parts at random. It trains 10 new trees, each one on nine parts of the data. It then examines the predictive accuracy of each new tree on the data not included in training that tree. This method gives a good estimate of the predictive accuracy of the resulting tree, since it tests the new trees on new data.

**Example: Cross Validating a Regression Tree.** Examine the resubstitution and cross-validation accuracy of a regression tree for predicting mileage based on the carsmall data:

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
rtree = RegressionTree.fit(X,MPG);
resuberror = resubLoss(rtree)

resuberror =
    4.7188
```

The resubstitution loss for a regression tree is the mean-squared error. The resulting value indicates that a typical predictive error for the tree is about the square root of 4.7, or a bit over 2.

Now calculate the error by cross validating the tree:

```
rng('default')
cvrtree = crossval(rtree);
cvloss = kfoldLoss(cvrtree)

cvloss =
    23.8065
```

The cross-validated loss is almost 25, meaning a typical predictive error for the tree on new data is about 5. This demonstrates that cross-validated loss is usually higher than simple resubstitution loss.

### **Control Depth or “Leafiness”**

When you grow a decision tree, consider its simplicity and predictive power. A deep tree with many leaves is usually highly accurate on the training data. However, the tree is not guaranteed to show a comparable accuracy on an



independent test set. A leafy tree tends to overtrain, and its test accuracy is often far less than its training (resubstitution) accuracy. In contrast, a shallow tree does not attain high training accuracy. But a shallow tree can be more robust — its training accuracy could be close to that of a representative test set. Also, a shallow tree is easy to interpret.

If you do not have enough data for training and test, estimate tree accuracy by cross validation.

For an alternative method of controlling the tree depth, see “Pruning” on page 16-44.

**Example: Selecting Appropriate Tree Depth.** This example shows how to control the depth of a decision tree, and how to choose an appropriate depth.

- 1 Load the ionosphere data:

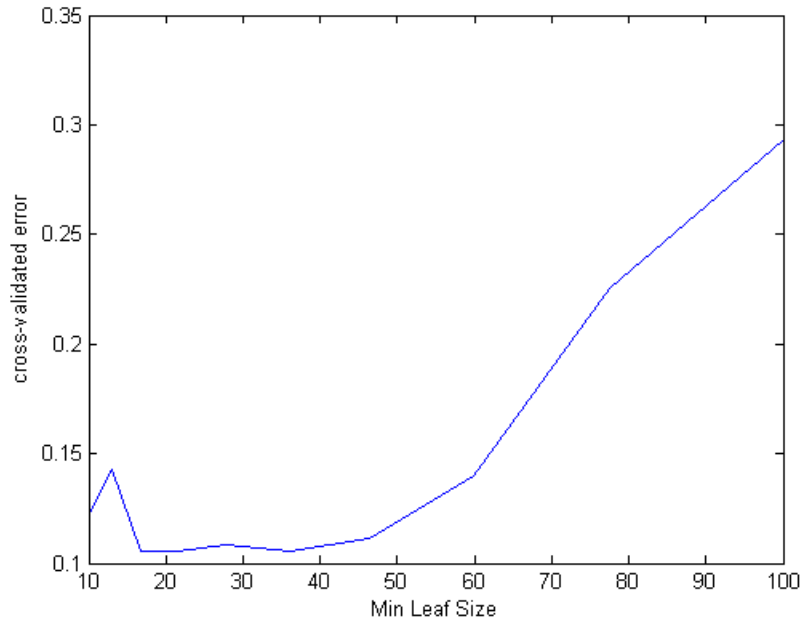
```
load ionosphere
```

- 2 Generate minimum leaf occupancies for classification trees from 10 to 100, spaced exponentially apart:

```
leafs = logspace(1,2,10);
```

- 3 Create cross validated classification trees for the ionosphere data with minimum leaf occupancies from leafs:

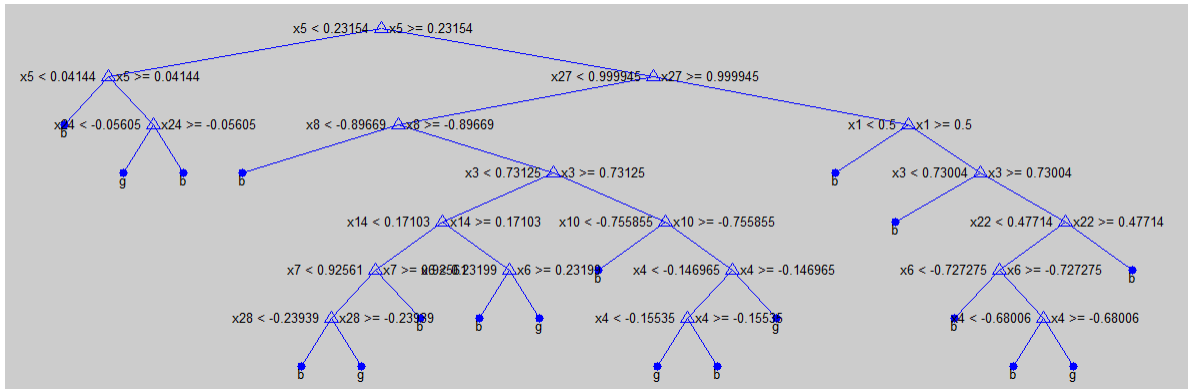
```
rng('default')
N = numel(leafs);
err = zeros(N,1);
for n=1:N
    t = ClassificationTree.fit(X,Y,'crossval','on',...
        'minleaf',leafs(n));
    err(n) = kfoldLoss(t);
end
plot(leafs,err);
xlabel('Min Leaf Size');
ylabel('cross-validated error');
```



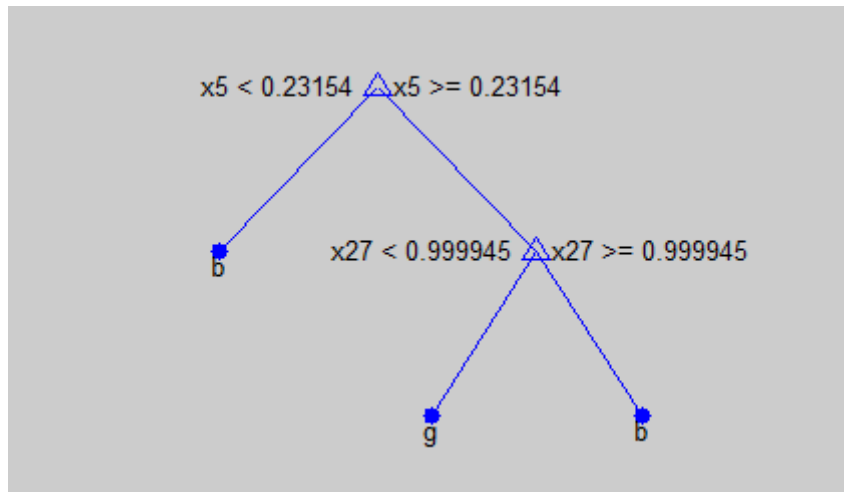
The best leaf size is between about 20 and 50 observations per leaf.

- 4 Compare the near-optimal tree with at least 40 observations per leaf with the default tree, which uses 10 observations per parent node and 1 observation per leaf.

```
DefaultTree = ClassificationTree.fit(X,Y);  
view(DefaultTree,'mode','graph')
```



```
OptimalTree = ClassificationTree.fit(X,Y,'minleaf',40);
view(OptimalTree,'mode','graph')
```



```
resubOpt = resubLoss(OptimalTree);
lossOpt = kfoldLoss(crossval(OptimalTree));
resubDefault = resubLoss(DefaultTree);
lossDefault = kfoldLoss(crossval(DefaultTree));
resubOpt,resubDefault,lossOpt,lossDefault
```

```
resubOpt =
```

```
0.0883
resubDefault =
0.0114
lossOpt =
0.1054
lossDefault =
0.1111
```

The near-optimal tree is much smaller and gives a much higher resubstitution error. Yet it gives similar accuracy for cross-validated data.

### **Pruning**

Pruning optimizes tree depth (leafiness) is by merging leaves on the same tree branch. “Control Depth or “Leafiness”” on page 16-40 describes one method for selecting the optimal depth for a tree. Unlike in that section, you do not need to grow a new tree for every node size. Instead, grow a deep tree, and prune it to the level you choose.

Prune a tree at the command line using the `prune` method (classification) or `prune` method (regression). Alternatively, prune a tree interactively with the tree viewer:

```
view(tree, 'mode', 'graph')
```

To prune a tree, the tree must contain a pruning sequence. By default, both `ClassificationTree.fit` and `RegressionTree.fit` calculate a pruning sequence for a tree during construction. If you construct a tree with the 'Prune' name-value pair set to 'off', or if you prune a tree to a smaller level, the tree does not contain the full pruning sequence. Generate the full pruning sequence with the `prune` method (classification) or `prune` method (regression).

**Example: Pruning a Classification Tree.** This example creates a classification tree for the ionosphere data, and prunes it to a good level.

**1** Load the ionosphere data:

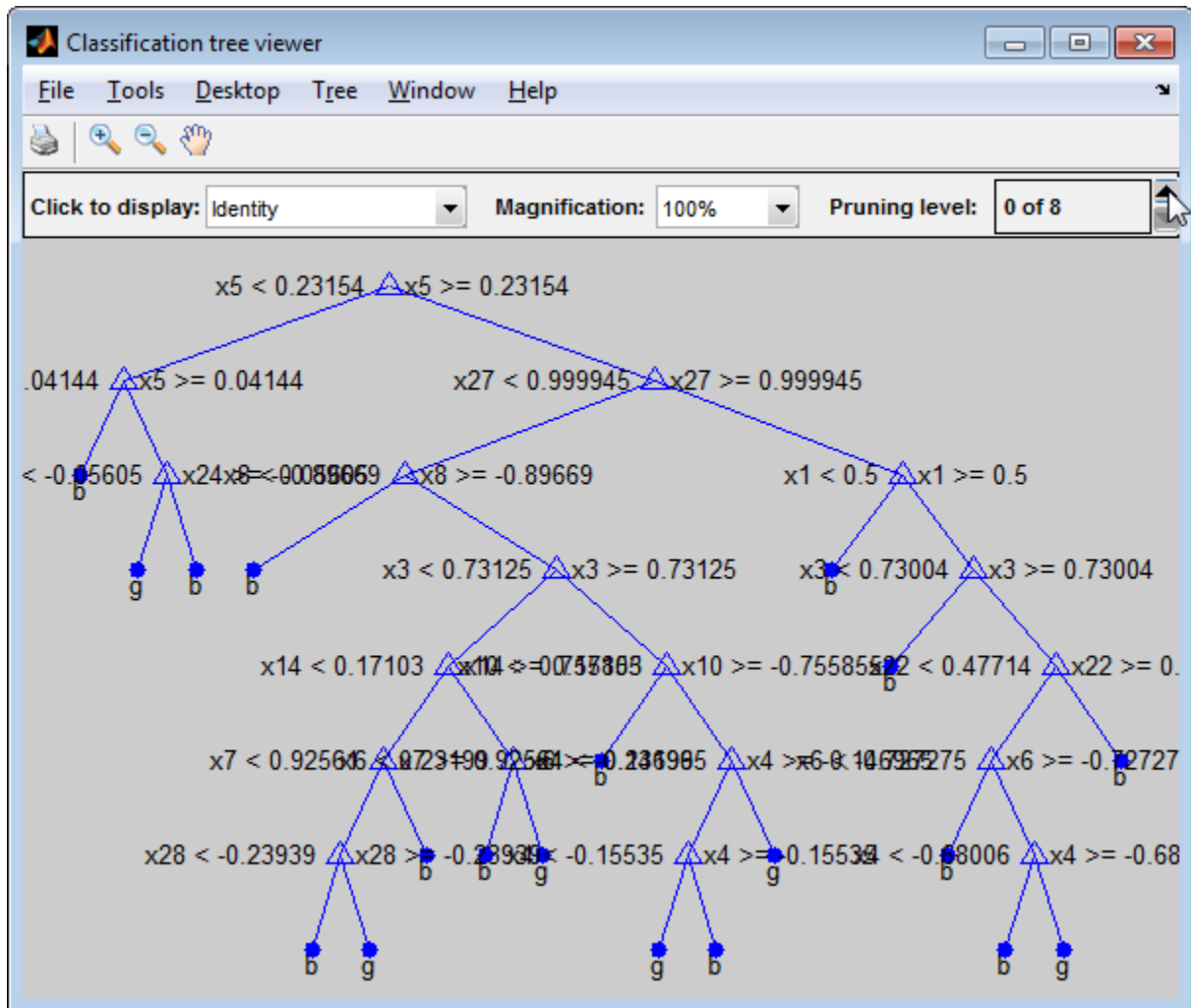
```
load ionosphere
```

**2** Construct a default classification tree for the data:

```
tree = ClassificationTree.fit(X,Y);
```

**3** View the tree in the interactive viewer:

```
view(tree, 'mode', 'graph')
```



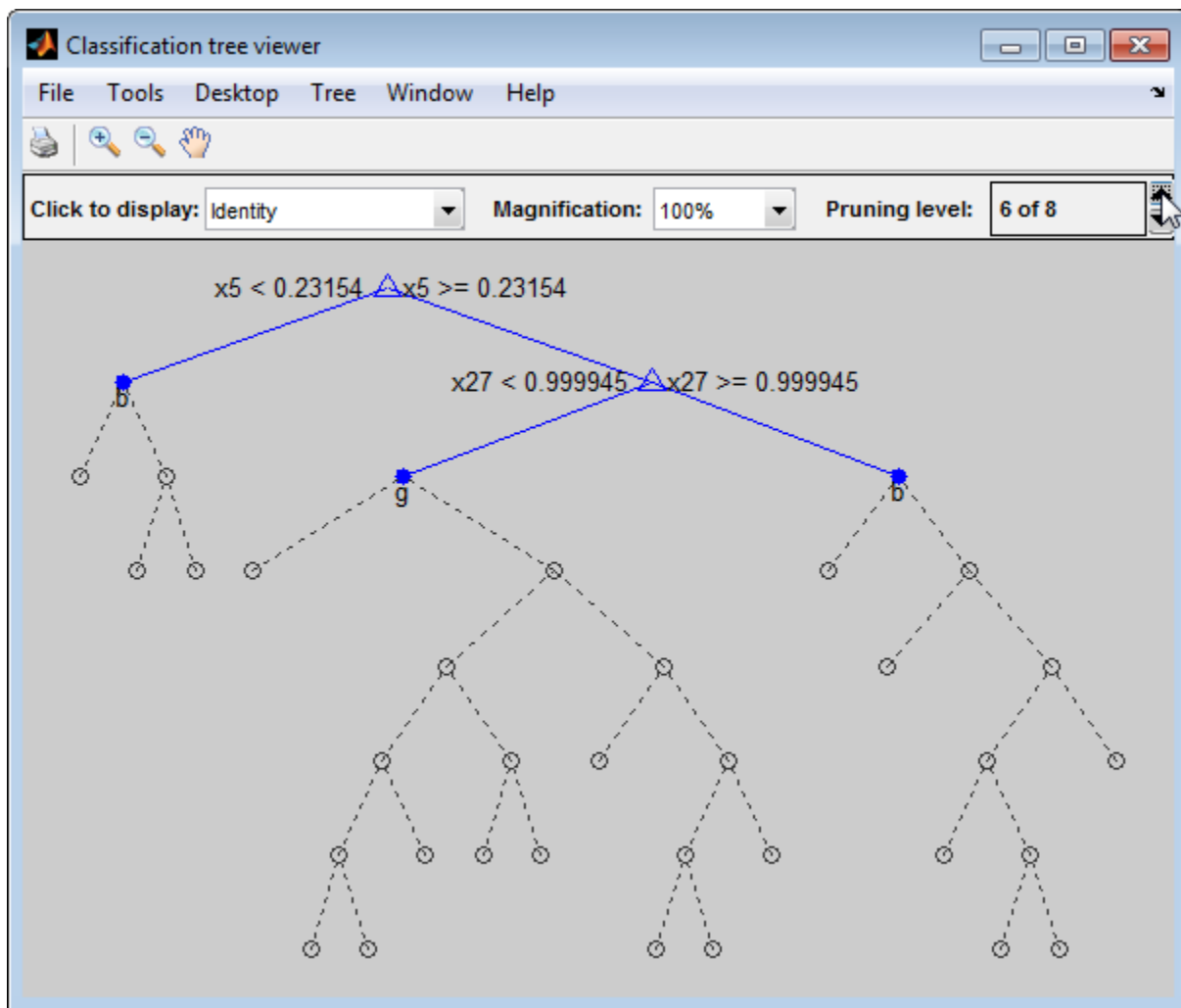
4 Find the optimal pruning level by minimizing cross-validated loss:

```
[~,~,~,bestlevel] = cvLoss(tree,...
    'subtrees','all','treesize','min')
```

```
bestlevel =
```

6

5 Prune the tree to level 6 in the interactive viewer:



The pruned tree is the same as the near-optimal tree in “Example: Selecting Appropriate Tree Depth” on page 16-41.

- 6 Set 'treesize' to 'se' (default) to find the maximal pruning level for which the tree error does not exceed the error from the best level plus one standard deviation:

```
[~,~,~,bestlevel] = cvLoss(tree,'subtrees','all')
```

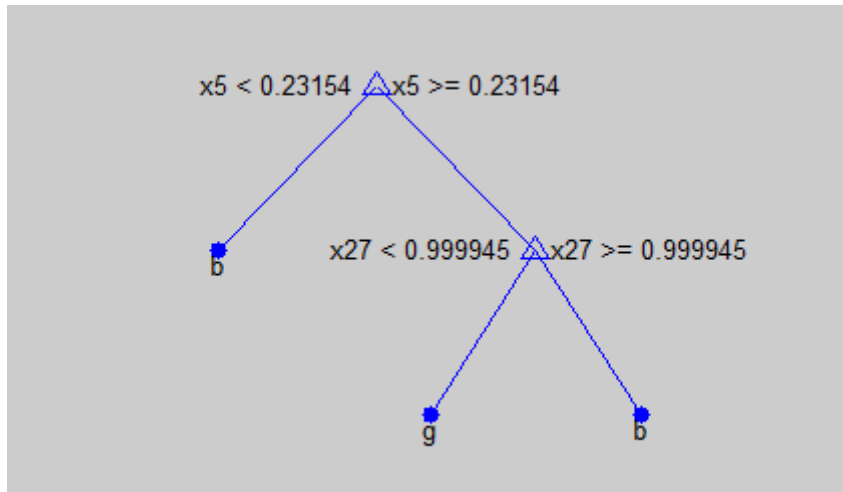
```
bestlevel =
```

```
6
```

In this case the level is the same for either setting of 'treesize'.

- 7 Prune the tree to use it for other purposes:

```
tree = prune(tree,'Level',6);  
view(tree,'mode','graph')
```



### Alternative: classregtree

The ClassificationTree and RegressionTree classes are new in MATLAB R2011a. Previously, you represented both classification trees and regression trees with a classregtree object. The new classes provide all the functionality of the classregtree class, and are more convenient when used in conjunction with “Ensemble Methods” on page 16-59.



Before the `classregtree` class, there were `treefit`, `treedisp`, `treeval`, `treeprune`, and `treetest` functions. Statistics Toolbox software maintains these only for backward compatibility.

### Example: Creating Classification Trees Using `classregtree`

This example uses Fisher's iris data in `fisheriris.mat` to create a classification tree for predicting species using measurements of sepal length, sepal width, petal length, and petal width as predictors. Here, the predictors are continuous and the response is categorical.

- 1 Load the data and use the `classregtree` constructor of the `classregtree` class to create the classification tree:

```
load fisheriris

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})

t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

`t` is a `classregtree` object and can be operated on with any class method.

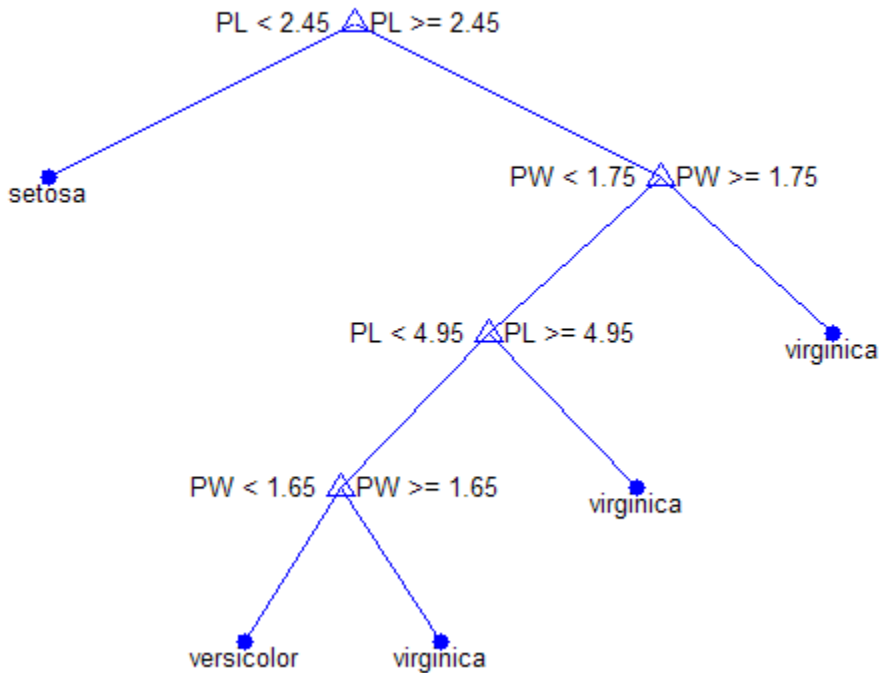
- 2 Use the `type` method of the `classregtree` class to show the type of the tree:

```
treetype = type(t)
treetype =
classification
```

`classregtree` creates a classification tree because `species` is a cell array of strings, and the response is assumed to be categorical.

**3** To view the tree, use the view method of the `classregtree` class:

```
view(t)
```



The tree predicts the response values at the circular leaf nodes based on a series of questions about the iris at the triangular branching nodes. A true answer to any question follows the branch to the left. A false follows the branch to the right.

**4** The tree does not use sepal measurements for predicting species. These can go unmeasured in new data, and you can enter them as NaN values for predictions. For example, to use the tree to predict the species of an iris with petal length 4.8 and petal width 1.6, type:

```

predicted = t([NaN NaN 4.8 1.6])
predicted =
    'versicolor'

```

The object allows for functional evaluation, of the form  $t(X)$ . This is a shorthand way of calling the `eval` method of the `classregtree` class. The predicted species is the left leaf node at the bottom of the tree in the previous view.

- 5** You can use a variety of methods of the `classregtree` class, such as `cutvar` and `cuttype` to get more information about the split at node 6 that makes the final distinction between `versicolor` and `virginica`:

```

var6 = cutvar(t,6) % What variable determines the split?
var6 =
    'PW'

```

```

type6 = cuttype(t,6) % What type of split is it?
type6 =
    'continuous'

```

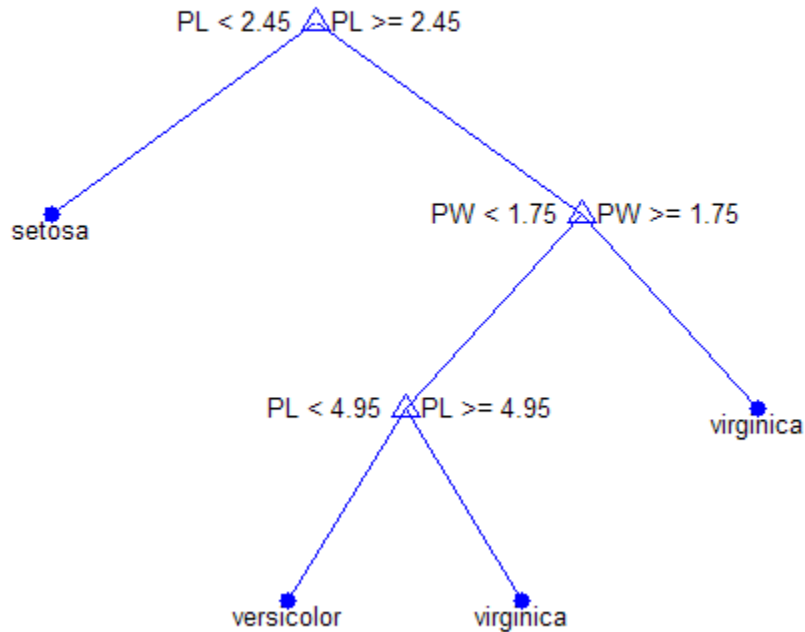
- 6** Classification trees fit the original (training) data well, but can do a poor job of classifying new values. Lower branches, especially, can be strongly affected by outliers. A simpler tree often avoids overfitting. You can use the `prune` method of the `classregtree` class to find the next largest tree from an optimal pruning sequence:

```

pruned = prune(t,'level',1)
pruned =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  class = versicolor
7  class = virginica

view(pruned)

```



To find the best classification tree, employing the techniques of resubstitution and cross validation, use the test method of the `classregtree` class.

### Example: Creating Regression Trees Using `classregtree`

This example uses the data on cars in `carsmall.mat` to create a regression tree for predicting mileage using measurements of weight and the number of cylinders as predictors. Here, one predictor (weight) is continuous and the other (cylinders) is categorical. The response (mileage) is continuous.

- 1 Load the data and use the `classregtree` constructor of the `classregtree` class to create the regression tree:

```

load carsmall

t = classregtree([Weight, Cylinders],MPG,...
                'cat',2,'splitmin',20,...
                'names',{'W','C'})

t =

Decision tree for regression
1  if W<3085.5 then node 2 elseif W>=3085.5 then node 3 else 23.7181
2  if W<2371 then node 4 elseif W>=2371 then node 5 else 28.7931
3  if C=8 then node 6 elseif C in {4 6} then node 7 else 15.5417
4  if W<2162 then node 8 elseif W>=2162 then node 9 else 32.0741
5  if C=6 then node 10 elseif C=4 then node 11 else 25.9355
6  if W<4381 then node 12 elseif W>=4381 then node 13 else 14.2963
7  fit = 19.2778
8  fit = 33.3056
9  fit = 29.6111
10 fit = 23.25
11 if W<2827.5 then node 14 elseif W>=2827.5 then node 15 else 27.2143
12 if W<3533.5 then node 16 elseif W>=3533.5 then node 17 else 14.8696
13 fit = 11
14 fit = 27.6389
15 fit = 24.6667
16 fit = 16.6
17 fit = 14.3889

```

t is a `classregtree` object and can be operated on with any of the methods of the class.

- 2** Use the `type` method of the `classregtree` class to show the type of the tree:

```

treetype = type(t)
treetype =
regression

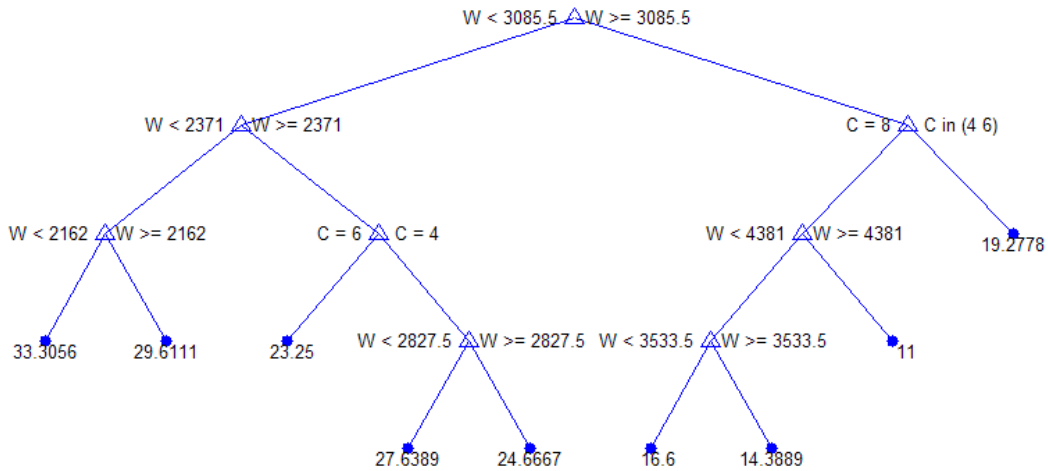
```

`classregtree` creates a regression tree because `MPG` is a numerical vector, and the response is assumed to be continuous.

- 3** To view the tree, use the `view` method of the `classregtree` class:

view(t)

Click to display:  Magnification:  Pruning level:



The tree predicts the response values at the circular leaf nodes based on a series of questions about the car at the triangular branching nodes. A true answer to any question follows the branch to the left; a false follows the branch to the right.

- 4 Use the tree to predict the mileage for a 2000-pound car with either 4, 6, or 8 cylinders:

```

mileage2K = t([2000 4; 2000 6; 2000 8])
mileage2K =
  33.3056
  33.3056
  33.3056
  
```

The object allows for functional evaluation, of the form `t(X)`. This is a shorthand way of calling the `eval` method of the `classregtree` class.

- 5 The predicted responses computed above are all the same. This is because they follow a series of splits in the tree that depend only on weight, terminating

at the leftmost leaf node in the view above. A 4000-pound car, following the right branch from the top of the tree, leads to different predicted responses:

```
mileage4K = t([4000 4; 4000 6; 4000 8])
mileage4K =
    19.2778
    19.2778
    14.3889
```

- 6 You can use a variety of other methods of the `classregtree` class, such as `cutvar`, `cuttype`, and `cutcategories`, to get more information about the split at node 3 that distinguishes the 8-cylinder car:

```
var3 = cutvar(t,3) % What variable determines the split?
var3 =
    'C'

type3 = cuttype(t,3) % What type of split is it?
type3 =
    'categorical'

c = cutcategories(t,3) % Which classes are sent to the left
                        % child node, and which to the right?
c =
    [8]    [1x2 double]
c{1}
ans =
    8
c{2}
ans =
    4    6
```

Regression trees fit the original (training) data well, but may do a poor job of predicting new values. Lower branches, especially, may be strongly affected by outliers. A simpler tree often avoids overfitting. To find the best regression tree, employing the techniques of resubstitution and cross validation, use the `test` method of the `classregtree` class.

## Splitting Categorical Predictors

### In this section...

“Challenges in Splitting Multilevel Predictors” on page 16-56

“Pull Left By Purity” on page 16-57

“Principle Component-Based Partitioning” on page 16-57

“One Versus All By Class” on page 16-58

### Challenges in Splitting Multilevel Predictors

When growing a classification tree, finding an optimal binary split for a categorical predictor with many levels is significantly more computationally challenging than finding a split for a continuous predictor. For a continuous predictor, a tree can split halfway between any two adjacent unique values of this predictor.

In contrast, to find an exact optimal binary split for a categorical predictor with  $L$  levels, a classification tree needs to consider  $2^{L-1}-1$  splits. To obtain this formula, observe that you can assign  $L$  distinct values to the left and right nodes in  $2^L$  ways. Two out of these  $2^L$  configurations leave either the left or right node empty, and therefore should be discarded. Now, divide by 2 because left and right can be swapped.

For regression and binary classification problems, with  $K = 2$  response classes, there is a computational shortcut [1]. The tree can order the categories by mean response (for regression) or class probability for one of the classes (for classification). Then, the optimal split is one of the  $L - 1$  splits for the ordered list. When  $K = 2$ , `ClassificationTree.fit` always uses an exact search.

Therefore, computational challenges really only arise when growing classification trees for data with  $K \geq 3$  classes. To reduce computation, there are several heuristic algorithms for finding a good split. When using `ClassificationTree.fit` to grow a classification tree, you can choose an algorithm for splitting categorical predictors using the `AlgorithmForCategorical` name-value pair argument. You can also set this algorithm when creating a classification template.



If you do not specify an algorithm, `ClassificationTree.fit` splits categorical predictors using the exact search algorithm, provided the predictor has at most `MaxCat` levels (the default is 10 levels, and, depending on your platform, you cannot perform an exact search on categorical predictors with more than 32 or 64 levels). Otherwise, `ClassificationTree.fit` chooses a good inexact search algorithm based on the number of classes and levels.

The available heuristic algorithms are: pull left by purity, a principle component-based partitioning, and one versus all by class.

### **Pull Left By Purity**

This algorithm starts with all  $L$  categorical levels on the right branch. Inspect the  $K$  categories that have the largest class probabilities for each class. Move the category with the maximum value of the split criterion to the left branch. Continue moving categories from right to left, recording the split criterion at each move, until the right child has only one category remaining. Out of this sequence, the chosen split is the one that maximizes the split criterion.

Select this pull left by purity algorithm by using the 'AlgorithmForCategorical', 'PullLeft' name-value pair in `ClassificationTree.fit`.

### **Principle Component-Based Partitioning**

This algorithm was developed by Coppersmith, Hong, and Hosking [2]. It finds a close-to-optimal binary partition of the  $L$  predictor levels by searching for a separating hyperplane that is perpendicular to the first principal component of the weighted covariance matrix of the centered class probability matrix.

The algorithm assigns a score to each of the  $L$  categories, computed as the inner product between the found principal component and the vector of class probabilities for that category. Then, the chosen split is the one of the  $L - 1$  splits of the scores that maximizes the split criterion.

Select this principle component-based partitioning by using the 'AlgorithmForCategorical', 'PCA' name-value pair in `ClassificationTree.fit`.

## One Versus All By Class

This algorithm starts with all  $L$  categorical levels on the right branch. For each of the  $K$  classes, order the categories based on their probability for that class.

For the first class, move each category to the left branch in order, recording the split criterion at each move. Repeat for the remaining classes. Out of this sequence, the chosen split is the one that maximizes the split criterion.

Select this one versus all by class algorithm by using the 'AlgorithmForCategorical', 'OVAByClass' name-value pair in `ClassificationTree.fit`.

## References

[1] Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman & Hall, Boca Raton, 1993.

[2] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. "Partitioning Nominal Attributes in Decision Trees." *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.

## See Also

`ClassificationTree.fit` | `template` | `RegressionTree.fit` |

## Related Examples

- "How the Fit Methods Create Trees" on page 16-36

## Concepts

- "What Are Classification Trees and Regression Trees?" on page 16-31

## Ensemble Methods

### In this section...

“Framework for Ensemble Learning” on page 16-59  
“Basic Ensemble Examples” on page 16-69  
“Test Ensemble Quality” on page 16-71  
“Classification with Imbalanced Data” on page 16-77  
“Classification: Imbalanced Data or Unequal Misclassification Costs” on page 16-83  
“Classification with Many Categorical Levels” on page 16-91  
“Surrogate Splits” on page 16-95  
“LPBoost and TotalBoost for Small Ensembles” on page 16-98  
“Ensemble Regularization” on page 16-103  
“Tuning RobustBoost” on page 16-115  
“Random Subspace Classification” on page 16-119  
“TreeBagger Examples” on page 16-125  
“Ensemble Algorithms” on page 16-146

### Framework for Ensemble Learning

You have several methods for melding results from many weak learners into one high-quality ensemble predictor. These methods closely follow the same syntax, so you can try different methods with minor changes in your commands.

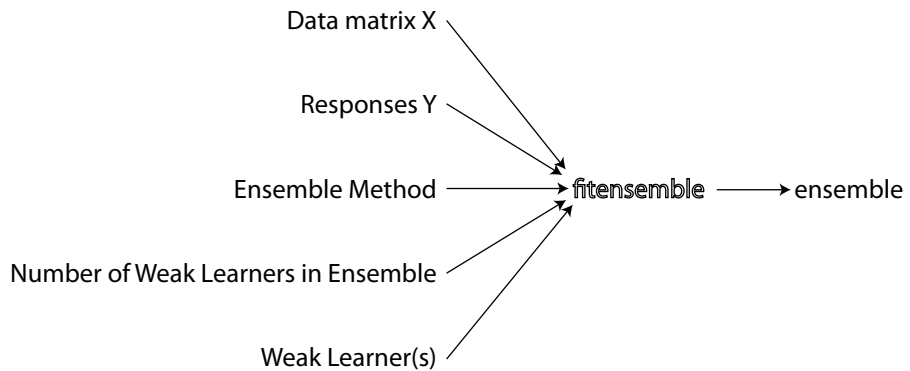
Create an ensemble with the `fitensemble` function. Its syntax is

```
ens = fitensemble(X,Y,model,numberens,learners)
```

- X is the matrix of data. Each row contains one observation, and each column contains one predictor variable.
- Y is the vector of responses, with the same number of observations as the rows in X.

- `model` is a string naming the type of ensemble.
- `numberens` is the number of weak learners in `ens` from each element of `learners`. So the number of elements in `ens` is `numberens` times the number of elements in `learners`.
- `learners` is either a string naming a weak learner, a weak learner template, or a cell array of such templates.

Pictorially, here is the information you need to create an ensemble:



For all classification or nonlinear regression problems, follow these steps to create an ensemble:

- 1** “Put Predictor Data in a Matrix” on page 16-61
- 2** “Prepare the Response Data” on page 16-61
- 3** “Choose an Applicable Ensemble Method” on page 16-62
- 4** “Set the Number of Ensemble Members” on page 16-65
- 5** “Prepare the Weak Learners” on page 16-66
- 6** “Call `fitensemble`” on page 16-68

## Put Predictor Data in a Matrix

All supervised learning methods start with a data matrix, usually called  $X$  in this documentation. Each row of  $X$  represents one observation. Each column of  $X$  represents one variable, or predictor.

## Prepare the Response Data

You can use a wide variety of data types for response data.

- For regression ensembles,  $Y$  must be a numeric vector with the same number of elements as the number of rows of  $X$ .
- For classification ensembles,  $Y$  can be any of the following data types. This table also contains the method of including missing entries.

Data Type	Missing Entry
Numeric vector	NaN
Categorical vector	<undefined>
Character array	Row of spaces
Cell array of strings	' '
Logical vector	(not possible to represent)

`fitensemble` ignores missing values in  $Y$  when creating an ensemble.

For example, suppose your response data consists of three observations in the following order: `true`, `false`, `true`. You could express  $Y$  as:

- `[1;0;1]` (numeric vector)
- `nominal({'true','false','true'})` (categorical vector)
- `[true;false;true]` (logical vector)
- `['true ','false ','true ']` (character array, padded with spaces so each row has the same length)
- `{'true','false','true'}` (cell array of strings)

Use whichever data type is most convenient. Because you cannot represent missing values with logical entries, do not use logical entries when you have missing values in  $Y$ .

### **Choose an Applicable Ensemble Method**

`fitensemble` uses one of these algorithms to create an ensemble.

- For classification with two classes:
  - 'AdaBoostM1'
  - 'LogitBoost'
  - 'GentleBoost'
  - 'RobustBoost' (requires an Optimization Toolbox license)
  - 'LPBoost' (requires an Optimization Toolbox license)
  - 'TotalBoost' (requires an Optimization Toolbox license)
  - 'RUSBoost'
  - 'Subspace'
  - 'Bag'
- For classification with three or more classes:
  - 'AdaBoostM2'
  - 'LPBoost' (requires an Optimization Toolbox license)
  - 'TotalBoost' (requires an Optimization Toolbox license)
  - 'RUSBoost'
  - 'Subspace'
  - 'Bag'
- For regression:
  - 'LSBoost'
  - 'Bag'

'Bag' applies to all methods. When using 'Bag', indicate whether you want a classifier or regressor with the type name-value pair set to 'classification' or 'regression'.

For descriptions of the various algorithms, see “Ensemble Algorithms” on page 16-146.

See “Suggestions for Choosing an Appropriate Ensemble Algorithm” on page 16-64.

This table lists characteristics of the various algorithms. In the table titles:

- **Regress.** — Regression
- **Classif.** — Classification
- **Preds.** — Predictors
- **Imbalance** — Good for imbalanced data (one class has many more observations than the other)
- **Stop** — Algorithm self-terminates
- **Sparse** — Requires fewer weak learners than other ensemble algorithms

<b>Algorithm</b>	<b>Regress.</b>	<b>Binary Classif.</b>	<b>Binary Classif. Multi- Level Preds.</b>	<b>Classif. 3+ Classes</b>	<b>Class Imbalance</b>	<b>Stop</b>	<b>Sparse</b>
Bag	×	×		×			
AdaBoostM1		×					
AdaBoostM2				×			
LogitBoost		×	×				
GentleBoost		×	×				
RobustBoost		×					
LPBoost		×		×		×	×
TotalBoost		×		×		×	×

Algorithm	Regress.	Binary Classif.	Binary Classif. Multi-Level Preds.	Classif. 3+ Classes	Class Imbalance	Stop	Sparse
RUSBoost		×		×	×		
LSBoost	×						
Subspace		×		×			

RobustBoost, LPBoost, and TotalBoost require an Optimization Toolbox license. Try TotalBoost before LPBoost, as TotalBoost can be more robust.

### Suggestions for Choosing an Appropriate Ensemble Algorithm.

- **Regression** — Your choices are LSBoost or Bag. See “General Characteristics of Ensemble Algorithms” on page 16-65 for the main differences between boosting and bagging.
- **Binary Classification** — Try AdaBoostM1 first, with these modifications:

Data Characteristic	Recommended Algorithm
Many predictors	Subspace
Skewed data (many more observations of one class)	RUSBoost
Categorical predictors with over 31 levels	LogitBoost or GentleBoost
Label noise (some training data has the wrong class)	RobustBoost
Many observations	Avoid LPBoost, TotalBoost, and Bag

- **Multiclass Classification** — Try AdaBoostM2 first, with these modifications:



<b>Data Characteristic</b>	<b>Recommended Algorithm</b>
Many predictors	Subspace
Skewed data (many more observations of one class)	RUSBoost
Many observations	Avoid LPBoost, TotalBoost, and Bag

For details of the algorithms, see “Ensemble Algorithms” on page 16-146.

### **General Characteristics of Ensemble Algorithms.**

- Bag generally constructs deep trees. This construction is both time consuming and memory-intensive. This also leads to relatively slow predictions.
- Boost algorithms generally use very shallow trees. This construction uses relatively little time or memory. However, for effective predictions, boosted trees might need more ensemble members than bagged trees. Therefore it is not always clear which class of algorithms is superior.
- Bag can estimate the generalization error without additional cross validation. See `oobLoss`.
- Except for Subspace, all boosting and bagging algorithms are based on tree learners. Subspace can use either discriminant analysis or  $k$ -nearest neighbor learners.

For details of the characteristics of individual ensemble members, see “Characteristics of Algorithms” on page 16-7.

### **Set the Number of Ensemble Members**

Choosing the size of an ensemble involves balancing speed and accuracy.

- Larger ensembles take longer to train and to generate predictions.
- Some ensemble algorithms can become overtrained (inaccurate) when too large.

To set an appropriate size, consider starting with several dozen to several hundred members in an ensemble, training the ensemble, and then checking the ensemble quality, as in “Example: Test Ensemble Quality” on page 16-72. If it appears that you need more members, add them using the `resume` method (classification) or the `resume` method (regression). Repeat until adding more members does not improve ensemble quality.

---

**Tip** For classification, the `LPBoost` and `TotalBoost` algorithms are self-terminating, meaning you do not have to investigate the appropriate ensemble size. Try setting `numberens` to 500. The algorithms usually terminate with fewer members.

---

## Prepare the Weak Learners

Currently the weak learner types are:

- 'Discriminant' (for Subspace ensemble)
- 'KNN' (for Subspace ensemble)
- 'Tree' (for any ensemble except Subspace)

There are two ways to set the weak learner type in the ensemble:

- To create an ensemble with default weak learner options, pass in the string as the weak learner. For example,

```
ens = fitensemble(X,Y,'AdaBoostM2',50,'Tree');  
% or  
ens = fitensemble(X,Y,'Subspace',50,'KNN');
```

- To create an ensemble with nondefault weak learner options, create a nondefault weak learner using the appropriate `template` method. For example, if you have missing data, and want to use trees with surrogate splits for better accuracy:

```
templ = ClassificationTree.template('Surrogate','all');  
ens = fitensemble(X,Y,'AdaBoostM2',50,templ);
```

To grow trees with roughly 10 leaf nodes per tree:

```
templ = ClassificationTree.template('MinLeaf',size(X,1)/10);
ens = fitensemble(X,Y,'AdaBoostM2',50,templ);
```

While you can give `fitensemble` a cell array of learner templates, the most common usage is to give just one weak learner template.

For examples using a template, see “Example: Unequal Classification Costs” on page 16-85 and “Surrogate Splits” on page 16-95.

Decision trees can handle NaN values in  $X$ . Such values are called “missing.” If you have some missing values in a row of  $X$ , a decision tree finds optimal splits using nonmissing values only. If an entire row consists of NaN, `fitensemble` ignores that row. If you have data with a large fraction of missing values in  $X$ , use surrogate decision splits. For examples of surrogate splits, see “Example: Unequal Classification Costs” on page 16-85 and “Surrogate Splits” on page 16-95.

### Common Settings for Tree Weak Learners.

- The depth of a weak learner tree makes a difference for training time, memory usage, and predictive accuracy. You control the depth with two parameters:
  - `MinLeaf` — Each leaf has at least `MinLeaf` observations. Set small values of `MinLeaf` to get a deep tree.
  - `MinParent` — Each branch node in the tree has at least `MinParent` observations. Set small values of `MinParent` to get a deep tree.

If you supply both `MinParent` and `MinLeaf`, the learner uses the setting that gives larger leaves:

$$\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$$

- `Surrogate` — Grow decision trees with surrogate splits when `Surrogate` is 'on'. Use surrogate splits when your data has missing values.

---

**Note** Surrogate splits cause training to be slower and use more memory.

---

### Call `fitensemble`

The syntax of `fitensemble` is

```
ens = fitensemble(X,Y,model,numberens,learners)
```

- `X` is the matrix of data. Each row contains one observation, and each column contains one predictor variable.
- `Y` is the responses, with the same number of observations as rows in `X`.
- `model` is a string naming the type of ensemble.
- `numberens` is the number of weak learners in `ens` from each element of `learners`. So the number of elements in `ens` is `numberens` times the number of elements in `learners`.
- `learners` is a string naming a weak learner, a weak learner template, or a cell array of such strings and templates.

The result of `fitensemble` is an ensemble object, suitable for making predictions on new data. For a basic example of creating a classification ensemble, see “Create a Classification Ensemble” on page 16-69. For a basic example of creating a regression ensemble, see “Creating a Regression Ensemble” on page 16-70.

**Where to Set Name-Value Pairs.** There are several name-value pairs you can pass to `fitensemble`, and several that apply to the weak learners (`ClassificationDiscriminant.template`, `ClassificationKNN.template`, `ClassificationTree.template` and `RegressionTree.template`). To determine which option (name-value pair) is appropriate, the ensemble or the weak learner:

- Use template name-value pairs to control the characteristics of the weak learners.
- Use `fitensemble` name-value pairs to control the ensemble as a whole, either for algorithms or for structure.

For example, to have an ensemble of boosted classification trees with each tree deeper than the default, set the `ClassificationTree.template` name-value pairs (`MinLeaf` and `MinParent`) to smaller values than the defaults. This causes the trees to be leafier (deeper).

To name the predictors in the ensemble (part of the structure of the ensemble), use the `PredictorNames` name-value pair in `fitensemble`.

## Basic Ensemble Examples

### Create a Classification Ensemble

Create a classification tree ensemble for the Fisher iris data, and use it to predict the classification of a flower with average measurements.

- 1 Load the data:

```
load fisheriris
```

- 2 The predictor data `X` is the `meas` matrix.

- 3 The response data `Y` is the `species` cell array.

- 4 For classification trees with three or more classes, “Suggestions for Choosing an Appropriate Ensemble Algorithm” on page 16-64 suggests using an `'AdaBoostM2'` ensemble.

- 5 For this example, arbitrarily take an ensemble of 100 trees.

- 6 Use a default tree template.

- 7 Create the ensemble:

```
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree')
```

```
ens =
```

```
classreg.learning.classif.ClassificationEnsemble
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}
  ResponseName: 'Y'
  ClassNames: {'setosa' 'versicolor' 'virginica'}
  ScoreTransform: 'none'
  NObservations: 150
  NTrained: 100
  Method: 'AdaBoostM2'
  LearnerNames: {'Tree'}
```

```
ReasonForTermination: [1x77 char]
                   FitInfo: [100x1 double]
                   FitInfoDescription: {2x1 cell}
```

Properties, Methods

- 8** Predict the classification of a flower with average measurements:

```
flower = predict(ens,mean(meas))

flower =

    'versicolor'
```

### **Creating a Regression Ensemble**

Create a regression ensemble to predict mileage of cars based on their horsepower and weight, trained on the `carsmall` data. Use the resulting ensemble to predict the mileage of a car with 150 horsepower weighing 2750 lbs.

- 1** Load the data:

```
load carsmall
```

- 2** Prepare the input data.

```
X = [Horsepower Weight];
```

- 3** The response data `Y` is MPG.

- 4** The only boosted regression ensemble type is `'LSBoost'`.

- 5** For this example, arbitrarily take an ensemble of 100 trees.

- 6** Use a default tree template.

- 7** Create the ensemble:

```
ens = fitensemble(X,MPG,'LSBoost',100,'Tree')

ens =
```

```

classreg.learning.regr.RegressionEnsemble
    PredictorNames: {'x1' 'x2'}
    ResponseName: 'Y'
    ResponseTransform: 'none'
    NObservations: 94
    NTrained: 100
    Method: 'LSBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: [1x77 char]
    FitInfo: [100x1 double]
    FitInfoDescription: {2x1 cell}
    Regularization: []

```

Properties, Methods

- 8** Predict the mileage of a car with 150 horsepower weighing 2750 lbs:

```
mileage = ens.predict([150 2750])
```

```
mileage =
```

```
22.4180
```

## Test Ensemble Quality

Usually you cannot evaluate the predictive quality of an ensemble based on its performance on training data. Ensembles tend to “overtrain,” meaning they produce overly optimistic estimates of their predictive power. This means the result of `resubLoss` for classification (`resubLoss` for regression) usually indicates lower error than you get on new data.

To obtain a better idea of the quality of an ensemble, use one of these methods:

- Evaluate the ensemble on an independent test set (useful when you have a lot of training data).
- Evaluate the ensemble by cross validation (useful when you don’t have a lot of training data).

- Evaluate the ensemble on out-of-bag data (useful when you create a bagged ensemble with `fitensemble`).

### Example: Test Ensemble Quality

This example uses a bagged ensemble so it can use all three methods of evaluating ensemble quality.

- 1 Generate an artificial dataset with 20 predictors. Each entry is a random number from 0 to 1. The initial classification:

$$Y = 1 \text{ when } X(1) + X(2) + X(3) + X(4) + X(5) > 2.5$$
$$Y = 0 \text{ otherwise.}$$

```
rng(1,'twister') % for reproducibility
X = rand(2000,20);
Y = sum(X(:,1:5),2) > 2.5;
```

In addition, to add noise to the results, randomly switch 10% of the classifications:

```
idx = randsample(2000,200);
Y(idx) = ~Y(idx);
```

### 2 Independent Test Set

Create independent training and test sets of data. Use 70% of the data for a training set by calling `cvpartition` using the `holdout` option:

```
cvpart = cvpartition(Y,'holdout',0.3);
Xtrain = X(training(cvpart),:);
Ytrain = Y(training(cvpart),:);
Xtest = X(test(cvpart),:);
Ytest = Y(test(cvpart),:);
```

- 3 Create a bagged classification ensemble of 200 trees from the training data:

```
bag = fitensemble(Xtrain,Ytrain,'Bag',200,'Tree',...
    'type','classification')

bag =
```

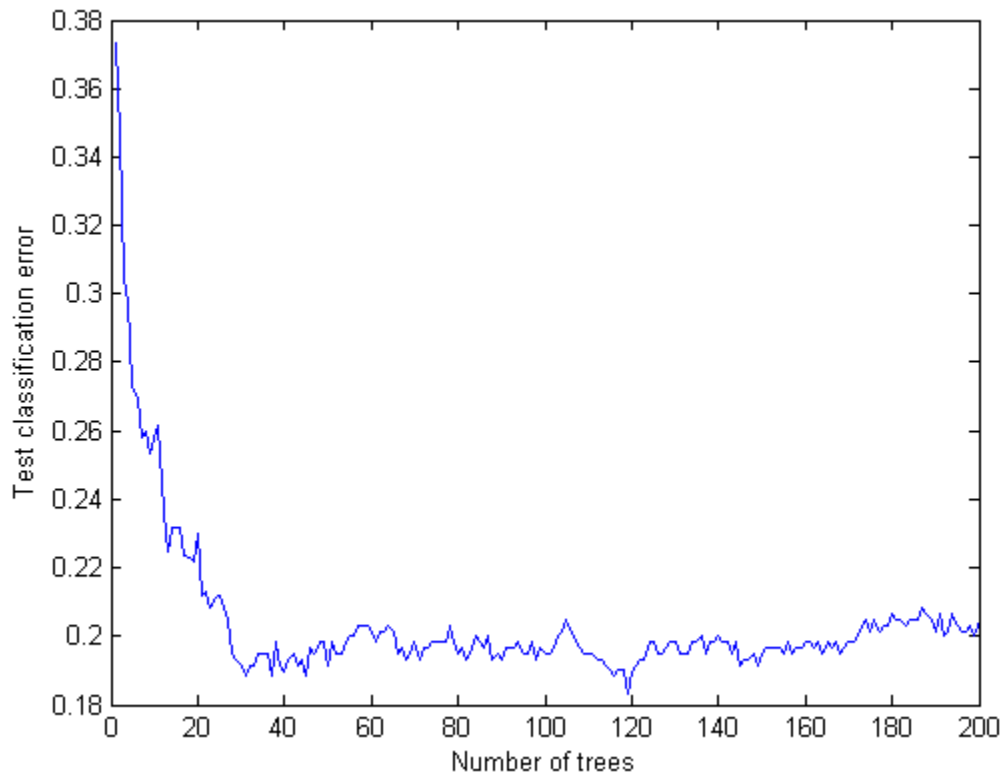


```
classreg.learning.classif.ClassificationBaggedEnsemble
    PredictorNames: {1x20 cell}
    ResponseName: 'Y'
    ClassNames: [0 1]
    ScoreTransform: 'none'
    NObservations: 1400
    NTrained: 200
    Method: 'Bag'
    LearnerNames: {'Tree'}
    ReasonForTermination: [1x77 char]
    FitInfo: []
    FitInfoDescription: 'None'
    FResample: 1
    Replace: 1
    UseObsForLearner: [1400x200 logical]
```

Properties, Methods

- 4 Plot the loss (misclassification) of the test data as a function of the number of trained trees in the ensemble:

```
figure;
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
xlabel('Number of trees');
ylabel('Test classification error');
```



### 5 Cross Validation

Generate a five-fold cross-validated bagged ensemble:

```
cv = fitensemble(X,Y,'Bag',200,'Tree',...
    'type','classification','kfold',5)
```

```
cv =
```

```
classreg.learning.partition.ClassificationPartitionedEnsemble
    CrossValidatedModel: 'Bag'
        PredictorNames: {1x20 cell}
    CategoricalPredictors: []
```

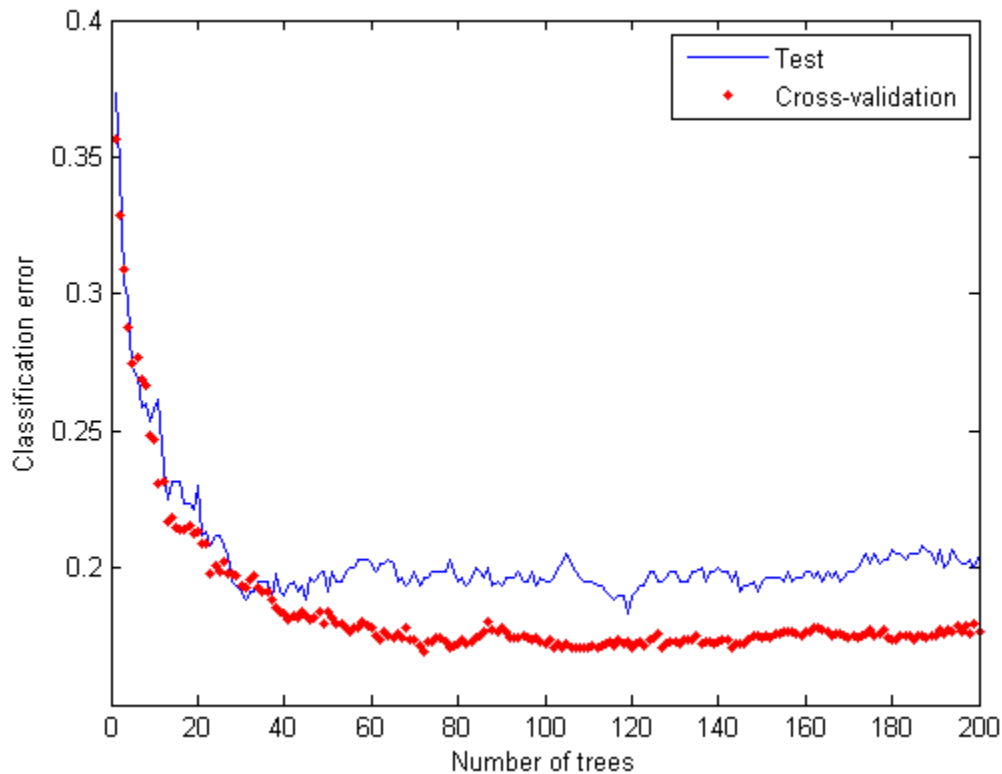
```
ResponseName: 'Y'  
NObservations: 2000  
KFold: 5  
Partition: [1x1 cvpartition]  
NTrainedPerFold: [200 200 200 200 200]  
ClassNames: [0 1]  
ScoreTransform: 'none'
```

### Properties, Methods

- 6 Examine the cross-validation loss as a function of the number of trees in the ensemble:

```
figure;  
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));  
hold on;  
plot(kfoldLoss(cv,'mode','cumulative'),'r.');
```

hold off;  
xlabel('Number of trees');  
ylabel('Classification error');  
legend('Test','Cross-validation','Location','NE');



Cross validating gives comparable estimates to those of the independent set.

### 7 Out-of-Bag Estimates

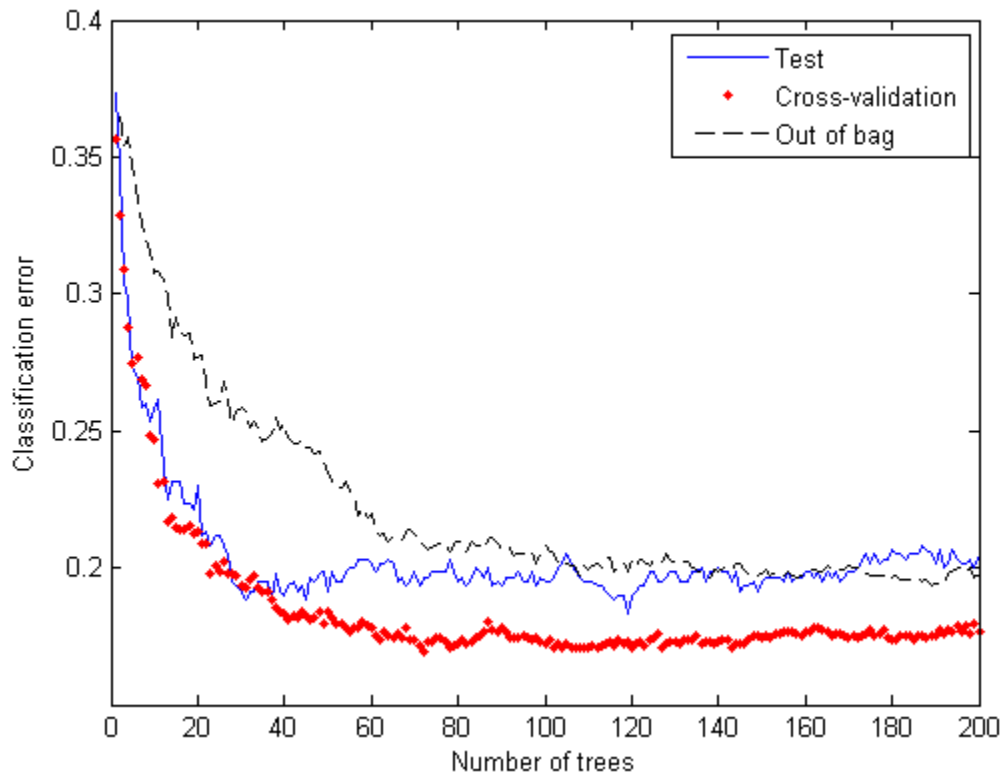
Generate the loss curve for out-of-bag estimates, and plot it along with the other curves:

```
figure;
plot(loss(bag,Xtest,Ytest,'mode','cumulative'));
hold on;
plot(kfoldLoss(cv,'mode','cumulative'),'r. ');
plot(oobLoss(bag,'mode','cumulative'),'k--');
```

```

hold off;
xlabel('Number of trees');
ylabel('Classification error');
legend('Test','Cross-validation','Out of bag','Location','NE');

```



The out-of-bag estimates are again comparable to those of the other methods.

## Classification with Imbalanced Data

This example shows how to classify when one class has many more observations than another. Try the RUSBoost algorithm first, because it is designed to handle this case.

This example uses the “Cover type” data from the UCI machine learning archive, described in <http://archive.ics.uci.edu/ml/datasets/Covertype>. The data classifies types of forest (ground cover), based on predictors such as elevation, soil type, and distance to water. The data has over 500,000 observations and over 50 predictors, so training and using a classifier is time consuming.

Blackard and Dean [1] describe a neural net classification of this data. They quote a 70.6% classification accuracy. RUSBoost obtains over 76% classification accuracy; see steps 6 and 7.

**Step 1. Obtain the data.**

```
urlwrite('http://archive.ics.uci.edu/ml/machine-learning-databases/covtype/
```

Then, extract the data from the `forestcover.gz` file. The data is in the `covtype.data` file.

**Step 2. Import the data and prepare it for classification.**

Import the data into your workspace. Extract the last data column into a variable named `Y`.

```
load covtype.data
Y = covtype(:,end);
covtype(:,end) = [];
```

**Step 3. Examine the response data.**

```
tabulate(Y)
```

Value	Count	Percent
1	211840	36.46%
2	283301	48.76%
3	35754	6.15%
4	2747	0.47%
5	9493	1.63%
6	17367	2.99%
7	20510	3.53%

There are hundreds of thousands of data points. Those of class 4 are less than 0.5% of the total. This imbalance indicates that RUSBoost is an appropriate algorithm.

#### Step 4. Partition the data for quality assessment.

Use half the data to fit a classifier, and half to examine the quality of the resulting classifier.

```
part = cvpartition(Y,'holdout',0.5);
istrain = training(part); % data for fitting
istest = test(part); % data for quality assessment
tabulate(Y(istrain))
```

Value	Count	Percent
1	105920	36.46%
2	141651	48.76%
3	17877	6.15%
4	1374	0.47%
5	4746	1.63%
6	8683	2.99%
7	10255	3.53%

#### Step 5. Create the ensemble.

Use deep trees for higher ensemble accuracy. To do so, set the trees to have minimal leaf size of 5. Set `LearnRate` to 0.1 in order to achieve higher accuracy as well. The data is large, and, with deep trees, creating the ensemble is time consuming.

```
t = ClassificationTree.template('minleaf',5);
tic
rusTree = fitensemble(covtype(istrain,:),Y(istrain),'RUSBoost',1000,t,...
    'LearnRate',0.1,'nprint',100);
toc
```

```
Training RUSBoost...
Grown weak learners: 100
Grown weak learners: 200
Grown weak learners: 300
Grown weak learners: 400
```

```
Grown weak learners: 500
Grown weak learners: 600
Grown weak learners: 700
Grown weak learners: 800
Grown weak learners: 900
Grown weak learners: 1000
Elapsed time is 918.258401 seconds.
```

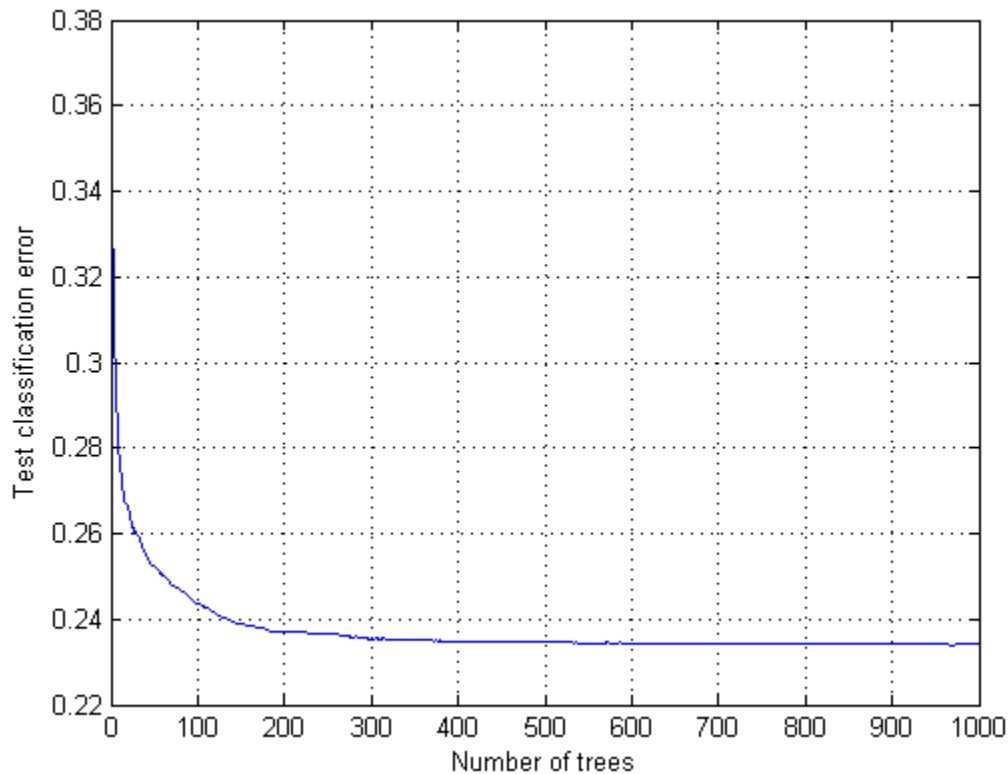
### **Step 6. Inspect the classification error.**

Plot the classification error against the number of members in the ensemble.

```
figure;
tic
plot(loss(rusTree,covtype(istest,:),Y(istest),'mode','cumulative'));
toc
grid on;
xlabel('Number of trees');
ylabel('Test classification error');
```

```
Elapsed time is 775.646935 seconds.
```





The ensemble achieves a classification error of under 24% using 150 or more trees. It achieves the lowest error for 400 or more trees.

Examine the confusion matrix for each class as a percentage of the true class.

```
tic
Yfit = predict(rusTree,covtype(istest,:));
toc
tab = tabulate(Y(istest));
bsxfun(@rdivide,confusionmat(Y(istest),Yfit),tab(:,2))*100
```

Elapsed time is 427.293168 seconds.

```
ans =

Columns 1 through 6

83.3771    7.4056    0.0736         0    1.7051    0.2681
18.3156   66.4652    2.1193    0.0162    9.3435    2.8239
         0    0.0839   90.8038    2.3885    0.6545    6.0693
         0         0    2.4763   95.8485         0    1.6752
         0    0.2739    0.6530         0   98.6518    0.4213
         0    0.1036    3.8346    1.1400    0.4030   94.5187
0.2340         0         0         0    0.0195         0

Column 7

7.1705
0.9163
0
0
0
0
99.7465
```

All classes except class 2 have over 80% classification accuracy, and classes 3 through 7 have over 90% accuracy. But class 2 makes up close to half the data, so the overall accuracy is not that high.

### Step 7. Compact the ensemble.

The ensemble is large. Remove the data using the compact method.

```
cmpctRus = compact(rusTree);
```

```
sz(1) = whos('rusTree');
sz(2) = whos('cmpctRus');
[sz(1).bytes sz(2).bytes]
```

```
ans =

1.0e+09 *

1.6947    0.9790
```

The compacted ensemble is about half the size of the original.

Remove half the trees from `cmpctRus`. This action is likely to have minimal effect on the predictive performance, based on the observation that 400 out of 1000 trees give nearly optimal accuracy.

```
cmpctRus = removeLearners(cmpctRus, [500:1000]);
```

```
sz(3) = whos('cmpctRus');
sz(3).bytes
```

```
ans =
```

```
475495669
```

The reduced compact ensemble takes about a quarter the memory of the full ensemble. Its overall loss rate is under 24%:

```
L = loss(cmpctRus, covtype(istest,:), Y(istest))
```

```
L =
```

```
0.2326
```

The predictive accuracy on new data might differ, because the ensemble accuracy might be biased. The bias arises because the same data used for assessing the ensemble was used for reducing the ensemble size. To obtain an unbiased estimate of requisite ensemble size, you should use cross validation. However, that procedure is time consuming.

## **Classification: Imbalanced Data or Unequal Misclassification Costs**

In many real-world applications, you might prefer to treat classes in your data asymmetrically. For example, you might have data with many more observations of one class than of any other. Or you might work on a problem in which misclassifying observations of one class has more severe consequences than misclassifying observations of another class. In such situations, you can use two optional parameters for `fitensemble`: `prior` and `cost`.

By using `prior`, you set prior class probabilities (that is, class probabilities used for training). Use this option if some classes are under- or overrepresented in your training set. For example, you might obtain your training data by simulation. Because simulating class A is more expensive than class B, you opt to generate fewer observations of class A and more observations of class B. You expect, however, that class A and class B are mixed in a different proportion in the real world. In this case, set prior probabilities for class A and B approximately to the values you expect to observe in the real world. `fitensemble` normalizes prior probabilities to make them add up to 1; multiplying all prior probabilities by the same positive factor does not affect the result of classification.

If classes are adequately represented in the training data but you want to treat them asymmetrically, use the `cost` parameter. Suppose you want to classify benign and malignant tumors in cancer patients. Failure to identify a malignant tumor (false negative) has far more severe consequences than misidentifying benign as malignant (false positive). You should assign high cost to misidentifying malignant as benign and low cost to misidentifying benign as malignant.

You must pass misclassification costs as a square matrix with nonnegative elements. Element  $C(i, j)$  of this matrix is the cost of classifying an observation into class  $j$  if the true class is  $i$ . The diagonal elements  $C(i, i)$  of the cost matrix must be 0. For the previous example, you can choose malignant tumor to be class 1 and benign tumor to be class 2. Then you can set the cost matrix to

$$\begin{bmatrix} 0 & c \\ 1 & 0 \end{bmatrix}$$

where  $c > 1$  is the cost of misidentifying a malignant tumor as benign. Costs are relative—multiplying all costs by the same positive factor does not affect the result of classification.

If you have only two classes, `fitensemble` adjusts their prior probabilities using  $\tilde{P}_i = C_{ij}P_i$  for class  $i = 1, 2$  and  $j \neq i$ .  $P_i$  are prior probabilities either passed into `fitensemble` or computed from class frequencies in the training

data, and  $\tilde{P}_i$  are adjusted prior probabilities. Then `fitensemble` uses the default cost matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and these adjusted probabilities for training its weak learners. Manipulating the cost matrix is thus equivalent to manipulating the prior probabilities.

If you have three or more classes, `fitensemble` also converts input costs into adjusted prior probabilities. This conversion is more complex. First, `fitensemble` attempts to solve a matrix equation described in Zhou and Liu [20]. If it fails to find a solution, `fitensemble` applies the “average cost” adjustment described in Breiman et al. [6]. For more information, see Zadrozny, Langford, and Abe [19].

### Example: Unequal Classification Costs

This example uses data on patients with hepatitis to see if they live or die as a result of the disease. The data is described at <http://archive.ics.uci.edu/ml/datasets/Hepatitis>.

- 1 Load the data into a file named `hepatitis.txt`:

```
s = urlread(['http://archive.ics.uci.edu/ml/' ...
            'machine-learning-databases/hepatitis/hepatitis.data']);
fid = fopen('hepatitis.txt','w');
fwrite(fid,s);
fclose(fid);
```

- 2 Load the data `hepatitis.txt` into a dataset, with variable names describing the fields in the data:

```
VarNames = {'die_or_live' 'age' 'sex' 'steroid' 'antivirals' 'fatigue' ...  
            'malaise' 'anorexia' 'liver_big' 'liver_firm' 'spleen_palpable' ...  
            'spiders' 'ascites' 'varices' 'bilirubin' 'alk_phosphate' 'sgot' ...  
            'albumin' 'protime' 'histology'};  
ds = dataset('file','hepatitis.txt','VarNames',VarNames,...  
            'Delimiter',' ',' ','ReadVarNames',false,'TreatAsEmpty','?',...  
            'Format','%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f%f');
```

ds is a dataset with 155 observations and 20 variables:

```
size(ds)
```

```
ans =
```

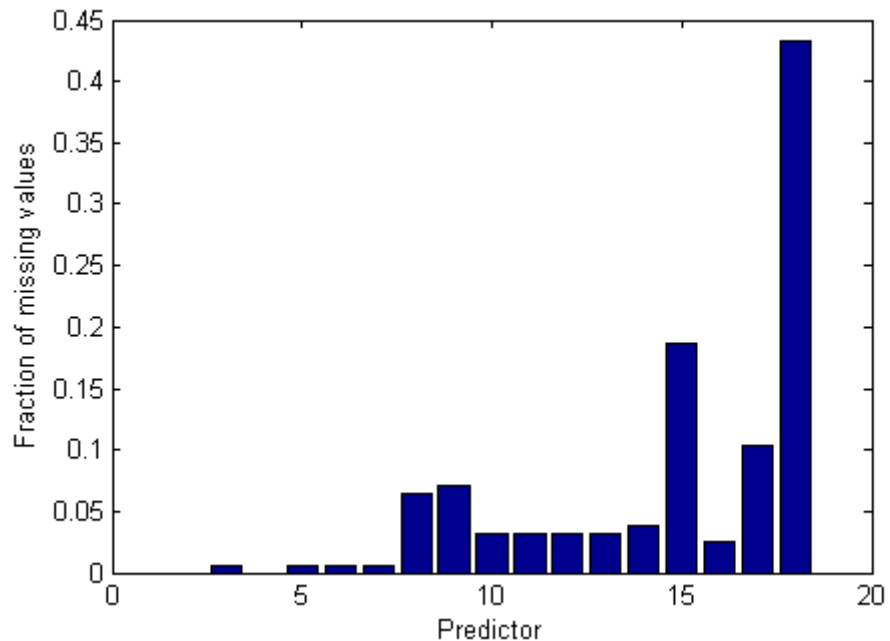
```
    155    20
```

- 3 Convert the data in the dataset to the format for ensembles: a numeric matrix of predictors, and a cell array with outcome names: 'Die' or 'Live'. The first field in the dataset has the outcomes.

```
X = double(ds(:,2:end));  
ClassNames = {'Die' 'Live'};  
Y = ClassNames(ds.die_or_live);
```

- 4 Inspect the data for missing values:

```
figure;  
bar(sum(isnan(X),1)/size(X,1));  
xlabel('Predictor');  
ylabel('Fraction of missing values');
```



Most predictors have missing values, and one has nearly 45% of missing values. Therefore, use decision trees with surrogate splits for better accuracy. Because the dataset is small, training time with surrogate splits should be tolerable.

- 5** Create a classification tree template that uses surrogate splits:

```
rng(0,'twister') % for reproducibility
t = ClassificationTree.template('surrogate','all');
```

- 6** Examine the data or the description of the data to see which predictors are categorical:

```
X(1:5,:)
```

```
ans =
```

```
Columns 1 through 6
```

```

30.0000    2.0000    1.0000    2.0000    2.0000    2.0000
50.0000    1.0000    1.0000    2.0000    1.0000    2.0000
78.0000    1.0000    2.0000    2.0000    1.0000    2.0000
31.0000    1.0000         NaN    1.0000    2.0000    2.0000
34.0000    1.0000    2.0000    2.0000    2.0000    2.0000

```

Columns 7 through 12

```

2.0000    1.0000    2.0000    2.0000    2.0000    2.0000
2.0000    1.0000    2.0000    2.0000    2.0000    2.0000
2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
2.0000    2.0000    2.0000    2.0000    2.0000    2.0000

```

Columns 13 through 18

```

2.0000    1.0000    85.0000    18.0000    4.0000         NaN
2.0000    0.9000   135.0000    42.0000    3.5000         NaN
2.0000    0.7000    96.0000    32.0000    4.0000         NaN
2.0000    0.7000    46.0000    52.0000    4.0000    80.0000
2.0000    1.0000         NaN   200.0000    4.0000         NaN

```

Column 19

```

1.0000
1.0000
1.0000
1.0000
1.0000

```

It appears that predictors 2 through 13 are categorical, as well as predictor 19. You can confirm this inference with the dataset description at <http://archive.ics.uci.edu/ml/datasets/Hepatitis>.

**7** List the categorical variables:

```
ncat = [2:13,19];
```

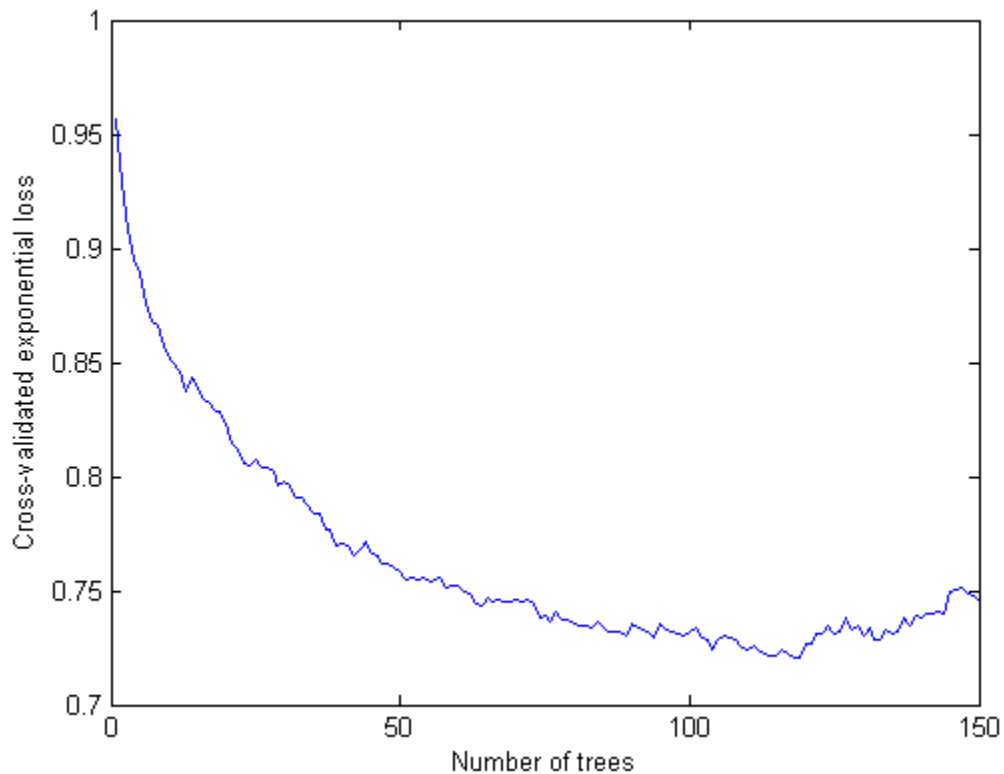
**8** Create a cross-validated ensemble using 150 learners and the GentleBoost algorithm:



```

a = fitensemble(X,Y,'GentleBoost',150,t,...
    'PredictorNames',VarNames(2:end),'LearnRate',0.1,...
    'CategoricalPredictors',ncat,'kfold',5);
figure;
plot(kfoldLoss(a,'mode','cumulative','lossfun','exponential'));
xlabel('Number of trees');
ylabel('Cross-validated exponential loss');

```



- 9 Inspect the confusion matrix to see which people the ensemble predicts correctly:

```

[Yfit,Sfit] = kfoldPredict(a); %
confusionmat(Y,Yfit,'order',ClassNames)

```

```
ans =
    18    14
    11   112
```

Of the 123 people who live, the ensemble predicts correctly that 112 will live. But for the 32 people who die of hepatitis, the ensemble only predicts correctly that about half will die of hepatitis.

**10** There are two types of error in the predictions of the ensemble:

- Predicting that the patient lives, but the patient dies
- Predicting that the patient dies, but the patient lives

Suppose you believe that the first error is five times worse than the second. Make a new classification cost matrix that reflects this belief:

```
cost.ClassNames = ClassNames;
cost.ClassificationCosts = [0 5; 1 0];
```

**11** Create a new cross-validated ensemble using `cost` as misclassification cost, and inspect the resulting confusion matrix:

```
aC = fitensemble(X,Y,'GentleBoost',150,t,...
    'PredictorNames',VarNames(2:end),'LearnRate',0.1,...
    'CategoricalPredictors',ncat,'kfold',5,...
    'cost',cost);
[YfitC,SfitC] = kfoldPredict(aC);
confusionmat(Y,YfitC,'order',ClassNames)
```

```
ans =
    19    13
     8   115
```

As expected, the new ensemble does a better job classifying the people who die. Somewhat surprisingly, the new ensemble also does a better job classifying the people who live, though the result is not statistically significantly better. The results of the cross validation are random, so this result is simply a statistical fluctuation. The result seems to indicate that the classification of people who live is not very sensitive to the cost.

## Classification with Many Categorical Levels

Generally, you cannot use classification with more than 31 levels in any categorical predictor. However, two boosting algorithms can classify data with many categorical predictor levels and binary responses: `LogitBoost` and `GentleBoost`. For details, see “`LogitBoost`” on page 16-153 and “`GentleBoost`” on page 16-153.

This example uses demographic data from the U.S. Census, available at <http://archive.ics.uci.edu/ml/machine-learning-databases/adult/>. The objective of the researchers who posted the data is predicting whether an individual makes more than \$50,000/year, based on a set of characteristics. You can see details of the data, including predictor names, in the `adult.names` file at the site.

- 1 Load the 'adult.data' file from the UCI Machine Learning Repository:

```
s = urlread(['http://archive.ics.uci.edu/ml/' ...
            'machine-learning-databases/adult/adult.data']);
```

- 2 'adult.data' represents missing data as '?'. Replace instances of missing data with the blank string '':

```
s = strrep(s, '?', '');
```

- 3 Put the data into a MATLAB dataset array:

```
fid = fopen('adult.txt','w');
fwrite(fid,s);
fclose(fid);
clear s;
VarNames = {'age' 'workclass' 'fnlwtg' 'education' 'education_num' ...
            'marital_status' 'occupation' 'relationship' 'race' ...
            'sex' 'capital_gain' 'capital_loss' ...
            'hours_per_week' 'native_country' 'income'};
ds = dataset('file','adult.txt','VarNames',VarNames,...
            'Delimiter',',','ReadVarNames',false,'Format',...
            '%u%s%u%s%u%s%%s%%s%%s%u%u%u%s%');
cat = ~datasetfun(@isnumeric,ds(:,1:end-1)); % Logical indices
%                                         of categorical variables
```

```
catcol = find(cat); % indices of categorical variables
```

- 4** Many predictors in the data are categorical. Convert those fields in the dataset array to nominal:

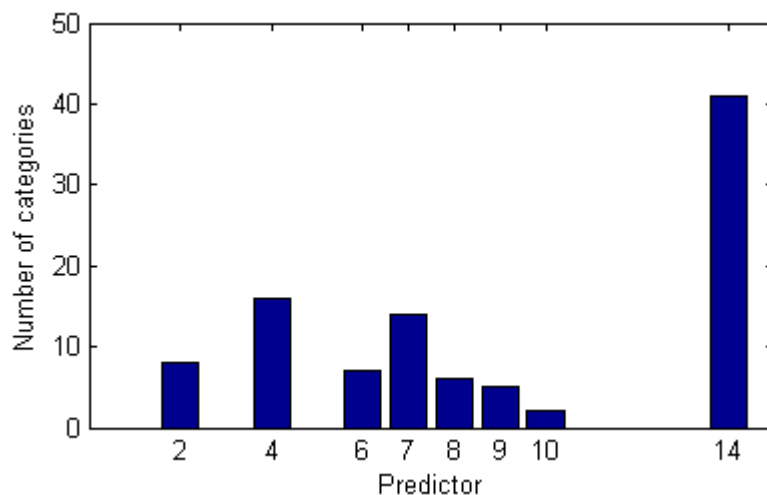
```
ds.workclass = nominal(ds.workclass);  
ds.education = nominal(ds.education);  
ds.marital_status = nominal(ds.marital_status);  
ds.occupation = nominal(ds.occupation);  
ds.relationship = nominal(ds.relationship);  
ds.race = nominal(ds.race);  
ds.sex = nominal(ds.sex);  
ds.native_country = nominal(ds.native_country);  
ds.income = nominal(ds.income);
```

- 5** Convert the dataset array into numerical variables for `fitensemble`:

```
X = double(ds(:,1:end-1));  
Y = ds.income;
```

- 6** Some variables have many levels. Plot the number of levels of each predictor:

```
ncat = zeros(1,numel(catcol));  
for c=1:numel(catcol)  
    [~,gn] = grp2idx(X(:,catcol(c)));  
    ncat(c) = numel(gn);  
end  
figure;  
bar(catcol,ncat);  
xlabel('Predictor');  
ylabel('Number of categories');
```



Predictor 14 ('native\_country') has more than 40 categorical levels. For binary classification, `ClassificationTree.fit` uses a computational shortcut to find an optimal split for categorical predictors with many categories. For classification with more than two classes, you can choose a heuristic algorithm for finding a good split. See “Splitting Categorical Predictors” on page 16-56.

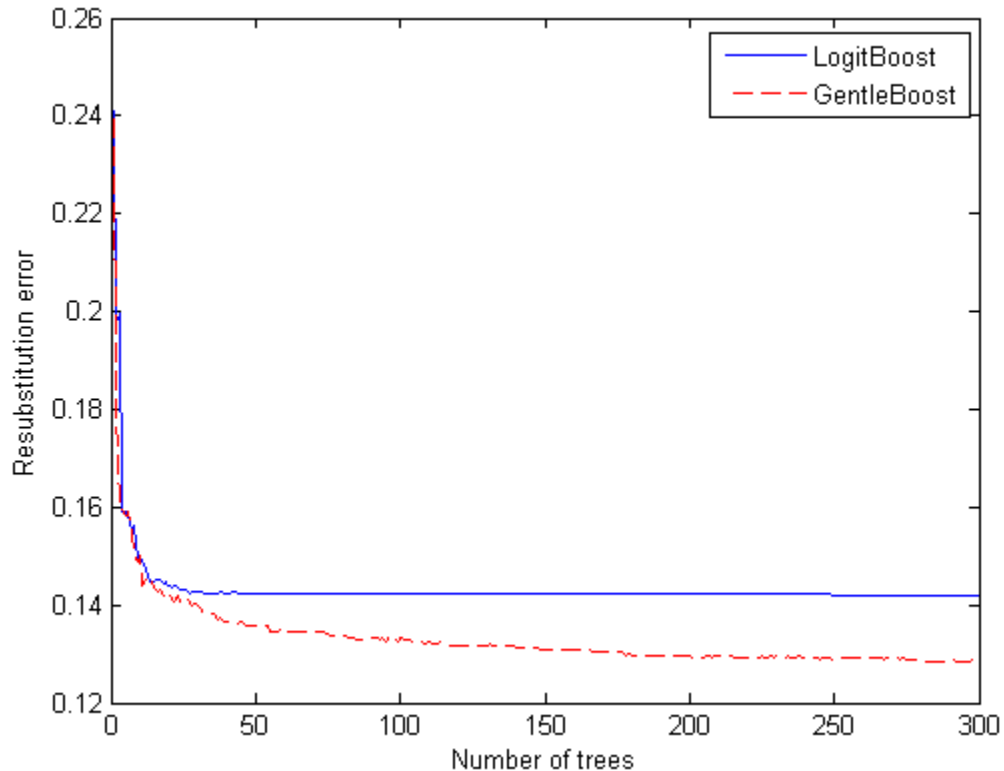
**7** Create classification ensembles using both `LogitBoost` and `GentleBoost`:

```
lb = fitensemble(X,Y,'LogitBoost',300,'Tree','CategoricalPredictors',cat,...
    'PredictorNames',VarNames(1:end-1),'ResponseName','income');
gb = fitensemble(X,Y,'GentleBoost',300,'Tree','CategoricalPredictors',cat,...
    'PredictorNames',VarNames(1:end-1),'ResponseName','income');
```

**8** Examine the resubstitution error for the two ensembles:

```
figure;
plot(resubLoss(lb,'mode','cumulative'));
hold on
plot(resubLoss(gb,'mode','cumulative'),'r--');
hold off
xlabel('Number of trees');
ylabel('Resubstitution error');
```

```
legend('LogitBoost','GentleBoost','Location','NE');
```

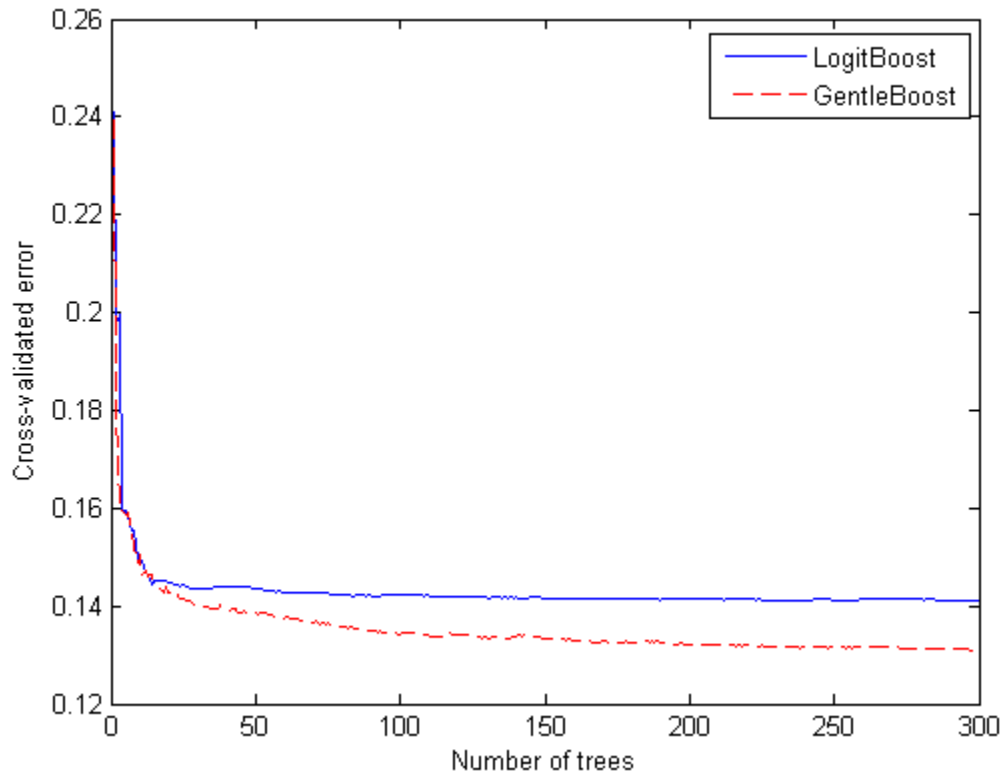


The GentleBoost algorithm has slightly smaller resubstitution error.

- 9 Estimate the generalization error for the two algorithms by cross validation.

```
lbcv = crossval(lb,'kfold',5);
gbcv = crossval(gb,'kfold',5);
figure;
plot(kfoldLoss(lbcv,'mode','cumulative'));
hold on
plot(kfoldLoss(gbcv,'mode','cumulative'),'r--');
hold off
```

```
xlabel('Number of trees');  
ylabel('Cross-validated error');  
legend('LogitBoost', 'GentleBoost', 'Location', 'NE');
```



The cross-validated loss is nearly the same as the resubstitution error.

## Surrogate Splits

When you have missing data, trees and ensembles of trees give better predictions when they include surrogate splits. Furthermore, estimates of predictor importance are often different with surrogate splits. Eliminating unimportant predictors can save time and memory for predictions, and can make predictions easier to understand.

This example shows the effects of surrogate splits for predictions for data containing missing entries in the test set.

- 1** Load sample data. Partition it into a training and test set.

```
load ionosphere;

rng(10) % for reproducibility
cv = cvpartition(Y,'holdout',0.3);
Xtrain = X(training(cv),:);
Ytrain = Y(training(cv));
Xtest = X(test(cv),:);
Ytest = Y(test(cv));
```

- 2** Bag decision trees with and without surrogate splits.

```
b = fitensemble(Xtrain,Ytrain,'Bag',50,'Tree',...
    'type','class');

templS = ClassificationTree.template('surrogate','on');
bs = fitensemble(Xtrain,Ytrain,'Bag',50,templS,...
    'type','class');
```

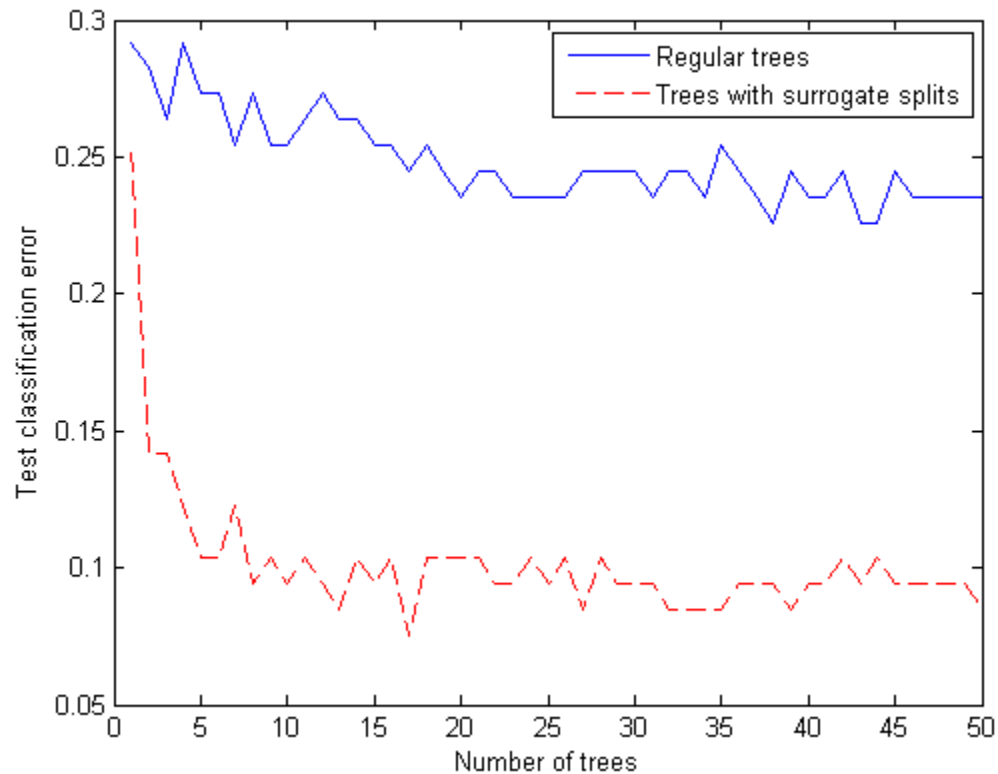
- 3** Suppose half of the values in the test set are missing.

```
Xtest(rand(size(Xtest))>0.5) = NaN;
```

- 4** Test accuracy with and without surrogate splits.

```
figure;
plot(loss(b,Xtest,Ytest,'mode','cumulative'));
hold on;
plot(loss(bs,Xtest,Ytest,'mode','cumulative'),'r--');
legend('Regular trees','Trees with surrogate splits');
xlabel('Number of trees');
ylabel('Test classification error');
```





- 5** Check the statistical significance of the difference in results with the McNemar test. Convert the labels to a nominal data type to make it easier to check for equality.

```

Yfit = nominal(predict(b,Xtest));
YfitS = nominal(predict(bs,Xtest));
N10 = sum(Yfit==nominal(Ytest) & YfitS~=nominal(Ytest));
N01 = sum(Yfit~=nominal(Ytest) & YfitS==nominal(Ytest));
mcnemar = (abs(N10-N01) - 1)^2/(N10+N01);
pval = 1 - chi2cdf(mcnemar,1)

```

```
pval =
```

1.7683e-04

The extremely low  $p$ -value indicates that the ensemble with surrogate splits is better in a statistically significant manner.

## LPBoost and TotalBoost for Small Ensembles

This example shows how to obtain the benefits of the LPBoost and TotalBoost algorithms. These algorithms share two beneficial characteristics:

They are self-terminating, so you don't have to guess how many members to include.

They produce ensembles with some very small weights, so you can safely remove ensemble members.

Note that the algorithms in this example require an Optimization Toolbox license.

### Step 1. Load the data.

Load the ionosphere data.

```
load ionosphere
```

### Step 2. Create the classification ensembles.

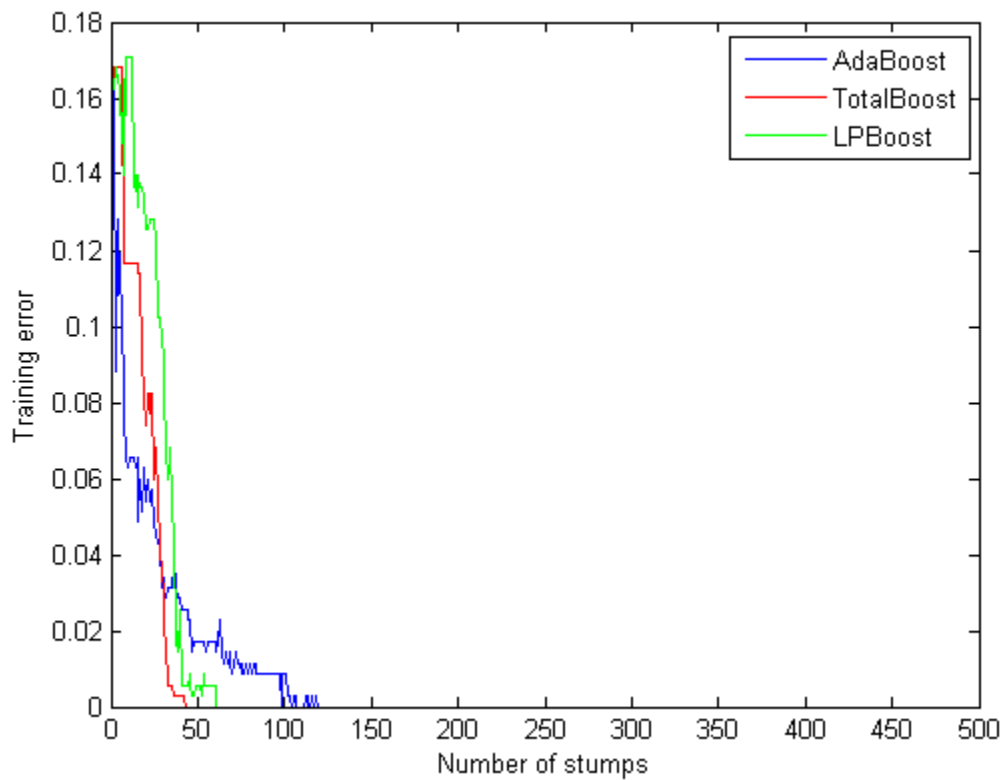
Create ensembles for classifying the ionosphere data using the LPBoost, TotalBoost, and, for comparison, AdaBoostM1 algorithms. It is hard to know how many members to include in an ensemble. For LPBoost and TotalBoost, try using 500. For comparison, also use 500 for AdaBoostM1.

```
rng default % for reproducibility
T = 500;
adaStump = fitensemble(X,Y,'AdaBoostM1',T,'Tree');
totalStump = fitensemble(X,Y,'TotalBoost',T,'Tree');
lpStump = fitensemble(X,Y,'LPBoost',T,'Tree');
figure;
plot(resubLoss(adaStump,'mode','cumulative'));
hold on
plot(resubLoss(totalStump,'mode','cumulative'),'r');
```

```

plot(resubLoss(lpStump,'mode','cumulative'),'g');
hold off
xlabel('Number of stumps');
ylabel('Training error');
legend('AdaBoost','TotalBoost','LPBoost','Location','NE');

```



All three algorithms achieve perfect prediction on the training data after a while. Examine the number of members in all three ensembles.

```
[adaStump.NTrained totalStump.NTrained lpStump.NTrained]
```

ans =

500 52 67

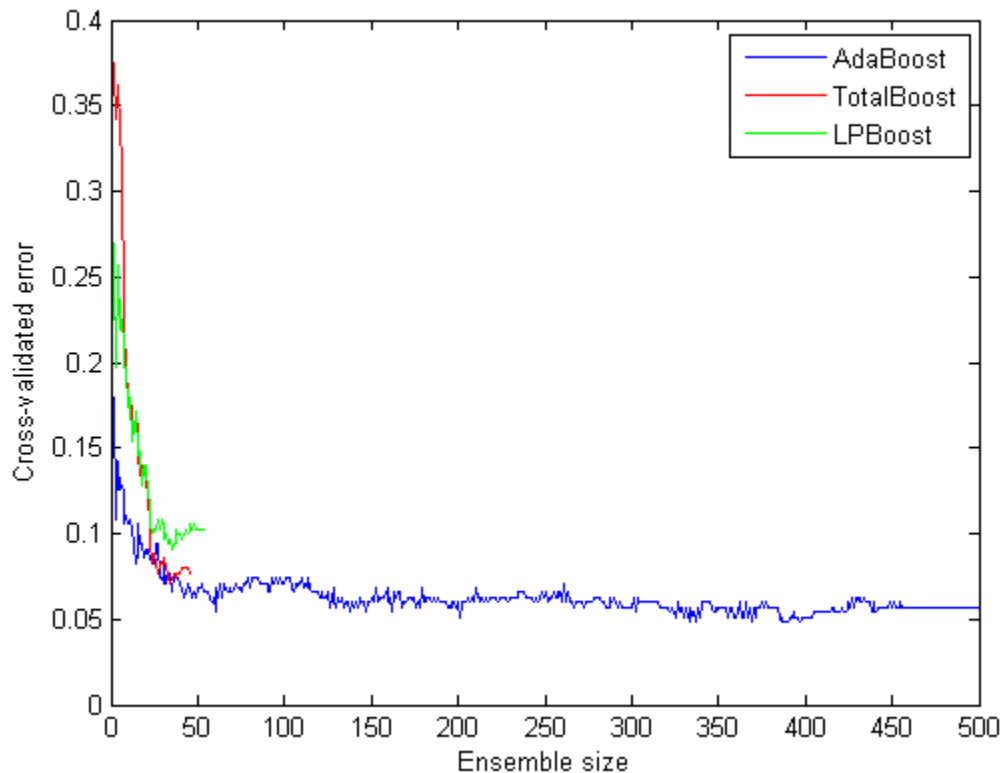
AdaBoostM1 trained all 500 members. The other two algorithms stopped training early.

**Step 3. Cross validate the ensembles.**

Cross validate the ensembles to better determine ensemble accuracy.

```
cvlp = crossval(lpStump, 'kfold', 5);
cvtotal = crossval(totalStump, 'kfold', 5);
cvada = crossval(adaStump, 'kfold', 5);

figure;
plot(kfoldLoss(cvada, 'mode', 'cumulative'));
hold on
plot(kfoldLoss(cvtotal, 'mode', 'cumulative'), 'r');
plot(kfoldLoss(cvlp, 'mode', 'cumulative'), 'g');
hold off
xlabel('Ensemble size');
ylabel('Cross-validated error');
legend('AdaBoost', 'TotalBoost', 'LPBoost', 'Location', 'NE');
```



It appears that each boosting algorithms achieves 10% or lower loss with 50 ensemble members, and AdaBoostM1 achieves near 6% error with 150 or more ensemble members.

#### Step 4. Compact and remove ensemble members.

To reduce the ensemble sizes, compact them, and then use the `removeLearners` method. The question is, how many learners should you remove? The cross-validated loss curves give you one measure. For another, examine the learner weights for LPBoost and TotalBoost after compacting.

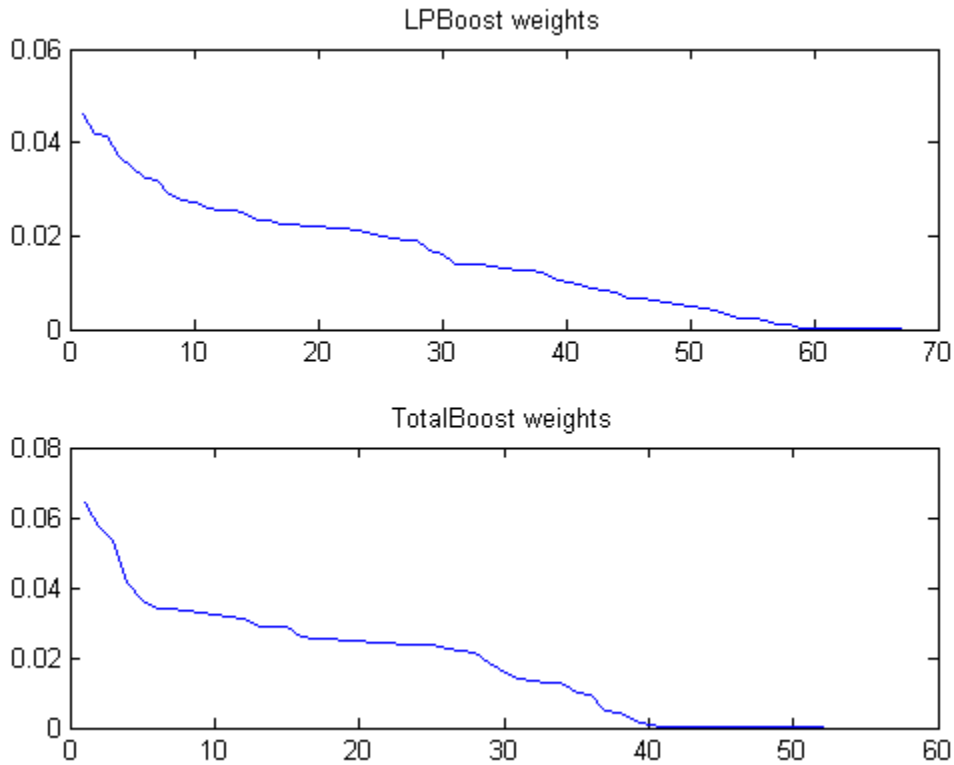
```
cada = compact(adaStump);
c1p = compact(lpStump);
```

```

ctotal = compact(totalStump);

figure
subplot(2,1,1)
plot(clp.TrainedWeights)
title('LPBoost weights')
subplot(2,1,2)
plot(ctotal.TrainedWeights)
title('TotalBoost weights')

```



Both LPBoost and TotalBoost show clear points where the ensemble member weights become negligible.

Remove the unimportant ensemble members.

```
cada = removeLearners(cada,150:cada.NTrained);
clp = removeLearners(clp,60:clp.NTrained);
ctotal = removeLearners(ctotal,40:ctotal.NTrained);
```

Check that removing these learners does not affect ensemble accuracy on the training data.

```
[loss(cada,X,Y) loss(clp,X,Y) loss(ctotal,X,Y)]
```

```
ans =
     0     0     0
```

Check the resulting compact ensemble sizes.

```
s(1) = whos('cada');
s(2) = whos('clp');
s(3) = whos('ctotal');
s.bytes
```

```
ans =
    605605
```

```
ans =
    241375
```

```
ans =
    160435
```

The sizes of the compact ensembles are approximately proportional to the number of members in each.

## Ensemble Regularization

Regularization is a process of choosing fewer weak learners for an ensemble in a way that does not diminish predictive performance. Currently you can regularize regression ensembles. (You can also regularize a discriminant analysis classifier in a non-ensemble context; see “Regularize a Discriminant Analysis Classifier” on page 15-23.)

The `regularize` method finds an optimal set of learner weights  $a_t$  that minimize

$$\sum_{n=1}^N w_n g \left( \left( \sum_{t=1}^T \alpha_t h_t(x_n) \right), y_n \right) + \lambda \sum_{t=1}^T |\alpha_t|.$$

Here

- $\lambda \geq 0$  is a parameter you provide, called the lasso parameter.
- $h_t$  is a weak learner in the ensemble trained on  $N$  observations with predictors  $x_n$ , responses  $y_n$ , and weights  $w_n$ .
- $g(f, y) = (f - y)^2$  is the squared error.

The ensemble is regularized on the same  $(x_n, y_n, w_n)$  data used for training, so

$$\sum_{n=1}^N w_n g \left( \left( \sum_{t=1}^T \alpha_t h_t(x_n) \right), y_n \right)$$

is the ensemble resubstitution error. The error is measured by mean squared error (MSE).

If you use  $\lambda = 0$ , `regularize` finds the weak learner weights by minimizing the resubstitution MSE. Ensembles tend to overtrain. In other words, the resubstitution error is typically smaller than the true generalization error. By making the resubstitution error even smaller, you are likely to make the ensemble accuracy worse instead of improving it. On the other hand, positive values of  $\lambda$  push the magnitude of the  $\alpha_t$  coefficients to 0. This often improves the generalization error. Of course, if you choose  $\lambda$  too large, all the optimal coefficients are 0, and the ensemble does not have any accuracy. Usually you can find an optimal range for  $\lambda$  in which the accuracy of the regularized ensemble is better or comparable to that of the full ensemble without regularization.

A nice feature of lasso regularization is its ability to drive the optimized coefficients precisely to 0. If a learner's weight  $\alpha_t$  is 0, this learner can be



excluded from the regularized ensemble. In the end, you get an ensemble with improved accuracy and fewer learners.

### **Example: Regularizing a Regression Ensemble**

This example uses data for predicting the insurance risk of a car based on its many attributes.

- 1** Load the `imports-85` data into the MATLAB workspace:

```
load imports-85;
```

- 2** Look at a description of the data to find the categorical variables and predictor names:

#### **Description**

```
Description =  
1985 Auto Imports Database from the UCI repository  
http://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.names  
Variables have been reordered to place variables with numeric values (referred  
to as "continuous" on the UCI site) to the left and categorical values to the  
right. Specifically, variables 1:16 are: symboling, normalized-losses,  
wheel-base, length, width, height, curb-weight, engine-size, bore, stroke,  
compression-ratio, horsepower, peak-rpm, city-mpg, highway-mpg, and price.  
Variables 17:26 are: make, fuel-type, aspiration, num-of-doors, body-style,  
drive-wheels, engine-location, engine-type, num-of-cylinders, and fuel-system.
```

The objective of this process is to predict the “symboling,” the first variable in the data, from the other predictors. “symboling” is an integer from -3 (good insurance risk) to 3 (poor insurance risk). You could use a classification ensemble to predict this risk instead of a regression ensemble. As stated in “Steps in Supervised Learning (Machine Learning)” on page 16-2, when you have a choice between regression and classification, you should try regression first. Furthermore, this example is to show regularization, which currently works only for regression.

- 3** Prepare the data for ensemble fitting:

```

Y = X(:,1);
X(:,1) = [];
VarNames = {'normalized-losses' 'wheel-base' 'length' 'width' 'height' ...
            'curb-weight' 'engine-size' 'bore' 'stroke' 'compression-ratio' ...
            'horsepower' 'peak-rpm' 'city-mpg' 'highway-mpg' 'price' 'make' ...
            'fuel-type' 'aspiration' 'num-of-doors' 'body-style' 'drive-wheels' ...
            'engine-location' 'engine-type' 'num-of-cylinders' 'fuel-system'};
catidx = 16:25; % indices of categorical predictors

```

**4** Create a regression ensemble from the data using 300 default trees:

```

ls = fitensemble(X,Y,'LSBoost',300,'Tree','LearnRate',0.1,...
                'PredictorNames',VarNames,'ResponseName','symboling',...
                'CategoricalPredictors',catidx)

```

```
ls =
```

```

classreg.learning.regr.RegressionEnsemble
    PredictorNames: {1x25 cell}
    ResponseName: 'symboling'
    ResponseTransform: 'none'
    NObservations: 205
    NTrained: 300
    Method: 'LSBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: [1x77 char]
    FitInfo: [300x1 double]
    FitInfoDescription: {2x1 cell}
    Regularization: []

```

Properties, Methods

The final line, `Regularization`, is empty (`[]`). To regularize the ensemble, you have to use the `regularize` method.

**5** Cross validate the ensemble, and inspect its loss curve.

```

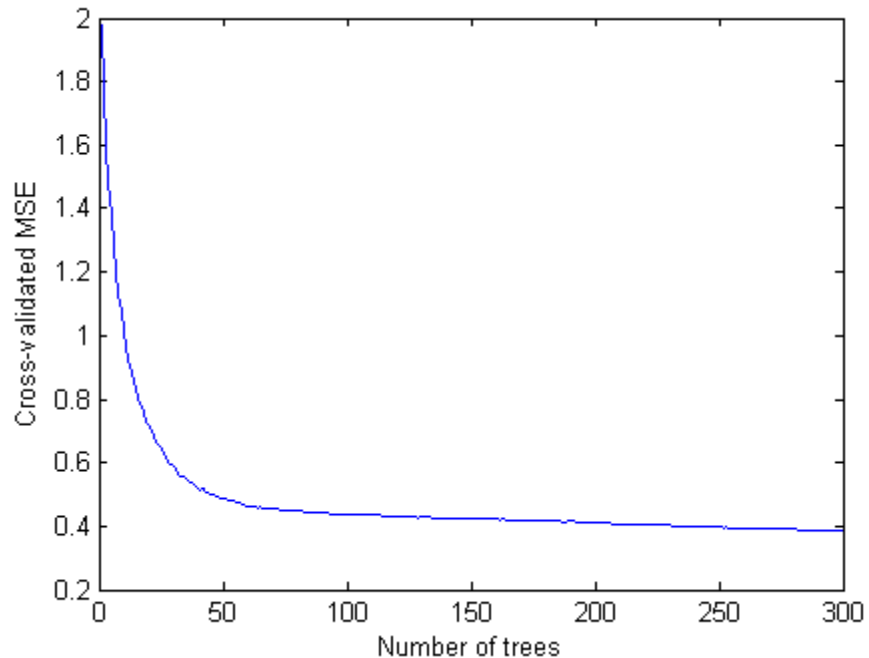
cv = crossval(ls,'kfold',5);
figure;

```

```

plot(kfoldLoss(cv,'mode','cumulative'));
xlabel('Number of trees');
ylabel('Cross-validated MSE');
ylim([0.2,2])

```



It appears you might obtain satisfactory performance from a smaller ensemble, perhaps one containing from 50 to 100 trees.

- 6 Call the `regularize` method to try to find trees that you can remove from the ensemble. By default, `regularize` examines 10 values of the lasso (Lambda) parameter spaced exponentially.

```
ls = regularize(ls)
```

```
ls =
```

```

classreg.learning.regr.ReggressionEnsemble
    PredictorNames: {1x25 cell}

```

```

        ResponseName: 'symboling'
    ResponseTransform: 'none'
        NObservations: 205
        NTrained: 300
        Method: 'LSBoost'
        LearnerNames: {'Tree'}
    ReasonForTermination: [1x77 char]
        FitInfo: [300x1 double]
    FitInfoDescription: {2x1 cell}
        Regularization: [1x1 struct]

```

Properties, Methods

The Regularization property is no longer empty.

- 7 Plot the resubstitution mean-squared error (MSE) and number of learners with nonzero weights against the lasso parameter. Separately plot the value at  $\text{Lambda}=0$ . Use a logarithmic scale because the values of  $\text{Lambda}$  are exponentially spaced.

```

figure;
semilogx(ls.Regularization.Lambda,ls.Regularization.ResubstitutionMSE);
line([1e-3 1e-3],[ls.Regularization.ResubstitutionMSE(1) ...
    ls.Regularization.ResubstitutionMSE(1)],...
    'marker','x','markersize',12,'color','b');
r0 = resubLoss(ls);
line([ls.Regularization.Lambda(2) ls.Regularization.Lambda(end)],...
    [r0 r0],'color','r','LineStyle','--');
xlabel('Lambda');
ylabel('Resubstitution MSE');
annotation('textbox',[0.5 0.22 0.5 0.05],'String','unregularized ensemble',...
    'color','r','FontSize',14,'LineStyle','none');

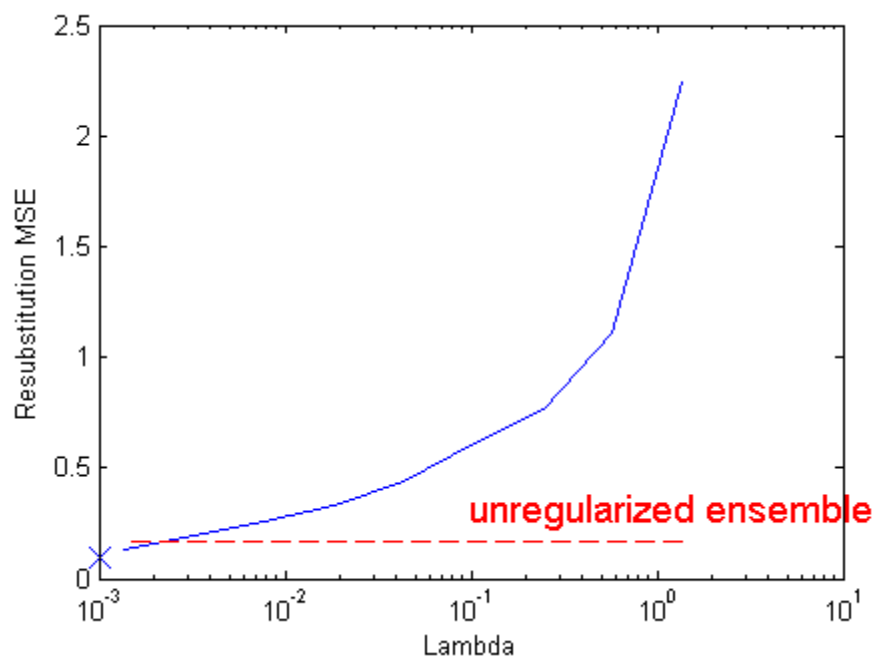
```

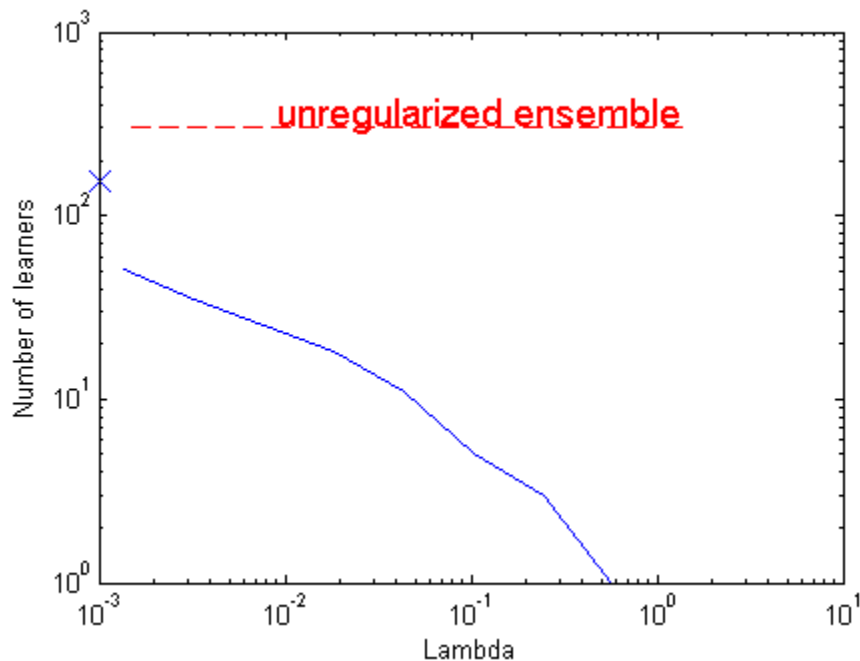
```

figure;
loglog(ls.Regularization.Lambda,sum(ls.Regularization.TrainedWeights>0,1));
line([1e-3 1e-3],...
    [sum(ls.Regularization.TrainedWeights(:,1)>0) ...
    sum(ls.Regularization.TrainedWeights(:,1)>0)],...

```

```
'marker','x','markersize',12,'color','b');  
line([ls.Regularization.Lambda(2) ls.Regularization.Lambda(end)],...  
      [ls.NTrained ls.NTrained],...  
      'color','r','LineStyle','--');  
xlabel('Lambda');  
ylabel('Number of learners');  
annotation('textbox',[0.3 0.8 0.5 0.05],'String','unregularized ensemble',...  
           'color','r','FontSize',14,'LineStyle','none');
```





- 8 The resubstitution MSE values are likely to be overly optimistic. To obtain more reliable estimates of the error associated with various values of Lambda, cross validate the ensemble using `cvshrink`. Plot the resulting cross-validation loss (MSE) and number of learners against Lambda.

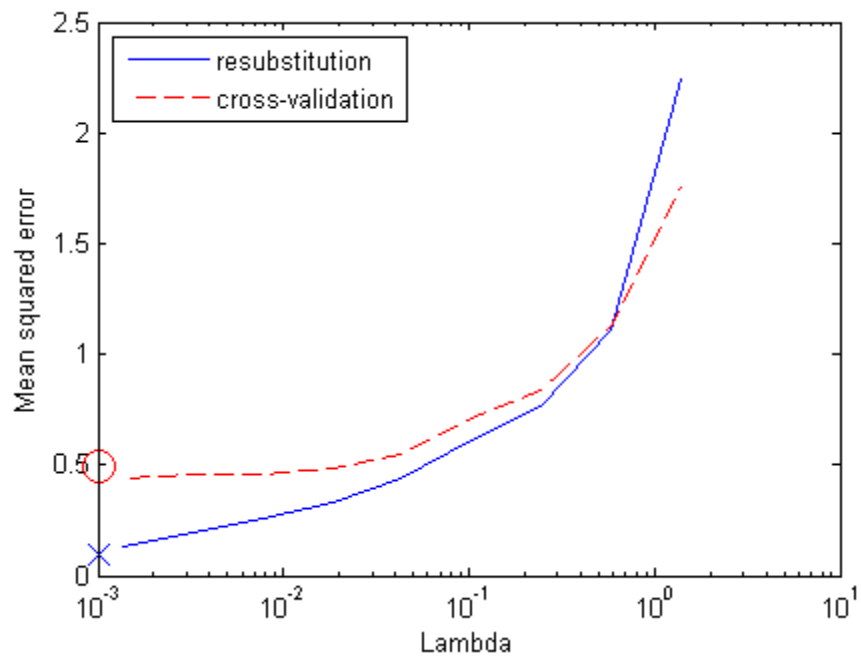
```
rng(0,'Twister') % for reproducibility
[mse,nlearn] = cvshrink(ls,'lambda',ls.Regularization.Lambda,'kfold',5);

figure;
semilogx(ls.Regularization.Lambda,ls.Regularization.ResubstitutionMSE);
hold;
semilogx(ls.Regularization.Lambda,mse,'r--');
hold off;
xlabel('Lambda');
ylabel('Mean squared error');
legend('resubstitution','cross-validation','Location','NW');
```

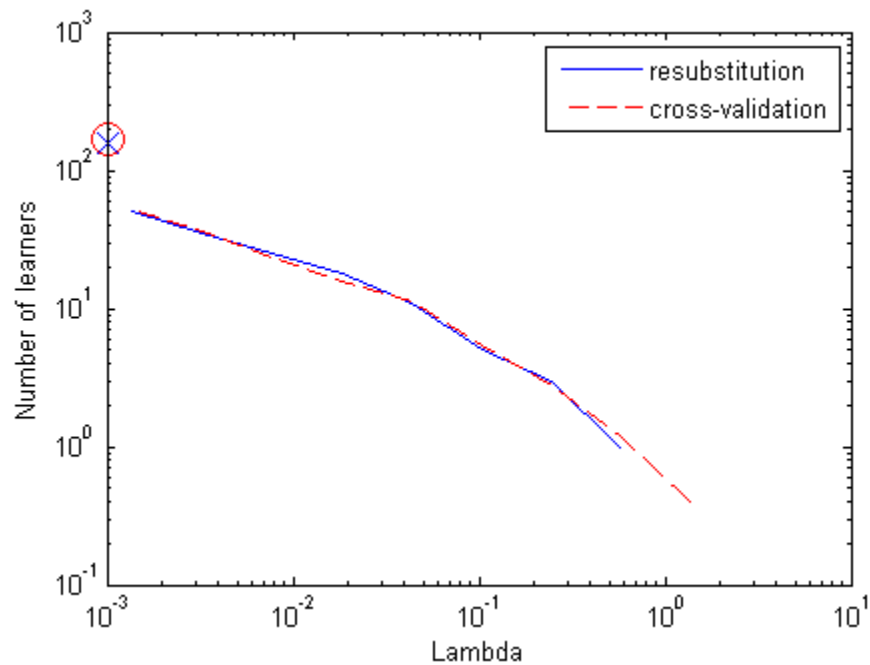
```
line([1e-3 1e-3],[ls.Regularization.ResubstitutionMSE(1) ...
    ls.Regularization.ResubstitutionMSE(1)],...
    'marker','x','markersize',12,'color','b');
line([1e-3 1e-3],[mse(1) mse(1)],'marker','o',...
    'markersize',12,'color','r','LineStyle','--');

figure;
loglog(ls.Regularization.Lambda,sum(ls.Regularization.TrainedWeights>0,1));
hold;
loglog(ls.Regularization.Lambda,nlearn,'r--');
hold off;
xlabel('Lambda');
ylabel('Number of learners');
legend('resubstitution','cross-validation','Location','NE');
line([1e-3 1e-3],...
    [sum(ls.Regularization.TrainedWeights(:,1)>0) ...
    sum(ls.Regularization.TrainedWeights(:,1)>0)],...
    'marker','x','markersize',12,'color','b');
line([1e-3 1e-3],[nlearn(1) nlearn(1)],'marker','o',...
```

```
'markersize',12,'color','r','LineStyle','--');
```







Examining the cross-validated error shows that the cross-validation MSE is almost flat for Lambda up to a bit over  $1e-2$ .

- 9 Examine `ls.Regularization.Lambda` to find the highest value that gives MSE in the flat region (up to a bit over  $1e-2$ ):

```
jj = 1:length(ls.Regularization.Lambda);
[jj;ls.Regularization.Lambda]
```

```
ans =
```

```
Columns 1 through 6
```

1.0000	2.0000	3.0000	4.0000	5.0000	6.0000
0	0.0014	0.0033	0.0077	0.0183	0.0435

```
Columns 7 through 10
```

```

7.0000    8.0000    9.0000   10.0000
0.1031    0.2446    0.5800    1.3754

```

Element 5 of `ls.Regularization.Lambda` has value 0.0183, the largest in the flat range.

- 10** Reduce the ensemble size using the `shrink` method. `shrink` returns a compact ensemble with no training data. The generalization error for the new compact ensemble was already estimated by cross validation in `mse(5)`.

```

cmp = shrink(ls,'weightcolumn',5)

cmp =

classreg.learning.regr.CompactRegressionEnsemble
    PredictorNames: {1x25 cell}
    ResponseName: 'symboling'
    ResponseTransform: 'none'
    NTrained: 15

```

Properties, Methods

There are only 15 trees in the new ensemble, notably reduced from the 300 in `ls`.

- 11** Compare the sizes of the ensembles:

```

sz(1) = whos('cmp'); sz(2) = whos('ls');
[sz(1).bytes sz(2).bytes]

ans =

    84544    1775691

```

The reduced ensemble is about 5% the size of the original.

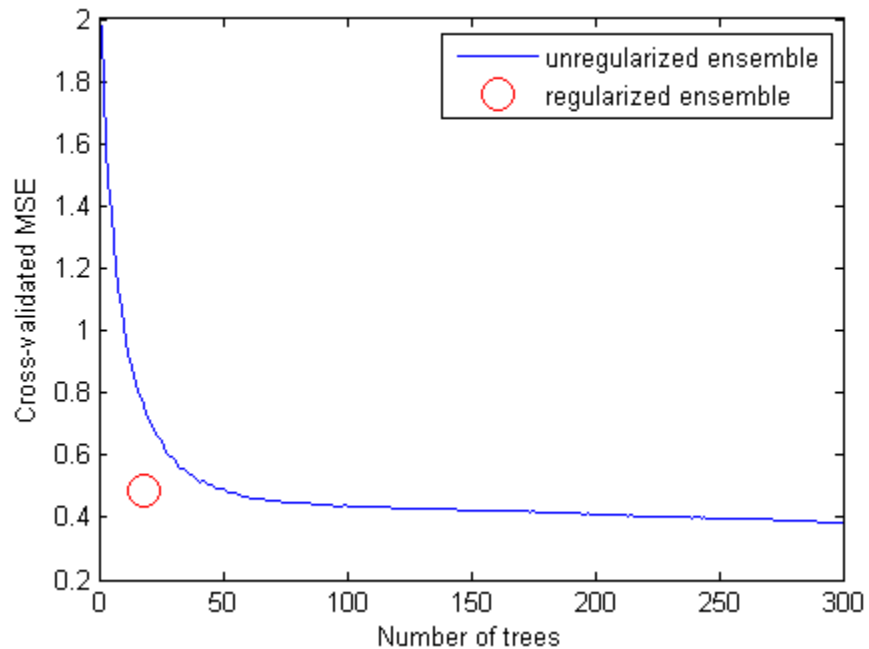
- 12** Compare the MSE of the reduced ensemble to that of the original ensemble:

```

figure;
plot(kfoldLoss(cv,'mode','cumulative'));
hold on

```

```
plot(cmp.NTrained,mse(5),'ro','MarkerSize',12);
xlabel('Number of trees');
ylabel('Cross-validated MSE');
legend('unregularized ensemble','regularized ensemble',...
      'Location','NE');
hold off
```



The reduced ensemble gives low loss while using many fewer trees.

## Tuning RobustBoost

The RobustBoost algorithm can make good classification predictions even when the training data has noise. However, the default RobustBoost parameters can produce an ensemble that does not predict well. This example shows one way of tuning the parameters for better predictive accuracy.

---

**Note** RobustBoost requires an Optimization Toolbox license.

---

- 1 Generate data with label noise. This example has twenty uniform random numbers per observation, and classifies the observation as 1 if the sum of the first five numbers exceeds 2.5 (so is larger than average), and 0 otherwise:

```
rng(0,'twister') % for reproducibility
Xtrain = rand(2000,20);
Ytrain = sum(Xtrain(:,1:5),2) > 2.5;
```

- 2 To add noise, randomly switch 10% of the classifications:

```
idx = randsample(2000,200);
Ytrain(idx) = ~Ytrain(idx);
```

- 3 Create an ensemble with AdaBoostM1 for comparison purposes:

```
ada = fitensemble(Xtrain,Ytrain,'AdaBoostM1',...
    300,'Tree','LearnRate',0.1);
```

- 4 Create an ensemble with RobustBoost. Because the data has 10% incorrect classification, perhaps an error goal of 15% is reasonable.

```
rb1 = fitensemble(Xtrain,Ytrain,'RobustBoost',300,...
    'Tree','RobustErrorGoal',0.15,'RobustMaxMargin',1);
```

- 5 Try setting a high value of the error goal, 0.6. You get an error:

```
rb2 = fitensemble(Xtrain,Ytrain,'RobustBoost',300,'Tree','RobustErrorGoal',0.6)
```

Error using RobustBoost>RobustBoost.RobustBoost

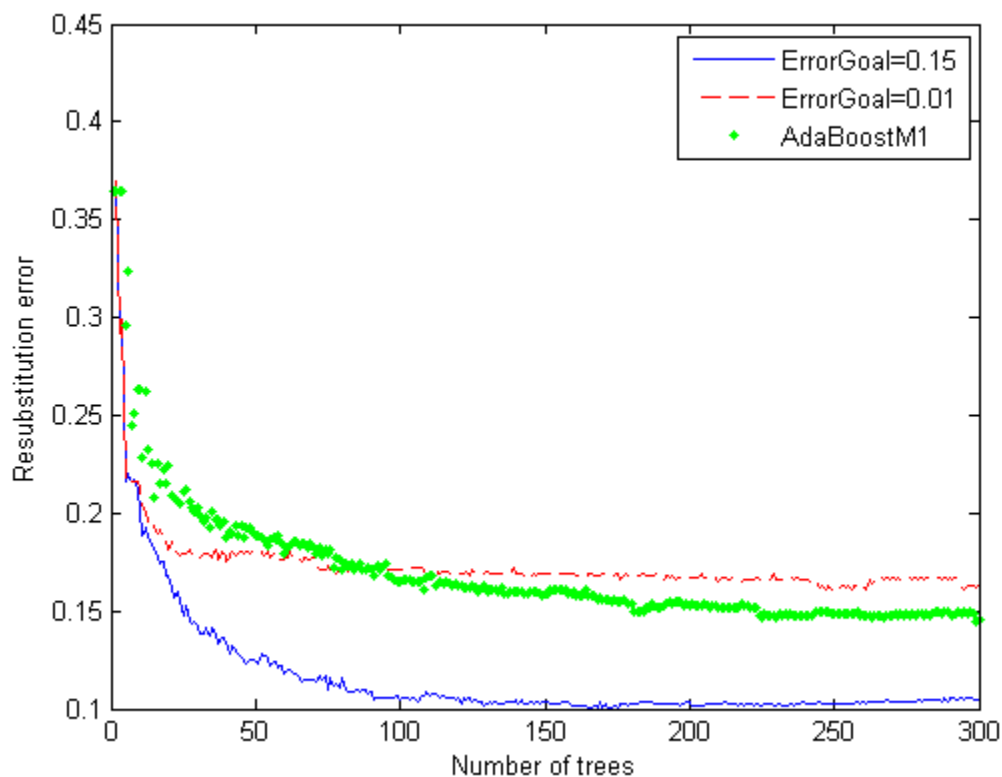
For the chosen values of 'RobustMaxMargin' and 'RobustMarginSigma', you must set 'RobustErrorGoal' to a value between 0 and 0.5.

- 6 Create an ensemble with very optimistic error goal, 0.01:

```
rb2 = fitensemble(Xtrain,Ytrain,'RobustBoost',300,...
    'Tree','RobustErrorGoal',0.01);
```

**7** Compare the resubstitution error of the four ensembles:

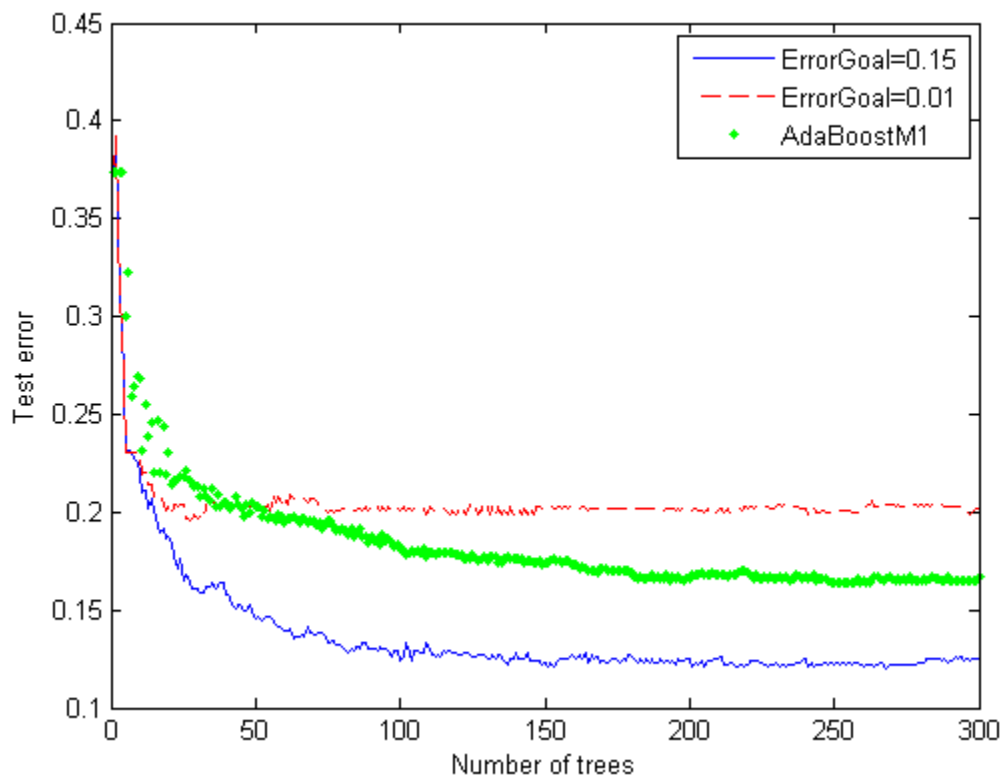
```
figure
plot(resubLoss(rb1,'mode','cumulative'));
hold on
plot(resubLoss(rb2,'mode','cumulative'),'r--');
plot(resubLoss(ada,'mode','cumulative'),'g. ');
hold off;
xlabel('Number of trees');
ylabel('Resubstitution error');
legend('ErrorGoal=0.15','ErrorGoal=0.01',...
      'AdaBoostM1','Location','NE');
```



All the RobustBoost curves show lower resubstitution error than the AdaBoostM1 curve. The error goal of 0.15 curve shows the lowest resubstitution error over most of the range.

- 8 Generate test data to see the predictive power of the ensembles. Test the four ensembles:

```
Xtest = rand(2000,20);
Ytest = sum(Xtest(:,1:5),2) > 2.5;
idx = randsample(2000,200);
Ytest(idx) = -Ytest(idx);
figure;
plot(loss(rb1,Xtest,Ytest,'mode','cumulative'));
hold on
plot(loss(rb2,Xtest,Ytest,'mode','cumulative'),'r--');
plot(loss(ada,Xtest,Ytest,'mode','cumulative'),'g. ');
hold off;
xlabel('Number of trees');
ylabel('Test error');
legend('ErrorGoal=0.15','ErrorGoal=0.01',...
      'AdaBoostM1','Location','NE');
```



The error curve for error goal 0.15 is lowest (best) in the plotted range. AdaBoostM1 has higher error than the curve for error goal 0.15. The curve for the too-optimistic error goal 0.01 remains substantially higher (worse) than the other algorithms for most of the plotted range.

## Random Subspace Classification

This example shows how to use a random subspace ensemble to increase the accuracy of classification. It also shows how to use cross validation to determine good parameters for both the weak learner template and the ensemble.

### 1. Load the data.

Load the ionosphere data. This data has 351 binary responses to 34 predictors.

```
load ionosphere;  
[N,D] = size(X)
```

```
N =  
    351
```

```
D =  
    34
```

```
resp = unique(Y)
```

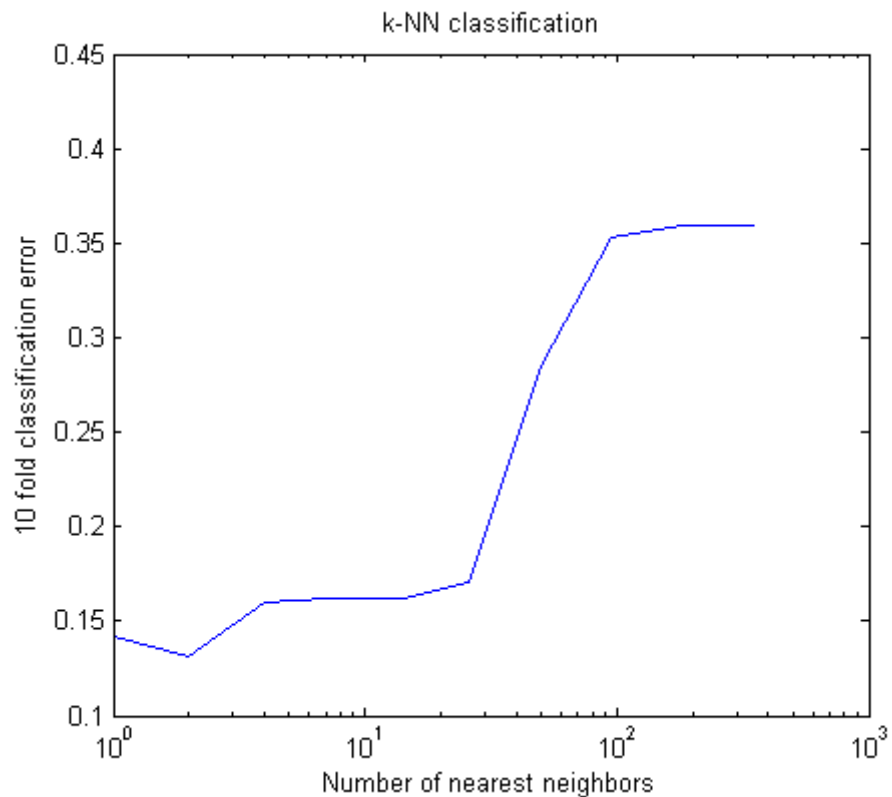
```
resp =  
    'b'  
    'g'
```

## **2. Choose the number of nearest neighbors.**

Find a good choice for  $k$ , the number of nearest neighbors in the classifier, by cross validation. Choose the number of neighbors approximately evenly spaced on a logarithmic scale.

```
rng(8000,'twister') % for reproducibility  
K = round(logspace(0,log10(N),10)); % number of neighbors  
cvloss = zeros(numel(K),1);  
for k=1:numel(K)  
    knn = ClassificationKNN.fit(X,Y,...  
        'NumNeighbors',K(k),'crossval','on');  
    cvloss(k) = kfoldLoss(knn);  
end  
figure; % plot the accuracy versus k  
semilogx(K,cvloss);  
xlabel('Number of nearest neighbors');  
ylabel('10 fold classification error');  
title('k-NN classification');
```





The lowest cross-validation error occurs for  $k = 2$ .

### 3. Create the ensembles.

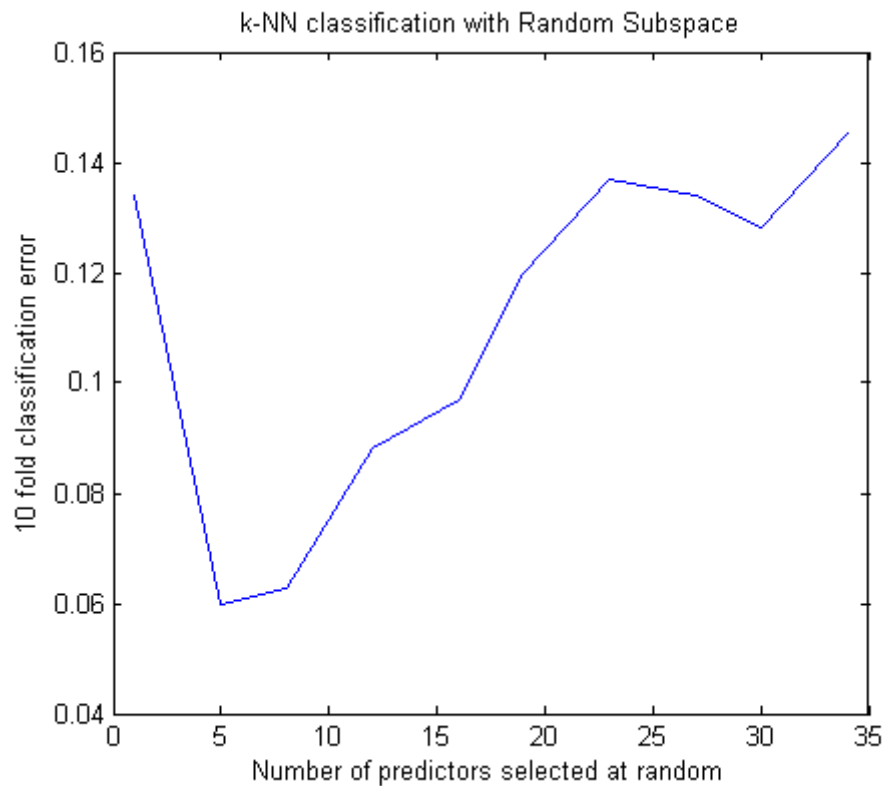
Create ensembles for 2-nearest neighbor classification with various numbers of dimensions, and examine the cross-validated loss of the resulting ensembles.

This step takes a long time. To keep track of the progress, print a message as each dimension finishes.

```
NPredToSample = round(linspace(1,D,10)); % linear spacing of dimensions
cvloss = zeros(numel(NPredToSample),1);
learner = ClassificationKNN.template('NumNeighbors',2);
```

```
for npred=1:numel(NPredToSample)
    subspace = fitensemble(X,Y,'subspace',200,learner,...
        'npredtosample',NPredToSample(npred),'crossval','on');
    cvloss(npred) = kfoldLoss(subspace);
    fprintf('Random Subspace %i done.\n',npred);
end
figure; % plot the accuracy versus dimension
plot(NPredToSample,cvloss);
xlabel('Number of predictors selected at random');
ylabel('10 fold classification error');
title('k-NN classification with Random Subspace');
```

```
Random Subspace 1 done.
Random Subspace 2 done.
Random Subspace 3 done.
Random Subspace 4 done.
Random Subspace 5 done.
Random Subspace 6 done.
Random Subspace 7 done.
Random Subspace 8 done.
Random Subspace 9 done.
Random Subspace 10 done.
```



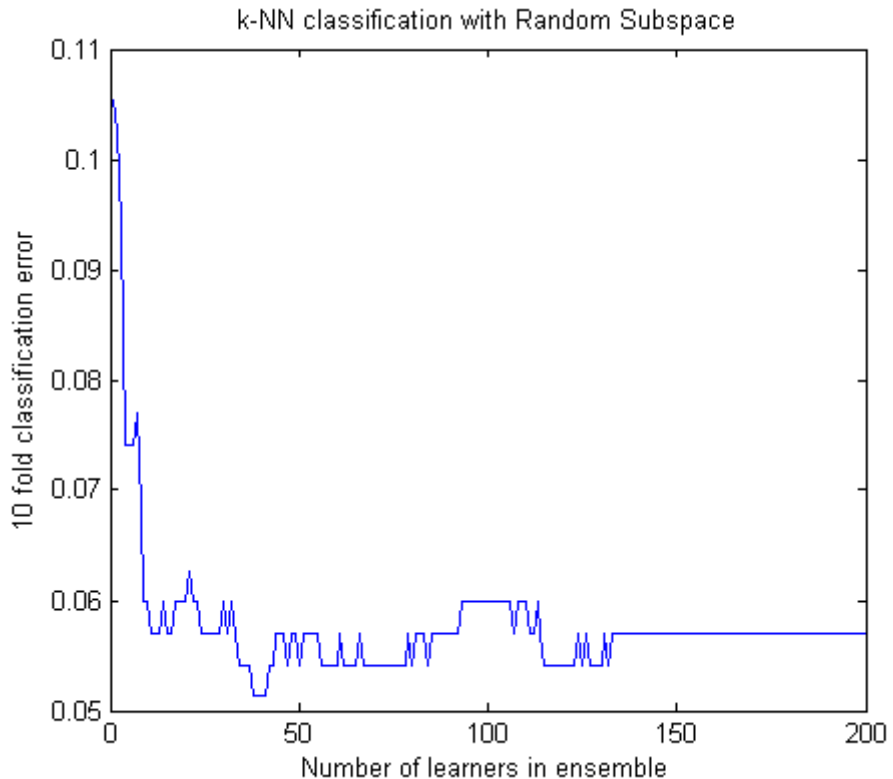
The ensemble with five predictors per learner has the lowest cross-validated error. The error rate, about 0.06, is about half the amount for the individual learners, which was over 0.10.

#### 4. Find a good ensemble size.

Find the smallest number of learners in the ensemble that still give good classification.

```
ens = fitensemble(X,Y,'subspace',200,learner,...
    'npredtosample',5,'crossval','on');
figure; % plot the accuracy versus number in ensemble
plot(kfoldLoss(ens,'mode','cumulative'))
xlabel('Number of learners in ensemble');
```

```
ylabel('10 fold classification error');  
title('k-NN classification with Random Subspace');
```



There seems to be no advantage in an ensemble with more than 150 or so learners. It is not clear whether there is any advantage with more than 50 learners. It is possible that 25 learners gives good predictions.

##### 5. Create a final ensemble.

Construct a final ensemble with 50 learners. Compact the ensemble and see if the compacted version saves an appreciable amount of memory.

```
ens = fitensemble(X,Y,'subspace',50,learner,...  
    'npretdosample',5);
```

```

cens = compact(ens);
s1 = whos('ens');s2 = whos('cens');
[s1.bytes s2.bytes] % si.bytes = size in bytes

ans =
    2317803    2087370

```

The compact ensemble is about 10% smaller than the full ensemble. Both give the same predictions.

## TreeBagger Examples

TreeBagger ensembles have more functionality than those constructed with `fitensemble`; see [TreeBagger Features Not in fitensemble](#) on page 16-151. Also, some property and method names differ from their `fitensemble` counterparts. This section contains examples of workflow for regression and classification that use this extra TreeBagger functionality.

### Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger

In this example, use a database of 1985 car imports with 205 observations, 25 input variables, and one response variable, insurance risk rating, or “symboling.” The first 15 variables are numeric and the last 10 are categorical. The symboling index takes integer values from  $-3$  to  $3$ .

- 1 Load the dataset and split it into predictor and response arrays:

```

load imports-85;
Y = X(:,1);
X = X(:,2:end);

```

- 2 Because bagging uses randomized data drawings, its exact outcome depends on the initial random seed. To reproduce the exact results in this example, use the random stream settings:

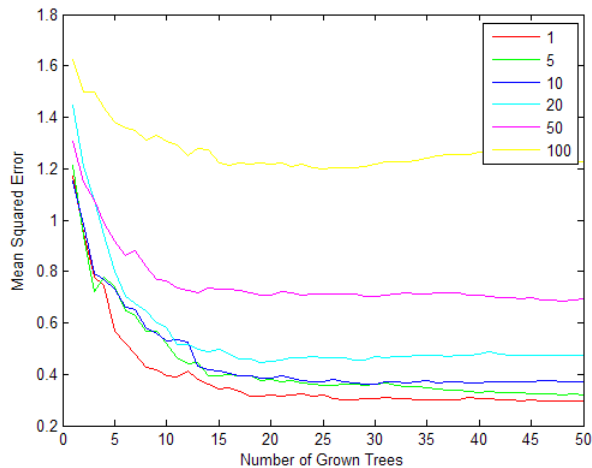
```

rng(1945,'twister')

```

**Finding the Optimal Leaf Size.** For regression, the general rule is to set leaf size to 5 and select one third of input features for decision splits at random. In the following step, verify the optimal leaf size by comparing mean-squared errors obtained by regression for various leaf sizes. `oobError` computes MSE versus the number of grown trees. You must set `oobpred` to 'on' to obtain out-of-bag predictions later.

```
leaf = [1 5 10 20 50 100];
col = 'rgbcmy';
figure(1);
for i=1:length(leaf)
    b = TreeBagger(50,X,Y,'method','r','oobpred','on',...
        'cat',16:25,'minleaf',leaf(i));
    plot(oobError(b),col(i));
    hold on;
end
xlabel('Number of Grown Trees');
ylabel('Mean Squared Error');
legend({'1' '5' '10' '20' '50' '100'},'Location','NorthEast');
hold off;
```



The red (leaf size 1) curve gives the lowest MSE values.

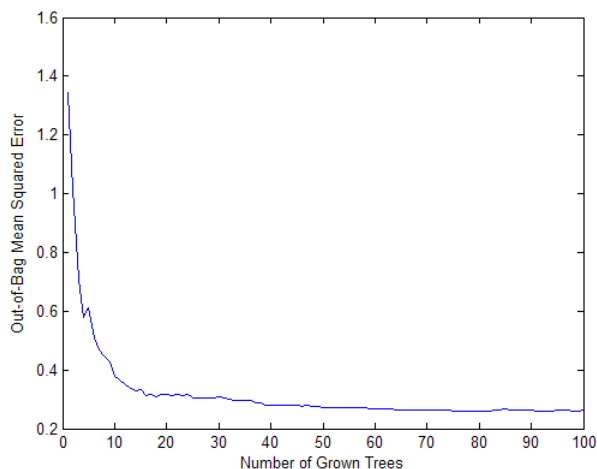
## Estimating Feature Importance.

- 1 In practical applications, you typically grow ensembles with hundreds of trees. Only 50 trees were used in “Finding the Optimal Leaf Size” on page 16-126 for faster processing. Now that you have estimated the optimal leaf size, grow a larger ensemble with 100 trees and use it for estimation of feature importance:

```
b = TreeBagger(100,X,Y,'method','r','oobvarimp','on',...
'cat',16:25,'minleaf',1);
```

- 2 Inspect the error curve again to make sure nothing went wrong during training:

```
figure(2);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Mean Squared Error');
```



Prediction ability should depend more on important features and less on unimportant features. You can use this idea to measure feature importance.

For each feature, you can permute the values of this feature across all of the observations in the data set and measure how much worse the mean-squared

error (MSE) becomes after the permutation. You can repeat this for each feature.

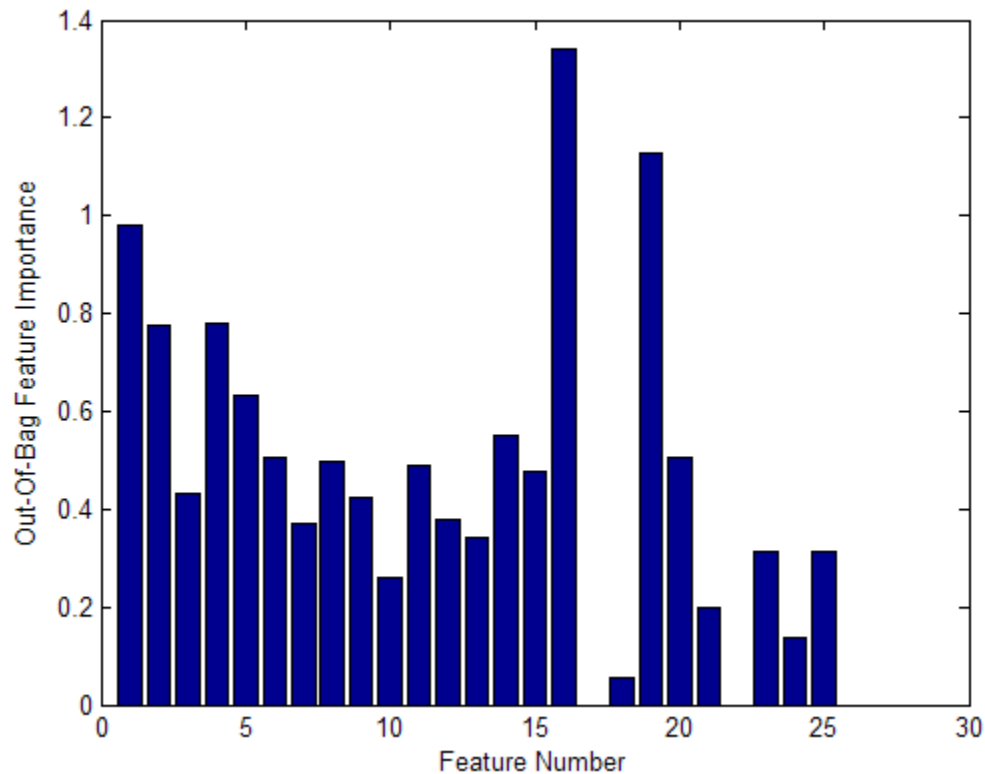
- 1 Using the following code, plot the increase in MSE due to permuting out-of-bag observations across each input variable. The `OOBPermutedVarDeltaError` array stores the increase in MSE averaged over all trees in the ensemble and divided by the standard deviation taken over the trees, for each variable. The larger this value, the more important the variable. Imposing an arbitrary cutoff at 0.65, you can select the five most important features.

```
figure(3);
bar(b.OOBPermutedVarDeltaError);
xlabel('Feature Number');
ylabel('Out-Of-Bag Feature Importance');
idxvar = find(b.OOBPermutedVarDeltaError>0.65)

idxvar =

     1     2     4    16    19
```





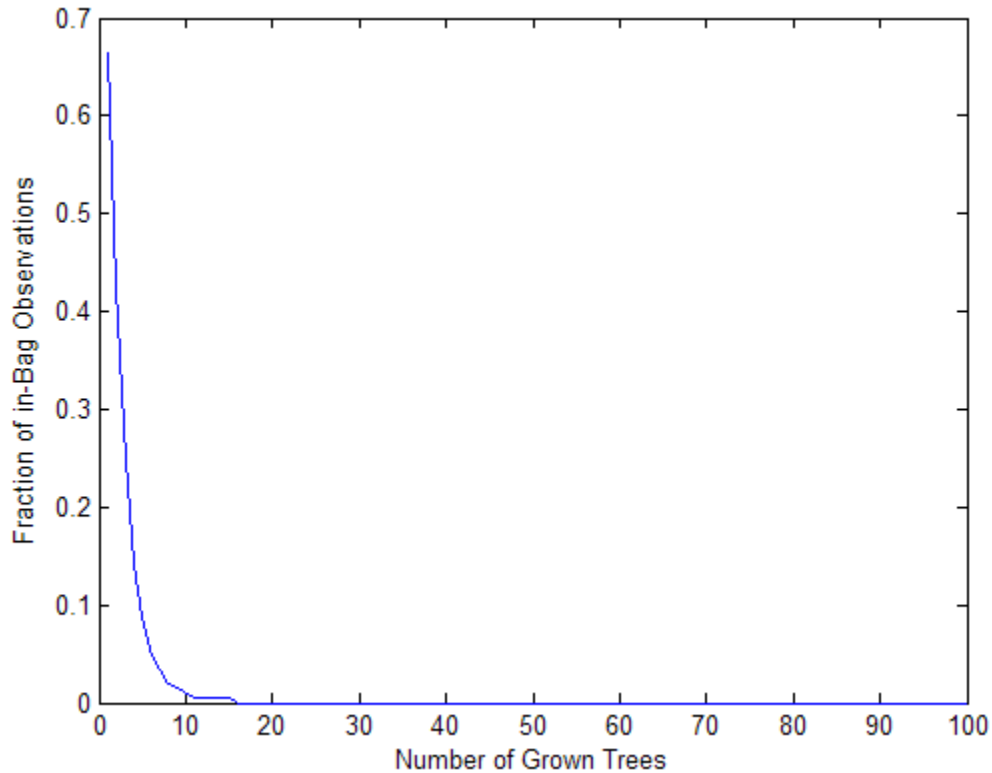
- 2** The `OOBIndices` property of `TreeBagger` tracks which observations are out of bag for what trees. Using this property, you can monitor the fraction of observations in the training data that are in bag for all trees. The curve starts at approximately  $2/3$ , the fraction of unique observations selected by one bootstrap replica, and goes down to 0 at approximately 10 trees.

```

finbag = zeros(1,b.NTrees);
for t=1:b.NTrees
    finbag(t) = sum(all(~b.OOBIndices(:,1:t),2));
end
finbag = finbag / size(X,1);
figure(4);
plot(finbag);

```

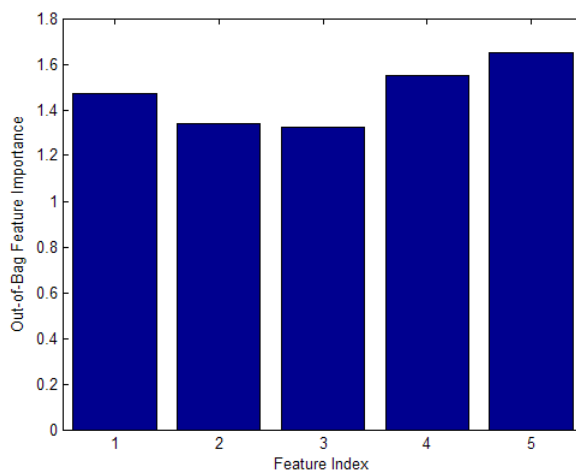
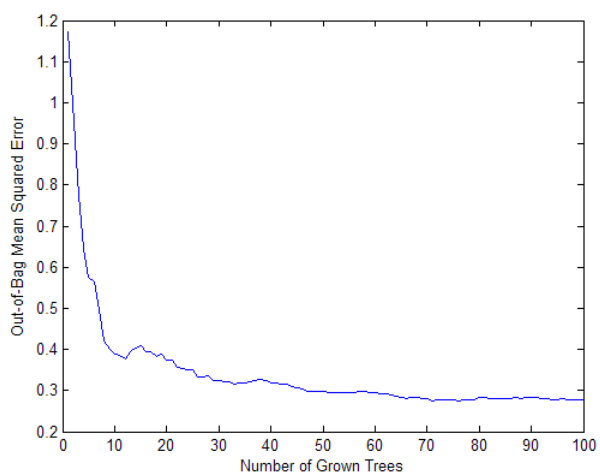
```
xlabel('Number of Grown Trees');  
ylabel('Fraction of in-Bag Observations');
```



**Growing Trees on a Reduced Set of Features.** Using just the five most powerful features selected in “Estimating Feature Importance” on page 16-127, determine if it is possible to obtain a similar predictive power. To begin, grow 100 trees on these features only. The first three of the five selected features are numeric and the last two are categorical.

```
b5v = TreeBagger(100,X(:,idxvar),Y,'method','r',...  
'oobvarimp','on','cat',4:5,'minleaf',1);  
figure(5);  
plot(oobError(b5v));
```

```
xlabel('Number of Grown Trees');  
ylabel('Out-of-Bag Mean Squared Error');  
figure(6);  
bar(b5v.00BPermutedVarDeltaError);  
xlabel('Feature Index');  
ylabel('Out-of-Bag Feature Importance');
```



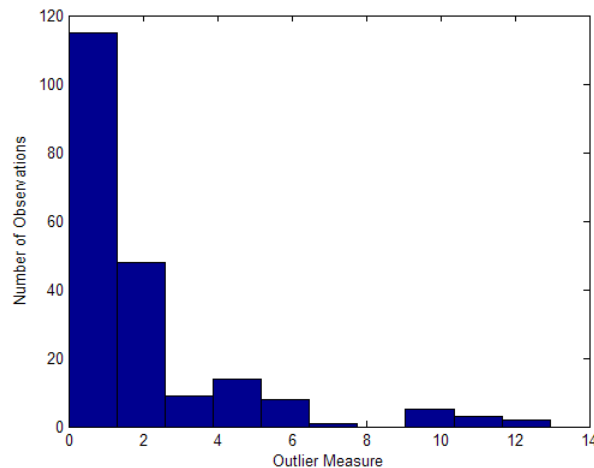
These five most powerful features give the same MSE as the full set, and the ensemble trained on the reduced set ranks these features similarly to each other. Features 1 and 2 from the reduced set perhaps could be removed without a significant loss in the predictive power.

**Finding Outliers.** To find outliers in the training data, compute the proximity matrix using `fillProximities`:

```
b5v = fillProximities(b5v);
```

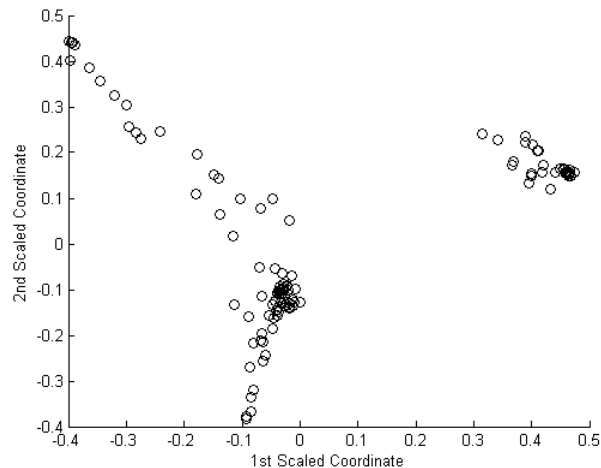
The method normalizes this measure by subtracting the mean outlier measure for the entire sample, taking the magnitude of this difference and dividing the result by the median absolute deviation for the entire sample:

```
figure(7);  
hist(b5v.OutlierMeasure);  
xlabel('Outlier Measure');  
ylabel('Number of Observations');
```



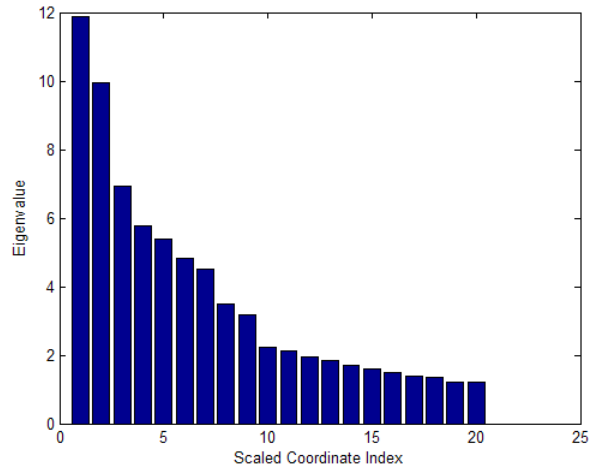
**Discovering Clusters in the Data.** By applying multidimensional scaling to the computed matrix of proximities, you can inspect the structure of the input data and look for possible clusters of observations. The `mdsProx` method returns scaled coordinates and eigenvalues for the computed proximity matrix. If run with the `colors` option, this method makes a scatter plot of two scaled coordinates, first and second by default.

```
figure(8);  
[~,e] = mdsProx(b5v,'colors','k');  
xlabel('1st Scaled Coordinate');  
ylabel('2nd Scaled Coordinate');
```



Assess the relative importance of the scaled axes by plotting the first 20 eigenvalues:

```
figure(9);  
bar(e(1:20));  
xlabel('Scaled Coordinate Index');  
ylabel('Eigenvalue');
```



**Saving the Ensemble Configuration for Future Use.** To use the trained ensemble for predicting the response on unseen data, store the ensemble to disk and retrieve it later. If you do not want to compute predictions for out-of-bag data or reuse training data in any other way, there is no need to store the ensemble object itself. Saving the compact version of the ensemble would be enough in this case. Extract the compact object from the ensemble:

```
c = compact(b5v)
```

```
c =
```

```
Ensemble with 100 decision trees:
```

```
Method:          regression
```

```
Nvars:          5
```

This object can be now saved in a \*.mat file as usual.

### Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger

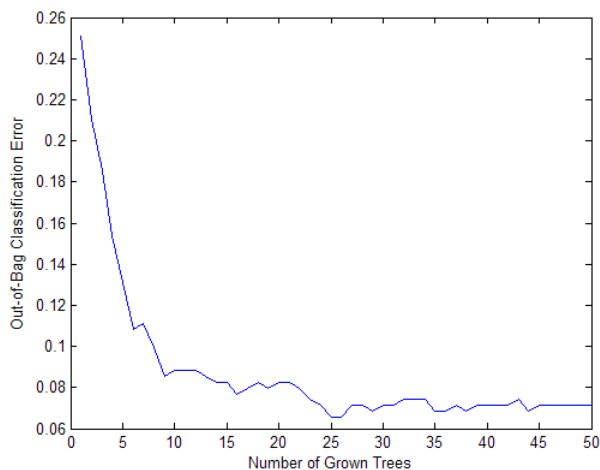
You can also use ensembles of decision trees for classification. For this example, use ionosphere data with 351 observations and 34 real-valued predictors. The response variable is categorical with two levels:

- 'g' for good radar returns
- 'b' for bad radar returns

The goal is to predict good or bad returns using a set of 34 measurements. The workflow resembles that for “Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger” on page 16-125.

- 1 Fix the initial random seed, grow 50 trees, inspect how the ensemble error changes with accumulation of trees, and estimate feature importance. For classification, it is best to set the minimal leaf size to 1 and select the square root of the total number of features for each decision split at random. These are the default settings for a TreeBagger used for classification.

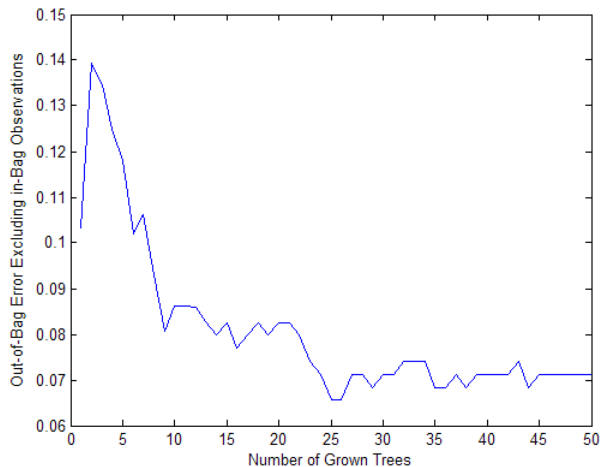
```
load ionosphere;
rng(1945, 'twister')
b = TreeBagger(50,X,Y, 'oobvarimp', 'on');
figure(10);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Classification Error');
```



- 2 The method trains ensembles with few trees on observations that are in bag for all trees. For such observations, it is impossible to compute the true

out-of-bag prediction, and `TreeBagger` returns the most probable class for classification and the sample mean for regression. You can change the default value returned for in-bag observations using the `DefaultYfit` property. If you set the default value to an empty string for classification, the method excludes in-bag observations from computation of the out-of-bag error. In this case, the curve is more variable when the number of trees is small, either because some observations are never out of bag (and are therefore excluded) or because their predictions are based on few trees.

```
b.DefaultYfit = '';
figure(11);
plot(oobError(b));
xlabel('Number of Grown Trees');
ylabel('Out-of-Bag Error Excluding in-Bag Observations');
```



- 3** The `OOBIndices` property of `TreeBagger` tracks which observations are out of bag for what trees. Using this property, you can monitor the fraction of observations in the training data that are in bag for all trees. The curve starts at approximately  $2/3$ , the fraction of unique observations selected by one bootstrap replica, and goes down to 0 at approximately 10 trees.

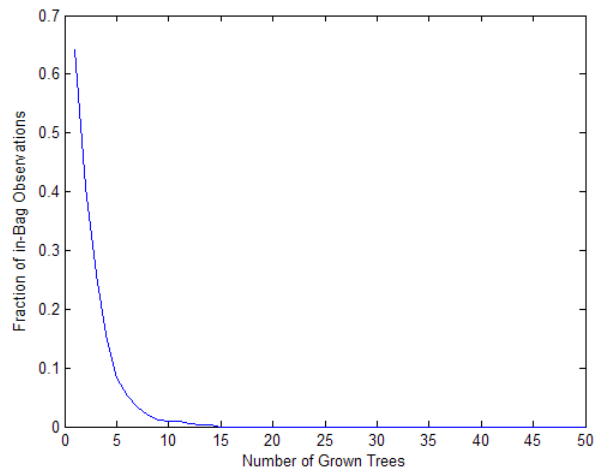
```
finbag = zeros(1,b.NTrees);
for t=1:b.NTrees
    finbag(t) = sum(all(~b.OOBIndices(:,1:t),2));
```



```

end
finbag = finbag / size(X,1);
figure(12);
plot(finbag);
xlabel('Number of Grown Trees');
ylabel('Fraction of in-Bag Observations');

```



#### 4 Estimate feature importance:

```

figure(13);
bar(b.OOBPermutedVarDeltaError);
xlabel('Feature Index');
ylabel('Out-of-Bag Feature Importance');
idxvar = find(b.OOBPermutedVarDeltaError>0.8)

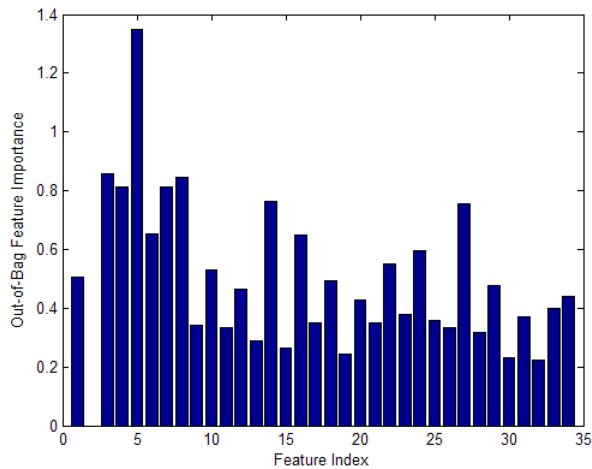
idxvar =

```

```

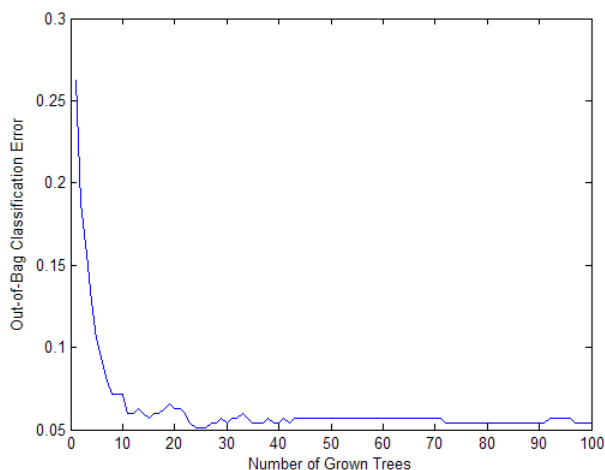
    3     4     5     7     8

```



- 5 Having selected the five most important features, grow a larger ensemble on the reduced feature set. Save time by not permuting out-of-bag observations to obtain new estimates of feature importance for the reduced feature set (set `oobvarimp` to `'off'`). You would still be interested in obtaining out-of-bag estimates of classification error (set `oobpred` to `'on'`).

```
b5v = TreeBagger(100,X(:,idxvar),Y,'oobpred','on');  
figure(14);  
plot(oobError(b5v));  
xlabel('Number of Grown Trees');  
ylabel('Out-of-Bag Classification Error');
```

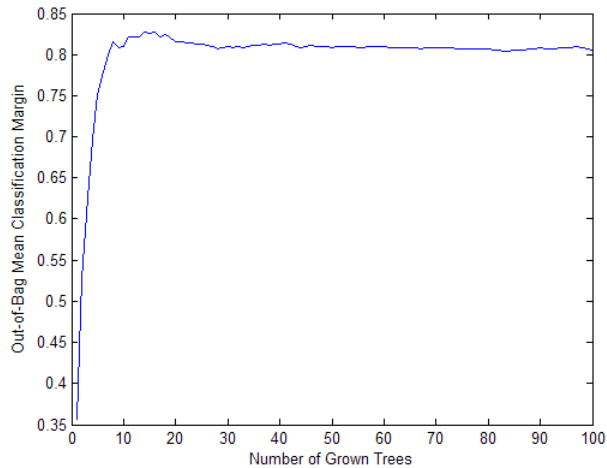


- 6 For classification ensembles, in addition to classification error (fraction of misclassified observations), you can also monitor the average classification margin. For each observation, the *margin* is defined as the difference between the score for the true class and the maximal score for other classes predicted by this tree. The cumulative classification margin uses the scores averaged over all trees and the mean cumulative classification margin is the cumulative margin averaged over all observations. The `oobMeanMargin` method with the `'mode'` argument set to `'cumulative'` (default) shows how the mean cumulative margin changes as the ensemble grows: every new element in the returned array represents the cumulative margin obtained by including a new tree in the ensemble. If training is successful, you would expect to see a gradual increase in the mean classification margin.

For decision trees, a classification score is the probability of observing an instance of this class in this tree leaf. For example, if the leaf of a grown decision tree has five 'good' and three 'bad' training observations in it, the scores returned by this decision tree for any observation fallen on this leaf are  $5/8$  for the 'good' class and  $3/8$  for the 'bad' class. These probabilities are called 'scores' for consistency with other classifiers that might not have an obvious interpretation for numeric values of returned predictions.

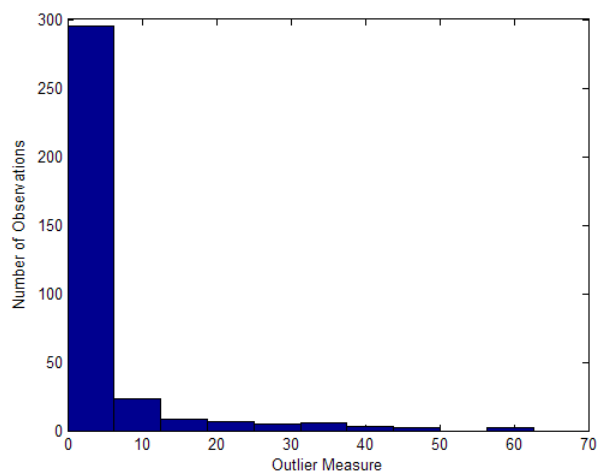
```
figure(15);
plot(oobMeanMargin(b5v));
```

```
xlabel('Number of Grown Trees');  
ylabel('Out-of-Bag Mean Classification Margin');
```



- 7** Compute the matrix of proximities and look at the distribution of outlier measures. Unlike regression, outlier measures for classification ensembles are computed within each class separately.

```
b5v = fillProximities(b5v);  
figure(16);  
hist(b5v.OutlierMeasure);  
xlabel('Outlier Measure');  
ylabel('Number of Observations');
```



- 8 All extreme outliers for this dataset come from the 'good' class:

```
b5v.Y(b5v.OutlierMeasure>40)
```

```
ans =
```

```
'g'
'g'
'g'
'g'
'g'
```

- 9 As for regression, you can plot scaled coordinates, displaying the two classes in different colors using the `colors` argument of `mdsProx`. This argument takes a string in which every character represents a color. To find the order of classes used by the ensemble, look at the `ClassNames` property:

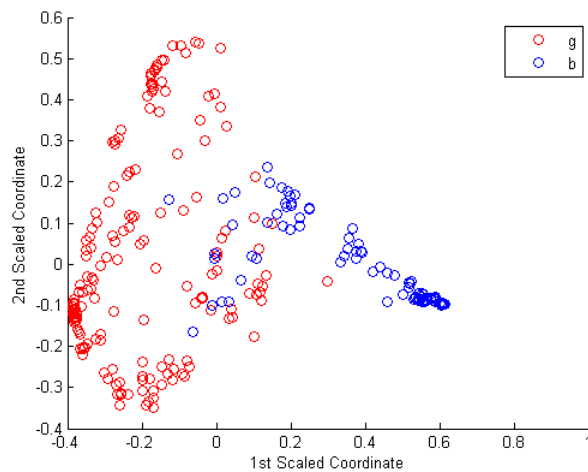
```
b5v.ClassNames
```

```
ans =
```

```
'g'
'b'
```

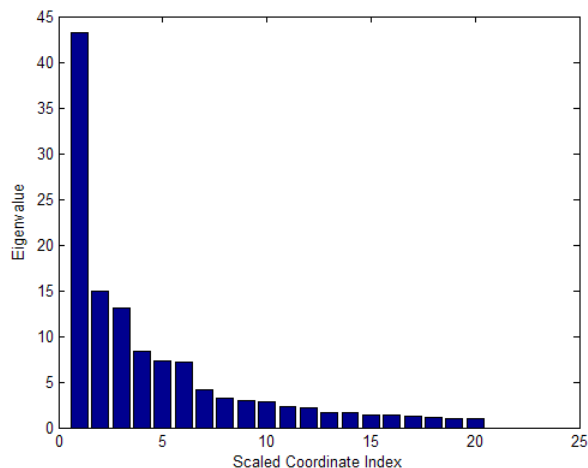
The 'good' class is first and the 'bad' class is second. Display scaled coordinates using red for 'good' and blue for 'bad' observations:

```
figure(17);  
[s,e] = mdsProx(b5v,'colors','rb');  
xlabel('1st Scaled Coordinate');  
ylabel('2nd Scaled Coordinate');
```



- 10** Plot the first 20 eigenvalues obtained by scaling. The first eigenvalue in this case clearly dominates and the first scaled coordinate is most important.

```
figure(18);  
bar(e(1:20));  
xlabel('Scaled Coordinate Index');  
ylabel('Eigenvalue');
```



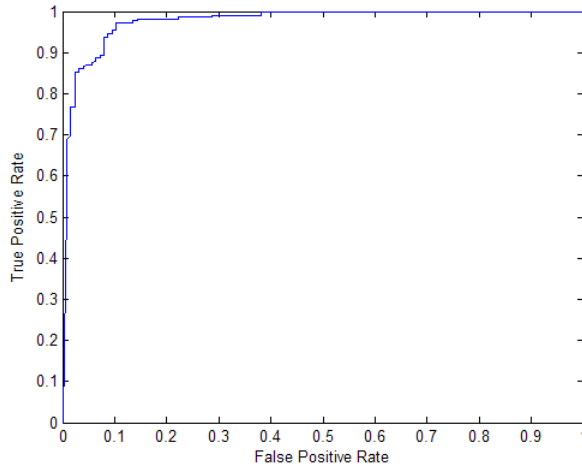
**Plotting a Classification Performance Curve.** Another way of exploring the performance of a classification ensemble is to plot its Receiver Operating Characteristic (ROC) curve or another performance curve suitable for the current problem. First, obtain predictions for out-of-bag observations. For a classification ensemble, the `oobPredict` method returns a cell array of classification labels ('g' or 'b' for ionosphere data) as the first output argument and a numeric array of scores as the second output argument. The returned array of scores has two columns, one for each class. In this case, the first column is for the 'good' class and the second column is for the 'bad' class. One column in the score matrix is redundant because the scores represent class probabilities in tree leaves and by definition add up to 1.

```
[Yfit,Sfit] = oobPredict(b5v);
```

Use the `perfcurve` utility (see “Performance Curves” on page 15-41) to compute a performance curve. By default, `perfcurve` returns the standard ROC curve, which is the true positive rate versus the false positive rate. `perfcurve` requires true class labels, scores, and the positive class label for input. In this case, choose the 'good' class as positive. The scores for this class are in the first column of `Sfit`.

```
[fpr,tpr] = perfcurve(b5v.Y,Sfit(:,1),'g');
figure(19);
plot(fpr,tpr);
```

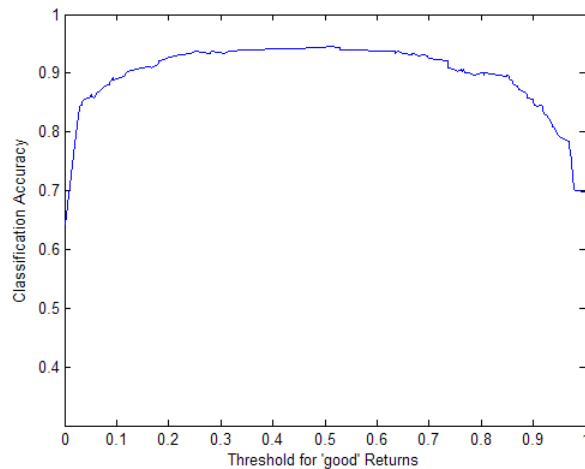
```
xlabel('False Positive Rate');
ylabel('True Positive Rate');
```



Instead of the standard ROC curve, you might want to plot, for example, ensemble accuracy versus threshold on the score for the 'good' class. The `ycrit` input argument of `perfcurve` lets you specify the criterion for the  $y$ -axis, and the third output argument of `perfcurve` returns an array of thresholds for the positive class score. Accuracy is the fraction of correctly classified observations, or equivalently, 1 minus the classification error.

```
[fpr,accu,thre] = perfcurve(b5v.Y,Sfit(:,1),'g','ycrit','accu');
figure(20);
plot(thre,accu);
xlabel('Threshold for ''good'' Returns');
ylabel('Classification Accuracy');
```





The curve shows a flat region indicating that any threshold from 0.2 to 0.6 is a reasonable choice. By default, the function assigns classification labels using 0.5 as the boundary between the two classes. You can find exactly what accuracy this corresponds to:

```
i50 = find(accu>=0.50,1,'first')
accu(abs(thre-0.5)<eps)
```

```
returns
```

```
i50 =
     2
```

```
ans =
     0.9430
```

The maximal accuracy is a little higher than the default one:

```
[maxaccu,iaccu] = max(accu)
```

```
returns
```

```
maxaccu =
     0.9459
```

```
iaccu =  
    91
```

The optimal threshold is therefore:

```
thre(iaccu)
```

```
ans =  
    0.5056
```

## Ensemble Algorithms

- “AdaBoostM1” on page 16-146
- “AdaBoostM2” on page 16-148
- “Bag” on page 16-149
- “GentleBoost” on page 16-153
- “LogitBoost” on page 16-153
- “LPBoost” on page 16-155
- “LSBoost” on page 16-156
- “RobustBoost” on page 16-157
- “RUSBoost” on page 16-158
- “Subspace” on page 16-159
- “TotalBoost” on page 16-160

### AdaBoostM1

AdaBoostM1 is a very popular boosting algorithm for binary classification. The algorithm trains learners sequentially. For every learner with index  $t$ , AdaBoostM1 computes the weighted classification error

$$\varepsilon_t = \sum_{n=1}^N d_n^{(t)} \mathbb{I}(y_n \neq h_t(x_n)),$$

where

- $x_n$  is a vector of predictor values for observation  $n$ .
- $y_n$  is the true class label.
- $h_t$  is the prediction of learner (hypothesis) with index  $t$ .
- $\mathbb{I}$  is the indicator function.
- $d_n^{(t)}$  is the weight of observation  $n$  at step  $t$ .

AdaBoostM1 then increases weights for observations misclassified by learner  $t$  and reduces weights for observations correctly classified by learner  $t$ . The next learner  $t + 1$  is then trained on the data with updated weights  $d_n^{(t+1)}$ .

After training finishes, AdaBoostM1 computes prediction for new data using

$$f(x) = \sum_{t=1}^T \alpha_t h_t(x),$$

where

$$\alpha_t = \frac{1}{2} \log \frac{1 - \varepsilon_t}{\varepsilon_t}$$

are weights of the weak hypotheses in the ensemble.

Training by AdaBoostM1 can be viewed as stagewise minimization of the exponential loss

$$\sum_{n=1}^N w_n \exp(-y_n f(x_n)),$$

where

- $y_n \in \{-1, +1\}$  is the true class label.

- $w_n$  are observation weights normalized to add up to 1.
- $f(x_n) \in (-\infty, +\infty)$  is the predicted classification score.

The observation weights  $w_n$  are the original observation weights you passed to `fitensemble`.

The second output from the `predict` method of an `AdaBoostM1` classification ensemble is an  $N$ -by-2 matrix of classification scores for the two classes and  $N$  observations. The second column in this matrix is always equal to minus the first column. `predict` returns two scores to be consistent with multiclass models, though this is redundant because the second column is always the negative of the first.

Most often `AdaBoostM1` is used with decision stumps (default) or shallow trees. If boosted stumps give poor performance, try setting the minimal parent node size to one quarter of the training data.

By default, the learning rate for boosting algorithms is 1. If you set the learning rate to a lower number, the ensemble learns at a slower rate, but can converge to a better solution. 0.1 is a popular choice for the learning rate. Learning at a rate less than 1 is often called “shrinkage”.

For examples using `AdaBoostM1`, see “Tuning `RobustBoost`” on page 16-115.

For references related to `AdaBoostM1`, see Freund and Schapire [9], Schapire et al. [15], Friedman, Hastie, and Tibshirani [11], and Friedman [10].

## AdaBoostM2

`AdaBoostM2` is an extension of `AdaBoostM1` for multiple classes. Instead of weighted classification error, `AdaBoostM2` uses weighted pseudo-loss for  $N$  observations and  $K$  classes

$$\varepsilon_t = \frac{1}{2} \sum_{n=1}^N \sum_{k \neq y_n} d_{n,k}^{(t)} (1 - h_t(x_n, y_n) + h_t(x_n, k)),$$

where

- $h_t(x_n, k)$  is the confidence of prediction by learner at step  $t$  into class  $k$  ranging from 0 (not at all confident) to 1 (highly confident).
- $d_{n,k}^{(t)}$  are observation weights at step  $t$  for class  $k$ .
- $y_n$  is the true class label taking one of the  $K$  values.
- The second sum is over all classes other than the true class  $y_n$ .

Interpreting the pseudo-loss is harder than classification error, but the idea is the same. Pseudo-loss can be used as a measure of the classification accuracy from any learner in an ensemble. Pseudo-loss typically exhibits the same behavior as a weighted classification error for AdaBoostM1: the first few learners in a boosted ensemble give low pseudo-loss values. After the first few training steps, the ensemble begins to learn at a slower pace, and the pseudo-loss value approaches 0.5 from below.

For examples using AdaBoostM2, see “Create a Classification Ensemble” on page 16-69.

For references related to AdaBoostM2, see Freund and Schapire [9].

## Bag

*Bagging*, which stands for “bootstrap aggregation,” is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, generate many bootstrap replicas of this dataset and grow decision trees on these replicas. Obtain each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, take an average over predictions from individual trees.

Bagged decision trees were introduced in MATLAB R2009a as `TreeBagger`. The `fitensemble` function lets you bag in a manner consistent with boosting. An ensemble of bagged trees, either `ClassificationBaggedEnsemble` or `RegressionBaggedEnsemble`, returned by `fitensemble` offers almost the same functionality as `TreeBagger`. Discrepancies between `TreeBagger` and the new framework are described in detail in `TreeBagger Features Not in fitensemble` on page 16-151.

Bagging works by training learners on resampled versions of the data. This resampling is usually done by bootstrapping observations, that is, selecting  $N$  out of  $N$  observations with replacement for every new learner. In addition, every tree in the ensemble can randomly select predictors for decision splits—a technique known to improve the accuracy of bagged trees.

By default, the minimal leaf sizes for bagged trees are set to 1 for classification and 5 for regression. Trees grown with the default leaf size are usually very deep. These settings are close to optimal for the predictive power of an ensemble. Often you can grow trees with larger leaves without losing predictive power. Doing so reduces training and prediction time, as well as memory usage for the trained ensemble.

Another important parameter is the number of predictors selected at random for every decision split. This random selection is made for every split, and every deep tree involves many splits. By default, this parameter is set to a square root of the number of predictors for classification, and one third of predictors for regression.

Several features of bagged decision trees make them a unique algorithm. Drawing  $N$  out of  $N$  observations with replacement omits on average 37% of observations for each decision tree. These are “out-of-bag” observations. You can use them to estimate the predictive power and feature importance. For each observation, you can estimate the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. You can then compare the computed prediction against the observed response for this observation. By comparing the out-of-bag predicted responses against the observed responses for all observations used for training, you can estimate the average out-of-bag error. This out-of-bag average is an unbiased estimator of the true ensemble error. You can also obtain out-of-bag estimates of feature importance by randomly permuting out-of-bag data across one variable or column at a time and estimating the increase in the out-of-bag error due to this permutation. The larger the increase, the more important the feature. Thus, you need not supply test data for bagged ensembles because you obtain reliable estimates of the predictive power and feature importance in the process of training, which is an attractive feature of bagging.

Another attractive feature of bagged decision trees is the proximity matrix. Every time two observations land on the same leaf of a tree, their proximity

increases by 1. For normalization, sum these proximities over all trees in the ensemble and divide by the number of trees. The resulting matrix is symmetric with diagonal elements equal to 1 and off-diagonal elements ranging from 0 to 1. You can use this matrix for finding outlier observations and discovering clusters in the data through multidimensional scaling.

For examples using bagging, see:

- “Example: Test Ensemble Quality” on page 16-72
- “Surrogate Splits” on page 16-95
- “Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger” on page 16-125
- “Workflow Example: Classifying Radar Returns for Ionosphere Data with TreeBagger” on page 16-134

For references related to bagging, see Breiman [3], [4], and [5].

**Comparison of TreeBagger and Bagged Ensembles.** `fitensemble` produces bagged ensembles that have most, but not all, of the functionality of TreeBagger objects. Additionally, some functionality has different names in the new bagged ensembles.

### TreeBagger Features Not in `fitensemble`

Feature	TreeBagger Property	TreeBagger Method
Computation of proximity matrix	Proximity	<code>fillProximities</code> , <code>mdsProx</code>
Computation of outliers	OutlierMeasure	N/A
Out-of-bag estimates of predictor importance	<code>OOBPermutedVarDeltaError</code> , <code>OOBPermutedVarDeltaMeanMargin</code> , <code>OOBPermutedVarCountRaiseMargin</code>	N/A
Merging two ensembles trained separately	N/A	<code>append</code>
Parallel computation for creating ensemble	Set the <code>UseParallel</code> name-value pair to <code>true</code>	N/A

### Differing Names Between TreeBagger and Bagged Ensembles

Feature	TreeBagger	Bagged Ensembles
Split criterion contributions for each predictor	DeltaCritDecisionSplit property	First output of predictorImportance (classification) or predictorImportance (regression)
Predictor associations	VarAssoc property	Second output of predictorImportance (classification) or predictorImportance (regression)
Error (misclassification probability or mean-squared error)	error and oobError methods	loss and oobLoss methods (classification); loss and oobLoss methods (regression)
Train additional trees and add to ensemble	growTrees method	resume method (classification); resume method (regression)
Mean classification margin per tree	meanMargin and oobMeanMargin methods	edge and oobEdge methods (classification)

In addition, two important changes were made to training and prediction for bagged classification ensembles:

- If you pass a misclassification cost matrix to `TreeBagger`, it passes this matrix along to the trees. If you pass a misclassification cost matrix to `fitensemble`, it uses this matrix to adjust the class prior probabilities. `fitensemble` then passes the adjusted prior probabilities and the default cost matrix to the trees. The default cost matrix is `ones(K) - eye(K)` for  $K$  classes.
- Unlike the `loss` and `edge` methods in the new framework, the `TreeBagger` `error` and `meanMargin` methods do not normalize input observation weights of the prior probabilities in the respective class.



## GentleBoost

GentleBoost (also known as Gentle AdaBoost) combines features of AdaBoostM1 and LogitBoost. Like AdaBoostM1, GentleBoost minimizes the exponential loss. But its numeric optimization is set up differently. Like LogitBoost, every weak learner fits a regression model to response values  $y_n \in \{-1, +1\}$ . This makes GentleBoost another good candidate for binary classification of data with multilevel categorical predictors.

`fitensemble` computes and stores the mean-squared error in the `FitInfo` property of the ensemble object. The mean-squared error is

$$\sum_{n=1}^N d_n^{(t)} (\tilde{y}_n - h_t(x_n))^2,$$

where

- $d_n^{(t)}$  are observation weights at step  $t$  (the weights add up to 1).
- $h_t(x_n)$  are predictions of the regression model  $h_t$  fitted to response values  $y_n$ .

As the strength of individual learners weakens, the weighted mean-squared error approaches 1.

For examples using GentleBoost, see “Example: Unequal Classification Costs” on page 16-85 and “Classification with Many Categorical Levels” on page 16-91.

For references related to GentleBoost, see Friedman, Hastie, and Tibshirani [11].

## LogitBoost

LogitBoost is another popular algorithm for binary classification. LogitBoost works similarly to AdaBoostM1, except it minimizes binomial deviance

$$\sum_{n=1}^N w_n \log(1 + \exp(-2y_n f(x_n))),$$

where

- $y_n \in \{-1, +1\}$  is the true class label.
- $w_n$  are observation weights normalized to add up to 1.
- $f(x_n) \in (-\infty, +\infty)$  is the predicted classification score.

Binomial deviance assigns less weight to badly misclassified observations (observations with large negative values of  $y_n f(x_n)$ ). `LogitBoost` can give better average accuracy than `AdaBoostM1` for data with poorly separable classes.

Learner  $t$  in a `LogitBoost` ensemble fits a regression model to response values

$$\tilde{y}_n = \frac{y_n^* - p_t(x_n)}{p_t(x_n)(1 - p_t(x_n))},$$

where

- $y_n^* \in \{0, +1\}$  are relabeled classes (0 instead of  $-1$ ).
- $p_t(x_n)$  is the current ensemble estimate of the probability for observation  $x_n$  to be of class 1.

Fitting a regression model at each boosting step turns into a great computational advantage for data with multilevel categorical predictors. Take a categorical predictor with  $L$  levels. To find the optimal decision split on such a predictor, classification tree needs to consider  $2^{L-1} - 1$  splits. A regression tree needs to consider only  $L - 1$  splits, so the processing time can be much shorter. `LogitBoost` is recommended for categorical predictors with many levels.

`fitensemble` computes and stores the mean-squared error in the `FitInfo` property of the ensemble object. The mean-squared error is

$$\sum_{n=1}^N a_n^{(t)} (\tilde{y}_n - h_t(x_n))^2,$$

where

- $d_n^{(t)}$  are observation weights at step  $t$  (the weights add up to 1).
- $h_t(x_n)$  are predictions of the regression model  $h_t$  fitted to response values  $\tilde{y}_n$ .

Values  $y_n$  can range from  $-\infty$  to  $+\infty$ , so the mean-squared error does not have well-defined bounds.

For examples using **LogitBoost**, see “Classification with Many Categorical Levels” on page 16-91.

For references related to **LogitBoost**, see Friedman, Hastie, and Tibshirani [11].

## LPBoost

**LPBoost** (linear programming boost), like **TotalBoost**, performs multiclass classification by attempting to maximize the minimal *margin* in the training set. This attempt uses optimization algorithms, namely linear programming for **LPBoost**. So you need an Optimization Toolbox license to use **LPBoost** or **TotalBoost**.

The margin of a classification is the difference between the predicted soft classification *score* for the true class, and the largest score for the false classes. For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node. For more information, see “Definitions” on page 21-1868 in *margin*.

Why maximize the minimal margin? For one thing, the generalization error (the error on new data) is the probability of obtaining a negative margin. Schapire and Singer [16] establish this inequality on the probability of obtaining a negative margin:

$$P_{\text{test}}(m \leq 0) \leq P_{\text{train}}(m \leq \theta) + O\left(\frac{1}{\sqrt{N}} \sqrt{\frac{V \log^2(N/V)}{\theta^2} + \log(1/\delta)}\right).$$

Here  $m$  is the margin,  $\theta$  is any positive number,  $V$  is the Vapnik-Chervonenkis dimension of the classifier space,  $N$  is the size of the training set, and  $\delta$  is a small positive number. The inequality holds with probability  $1-\delta$  over many i.i.d. training and test sets. This inequality says: To obtain a low generalization error, minimize the number of observations below margin  $\theta$  in the training set.

LPBoost iteratively maximizes the minimal margin through a sequence of linear programming problems. Equivalently, by duality, LPBoost minimizes the maximal *edge*, where edge is the weighted mean margin (see “Definitions” on page 21-765). At each iteration, there are more constraints in the problem. So, for large problems, the optimization problem becomes increasingly constrained, and slow to solve.

LPBoost typically creates ensembles with many learners having weights that are orders of magnitude smaller than those of other learners. Therefore, to better enable you to remove the unimportant ensemble members, the compact method reorders the members of an LPBoost ensemble from largest weight to smallest. Therefore, you can easily remove the least important members of the ensemble using the `removeLearners` method.

For examples using LPBoost, see “LPBoost and TotalBoost for Small Ensembles” on page 16-98.

For references related to LPBoost, see Warmuth, Liao, and Ratsch [18].

## LSBoost

LSBoost (least squares boosting) fits regression ensembles. At every step, the ensemble fits a new learner to the difference between the observed response and the aggregated prediction of all learners grown previously. The ensemble fits to minimize mean-squared error.

You can use LSBoost with shrinkage by passing in the `LearnRate` parameter. By default this parameter is set to 1, and the ensemble learns at the maximal speed. If you set `LearnRate` to a value from 0 to 1, the ensemble fits every new learner to  $y_n - \eta f(x_n)$ , where

- $y_n$  is the observed response.

- $f(x_n)$  is the aggregated prediction from all weak learners grown so far for observation  $x_n$ .
- $\eta$  is the learning rate.

For examples using `LSBoost`, see “Creating a Regression Ensemble” on page 16-70 and “Example: Regularizing a Regression Ensemble” on page 16-105.

For references related to `LSBoost`, see Hastie, Tibshirani, and Friedman [12], Chapters 7 (Model Assessment and Selection) and 15 (Random Forests).

## RobustBoost

Boosting algorithms such as `AdaBoostM1` and `LogitBoost` increase weights for misclassified observations at every boosting step. These weights can become very large. If this happens, the boosting algorithm sometimes concentrates on a few misclassified observations and neglects the majority of training data. Consequently the average classification accuracy suffers.

In this situation, you can try using `RobustBoost`. This algorithm does not assign almost the entire data weight to badly misclassified observations. It can produce better average classification accuracy.

Unlike `AdaBoostM1` and `LogitBoost`, `RobustBoost` does not minimize a specific loss function. Instead, it maximizes the number of observations with the classification margin above a certain threshold.

`RobustBoost` trains based on time evolution. The algorithm starts at  $t = 0$ . At every step, `RobustBoost` solves an optimization problem to find a positive step in time  $\Delta t$  and a corresponding positive change in the average margin for training data  $\Delta m$ . `RobustBoost` stops training and exits if at least one of these three conditions is true:

- Time  $t$  reaches 1.
- `RobustBoost` cannot find a solution to the optimization problem with positive updates  $\Delta t$  and  $\Delta m$ .
- `RobustBoost` grows as many learners as you requested.

Results from `RobustBoost` can be usable for any termination condition. Estimate the classification accuracy by cross validation or by using an independent test set.

To get better classification accuracy from `RobustBoost`, you can adjust three parameters in `fitensemble`: `RobustErrorGoal`, `RobustMaxMargin`, and `RobustMarginSigma`. Start by varying values for `RobustErrorGoal` from 0 to 1. The maximal allowed value for `RobustErrorGoal` depends on the two other parameters. If you pass a value that is too high, `fitensemble` produces an error message showing the allowed range for `RobustErrorGoal`.

For examples using `RobustBoost`, see “Tuning `RobustBoost`” on page 16-115.

For references related to `RobustBoost`, see Freund [8].

### **RUSBoost**

`RUSBoost` is especially effective at classifying imbalanced data, meaning some class in the training data has many fewer members than another. `RUS` stands for Random Under Sampling. The algorithm takes  $N$ , the number of members in the class with the fewest members in the training data, as the basic unit for sampling. Classes with more members are under sampled by taking only  $N$  observations of every class. In other words, if there are  $K$  classes, then, for each weak learner in the ensemble, `RUSBoost` takes a subset of the data with  $N$  observations from each of the  $K$  classes. The boosting procedure follows the procedure in “`AdaBoostM2`” on page 16-148 for reweighting and constructing the ensemble.

When you construct a `RUSBoost` ensemble, there is an optional name-value pair called `RatioToSmallest`. Give a vector of  $K$  values, each value representing the multiple of  $N$  to sample for the associated class. For example, if the smallest class has  $N = 100$  members, then `RatioToSmallest = [2,3,4]` means each weak learner has 200 members in class 1, 300 in class 2, and 400 in class 3. If `RatioToSmallest` leads to a value that is larger than the number of members in a particular class, then `RUSBoost` samples the members with replacement. Otherwise, `RUSBoost` samples the members without replacement.

For examples using `RUSBoost`, see “Classification with Imbalanced Data” on page 16-77.

For references related to RUSBoost, see Seiffert et al. [17].

## Subspace

Use random subspace ensembles (`Subspace`) to improve the accuracy of discriminant analysis (`ClassificationDiscriminant`) or  $k$ -nearest neighbor (`ClassificationKNN`) classifiers. `Subspace` ensembles also have the advantage of using less memory than ensembles with all predictors, and can handle missing values (NaNs).

The basic random subspace algorithm uses these parameters.

- $m$  is the number of dimensions (variables) to sample in each learner. Set  $m$  using the `NPredToSample` name-value pair.
- $d$  is the number of dimensions in the data, which is the number of columns (predictors) in the data matrix  $X$ .
- $n$  is the number of learners in the ensemble. Set  $n$  using the `NLearn` input.

The basic random subspace algorithm performs the following steps:

- 1** Choose without replacement a random set of  $m$  predictors from the  $d$  possible values.
- 2** Train a weak learner using just the  $m$  chosen predictors.
- 3** Repeat steps 1 and 2 until there are  $n$  weak learners.
- 4** Predict by taking an average of the score prediction of the weak learners, and classify the category with the highest average score.

You can choose to create a weak learner for every possible set of  $m$  predictors from the  $d$  dimensions. To do so, set  $n$ , the number of learners, to `'AllPredictorCombinations'`. In this case, there are `nchoosek(size(X,2),NPredToSample)` weak learners in the ensemble.

`fitensemble` downweights predictors after choosing them for a learner, so subsequent learners have a lower chance of using a predictor that was previously used. This weighting tends to make predictors more evenly distributed among learners than in uniform weighting.

For examples using `Subspace`, see “Random Subspace Classification” on page 16-119.

For references related to random subspace ensembles, see Ho [13].

### TotalBoost

`TotalBoost`, like linear programming boost (`LPBoost`), performs multiclass classification by attempting to maximize the minimal *margin* in the training set. This attempt uses optimization algorithms, namely quadratic programming for `TotalBoost`. So you need an Optimization Toolbox license to use `LPBoost` or `TotalBoost`.

The margin of a classification is the difference between the predicted soft classification *score* for the true class, and the largest score for the false classes. For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node. For more information, see “Definitions” on page 21-1868 in `margin`.

Why maximize the minimal margin? For one thing, the generalization error (the error on new data) is the probability of obtaining a negative margin. Schapire and Singer [16] establish this inequality on the probability of obtaining a negative margin:

$$P_{\text{test}}(m \leq 0) \leq P_{\text{train}}(m \leq \theta) + O\left(\frac{1}{\sqrt{N}} \sqrt{\frac{V \log^2(N/V)}{\theta^2} + \log(1/\delta)}\right).$$

Here  $m$  is the margin,  $\theta$  is any positive number,  $V$  is the Vapnik-Chervonenkis dimension of the classifier space,  $N$  is the size of the training set, and  $\delta$  is a small positive number. The inequality holds with probability  $1-\delta$  over many i.i.d. training and test sets. This inequality says: To obtain a low generalization error, minimize the number of observations below margin  $\theta$  in the training set.

`TotalBoost` minimizes a proxy of the Kullback-Leibler divergence between the current weight distribution and the initial weight distribution, subject to



the constraint that the *edge* (the weighted margin) is below a certain value. The proxy is a quadratic expansion of the divergence:

$$D(W, W_0) = \sum_{n=1}^N \log \frac{W(n)}{W_0(n)} \approx \sum_{n=1}^N \left( 1 + \frac{W(n)}{W_0(n)} \right) \Delta + \frac{1}{2W(n)} \Delta^2,$$

where  $\Delta$  is the difference between  $W(n)$ , the weights at the current and next iteration, and  $W_0$ , the initial weight distribution, which is uniform. This optimization formulation keeps weights from becoming zero. At each iteration, there are more constraints in the problem. So, for large problems, the optimization problem becomes increasingly constrained, and slow to solve.

TotalBoost typically creates ensembles with many learners having weights that are orders of magnitude smaller than those of other learners. Therefore, to better enable you to remove the unimportant ensemble members, the compact method reorders the members of a TotalBoost ensemble from largest weight to smallest. Therefore you can easily remove the least important members of the ensemble using the `removeLearners` method.

For examples using TotalBoost, see “LPBoost and TotalBoost for Small Ensembles” on page 16-98.

For references related to TotalBoost, see Warmuth, Liao, and Ratsch [18].

## Support Vector Machines (SVM)

In this section...
“Understanding Support Vector Machines” on page 16-162
“Using Support Vector Machines” on page 16-168
“Nonlinear Classifier with Gaussian Kernel” on page 16-170
“SVM Classification with Cross Validation” on page 16-174

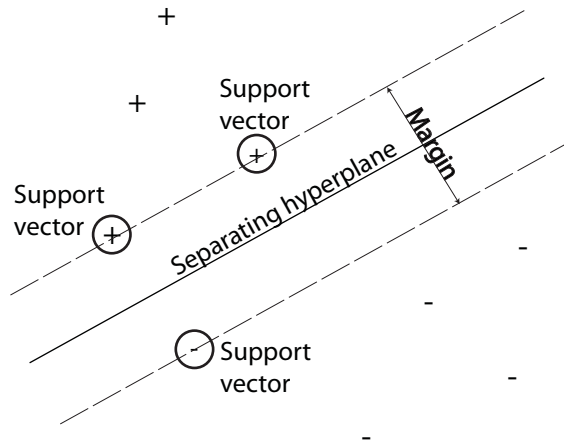
### Understanding Support Vector Machines

- “Separable Data” on page 16-162
- “Nonseparable Data” on page 16-165
- “Nonlinear Transformation with Kernels” on page 16-167

#### Separable Data

You can use a support vector machine (SVM) when your data has exactly two classes. An SVM classifies data by finding the best hyperplane that separates all data points of one class from those of the other class. The *best* hyperplane for an SVM means the one with the largest *margin* between the two classes. Margin means the maximal width of the slab parallel to the hyperplane that has no interior data points.

The *support vectors* are the data points that are closest to the separating hyperplane; these points are on the boundary of the slab. The following figure illustrates these definitions, with + indicating data points of type 1, and – indicating data points of type –1.



**Mathematical Formulation: Primal.** This discussion follows Hastie, Tibshirani, and Friedman [12] and Christianini and Shawe-Taylor [7].

The data for training is a set of points (vectors)  $x_i$  along with their categories  $y_i$ . For some dimension  $d$ , the  $x_i \in R^d$ , and the  $y_i = \pm 1$ . The equation of a hyperplane is

$$\langle w, x \rangle + b = 0,$$

where  $w \in R^d$ ,  $\langle w, x \rangle$  is the inner (dot) product of  $w$  and  $x$ , and  $b$  is real.

The following problem defines the *best* separating hyperplane. Find  $w$  and  $b$  that minimize  $\|w\|$  such that for all data points  $(x_i, y_i)$ ,

$$y_i(\langle w, x_i \rangle + b) \geq 1.$$

The support vectors are the  $x_i$  on the boundary, those for which  $y_i(\langle w, x_i \rangle + b) = 1$ .

For mathematical convenience, the problem is usually given as the equivalent problem of minimizing  $\langle w, w \rangle / 2$ . This is a quadratic programming problem. The optimal solution  $w, b$  enables classification of a vector  $z$  as follows:

$$\text{class}(z) = \text{sign}(\langle w, z \rangle + b).$$

**Mathematical Formulation: Dual.** It is computationally simpler to solve the dual quadratic programming problem. To obtain the dual, take positive Lagrange multipliers  $\alpha_i$  multiplied by each constraint, and subtract from the objective function:

$$L_P = \frac{1}{2} \langle w, w \rangle - \sum_i \alpha_i (y_i (\langle w, x_i \rangle + b) - 1),$$

where you look for a stationary point of  $L_P$  over  $w$  and  $b$ . Setting the gradient of  $L_P$  to 0, you get

$$\begin{aligned} w &= \sum_i \alpha_i y_i x_i \\ 0 &= \sum_i \alpha_i y_i. \end{aligned} \tag{16-1}$$

Substituting into  $L_P$ , you get the dual  $L_D$ :

$$L_D = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle,$$

which you maximize over  $\alpha_i \geq 0$ . In general, many  $\alpha_i$  are 0 at the maximum. The nonzero  $\alpha_i$  in the solution to the dual problem define the hyperplane, as seen in Equation 16-1, which gives  $w$  as the sum of  $\alpha_i y_i x_i$ . The data points  $x_i$  corresponding to nonzero  $\alpha_i$  are the *support vectors*.

The derivative of  $L_D$  with respect to a nonzero  $\alpha_i$  is 0 at an optimum. This gives

$$y_i (\langle w, x_i \rangle + b) - 1 = 0.$$

In particular, this gives the value of  $b$  at the solution, by taking any  $i$  with nonzero  $\alpha_i$ .

The dual is a standard quadratic programming problem. For example, the Optimization Toolbox quadprog solver solves this type of problem.

## Nonseparable Data

Your data might not allow for a separating hyperplane. In that case, SVM can use a *soft margin*, meaning a hyperplane that separates many, but not all data points.

There are two standard formulations of soft margins. Both involve adding slack variables  $s_i$  and a penalty parameter  $C$ .

- The  $L^1$ -norm problem is:

$$\min_{w,b,s} \left( \frac{1}{2} \langle w, w \rangle + C \sum_i s_i \right)$$

such that

$$\begin{aligned} y_i (\langle w, x_i \rangle + b) &\geq 1 - s_i \\ s_i &\geq 0. \end{aligned}$$

The  $L^1$ -norm refers to using  $s_i$  as slack variables instead of their squares. The SMO `svmtrain` method minimizes the  $L^1$ -norm problem.

- The  $L^2$ -norm problem is:

$$\min_{w,b,s} \left( \frac{1}{2} \langle w, w \rangle + C \sum_i s_i^2 \right)$$

subject to the same constraints. The QP `svmtrain` method minimizes the  $L^2$ -norm problem.

In these formulations, you can see that increasing  $C$  places more weight on the slack variables  $s_i$ , meaning the optimization attempts to make a stricter separation between classes. Equivalently, reducing  $C$  towards 0 makes misclassification less important.

**Mathematical Formulation: Dual.** For easier calculations, consider the  $L^1$  dual problem to this soft-margin formulation. Using Lagrange multipliers  $\mu_i$ , the function to minimize for the  $L^1$ -norm problem is:

$$L_P = \frac{1}{2} \langle w, w \rangle + C \sum_i s_i - \sum_i \alpha_i (y_i (\langle w, x_i \rangle + b) - (1 - s_i)) - \sum_i \mu_i s_i,$$

where you look for a stationary point of  $L_P$  over  $w$ ,  $b$ , and positive  $s_i$ . Setting the gradient of  $L_P$  to 0, you get

$$\begin{aligned} b &= \sum_i \alpha_i y_i x_i \\ \sum_i \alpha_i y_i &= 0 \\ \alpha_i &= C - \mu_i \\ \alpha_i, \mu_i, s_i &\geq 0. \end{aligned}$$

These equations lead directly to the dual formulation:

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle$$

subject to the constraints

$$\begin{aligned} \sum_i y_i \alpha_i &= 0 \\ 0 \leq \alpha_i &\leq C. \end{aligned}$$

The final set of inequalities,  $0 \leq \alpha_i \leq C$ , shows why  $C$  is sometimes called a *box constraint*.  $C$  keeps the allowable values of the Lagrange multipliers  $\alpha_i$  in a “box”, a bounded region.

The gradient equation for  $b$  gives the solution  $b$  in terms of the set of nonzero  $\alpha_i$ , which correspond to the support vectors.

You can write and solve the dual of the  $L^2$ -norm problem in an analogous manner. For details, see Christianini and Shawe-Taylor [7], Chapter 6.

**svmtrain Implementation.** Both dual soft-margin problems are quadratic programming problems. Internally, `svmtrain` has several different algorithms for solving the problems. The default Sequential Minimal Optimization (SMO) algorithm minimizes the one-norm problem. SMO is a relatively fast algorithm. If you have an Optimization Toolbox license, you can choose to use `quadprog` as the algorithm. `quadprog` minimizes the  $L^2$ -norm problem. `quadprog` uses a good deal of memory, but solves quadratic programs to a high degree of precision (see Bottou and Lin [2]). For details, see the `svmtrain` function reference page.

## Nonlinear Transformation with Kernels

Some binary classification problems do not have a simple hyperplane as a useful separating criterion. For those problems, there is a variant of the mathematical approach that retains nearly all the simplicity of an SVM separating hyperplane.

This approach uses these results from the theory of reproducing kernels:

- There is a class of functions  $K(x,y)$  with the following property. There is a linear space  $S$  and a function  $\varphi$  mapping  $x$  to  $S$  such that

$$K(x,y) = \langle \varphi(x), \varphi(y) \rangle.$$

The dot product takes place in the space  $S$ .

- This class of functions includes:
  - Polynomials: For some positive integer  $d$ ,

$$K(x,y) = (1 + \langle x,y \rangle)^d.$$

- Radial basis function: For some positive number  $\sigma$ ,

$$K(x,y) = \exp(-\langle (x-y), (x-y) \rangle / (2\sigma^2)).$$

- Multilayer perceptron (neural network): For a positive number  $p_1$  and a negative number  $p_2$ ,

$$K(x,y) = \tanh(p_1 \langle x,y \rangle + p_2).$$

---

**Note** Not every set of  $p_1$  and  $p_2$  gives a valid reproducing kernel.

---

The mathematical approach using kernels relies on the computational method of hyperplanes. All the calculations for hyperplane classification use nothing more than dot products. Therefore, nonlinear kernels can use identical calculations and solution algorithms, and obtain classifiers that are nonlinear. The resulting classifiers are hypersurfaces in some space  $S$ , but the space  $S$  does not have to be identified or examined.

## Using Support Vector Machines

As with any supervised learning model, you first train a support vector machine, then use the trained machine to classify (predict) new data. In addition, to obtain satisfactory predictive accuracy, you can use various SVM kernel functions, and you must tune the parameters of the kernel functions.

- “Training an SVM Classifier” on page 16-168
- “Classifying New Data with an SVM Classifier” on page 16-169
- “Tuning an SVM Classifier” on page 16-169

### Training an SVM Classifier

Train an SVM classifier with the `svmtrain` function. The most common syntax is:

```
SVMstruct = svmtrain(data,groups,'Kernel_Function','rbf');
```

The inputs are:

- `data` — Matrix of data points, where each row is one observation, and each column is one feature.
- `groups` — Column vector with each row corresponding to the value of the corresponding row in `data`. `groups` should have only two types of entries. So `groups` can have logical entries, or can be a double vector or cell array with two values.



- `Kernel_Function` — The default value of `'linear'` separates the data by a hyperplane. The value `'rbf'` uses a Gaussian radial basis function. Hsu, Chang, and Lin [14] suggest using `'rbf'` as your first try.

The resulting structure, `SVMstruct`, contains the optimized parameters from the SVM algorithm, enabling you to classify new data.

For more name-value pairs you can use to control the training, see the `svmtrain` reference page.

### Classifying New Data with an SVM Classifier

Classify new data with the `svmclassify` function. The syntax for classifying new data with a `SVMstruct` structure is:

```
newClasses = svmclassify(SVMstruct,newData)
```

The resulting vector, `newClasses`, represents the classification of each row in `newData`.

### Tuning an SVM Classifier

Hsu, Chang, and Lin [14] recommend tuning parameters of your classifier according to this scheme:

- Start with `Kernel_Function` set to `'rbf'` and default parameters.
- Try different parameters for training, and check via cross validation to obtain the best parameters.

The most important parameters to try changing are:

- `boxconstraint` — One strategy is to try a geometric sequence of the box constraint parameter. For example, take 11 values, from  $1e-5$  to  $1e5$  by a factor of 10.
- `rbf_sigma` — One strategy is to try a geometric sequence of the RBF sigma parameter. For example, take 11 values, from  $1e-5$  to  $1e5$  by a factor of 10.

For the various parameter settings, try cross validating the resulting classifier. Use `crossval` with 5-way or the default 10-way cross validation.

After obtaining a reasonable initial parameter, you might want to refine your parameters to obtain better accuracy. Start with your initial parameters and perform another cross validation step, this time using a factor of 1.2. Alternatively, optimize your parameters with `fminsearch`, as shown in “SVM Classification with Cross Validation” on page 16-174.

## Nonlinear Classifier with Gaussian Kernel

This example shows how to generate a nonlinear classifier with Gaussian kernel function. First, generate one class of points inside the unit disk in two dimensions, and another class of points in the annulus from radius 1 to radius 2. Then, generates a classifier based on the data with the Gaussian radial basis function kernel. The default linear classifier is obviously unsuitable for this problem, since the model is circularly symmetric. Set the box constraint parameter to `Inf` to make a strict classification, meaning no misclassified training points. Other kernel functions might not work with this strict box constraint, since they might be unable to provide a strict classification. Even though the `rbf` classifier can separate the classes, the result can be overtrained.

Generate 100 points uniformly distributed in the unit disk. To do so, generate a radius  $r$  as the square root of a uniform random variable, generate an angle  $t$  uniformly in  $(0, 2\pi)$ , and put the point at  $(r \cos(t), r \sin(t))$ .

```
r = sqrt(rand(100,1)); % radius
t = 2*pi*rand(100,1); % angle
data1 = [r.*cos(t), r.*sin(t)]; % points
```

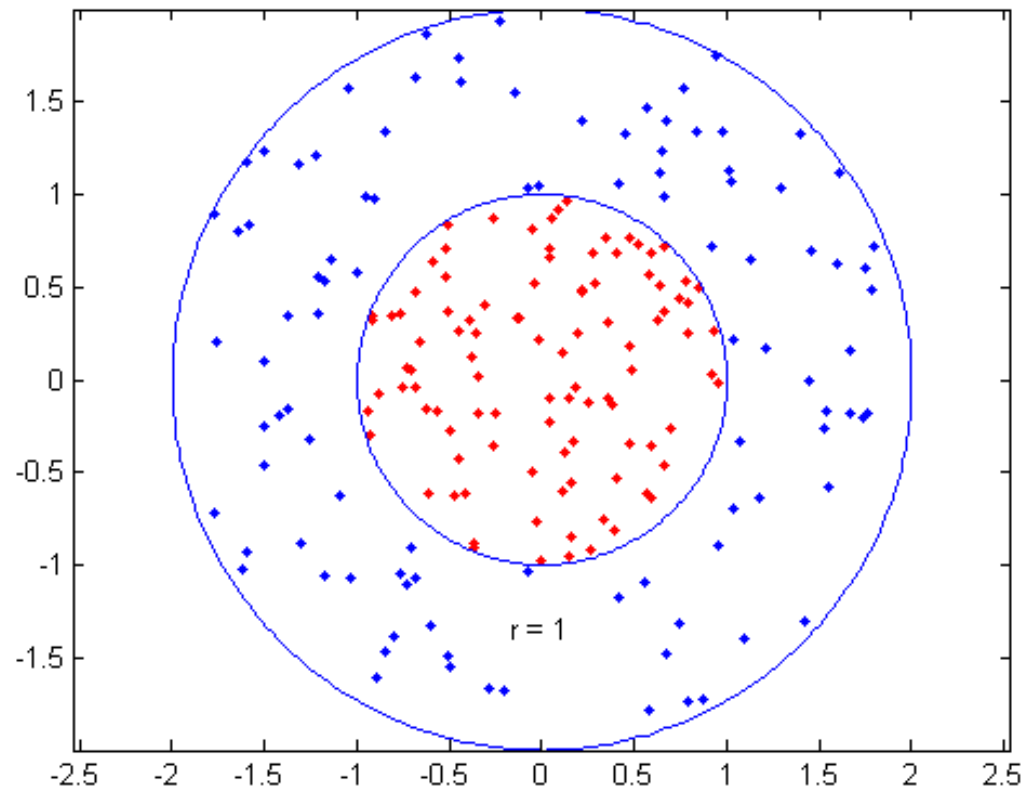
Generate 100 points uniformly distributed in the annulus. The radius is again proportional to a square root, this time a square root of the uniform distribution from 1 through 4.

```
r2 = sqrt(3*rand(100,1)+1); % radius
t2 = 2*pi*rand(100,1); % angle
data2 = [r2.*cos(t2), r2.*sin(t2)]; % points
```

Plot the points, and plot circles of radii 1 and 2 for comparison.

```
plot(data1(:,1),data1(:,2),'r.')
hold on
plot(data2(:,1),data2(:,2),'b.')
```

```
ezpolar(@(x)1);ezpolar(@(x)2);  
axis equal  
hold off
```

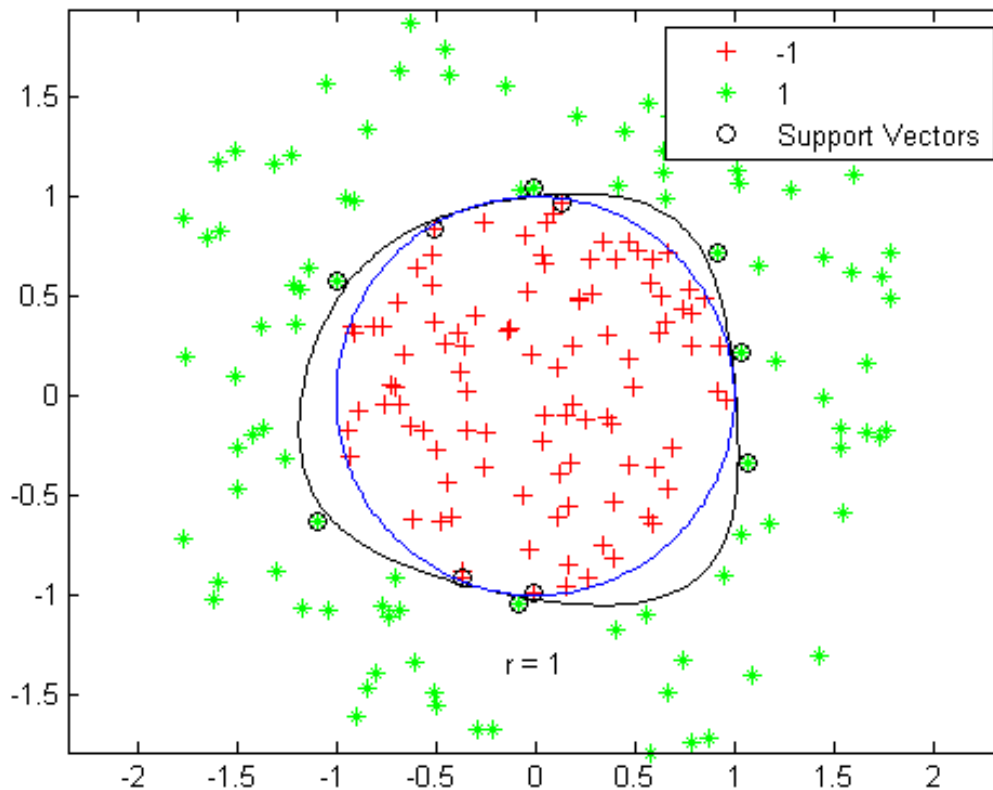


Put the data in one matrix, and make a vector of classifications.

```
data3 = [data1;data2];  
theclass = ones(200,1);  
theclass(1:100) = -1;
```

Train an SVM classifier with `Kernel_Function` set to `'rbf'` and `boxconstraint` set to `Inf`.

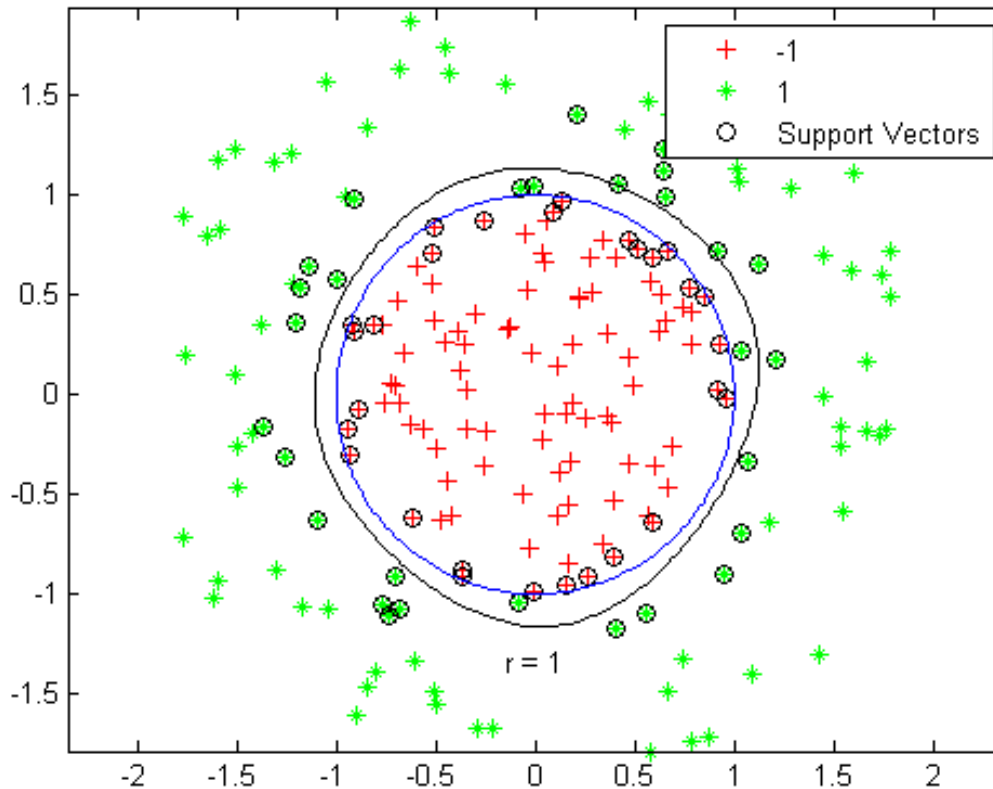
```
c1 = svmtrain(data3,theclass,'Kernel_Function','rbf',...  
    'boxconstraint',Inf,'showplot',true);  
hold on  
axis equal  
ezpolar(@(x)1)  
hold off
```



svmtrain generates a classifier that is close to a circle of radius 1. The difference is due to the random training data.

Training with the default parameters makes a more nearly circular classification boundary, but one that misclassifies some training data.

```
c1 = svmtrain(data3,theclass,'Kernel_Function','rbf',...  
             'showplot',true);  
hold on  
axis equal  
ezpolar(@(x)1)  
hold off
```



### SVM Classification with Cross Validation

This example classifies points from a Gaussian mixture model. The model is described in Hastie, Tibshirani, and Friedman [12], page 17. It begins with generating 10 base points for a “green” class, distributed as 2-D independent normals with mean (1,0) and unit variance. It also generates 10 base points for a “red” class, distributed as 2-D independent normals with mean (0,1) and unit variance. For each class (green and red), generate 100 random points as follows:

- 1 Choose a base point  $m$  of the appropriate color uniformly at random.
- 2 Generate an independent random point with 2-D normal distribution with mean  $m$  and variance  $I/5$ , where  $I$  is the 2-by-2 identity matrix.

After generating 100 green and 100 red points, classify them using `svmtrain`, and tune the classification using cross validation.

To generate the points and classifier:

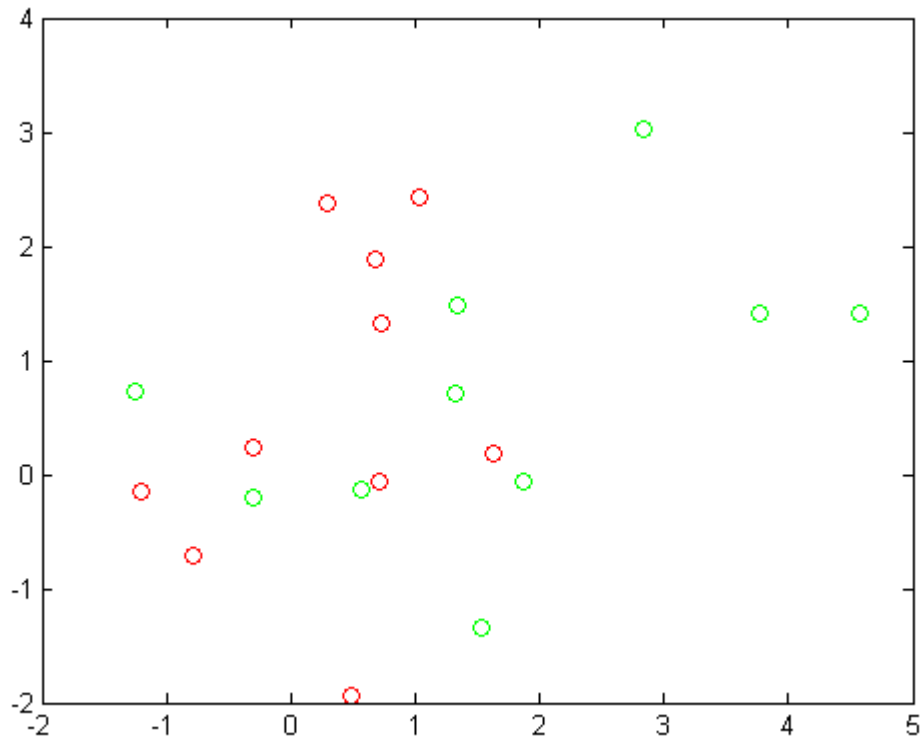
- 1 Generate the 10 base points for each class:

```
grnpop = mvnrnd([1,0],eye(2),10);  
redpop = mvnrnd([0,1],eye(2),10);
```

- 2 View the base points:

```
plot(grnpop(:,1),grnpop(:,2),'go')  
hold on  
plot(redpop(:,1),redpop(:,2),'ro')  
hold off
```

Since many red base points are close to green base points, it is difficult to classify the data points.



**3** Generate the 100 data points of each class:

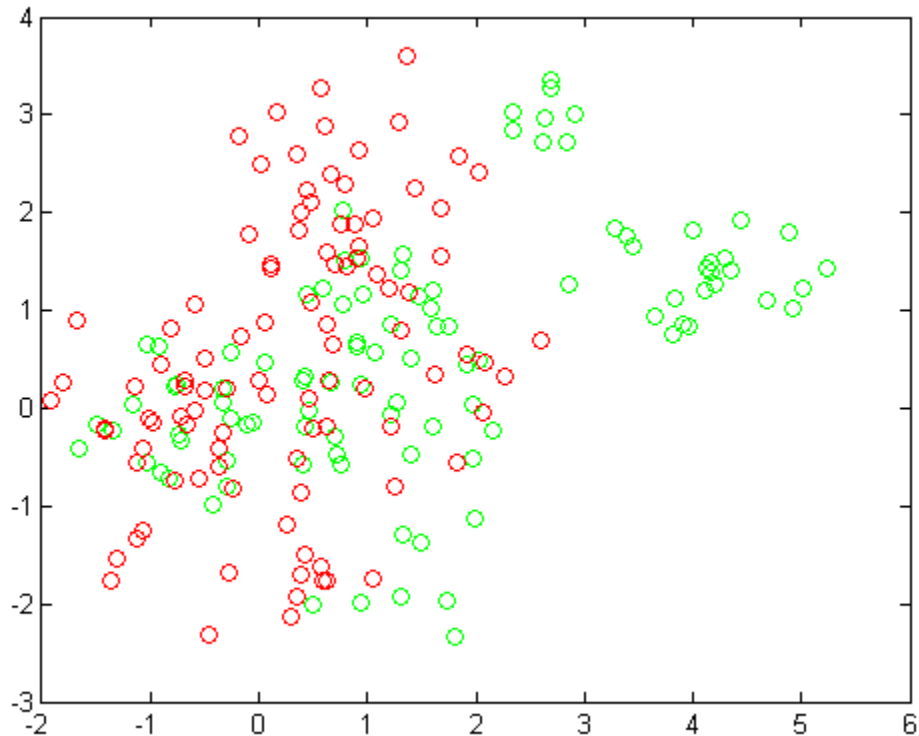
```
redpts = zeros(100,2);grnpts = redpts;  
for i = 1:100  
    grnpts(i,:) = mvnrnd(grnpop(randi(10),:),eye(2)*0.2);  
    redpts(i,:) = mvnrnd(redpop(randi(10),:),eye(2)*0.2);  
end
```

**4** View the data points:

```
figure  
plot(grnpts(:,1),grnpts(:,2),'go')  
hold on
```



```
plot(redpts(:,1),redpts(:,2),'ro')
hold off
```



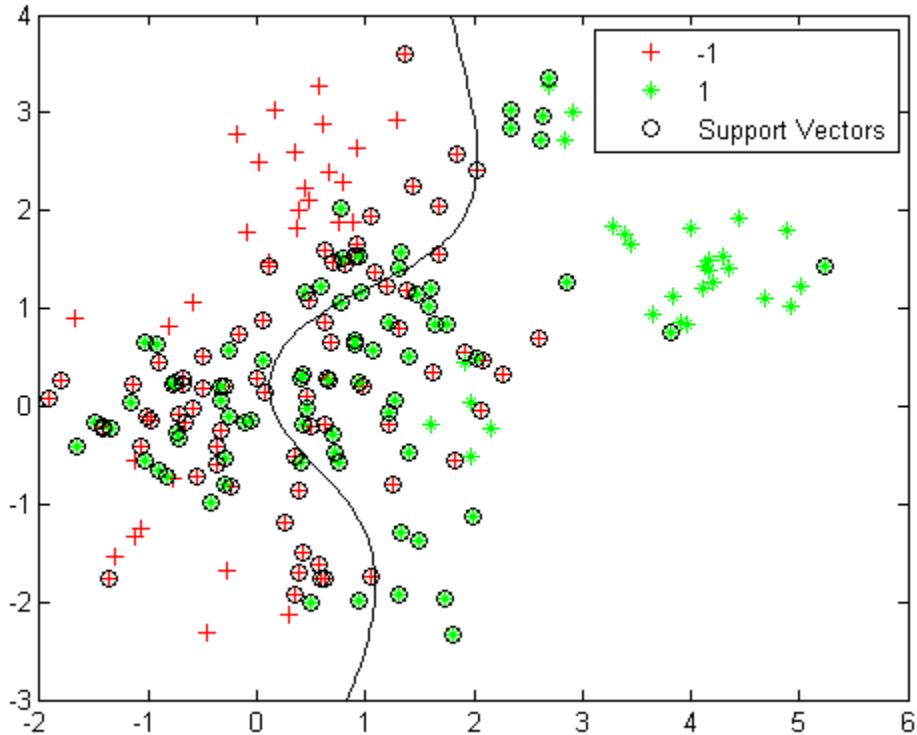
- 5** Put the data into one matrix, and make a vector `grp` that labels the class of each point:

```
cdata = [grnpts;redpts];
grp = ones(200,1);
% green label 1, red label -1
grp(101:200) = -1;
```

- 6** Check the basic classification of all the data using the default parameters:

```
svmStruct = svmtrain(cdata,grp,'Kernel_Function','rbf',...
```

```
'showplot',true);
```



- 7 Write a function called `crossfun` to calculate the predicted classification `yfit` from a test vector `xtest`, when the SVM is trained on a sample `xtrain` that has classification `ytrain`. Since you want to find the best parameters `rbf_sigma` and `boxconstraint`, include those in the function.

```
function yfit = ...
    crossfun(xtrain,ytrain,xtest,rbf_sigma,boxconstraint)

% Train the model on xtrain, ytrain,
% and get predictions of class of xtest
svmStruct = svmtrain(xtrain,ytrain,'Kernel_Function','rbf',...
```

```
'rbf_sigma',rbf_sigma,'boxconstraint',boxconstraint);
yfit = svmclassify(svmStruct,xtest);
```

- 8** Set up a partition for cross validation. This step causes the cross validation to be fixed. Without this step, the cross validation is random, so a minimization procedure can find a spurious local minimum.

```
c = cvpartition(200,'kfold',10);
```

- 9** Set up a function that takes an input  $z=[\text{rbf\_sigma},\text{boxconstraint}]$ , and returns the cross-validation value of  $\exp(z)$ . The reason to take  $\exp(z)$  is twofold:

- `rbf_sigma` and `boxconstraint` must be positive.
- You should look at points spaced approximately exponentially apart.

This function handle computes the cross validation at parameters  $\exp([\text{rbf\_sigma},\text{boxconstraint}])$ :

```
minfn = @(z)crossval('mcr',cdata,grp,'Predfun', ...
    @(xtrain,ytrain,xtest)crossfun(xtrain,ytrain,...
    xtest,exp(z(1)),exp(z(2))),'partition',c);
```

- 10** Search for the best parameters  $[\text{rbf\_sigma},\text{boxconstraint}]$  with `fminsearch`, setting looser tolerances than the defaults.

---

**Tip** If you have a Global Optimization Toolbox license, use `patternsearch` for faster, more reliable minimization. Give bounds on the components of  $z$  to keep the optimization in a sensible region, such as  $[-5,5]$ , and give a relatively loose `TolMesh` tolerance.

---

```
opts = optimset('TolX',5e-4,'TolFun',5e-4);
[searchmin fval] = fminsearch(minfn,randn(2,1),opts)
```

```
searchmin =
    0.9758
   -0.1569
```

```
fval =  
    0.3350
```

The best parameters [rbf\_sigma;boxconstraint] in this run are:

```
z = exp(searchmin)  
z =  
    2.6534  
    0.8548
```

- 11** Since the result of `fminsearch` can be a local minimum, not a global minimum, try again with a different starting point to check that your result is meaningful:

```
[searchmin fval] = fminsearch(minfn,randn(2,1),opts)
```

```
searchmin =  
    0.2778  
    0.6395
```

```
fval =  
    0.3100
```

The best parameters [rbf\_sigma;boxconstraint] in this run are:

```
z = exp(searchmin)  
z =  
    1.3202  
    1.8956
```

- 12** Try another search:

```
[searchmin fval] = fminsearch(minfn,randn(2,1),opts)
```

```
searchmin =  
   -0.0749  
    0.6085
```

```
fval =  
    0.2850
```

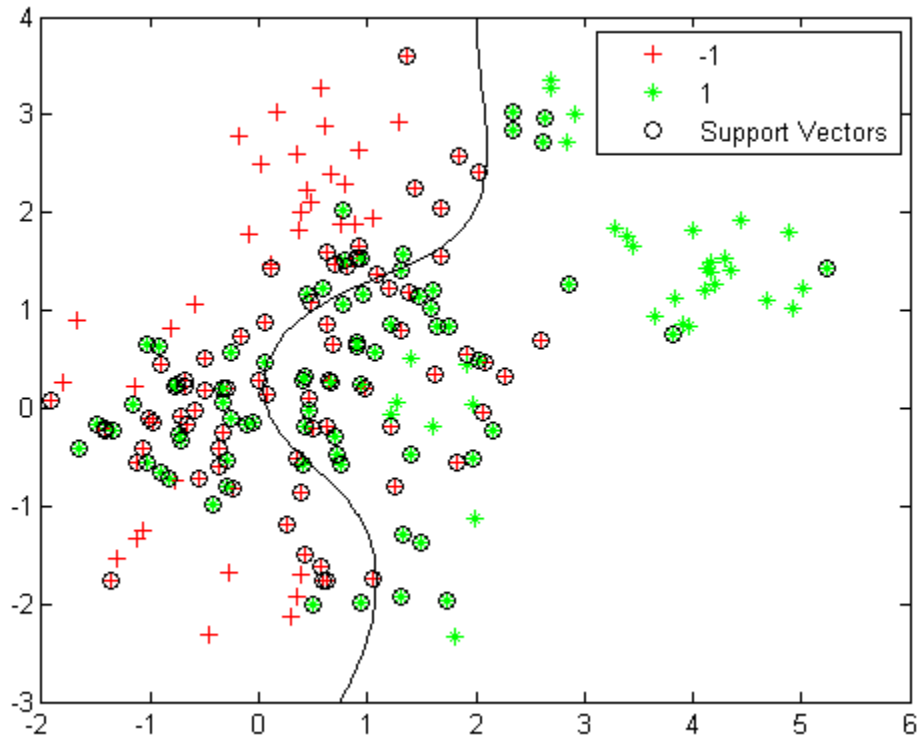
The third search obtains the lowest function value. The final parameters are:

```
z = exp(searchmin)
z =
    0.9278
    1.8376
```

The default parameters [1,1] are close to optimal for this data and partition.

**13** Use the z parameters to train a new SVM classifier:

```
svmStruct = svmtrain(cdata,grp,'Kernel_Function','rbf',...
    'rbf_sigma',z(1),'boxconstraint',z(2),'showplot',true);
```



**14** Generate and classify some new data points:

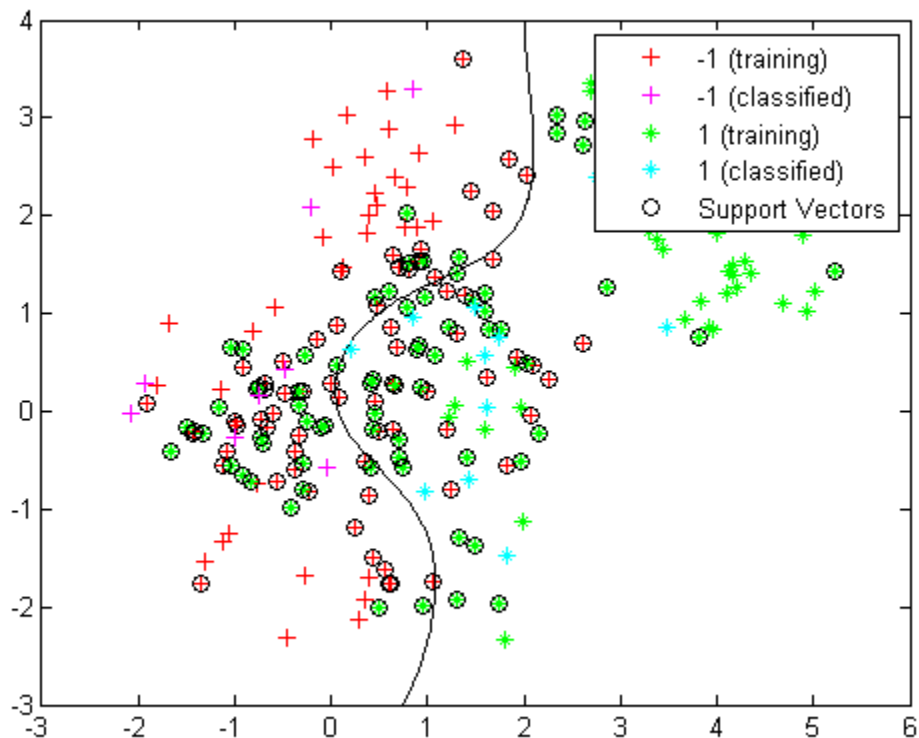
```

grnobj = gmdistribution(grnpop,.2*eye(2));
redobj = gmdistribution(redpop,.2*eye(2));

newData = random(grnobj,10);
newData = [newData;random(redobj,10)];
grpData = ones(20,1);
grpData(11:20) = -1; % red = -1

v = svmclassify(svmStruct,newData,'showplot',true);

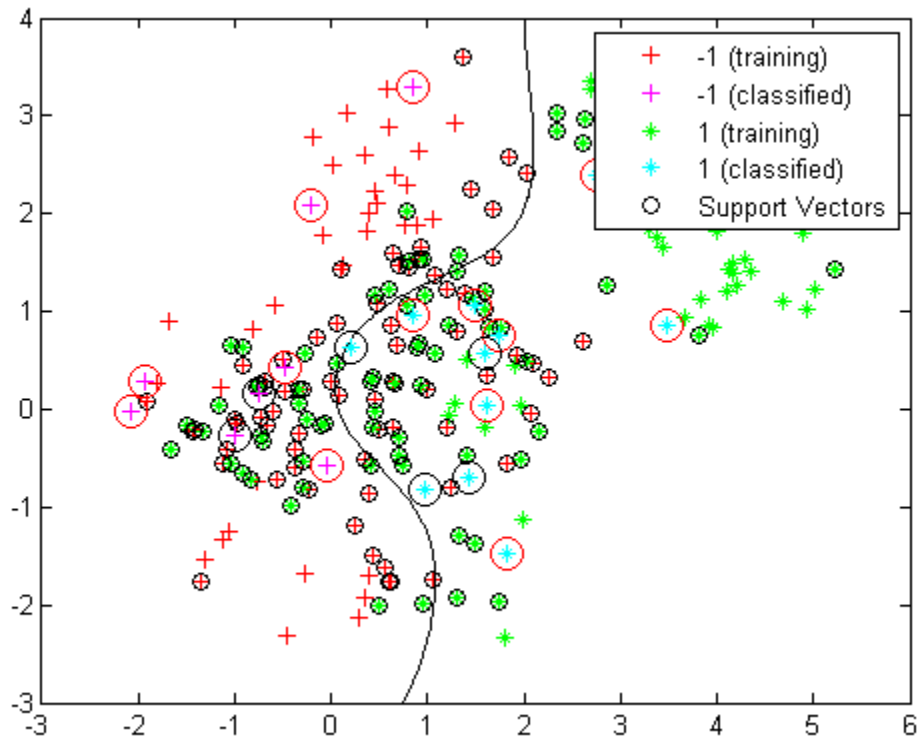
```



**15** See which new data points are correctly classified. Circle the correctly classified points in red, and the incorrectly classified points in black.

```
mydiff = (v == grpData); % classified correctly
hold on
for ii = mydiff % plot red circles around correct pts
    plot(newData(ii,1),newData(ii,2),'ro','MarkerSize',12)
end

for ii = not(mydiff) % plot black circles around incorrect pts
    plot(newData(ii,1),newData(ii,2),'ko','MarkerSize',12)
end
hold off
```



## Bibliography

- [1] Blackard, J. A. and D. J. Dean. *Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables*. Computers and Electronics in Agriculture 24, pp. 131–151, 1999.
- [2] Bottou, L., and Chih-Jen Lin. *Support Vector Machine Solvers*. Available at <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.4209&rep=rep1&type=pdf>.
- [3] Breiman, L. *Bagging Predictors*. Machine Learning 26, pp. 123–140, 1996.
- [4] Breiman, L. *Random Forests*. Machine Learning 45, pp. 5–32, 2001.
- [5] Breiman, L.  
<http://www.stat.berkeley.edu/~breiman/RandomForests/>
- [6] Breiman, L., et al. *Classification and Regression Trees*. Chapman & Hall, Boca Raton, 1993.
- [7] Christianini, N., and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press, Cambridge, UK, 2000.
- [8] Freund, Y. *A more robust boosting algorithm*. arXiv:0905.2138v1, 2009.
- [9] Freund, Y. and R. E. Schapire. *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting*. J. of Computer and System Sciences, Vol. 55, pp. 119–139, 1997.
- [10] Friedman, J. *Greedy function approximation: A gradient boosting machine*. Annals of Statistics, Vol. 29, No. 5, pp. 1189–1232, 2001.
- [11] Friedman, J., T. Hastie, and R. Tibshirani. *Additive logistic regression: A statistical view of boosting*. Annals of Statistics, Vol. 28, No. 2, pp. 337–407, 2000.
- [12] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, second edition. Springer, New York, 2008.



- [13] Ho, T. K. *The random subspace method for constructing decision forests*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 20, No. 8, pp. 832–844, 1998.
- [14] Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. *A Practical Guide to Support Vector Classification*. Available at <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf>.
- [15] Schapire, R. E. et al. *Boosting the margin: A new explanation for the effectiveness of voting methods*. Annals of Statistics, Vol. 26, No. 5, pp. 1651–1686, 1998.
- [16] Schapire, R., and Y. Singer. *Improved boosting algorithms using confidence-rated predictions*. Machine Learning, Vol. 37, No. 3, pp. 297–336, 1999.
- [17] Seiffert, C., T. Khoshgoftaar, J. Hulse, and A. Napolitano. *RUSBoost: Improving classification performance when training data is skewed*. 19th International Conference on Pattern Recognition, pp. 1–4, 2008.
- [18] Warmuth, M., J. Liao, and G. Ratsch. *Totally corrective boosting algorithms that maximize the margin*. Proc. 23rd Int'l. Conf. on Machine Learning, ACM, New York, pp. 1001–1008, 2006.
- [19] Zadrozny, B., J. Langford, and N. Abe. *Cost-Sensitive Learning by Cost-Proportionate Example Weighting*. CiteSeerX. [Online] 2003. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.9780>
- [20] Zhou, Z.-H. and X.-Y. Liu. *On Multi-Class Cost-Sensitive Learning*. CiteSeerX. [Online] 2006. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.9999>



# Markov Models

---

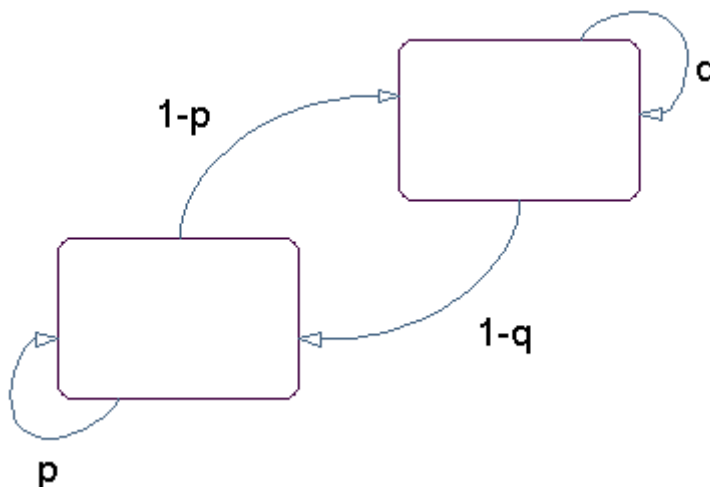
- “Introduction to Markov Models” on page 17-2
- “Markov Chains” on page 17-3
- “Hidden Markov Models (HMM)” on page 17-5

## Introduction to Markov Models

Markov processes are examples of stochastic processes—processes that generate random sequences of outcomes or *states* according to certain probabilities. Markov processes are distinguished by being memoryless—their next state depends only on their current state, not on the history that led them there. Models of Markov processes are used in a wide variety of applications, from daily stock prices to the positions of genes in a chromosome.

## Markov Chains

A Markov model is given visual representation with a *state diagram*, such as the one below.



### State Diagram for a Markov Model

The rectangles in the diagram represent the possible states of the process you are trying to model, and the arrows represent transitions between states. The label on each arrow represents the probability of that transition. At each step of the process, the model may generate an output, or *emission*, depending on which state it is in, and then make a transition to another state. An important characteristic of Markov models is that the next state depends only on the current state, and not on the history of transitions that lead to the current state.

For example, for a sequence of coin tosses the two states are heads and tails. The most recent coin toss determines the current state of the model and each subsequent toss determines the transition to the next state. If the coin is fair, the transition probabilities are all  $1/2$ . The emission might simply be the current state. In more complicated models, random processes at each state will generate emissions. You could, for example, roll a die to determine the emission at any step.

*Markov chains* are mathematical descriptions of Markov models with a discrete set of states. Markov chains are characterized by:

- A set of states  $\{1, 2, \dots, M\}$
- An  $M$ -by- $M$  *transition matrix*  $T$  whose  $i,j$  entry is the probability of a transition from state  $i$  to state  $j$ . The sum of the entries in each row of  $T$  must be 1, because this is the sum of the probabilities of making a transition from a given state to each of the other states.
- A set of possible outputs, or *emissions*,  $\{s_1, s_2, \dots, s_N\}$ . By default, the set of emissions is  $\{1, 2, \dots, N\}$ , where  $N$  is the number of possible emissions, but you can choose a different set of numbers or symbols.
- An  $M$ -by- $N$  *emission matrix*  $E$  whose  $i,k$  entry gives the probability of emitting symbol  $s_k$  given that the model is in state  $i$ .

Markov chains begin in an *initial state*  $i_0$  at step 0. The chain then transitions to state  $i_1$  with probability  $T_{1i_1}$ , and emits an output  $s_{k_1}$  with probability  $E_{i_1k_1}$ . Consequently, the probability of observing the sequence of states  $i_1i_2\dots i_r$  and the sequence of emissions  $s_{k_1}s_{k_2}\dots s_{k_r}$  in the first  $r$  steps, is

$$T_{1i_1} E_{i_1k_1} T_{i_1i_2} E_{i_2k_2} \dots T_{i_{r-1}i_r} E_{i_rk_r}$$

## Hidden Markov Models (HMM)

In this section...
“Introduction to Hidden Markov Models (HMM)” on page 17-5
“Analyzing Hidden Markov Models” on page 17-7

### Introduction to Hidden Markov Models (HMM)

A *hidden Markov model* (HMM) is one in which you observe a sequence of emissions, but do not know the sequence of states the model went through to generate the emissions. Analyses of hidden Markov models seek to recover the sequence of states from the observed data.

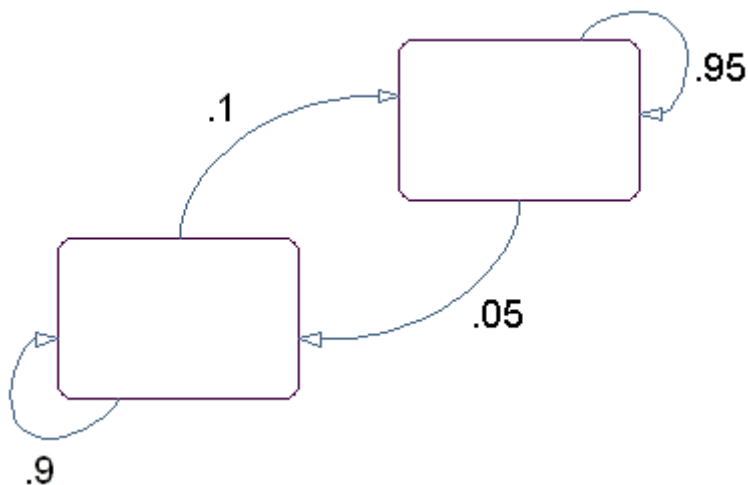
As an example, consider a Markov model with two states and six possible emissions. The model uses:

- A red die, having six sides, labeled 1 through 6.
- A green die, having twelve sides, five of which are labeled 2 through 6, while the remaining seven sides are labeled 1.
- A weighted red coin, for which the probability of heads is .9 and the probability of tails is .1.
- A weighted green coin, for which the probability of heads is .95 and the probability of tails is .05.

The model creates a sequence of numbers from the set {1, 2, 3, 4, 5, 6} with the following rules:

- Begin by rolling the red die and writing down the number that comes up, which is the emission.
- Toss the red coin and do one of the following:
  - If the result is heads, roll the red die and write down the result.
  - If the result is tails, roll the green die and write down the result.
- At each subsequent step, you flip the coin that has the same color as the die you rolled in the previous step. If the coin comes up heads, roll the same die as in the previous step. If the coin comes up tails, switch to the other die.

The state diagram for this model has two states, red and green, as shown in the following figure.



You determine the emission from a state by rolling the die with the same color as the state. You determine the transition to the next state by flipping the coin with the same color as the state.

The transition matrix is:

$$T = \begin{bmatrix} 0.9 & 0.1 \\ 0.05 & 0.95 \end{bmatrix}$$

The emissions matrix is:

$$E = \begin{bmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{7}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} & \frac{1}{12} \end{bmatrix}$$

The model is not hidden because you know the sequence of states from the colors of the coins and dice. Suppose, however, that someone else is generating



the emissions without showing you the dice or the coins. All you see is the sequence of emissions. If you start seeing more 1s than other numbers, you might suspect that the model is in the green state, but you cannot be sure because you cannot see the color of the die being rolled.

Hidden Markov models raise the following questions:

- Given a sequence of emissions, what is the most likely state path?
- Given a sequence of emissions, how can you estimate transition and emission probabilities of the model?
- What is the *forward probability* that the model generates a given sequence?
- What is the *posterior probability* that the model is in a particular state at any point in the sequence?

## Analyzing Hidden Markov Models

- “Generating a Test Sequence” on page 17-8
- “Estimating the State Sequence” on page 17-8
- “Estimating Transition and Emission Matrices” on page 17-9
- “Estimating Posterior State Probabilities” on page 17-11
- “Changing the Initial State Distribution” on page 17-12

Statistics Toolbox functions related to hidden Markov models are:

- `hmmgenerate` — Generates a sequence of states and emissions from a Markov model
- `hmmestimate` — Calculates maximum likelihood estimates of transition and emission probabilities from a sequence of emissions and a known sequence of states
- `hmmtrain` — Calculates maximum likelihood estimates of transition and emission probabilities from a sequence of emissions
- `hmmviterbi` — Calculates the most probable state path for a hidden Markov model

- `hmmdecode` — Calculates the posterior state probabilities of a sequence of emissions

This section shows how to use these functions to analyze hidden Markov models.

### Generating a Test Sequence

The following commands create the transition and emission matrices for the model described in the “Introduction to Hidden Markov Models (HMM)” on page 17-5:

```
TRANS = [.9 .1; .05 .95];  
  
EMIS = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;...  
7/12, 1/12, 1/12, 1/12, 1/12, 1/12];
```

To generate a random sequence of states and emissions from the model, use `hmmgenerate`:

```
[seq,states] = hmmgenerate(1000,TRANS,EMIS);
```

The output `seq` is the sequence of emissions and the output `states` is the sequence of states.

`hmmgenerate` begins in state 1 at step 0, makes the transition to state  $i_1$  at step 1, and returns  $i_1$  as the first entry in `states`. To change the initial state, see “Changing the Initial State Distribution” on page 17-12.

### Estimating the State Sequence

Given the transition and emission matrices `TRANS` and `EMIS`, the function `hmmviterbi` uses the Viterbi algorithm to compute the most likely sequence of states the model would go through to generate a given sequence `seq` of emissions:

```
likelystates = hmmviterbi(seq, TRANS, EMIS);
```

`likelystates` is a sequence the same length as `seq`.

To test the accuracy of `hmmviterbi`, compute the percentage of the actual sequence `states` that agrees with the sequence `likelystates`.

```
sum(states==likelystates)/1000
ans =
    0.8200
```

In this case, the most likely sequence of states agrees with the random sequence 82% of the time.

## Estimating Transition and Emission Matrices

- “Using `hmmestimate`” on page 17-9
- “Using `hmmtrain`” on page 17-10

The functions `hmmestimate` and `hmmtrain` estimate the transition and emission matrices `TRANS` and `EMIS` given a sequence `seq` of emissions.

**Using `hmmestimate`.** The function `hmmestimate` requires that you know the sequence of states `states` that the model went through to generate `seq`.

The following takes the emission and state sequences and returns estimates of the transition and emission matrices:

```
[TRANS_EST, EMIS_EST] = hmmestimate(seq, states)

TRANS_EST =
    0.8989    0.1011
    0.0585    0.9415

EMIS_EST =
    0.1721    0.1721    0.1749    0.1612    0.1803    0.1393
    0.5836    0.0741    0.0804    0.0789    0.0726    0.1104
```

You can compare the outputs with the original transition and emission matrices, `TRANS` and `EMIS`:

```
TRANS
TRANS =
    0.9000    0.1000
    0.0500    0.9500

EMIS
```

```
EMIS =
0.1667    0.1667    0.1667    0.1667    0.1667    0.1667
0.5833    0.0833    0.0833    0.0833    0.0833    0.0833
```

**Using hmmtrain.** If you do not know the sequence of states `states`, but you have initial guesses for `TRANS` and `EMIS`, you can still estimate `TRANS` and `EMIS` using `hmmtrain`.

Suppose you have the following initial guesses for `TRANS` and `EMIS`.

```
TRANS_GUESS = [.85 .15; .1 .9];
EMIS_GUESS = [.17 .16 .17 .16 .17 .17;.6 .08 .08 .08 .08 08];
```

You estimate `TRANS` and `EMIS` as follows:

```
[TRANS_EST2, EMIS_EST2] = hmmtrain(seq, TRANS_GUESS, EMIS_GUESS)
```

```
TRANS_EST2 =
0.2286    0.7714
0.0032    0.9968
```

```
EMIS_EST2 =
0.1436    0.2348    0.1837    0.1963    0.2350    0.0066
0.4355    0.1089    0.1144    0.1082    0.1109    0.1220
```

`hmmtrain` uses an iterative algorithm that alters the matrices `TRANS_GUESS` and `EMIS_GUESS` so that at each step the adjusted matrices are more likely to generate the observed sequence, `seq`. The algorithm halts when the matrices in two successive iterations are within a small tolerance of each other.

If the algorithm fails to reach this tolerance within a maximum number of iterations, whose default value is 100, the algorithm halts. In this case, `hmmtrain` returns the last values of `TRANS_EST` and `EMIS_EST` and issues a warning that the tolerance was not reached.

If the algorithm fails to reach the desired tolerance, increase the default value of the maximum number of iterations with the command:

```
hmmtrain(seq,TRANS_GUESS,EMIS_GUESS,'maxiterations',maxiter)
```

where `maxiter` is the maximum number of steps the algorithm executes.

Change the default value of the tolerance with the command:

```
hmmtrain(seq, TRANS_GUESS, EMIS_GUESS, 'tolerance', tol)
```

where `tol` is the desired value of the tolerance. Increasing the value of `tol` makes the algorithm halt sooner, but the results are less accurate.

Two factors reduce the reliability of the output matrices of `hmmtrain`:

- The algorithm converges to a local maximum that does not represent the true transition and emission matrices. If you suspect this, use different initial guesses for the matrices `TRANS_EST` and `EMIS_EST`.
- The sequence `seq` may be too short to properly train the matrices. If you suspect this, use a longer sequence for `seq`.

### Estimating Posterior State Probabilities

The posterior state probabilities of an emission sequence `seq` are the conditional probabilities that the model is in a particular state when it generates a symbol in `seq`, given that `seq` is emitted. You compute the posterior state probabilities with `hmmdecode`:

```
PSTATES = hmmdecode(seq, TRANS, EMIS)
```

The output `PSTATES` is an  $M$ -by- $L$  matrix, where  $M$  is the number of states and  $L$  is the length of `seq`. `PSTATES(i, j)` is the conditional probability that the model is in state  $i$  when it generates the  $j$ th symbol of `seq`, given that `seq` is emitted.

`hmmdecode` begins with the model in state 1 at step 0, prior to the first emission. `PSTATES(i, 1)` is the probability that the model is in state  $i$  at the following step 1. To change the initial state, see “Changing the Initial State Distribution” on page 17-12.

To return the logarithm of the probability of the sequence `seq`, use the second output argument of `hmmdecode`:

```
[PSTATES, logpseq] = hmmdecode(seq, TRANS, EMIS)
```

The probability of a sequence tends to 0 as the length of the sequence increases, and the probability of a sufficiently long sequence becomes less

than the smallest positive number your computer can represent. `hmmdecode` returns the logarithm of the probability to avoid this problem.

### Changing the Initial State Distribution

By default, Statistics Toolbox hidden Markov model functions begin in state 1. In other words, the distribution of initial states has all of its probability mass concentrated at state 1. To assign a different distribution of probabilities,  $p = [p_1, p_2, \dots, p_M]$ , to the  $M$  initial states, do the following:

- 1 Create an  $M+1$ -by- $M+1$  augmented transition matrix,  $\hat{T}$  of the following form:

$$\hat{T} = \begin{bmatrix} 0 & p \\ 0 & T \end{bmatrix}$$

where  $T$  is the true transition matrix. The first column of  $\hat{T}$  contains  $M+1$  zeros.  $p$  must sum to 1.

- 2 Create an  $M+1$ -by- $N$  augmented emission matrix,  $\hat{E}$ , that has the following form:

$$\hat{E} = \begin{bmatrix} 0 \\ E \end{bmatrix}$$

If the transition and emission matrices are `TRANS` and `EMIS`, respectively, you create the augmented matrices with the following commands:

```
TRANS_HAT = [0 p; zeros(size(TRANS,1),1) TRANS];
```

```
EMIS_HAT = [zeros(1,size(EMIS,2)); EMIS];
```

# Design of Experiments

---

- “Design of Experiments” on page 18-2
- “Full Factorial Designs” on page 18-3
- “Fractional Factorial Designs” on page 18-5
- “Response Surface Designs” on page 18-9
- “D-Optimal Designs” on page 18-15

## Design of Experiments

Passive data collection leads to a number of problems in statistical modeling. Observed changes in a response variable may be correlated with, but not caused by, observed changes in individual *factors* (process variables). Simultaneous changes in multiple factors may produce interactions that are difficult to separate into individual effects. Observations may be dependent, while a model of the data considers them to be independent.

Designed experiments address these problems. In a designed experiment, the data-producing process is actively manipulated to improve the quality of information and to eliminate redundant data. A common goal of all experimental designs is to collect data as parsimoniously as possible while providing sufficient information to accurately estimate model parameters.

For example, a simple model of a response  $y$  in an experiment with two controlled factors  $x_1$  and  $x_2$  might look like this:

$$y = \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_1x_2 + \varepsilon$$

Here  $\varepsilon$  includes both experimental error and the effects of any uncontrolled factors in the experiment. The terms  $\beta_1x_1$  and  $\beta_2x_2$  are *main effects* and the term  $\beta_3x_1x_2$  is a two-way *interaction effect*. A designed experiment would systematically manipulate  $x_1$  and  $x_2$  while measuring  $y$ , with the objective of accurately estimating  $\beta_0$ ,  $\beta_1$ ,  $\beta_2$ , and  $\beta_3$ .



## Full Factorial Designs

### In this section...

“Multilevel Designs” on page 18-3

“Two-Level Designs” on page 18-4

### Multilevel Designs

To systematically vary experimental factors, assign each factor a discrete set of *levels*. Full factorial designs measure response variables using every *treatment* (combination of the factor levels). A full factorial design for  $n$  factors with  $N_1, \dots, N_n$  levels requires  $N_1 \times \dots \times N_n$  experimental runs—one for each treatment. While advantageous for separating individual effects, full factorial designs can make large demands on data collection.

As an example, suppose a machine shop has three machines and four operators. If the same operator always uses the same machine, it is impossible to determine if a machine or an operator is the cause of variation in production. By allowing every operator to use every machine, effects are separated. A full factorial list of treatments is generated by the Statistics Toolbox function `fullfact`:

```
dFF = fullfact([3,4])
```

```
dFF =
```

```

1     1
2     1
3     1
1     2
2     2
3     2
1     3
2     3
3     3
1     4
2     4
3     4
```

Each of the  $3 \times 4 = 12$  rows of `dFF` represent one machine/operator combination.

## Two-Level Designs

Many experiments can be conducted with two-level factors, using *two-level designs*. For example, suppose the machine shop in the previous example always keeps the same operator on the same machine, but wants to measure production effects that depend on the composition of the day and night shifts. The Statistics Toolbox function `ff2n` generates a full factorial list of treatments:

```
dFF2 = ff2n(4)
dFF2 =
     0     0     0     0
     0     0     0     1
     0     0     1     0
     0     0     1     1
     0     1     0     0
     0     1     0     1
     0     1     1     0
     0     1     1     1
     1     0     0     0
     1     0     0     1
     1     0     1     0
     1     0     1     1
     1     1     0     0
     1     1     0     1
     1     1     1     0
     1     1     1     1
```

Each of the  $2^4 = 16$  rows of `dFF2` represent one schedule of operators for the day (0) and night (1) shifts.

## Fractional Factorial Designs

### In this section...

“Introduction to Fractional Factorial Designs” on page 18-5

“Plackett-Burman Designs” on page 18-5

“General Fractional Designs” on page 18-6

### Introduction to Fractional Factorial Designs

Two-level designs are sufficient for evaluating many production processes. Factor levels of  $\pm 1$  can indicate categorical factors, normalized factor extremes, or simply “up” and “down” from current factor settings. Experimenters evaluating process *changes* are interested primarily in the factor directions that lead to process improvement.

For experiments with many factors, two-level full factorial designs can lead to large amounts of data. For example, a two-level full factorial design with 10 factors requires  $2^{10} = 1024$  runs. Often, however, individual factors or their interactions have no distinguishable effects on a response. This is especially true of higher order interactions. As a result, a well-designed experiment can use fewer runs for estimating model parameters.

Fractional factorial designs use a fraction of the runs required by full factorial designs. A subset of experimental treatments is selected based on an evaluation (or assumption) of which factors and interactions have the most significant effects. Once this selection is made, the experimental design must separate these effects. In particular, significant effects should not be *confounded*, that is, the measurement of one should not depend on the measurement of another.

### Plackett-Burman Designs

*Plackett-Burman designs* are used when only main effects are considered significant. Two-level Plackett-Burman designs require a number of experimental runs that are a multiple of 4 rather than a power of 2. The MATLAB function `hadamard` generates these designs:

`dPB = hadamard(8)`

```

dPB =
  1  1  1  1  1  1  1  1
  1 -1  1 -1  1 -1  1 -1
  1  1 -1 -1  1  1 -1 -1
  1 -1 -1  1  1 -1 -1  1
  1  1  1  1 -1 -1 -1 -1
  1 -1  1 -1 -1  1 -1  1
  1  1 -1 -1 -1 -1  1  1
  1 -1 -1  1 -1  1  1 -1

```

Binary factor levels are indicated by  $\pm 1$ . The design is for eight runs (the rows of dPB) manipulating seven two-level factors (the last seven columns of dPB). The number of runs is a fraction  $8/2^7 = 0.0625$  of the runs required by a full factorial design. Economy is achieved at the expense of confounding main effects with any two-way interactions.

## General Fractional Designs

At the cost of a larger fractional design, you can specify which interactions you wish to consider significant. A design of *resolution R* is one in which no  $n$ -factor interaction is confounded with any other effect containing less than  $R - n$  factors. Thus, a resolution III design does not confound main effects with one another but may confound them with two-way interactions (as in “Plackett-Burman Designs” on page 18-5), while a resolution IV design does not confound either main effects or two-way interactions but may confound two-way interactions with each other.

Specify general fractional factorial designs using a full factorial design for a selected subset of *basic factors* and *generators* for the remaining factors. Generators are products of the basic factors, giving the levels for the remaining factors. Use the Statistics Toolbox function `fracfact` to generate these designs:

```

dfF = fracfact('a b c d bcd acd')
dfF =
  -1  -1  -1  -1  -1  -1
  -1  -1  -1  1  1  1
  -1  -1  1  -1  1  1
  -1  -1  1  1  -1  -1
  -1  1  -1  -1  1  -1

```

-1	1	-1	1	-1	1
-1	1	1	-1	-1	1
-1	1	1	1	1	-1
1	-1	-1	-1	-1	1
1	-1	-1	1	1	-1
1	-1	1	-1	1	-1
1	-1	1	1	-1	1
1	1	-1	-1	1	1
1	1	-1	1	-1	-1
1	1	1	-1	-1	-1
1	1	1	1	1	1

This is a six-factor design in which four two-level basic factors (a, b, c, and d in the first four columns of `dfF`) are measured in every combination of levels, while the two remaining factors (in the last three columns of `dfF`) are measured only at levels defined by the generators `bcd` and `acd`, respectively. Levels in the generated columns are products of corresponding levels in the columns that make up the generator.

The challenge of creating a fractional factorial design is to choose basic factors and generators so that the design achieves a specified resolution in a specified number of runs. Use the Statistics Toolbox function `fracfactgen` to find appropriate generators:

```
generators = fracfactgen('a b c d e f',4,4)
generators =
    'a'
    'b'
    'c'
    'd'
    'bcd'
    'acd'
```

These are generators for a six-factor design with factors a through f, using  $2^4 = 16$  runs to achieve resolution IV. The `fracfactgen` function uses an efficient search algorithm to find generators that meet the requirements.

An optional output from `fracfact` displays the *confounding pattern* of the design:

```
[dfF,confounding] = fracfact(generators);
```

```

confounding
confounding =
  'Term'      'Generator'  'Confounding'
  'X1'        'a'          'X1'
  'X2'        'b'          'X2'
  'X3'        'c'          'X3'
  'X4'        'd'          'X4'
  'X5'        'bcd'       'X5'
  'X6'        'acd'       'X6'
  'X1*X2'     'ab'        'X1*X2 + X5*X6'
  'X1*X3'     'ac'        'X1*X3 + X4*X6'
  'X1*X4'     'ad'        'X1*X4 + X3*X6'
  'X1*X5'     'abcd'      'X1*X5 + X2*X6'
  'X1*X6'     'cd'        'X1*X6 + X2*X5 + X3*X4'
  'X2*X3'     'bc'        'X2*X3 + X4*X5'
  'X2*X4'     'bd'        'X2*X4 + X3*X5'
  'X2*X5'     'cd'        'X1*X6 + X2*X5 + X3*X4'
  'X2*X6'     'abcd'      'X1*X5 + X2*X6'
  'X3*X4'     'cd'        'X1*X6 + X2*X5 + X3*X4'
  'X3*X5'     'bd'        'X2*X4 + X3*X5'
  'X3*X6'     'ad'        'X1*X4 + X3*X6'
  'X4*X5'     'bc'        'X2*X3 + X4*X5'
  'X4*X6'     'ac'        'X1*X3 + X4*X6'
  'X5*X6'     'ab'        'X1*X2 + X5*X6'
  
```

The confounding pattern shows that main effects are effectively separated by the design, but two-way interactions are confounded with various other two-way interactions.

## Response Surface Designs

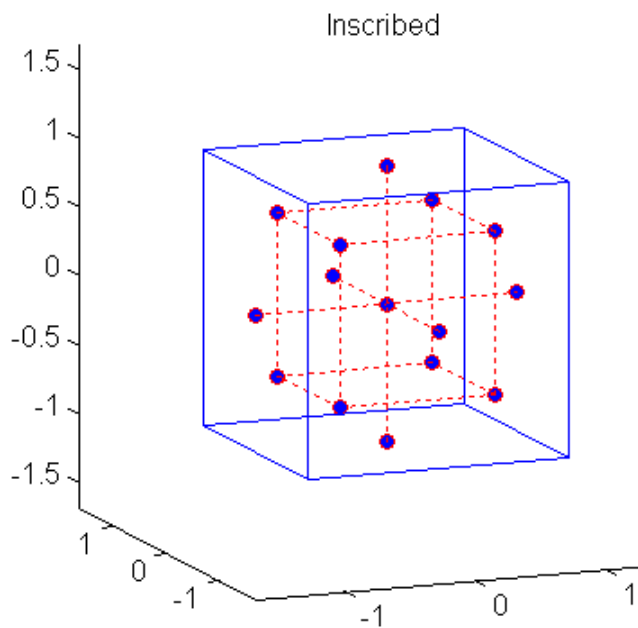
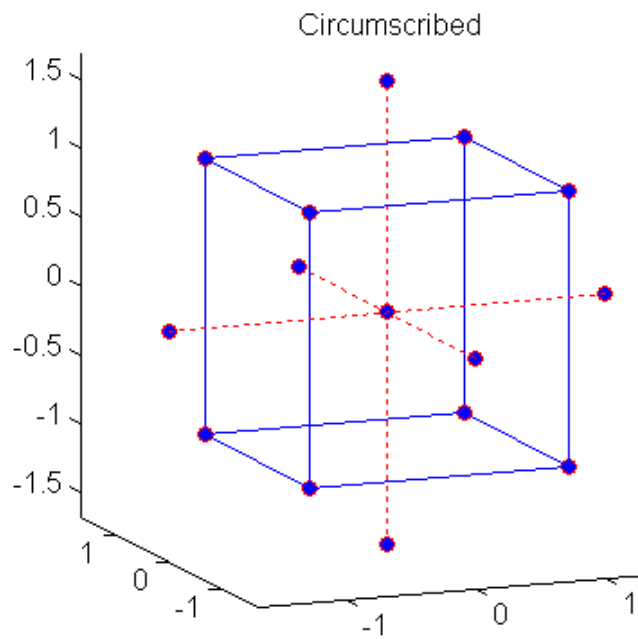
In this section...
“Introduction to Response Surface Designs” on page 18-9
“Central Composite Designs” on page 18-9
“Box-Behnken Designs” on page 18-13

### Introduction to Response Surface Designs

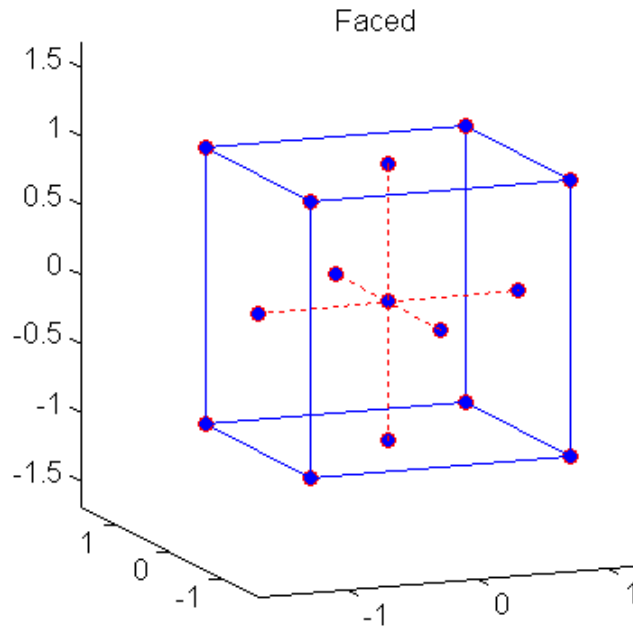
Quadratic response surfaces are simple models that provide a maximum or minimum without making additional assumptions about the form of the response. Quadratic models can be calibrated using full factorial designs with three or more levels for each factor, but these designs generally require more runs than necessary to accurately estimate model parameters. This section discusses designs for calibrating quadratic models that are much more efficient, using three or five levels for each factor, but not using all combinations of levels.

### Central Composite Designs

Central composite designs (CCDs), also known as Box-Wilson designs, are appropriate for calibrating full quadratic models. There are three types of CCDs—circumscribed, inscribed, and faced—pictured below:







Each design consists of a factorial design (the corners of a cube) together with *center* and *star* points that allow for estimation of second-order effects. For a full quadratic model with  $n$  factors, CCDs have enough design points to estimate the  $(n+2)(n+1)/2$  coefficients in a full quadratic model with  $n$  factors.

The type of CCD used (the position of the factorial and star points) is determined by the number of factors and by the desired properties of the design. The following table summarizes some important properties. A design is *rotatable* if the prediction variance depends only on the distance of the design point from the center of the design.

<b>Design</b>	<b>Rotatable</b>	<b>Factor Levels</b>	<b>Uses Points Outside <math>\pm 1</math></b>	<b>Accuracy of Estimates</b>
Circumscribed (CCC)	Yes	5	Yes	Good over entire design space
Inscribed (CCI)	Yes	5	No	Good over central subset of design space
Faced (CCF)	No	3	No	Fair over entire design space; poor for pure quadratic coefficients

Generate CCDs with the Statistics Toolbox function `ccdesign`:

```
dCC = ccdesign(3,'type','circumscribed')
```

```
dCC =
-1.0000 -1.0000 -1.0000
-1.0000 -1.0000  1.0000
-1.0000  1.0000 -1.0000
-1.0000  1.0000  1.0000
 1.0000 -1.0000 -1.0000
 1.0000 -1.0000  1.0000
 1.0000  1.0000 -1.0000
 1.0000  1.0000  1.0000
-1.6818         0         0
 1.6818         0         0
         0 -1.6818         0
         0  1.6818         0
         0         0 -1.6818
         0         0  1.6818
         0         0         0
         0         0         0
         0         0         0
         0         0         0
         0         0         0
         0         0         0
         0         0         0
         0         0         0
         0         0         0
         0         0         0
         0         0         0
```

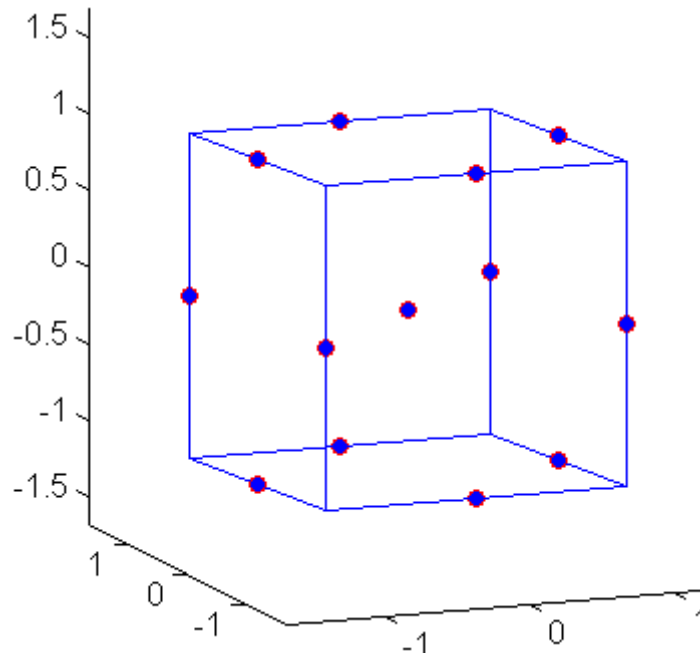
0            0            0

The repeated center point runs allow for a more uniform estimate of the prediction variance over the entire design space.

## Box-Behnken Designs

Like the designs described in “Central Composite Designs” on page 18-9, Box-Behnken designs are used to calibrate full quadratic models. Box-Behnken designs are rotatable and, for a small number of factors (four or less), require fewer runs than CCDs. By avoiding the corners of the design space, they allow experimenters to work around extreme factor combinations. Like an inscribed CCD, however, extremes are then poorly estimated.

The geometry of a Box-Behnken design is pictured in the following figure.



Design points are at the midpoints of edges of the design space and at the center, and do not contain an embedded factorial design.

Generate Box-Behnken designs with the Statistics Toolbox function `bbdesign`:

```
dBB = bbdesign(3)
```

```
dBB =  
    -1    -1     0  
    -1     1     0  
     1    -1     0  
     1     1     0  
    -1     0    -1  
    -1     0     1  
     1     0    -1  
     1     0     1  
     0    -1    -1  
     0    -1     1  
     0     1    -1  
     0     1     1  
     0     0     0  
     0     0     0  
     0     0     0
```

Again, the repeated center point runs allow for a more uniform estimate of the prediction variance over the entire design space.

## D-Optimal Designs

### In this section...

“Introduction to D-Optimal Designs” on page 18-15

“Generate D-Optimal Designs” on page 18-16

“Augment D-Optimal Designs” on page 18-19

“Specify Fixed Covariate Factors” on page 18-20

“Specify Categorical Factors” on page 18-21

“Specify Candidate Sets” on page 18-21

### Introduction to D-Optimal Designs

Traditional experimental designs (“Full Factorial Designs” on page 18-3, “Fractional Factorial Designs” on page 18-5, and “Response Surface Designs” on page 18-9) are appropriate for calibrating linear models in experimental settings where factors are relatively unconstrained in the region of interest. In some cases, however, models are necessarily nonlinear. In other cases, certain treatments (combinations of factor levels) may be expensive or infeasible to measure. *D-optimal designs* are model-specific designs that address these limitations of traditional designs.

A D-optimal design is generated by an iterative search algorithm and seeks to minimize the covariance of the parameter estimates for a specified model. This is equivalent to maximizing the determinant  $D = |X^T X|$ , where  $X$  is the design matrix of model terms (the columns) evaluated at specific treatments in the design space (the rows). Unlike traditional designs, D-optimal designs do not require orthogonal design matrices, and as a result, parameter estimates may be correlated. Parameter estimates may also be locally, but not globally, D-optimal.

There are several Statistics Toolbox functions for generating D-optimal designs:

Function	Description
candexch	Uses a row-exchange algorithm to generate a D-optimal design with a specified number of runs for a specified model and a specified candidate set. This is the second component of the algorithm used by rowexch.
candgen	Generates a candidate set for a specified model. This is the first component of the algorithm used by rowexch.
cordexch	Uses a coordinate-exchange algorithm to generate a D-optimal design with a specified number of runs for a specified model.
daugment	Uses a coordinate-exchange algorithm to augment an existing D-optimal design with additional runs to estimate additional model terms.
dcovary	Uses a coordinate-exchange algorithm to generate a D-optimal design with fixed covariate factors.
rowexch	Uses a row-exchange algorithm to generate a D-optimal design with a specified number of runs for a specified model. The algorithm calls candgen and then candexch. (Call candexch separately to specify a candidate set.)

The following sections explain how to use these functions to generate D-optimal designs.

---

**Note** The Statistics Toolbox function `rsmdemo` generates simulated data for experimental settings specified by either the user or by a D-optimal design generated by `cordexch`. It uses the `rstool` interface to visualize response surface models fit to the data, and it uses the `nlintool` interface to visualize a nonlinear model fit to the data.

---

## Generate D-Optimal Designs

Two Statistics Toolbox algorithms generate D-optimal designs:

- The `cordexch` function uses a coordinate-exchange algorithm
- The `rowexch` function uses a row-exchange algorithm

Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix  $X$  to increase  $D = |X^T X|$  at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally, D-optimal designs. Run each algorithm multiple times and select the best result for your final design. Both functions have a 'tries' parameter that automates this repetition and comparison.

At each step, the row-exchange algorithm exchanges an entire row of  $X$  with a row from a design matrix  $C$  evaluated at a *candidate set* of feasible treatments. The `rowexch` function automatically generates a  $C$  appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own  $C$  by calling `candexch` directly. In either case, if  $C$  is large, its static presence in memory can affect computation.

The coordinate-exchange algorithm, by contrast, does not use a candidate set. (Or rather, the candidate set is the entire design space.) At each step, the coordinate-exchange algorithm exchanges a single element of  $X$  with a new element evaluated at a neighboring point in design space. The absence of a candidate set reduces demands on memory, but the smaller scale of the search means that the coordinate-exchange algorithm is more likely to become trapped in a local minimum than the row-exchange algorithm.

For example, suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3 + \varepsilon$$

Use `cordexch` to generate a D-optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dCE,X] = cordexch(nfactors,nruns,'interaction','tries',10)
dCE =
    -1     1     1
    -1    -1    -1
     1     1     1
    -1     1    -1
     1    -1     1
```

$$X = \begin{matrix} & 1 & -1 & -1 \\ & -1 & -1 & 1 \\ & 1 & -1 & 1 & 1 & -1 & -1 & 1 \\ & 1 & -1 & -1 & -1 & 1 & 1 & 1 \\ & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ & 1 & -1 & 1 & -1 & -1 & 1 & -1 \\ & 1 & 1 & -1 & 1 & -1 & 1 & -1 \\ & 1 & 1 & -1 & -1 & -1 & -1 & 1 \\ & 1 & -1 & -1 & 1 & 1 & -1 & -1 \end{matrix}$$

Columns of the design matrix  $X$  are the model terms evaluated at each row of the design  $dCE$ . The terms appear in order from left to right:

- 1** Constant term
- 2** Linear terms (1, 2, 3)
- 3** Interaction terms (12, 13, 23)

Use  $X$  in a linear regression model fit to response data measured at the design points in  $dCE$ .

Use `rowexch` in a similar fashion to generate an equivalent design:

```
[dRE,X] = rowexch(nfactors,nruns,'interaction','tries',10)
dRE =
```

$$\begin{matrix} -1 & -1 & 1 \\ 1 & -1 & 1 \\ 1 & -1 & -1 \\ 1 & 1 & 1 \\ -1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & 1 & 1 \end{matrix}$$

$$X = \begin{matrix} 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 \end{matrix}$$



```
1 -1 1 1 -1 -1 1
```

## Augment D-Optimal Designs

In practice, you may want to add runs to a completed experiment to learn more about a process and estimate additional model coefficients. The `daugment` function uses a coordinate-exchange algorithm to augment an existing D-optimal design.

For example, the following eight-run design is adequate for estimating main effects in a four-factor model:

```
dCEmain = cordexch(4,8)
dCEmain =
    1 -1 -1 1
   -1 -1 1 1
   -1 1 -1 1
    1 1 1 -1
    1 1 1 1
   -1 1 -1 -1
    1 -1 -1 -1
   -1 -1 1 -1
```

To estimate the six interaction terms in the model, augment the design with eight additional runs:

```
dCEinteraction = daugment(dCEmain,8,'interaction')
dCEinteraction =
    1 -1 -1 1
   -1 -1 1 1
   -1 1 -1 1
    1 1 1 -1
    1 1 1 1
   -1 1 -1 -1
    1 -1 -1 -1
   -1 -1 1 -1
   -1 1 1 1
   -1 -1 -1 -1
    1 -1 1 -1
    1 1 -1 1
   -1 1 1 -1
```

```

1      1      -1     -1
1      -1     1      1
1      1      1      -1

```

The augmented design is full factorial, with the original eight runs in the first eight rows.

The 'start' parameter of the `candexch` function provides the same functionality as `daugment`, but uses a row exchange algorithm rather than a coordinate-exchange algorithm.

## Specify Fixed Covariate Factors

In many experimental settings, certain factors and their covariates are constrained to a fixed set of levels or combinations of levels. These cannot be varied when searching for an optimal design. The `dcovary` function allows you to specify fixed covariate factors in the coordinate exchange algorithm.

For example, suppose you want a design to estimate the parameters in a three-factor linear additive model, with eight runs that necessarily occur at different times. If the process experiences temporal linear drift, you may want to include the run time as a variable in the model. Produce the design as follows:

```

time = linspace(-1,1,8)';
[dCV,X] = dcovary(3,time,'linear')
dCV =
-1.0000    1.0000    1.0000   -1.0000
 1.0000   -1.0000   -1.0000   -0.7143
-1.0000   -1.0000   -1.0000   -0.4286
 1.0000   -1.0000    1.0000   -0.1429
 1.0000    1.0000   -1.0000    0.1429
-1.0000    1.0000   -1.0000    0.4286
 1.0000    1.0000    1.0000    0.7143
-1.0000   -1.0000    1.0000    1.0000
X =
 1.0000   -1.0000    1.0000    1.0000   -1.0000
 1.0000    1.0000   -1.0000   -1.0000   -0.7143
 1.0000   -1.0000   -1.0000   -1.0000   -0.4286
 1.0000    1.0000   -1.0000    1.0000   -0.1429

```

1.0000	1.0000	1.0000	-1.0000	0.1429
1.0000	-1.0000	1.0000	-1.0000	0.4286
1.0000	1.0000	1.0000	1.0000	0.7143
1.0000	-1.0000	-1.0000	1.0000	1.0000

The column vector `time` is a fixed factor, normalized to values between  $\pm 1$ . The number of rows in the fixed factor specifies the number of runs in the design. The resulting design `dCV` gives factor settings for the three controlled model factors at each time.

## Specify Categorical Factors

Categorical factors take values in a discrete set of levels. Both `cordexch` and `rowexch` have a `'categorical'` parameter that allows you to specify the indices of categorical factors and a `'levels'` parameter that allows you to specify a number of levels for each factor.

For example, the following eight-run design is for a linear additive model with five factors in which the final factor is categorical with three levels:

```
dCEcat = cordexch(5,8,'linear','categorical',5,'levels',3)
dCEcat =
    -1    -1     1     1     2
    -1    -1    -1    -1     3
     1     1     1     1     3
     1     1    -1    -1     2
     1    -1    -1     1     3
    -1     1    -1     1     1
    -1     1     1    -1     3
     1    -1     1    -1     1
```

## Specify Candidate Sets

The row-exchange algorithm exchanges rows of an initial design matrix  $X$  with rows from a design matrix  $C$  evaluated at a candidate set of feasible treatments. The `rowexch` function automatically generates a  $C$  appropriate for a specified model, operating in two steps by calling the `candgen` and `candexch` functions in sequence. Provide your own  $C$  by calling `candexch` directly.

For example, the following uses `rowexch` to generate a five-run design for a two-factor pure quadratic model using a candidate set that is produced internally:

```
dRE1 = rowexch(2,5,'purequadratic','tries',10)
dRE1 =
    -1     1
     0     0
     1    -1
     1     0
     1     1
```

The same thing can be done using `candgen` and `candexch` in sequence:

```
[dC,C] = candgen(2,'purequadratic') % Candidate set, C
dC =
    -1    -1
     0    -1
     1    -1
    -1     0
     0     0
     1     0
    -1     1
     0     1
     1     1
C =
     1    -1    -1     1     1
     1     0    -1     0     1
     1     1    -1     1     1
     1    -1     0     1     0
     1     0     0     0     0
     1     1     0     1     0
     1    -1     1     1     1
     1     0     1     0     1
     1     1     1     1     1
treatments = candexch(C,5,'tries',10) % D-opt subset
treatments =
     2
     1
     7
     3
```

```

      4
dRE2 = dC(treatments,:) % Display design
dRE2 =
      0     -1
     -1     -1
     -1      1
      1     -1
     -1      0

```

You can replace `C` in this example with a design matrix evaluated at your own candidate set. For example, suppose your experiment is constrained so that the two factors cannot have extreme settings simultaneously. The following produces a restricted candidate set:

```

constraint = sum(abs(dC),2) < 2; % Feasible treatments
my_dC = dC(constraint,:)
my_dC =
      0     -1
     -1      0
      0      0
      1      0
      0      1

```

Use the `x2fx` function to convert the candidate set to a design matrix:

```

my_C = x2fx(my_dC, 'purequadratic')
my_C =
      1      0     -1      0      1
      1     -1      0      1      0
      1      0      0      0      0
      1      1      0      1      0
      1      0      1      0      1

```

Find the required design in the same manner:

```

my_treatments = candexch(my_C,5,'tries',10) % D-opt subset
my_treatments =
      2
      4
      5
      1

```

```
      3
my_dRE = my_dC(my_treatments,:) % Display design
my_dRE =
    -1     0
     1     0
     0     1
     0    -1
     0     0
```

# Statistical Process Control

---

- “Introduction to Statistical Process Control” on page 19-2
- “Control Charts” on page 19-3
- “Capability Studies” on page 19-7

## **Introduction to Statistical Process Control**

Statistical process control (SPC) refers to a number of different methods for monitoring and assessing the quality of manufactured goods. Combined with methods from the design of experiments, SPC is used in programs that define, measure, analyze, improve, and control development and production processes. These programs are often implemented using “Design for Six Sigma” methodologies.



## Control Charts

A control chart displays measurements of process samples over time. The measurements are plotted together with user-defined *specification limits* and process-defined *control limits*. The process can then be compared with its specifications—to see if it is *in control* or *out of control*.

The chart is just a monitoring tool. Control activity might occur if the chart indicates an undesirable, systematic change in the process. The control chart is used to discover the variation, so that the process can be adjusted to reduce it.

Control charts are created with the `controlchart` function. Any of the following chart types may be specified:

- Xbar or mean
- Standard deviation
- Range
- Exponentially weighted moving average
- Individual observation
- Moving range of individual observations
- Moving average of individual observations
- Proportion defective
- Number of defectives
- Defects per unit
- Count of defects

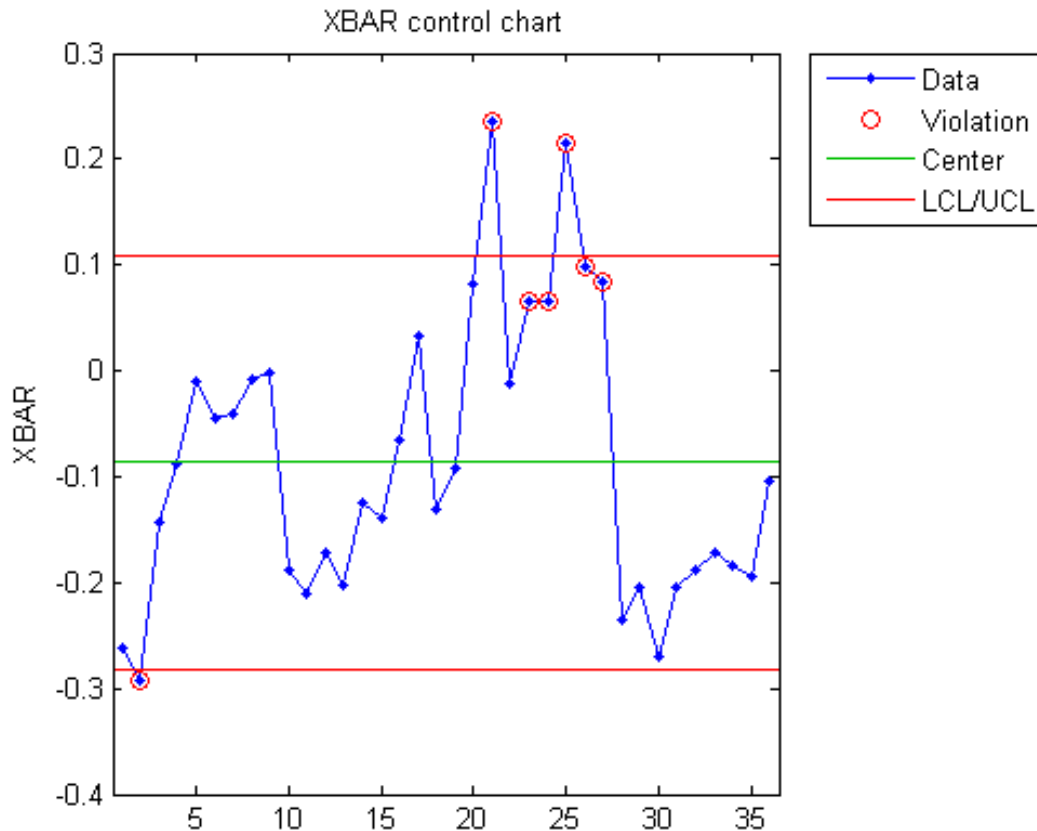
Control rules are specified with the `controlrules` function. The following example illustrates how to use Western Electric rules to mark out of control measurements on an Xbar chart.

First load the sample data.

```
load parts;
```

Construct the Xbar control chart using the Western Electric 2 rule (2 of 3 points at least 2 standard errors above the center line) to mark the out of control measurements.

```
st = controlchart(runout,'rules','we2');
```



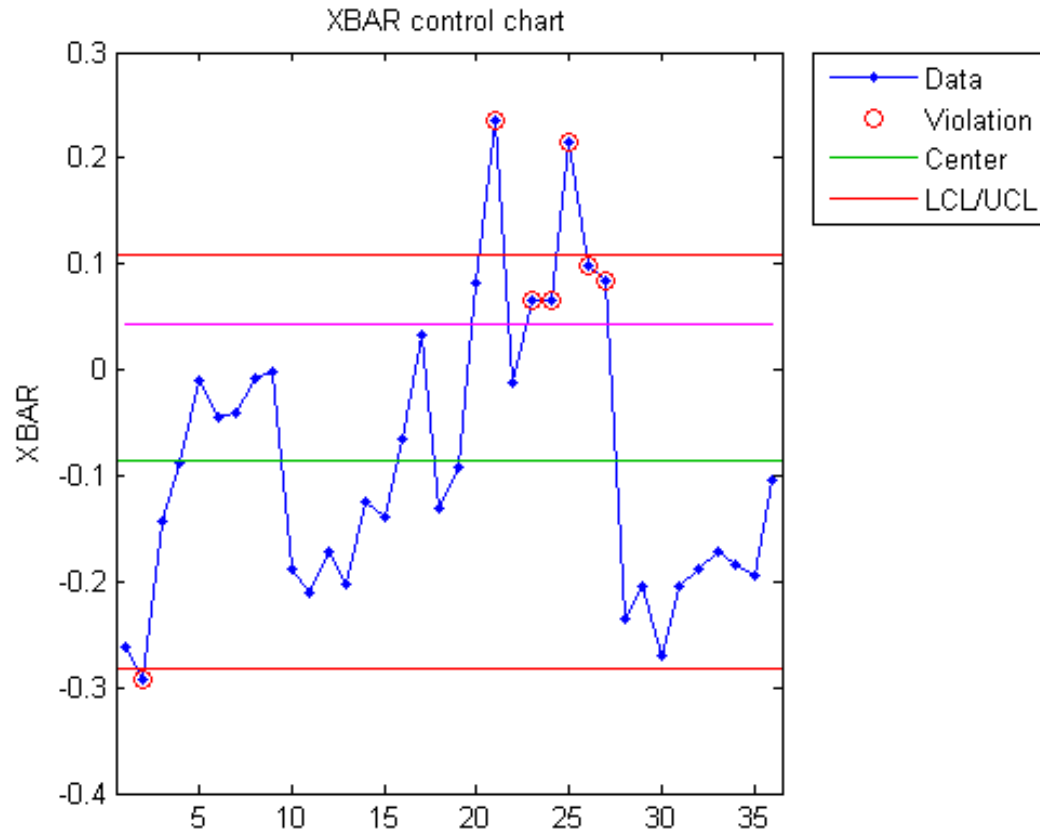
For a better understanding of the Western Electric 2 rule, calculate and plot the 2 standard errors line on the chart.

```
x = st.mean;  
cl = st.mu;
```

```

se = st.sigma./sqrt(st.n);
hold on
plot(c1+2*se, 'm')

```



Identify the measurements that violate the control rule.

```

R = controlrules('we2',x,c1,se);
I = find(R)

```

I =

21  
23  
24  
25  
26  
27

## Capability Studies

Before going into production, many manufacturers run a *capability study* to determine if their process will run within specifications enough of the time. *Capability indices* produced by such a study are used to estimate expected percentages of defective parts.

Capability studies are conducted with the `capability` function. The following capability indices are produced:

- `mu` — Sample mean
- `sigma` — Sample standard deviation
- `P` — Estimated probability of being within the lower (L) and upper (U) specification limits
- `P1` — Estimated probability of being below L
- `Pu` — Estimated probability of being above U
- `Cp` —  $(U-L)/(6*\text{sigma})$
- `Cp1` —  $(\text{mu}-L)/(3.*\text{sigma})$
- `Cpu` —  $(U-\text{mu})/(3.*\text{sigma})$
- `Cpk` —  $\min(\text{Cp1}, \text{Cpu})$

As an example, simulate a sample from a process with a mean of 3 and a standard deviation of 0.005:

```
data = normrnd(3,0.005,100,1);
```

Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99:

```
S = capability(data,[2.99 3.01])
```

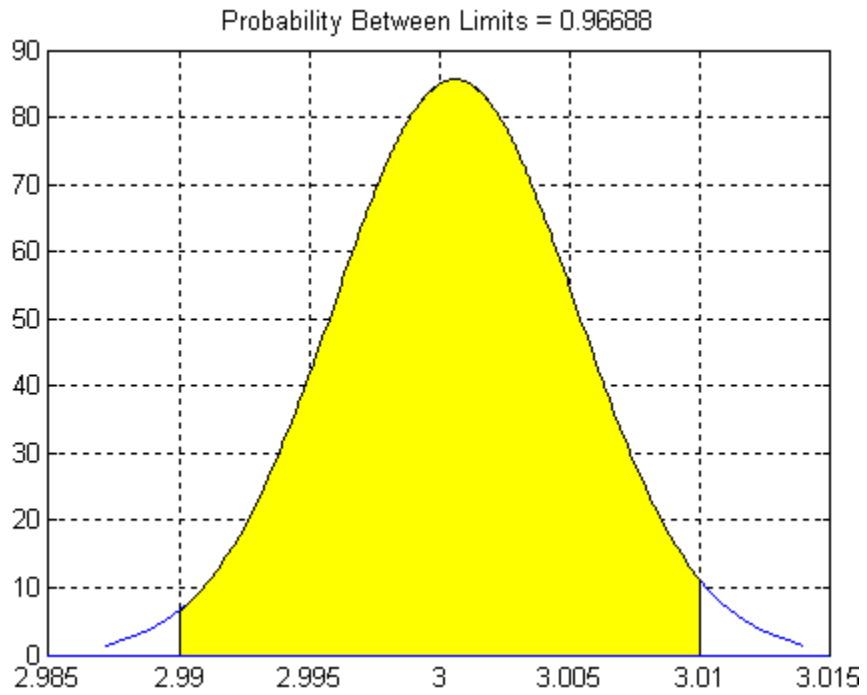
```
S =
```

```
    mu: 3.0006
   sigma: 0.0047
      P: 0.9669
     P1: 0.0116
     Pu: 0.0215
      Cp: 0.7156
```

Cp1: 0.7567  
Cpu: 0.6744  
Cpk: 0.6744

Visualize the specification and process widths:

```
capaplot(data,[2.99 3.01]);  
grid on
```



# Parallel Statistics

---

- “Quick Start Parallel Computing for Statistics Toolbox” on page 20-2
- “Concepts of Parallel Computing in Statistics Toolbox” on page 20-6
- “When to Run Statistical Functions in Parallel” on page 20-7
- “Working with parfor” on page 20-9
- “Reproducibility in Parallel Statistical Computations” on page 20-12
- “Examples of Parallel Statistical Functions” on page 20-18

## Quick Start Parallel Computing for Statistics Toolbox

---

**Note** To use parallel computing as described in this chapter, you must have a Parallel Computing Toolbox license.

---

In this section...
“What Is Parallel Statistics Functionality?” on page 20-2
“How To Compute in Parallel” on page 20-3
“Parallel Treebagger” on page 20-4

### What Is Parallel Statistics Functionality?

You can use any of the Statistics Toolbox functions with Parallel Computing Toolbox constructs such as `parfor` and `spmd`. However, some functions, such as those with interactive displays, can lose functionality in parallel. In particular, displays and interactive usage are not effective on workers (see “Vocabulary for Parallel Computation” on page 20-6).

Additionally, the following functions are enhanced to use parallel computing internally. These functions use `parfor` internally to parallelize calculations.

- `bootci`
- `bootstrp`
- `candexch`
- `cordexch`
- `crossval`
- `daugment`
- `dcovary`
- `jackknife`
- `kmeans`
- `lasso`



- `lassoglm`
- `nmmf`
- `plsregress`
- `rowexch`
- `sequentialfs`
- `TreeBagger`
- `TreeBagger.growTrees`

This chapter gives the simplest way to use these enhanced functions in parallel. For more advanced topics, including the issues of reproducibility and nested `parfor` loops, see the other sections in this chapter.

For information on parallel statistical computing at the command line, enter

```
help parallelstats
```

## How To Compute in Parallel

To have a function compute in parallel:

- 1 “Set Up a Parallel Environment” on page 20-3
- 2 “Set the UseParallel Option to true” on page 20-4
- 3 “Call the Function Using the Options Structure” on page 20-4

### Set Up a Parallel Environment

To run a statistical computation in parallel, first set up a parallel environment.

---

**Note** Setting up a parallel environment can take several seconds.

---

For a multicore machine, enter the following at the MATLAB command line:

```
parpool(n)
```

*n* is the number of workers you want to use.

### Set the UseParallel Option to true

Create an options structure with the `statset` function. To run in parallel, set the `UseParallel` option to true:

```
paroptions = statset('UseParallel',true);
```

### Call the Function Using the Options Structure

Call your function with syntax that uses the options structure. For example:

```
% Run crossval in parallel
cvMse = crossval('mse',x,y,'predfun',regf,'Options',paroptions);

% Run bootstrp in parallel
sts = bootstrp(100,@(x)[mean(x) std(x)],y,'Options',paroptions);

% Run TreeBagger in parallel
b = TreeBagger(50,meas,spec,'OOBPred','on','Options',paroptions);
```

For more complete examples of parallel statistical functions, see “Parallel Treebagger” on page 20-4 and “Examples of Parallel Statistical Functions” on page 20-18.

After you have finished computing in parallel, close the parallel environment:

```
delete mypool
```

---

**Tip** To save time, keep the pool open if you expect to compute in parallel again soon.

---

## Parallel Treebagger

To run the example “Workflow Example: Regression of Insurance Risk Rating for Car Imports with TreeBagger” on page 16-125 in parallel:

- 1 Set up the parallel environment to use two cores:

```
mypool = parpool(2)
Starting parpool using the 'local' profile ... connected to 2 workers.
```

```
mypool =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}
NumWorkers: 2
Cluster: [1x1 parallel.cluster.Local]
SpmEnabled: 1
```

- 2** Set the options to use parallel processing:

```
paroptions = statset('UseParallel',true);
```

- 3** Load the problem data and separate it into input and response:

```
load imports-85;
Y = X(:,1);
X = X(:,2:end);
```

- 4** Estimate feature importance using leaf size 1 and 1000 trees in parallel. Time the function for comparison purposes:

```
tic
b = TreeBagger(1000,X,Y,'Method','r','OOBVarImp','on',...
'cat',16:25,'MinLeaf',1,'Options',paroptions);
toc
```

Elapsed time is 16.696336 seconds.

- 5** Perform the same computation in serial for timing comparison:

```
tic
b = TreeBagger(1000,X,Y,'Method','r','OOBVarImp','on',...
'cat',16:25,'MinLeaf',1); % No options gives serial
toc
```

Elapsed time is 21.747950 seconds.

Computing in parallel took about 75% of the time of computing serially.

## Concepts of Parallel Computing in Statistics Toolbox

### In this section...

“Subtleties in Parallel Computing” on page 20-6

“Vocabulary for Parallel Computation” on page 20-6

### Subtleties in Parallel Computing

There are two main subtleties in parallel computations:

- Nested parallel evaluations (see “No Nested `parfor` Loops” on page 20-10). Only the outermost `parfor` loop runs in parallel, the others run serially.
- Reproducible results when using random numbers (see “Reproducibility in Parallel Statistical Computations” on page 20-12). How can you get exactly the same results when repeatedly running a parallel computation that uses random numbers?

### Vocabulary for Parallel Computation

- *worker* — An independent MATLAB session that runs code distributed by the *client*.
- *client* — The MATLAB session with which you interact, and that distributes jobs to workers.
- `parfor` — A Parallel Computing Toolbox function that distributes independent code segments to workers (see “Working with `parfor`” on page 20-9).
- *random stream* — A pseudorandom number generator, and the sequence of values it generates. MATLAB implements random streams with the `RandStream` class.
- *reproducible computation* — A computation that can be exactly replicated, even in the presence of random numbers (see “Reproducibility in Parallel Statistical Computations” on page 20-12).

## When to Run Statistical Functions in Parallel

In this section...
“Why Run in Parallel?” on page 20-7
“Factors Affecting Speed” on page 20-7
“Factors Affecting Results” on page 20-8

### Why Run in Parallel?

The main reason to run statistical computations in parallel is to gain speed, meaning to reduce the execution time of your program or functions. “Factors Affecting Speed” on page 20-7 discusses the main items affecting the speed of programs or functions. “Factors Affecting Results” on page 20-8 discusses details that can cause a parallel run to give different results than a serial run.

### Factors Affecting Speed

Some factors that can affect the speed of execution of parallel processing are:

- Parallel environment setup. It takes time to run `parpool` to begin computing in parallel. If your computation is fast, the setup time can exceed any time saved by computing in parallel.
- Parallel overhead. There is overhead in communication and coordination when running in parallel. If function evaluations are fast, this overhead could be an appreciable part of the total computation time. Thus, solving a problem in parallel can be slower than solving the problem serially. For an example, see *Improving Optimization Performance with Parallel Computing* in *MATLAB Digest*, March 2009.
- No nested `parfor` loops. This is described in “Working with `parfor`” on page 20-9. `parfor` does not work in parallel when called from within another `parfor` loop. If you have programmed your custom functions to take advantage of parallel processing, the limitation of no nested `parfor` loops can cause a parallel function to run slower than expected.
- When executing serially, `parfor` loops run slightly slower than `for` loops.
- Passing parameters. Parameters are automatically passed to worker sessions during the execution of parallel computations. If there are many

parameters, or they take a large amount of memory, passing parameters can slow the execution of your computation.

- Contention for resources: network and computing. If the pool of workers has low bandwidth or high latency, parallel computation can be slow.

## **Factors Affecting Results**

Some factors can affect results when using parallel processing. There are several caveats related to `parfor` listed in “Limitations” in the Parallel Computing Toolbox documentation. Some important factors are:

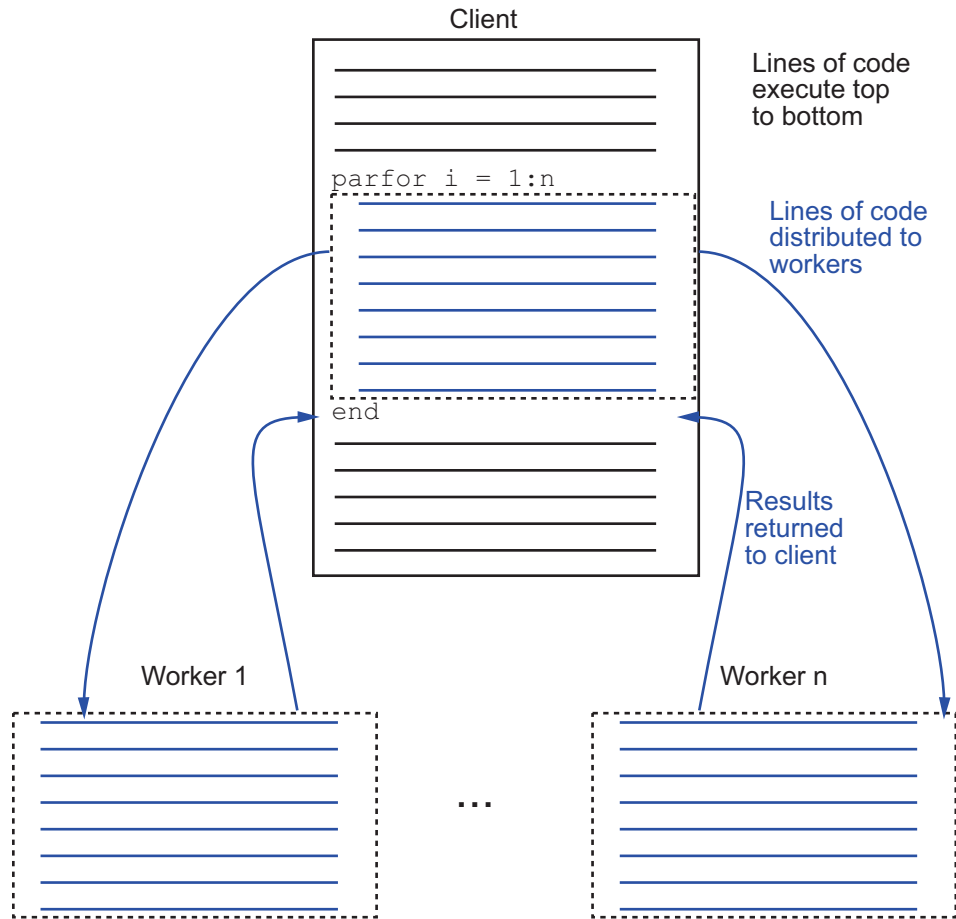
- Persistent or global variables. If any functions use persistent or global variables, these variables can take different values on different worker processors. Furthermore, they might not be cleared properly on the worker processors.
- Accessing external files. External files can be accessed unpredictably during a parallel computation. The order of computations is not guaranteed during parallel processing, so external files can be accessed in unpredictable order, leading to unpredictable results. Furthermore, if multiple processors try to read an external file simultaneously, the file can become locked, leading to a read error, and halting function execution.
- Noncomputational functions, such as `input`, `plot`, and `keyboard`, can behave badly when used in your custom functions. When called in a `parfor` loop, these functions are executed on worker machines. This can cause a worker to become nonresponsive, since it is waiting for input.
- `parfor` does not allow `break` or `return` statements.
- The random numbers you use can affect the results of your computations. See “Reproducibility in Parallel Statistical Computations” on page 20-12.

## Working with parfor

In this section...
“How Statistical Functions Use parfor” on page 20-9
“Characteristics of parfor” on page 20-10

### How Statistical Functions Use parfor

parfor is a Parallel Computing Toolbox function similar to a for loop. Parallel statistical functions call parfor internally. parfor distributes computations to worker processors.



## Characteristics of parfor

More caveats related to `parfor` appear in “Limitations” in the Parallel Computing Toolbox documentation.

### No Nested parfor Loops

`parfor` does not work in parallel when called from within another `parfor` loop, or from an `spmd` block. Parallelization occurs only at the outermost level.



Suppose, for example, you want to apply `jackknife` to your function `userfcn`, which calls `parfor`, and you want to call `jackknife` in a loop. The following figure shows three cases:

- 1 The outermost loop is `parfor`. Only that loop runs in parallel.
- 2 The outermost `parfor` loop is in `jackknife`. Only `jackknife` runs in parallel.
- 3 The outermost `parfor` loop is in `userfcn`. `userfcn` uses `parfor` in parallel.

**Bold** indicates the function that runs in parallel

① 

```
...
parfor i=1:10
    x(i)=jackknife(@userfcn,...)
...
end
```

← Only the outermost `parfor` loop runs in parallel

② 

```
...
for i=1:10
    x(i)=jackknife(@userfcn,...)
...
end
```

← If `UseParallel` = 'always' `jackknife` runs in parallel

③ 

```
...
for i=1:10
    x(i)=jackknife(@userfcn,...)
...
end
```

← If `UseParallel` = 'never' `userfcn` can use `parfor` in parallel

### When `parfor` Runs in Parallel

## Reproducibility in Parallel Statistical Computations

### In this section...

“Issues and Considerations in Reproducing Parallel Computations” on page 20-12

“Running Reproducible Parallel Computations” on page 20-13

“Parallel Statistical Computation Using Random Numbers” on page 20-14

### Issues and Considerations in Reproducing Parallel Computations

A *reproducible* computation is one that gives the same results every time it runs. Reproducibility is important for:

- Debugging — To correct an anomalous result, you need to reproduce the result.
- Confidence — When you can reproduce results, you can investigate and understand them.
- Modifying existing code — When you change existing code, you want to ensure that you do not break anything.

Generally, you do not need to ensure reproducibility for your computation. Often, when you want reproducibility, the simplest technique is to run in serial instead of in parallel. In serial computation you can simply call the `rng` function as follows:

```
s = rng % Obtain the current state of the random stream
% run the statistical function
rng(s) % Reset the stream to the previous state
% run the statistical function again, obtain identical results
```

This section addresses the case when your function uses random numbers, and you want reproducible results in parallel. This section also addresses the case when you want the same results in parallel as in serial.

## Running Reproducible Parallel Computations

To run a Statistics Toolbox function reproducibly:

- 1 Set the `UseSubstreams` option to `true`.
- 2 Set the `Streams` option to a type that supports substreams: `'mlfg6331_64'` or `'mrg32k3a'`. For information on these streams, see “Choosing a Random Number Generator” in the MATLAB Mathematics documentation.
- 3 To compute in parallel, set the `UseParallel` option to `true`.
- 4 Call the function with the options structure.
- 5 To reproduce the computation, reset the stream, then call the function again.

To understand why this technique gives reproducibility, see “How Substreams Enable Reproducible Parallel Computations” on page 20-14.

For example, to use the `'mlfg6331_64'` stream for reproducible computation:

- 1 Create an appropriate options structure:

```
s = RandStream('mlfg6331_64');
options = statset('UseParallel',true, ...
    'Streams',s,'UseSubstreams',true);
```
- 2 Run your parallel computation. For instructions, see “Quick Start Parallel Computing for Statistics Toolbox” on page 20-2.
- 3 Reset the random stream:

```
reset(s);
```
- 4 Rerun your parallel computation. You obtain identical results.

For an example of a parallel computation run this reproducible way, see “Reproducible Parallel Bootstrap” on page 20-24.

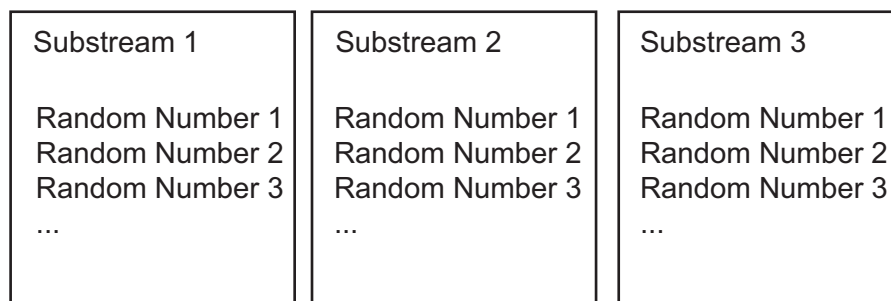
## Parallel Statistical Computation Using Random Numbers

### What Are Substreams?

A *substream* is a portion of a random stream that `RandStream` can access quickly. There is a number  $M$  such that for any positive integer  $k$ , `RandStream` can go the  $kM$ th pseudorandom number in the stream. From that point, `RandStream` can generate the subsequent entries in the stream. Currently, `RandStream` has  $M = 2^{72}$ , about  $5e21$ , or more.



The entries in different substreams have good statistical properties, similar to the properties of entries in a single stream: independence, and lack of  $k$ -way correlation at various lags. The substreams are so long that you can view the substreams as being independent streams, as in the following picture.



Two `RandStream` stream types support substreams: `'mlfg6331_64'` and `'mrg32k3a'`.

### How Substreams Enable Reproducible Parallel Computations

When MATLAB performs computations in parallel with `parfor`, each worker receives loop iterations in an unpredictable order. Therefore, you cannot

predict which worker gets which iteration, so cannot determine the random numbers associated with each iteration.

Substreams allow MATLAB to tie each iteration to a particular sequence of random numbers. `parfor` gives each iteration an index. The iteration uses the index as the substream number. Since the random numbers are associated with the iterations, not with the workers, the entire computation is reproducible.

To obtain reproducible results, simply reset the stream, and all the substreams generate identical random numbers when called again. This method succeeds when all the workers use the same stream, and the stream supports substreams. This concludes the discussion of how the procedure in “Running Reproducible Parallel Computations” on page 20-13 gives reproducible parallel results.

### **Random Numbers on the Client or Workers**

A few functions generate random numbers on the client before distributing them to parallel workers. The workers do not use random numbers, so operate purely deterministically. For these functions, you can run a parallel computation reproducibly using any random stream type.

The functions that operate this way include:

- `crossval`
- `plsregress`
- `sequentialfs`

To obtain identical results, reset the random stream on the client, or the random stream you pass to the client. For example:

```
s = rng % Obtain the current state of the random stream
% run the statistical function
rng(s) % Reset the stream to the previous state
% run the statistical function again, obtain identical results
```

While this method enables you to run reproducibly in parallel, the results can differ from a serial computation. The reason for the difference is `parfor` loops run in reverse order from `for` loops. Therefore, a serial computation can

generate random numbers in a different order than a parallel computation. For unequivocal reproducibility, use the technique in “Running Reproducible Parallel Computations” on page 20-13.

### Distributing Streams Explicitly

For testing or comparison using particular random number algorithms, you must set the random number generators. How do you set these generators in parallel, or initialize streams on each worker in a particular way? Or you might want to run a computation using a different sequence of random numbers than any other you have run. How can you ensure the sequence you use is statistically independent?

Parallel Statistics Toolbox functions allow you to set random streams on each worker explicitly. For information on *creating* multiple streams, enter `help RandStream/create` at the command line. To create four independent streams using the 'mrg32k3a' generator:

```
s = RandStream.create('mrg32k3a','NumStreams',4,...  
    'CellOutput',true);
```

Pass these streams to a statistical function using the `Streams` option. For example:

```
parpool(4) % if you have at least 4 cores  
s = RandStream.create('mrg32k3a','NumStreams',4,...  
    'CellOutput',true); % create 4 independent streams  
paroptions = statset('UseParallel',true,...  
    'Streams',s); % set the 4 different streams  
x = [randn(700,1); 4 + 2*randn(300,1)];  
latt = -4:0.01:12;  
myfun = @(X) ksdensity(X,latt);  
pdfestimate = myfun(x);  
B = bootstrp(200,myfun,x,'Options',paroptions);
```

This method of distributing streams gives each worker a different stream for the computation. However, it does not allow for a reproducible computation, because the workers perform the 200 bootstraps in an unpredictable order. If you want to perform a reproducible computation, use substreams as described in “Running Reproducible Parallel Computations” on page 20-13.

If you set the `UseSubstreams` option to `true`, then set the `Streams` option to a single random stream of the type that supports substreams (`'m1fg6331_64'` or `'mrg32k3a'`). This setting gives reproducible computations.

## Examples of Parallel Statistical Functions

### In this section...

“Parallel Jackknife” on page 20-18

“Parallel Cross Validation” on page 20-19

“Parallel Bootstrap” on page 20-21

### Parallel Jackknife

This example is from the jackknife function reference page, but runs in parallel.

```
mypool=parpool()
Starting parpool using the 'local' profile ... connected to 2 workers.
```

```
mypool =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}
  NumWorkers: 2
  IdleTimeout: 30
  Cluster: [1x1 parallel.cluster.Local]
  RequestQueue: [1x1 parallel.RequestQueue]
  SpmdEnabled: 1
```

```
opts = statset('UseParallel',true);
sigma = 5;
rng('default')
y = normrnd(0,sigma,100,1);
m = jackknife(@var, y,1,'Options',opts);
n = length(y);
bias = -sigma^2 / n % known bias formula
jbias = (n - 1)*(mean(m)-var(y,1)) % jackknife bias estimate
```

```
bias =
```

```
-0.2500
```



```
jbias =
    -0.3378
```

This simple example is not a good candidate for parallel computation:

```
% How long to compute in serial?
tic;m = jackknife(@var,y,1);toc
Elapsed time is 0.022771 seconds.
```

```
% How long to compute in parallel?
tic;m = jackknife(@var,y,1,'Options',opts);toc
Elapsed time is 0.299066 seconds.
```

jackknife does not use random numbers, so gives the same results every time, whether run in parallel or serial.

## Parallel Cross Validation

- “Simple Parallel Cross Validation” on page 20-19
- “Reproducible Parallel Cross Validation” on page 20-20

### Simple Parallel Cross Validation

This example is the same as the first in the `crossval` function reference page, but runs in parallel.

```
mypool = parpool()
Starting parpool using the 'local' profile ... connected to 2 workers.

mypool =
```

Pool with properties:

```
AttachedFiles: {0x1 cell}
NumWorkers: 2
IdleTimeout: 30
Cluster: [1x1 parallel.cluster.Local]
```

```
RequestQueue: [1x1 parallel.RequestQueue]
SpmEnabled: 1opts = statset('UseParallel',true);

load('fisheriris');
y = meas(:,1);
X = [ones(size(y,1),1),meas(:,2:4)];
regf=@(XTRAIN,ytrain,XTEST)(XTEST*regress(ytrain,XTRAIN));

cvMse = crossval('mse',X,y,'Predfun',regf,'Options',opts)

cvMse =

    0.1028
```

This simple example is not a good candidate for parallel computation:

```
% How long to compute in serial?
tic;cvMse = crossval('mse',X,y,'Predfun',regf);toc
Elapsed time is 0.073438 seconds.

% How long to compute in parallel?
tic;cvMse = crossval('mse',X,y,'Predfun',regf,...
    'Options',opts);toc
Elapsed time is 0.289585 seconds.
```

### Reproducible Parallel Cross Validation

To run `crossval` in parallel in a reproducible fashion, set the options and reset the random stream appropriately (see “Running Reproducible Parallel Computations” on page 20-13).

```
mypool = parpool()
```

Starting `parpool` using the 'local' profile ... connected to 2 workers.

```
mypool =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}
```

```

        NumWorkers: 2
        IdleTimeout: 30
            Cluster: [1x1 parallel.cluster.Local]
        RequestQueue: [1x1 parallel.RequestQueue]
        SpmdEnabled: 1

s = RandStream('mlfg6331_64');
opts = statset('UseParallel',true,...
    'Streams',s,'UseSubstreams',true);

load('fisheriris');
y = meas(:,1);
X = [ones(size(y,1),1),meas(:,2:4)];
regf=@(XTRAIN,ytrain,XTEST)(XTEST*regress(ytrain,XTRAIN));

cvMse = crossval('mse',X,y,'Predfun',regf,'Options',opts)

cvMse =

    0.1020

Reset the stream:

reset(s)
cvMse = crossval('mse',X,y,'Predfun',regf,'Options',opts)

cvMse =

    0.1020

```

## Parallel Bootstrap

- “Bootstrap in Serial and Parallel” on page 20-21
- “Reproducible Parallel Bootstrap” on page 20-24

### Bootstrap in Serial and Parallel

Here is an example timing a bootstrap in parallel versus in serial. The example generates data from a mixture of two Gaussians, constructs a

nonparametric estimate of the resulting data, and uses a bootstrap to get a sense of the sampling variability.

**1** Generate the data:

```
% Generate a random sample of size 1000,  
% from a mixture of two Gaussian distributions  
x = [randn(700,1); 4 + 2*randn(300,1)];
```

**2** Construct a nonparametric estimate of the density from the data:

```
latt = -4:0.01:12;  
myfun = @(X) ksdensity(X,latt);  
pdfestimate = myfun(x);
```

**3** Bootstrap the estimate to get a sense of its sampling variability. Run the bootstrap in serial for timing comparison.

```
tic;B = bootstrp(200,myfun,x);toc
```

Elapsed time is 10.878654 seconds.

**4** Run the bootstrap in parallel for timing comparison:

```
mypool = parpool()  
Starting parpool using the 'local' profile ... connected to 2 workers.
```

```
mypool =
```

```
Pool with properties:
```

```
AttachedFiles: {0x1 cell}  
NumWorkers: 2  
IdleTimeout: 30  
Cluster: [1x1 parallel.cluster.Local]  
RequestQueue: [1x1 parallel.RequestQueue]  
SpmEnabled: 1
```

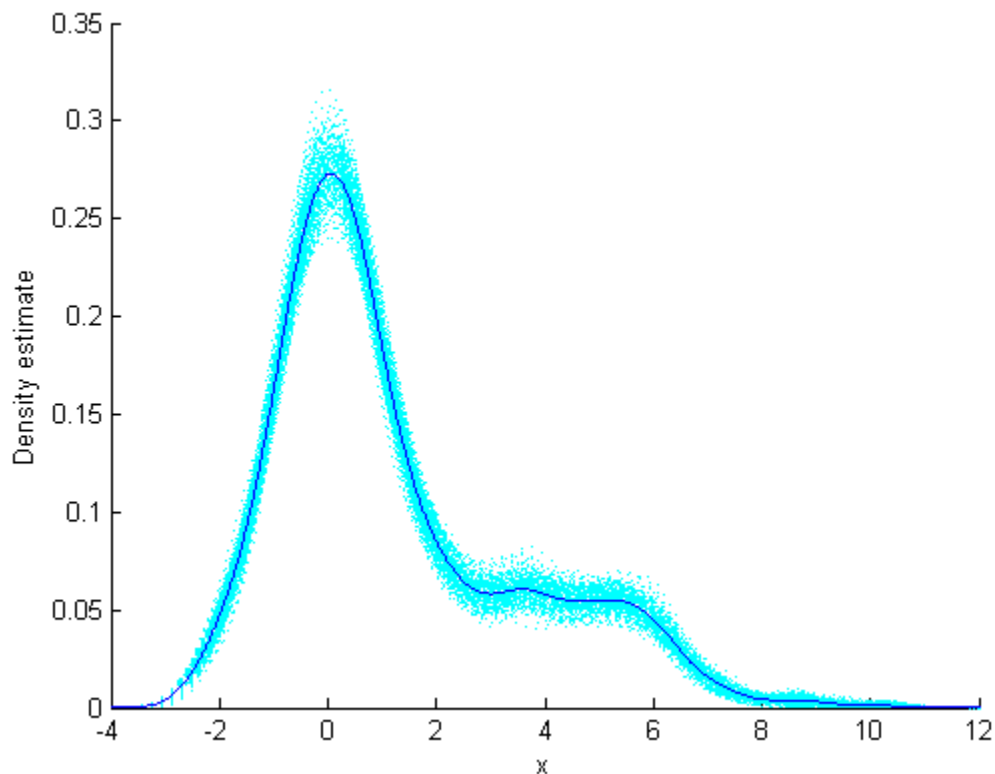
```
opt = statset('UseParallel',true);  
tic;B = bootstrp(200,myfun,x,'Options',opt);toc
```

Elapsed time is 6.304077 seconds.

Computing in parallel is nearly twice as fast as computing in serial for this example.

Overlay the `ksdensity` density estimate with the 200 bootstrapped estimates obtained in the parallel bootstrap. You can get a sense of how to assess the accuracy of the density estimate from this plot.

```
hold on
for i=1:size(B,1),
    plot(latt,B(i,:), 'c: ')
end
plot(latt,pdfestimate);
xlabel('x');ylabel('Density estimate')
```



### Reproducible Parallel Bootstrap

To run the example in parallel in a reproducible fashion, set the options appropriately (see “Running Reproducible Parallel Computations” on page 20-13). First set up the problem and parallel environment as in “Bootstrap in Serial and Parallel” on page 20-21. Then set the options to use substreams along with a stream that supports substreams.

```
s = RandStream('mlfg6331_64'); % has substreams
opts = statset('UseParallel',true,...
    'Streams',s,'UseSubstreams',true);
B2 = bootstrp(200,myfun,x,'Options',opts);
```

To rerun the bootstrap and get the same result:

```
reset(s) % set the stream to initial state  
B3 = bootstrp(200,myfun,x,'Options',opts);  
isequal(B2,B3) % check if same results
```

```
ans =  
    1
```





# Functions — Alphabetical List

---

# addedvarplot

---

**Purpose** Added variable plot

**Syntax** `addedvarplot(X,y,num,inmodel)`  
`addedvarplot(X,y,num,inmodel,stats)`

**Description** `addedvarplot(X,y,num,inmodel)` displays an added variable plot using the predictive terms in  $X$ , the response values in  $y$ , the added term in column `num` of  $X$ , and the model with current terms specified by `inmodel`.  $X$  is an  $n$ -by- $p$  matrix of  $n$  observations of  $p$  predictive terms.  $y$  is vector of  $n$  response values. `num` is a scalar index specifying the column of  $X$  with the term to be added. `inmodel` is a logical vector of  $p$  elements specifying the columns of  $X$  in the current model. By default, all elements of `inmodel` are `false`.

---

**Note** `addedvarplot` automatically includes a constant term in all models. Do not enter a column of 1s directly into  $X$ .

---

`addedvarplot(X,y,num,inmodel,stats)` uses the `stats` output from the `stepwisefit` function to improve the efficiency of repeated calls to `addedvarplot`. Otherwise, this syntax is equivalent to the previous syntax.

Added variable plots are used to determine the unique effect of adding a new term to a multilinear model. The plot shows the relationship between the part of the response unexplained by terms already in the model and the part of the new term unexplained by terms already in the model. The “unexplained” parts are measured by the residuals of the respective regressions. A scatter of the residuals from the two regressions forms the added variable plot.

In addition to the scatter of residuals, the plot produced by `addedvarplot` shows 95% confidence intervals on predictions from the fitted line. The fitted line has intercept zero because, under typical linear model assumptions, both of the plotted variables have mean zero. The slope of the fitted line is the coefficient that the new term would have if it were added to the model with terms `inmodel`.

Added variable plots are sometimes known as partial regression leverage plots.

## Examples

Load the data in `hald.mat`, which contains observations of the heat of reaction of various cement mixtures:

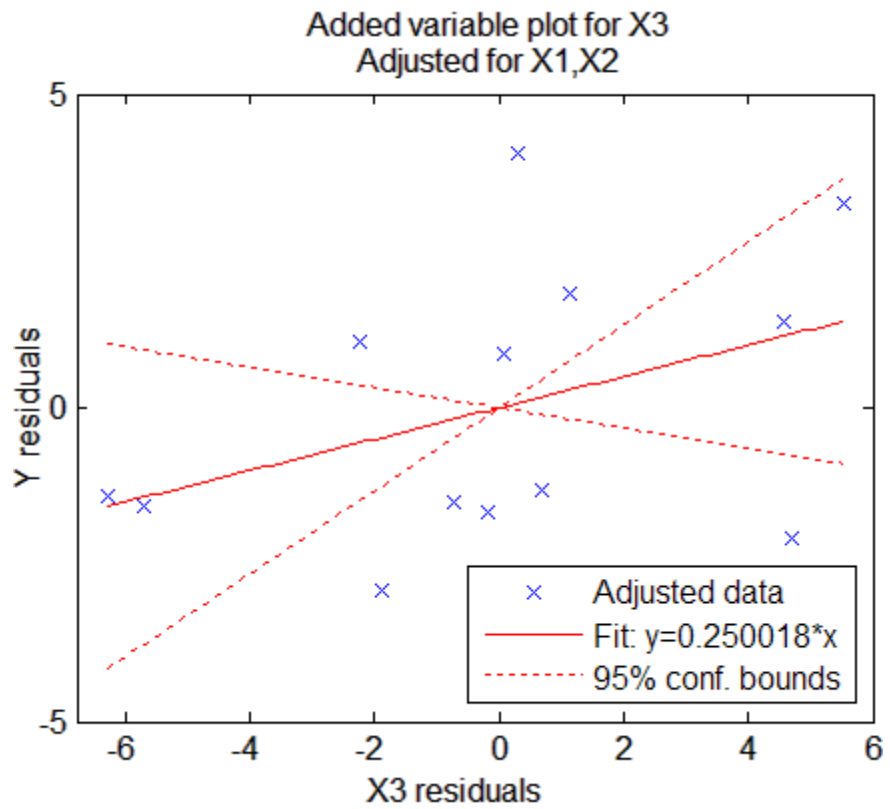
```
load hald
whos
```

Name	Size	Bytes	Class	Attributes
Description	22x58	2552	char	
hald	13x5	520	double	
heat	13x1	104	double	
ingredients	13x4	416	double	

Create an added variable plot to investigate the addition of the third column of `ingredients` to a model consisting of the first two columns:

```
inmodel = [true true false false];
addedvarplot(ingredients,heat,3,inmodel)
```

# addedvarplot



The wide scatter and the low slope of the fitted line are evidence against the statistical significance of adding the third column to the model.

## See Also

`stepwisefit` | `stepwise`

**Purpose** Add levels to categorical array

**Syntax** `B = addlevels(A,newlevels)`

**Description** `B = addlevels(A,newlevels)` adds new levels to the categorical array `A`. `newlevels` is a cell array of strings or a 2-D character matrix that specifies the levels to add. `addlevels` adds the new levels at the end of the list of possible categorical levels in `A`, but does not modify the value of any element. `B` does not contain elements at the new levels.

## Examples **Example 1**

Add levels for additional species in Fisher's iris data:

```
load fisheriris
species = nominal(species,...
                  {'Species1','Species2','Species3'},...
                  {'setosa','versicolor','virginica'});
species = addlevels(species,{'Species4','Species5'});
getlabels(species)
ans =
    'Species1' 'Species2' 'Species3' 'Species4' 'Species5'
```

## Example 2

- 1 Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
                  'delimiter',';',...
                  'ReadObsNames',true);
```

- 2 Make the `{0,1}`-valued variable `smoke` nominal, and change the labels to 'No' and 'Yes':

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

## categorical.addlevels

---

- 3** Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                           {'0-5 Years', '5-10 Years', 'LongTerm'});
```

- 4** Assuming the nonsmokers have never smoked, relabel the 'No' level:

```
patients.smoke = setlabels(patients.smoke, 'Never', 'No');
```

- 5** Drop the undifferentiated 'Yes' level from `smoke`:

```
patients.smoke = droplevels(patients.smoke, 'Yes');
```

Warning: `OLDLEVELS` contains categorical levels that were present in `A`, caused some array elements to have undefined levels.

Note that smokers now have an undefined level.

- 6** Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

### See Also

`droplevels` | `getlabels` | `islevel` | `mergelevels` | `reorderlevels`

**Purpose**

Add listener for event

**Syntax**

```
e1 = addlistener(hsource, 'eventname', callback)
e1 = addlistener(hsource, property, 'eventname', callback)
```

**Description**

`e1 = addlistener(hsource, 'eventname', callback)` creates a listener for the event named `eventname`, the source of which is handle object `hsource`. If `hsource` is an array of source handles, the listener responds to the named event on any handle in the array. `callback` is a function handle that is invoked when the event is triggered.

`e1 = addlistener(hsource, property, 'eventname', callback)` adds a listener for a property event. `eventname` must be one of the strings `'PreGet'`, `'PostGet'`, `'PreSet'`, and `'PostSet'`. `property` must be either a property name or cell array of property names, or a `meta.property` or array of `meta.property`. The properties must belong to the class of `hsource`. If `hsource` is scalar, `property` can include dynamic properties.

For all forms, `addlistener` returns an `event.listener`. To remove a listener, delete the object returned by `addlistener`. For example, `delete(e1)` calls the handle class `delete` method to remove the listener and delete it from the workspace.

**See Also**

`delete` | `dynamicprops` | `event.listener` | `events` | `meta.property`  
| `notify` | `grandstream` | `reset`

# GeneralizedLinearModel.addTerms

---

<b>Purpose</b>	Add terms to generalized linear model
<b>Syntax</b>	<code>mdl1 = addTerms(mdl, terms)</code>
<b>Description</b>	<code>mdl1 = addTerms(mdl, terms)</code> returns a generalized linear model the same as <code>mdl</code> but with additional terms.
<b>Input Arguments</b>	<p><b>mdl</b></p> <p>Generalized linear model, as constructed by <code>fitglm</code> or <code>stepwiseglm</code>.</p> <p><b>terms</b></p> <p>Terms to add to the <code>mdl</code> regression model. Specify as either a:</p> <ul style="list-style-type: none"><li>• Text string representing one or more terms to add. For details, see “Wilkinson Notation” on page 21-8.</li><li>• Row or rows in the terms matrix (see <code>modelspec</code> in <code>fitglm</code>). For example, if there are three variables A, B, and C:  <code>[0 0 0]</code> represents a constant term or intercept <code>[0 1 0]</code> represents B; equivalently, <math>A^0 * B^1 * C^0</math> <code>[1 0 1]</code> represents <math>A * C</math> <code>[2 0 0]</code> represents <math>A^2</math> <code>[0 1 2]</code> represents <math>B * (C^2)</math></li></ul>
<b>Output Arguments</b>	<p><b>mdl1</b></p> <p>Generalized linear model, the same as <code>mdl</code> but with additional terms given in <code>terms</code>. You can set <code>mdl1</code> equal to <code>mdl</code> to overwrite <code>mdl</code>.</p>
<b>Definitions</b>	<p><b>Wilkinson Notation</b></p> <p>Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.</p>



Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

For details, see Wilkinson and Rogers [1].

## Examples

### Add a term to a generalized linear regression model

Create a model using just one predictor, then add a second.

Generate artificial data for the model, Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data. Use just the first predictor in the model.

```
mdl = fitglm(X,y,...
```

# GeneralizedLinearModel.addTerms

---

```
'y ~ x1', 'distr', 'poisson')

mdl =

Generalized Linear regression model:
  log(y) ~ 1 + x1
  Distribution = Poisson

Estimated Coefficients:
              Estimate    SE          tStat    pValue
(Intercept)  2.7784      0.014043   197.85    0
x1           1.1732      0.0033653  348.6    0

100 observations, 98 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.25e+05, p-value = 0

Add the second predictor to the model.

mdl1 = addTerms(mdl, 'x2')

mdl1 =

Generalized Linear regression model:
  log(y) ~ 1 + x1 + x2
  Distribution = Poisson

Estimated Coefficients:
              Estimate    SE          tStat    pValue
(Intercept)  1.0405      0.022122   47.034    0
x1           0.9968      0.003362   296.49    0
x2           1.987       0.0063433  313.24    0

100 observations, 97 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 2.95e+05, p-value = 0
```

## References

[1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. J. Royal Statistics Society 22, pp. 392–399, 1973.

## Alternatives

step adds or removes terms from a model using a greedy one-step algorithm.

## See Also

GeneralizedLinearModel | removeTerms | step | stepwiseglm

## Concepts

- “Generalized Linear Models” on page 10-12

# LinearModel.addTerms

---

<b>Purpose</b>	Add terms to linear regression model
<b>Syntax</b>	<code>mdl1 = addTerms(mdl, terms)</code>
<b>Description</b>	<code>mdl1 = addTerms(mdl, terms)</code> returns a linear model the same as <code>mdl</code> but with additional terms.
<b>Input Arguments</b>	<p><b>mdl</b></p> <p>Linear model, as constructed by <code>fitlm</code> or <code>stepwiselm</code>.</p> <p><b>terms</b></p> <p>Terms to add to the <code>mdl</code> regression model. Specify as either a:</p> <ul style="list-style-type: none"><li>• Text string representing one or more terms to add. For details, see “Wilkinson Notation” on page 21-12.</li><li>• Row or rows in the terms matrix (see <code>modelspec</code> in <code>fitlm</code>). For example, if there are three variables A, B, and C:  <code>[0 0 0]</code> represents a constant term or intercept <code>[0 1 0]</code> represents B; equivalently, <math>A^0 * B^1 * C^0</math> <code>[1 0 1]</code> represents <math>A * C</math> <code>[2 0 0]</code> represents <math>A^2</math> <code>[0 1 2]</code> represents <math>B * (C^2)</math></li></ul>
<b>Output Arguments</b>	<p><b>mdl1</b></p> <p>Linear model, the same as <code>mdl</code> but with additional terms given in terms. You can set <code>mdl1</code> equal to <code>mdl</code> to overwrite <code>mdl</code>.</p>
<b>Definitions</b>	<p><b>Wilkinson Notation</b></p> <p>Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.</p>

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

For details, see Wilkinson and Rogers [1].

## Examples

### Add a Term to a Model

Create a model of the `carsmall` data without any interactions, then add an interaction term.

Load the `carsmall` data and make a model of the MPG as a function of weight and model year.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Add an interaction term to `mdl`.

```
terms = 'Year*Weight';
mdl1 = addTerms(mdl,terms)
```

# LinearModel.addTerms

---

```
mdl1 =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight*Year + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	48.045	6.779	7.0874	3.3967e-10
Weight	-0.012624	0.0041455	-3.0454	0.0030751
Year_76	2.7768	3.0538	0.90931	0.3657
Year_82	16.416	4.9802	3.2962	0.0014196
Weight:Year_76	-0.00020693	0.00092403	-0.22394	0.82333
Weight:Year_82	-0.0032574	0.0018919	-1.7217	0.088673
Weight^2	1.0121e-06	6.12e-07	1.6538	0.10177

```
Number of observations: 94, Error degrees of freedom: 87
```

```
Root Mean Squared Error: 2.76
```

```
R-squared: 0.89, Adjusted R-Squared 0.882
```

```
F-statistic vs. constant model: 117, p-value = 1.88e-39
```

## References

[1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. J. Royal Statistics Society 22, pp. 392–399, 1973.

## Alternatives

Use `stepwiselm` to select a model from a starting model, continuing until no single step is beneficial.

Use `removeTerms` to remove particular terms.

Use `step` to optimally improve the model by adding or removing terms.

## See Also

`LinearModel` | `removeTerms` | `step` | `stepwiselm`

## How To

- “Linear Regression” on page 9-12

<b>Purpose</b>	Anderson-Darling test
<b>Syntax</b>	<pre>h = adtest(x) h = adtest(x,Name,Value) [h,p] = adtest( ___ ) [h,p,adstat,cv] = adtest( ___ )</pre>
<b>Description</b>	<p><code>h = adtest(x)</code> returns a test decision for the null hypothesis that the data in vector <code>x</code> is from a population with a normal distribution, using the Anderson-Darling test. The alternative hypothesis is that <code>x</code> is not from a population with a normal distribution. The result <code>h</code> is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.</p> <p><code>h = adtest(x,Name,Value)</code> returns a test decision for the Anderson-Darling test with additional options specified by one or more name-value pair arguments. For example, you can specify a null distribution other than normal, or select an alternative method for calculating the <i>p</i>-value.</p> <p><code>[h,p] = adtest( ___ )</code> also returns the <i>p</i>-value, <code>p</code>, of the Anderson-Darling test, using any of the input arguments from the previous syntaxes.</p> <p><code>[h,p,adstat,cv] = adtest( ___ )</code> also returns the test statistic, <code>adstat</code>, and the critical value, <code>cv</code>, for the Anderson-Darling test.</p>
<b>Input Arguments</b>	<p><b>x - Sample data</b> vector</p> <p>Sample data, specified as a vector. Missing observations in <code>x</code>, indicated by NaN, are ignored.</p> <p><b>Data Types</b> single   double</p>

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

**Example:** `'Alpha', 0.01, 'MCTol', 0.01` conducts the hypothesis test at the 1% significance level, and determines the p-value,  $p$ , using a Monte Carlo simulation with a maximum Monte Carlo standard error for  $p$  of 0.01.

## 'Distribution' - Hypothesized distribution

`'norm'` (default) | `'exp'` | `'ev'` | `'logn'` | `'weibull'` | probability distribution object

Hypothesized distribution of data vector  $x$ , specified as the comma-separated pair consisting of `'Distribution'` and one of the following.

<code>'norm'</code>	Normal distribution
<code>'exp'</code>	Exponential distribution
<code>'ev'</code>	Extreme value distribution
<code>'logn'</code>	Lognormal distribution
<code>'weibull'</code>	Weibull distribution

In this case, you do not need to specify population parameters. Instead, `adtest` estimates the distribution parameters from the sample data and tests  $x$  against a composite hypothesis that it comes from the selected distribution family with parameters unspecified.

Alternatively, you can specify any continuous probability distribution object for the null distribution. In this case, you must specify all the distribution parameters, and `adtest` tests  $x$  against a simple hypothesis that it comes from the given distribution with its specified parameters.



**Example:** 'Distribution','exp'

### **'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

### **Data Types**

single | double

### **'MCTol' - Maximum Monte Carlo standard error**

positive scalar value

Maximum Monte Carlo standard error for the  $p$ -value,  $p$ , specified as the comma-separated pair consisting of 'MCTol' and a positive scalar value. If you use MCTol, adtest determines  $p$  using a Monte Carlo simulation, and the name-value pair argument Asymptotic must have the value false.

**Example:** 'MCTol',0.01

### **Data Types**

single | double

### **'Asymptotic' - Method for calculating $p$ -value**

false (default) | true

Method for calculating the  $p$ -value of the Anderson-Darling test, specified as the comma-separated pair consisting of 'Asymptotic' and either true or false. If you specify 'true', adtest estimates the  $p$ -value using the limiting distribution of the Anderson-Darling test statistic. If you specify false, adtest calculates the  $p$ -value based on an analytical formula. For sample sizes greater than 120, the limiting distribution estimate is likely to be more accurate than the small sample size approximation method.

## Output Arguments

- If you specify a distribution family with unknown parameters for the Distribution name-value pair, `Asymptotic` must be `false`.
- If you use `MCTol` to calculate the  $p$ -value using a Monte Carlo simulation, `Asymptotic` must be `false`.

**Example:** `'Asymptotic',true`

### Data Types

logical

### **h** - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** - $p$ -value

scalar value in the range [0,1]

$p$ -value of the Anderson-Darling test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis.  $p$  is calculated using one of these methods:

- If the hypothesized distribution is a fully specified probability distribution object, `adtest` calculates  $p$  analytically. If `'Asymptotic'` is `true`, `adtest` uses the asymptotic distribution of the test statistic. If you specify a value for `'MCTol'`, `adtest` uses a Monte Carlo simulation.
- If the hypothesized distribution is specified as a distribution family with unknown parameters, `adtest` retrieves the critical value from a table and uses inverse interpolation to determine the  $p$ -value. If you specify a value for `'MCTol'`, `adtest` uses a Monte Carlo simulation.

**adstat - Test statistic**

scalar value

Test statistic for the Anderson-Darling test, returned as a scalar value.

- If the hypothesized distribution is a fully specified probability distribution object, `adtest` computes `adstat` using specified parameters.
- If the hypothesized distribution is specified as a distribution family with unknown parameters, `adtest` computes `adstat` using parameters estimated from the sample data.

**cv - Critical value**

scalar value

Critical value for the Anderson-Darling test at the significance level Alpha, returned as a scalar value. `adtest` determines `cv` by interpolating into a table based on the specified Alpha significance level.

**Examples****Test for a Normal Distribution**

Load the data set. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Test the null hypothesis that the exam grades come from a normal distribution. You do not need to specify values for the population parameters.

```
[h,p,adstat,cv] = adtest(x);
```

```
h =  
    0
```

```
p =  
    0.1854
```

```
adstat =  
    0.5194  
  
cv =  
    0.7470
```

The returned value of  $h = 0$  indicates that `adtest` fails to reject the null hypothesis at the default 5% significance level.

## Test for an Extreme Value Distribution

Load the data set. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Test the null hypothesis that the exam grades come from an extreme value distribution. You do not need to specify values for the population parameters.

```
[h,p] = adtest(x,'Distribution','ev')  
  
h =  
    0  
  
p =  
    0.0714
```

The returned value of  $h = 0$  indicates that `adtest` fails to reject the null hypothesis at the default 5% significance level.

## Specify the Hypothesized Distribution Using a Probability Distribution Object

Load the data set. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;
```

```
x = grades(:,1);
```

Create a normal probability distribution object with mean  $\mu = 75$  and standard deviation  $\sigma = 10$ .

```
dist = makedist('normal','mu',75,'sigma',10)
```

```
dist =
```

```
prob.NormalDistribution  
Package: prob
```

```
Normal distribution  
mu = 75  
sigma = 10
```

```
Properties, Methods
```

Test the null hypothesis that  $x$  comes from the hypothesized normal distribution.

```
[h,p] = adtest(x,'Distribution',dist)
```

```
h =  
    0
```

```
p =  
    0.4687
```

The returned value of  $h = 0$  indicates that `adtest` fails to reject the null hypothesis at the default 5% significance level.

## Definitions

### Anderson-Darling Test

The Anderson-Darling test is commonly used to test whether a data sample comes from a normal distribution. However, it can be used to test for another hypothesized distribution, even if you do not fully specify the distribution parameters. Instead, the test estimates any unknown parameters from the data sample.

The test statistic belongs to the family of quadratic empirical distribution function statistics, which measure the distance between the hypothesized distribution,  $F(x)$  and the empirical cdf,  $F_n(x)$  as

$$n \int_{-\infty}^{\infty} (F_n(x) - F(x))^2 w(x) dF(x),$$

over the ordered sample values  $x_1 < x_2 < \dots < x_n$ , where  $w(x)$  is a weight function and  $n$  is the number of data points in the sample.

The weight function for the Anderson-Darling test is

$$w(x) = [F(x)(1 - F(x))]^{-1},$$

which places greater weight on the observations in the tails of the distribution, thus making the test more sensitive to outliers and better at detecting departure from normality in the tails of the distribution.

The Anderson-Darling test statistic is

$$A_n^2 = -n - \sum_{i=1}^n \frac{2i-1}{n} [\ln(F(X_i)) + \ln(1 - F(X_{n+1-i}))],$$

where  $\{X_1 < \dots < X_n\}$  are the ordered sample data points and  $n$  is the number of data points in the sample.

In `adtest`, the decision to reject or not reject the null hypothesis is based on comparing the  $p$ -value for the hypothesis test with the specified significance level, not on comparing the test statistic with the critical value.

## Monte Carlo Standard Error

The Monte Carlo standard error is the error due to simulating the  $p$ -value.

The Monte Carlo standard error is calculated as

$$SE = \sqrt{\frac{(\hat{p})(1-\hat{p})}{\text{mcreps}}},$$

where  $\hat{p}$  is the estimated  $p$ -value of the hypothesis test, and `mcreps` is the number of Monte Carlo replications performed.

`adtest` chooses the number of Monte Carlo replications, `mcreps`, large enough to make the Monte Carlo standard error for  $\hat{p}$  less than the value specified for `MCTol`.

**See Also**

`kstest` | `jbtest`

# gmdistribution.AIC property

---

**Purpose** Akaike Information Criterion

**Description** The Akaike Information Criterion:  $2 \cdot N \log L + 2 \cdot m$ , where  $m$  is the number of estimated parameters.

---

**Note** This property applies only to gmdistribution objects constructed with `fit`.

---



**Purpose**

Andrews plot

**Syntax**

```
andrewsplot(X)
andrewsplot(X, ..., 'Standardize', standopt)
andrewsplot(X, ..., 'Quantile', alpha)
andrewsplot(X, ..., 'Group', group)
andrewsplot(X, ..., 'PropName', PropVal, ...)
h = andrewsplot(X, ...)
```

**Description**

`andrewsplot(X)` creates an Andrews plot of the multivariate data in the matrix  $X$ . The rows of  $X$  correspond to observations, the columns to variables. Andrews plots represent each observation by a function  $f(t)$  of a continuous dummy variable  $t$  over the interval  $[0, 1]$ .  $f(t)$  is defined for the  $i$ th observation in  $X$  as

$$f(t) = X(i,1) / \sqrt{2} + X(i,2) \sin(2\pi t) + X(i,3) \cos(2\pi t) + \dots$$

`andrewsplot` treats NaN values in  $X$  as missing values and ignores the corresponding rows.

`andrewsplot(X, ..., 'Standardize', standopt)` creates an Andrews plot where *standopt* is one of the following:

- 'on' — scales each column of  $X$  to have mean 0 and standard deviation 1 before making the plot.
- 'PCA' — creates an Andrews plot from the principal component scores of  $X$ , in order of decreasing eigenvalue. (See `pca`.)
- 'PCAStd' — creates an Andrews plot using the standardized principal component scores. (See `pca`.)

`andrewsplot(X, ..., 'Quantile', alpha)` plots only the median and the  $\alpha$  and  $(1 - \alpha)$  quantiles of  $f(t)$  at each value of  $t$ . This is useful if  $X$  contains many observations.

`andrewsplot(X, ..., 'Group', group)` plots the data in different groups with different colors. Groups are defined by `group`, a numeric array containing a group index for each observation. `group` can also be a

# andrewsplot

---

categorical array, character matrix, or cell array of strings containing a group name for each observation.

`andrewsplot(X,...,'PropName',PropVal,...)` sets optional lineseries object properties to the specified values for all lineseries objects created by `andrewsplot`. (See `lineseriesproperties`.)

`h = andrewsplot(X,...)` returns a column vector of handles to the lineseries objects created by `andrewsplot`, one handle per row of `X`. If you use the `'Quantile'` input parameter, `h` contains one handle for each of the three lineseries objects created. If you use both the `'Quantile'` and the `'Group'` input parameters, `h` contains three handles for each group.

## Examples

### Create Andrews Plot to Visualize Grouped Data

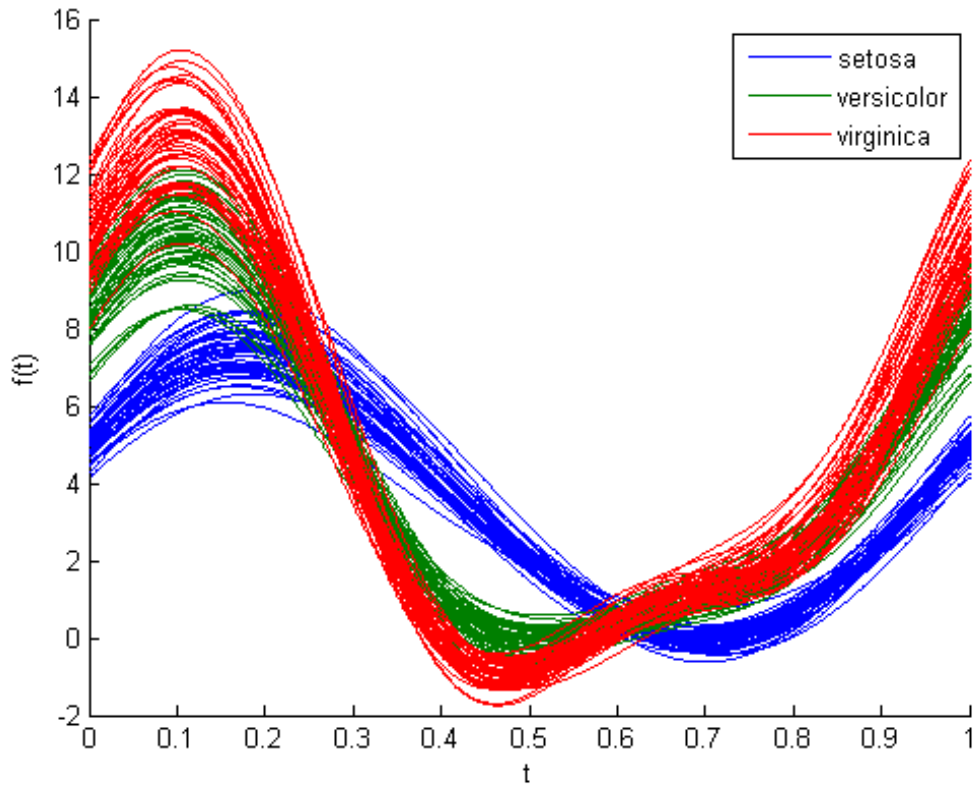
This example shows how to create an Andrews plot to visualize grouped sample data.

Load the sample data.

```
load fisheriris
```

Create an Andrews plot, grouping the sample data by species.

```
andrewsplot(meas,'group',species)
```

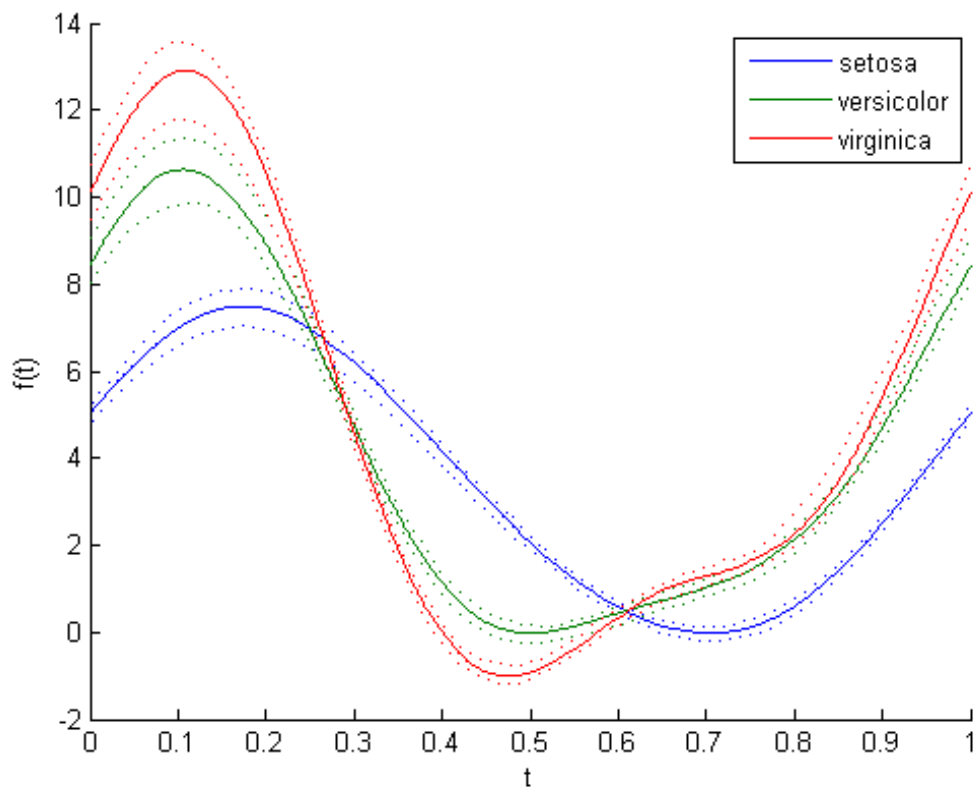


Create a second, simplified Andrews plot that only displays the median and quartiles of each group.

```
andrewsplot(meas, 'group', species, 'quantile', .25)
```

# andrewsplot

---



**See Also**      [parallelcoords](#) | [glyphplot](#)

**How To**      • “Grouping Variables” on page 2-51

## Purpose

Analysis of variance for linear model

## Syntax

```
tbl = anova mdl
tbl = anova mdl, anovatype
tbl = anova mdl, anovatype, sstype
```

## Description

`tbl = anova(mdl)` returns a dataset array with summary ANOVA statistics.

`tbl = anova(mdl, anovatype)` returns ANOVA statistics of the chosen type.

`tbl = anova(mdl, anovatype, sstype)` computes ANOVA statistics using the chosen type of sum of squares.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **anovatype**

ANOVA type:

- 'component' — `tbl` displays a 'components' ANOVA table, with sums of squares and  $F$  tests attributable to each term in the model except the constant term.
- 'summary' — `tbl` displays a summary ANOVA table with an  $F$  test for the model as a whole.
  - If there are both linear and higher-order terms, there is also an  $F$  test for the higher-order terms as a group.
  - If there are replications (multiple observations sharing the same predictor values), there is also an  $F$  test for lack-of-fit computed by decomposing the residual sum of squares into a sum of squares for the replicated observations and the remaining sum of squares.

**Default:** 'component'

## **sstype**

When anovatype is 'component', choose the sum of squares type:

- 1
- 2
- 3
- 'h'

For details, see “Sum of Squares” on page 21-58.

**Default:** 'h'

## **Output Arguments**

### **tbl**

Dataset array containing summary ANOVA statistics. `tbl` depends on `anovatype`:

- 'component':
  - Sum of squares
  - Degrees of freedom
  - Mean squares
  - $F$  statistic
  - $p$ -value
  - Formula used for model
- 'summary':
  - Total Sum of Squares
  - Model Sum of Squares
    - Linear Sum of Squares (present if model has powers or interactions)

- Nonlinear Sum of Squares (present if model has powers or interactions)
- Residual Sum of Squares
  - Lack-of-fit Sum of Squares (present if model has replicates)
  - Pure error Sum of Squares (present if model has replicates)

## Examples

### Component ANOVA Table

Create a component ANOVA table from a model of the `carsmall` data.

Load the `carsmall` data and make a model of the MPG as a function of weight and model year.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Create an ANOVA table.

```
tbl = anova(mdl)
```

```
tbl =
```

	SumSq	DF	MeanSq	F	pValue
Weight	2050.2	1	2050.2	265.11	1.9885e-28
Year	849.55	2	424.77	54.927	2.9042e-16
Weight^2	76.688	1	76.688	9.9164	0.0022303
Error	688.27	89	7.7334		

### Summary ANOVA Table

Create a summary ANOVA table from a model of the `carsmall` data.

Load the `carsmall` data and make a model of the MPG as a function of weight and model year.

# LinearModel.anova

---

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Create a summary ANOVA table.

```
tbl = anova(mdl,'summary')
```

```
tbl =
```

	SumSq	DF	MeanSq	F	pValue
Total	6005.3	93	64.573		
Model	5317	4	1329.3	171.88	5.5208e-41
. Linear	5240.3	3	1746.8	225.87	1.7302e-41
. Nonlinear	76.688	1	76.688	9.9164	0.0022303
Residual	688.27	89	7.7334		
. Lack of fit	663.77	86	7.7183	0.9451	0.62874
. Pure error	24.5	3	8.1667		

The summary ANOVA table shows tests for groups of terms. The nonlinear group consists of just the  $\text{Weight}^2$  term, so it has the same  $p$ -value as that term in “Component ANOVA Table” on page 21-31. The  $F$  statistic comparing the residual sum of squares to a “pure error” estimate from replicated observations shows no evidence of lack of fit.

## Alternatives

More complete ANOVA statistics are available in the `anova1`, `anova2`, and `anovan` functions.

## See Also

`LinearModel`

## Tutorials

- “ANOVA” on page 9-23

## How To

- “Linear Regression” on page 9-12



## Purpose

Analysis of variance for linear mixed-effects model

## Syntax

```
stats = anova(lme)
stats = anova(lme,Name,Value)
```

## Description

`stats = anova(lme)` returns the dataset array `stats` that includes the results of the  $F$ -tests for each fixed-effects term in the linear mixed-effects model `lme`.

`stats = anova(lme,Name,Value)` also returns the dataset array `stats` with additional options specified by one or more `Name,Value` pair arguments.

## Tips

- For each fixed-effects term, `anova` performs an  $F$ -test (marginal test), that all coefficients representing the fixed-effects term are 0. To perform tests for type III hypotheses, you must set the 'DummyVarCoding' name-value pair argument to 'effects' contrasts while fitting your linear mixed-effects model.

## Input Arguments

### **lme - Linear mixed-effects model**

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'DFMethod' - Method for computing approximate degrees of freedom**

'Residual' (default) | 'Satterthwaite' | 'None'

# LinearMixedModel.anova

---

Method for computing approximate degrees of freedom to use in the  $F$ -test, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'Residual'	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'Satterthwaite'	Satterthwaite approximation.
'None'	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

**Example:** 'DFMethod', 'Satterthwaite'

## Output Arguments

### **stats** - Results of $F$ -tests for fixed-effects terms

dataset array

Results of  $F$ -tests for fixed-effects terms, returned as a dataset array with the following columns.

Term	Name of the fixed effects term
Fstat	$F$ -statistic for the term
DF1	Numerator degrees of freedom for the $F$ -statistic
DF2	Denominator degrees of freedom for the $F$ -statistic
pValue	$p$ -value of the test for the term

There is one row for each fixed-effects term. Each term is a continuous variable, a grouping variable, or an interaction

between two or more continuous or grouping variables. For each fixed-effects term, `anova` performs an  $F$ -test (marginal test) to determine if all coefficients representing the fixed-effects term are 0. To perform tests for the type III hypothesis, you must use the 'effects' contrasts while fitting the linear mixed-effects model.

## Examples

### F-Tests for Fixed Effects

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift. Use the restricted maximum likelihood method and 'effects' contrasts.

'effects' contrasts indicate that the coefficients sum to 0, and `fitlme` creates two contrast-coded variables in the fixed-effects design matrix,  $X_1$  and  $X_2$ , where

# LinearMixedModel.anova

$$\text{Shift\_Evening} = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases} \quad \text{and} \quad \text{Shift\_Morning} = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}.$$

The model corresponds to

$$\text{Morning Shift: } QCDev_{im} = \beta_0 + \beta_2 \text{Shift} - \text{Morning}_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 5,$$

$$\text{Evening Shift: } QCDev_{im} = \beta_0 + \beta_1 \text{Shift} - \text{Evening}_i + b_{0m} + \varepsilon_{im},$$

$$\text{Night Shift: } QCDev_{im} = \beta_0 - \beta_1 \text{Shift} - \text{Evening}_i - \beta_2 \text{Shift} - \text{Morning}_i + b_{0m} + \varepsilon_{im},$$

where  $b \sim N(0, \sigma^2_b)$  and  $\varepsilon \sim N(0, \sigma^2)$ .

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)', ...  
'FitMethod', 'REML', 'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
QCDev ~ 1 + Shift + (1 | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.6525	0.94109	3.8812	12	0.0021
'Shift_Evening'	-0.53293	0.31206	-1.7078	12	0.11

```
'Shift_Morning'          -0.91973    0.31206    -2.9473    12    0
```

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	2.0457

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.85462	0.52357	1.395

Perform an  $F$ -test to determine if all fixed-effects coefficients are 0.

```
anova(lme)
```

```
ans =
```

```
ANOVA marginal tests: DFMethod = 'Residual'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	15.063	1	12	0.0021832
'Shift'	11.091	2	12	0.0018721

The  $p$ -value for the constant term, 0.0021832, is the same as in the coefficient table in the `lme` display. The  $p$ -value of 0.0018721 for `Shift` measures the combined significance for both coefficients representing `Shift`.

## Split-Plot Experiment

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

# LinearMixedModel.anova

---

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently. Use the 'effects' contrasts when fitting the data for the type III sum of squares.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)  
'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	60
Fixed effects coefficients	20
Random effects coefficients	18
Covariance parameters	3

```
Formula:
```

```
Yield ~ 1 + Tomato*Fertilizer + (1 | Soil) + (1 | Soil:Tomato)
```

# LinearMixedModel.anova

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
522.57	570.74	-238.29	476.57

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat
'(Intercept)'	104.6	3.3008	31
'Tomato_Cherry'	1.4	5.9353	0.235
'Tomato_Grape'	-7.7667	5.9353	-1.30
'Tomato_Heirloom'	-11.183	5.9353	-1.88
'Tomato_Plum'	30.233	5.9353	5.09
'Fertilizer_1'	-28.267	2.3475	-12.0
'Fertilizer_2'	-1.9333	2.3475	-0.823
'Fertilizer_3'	10.733	2.3475	4.57
'Tomato_Cherry:Fertilizer_1'	-0.73333	4.6951	-0.156
'Tomato_Grape:Fertilizer_1'	-7.5667	4.6951	-1.61
'Tomato_Heirloom:Fertilizer_1'	5.1833	4.6951	1.10
'Tomato_Plum:Fertilizer_1'	2.7667	4.6951	0.589
'Tomato_Cherry:Fertilizer_2'	7.6	4.6951	1.62
'Tomato_Grape:Fertilizer_2'	-1.9	4.6951	-0.404
'Tomato_Heirloom:Fertilizer_2'	5.5167	4.6951	1.17
'Tomato_Plum:Fertilizer_2'	-3.9	4.6951	-0.830
'Tomato_Cherry:Fertilizer_3'	-6.0667	4.6951	-1.29
'Tomato_Grape:Fertilizer_3'	3.7667	4.6951	0.802
'Tomato_Heirloom:Fertilizer_3'	3.1833	4.6951	0.678
'Tomato_Plum:Fertilizer_3'	1.1	4.6951	0.234

Random effects covariance parameters (95% CIs):

Group: Soil (3 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	2.5028

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	10.225

# LinearMixedModel.anova

---

```
Group: Error
      Name      Estimate    Lower    Upper
  'Res Std'    10.499      8.5389   12.908
```

Perform an analysis of variance to test for the fixed-effects.

```
anova(lme)
```

```
ans =
```

```
ANOVA marginal tests: DFMethod = 'Residual'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	1004.2	1	40	5.9086e-30
'Tomato'	7.1663	4	40	0.00018935
'Fertilizer'	58.833	3	40	1.0024e-14
'Tomato:Fertilizer'	1.4182	12	40	0.19804

The  $p$ -value for the constant term,  $5.9086e-30$ , is the same as in the coefficient table in the `lme` display. The  $p$ -values of  $0.00018935$ ,  $1.0024e-14$ , and  $0.19804$  for `Tomato`, `Fertilizer`, and `Tomato:Fertilizer` represent the combined significance for all tomato coefficients, fertilizer coefficients, and coefficients representing the interaction between the tomato and fertilizer, respectively. The  $p$ -value of  $0.19804$  indicates that the interaction between tomato and fertilizer is not significant.

## Satterthwaite Approximation for Degrees of Freedom

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```



weight contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define Subject and Program as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);  
ds.Subject = nominal(ds.Subject);  
ds.Program = nominal(ds.Program);
```

Fit the model using the 'effects' contrasts.

```
lme = fitlme(ds,'y ~ InitialWeight + Program*Week + (Week|Subject)', 'D'  
  
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

Formula:

```
y ~ 1 + InitialWeight + Program*Week + (1 + Week | Subject)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.77122	0.24309	3.1725	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_A'	-0.11017	0.080377	-1.3707	111

# LinearMixedModel.anova

'Program_B'	0.25061	0.08045	3.1151	111
'Program_C'	-0.14344	0.080475	-1.7824	111
'Week'	0.19881	0.033727	5.8946	111
'Program_A:Week'	-0.025607	0.058417	-0.43835	111
'Program_B:Week'	0.013164	0.058417	0.22535	111
'Program_C:Week'	0.0049357	0.058417	0.084492	111

Random effects covariance parameters (95% CIs):  
Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	
'(Intercept)'	'(Intercept)'	'std'	0.18407	C
'Week'	'(Intercept)'	'corr'	0.66841	C
'Week'	'Week'	'std'	0.15033	C

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The  $p$ -values 0.022863 and 4.1099e-08 indicate significant effects of the initial weights of the subjects and the time factor in the amount of weight lost. The weight loss of subjects who are in program B is significantly different relative to the weight loss of subjects that are in program A. The lower and upper limits of the covariance parameters for the random effects do not include zero, thus they are significant.

Perform an F-test that all fixed-effects coefficients are zero.

```
anova(lme)
```

```
ans =
```

ANOVA marginal tests: DFMethod = 'Residual'

Term	FStat	DF1	DF2	pValue
'(Intercept)'	10.065	1	111	0.0019549
'InitialWeight'	5.326	1	111	0.022863
'Program'	3.6798	3	111	0.014286

'Week'	34.747	1	111	4.1099e-08
'Program:Week'	0.066648	3	111	0.97748

The  $p$ -values for the constant term, initial weight, and week are the same as in the coefficient table in the previous `lme` output display. The  $p$ -value of 0.014286 for `Program` represents the combined significance for all program coefficients. Similarly, the  $p$ -value for the interaction between program and week (`Program:Week`) measures the combined significance for all coefficients representing this interaction.

Now, use the Satterthwaite method to compute the degrees of freedom.

```
anova(lme, 'DFMethod', 'Satterthwaite')
```

```
ans =
```

```
ANOVA marginal tests: DFMethod = 'Satterthwaite'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	10.065	1	20.445	0.004695
'InitialWeight'	5.326	1	20	0.031827
'Program'	3.6798	3	19.14	0.030233
'Week'	34.747	1	20	9.1347e-06
'Program:Week'	0.066648	3	20	0.97697

The Satterthwaite method produces smaller denominator degrees of freedom and slightly larger  $p$ -values.

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

# anova1

---

**Purpose** One-way analysis of variance

**Syntax**

```
p = anova1(X)
p = anova1(X,group)
p = anova1(X,group,displayopt)
[p,table] = anova1(...)
[p,table,stats] = anova1(...)
```

**Description** `p = anova1(X)` performs balanced one-way ANOVA for comparing the means of two or more columns of data in the matrix `X`, where each column represents an independent sample containing mutually independent observations. The function returns the  $p$ -value under the null hypothesis that all samples in `X` are drawn from populations with the same mean.

If `p` is near zero, it casts doubt on the null hypothesis and suggests that at least one sample mean is significantly different than the other sample means. Common significance levels are 0.05 or 0.01.

The `anova1` function displays two figures, the standard ANOVA table and a box plot of the columns of `X`.

The standard ANOVA table divides the variability of the data into two parts:

- Variability due to the differences among the column means (variability *between* groups)
- Variability due to the differences between the data in each column and the column mean (variability *within* groups)

The standard ANOVA table has six columns:

- 1 The source of the variability.
- 2 The sum of squares (SS) due to each source.
- 3 The degrees of freedom (df) associated with each source.
- 4 The mean squares (MS) for each source, which is the ratio  $SS/df$ .

5 The  $F$ -statistic, which is the ratio of the mean squares.

6 The  $p$ -value, which is derived from the cdf of  $F$ .

The box plot of the columns of  $X$  suggests the size of the  $F$ -statistic and the  $p$ -value. Large differences in the center lines of the boxes correspond to large values of  $F$  and correspondingly small values of  $p$ .

anova1 treats NaN values as missing, and disregards them.

`p = anova1(X,group)` performs ANOVA by group. For more information on grouping variables, see “Grouping Variables” on page 2-51.

If  $X$  is a matrix, anova1 treats each column as a separate group, and evaluates whether the population means of the columns are equal. This form of anova1 is appropriate when each group has the same number of elements (balanced ANOVA). `group` can be a character array or a cell array of strings, with one row per column of  $X$ , containing group names. Enter an empty array (`[]`) or omit this argument if you do not want to specify group names.

If  $X$  is a vector, `group` must be a categorical variable, vector, string array, or cell array of strings with one name for each element of  $X$ .  $X$  values corresponding to the same value of `group` are placed in the same group. This form of anova1 is appropriate when groups have different numbers of elements (unbalanced ANOVA).

If `group` contains empty or NaN-valued cells or strings, the corresponding observations in  $X$  are disregarded.

`p = anova1(X,group,displayopt)` enables the ANOVA table and box plot displays when `displayopt` is 'on' (default) and suppresses the displays when `displayopt` is 'off'. Notches in the boxplot provide a test of group medians (see `boxplot`) different from the  $F$  test for means in the ANOVA table.

`[p,table] = anova1(...)` returns the ANOVA table (including column and row labels) in the cell array `table`. Copy a text version of the ANOVA table to the clipboard using the Copy Text item on the **Edit** menu.

`[p,table,stats] = anova1(...)` returns a structure `stats` used to perform a follow-up multiple comparison test. `anova1` evaluates the hypothesis that the samples all have the same mean against the alternative that the means are not all the same. Sometimes it is preferable to perform a test to determine which pairs of means are significantly different, and which are not. Use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

## Assumptions

The ANOVA test makes the following assumptions about the data in `X`:

- All sample populations are normally distributed.
- All sample populations have equal variance.
- All observations are mutually independent.

The ANOVA test is known to be robust with respect to modest violations of the first two assumptions.

## Examples

### Example 1

Create `X` with columns that are constants plus random normal disturbances with mean zero and standard deviation one:

```
X = meshgrid(1:5)
```

```
X =  
  1  2  3  4  5  
  1  2  3  4  5  
  1  2  3  4  5  
  1  2  3  4  5  
  1  2  3  4  5
```

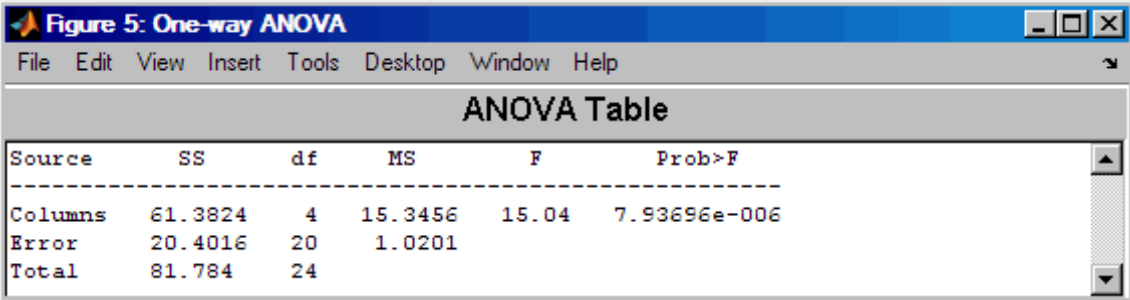
```
X = X + normrnd(0,1,5,5)
```

```
X =  
  1.3550    2.0662    2.4688    5.9447    5.4897  
  2.0693    1.7611    1.4864    4.8826    6.3222  
  2.1919    0.7276    3.1905    4.8768    4.6841  
  2.7620    1.8179    3.9506    4.4678    4.9291  
 -0.3626    1.1685    3.5742    2.1945    5.9465
```

Perform one-way ANOVA:

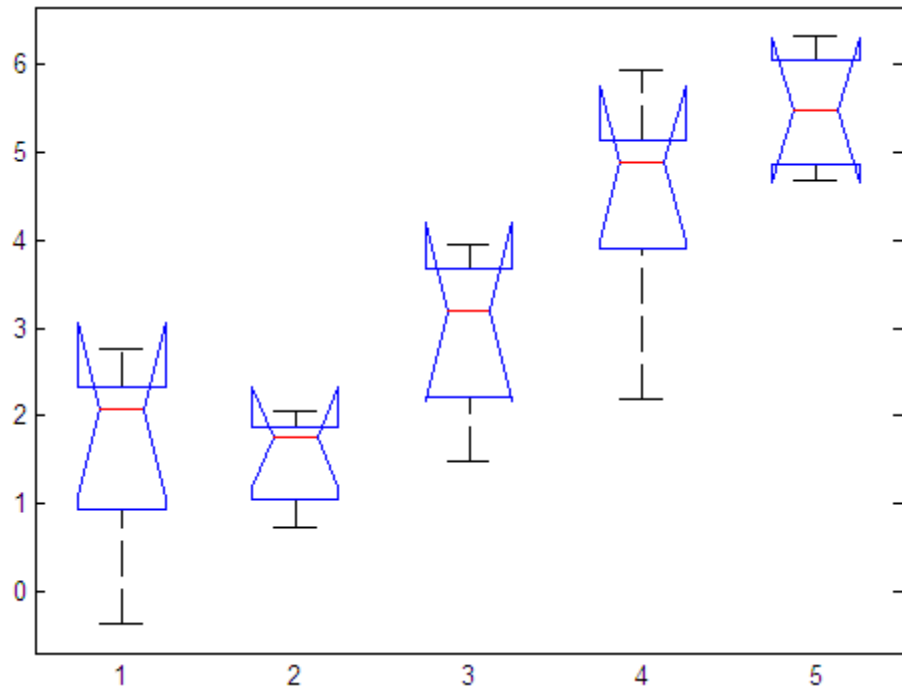
```
p = anova1(X)
```

```
p =  
7.9370e-006
```



The screenshot shows a software window titled "Figure 5: One-way ANOVA" with a menu bar containing "File", "Edit", "View", "Insert", "Tools", "Desktop", "Window", and "Help". The main content is an "ANOVA Table" with the following data:

Source	SS	df	MS	F	Prob>F
Columns	61.3824	4	15.3456	15.04	7.93696e-006
Error	20.4016	20	1.0201		
Total	81.784	24			



The very small  $p$  value indicates that differences between column means are highly significant. The probability of this outcome under the null hypothesis (that samples drawn from the same population would have means differing by the amounts seen in  $X$ ) is equal to the  $p$ -value.

## Example 2

The following example is from a study of the strength of structural beams in Hogg. The vector `strength` measures deflections of beams in thousandths of an inch under 3,000 pounds of force. The vector `alloy` identifies each beam as steel ('st'), alloy 1 ('a11'), or alloy 2 ('a12'). (Although `alloy` is sorted in this example, grouping variables do not need to be sorted.) The null hypothesis is that steel beams are equal in strength to beams made of the two more expensive alloys.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...
```



```

78 75 76 77 79 79 77 78 82 79];

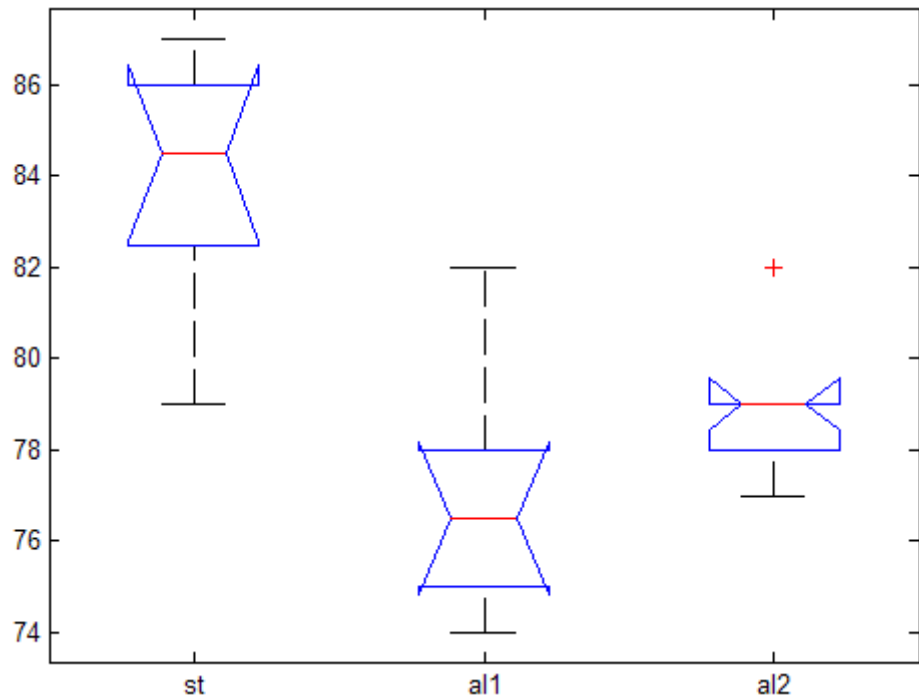
alloy = {'st','st','st','st','st','st','st','st',...
        'al1','al1','al1','al1','al1','al1',...
        'al2','al2','al2','al2','al2','al2'};

p = anova1(strength,alloy)
p =
1.5264e-004

```

Source	SS	df	MS	F	Prob>F
Columns	184.8	2	92.4	15.4	0.0002
Error	102	17	6		
Total	286.8	19			

# anova1



The  $p$ -value suggests rejection of the null hypothesis. The box plot shows that steel beams deflect more than beams made of the more expensive alloys.

## References

[1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

## See Also

[anova2](#) | [anovan](#) | [boxplot](#) | [manova1](#) | [multcompare](#)

## How To

- “Grouping Variables” on page 2-51

**Purpose** Two-way analysis of variance

**Syntax**

```
p = anova2(X, reps)
p = anova2(X, reps, displayopt)
[p, table] = anova2(...)
[p, table, stats] = anova2(...)
```

**Description** `p = anova2(X, reps)` performs a balanced two-way ANOVA for comparing the means of two or more columns and two or more rows of the observations in  $X$ . The data in different columns represent changes in factor  $A$ . The data in different rows represent changes in factor  $B$ . If there is more than one observation for each combination of factors, input `reps` indicates the number of replicates in each position, which must be constant. (For unbalanced designs, use `anovan`.)

The matrix below shows the format for a set-up where column factor  $A$  has two levels, row factor  $B$  has three levels, and there are two replications (`reps = 2`). The subscripts indicate row, column, and replicate, respectively.

$$\begin{array}{cc}
 A = 1 & A = 2 \\
 \left[ \begin{array}{cc}
 x_{111} & x_{121} \\
 x_{112} & x_{122} \\
 x_{211} & x_{221} \\
 x_{212} & x_{222} \\
 x_{311} & x_{321} \\
 x_{312} & x_{322}
 \end{array} \right] \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} B = 1 \\ \\ B = 2 \\ \\ B = 3 \end{array}
 \end{array}$$

When `reps` is 1 (default), `anova2` returns two  $p$ -values in vector `p`:

- 1 The  $p$  value for the null hypothesis,  $H_{0A}$ , that all samples from factor  $A$  (i.e., all column-samples in  $X$ ) are drawn from the same population
- 2 The  $p$  value for the null hypothesis,  $H_{0B}$ , that all samples from factor  $B$  (i.e., all row-samples in  $X$ ) are drawn from the same population

When `reps` is greater than 1, `anova2` returns a third  $p$  value in vector `p`:

- 3 The  $p$  value for the null hypothesis,  $H_{0AB}$ , that the effects due to factors A and B are *additive* (i.e., that there is no interaction between factors A and B)

If any  $p$  value is near zero, this casts doubt on the associated null hypothesis. A sufficiently small  $p$  value for  $H_{0A}$  suggests that at least one column-sample mean is significantly different than the other column-sample means; i.e., there is a main effect due to factor A. A sufficiently small  $p$  value for  $H_{0B}$  suggests that at least one row-sample mean is significantly different than the other row-sample means; i.e., there is a main effect due to factor B. A sufficiently small  $p$  value for  $H_{0AB}$  suggests that there is an interaction between factors A and B. The choice of a limit for the  $p$  value to determine whether a result is “statistically significant” is left to the researcher. It is common to declare a result significant if the  $p$  value is less than 0.05 or 0.01.

`anova2` also displays a figure showing the standard ANOVA table, which divides the variability of the data in  $X$  into three or four parts depending on the value of `reps`:

- The variability due to the differences among the column means
- The variability due to the differences among the row means
- The variability due to the interaction between rows and columns (if `reps` is greater than its default value of one)
- The remaining variability not explained by any systematic source

The ANOVA table has five columns:

- The first shows the source of the variability.
- The second shows the Sum of Squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the Mean Squares (MS), which is the ratio SS/df.

- The fifth shows the  $F$  statistics, which is the ratio of the mean squares.

`p = anova2(X, reps, displayopt)` enables the ANOVA table display when `displayopt` is 'on' (default) and suppresses the display when `displayopt` is 'off'.

`[p, table] = anova2(...)` returns the ANOVA table (including column and row labels) in cell array `table`. (Copy a text version of the ANOVA table to the clipboard by using the Copy Text item on the **Edit** menu.)

`[p, table, stats] = anova2(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test.

The `anova2` test evaluates the hypothesis that the row, column, and interaction effects are all the same, against the alternative that they are not all the same. Sometimes it is preferable to perform a test to determine *which pairs* of effects are significantly different, and which are not. Use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

## Examples

The data below come from a study of popcorn brands and popper type (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper type (Oil and Air.) The study popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

```
load popcorn
```

```
popcorn
popcorn =
    5.5000    4.5000    3.5000
    5.5000    4.5000    4.0000
    6.0000    4.0000    3.0000
    6.5000    5.0000    4.0000
    7.0000    5.5000    5.0000
    7.0000    5.0000    4.5000
```

## anova2

```
p = anova2(popcorn,3)
p =
    0.0000    0.0001    0.7462
```

Source	SS	df	MS	F	Prob>F
Columns	15.75	2	7.875	56.7	0
Rows	4.5	1	4.5	32.4	0.0001
Interaction	0.0833	2	0.04167	0.3	0.7462
Error	1.6667	12	0.13889		
Total	22	17			

The vector `p` shows the  $p$ -values for the three brands of popcorn, 0.0000, the two popper types, 0.0001, and the interaction between brand and popper type, 0.7462. These values indicate that both popcorn brand and popper type affect the yield of popcorn, but there is no evidence of a synergistic (interaction) effect of the two.

The conclusion is that you can get the greatest yield using the Gourmet brand and an Air popper (the three values `popcorn(4:6,1)`).

### References

[1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

### See Also

`anova1` | `anovan`

**Purpose** *N*-way analysis of variance

**Syntax**

```
p = anovan(y,group)
p = anovan(y,group,param,val)
[p,table] = anovan(y,group,param,val)
[p,table,stats] = anovan(y,group,param,val)
[p,table,stats,terms] = anovan(y,group,param,val)
```

**Description** `p = anovan(y,group)` performs multiway (n-way) analysis of variance (ANOVA) for testing the effects of multiple factors on the mean of the vector `y`. This test compares the variance explained by factors to the left over variance that cannot be explained. The factors and factor levels of the observations in `y` are assigned by the cell array `group`. Each of the cells in the cell array `group` contains a list of factor levels identifying the observations in `y` with respect to one of the factors. The list within each cell can be a categorical array, numeric vector, character matrix, or single-column cell array of strings, and must have the same number of elements as `y`. The fitted ANOVA model includes the main effects of each grouping variable. All grouping variables are treated as fixed effects by default. The result `p` is a vector of *p*-values, one per term. For an example, see “Example of Three-Way ANOVA” on page 21-59.

`p = anovan(y,group,param,val)` specifies one or more of the parameter name/value pairs described in the following table.

Parameter	Value
'alpha'	A number between 0 and 1 requesting 100(1 – <i>alpha</i> )% confidence bounds (default 0.05 for 95% confidence)
'continuous'	A vector of indices indicating which grouping variables should be treated as continuous predictors rather than as categorical predictors.
'display'	'on' displays an ANOVA table (the default) 'off' omits the display

Parameter	Value
'model'	The type of model used. See “Model Type” on page 21-57 for a description of this parameter.
'nested'	A matrix M of 0's and 1's specifying the nesting relationships among the grouping variables. M(i,j) is 1 if variable i is nested in variable j.
'random'	A vector of indices indicating which grouping variables are random effects (all are fixed by default). See “ANOVA with Random Effects” on page 8-19 for an example of how to use 'random'.
'sstype'	1, 2, 3 (default), or h specifies the type of sum of squares. See “Sum of Squares” on page 21-58 for a description of this parameter.
'varnames'	A character matrix or a cell array of strings specifying names of grouping variables, one per grouping variable. When you do not specify 'varnames', the default labels 'X1', 'X2', 'X3', ..., 'XN' are used. See “ANOVA with Random Effects” on page 8-19 for an example of how to use 'varnames'.

`[p,table] = anovan(y,group,param,val)` returns the ANOVA table (including factor labels) in cell array `table`. (Copy a text version of the ANOVA table to the clipboard by using the Copy Text item on the **Edit** menu.)

`[p,table,stats] = anovan(y,group,param,val)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test with the `multcompare` function. See “The Stats Structure” on page 21-62The Stats Structure for more information.

`[p,table,stats,terms] = anovan(y,group,param,val)` returns the main and interaction terms used in the ANOVA computations. The terms are encoded in the output matrix `terms` using the same format described above for input 'model'. When you specify 'model' itself in this matrix format, the matrix returned in `terms` is identical.



## Model Type

This section explains how to use the argument 'model' with the syntax:

```
[...] = anovan(y,group,'model',modeltype)
```

The argument *modeltype*, which specifies the type of model the function uses, can be any one of the following:

- 'linear' — The default 'linear' model computes only the *p*-values for the null hypotheses on the *N* main effects.
- 'interaction' — The 'interaction' model computes the *p*-values

for null hypotheses on the *N* main effects and the  $\binom{N}{2}$  two-factor interactions.

- 'full' — The 'full' model computes the *p*-values for null hypotheses on the *N* main effects and interactions at all levels.
- An integer — For an integer value of *modeltype*, *k* ( $k \leq N$ ), *anovan* computes all interaction levels through the *k*th level. For example, the value 3 means main effects plus two- and three-factor interactions. The values *k* = 1 and *k* = 2 are equivalent to the 'linear' and 'interaction' specifications, respectively, while the value *k* = *N* is equivalent to the 'full' specification.
- A matrix of term definitions having the same form as the input to the *x2fx* function. All entries must be 0 or 1 (no higher powers).

For more precise control over the main and interaction terms that *anovan* computes, *modeltype* can specify a matrix containing one row for each main or interaction term to include in the ANOVA model. Each row defines one term using a vector of *N* zeros and ones. The table below illustrates the coding for a 3-factor ANOVA.

Matrix Row	ANOVA Term
[1 0 0]	Main term A
[0 1 0]	Main term B

<b>Matrix Row</b>	<b>ANOVA Term</b>
[0 0 1]	Main term C
[1 1 0]	Interaction term AB
[1 0 1]	Interaction term AC
[0 1 1]	Interaction term BC
[1 1 1]	Interaction term ABC

For example, if *modeltype* is the matrix [0 1 0;0 0 1;0 1 1], the output vector *p* contains the *p*-values for the null hypotheses on the main effects B and C and the interaction effect BC, in that order. A simple way to generate the *modeltype* matrix is to modify the *terms* output, which codes the terms in the current model using the format described above. If *anovan* returns [0 1 0;0 0 1;0 1 1] for *terms*, for example, and there is no significant result for interaction BC, you can recompute the ANOVA on just the main effects B and C by specifying [0 1 0;0 0 1] for *modeltype*.

### Sum of Squares

This section explains how to use the argument '*sstype*' with the syntax:

```
[...] = anovan(y,group, 'sstype', type)
```

This syntax computes the ANOVA using the type of sum of squares specified by *type*, which can be 1, 2, 3, or h. While the numbers 1 – 3 designate Type 1, Type 2, or Type 3 sum of squares, respectively, h represents a hierarchical model similar to type 2, but with continuous as well as categorical factors used to determine the hierarchy of terms. The default value is 3. For a model containing main effects but no interactions, the value of *type* only influences computations on unbalanced data.

The sum of squares for any term is determined by comparing two models. The Type 1 sum of squares for a term is the reduction in residual sum of squares obtained by adding that term to a fit that already includes the terms listed before it. The Type 2 sum of squares is

the reduction in residual sum of squares obtained by adding that term to a model consisting of all other terms that do not contain the term in question. The Type 3 sum of squares is the reduction in residual sum of squares obtained by adding that term to a model containing all other terms, but with their effects constrained to obey the usual “sigma restrictions” that make models estimable.

Suppose you are fitting a model with two factors and their interaction, and that the terms appear in the order  $A, B, AB$ . Let  $R(\cdot)$  represent the residual sum of squares for a model, so for example  $R(A, B, AB)$  is the residual sum of squares fitting the whole model,  $R(A)$  is the residual sum of squares fitting just the main effect of  $A$ , and  $R(1)$  is the residual sum of squares fitting just the mean. The three types of sums of squares are as follows:

Term	Type 1 Sum of Squares	Type 2 Sum of Squares	Type 3 Sum of Squares
$A$	$R(1) - R(A)$	$R(B) - R(A, B)$	$R(B, AB) - R(A, B, AB)$
$B$	$R(A) - R(A, B)$	$R(A) - R(A, B)$	$R(A, AB) - R(A, B, AB)$
$AB$	$R(A, B) - R(A, B, AB)$	$R(A, B) - R(A, B, AB)$	$R(A, B) - R(A, B, AB)$

The models for Type 3 sum of squares have sigma restrictions imposed. This means, for example, that in fitting  $R(B, AB)$ , the array of  $AB$  effects is constrained to sum to 0 over  $A$  for each value of  $B$ , and over  $B$  for each value of  $A$ .

### Example of Three-Way ANOVA

As an example of three-way ANOVA, consider the vector  $y$  and group inputs below.

```

y = [52.7 57.5 45.9 44.5 53.0 57.0 45.9 44.0]';
g1 = [1 2 1 2 1 2 1 2];
g2 = {'hi'; 'hi'; 'lo'; 'lo'; 'hi'; 'hi'; 'lo'; 'lo'};
g3 = {'may'; 'may'; 'may'; 'may'; 'june'; 'june'; 'june'; 'june'};

```

This defines a three-way ANOVA with two levels of each factor. Every observation in  $y$  is identified by a combination of factor levels. If the factors are A, B, and C, then observation  $y(1)$  is associated with

- Level 1 of factor A
- Level 'hi' of factor B
- Level 'may' of factor C

Similarly, observation  $y(6)$  is associated with

- Level 2 of factor A
- Level 'hi' of factor B
- Level 'june' of factor C

To compute the ANOVA, enter

```
p = anovan(y, {g1 g2 g3})
p =
    0.4174
    0.0028
    0.9140
```

Output vector  $p$  contains  $p$ -values for the null hypotheses on the  $N$  main effects. Element  $p(1)$  contains the  $p$  value for the null hypotheses,  $H_{0A}$ , that samples at all levels of factor A are drawn from the same population; element  $p(2)$  contains the  $p$  value for the null hypotheses,  $H_{0B}$ , that samples at all levels of factor B are drawn from the same population; and so on.

If any  $p$  value is near zero, this casts doubt on the associated null hypothesis. For example, a sufficiently small  $p$  value for  $H_{0A}$  suggests that at least one A-sample mean is significantly different from the other A-sample means; that is, there is a main effect due to factor A. You need to choose a bound for the  $p$  value to determine whether a result is statistically significant. It is common to declare a result significant if the  $p$  value is less than 0.05 or 0.01.

anovan also displays a figure showing the standard ANOVA table, which by default divides the variability of the data in  $x$  into

- The variability due to differences between the levels of each factor accounted for in the model (one row for each factor)
- The remaining variability not explained by any systematic source

The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the sum of squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the mean squares (MS), which is the ratio  $SS/df$ .
- The fifth shows the  $F$  statistics, which are the ratios of the mean squares.
- The sixth shows the  $p$ -values for the  $F$  statistics.

The table is shown in the following figure:

Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
X1	3.781	1	3.781	0.82	0.4174
X2	199.001	1	199.001	42.95	0.0028
X3	0.061	1	0.061	0.01	0.914
Error	18.535	4	4.634		
Total	221.379	7			

Constrained (Type III) sums of squares.

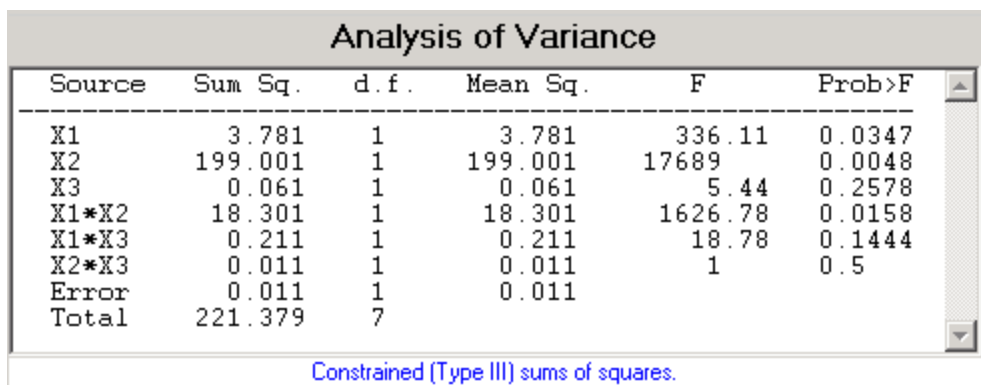
### Two-Factor Interactions

By default, anovan computes  $p$ -values just for the three main effects. To also compute  $p$ -values for the two-factor interactions,  $X1*X2$ ,  $X1*X3$ , and  $X2*X3$ , add the name/value pair 'model', 'interaction' as input arguments.

## anovan

```
p = anovan(y,{g1 g2 g3},'model','interaction')
p =
    0.0347
    0.0048
    0.2578
    0.0158
    0.1444
    0.5000
```

The first three entries of  $p$  are the  $p$ -values for the main effects. The last three entries are the  $p$ -values for the two-factor interactions. You can determine the order in which the two-factor interactions occur from the ANOVAN table shown in the following figure.



Source	Sum Sq.	d. f.	Mean Sq.	F	Prob>F
X1	3.781	1	3.781	336.11	0.0347
X2	199.001	1	199.001	17689	0.0048
X3	0.061	1	0.061	5.44	0.2578
X1*X2	18.301	1	18.301	1626.78	0.0158
X1*X3	0.211	1	0.211	18.78	0.1444
X2*X3	0.011	1	0.011	1	0.5
Error	0.011	1	0.011		
Total	221.379	7			

Constrained (Type III) sums of squares.

### The Stats Structure

The `anovan` test evaluates the hypothesis that the different levels of a factor (or more generally, a term) have the same effect, against the alternative that they do not all have the same effect. Sometimes it is preferable to perform a test to determine which pairs of levels are significantly different, and which are not. Use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

The `stats` structure contains the fields listed below, in addition to a number of other fields required for doing multiple comparisons using the `multcompare` function:

Field	Description
<code>coeffs</code>	Estimated coefficients
<code>coeffnames</code>	Name of term for each coefficient
<code>vars</code>	Matrix of grouping variable values for each term
<code>resid</code>	Residuals from the fitted model

The `stats` structure also contains the following fields if there are random effects:

Field	Description
<code>ems</code>	Expected mean squares
<code>denom</code>	Denominator definition
<code>rtnames</code>	Names of random terms
<code>varest</code>	Variance component estimates (one per random term)
<code>varci</code>	Confidence intervals for variance components

## Examples

“Example: Two-Way ANOVA” on page 8-10 shows how to use `anova2` to analyze the effects of two factors on a response in a balanced design. For a design that is not balanced, use `anovan` instead.

The data in `carbig.mat` gives measurements on 406 cars. Use `anovan` to study how the mileage depends on where and when the cars were made:

```
load carbig

p = anovan(MPG,{org when},'model',2,'sstype',3,...
           'varnames',{'Origin';'Mfg date'})
p =
```

0  
0  
0.3059

Source	Sum Sq.	d.f.	Mean Sq.	F	Prob>F
Origin	5727.2	2	2863.58	115.09	0
Mfg date	4710.3	2	2355.15	94.65	0
Origin*Mfg date	120.5	4	30.12	1.21	0.3059
Error	9679.1	389	24.88		
Total	24252.6	397			

Constrained (Type III) sums of squares.

The  $p$  value for the interaction term is not small, indicating little evidence that the effect of the year or manufacture (when) depends on where the car was made (org). The linear effects of those two factors, however, are significant.

## References

[1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.

## See Also

anova1 | anova2 | multcompare



**Purpose**

Ansari-Bradley test

**Syntax**

```
h = ansaribradley(x,y)
h = ansaribradley(x,y,Name,Value)
[h,p] = ansaribradley( ___ )
[h,p,stats] = ansaribradley( ___ )
```

**Description**

`h = ansaribradley(x,y)` returns a test decision for the null hypothesis that the data in vectors `x` and `y` comes from the same distribution, using the Ansari-Bradley test. The alternative hypothesis is that the data in `x` and `y` comes from distributions with the same median and shape but different dispersions (e.g., variances). The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.

`h = ansaribradley(x,y,Name,Value)` returns a test decision for the Ansari-Bradley test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level, conduct a one-sided test, or use a normal approximation to calculate the value of the test statistic.

`[h,p] = ansaribradley( ___ )` also returns the *p*-value, `p`, of the test, using any of the input arguments in the previous syntaxes.

`[h,p,stats] = ansaribradley( ___ )` also returns the structure `stats` containing information about the test statistic.

**Input Arguments****x - Sample data**

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array.

- If `x` and `y` are specified as vectors, they do not need to be the same length.

- If `x` and `y` are specified as matrices, they must have the same number of columns. `ansaribradley` performs separate tests along each column and returns a vector of results.
- If `x` and `y` are specified as multidimensional arrays, `ansaribradley` works along the first nonsingleton dimension. `x` and `y` must have the same size along all remaining dimensions.

## Data Types

single | double

## **y** - Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array.

- If `x` and `y` are specified as vectors, they do not need to be the same length.
- If `x` and `y` are specified as matrices, they must have the same number of columns. `ansaribradley` performs separate tests along each column and returns a vector of results.
- If `x` and `y` are specified as multidimensional arrays, `ansaribradley` works along the first nonsingleton dimension. `x` and `y` must have the same size along all remaining dimensions.

## Data Types

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'Tail', 'right', 'Alpha', 0.01` specifies a right-tailed hypothesis test at the 1% significance level.

## 'Alpha' - Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

### **Data Types**

single | double

### **'Dim' - Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim',1 tests the column means, while 'Dim',2 tests the row means.

**Example:** 'Dim',2

### **Data Types**

single | double

### **'Tail' - Type of alternative hypothesis**

'both' (default) | 'left' | 'right'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

- |         |   |
|---------|---|
| 'both'  | Test the alternative hypothesis that the dispersion parameters of x and y are not equal.      |
| 'right' | Test the alternative hypothesis that the dispersion parameter of x is greater than that of y. |
| 'left'  | Test the alternative hypothesis that the dispersion parameter of x is less than that of y.    |

**Example:** 'Tail','right'

### **'Method' - Computation method**

'exact' | 'approximate'

Computation method for the test statistic, specified as the comma-separated pair consisting of 'Method' and one of the following.

'exact' Compute  $p$  using an exact calculation of the distribution of the test statistic  $W$ . This is the default if  $n$ , the total number of rows in  $x$  and  $y$ , is 25 or less. Note that  $n$  is computed before any NaN values (representing missing data) are removed.

'approximate' Compute  $p$  using a normal approximation for the statistic  $W^*$ . This is the default if  $n$ , the total number of rows in  $x$  and  $y$ , is greater than 25.

**Example:** 'Method', 'exact'

## Output Arguments

### **h** - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** - **p-value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **stats** - Test statistics

structure

Test statistics for the Ansari-Bradley test, returned as a structure containing:

- `W` — Value of the test statistic, which is the sum of the Ansari-Bradley ranks for the `x` sample.
- `Wstar` — Approximate normal statistic  $W^*$ .

## Examples

### Test for Equal Variances

Load the sample data. Create data vectors of miles per gallon (MPG) measurements for the model years 1982 and 1976.

```
load carsmall;
x = MPG(Model_Year==82);
y = MPG(Model_Year==76);
```

Test the null hypothesis that the miles per gallon measured in cars from 1982 and 1976 have equal variances.

```
[h,p,stats] = ansaribradley(x,y)
```

```
h =
    0

p =
    0.8426

stats =
    W: 526.9000
   Wstar: 0.1986
```

The returned value of `h = 0` indicates that `ansaribradley` does not reject the null hypothesis at the default 5% significance level.

### One-Sided Hypothesis Test

Load the sample data. Create data vectors of miles per gallon (MPG) measurements for the model years 1982 and 1976.

```
load carsmall;
x = MPG(Model_Year==82);
y = MPG(Model_Year==76);
```

Test the null hypothesis that the miles per gallon measured in cars from 1982 and 1976 have equal variances, against the alternative hypothesis that the variance of cars from 1982 is greater than that of cars from 1976.

```
[h,p,stats] = ansaribradley(x,y,'Tail','right')
```

```
h =
    0
```

```
p =
    0.5787
```

```
stats =
    W: 526.9000
    Wstar: 0.1986
```

The returned value of  $h = 0$  indicates that `ansaribradley` does not reject the null hypothesis that the variance in miles per gallon is the same for the two model years, when the alternative is that the variance of cars from 1982 is greater than that of cars from 1976.

## Definitions

### Ansari-Bradley Test

The Ansari-Bradley test is a nonparametric alternative to the two-sample  $F$ -test of equal variances. It does not require the assumption that  $x$  and  $y$  come from normal distributions. The dispersion of a distribution is generally measured by its variance or standard deviation, but the Ansari-Bradley test can be used with samples from distributions that do not have finite variances.

This test requires that the samples have equal medians. Under that assumption, and if the distributions of the samples are continuous and identical, the test is independent of the distributions. If the samples

do not have the same medians, the results can be misleading. In that case, Ansari and Bradley recommend subtracting the median, but then the distribution of the resulting test under the null hypothesis is no longer independent of the common distribution of  $x$  and  $y$ . If you want to perform the tests with medians subtracted, you should subtract the medians from  $x$  and  $y$  before calling `ansaribradley`.

### **Multidimensional Array**

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### **First Nonsingleton Dimension**

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

### **See Also**

`vartest2` | `vartestn` | `ttest2`

**Purpose** Interactive analysis of covariance

**Syntax**

```
aoctool(x,y,group)
aoctool(x,y,group,alpha)
aoctool(x,y,group,alpha,xname,yname,gname)
aoctool(x,y,group,alpha,xname,yname,gname,displayopt)
aoctool(x,y,group,alpha,xname,yname,gname,displayopt,model)
h = aoctool(...)
[h,atab,ctab] = aoctool(...)
[h,atab,ctab,stats] = aoctool(...)
```

**Description** `aoctool(x,y,group)` fits a separate line to the column vectors, `x` and `y`, for each group defined by the values in the array `group`. `group` may be a categorical variable, vector, character array, or cell array of strings. These types of models are known as one-way analysis of covariance (ANOCOVA) models. The output consists of three figures:

- An interactive graph of the data and prediction curves
- An ANOVA table
- A table of parameter estimates

You can use the figures to change models and to test different parts of the model. More information about interactive use of the `aoctool` function appears in “Analysis of Covariance Tool” on page 8-27.

`aoctool(x,y,group,alpha)` determines the confidence levels of the prediction intervals. The confidence level is  $100(1-\alpha)\%$ . The default value of `alpha` is 0.05.

`aoctool(x,y,group,alpha,xname,yname,gname)` specifies the name to use for the `x`, `y`, and `g` variables in the graph and tables. If you enter simple variable names for the `x`, `y`, and `g` arguments, the `aoctool` function uses those names. If you enter an expression for one of these arguments, you can specify a name to use in place of that expression by supplying these arguments. For example, if you enter `m(:,2)` as the `x` argument, you might choose to enter `'Col 2'` as the `xname` argument.



`aocool(x,y,group,alpha,xname,yname,gname,displayopt)` enables the graph and table displays when `displayopt` is 'on' (default) and suppresses those displays when `displayopt` is 'off'.

`aocool(x,y,group,alpha,xname,yname,gname,displayopt,model)` specifies the initial model to fit. The value of `model` can be any of the following:

- 'same mean' — Fit a single mean, ignoring grouping
- 'separate means' — Fit a separate mean to each group
- 'same line' — Fit a single line, ignoring grouping
- 'parallel lines' — Fit a separate line to each group, but constrain the lines to be parallel
- 'separate lines' — Fit a separate line to each group, with no constraints

`h = aocool(...)` returns a vector of handles to the line objects in the plot.

`[h,atab,ctab] = aocool(...)` returns cell arrays containing the entries in ANOVA table (`atab`) and the table of coefficient estimates (`ctab`). (You can copy a text version of either table to the clipboard by using the Copy Text item on the **Edit** menu.)

`[h,atab,ctab,stats] = aocool(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The ANOVA table output includes tests of the hypotheses that the slopes or intercepts are all the same, against a general alternative that they are not all the same. Sometimes it is preferable to perform a test to determine which pairs of values are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input. You can test either the slopes, the intercepts, or population marginal means (the heights of the curves at the mean x value).

## Examples

This example illustrates how to fit different models non-interactively. After loading the smaller car data set and fitting a separate-slopes model, you can examine the coefficient estimates.

```
load carsmall
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
                   ' ',' ',' ','off','separate lines');
c(:,1:2)
ans =
    'Term'          'Estimate'
    'Intercept'    [45.97983716833132]
    ' 70'          [-8.58050531454973]
    ' 76'          [-3.89017396094922]
    ' 82'          [12.47067927549897]
    'Slope'        [-0.00780212907455]
    ' 70'          [ 0.00195840368824]
    ' 76'          [ 0.00113831038418]
    ' 82'          [-0.00309671407243]
```

Roughly speaking, the lines relating MPG to Weight have an intercept close to 45.98 and a slope close to -0.0078. Each group's coefficients are offset from these values somewhat. For instance, the intercept for the cars made in 1970 is  $45.98 - 8.58 = 37.40$ .

Next, try a fit using parallel lines. (The ANOVA table shows that the parallel-lines fit is significantly worse than the separate-lines fit.)

```
[h,a,c,s] = aoctool(Weight,MPG,Model_Year,0.05,...
                   ' ',' ',' ','off','parallel lines');
c(:,1:2)
ans =
    'Term'          'Estimate'
    'Intercept'    [43.38984085130596]
    ' 70'          [-3.27948192983761]
    ' 76'          [-1.35036234809006]
```

```
' 82'      [ 4.62984427792768]
'Slope'    [-0.00664751826198]
```

Again, there are different intercepts for each group, but this time the slopes are constrained to be the same.

**See Also**

[anova1](#) | [multcompare](#) | [polytool](#)

# TreeBagger.append

---

**Purpose** Append new trees to ensemble

**Syntax** `B = append(B, other)`

**Description** `B = append(B, other)` appends the trees from the `other` ensemble to those in `B`. This method checks for consistency of the `X` and `Y` properties of the two ensembles, as well as consistency of their compact objects and out-of-bag indices, before appending the trees. The output ensemble `B` takes training parameters such as `FBoot`, `Prior`, `Cost`, and `other` from the `B` input. There is no attempt to check if these training parameters are consistent between the two objects.

**See Also** `CompactTreeBagger.combine`

# ProbDistKernel.BandWidth property

---

<b>Purpose</b>	Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object
<b>Description</b>	BandWidth is a read-only property of the ProbDistKernel class. BandWidth is a value specifying the width of the kernel smoothing function used to compute a nonparametric estimate of the probability distribution when creating a ProbDistKernel object.
<b>Values</b>	<p>For a distribution specified to cover only the positive numbers or only a finite interval, the data are transformed before the kernel density is applied, and the bandwidth is on the scale of the transformed data.</p> <p>Use this information to view and compare the width of the kernel smoothing function used to create distributions.</p>
<b>See Also</b>	<code>ksdensity</code>

# barttest

---

**Purpose** Bartlett's test

**Syntax**  
`ndim = barttest(X,alpha)`  
`[ndim,prob,chisquare] = barttest(X,alpha)`

**Description** `ndim = barttest(X,alpha)` returns the number of dimensions necessary to explain the nonrandom variation in the data matrix  $X$ , using the significance probability  $\alpha$ . The dimension is determined by a series of hypothesis tests. The test for `ndim=1` tests the hypothesis that the variances of the data values along each principal component are equal, the test for `ndim=2` tests the hypothesis that the variances along the second through last components are equal, and so on.

`[ndim,prob,chisquare] = barttest(X,alpha)` returns the number of dimensions, the significance values for the hypothesis tests, and the  $\chi^2$  values associated with the tests.

## **Examples** **Determine Dimensions Needed to Explain Nonrandom Data Variation**

This example shows how to determine the number of dimensions needed to explain nonrandom data variation, using Bartlett's test.

Generate a 20-by-6 matrix of sample data from a multinomial distribution.

```
rng('default'); % For reproducibility
X = mvnrnd([0 0],[1 0.99; 0.99 1],20);
X(:,3:4) = mvnrnd([0 0],[1 0.99; 0.99 1],20);
X(:,5:6) = mvnrnd([0 0],[1 0.99; 0.99 1],20);
```

Determine the number of dimensions necessary to explain the nonrandom variation in data matrix  $X$ . Report the significance values for the hypothesis tests.

```
[ndim, prob] = barttest(X,0.05)
```

```
ndim =  
      3  
  
prob =  
      0.0000  
      0.0000  
      0.0000  
      0.5148  
      0.3370
```

The results indicate that three dimensions are necessary to explain the nonrandom variation in X.

**See Also**

[pca](#) | [pcacov](#) | [pcares](#)

# bbdesign

---

**Purpose** Box-Behnken design

**Syntax**  
`dBB = bbdesign(n)`  
`[dBB,blocks] = bbdesign(n)`  
`[...] = bbdesign(n,param,val)`

**Description** `dBB = bbdesign(n)` generates a Box-Behnken design for  $n$  factors.  $n$  must be an integer 3 or larger. The output matrix `dBB` is  $m$ -by- $n$ , where  $m$  is the number of runs in the design. Each row represents one run, with settings for all factors represented in the columns. Factor values are normalized so that the cube points take values between -1 and 1.

`[dBB,blocks] = bbdesign(n)` requests a blocked design. The output `blocks` is an  $m$ -by-1 vector of block numbers for each run. Blocks indicate runs that are to be measured under similar conditions to minimize the effect of inter-block differences on the parameter estimates.

`[...] = bbdesign(n,param,val)` specifies one or more optional parameter/value pairs for the design. The following table lists valid parameter/value pairs.

Parameter	Description	Values
'center'	Number of center points.	Integer. The default depends on $n$ .
'blocksize'	Maximum number of points per block.	Integer. The default is Inf.

## Examples

The following creates a 3-factor Box-Behnken design:

```
dBB = bbdesign(3)
dBB =
    -1    -1     0
    -1     1     0
     1    -1     0
```



```

1      1      0
-1     0     -1
-1     0      1
1      0     -1
1      0      1
0     -1     -1
0     -1      1
0      1     -1
0      1      1
0      0      0
0      0      0
0      0      0

```

The center point is run 3 times to allow for a more uniform estimate of the prediction variance over the entire design space.

Visualize the design as follows:

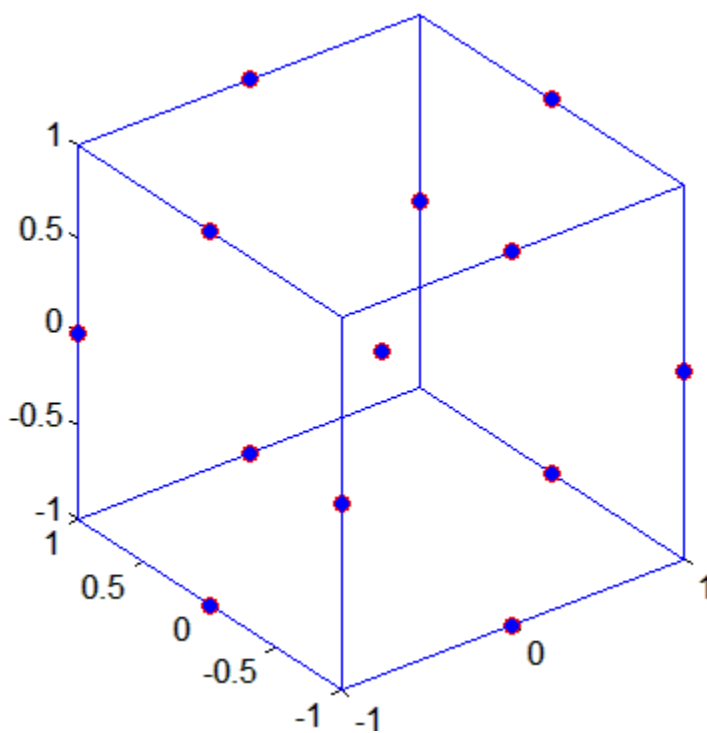
```

plot3(dBB(:,1),dBB(:,2),dBB(:,3),'ro',...
      'MarkerFaceColor','b')
X = [1 -1 -1 -1 1 -1 -1 -1 1 1 -1 -1; ...
     1 1 1 -1 1 1 1 -1 1 1 -1 -1];
Y = [-1 -1 1 -1 -1 -1 1 -1 1 -1 1 -1; ...
     1 -1 1 1 1 -1 1 1 1 -1 1 -1];
Z = [1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1; ...
     1 1 1 1 -1 -1 -1 -1 1 1 1 1];
line(X,Y,Z,'Color','b')
axis square equal

```

# bbdesign

---



**See Also**

`ccdesign`

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Beta probability distribution object

**Description** `prob.BetaDistribution` is an object consisting of parameters, a model description, and sample data for a beta probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Beta')` creates a beta probability distribution object using the default parameter values.

`pd = makedist('Beta','a',a,'b',b)` creates a beta probability distribution object using the specified parameter values.

### Input Arguments

#### **a** - First shape parameter

1 (default) | positive scalar value

First shape parameter of the beta distribution, specified as a positive scalar value.

#### Data Types

single | double

#### **b** - Second shape parameter

1 (default) | positive scalar value

Second shape parameter of the beta distribution, specified as a positive scalar value.

#### Data Types

single | double

**Properties** **a**

First shape parameter of the beta distribution, stored as a positive scalar value.

# prob.BetaDistribution

---

## Data Types

single | double

## **b**

Second shape parameter of the beta distribution, stored as a positive scalar value.

## Data Types

single | double

## DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

## InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated. If IsTruncated equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## **NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## **Data Types**

single | double

## **ParameterCovariance**

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## **Data Types**

single | double

## **ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## **Data Types**

char

## **ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## **Data Types**

logical

## **ParameterNames**

# prob.BetaDistribution

---

Names of distribution parameters, stored as a cell array of strings.  
This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This  
property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a  
vector containing the lower and upper truncation boundaries.  
This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object

random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Beta Distribution

The beta distribution describes a family of curves that are unique in that they are nonzero only on the interval (0,1). A more general version of the distribution assigns parameters to the endpoints of the interval.

The beta distribution uses the following parameters.

Parameter	Description	Support
a	First shape parameter	$a > 0$
b	Second shape parameter	$b > 0$

The probability density function (pdf) is

# prob.BetaDistribution

---

$$f(x|a,b) = \frac{1}{B(a,b)} x^{a-1} (1-x)^{b-1} \quad ; \quad 0 < x < 1,$$

where  $B(\cdot)$  is the beta function.

## Examples

### Create a Beta Distribution Object Using Default Parameters

Create a beta distribution object using the default parameter values.

```
pd = makedist('Beta')
```

```
pd =
```

```
BetaDistribution
```

```
Beta distribution
```

```
a = 1
```

```
b = 1
```

### Create a Beta Distribution Object Using Specified Parameters

Create a beta distribution object by specifying the parameter values.

```
pd = makedist('Beta','a',2,'b',4)
```

```
pd =
```

```
BetaDistribution
```

```
Beta distribution
```

```
a = 2
```

```
b = 4
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```



0.3333

## See Also

`makedist` | `fitdist` | `dfittool`

## Concepts

- “Beta Distribution” on page B-5
- Class Attributes
- Property Attributes

# betacdf

---

**Purpose** Beta cumulative distribution function

**Syntax**  
`p = betacdf(x,a,b)`  
`p = betacdf(x,a,b,'upper')`

**Description** `p = betacdf(x,a,b)` returns the beta cdf at each of the values in `x` using the corresponding parameters in `a` and `b`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `a` and `b` must all be positive, and the values in `x` must lie on the interval `[0,1]`.

`p = betacdf(x,a,b,'upper')` returns the complement of the beta cdf at each of the values in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The beta cdf for a given value `x` and given pair of parameters `a` and `b` is

$$p = F(x | a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$$

where  $B(\cdot)$  is the Beta function.

## Examples **Compute Beta Distribution CDF**

Compute the cdf for a beta distribution with parameters `a = 2` and `b = 2`.

```
x = 0.1:0.2:0.9;  
a = 2;  
b = 2;  
p = betacdf(x,a,b)
```

```
p =  
    0.0280    0.2160    0.5000    0.7840    0.9720
```

```
a = [1 2 3];
```

```
p = betacdf(0.5,a,a)
```

```
p =  
    0.5000  0.5000  0.5000
```

**See Also**

[cdf](#) | [betapdf](#) | [betainv](#) | [betastat](#) | [betalike](#) | [betarnd](#) | [betafit](#)

**How To**

- “Beta Distribution” on page B-5

# betafit

---

**Purpose** Beta parameter estimates

**Syntax**  
`phat = betafit(data)`  
`[phat,pci] = betafit(data,alpha)`

**Description** `phat = betafit(data)` computes the maximum likelihood estimates of the beta distribution parameters  $a$  and  $b$  from the data in the vector `data` and returns a column vector containing the  $a$  and  $b$  estimates, where the beta cdf is given by

$$F(x | a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$$

and  $B(\cdot)$  is the Beta function. The elements of `data` must lie in the open interval  $(0, 1)$ , where the beta distribution is defined. However, it is sometimes also necessary to fit a beta distribution to data that include exact zeros or ones. For such data, the beta likelihood function is unbounded, and standard maximum likelihood estimation is not possible. In that case, `betafit` maximizes a modified likelihood that incorporates the zeros or ones by treating them as if they were values that have been left-censored at `sqrt(realmin)` or right-censored at `1-eps/2`, respectively.

`[phat,pci] = betafit(data,alpha)` returns confidence intervals on the  $a$  and  $b$  parameters in the 2-by-2 matrix `pci`. The first column of the matrix contains the lower and upper confidence bounds for parameter  $a$ , and the second column contains the confidence bounds for parameter  $b$ . The optional input argument `alpha` is a value in the range  $[0, 1]$  specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals. The confidence intervals are based on a normal approximation for the distribution of the logs of the parameter estimates.

**Examples** This example generates 100 beta distributed observations. The true  $a$  and  $b$  parameters are 4 and 3, respectively. Compare these to the

values returned in `p` by the beta fit. Note that the columns of `ci` both bracket the true parameters.

```
data = betarnd(4,3,100,1);
[p,ci] = betafit(data,0.01)
p =
    5.5328    3.8097
ci =
    3.6538    2.6197
    8.3781    5.5402
```

## References

[1] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 95.

## See Also

`mle` | `betapdf` | `betainv` | `betastat` | `betalike` | `betarnd` | `betacdf`

## How To

- “Beta Distribution” on page B-5

# betainv

---

**Purpose** Beta inverse cumulative distribution function

**Syntax** `X = betainv(P,A,B)`

**Description** `X = betainv(P,A,B)` computes the inverse of the beta cdf with parameters specified by A and B for the corresponding probabilities in P. P, A, and B can be vectors, matrices, or multidimensional arrays that are all the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in A and B must all be positive, and the values in P must lie on the interval [0, 1].

The inverse beta cdf for a given probability  $p$  and a given pair of parameters  $a$  and  $b$  is

$$x = F^{-1}(p | a, b) = \{x : F(x | a, b) = p\}$$

where

$$p = F(x | a, b) = \frac{1}{B(a, b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$$

and  $B(\cdot)$  is the Beta function. Each element of output X is the value whose cumulative probability under the beta cdf defined by the corresponding parameters in A and B is specified by the corresponding value in P.

**Algorithms** The `betainv` function uses Newton's method with modifications to constrain steps to the allowable range for  $x$ , i.e., [0 1].

**Examples**

```
p = [0.01 0.5 0.99];  
x = betainv(p,10,5)  
x =  
    0.3726    0.6742    0.8981
```

According to this result, for a beta cdf with  $a = 10$  and  $b = 5$ , a value less than or equal to 0.3726 occurs with probability 0.01. Similarly,

values less than or equal to 0.6742 and 0.8981 occur with respective probabilities 0.5 and 0.99.

**See Also**

`icdf` | `betapdf` | `betafit` | `betainv` | `betastat` | `betalike` | `betarnd`  
| `betacdf`

**How To**

- “Beta Distribution” on page B-5

# betalike

---

**Purpose** Beta negative log-likelihood

**Syntax**  
`nlogL = betalike(params,data)`  
`[nlogL,AVAR] = betalike(params,data)`

**Description** `nlogL = betalike(params,data)` returns the negative of the beta log-likelihood function for the beta parameters  $a$  and  $b$  specified in vector `params` and the observations specified in the column vector `data`.

The elements of `data` must lie in the open interval  $(0, 1)$ , where the beta distribution is defined. However, it is sometimes also necessary to fit a beta distribution to data that include exact zeros or ones. For such data, the beta likelihood function is unbounded, and standard maximum likelihood estimation is not possible. In that case, `betalike` computes a modified likelihood that incorporates the zeros or ones by treating them as if they were values that have been left-censored at `sqrt(realmin)` or right-censored at `1-eps/2`, respectively.

`[nlogL,AVAR] = betalike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates if the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`betalike` is a utility function for maximum likelihood estimation of the beta distribution. The likelihood assumes that all the elements in the data sample are mutually independent. Since `betalike` returns the negative beta log-likelihood function, minimizing `betalike` using `fminsearch` is the same as maximizing the likelihood.

**Examples** This example continues the `betafit` example, which calculates estimates of the beta parameters for some randomly generated beta distributed data.

```
r = betarnd(4,3,100,1);  
[nlogl,AVAR] = betalike(betafit(r),r)  
nlogl =
```



-27.5996

AVAR =

0.2783	0.1316
0.1316	0.0867

**See Also**

[betapdf](#) | [betafit](#) | [betainv](#) | [betastat](#) | [betarnd](#) | [betacdf](#)

**How To**

- “Beta Distribution” on page B-5

# betapdf

---

**Purpose** Beta probability density function

**Syntax** `Y = betapdf(X,A,B)`

**Description** `Y = betapdf(X,A,B)` computes the beta pdf at each of the values in `X` using the corresponding parameters in `A` and `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs. The parameters in `A` and `B` must all be positive, and the values in `X` must lie on the interval `[0, 1]`.

The beta probability density function for a given value  $x$  and given pair of parameters  $a$  and  $b$  is

$$y = f(x | a, b) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1} I_{(0,1)}(x)$$

where  $B(\cdot)$  is the Beta function. The indicator function  $I_{(0,1)}(x)$  ensures that only values of  $x$  in the range  $(0, 1)$  have nonzero probability. The uniform distribution on  $(0, 1)$  is a degenerate case of the beta pdf where  $a = 1$  and  $b = 1$ .

A *likelihood function* is the pdf viewed as a function of the parameters. Maximum likelihood estimators (MLEs) are the values of the parameters that maximize the likelihood function for a fixed value of  $x$ .

**Examples**

```
a = [0.5 1; 2 4]
a =
    0.5000    1.0000
    2.0000    4.0000
y = betapdf(0.5, a, a)
y =
    0.6366    1.0000
    1.5000    2.1875
```

**See Also** pdf | betafit | betainv | betastat | betalike | betarnd | betacdf

**How To**

- “Beta Distribution” on page B-5

# betarnd

---

**Purpose** Beta random numbers

**Syntax**  
R = betarnd(A,B)  
R = betarnd(A,B,m,n,...)  
R = betarnd(A,B,[m,n,...])

**Description** R = betarnd(A,B) generates random numbers from the beta distribution with parameters specified by A and B. A and B can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of R. A scalar input for A or B is expanded to a constant array with the same dimensions as the other input.

R = betarnd(A,B,m,n,...) or R = betarnd(A,B,[m,n,...]) generates an m-by-n-by-... array containing random numbers from the beta distribution with parameters A and B. A and B can each be scalars or arrays of the same size as R.

**Examples**

```
a = [1 1;2 2];  
b = [1 2;1 2];  
  
r = betarnd(a,b)  
r =  
    0.6987    0.6139  
    0.9102    0.8067  
  
r = betarnd(10,10,[1 5])  
r =  
    0.5974    0.4777    0.5538    0.5465    0.6327  
  
r = betarnd(4,2,2,3)  
r =  
    0.3943    0.6101    0.5768  
    0.5990    0.2760    0.5474
```

**See Also** random | betapdf | betafit | betainv | betastat | betalike | betacdf

**How To**

- “Beta Distribution” on page B-5

# betastat

---

**Purpose** Beta mean and variance

**Syntax** `[M,V] = betastat(A,B)`

**Description** `[M,V] = betastat(A,B)`, with  $A > 0$  and  $B > 0$ , returns the mean of and variance for the beta distribution with parameters specified by  $A$  and  $B$ .  $A$  and  $B$  can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $A$  or  $B$  is expanded to a constant array with the same dimensions as the other input.

The mean of the beta distribution with parameters  $a$  and  $b$  is  $a / (a + b)$  and the variance is

$$\frac{ab}{(a+b+1)(a+b)^2}$$

**Examples** If parameters  $a$  and  $b$  are equal, the mean is  $1/2$ .

```
a = 1:6;
[m,v] = betastat(a,a)
m =
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000
v =
    0.0833    0.0500    0.0357    0.0278    0.0227    0.0192
```

**See Also** `betapdf` | `betafit` | `betainv` | `betalike` | `betarnd` | `betacdf`

**How To** • “Beta Distribution” on page B-5

**Purpose** Bayes Information Criterion

**Description** The Bayes Information Criterion:  $2 \cdot N \log L + m \cdot \log(n)$ , where  $n$  is the number of observations and  $m$  is the number of estimated parameters.

# prob.BinomialDistribution

---

**Superclasses**    `ToolboxFittableParametricDistribution`

**Purpose**            Binomial probability distribution object

**Description**     `prob.BinomialDistribution` is an object consisting of parameters, a model description, and sample data for a binomial probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction**    `pd = makedist('Binomial')` creates a binomial probability distribution object using the default parameter values.

`pd = makedist('Binomial','N',n,'p',p)` creates a binomial probability distribution object using the specified parameter values.

## Input Arguments

### **n - Number of trials**

1 (default) | positive integer value

Number of trials for the binomial distribution, specified as a positive integer value.

### **Data Types**

single | double

### **p - Probability of success**

0.5 (default) | positive scalar value in the range [0,1]

Probability of success of any individual trial for the binomial distribution, specified as a positive scalar value in the range [0,1].

### **Data Types**

single | double



## Properties

### **N**

Number of trials for the binomial distribution, stored as a positive integer value.

#### **Data Types**

single | double

### **p**

Probability of success of any individual trial for the binomial distribution, stored as a positive scalar value in the range [0,1].

#### **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

#### **Data Types**

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

#### **Data Types**

single | double

### **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

# prob.BinomialDistribution

---

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object

# prob.BinomialDistribution

---

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Binomial Distribution

The binomial distribution models the total number of successes in repeated trials from an infinite population under the following conditions:

- Only two outcomes are possible for each of  $n$  trials.
- The probability of success for each trial is constant.
- All trials are independent of each other.

The binomial distribution uses the following parameters.

Parameter	Description	Support
N	Number of trials	positive integer
p	Probability of success	$0 \leq p \leq 1$

The probability density function (pdf) is

$$f(x | n, p) = \binom{n}{x} p^x (1-p)^{n-x} \quad ; \quad x = 0, 1, 2, \dots, n,$$

where  $x$  is the number of successes in  $n$  trials of a Bernoulli process with probability of success  $p$ .

## Examples

### Create a Binomial Distribution Object Using Default Parameters

Create a binomial distribution object using the default parameter values.

```
pd = makedist('Binomial')
```

```
pd =
```

```
BinomialDistribution
```

```
Binomial distribution
```

```
N = 1
```

```
p = 0.5
```

### Create a Binomial Distribution Object Using Specified Parameters

Create a binomial distribution object by specifying the parameter values.

```
pd = makedist('Binomial','N',30,'p',0.25)
```

```
pd =
```

# prob.BinomialDistribution

---

```
BinomialDistribution
```

```
Binomial distribution
```

```
N = 30
```

```
p = 0.25
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
7.5000
```

## See Also

```
makedist | fitdist | dfittool
```

## Concepts

- “Binomial Distribution” on page B-9
- “Bernoulli Distribution” on page B-3
- Class Attributes
- Property Attributes

**Purpose** Binomial cumulative distribution function

**Syntax**  
`y = binocdf(x,N,p)`  
`y = binocdf(x,N,p,'upper')`

**Description** `y = binocdf(x,N,p)` computes a binomial cdf at each of the values in `x` using the corresponding number of trials in `N` and probability of success for each trial in `p`. `x`, `N`, and `p` can be vectors, matrices, or multidimensional arrays that are all the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs. The values in `N` must all be positive integers, the values in `x` must lie on the interval  $[0, N]$ , and the values in `p` must lie on the interval  $[0, 1]$ .

`y = binocdf(x,N,p,'upper')` returns the complement of the binomial cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The binomial cdf for a given value  $x$  and a given pair of parameters  $n$  and  $p$  is

$$y = F(x | n, p) = \sum_{i=0}^x \binom{n}{i} p^i (1-p)^{(n-i)} I_{(0,1,\dots,n)}(i).$$

The result,  $y$ , is the probability of observing up to  $x$  successes in  $n$  independent trials, where the probability of success in any given trial is

$p$ . The indicator function  $I_{(0,1,\dots,n)}(i)$  ensures that  $x$  only adopts values of  $0, 1, \dots, n$ .

## Examples **Compute Binomial CDF**

If a baseball team plays 162 games in a season and has a 50-50 chance of winning any game, then the probability of that team winning more than 100 games in a season is:

```
1 - binocdf(100,162,0.5)
```

# binocdf

---

```
ans =
```

```
0.0010
```

The result is 0.001 (i.e.,  $1 - 0.999$ ). If a team wins 100 or more games in a season, this result suggests that it is likely that the team's true probability of winning any game is greater than 0.5.

## See Also

[cdf](#) | [binopdf](#) | [binoinv](#) | [binostat](#) | [binofit](#) | [binornd](#)

## How To

- “Binomial Distribution” on page B-9



**Purpose**

Binomial parameter estimates

**Syntax**

```
phat = binofit(x,n)
[phat,pci] = binofit(x,n)
[phat,pci] = binofit(x,n,alpha)
```

**Description**

`phat = binofit(x,n)` returns a maximum likelihood estimate of the probability of success in a given binomial trial based on the number of successes, `x`, observed in `n` independent trials. If `x = (x(1), x(2), ... x(k))` is a vector, `binofit` returns a vector of the same size as `x` whose `i`th entry is the parameter estimate for `x(i)`. All `k` estimates are independent of each other. If `n = (n(1), n(2), ..., n(k))` is a vector of the same size as `x`, the binomial fit, `binofit`, returns a vector whose `i`th entry is the parameter estimate based on the number of successes `x(i)` in `n(i)` independent trials. A scalar value for `x` or `n` is expanded to the same size as the other input.

`[phat,pci] = binofit(x,n)` returns the probability estimate, `phat`, and the 95% confidence intervals, `pci`. `binofit` uses the Clopper-Pearson method to calculate confidence intervals.

`[phat,pci] = binofit(x,n,alpha)` returns the  $100(1 - \alpha)\%$  confidence intervals. For example, `alpha = 0.01` yields 99% confidence intervals.

---

**Note** `binofit` behaves differently than other Statistics Toolbox functions that compute parameter estimates, in that it returns independent estimates for each entry of `x`. By comparison, `expfit` returns a single parameter estimate based on all the entries of `x`.

---

Unlike most other distribution fitting functions, the `binofit` function treats its input `x` vector as a collection of measurements from separate samples. If you want to treat `x` as a single sample and compute a single parameter estimate for it, you can use `binofit(sum(x),sum(n))` when `n` is a vector, and `binofit(sum(X),N*length(X))` when `n` is a scalar.

## Examples

This example generates a binomial sample of 100 elements, where the probability of success in a given trial is 0.6, and then estimates this probability from the outcomes in the sample.

```
r = binornd(100,0.6);  
[phat,pci] = binofit(r,100)  
phat =  
    0.5800  
pci =  
    0.4771    0.6780
```

The 95% confidence interval, `pci`, contains the true value, 0.6.

## References

[1] Johnson, N. L., S. Kotz, and A. W. Kemp. *Univariate Discrete Distributions*. Hoboken, NJ: Wiley-Interscience, 1993.

## See Also

`mle` | `binopdf` | `binocdf` | `binoinv` | `binostat` | `binornd`

## How To

- “Binomial Distribution” on page B-9

**Purpose** Binomial inverse cumulative distribution function

**Syntax** `X = binoinv(Y,N,P)`

**Description** `X = binoinv(Y,N,P)` returns the smallest integer  $X$  such that the binomial cdf evaluated at  $X$  is equal to or exceeds  $Y$ . You can think of  $Y$  as the probability of observing  $X$  successes in  $N$  independent trials where  $P$  is the probability of success in each trial. Each  $X$  is a positive integer less than or equal to  $N$ .

$Y$ ,  $N$ , and  $P$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in  $N$  must be positive integers, and the values in both  $P$  and  $Y$  must lie on the interval  $[0\ 1]$ .

**Examples** If a baseball team has a 50-50 chance of winning any game, what is a reasonable range of games this team might win over a season of 162 games?

```
binoinv([0.05 0.95],162,0.5)
ans =
    71    91
```

This result means that in 90% of baseball seasons, a .500 team should win between 71 and 91 games.

**See Also** `icdf` | `binopdf` | `binocdf` | `binofit` | `binostat` | `binornd`

**How To** • “Binomial Distribution” on page B-9

# binopdf

---

**Purpose** Binomial probability density function

**Syntax** `Y = binopdf(X,N,P)`

**Description** `Y = binopdf(X,N,P)` computes the binomial pdf at each of the values in `X` using the corresponding number of trials in `N` and probability of success for each trial in `P`. `Y`, `N`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions of the other inputs.

The parameters in `N` must be positive integers, and the values in `P` must lie on the interval `[0, 1]`.

The binomial probability density function for a given value  $x$  and given pair of parameters  $n$  and  $p$  is

$$y = f(x | n, p) = \binom{n}{x} p^x q^{(n-x)} I_{(0,1,\dots,n)}(x)$$

where  $q = 1 - p$ . The result,  $y$ , is the probability of observing  $x$  successes in  $n$  independent trials, where the probability of success in any *given* trial is  $p$ . The indicator function  $I_{(0,1,\dots,n)}(x)$  ensures that  $x$  only adopts values of  $0, 1, \dots, n$ .

**Examples** A Quality Assurance inspector tests 200 circuit boards a day. If 2% of the boards have defects, what is the probability that the inspector will find no defective boards on any given day?

```
binopdf(0,200,0.02)
ans =
    0.0176
```

What is the most likely number of defective boards the inspector will find?

```
defects=0:200;
y = binopdf(defects,200,.02);
```

```
[x,i]=max(y);  
defects(i)  
ans =  
    4
```

**See Also**

[pdf](#) | [binoinv](#) | [binocdf](#) | [binofit](#) | [binostat](#) | [binornd](#)

**How To**

- “Binomial Distribution” on page B-9

# binornd

---

**Purpose** Binomial random numbers

**Syntax**  
`R = binornd(N,P)`  
`R = binornd(N,P,m,n,...)`  
`R = binornd(N,P,[m,n,...])`

**Description** `R = binornd(N,P)` generates random numbers from the binomial distribution with parameters specified by the number of trials, `N`, and probability of success for each trial, `P`. `N` and `P` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input.

`R = binornd(N,P,m,n,...)` or `R = binornd(N,P,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the binomial distribution with parameters `N` and `P`. `N` and `P` can each be scalars or arrays of the same size as `R`.

**Algorithms** The `binornd` function uses the direct method using the definition of the binomial distribution as a sum of Bernoulli random variables.

**Examples**

```
n = 10:10:60;  
  
r1 = binornd(n,1./n)  
r1 =  
    2    1    0    1    1    2  
  
r2 = binornd(n,1./n,[1 6])  
r2 =  
    0    1    2    1    3    1  
  
r3 = binornd(n,1./n,1,6)  
r3 =  
    0    1    1    1    0    3
```

**See Also** `random` | `binoinv` | `binocdf` | `binofit` | `binostat` | `binopdf`

**How To**

- “Binomial Distribution” on page B-9

# binostat

---

**Purpose** Binomial mean and variance

**Syntax** `[M,V] = binostat(N,P)`

**Description** `[M,V] = binostat(N,P)` returns the mean of and variance for the binomial distribution with parameters specified by the number of trials,  $N$ , and probability of success for each trial,  $P$ .  $N$  and  $P$  can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $N$  or  $P$  is expanded to a constant array with the same dimensions as the other input.

The mean of the binomial distribution with parameters  $n$  and  $p$  is  $np$ . The variance is  $npq$ , where  $q = 1-p$ .

## Examples

```
n = logspace(1,5,5)
n =
    10    100   1000  10000 100000
```

```
[m,v] = binostat(n,1./n)
m =
    1    1    1    1    1
v =
    0.9000    0.9900    0.9990    0.9999    1.0000
```

```
[m,v] = binostat(n,1/2)
m =
    5    50   500   5000  50000
v =
    1.0e+04 *
    0.0003    0.0025    0.0250    0.2500    2.5000
```

**See Also** `binoinv` | `binocdf` | `binofit` | `binornd` | `binopdf`

**How To** • “Binomial Distribution” on page B-9



**Purpose**

Biplot

**Syntax**

```
biplot(coefs)
h = biplot(coefs, 'Name', Value)
```

**Description**

`biplot(coefs)` creates a biplot of the coefficients in the matrix `coefs`. The biplot is 2-D if `coefs` has two columns or 3-D if it has three columns. `coefs` usually contains principal component coefficients created with `pca`, `pcacov`, or factor loadings estimated with `factoran`. The axes in the biplot represent the principal components or latent factors (columns of `coefs`), and the observed variables (rows of `coefs`) are represented as vectors.

A biplot allows you to visualize the magnitude and sign of each variable's contribution to the first two or three principal components, and how each observation is represented in terms of those components.

`biplot` imposes a sign convention, forcing the element with largest magnitude in each column of `coefs` to be positive. This flips some of the vectors in `coefs` to the opposite direction, but often makes the plot easier to read. Interpretation of the plot is unaffected, because changing the sign of a coefficient vector does not change its meaning.

`biplot` scales the scores so that they fit on the plot: It divides each score by the maximum absolute value of all scores, and multiplies by the maximum coefficient length of `coefs`. Then `biplot` changes the sign of score coordinates according to the sign convention for the `coefs`.

`h = biplot(coefs, 'Name', Value)` specifies one or more name/value input pairs and returns a column vector of handles to the graphics objects created by `biplot`. The `h` contains, in order, handles corresponding to variables (line handles, followed by marker handles, followed by text handles), to observations (if present, marker handles followed by text handles), and to the axis lines.

**Input Arguments****Name-Value Pair Arguments****'Scores'**

# biplot

---

Plots both `coefs` and the `scores` in the matrix `scores` in the biplot. `scores` usually contains principal component scores created with `pca` or factor scores estimated with `factoran`. Each observation (row of `scores`) is represented as a point in the biplot.

## 'VarLabels'

Labels each vector (variable) with the text in the character array or cell array `varlabels`.

## 'ObsLabels'

Uses the text in the character array or cell array `obslabels` as observation names when displaying data cursors.

## 'Positive'

- 'true' — restricts the biplot to the positive quadrant (in 2-D) or octant (in 3-D).
- 'false' — makes the biplot over the range  $\pm \max(\text{coefs}(:))$  for all coordinates.

**Default:** false

## 'PropertyName'

Specifies optional property name/value pairs for all line graphics objects created by `biplot`.

## Examples

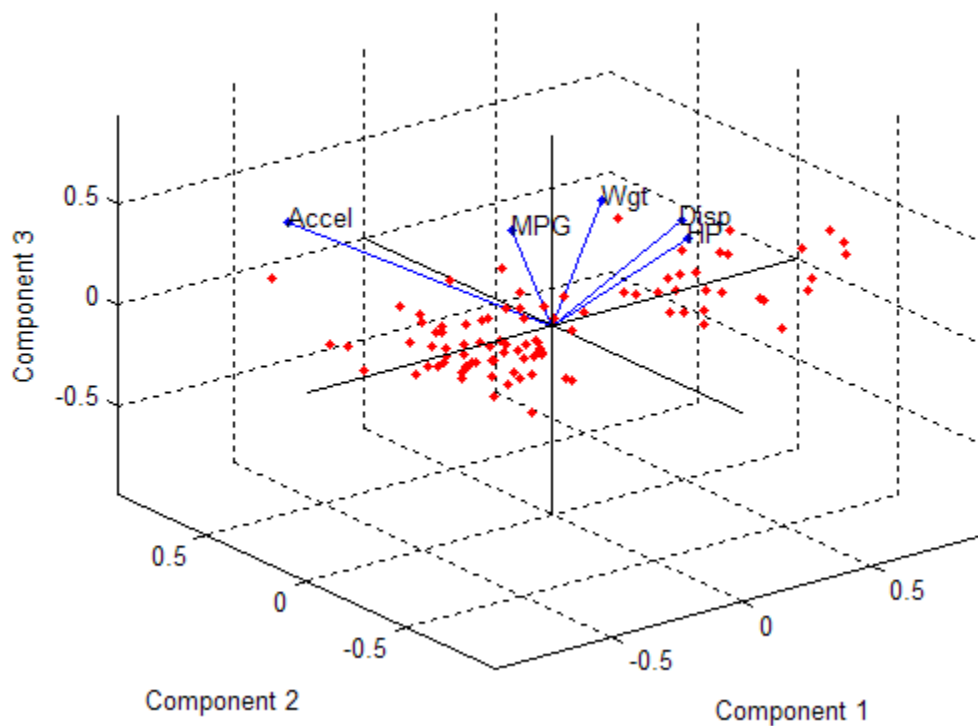
Perform a principal component analysis of the data in `carsmall.mat`:

```
load carsmall
x = [Acceleration Displacement Horsepower MPG Weight];
x = x(all(~isnan(x),2),:);

[coefs,score] = pca(zscore(x));
```

View the data and the original variables in the space of the first three principal components:

```
vbls = {'Accel','Disp','HP','MPG','Wgt'};  
biplot(coefs(:,1:3),'scores',score(:,1:3),...  
       'varlabels',vbls);
```

**See Also**

[factoran](#) | [nnmf](#) | [pca](#) | [pcacov](#) | [rotatefactors](#)

# prob.BirnbaumSaundersDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Birnbaum-Saunders probability distribution object

**Description** `prob.BirnbaumSaundersDistribution` is an object consisting of parameters, a model description, and sample data for a Birnbaum-Saunders probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('BirnbaumSaunders')` creates a Birnbaum-Saunders probability distribution object using the default parameter values.

`pd = makedist('BirnbaumSaunders', 'beta', beta, 'gamma', gamma)` creates a Birnbaum-Saunders distribution object using the specified parameter values.

## Input Arguments

### **beta** - Scale parameter

1 (default) | positive scalar value

Scale parameter of the Birnbaum-Saunders distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **gamma** - Shape parameter

1 (default) | nonnegative scalar value

Shape parameter of the Birnbaum-Saunders distribution, specified as a nonnegative scalar value.

### **Data Types**

single | double

## Properties

### **beta**

Scale parameter of the Birnbaum-Saunders distribution, stored as a positive scalar value.

### **Data Types**

single | double

### **gamma**

Shape parameter of the Birnbaum-Saunders distribution, stored as a nonnegative scalar value.

### **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### **Data Types**

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

### **Data Types**

single | double

### **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

# prob.BirnbaumSaundersDistribution

---

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object

# prob.BirnbaumSaundersDistribution

---

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Birnbaum-Saunders Distribution

The Birnbaum-Saunders distribution was originally proposed as a lifetime model for materials subject to cyclic patterns of stress and strain, where the ultimate failure of the material comes from the growth of a prominent flaw. It is also called the fatigue life distribution.

The Birnbaum-Saunders distribution uses the following parameters.



# prob.BirnbaumSaundersDistribution

Parameter	Description	Support
beta	Scale parameter	$\beta > 0$
gamma	Shape parameter	$\gamma \geq 0$

The probability density function (pdf) is

$$f(x | \beta, \gamma) = \frac{1}{\sqrt{2\pi}} \exp \left\{ -\frac{\left( \sqrt{\frac{x}{\beta}} - \sqrt{\frac{\beta}{x}} \right)^2}{2\gamma^2} \right\} \left( \frac{\sqrt{\frac{x}{\beta}} - \sqrt{\frac{\beta}{x}}}{2\gamma x} \right) ; \quad x > 0.$$

## Examples

### Create a Birnbaum-Saunders Distribution Object Using Default Parameters

Create a Birnbaum-Saunders distribution object using the default parameter values.

```
pd = makedist('BirnbaumSaunders')
```

```
pd =
```

```
BirnbaumSaundersDistribution
```

```
Birnbaum-Saunders distribution
```

```
beta = 1
```

```
gamma = 1
```

### Create a Birnbaum-Saunders Distribution Object Using Specified Parameter Values

Create a Birnbaum-Saunders distribution object by specifying the parameter values.

```
pd = makedist('BirnbaumSaunders','beta',2,'gamma',5)
```

# prob.BirnbaumSaundersDistribution

---

```
pd =
```

```
    BirnbaumSaundersDistribution
```

```
    Birnbaum-Saunders distribution
```

```
        beta = 2
```

```
        gamma = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
    27
```

## See Also

```
makedist | fitdist | dfittool
```

## Concepts

- “Birnbaum-Saunders Distribution” on page B-13
- Class Attributes
- Property Attributes

**Purpose**

Bootstrap confidence interval

**Syntax**

```
ci = bootci(nboot,bootfun,...)
ci = bootci(nboot,{bootfun,...},'alpha',alpha)
ci = bootci(nboot,{bootfun,...},...,'type',type)
ci = bootci(nboot,{bootfun,...},...,'type','student',
    'nbootstd',nbootstd)
ci = bootci(nboot,{bootfun,...},...,'type','student','stderr',
    stderr)
ci = bootci(nboot,{bootfun,...},...,'Weights',weights)
ci = bootci(nboot,{bootfun,...},...,'Options',options)
[ci,bootstat] = bootci(...)
```

**Description**

`ci = bootci(nboot,bootfun,...)` computes the 95% bootstrap confidence interval of the statistic computed by the function `bootfun`. `nboot` is a positive integer indicating the number of bootstrap samples used in the computation. `bootfun` is a function handle specified with `@`. The third and later input arguments to `bootci` are data (scalars, column vectors, or matrices) that are used to create inputs to `bootfun`. `bootci` creates each bootstrap sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). Scalar data are passed to `bootfun` unchanged.

If `bootfun` returns a scalar, `ci` is a vector containing the lower and upper bounds of the confidence interval. If `bootfun` returns a vector of length  $m$ , `ci` is an array of size 2-by- $m$ , where `ci(1,:)` are lower bounds and `ci(2,:)` are upper bounds. If `bootfun` returns an array of size  $m$ -by- $n$ -by- $p$ -by-..., `ci` is an array of size 2-by- $m$ -by- $n$ -by- $p$ -by-..., where `ci(1,:,:,:,...)` is an array of lower bounds and `ci(2,:,:,:,...)` is an array of upper bounds.

`ci = bootci(nboot,{bootfun,...},'alpha',alpha)` computes the  $100*(1-\alpha)$  bootstrap confidence interval of the statistic defined by the function `bootfun`. `bootfun` and the data that `bootci` passes to it are contained in a single cell array. `alpha` is a scalar between 0 and 1. The default value of `alpha` is 0.05.

`ci = bootci(nboot, {bootfun, ...}, ..., 'type', type)` computes the bootstrap confidence interval of the statistic defined by the function `bootfun`. `type` is the confidence interval type, chosen from among the following strings:

- `'norm'` or `'normal'` — Normal approximated interval with bootstrapped bias and standard error.
- `'per'` or `'percentile'` — Basic percentile method.
- `'cper'` or `'corrected percentile'` — Bias corrected percentile method.
- `'bca'` — Bias corrected and accelerated percentile method. This is the default.
- `'stud'` or `'student'` — Studentized confidence interval.

`ci = bootci(nboot, {bootfun, ...}, ..., 'type', 'student', 'nbootstd', nbootstd)` computes the studentized bootstrap confidence interval of the statistic defined by the function `bootfun`. The standard error of the bootstrap statistics is estimated using bootstrap, with `nbootstd` bootstrap data samples. `nbootstd` is a positive integer value. The default value of `nbootstd` is 100.

`ci = bootci(nboot, {bootfun, ...}, ..., 'type', 'student', 'stderr', stderr)` computes the studentized bootstrap confidence interval of statistics defined by the function `bootfun`. The standard error of the bootstrap statistics is evaluated by the function `stderr`. `stderr` is a function handle. `stderr` takes the same arguments as `bootfun` and returns the standard error of the statistic computed by `bootfun`.

`ci = bootci(nboot, {bootfun, ...}, ..., 'Weights', weights)` specifies observation weights. `weights` must be a vector of non-negative numbers with at least one positive element. The number of elements in `weights` must be equal to the number of rows in non-scalar input arguments to `bootfun`. To obtain one bootstrap replicate, `bootstrp` samples  $N$  out of  $N$  with replacement using these weights as multinomial sampling probabilities.

`ci = bootci(nboot,{bootfun,...},...,'Options',options)` specifies options that govern the computation of bootstrap iterations. One option requests that `bootci` perform bootstrap iterations using multiple processors, if the Parallel Computing Toolbox is available. Two options specify the random number streams to be used in bootstrap resampling. This argument is a struct that you can create with a call to `statset`. You can retrieve values of the individual fields with a call to `statget`. Applicable `statset` parameters are:

- `'UseParallel'` — If `true` and if a `parpool` of the Parallel Computing Toolbox is open, compute bootstrap iterations in parallel. If the Parallel Computing Toolbox is not installed, or a `parpool` is not open, computation occurs in serial mode. Default is `false`, or serial computation.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. Default is `false`. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`.
- `Streams` — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `bootci` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open Parallel pool
  - `UseParallel` is `true`
  - `UseSubstreams` is `false`
 In that case, use a cell array the same size as the Parallel pool.

`[ci,bootstat] = bootci(...)` also returns the bootstrapped statistic computed for each of the `nboot` bootstrap replicate samples. Each row of `bootstat` contains the results of applying `bootfun` to one bootstrap sample. If `bootfun` returns a matrix or array, then this output is converted to a row vector for storage in `bootstat`.

## Examples

Compute the confidence interval for the capability index in statistical process control:

```
y = normrnd(1,1,30,1);           % Simulated process data
```

# bootci

---

```
LSL = -3; USL = 3; % Process specifications
capable = @(x)(USL-LSL)./(6* std(x)); % Process capability
ci = bootci(2000, capable, y) % BCa confidence interval
ci =
    0.8122
    1.2657

sci = bootci(2000, {capable, y}, 'type', 'student') % Studentized ci
sci =
    0.7739
    1.2707
```

## See Also

[bootstrp](#) | [jackknife](#) | [statget](#) | [statset](#) | [randsample](#) | [parfor](#)

**Purpose**

Bootstrap sampling

**Syntax**

```
bootstat = bootstrap(nboot,bootfun,d1,...)
[bootstat,bootsam] = bootstrap(...)
bootstat = bootstrap(...,'Name',Value)
```

**Description**

`bootstat = bootstrap(nboot,bootfun,d1,...)` draws `nboot` bootstrap data samples, computes statistics on each sample using `bootfun`, and returns the results in the matrix `bootstat`. `nboot` must be a positive integer. `bootfun` is a function handle specified with `@`. Each row of `bootstat` contains the results of applying `bootfun` to one bootstrap sample. If `bootfun` returns a matrix or array, then this output is converted to a row vector for storage in `bootstat`.

The third and later input arguments (`d1,...`) are data (scalars, column vectors, or matrices) used to create inputs to `bootfun`. `bootstrap` creates each bootstrap sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). `bootfun` accepts scalar data unchanged.

`[bootstat,bootsam] = bootstrap(...)` returns an `n`-by-`nboot` matrix of bootstrap indices, `bootsam`. Each column in `bootsam` contains indices of the values that were drawn from the original data sets to constitute the corresponding bootstrap sample. For example, if `d1,...` each contain 16 values, and `nboot = 4`, then `bootsam` is a 16-by-4 matrix. The first column contains the indices of the 16 values drawn from `d1,...`, for the first of the four bootstrap samples, the second column contains the indices for the second of the four bootstrap samples, and so on. (The bootstrap indices are the same for all input data sets.) To get the output samples `bootsam` without applying a function, set `bootfun` to empty (`[]`).

`bootstat = bootstrap(...,'Name',Value)` uses additional arguments specified by one or more `Name,Value` pair arguments. The name-value pairs must appear after the data arguments. The available name-value pairs:

- `'Weights'` — Observation weights. The `weights` value must be a vector of nonnegative numbers with at least one positive element.

The number of elements in `weights` must be equal to the number of rows in non-scalar input arguments to `bootstrap`. To obtain one bootstrap replicate, `bootstrap` samples  $N$  out of  $N$  with replacement using these weights as multinomial sampling probabilities.

- 'Options' — The value is a structure that contains options specifying whether to compute bootstrap iterations in parallel, and specifying how to use random numbers during the bootstrap sampling. Create the options structure with `statset`. Applicable `statset` parameters are:
  - 'UseParallel' — If true and if a `parpool` of the Parallel Computing Toolbox is open, compute bootstrap iterations in parallel. If the Parallel Computing Toolbox is not installed, or a `parpool` is not open, computation occurs in serial mode. Default is false, meaning serial computation.
  - UseSubstreams — Set to true to compute in parallel in a reproducible fashion. Default is false. To compute reproducibly, set `Streams` to a type allowing substreams: 'mlfg6331\_64' or 'mrg32k3a'.
  - Streams — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `bootstrap` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
    - You have an open Parallel pool
    - UseParallel is true
    - UseSubstreams is falseIn that case, use a cell array the same size as the Parallel pool.

## Examples

### Bootstrapping a Correlation Coefficient Standard Error

Load a data set containing the LSAT scores and law-school GPA for 15 students. These 15 data points are resampled to create 1000 different data sets, and the correlation between the two variables is computed for each data set.



```
load lawdata
[bootstat,bootsam] = bootstrp(1000,@corr,lsat,gpa);
```

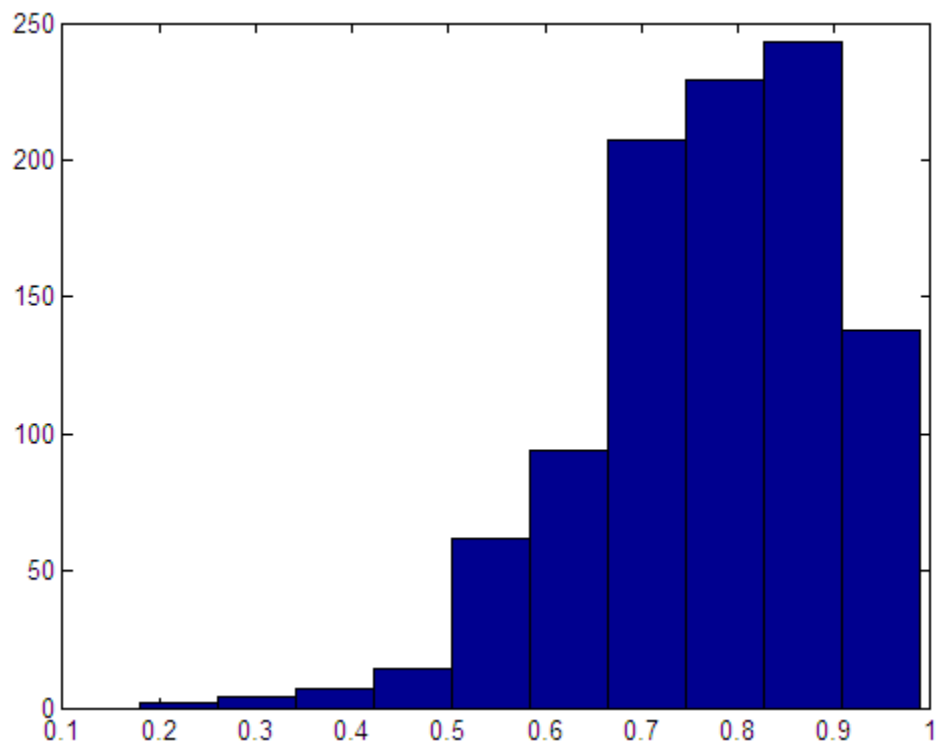
Display the first 5 bootstrapped correlation coefficients.

```
bootstat(1:5,:)
ans =
    0.6600
    0.7969
    0.5807
    0.8766
    0.9197
```

Display the indices of the data selected for the first 5 bootstrap samples.

```
bootsam(:,1:5)
ans =
     9     8    15    11    15
    14     7     6     7    14
     4     6    10     3    11
     3    10    11     9     2
    15     4    13     4    14
     9     4     5     2    10
     8     5     4     3    13
     1     9     1    15    11
    10     8     6    12     3
     1     4     5     2     8
     1     1    10     6     2
     3    10    15    10     8
    14     6    10     3     8
    13    12     1     2     4
    12     6     4     9     8
hist(bootstat)
```

# bootstrap



The histogram shows the variation of the correlation coefficient across all the bootstrap samples. The sample minimum is positive, indicating that the relationship between LSAT score and GPA is not accidental.

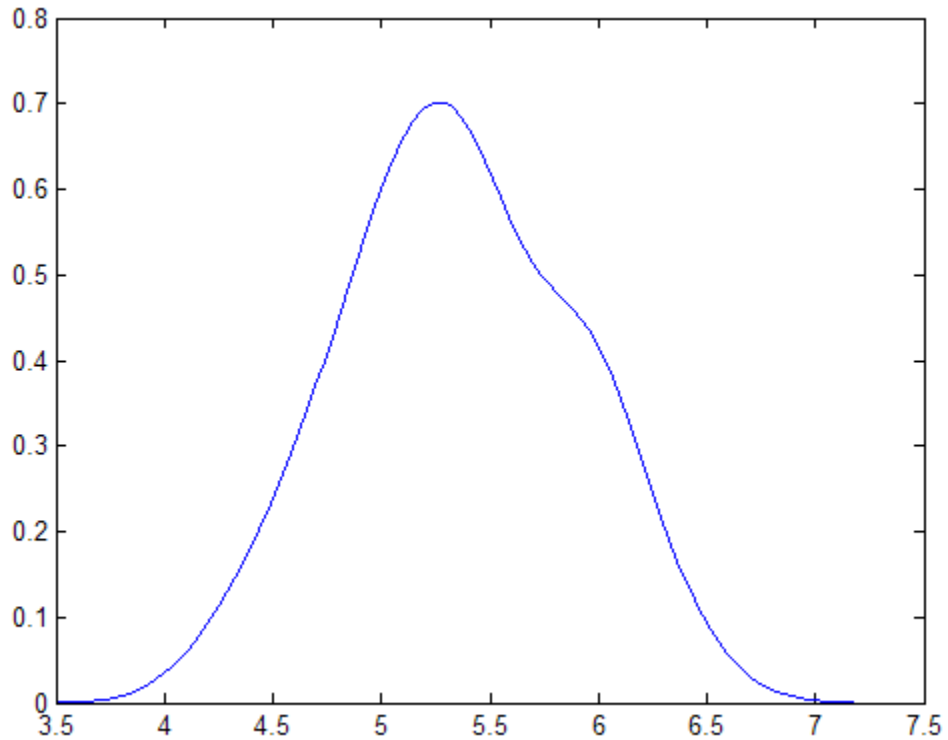
Finally, compute a bootstrap standard of error for the estimated correlation coefficient.

```
se = std(bootstat)
se =
    0.1327
```

### Estimating the Density of Bootstrapped Statistic

Compute a sample of 100 bootstrapped means of random samples taken from the vector Y, and plot an estimate of the density of these bootstrapped means:

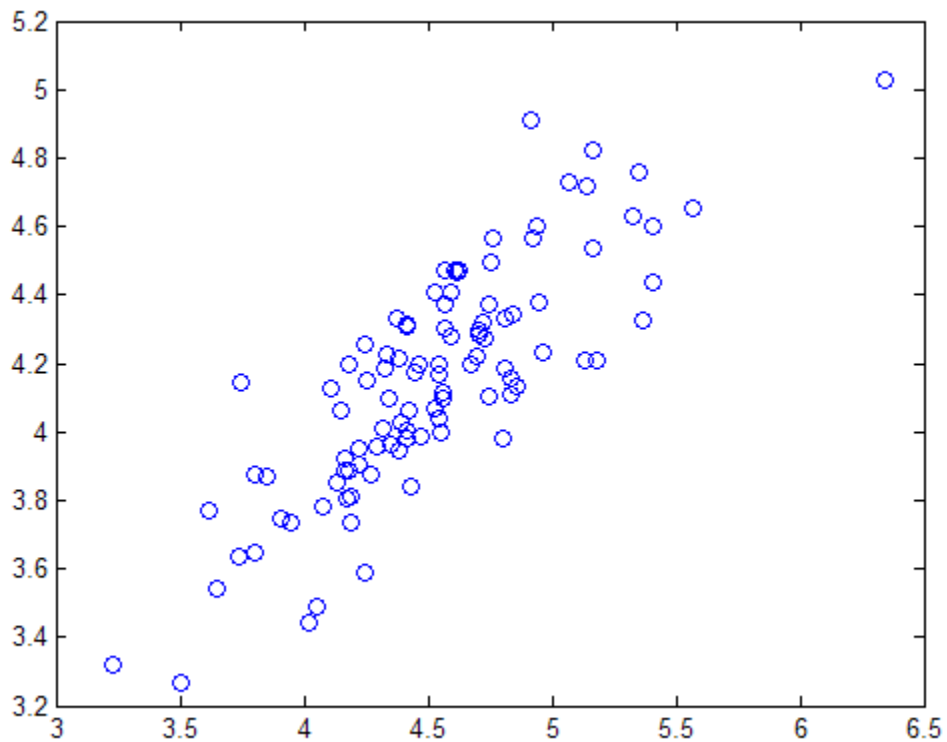
```
y = exprnd(5,100,1);  
m = bootstrp(100,@mean,y);  
[fi,xi] = ksdensity(m);  
plot(xi,fi);
```



## Bootstrapping More Than One Statistic

Compute a sample of 100 bootstrapped means and standard deviations of random samples taken from the vector Y, and plot the bootstrap estimate pairs:

```
y = exprnd(5,100,1);  
stats = bootstrap(100,@(x)[mean(x) std(x)],y);  
plot(stats(:,1),stats(:,2),'o')
```



### Bootstrapping a Regression Model

Estimate the standard errors for a coefficient vector in a linear regression by bootstrapping residuals:

```
load hald
x = [ones(size(heat)),ingredients];
y = heat;
b = regress(y,x);
yfit = x*b;
resid = y - yfit;
se = std(bootstrp(...
    1000,@(bootr)regress(yfit+bootr,x),resid));
```

### See Also

hist | bootci | ksdensity | parfor | random | randsample |  
RandStream | statget | statset

# boxplot

---

**Purpose** Box plot

**Syntax**  
`boxplot(X)`  
`boxplot(X,G)`  
`boxplot(axes,X,...)`  
`boxplot(...,'Name',value)`

**Description** `boxplot(X)` produces a box plot of the data in *X*. If *X* is a matrix, there is one box per column; if *X* is a vector, there is just one box. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually. For controlling how much the whiskers extend, see the 'whiskers' name-value pair argument.

`boxplot(X,G)` specifies one or more grouping variables *G*, producing a separate box for each set of *X* values sharing the same *G* value or values. Grouping variables must have one row per element of *X*, or one row per column of *X*. Specify a single grouping variable in *G* using a vector, a character array, a cell array of strings, or a vector categorical array; specify multiple grouping variables in *G* using a cell array of these variable types, such as {*G1 G2 G3*}, or by using a matrix. If multiple grouping variables are used, they must all be the same length. Groups that contain a NaN value or an empty string in a grouping variable are omitted, and are not counted in the number of groups considered by other parameters.

By default, character and string grouping variables are sorted in the order they initially appear in the data, categorical grouping variables are sorted by the order of their levels, and numeric grouping variables are sorted in numeric order. To control the order of groups, do one of the following:

- Use categorical variables in *G* and specify the order of their levels.
- Use the 'grouporder' parameter described below.
- Pre-sort your data.

`boxplot(axes,X,...)` creates the plot in the axes with handle axes.

`boxplot(..., 'Name', value)` specifies one or more optional parameter name/value pairs, as described in the following table. Specify *Name* in single quotes.

Name	Value
'plotstyle'	<ul style="list-style-type: none"> <li>'traditional' — Traditional box style. This is the default.</li> <li>'compact' — Box style designed for plots with many groups. This style changes the defaults for some other parameters, as described in the following table.</li> </ul>
'boxstyle'	<ul style="list-style-type: none"> <li>'outline' — Draws an unfilled box with dashed whiskers. This is the default.</li> <li>'filled' — Draws a narrow filled box with lines for whiskers.</li> </ul>
'colorgroup'	One or more grouping variables, of the same type as permitted for <code>G</code> , specifying that the box color should change when the specified variables change. The default is <code>[]</code> for no box color change.
'colors'	Colors for boxes, specified as a single color (such as 'r' or <code>[1 0 0]</code> ) or multiple colors (such as 'rgbm' or a three-column matrix of RGB values). The sequence is replicated or truncated as required, so for example 'rb' gives boxes that alternate in color. The default when no 'colorgroup' is specified is to use the same color scheme for all boxes. The default when 'colorgroup' is specified is a modified hsv colormap.

# boxplot

---

Name	Value
'datalim'	A two-element vector containing lower and upper limits, used by 'extrememode' to determine which points are extreme. The default is [-Inf Inf].
'extrememode'	<ul style="list-style-type: none"><li>• 'clip' — Moves data outside the <code>datalim</code> limits to the limit. This is the default.</li><li>• 'compress' — Evenly distributes data outside the <code>datalim</code> limits in a region just outside the limit, retaining the relative order of the points.</li></ul> <p>A dotted line marks the limit if any points are outside it, and two gray lines mark the compression region if any points are compressed. Values at +/-Inf can be clipped or compressed, but NaN values still do not appear on the plot. Box notches are drawn to scale and may extend beyond the bounds if the median is inside the limit; they are not drawn if the median is outside the limits.</p>
'factordirection'	<ul style="list-style-type: none"><li>• 'data' — Arranges factors with the first value next to the origin. This is the default.</li><li>• 'list' — Arranges factors left-to-right if on the <i>x</i> axis or top-to-bottom if on the <i>y</i> axis.</li><li>• 'auto' — Uses 'data' for numeric grouping variables and 'list' for strings.</li></ul>



Name	Value
'fullfactors'	<ul style="list-style-type: none"> <li>• 'off' — One group for each unique row of G. This is the default.</li> <li>• 'on' — Create a group for each possible combination of group variable values, including combinations that do not appear in the data.</li> </ul>
'factorseparator'	<p>Specifies which factors should have their values separated by a grid line. The value may be 'auto' or a vector of grouping variable numbers. For example, [1 2] adds a separator line when the first or second grouping variable changes value. 'auto' is [] for one grouping variable and [1] for two or more grouping variables. The default is [].</p>
'factorgap'	<p>Specifies an extra gap to leave between boxes when the corresponding grouping factor changes value, expressed as a percentage of the width of the plot. For example, with [3 1], the gap is 3% of the width of the plot between groups with different values of the first grouping variable, and 1% between groups with the same value of the first grouping variable but different values for the second. 'auto' specifies that boxplot should choose a gap automatically. The default is [].</p>
'grouporder'	<p>Order of groups for plotting, specified as a cell array of strings. With multiple grouping variables, separate values within each string with a comma. Using categorical arrays as grouping variables is an easier way to control the order of the boxes. The default is [], which does not reorder the boxes.</p>

# boxplot

Name	Value
'jitter'	Maximum distance $d$ to displace outliers along the factor axis by a uniform random amount, in order to make duplicate points visible. A $d$ of 1 makes the jitter regions just touch between the closest adjacent groups. The default is 0.
'labels'	<p>A character array, cell array of strings, or numeric vector of box labels. There may be one label per group or one label per X value. Multiple label variables may be specified via a numeric matrix or a cell array containing any of these types.</p> <hr/> <p><b>Tip</b> To remove labels from a plot, use the following command:</p> <pre>set(gca,'XTickLabel',{' '})</pre> <hr/>
'labelorientation'	<ul style="list-style-type: none"><li>• 'inline' — Rotates the labels to be vertical. This is the default when <code>plotstyle</code> is 'compact'.</li><li>• 'horizontal' — Leaves the labels horizontal. This is the default when <code>plotstyle</code> has the default value of 'traditional'.</li></ul> <p>When the labels are on the y axis, both settings leave the labels horizontal.</p>

Name	Value
'labelverbosity'	<ul style="list-style-type: none"> <li>• 'all' — Displays every label. This is the default.</li> <li>• 'minor' — Displays a label for a factor only when that factor has a different value from the previous group.</li> <li>• 'majorminor' — Displays a label for a factor when that factor or any factor major to it has a different value from the previous group.</li> </ul>
'medianstyle'	<ul style="list-style-type: none"> <li>• 'line' — Draws a line for the median. This is the default.</li> <li>• 'target' — Draws a black dot inside a white circle for the median.</li> </ul>
'notch'	<ul style="list-style-type: none"> <li>• 'on' — Draws comparison intervals using notches when plotstyle is 'traditional', or triangular markers when plotstyle is 'compact'.</li> <li>• 'marker' — Draws comparison intervals using triangular markers.</li> <li>• 'off' — Omits notches. This is the default.</li> </ul> <p>Two medians are significantly different at the 5% significance level if their intervals do not overlap. Interval endpoints are the extremes of the notches or the centers of the triangular markers. The extremes correspond to <math>q_2 - 1.57(q_3 - q_1)/\sqrt{n}</math> and <math>q_2 + 1.57(q_3 - q_1)/\sqrt{n}</math>, where <math>q_2</math> is the median (50th percentile), <math>q_1</math> and <math>q_3</math> are the 25th and 75th percentiles, respectively, and <math>n</math> is the number of observations without any NaN values. When</p>

# boxplot

Name	Value
	the sample size is small, notches may extend beyond the end of the box.
'orientation'	<ul style="list-style-type: none"><li>'vertical' — Plots <math>X</math> on the <math>y</math> axis. This is the default.</li><li>'horizontal' — Plots <math>X</math> on the <math>x</math> axis.</li></ul>
'outliersize'	Size of the marker used for outliers, in points. The default is 6 (6/72 inch).
'positions'	Box positions specified as a numeric vector with one entry per group or $X$ value. The default is $1:\text{numGroups}$ , where $\text{numGroups}$ is the number of groups.
'symbol'	Symbol and color to use for outliers, using the same values as the <code>LineStyle</code> parameter in <code>plot</code> . The default is 'r+'. If the symbol is omitted then the outliers are invisible; if the color is omitted then the outliers have the same color as their corresponding box.
'whisker'	Maximum whisker length $w$ . The default is a $w$ of 1.5. Points are drawn as outliers if they are larger than $q_3 + w(q_3 - q_1)$ or smaller than $q_1 - w(q_3 - q_1)$ , where $q_1$ and $q_3$ are the 25th and 75th percentiles, respectively. The default of 1.5 corresponds to approximately $\pm 2.7\sigma$ and 99.3 coverage if the data are normally distributed. The plotted whisker extends to the <i>adjacent value</i> , which is the most extreme data value that is not an outlier. Set <code>whisker</code> to 0 to give no whiskers and to make every point outside of $q_1$ and $q_3$ an outlier.
'widths'	A scalar or vector of box widths for when <code>boxstyle</code> is 'outline'. The default is half of the minimum separation between boxes,

Name	Value
	which is 0.5 when the <code>positions</code> argument takes its default value. The list of values is replicated or truncated as necessary.

When the `plotstyle` parameter takes the value `'compact'`, the following default values for other parameters apply.

Parameter	Default when <code>plotstyle</code> is <code>'compact'</code>
<code>'boxstyle'</code>	<code>'filled'</code>
<code>'factorseparator'</code>	<code>'auto'</code>
<code>'factorgap'</code>	<code>'auto'</code>
<code>'jitter'</code>	0.5
<code>'labelorientation'</code>	<code>'inline'</code>
<code>'labelverbosity'</code>	<code>'majorminor'</code>
<code>'medianstyle'</code>	<code>'target'</code>
<code>'outliersize'</code>	4
<code>'symbol'</code>	<code>'o'</code>

You can see data values and group names using the data cursor in the figure window. The cursor shows the original values of any points affected by the `datalim` parameter. You can label the group to which an outlier belongs using the `gname` function.

To modify graphics properties of a box plot component, use `findobj` with the `Tag` property to find the component's handle. `Tag` values for box plot components depend on parameter settings, and are listed in the table below.

# boxplot

---

Parameter Settings	Tag Values
All settings	<ul style="list-style-type: none"><li>'Box'</li><li>'Outliers'</li></ul>
When 'plotstyle' is 'traditional'	<ul style="list-style-type: none"><li>'Median'</li><li>'Upper Whisker'</li><li>'Lower Whisker'</li><li>'Upper Adjacent Value'</li><li>'Lower Adjacent Value'</li></ul>
When 'plotstyle' is 'compact'	<ul style="list-style-type: none"><li>'Whisker'</li><li>'MedianOuter'</li><li>'MedianInner'</li></ul>
When 'notch' is 'marker'	<ul style="list-style-type: none"><li>'NotchLo'</li><li>'NotchHi'</li></ul>

## Examples

### Create Box Plots for Grouped Data

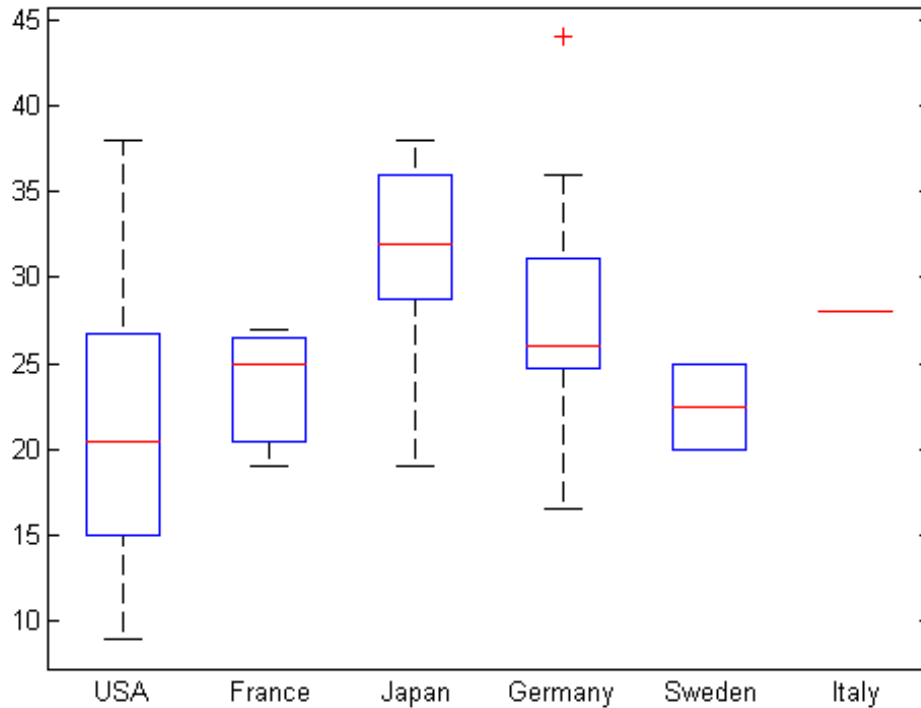
This example shows how to create box plots for grouped sample data.

Load the sample data.

```
load carsmall
```

Create a box plot of the miles per gallon (MPG) measurements from the sample data, grouped by the vehicles' country of origin, `Origin`.

```
boxplot(MPG,Origin)
```



## Create Notched Box Plots

This example shows how to create notched box plots.

Generate two sets of sample data. The first sample, `x1`, contains random numbers generated from a normal distribution with  $\mu = 5$  and  $\sigma = 1$ . The second sample, `x2`, contains random numbers generated from a normal distribution with  $\mu = 6$  and  $\sigma = 1$ .

```
rng('default'); % For reproducibility
```

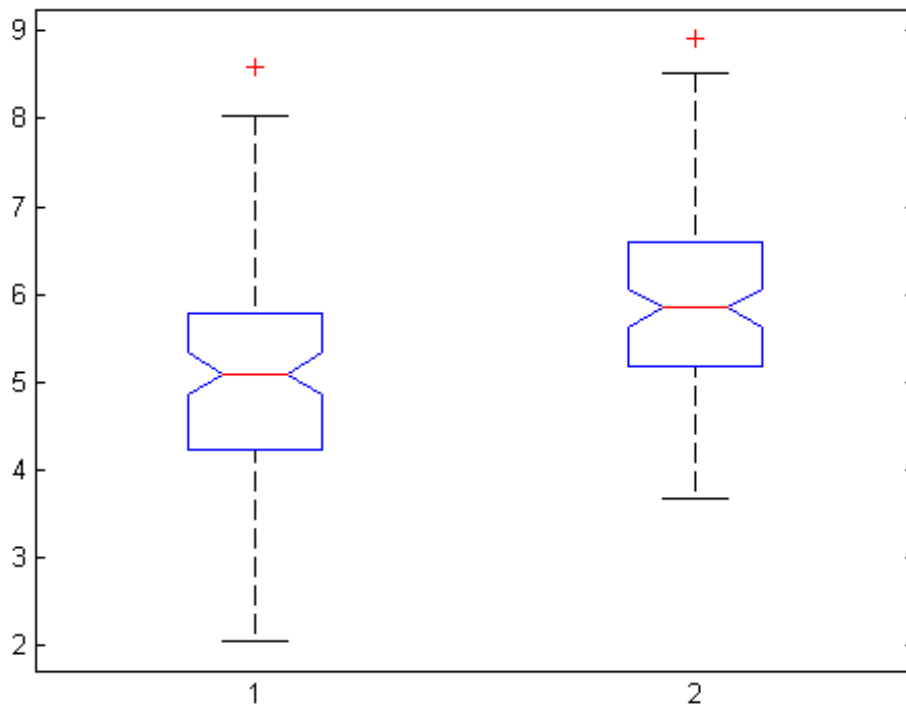
# boxplot

---

```
x1 = normrnd(5,1,100,1);  
x2 = normrnd(6,1,100,1);
```

Create notched box plots of x1 and x2.

```
boxplot([x1,x2], 'notch', 'on')
```

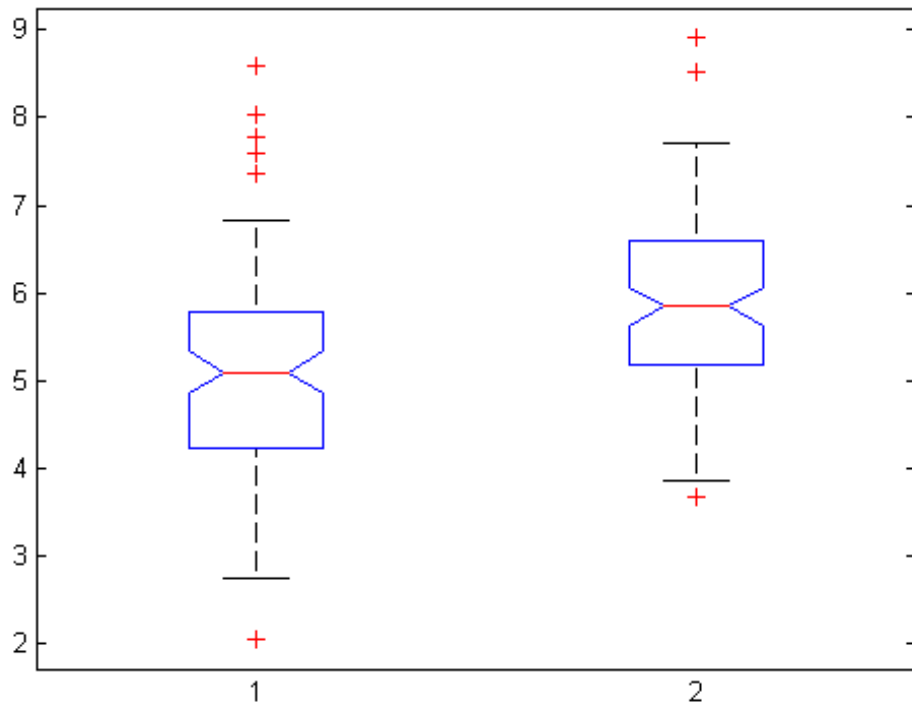


The difference between the medians of the two groups is approximately 1. Since the notches in the box plot do not overlap, you can conclude, with 95% confidence, that the true medians do differ.



The following figure shows the box plot for the same data with the length of the whiskers specified as 1.0 times the interquartile range. Points beyond the whiskers are displayed using +.

```
boxplot([x1,x2], 'notch', 'on', 'whisker', 1)
```



## Create Compact Box Plots

This example shows how to create box plots using compact formatting, which is useful when the sample data contains many groups.

# boxplot

---

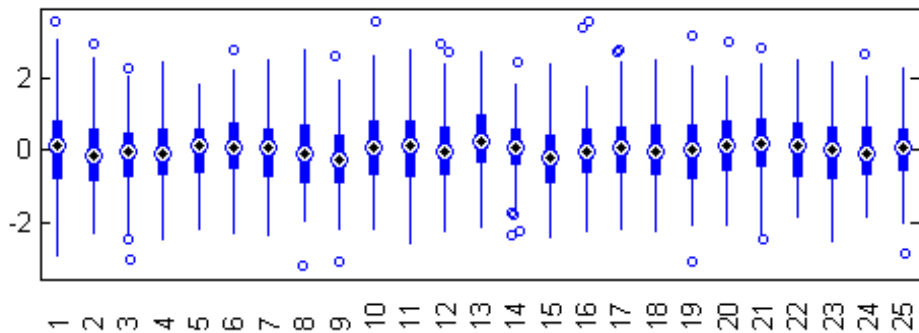
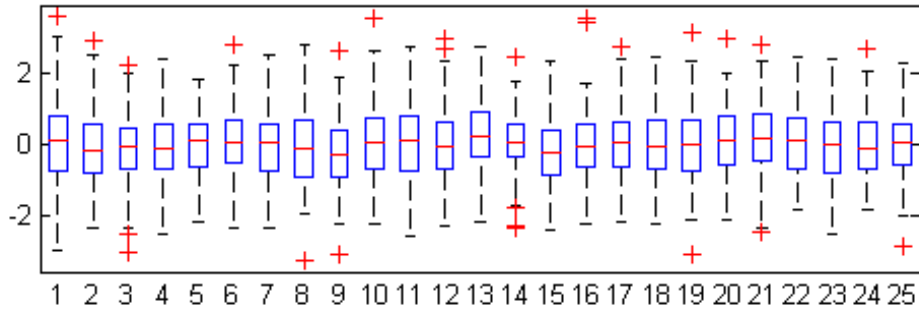
Create a 100-by-25 matrix of random numbers generated from a standard normal distribution to use as sample data.

```
rng('default'); % For reproducibility  
X = randn(100,25);
```

Create two box plots for the data in  $X$  on the same figure. The top plot uses the default box plot formatting. The bottom plot uses compact formatting.

```
subplot(2,1,1)  
boxplot(X)
```

```
subplot(2,1,2)  
boxplot(X,'plotstyle','compact')
```



**References**

- [1] McGill, R., J. W. Tukey, and W. A. Larsen. “Variations of Boxplots.” *The American Statistician*. Vol. 32, No. 1, 1978, pp. 12–16.
- [2] Velleman, P.F., and D.C. Hoaglin. *Applications, Basics, and Computing of Exploratory Data Analysis*. Pacific Grove, CA: Duxbury Press, 1981.
- [3] Nelson, L. S. “Evaluating Overlapping Confidence Intervals.” *Journal of Quality Technology*. Vol. 21, 1989, pp. 140–141.

# boxplot

---

## See Also

[anova1](#) | [axes\\_props](#) | [kruskalwallis](#) | [multcompare](#)

**Purpose**

Piecewise distribution boundaries

**Syntax**

```
[p,q] = boundary(obj)  
[p,q] = boundary(obj,i)
```

**Description**

[p,q] = boundary(obj) returns the boundary points between segments of the piecewise distribution object, obj. p is a vector of cumulative probabilities at each boundary. q is a vector of quantiles at each boundary.

[p,q] = boundary(obj,i) returns p and q for the *i*th boundary.

**Examples**

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);  
obj = paretotails(t,0.1,0.9);  
[p,q] = boundary(obj)  
p =  
    0.1000  
    0.9000  
q =  
   -1.7766  
    1.8432
```

**See Also**

paretotails | cdf | icdf | nsegments

# prob.BurrDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Burr probability distribution object

**Description** `prob.BurrDistribution` is an object consisting of parameters, a model description, and sample data for a Burr probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Burr')` creates a Burr probability distribution object using the default parameter values.

`pd = makedist('Burr','alpha',alpha,'c',c,'k',k)` creates a Burr probability distribution object using the specified parameter values.

## Input Arguments

### **alpha - Scale parameter**

1 (default) | positive scalar value

Scale parameter of the Burr distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **c - First shape parameter**

1 (default) | positive scalar value

First shape parameter of the Burr distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **k - Second shape parameter**

1 (default) | positive scalar value

Second shape parameter of the Burr distribution, specified as a positive scalar value.

**Data Types**

single | double

**Properties****alpha**

Scale parameter of the Burr distribution, stored as a positive scalar value.

**Data Types**

single | double

**c**

First shape parameter of the Burr distribution, stored as a positive scalar value.

**Data Types**

single | double

**k**

Second shape parameter of the Burr distribution, stored as a positive scalar value.

**Data Types**

single | double

**DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

**Data Types**

char

**InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.

# prob.BurrDistribution

---

- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value.

If IsTruncated equals 0, the distribution is not truncated.

If IsTruncated equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution.

The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription



Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the ParameterNames array is not fixed. If 1, the corresponding parameter in the ParameterNames array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

# prob.BurrDistribution

---

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Burr Distribution

The Burr distribution is a three-parameter family of distributions on the positive real line. It can fit a wide range of empirical data, and is used in various fields such as finance, hydrology, and reliability to model a variety of data types.

The Burr distribution uses the following parameters.

Parameter	Description	Support
alpha	Scale parameter	$\alpha > 0$
c	First shape parameter	$c > 0$
k	Second shape parameter	$k > 0$

The probability density function (pdf) is

$$f(x | \alpha, c, k) = \frac{\frac{kc}{\alpha} \left(\frac{x}{\alpha}\right)^{c-1}}{\left(1 + \left(\frac{x}{\alpha}\right)^c\right)^{k+1}} ; \quad x > 0.$$

## Examples

### Create a Burr Distribution Object Using Default Parameters

Create a Burr distribution object using the default parameter values.

```
pd = makedist('Burr')
```

```
pd =
```

```
BurrDistribution
```

# prob.BurrDistribution

---

```
Burr distribution
alpha = 1
c = 1
k = 1
```

## Create a Burr Distribution Object Using Specified Parameters

Create a Burr distribution object by specifying parameter values.

```
pd = makedist('Burr','alpha',1,'c',2,'k',5)
```

```
pd =
```

```
BurrDistribution

Burr distribution
alpha = 1
c = 2
k = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
0.4295
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Burr Type XII Distribution” on page B-15
- Class Attributes
- Property Attributes

# clustering.evaluation.CalinskiHarabaszEvaluation

---

**Superclasses** ClusterCriterion

**Purpose** Calinski-Harabasz criterion clustering evaluation object

**Description** clustering.evaluation.CalinskiHarabaszEvaluation is an object consisting of sample data, clustering data, and Calinski-Harabasz criterion values used to evaluate the optimal number of clusters. Create a Calinski-Harabasz criterion clustering evaluation object using evalclusters.

**Construction** `eva = evalclusters(x,clust,'CalinskiHarabasz')` creates a Calinski-Harabasz criterion clustering evaluation object.

`eva = evalclusters(x,clust,'CalinskiHarabasz',Name,Value)` creates a Calinski-Harabasz criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

### **x - Input data**

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

### **Data Types**

single | double

### **clust - Clustering algorithm**

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

# clustering.evaluation.CalinskiHarabaszEvaluation

---

'kmeans'	Cluster the data in $X$ using the kmeans clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in $X$ using the clusterdata agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in $X$ using the gmdistribution Gaussian mixture distribution algorithm, with 'SharedCov' set to true and 'Replicates' set to 5.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using the function\_handle (@) operator. The function must be of the form

```
C = clustfun(DATA,K)
```

where DATA is the data to be clustered, and K is the number of clusters. The output of clustfun must be one of the following:

- A vector of integers representing the cluster index for each observation in DATA. There must be K unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify clust as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'KList',[1:6]` specifies to test 1, 2, 3, 4, 5, and 6 clusters to find the optimal number.

## **KList** - List of number of clusters to evaluate

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of `'KList'` and a vector of positive integer values. You must specify `KList` when `clust` is a clustering algorithm name string or a function handle. When `criterion` is `'gap'`, `clust` must be a string or a function handle, and you must specify `KList`.

**Example:** `'KList',[1:6]`

## Properties

### **ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name string or function handle. If the clustering solutions are provided in the input, `ClusteringFunction` is empty.

### **CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name string.

### **CriterionValues**

Criterion values corresponding to each proposed number of clusters in `InspectedK`, stored as a vector of numerical values.

### **Distance**

# clustering.evaluation.CalinskiHarabaszEvaluation

---

Distance measure used for clustering data, stored as a valid distance measure name string.

## **InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

## **Missing**

Logical flag for excluded data, stored as a column vector of logical values. If `Missing` equals `true`, then the corresponding value in the data matrix `x` is not used in the clustering solution.

## **NumObservations**

Number of observations in the data matrix `X`, minus the number of missing (NaN) values in `X`, stored as a positive integer value.

## **OptimalK**

Optimal number of clusters, stored as a positive integer value.

## **OptimalY**

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

## **X**

Data used for clustering, stored as a matrix of numerical values.

## **Methods**

### **Inherited Methods**

<code>compact</code>	Compact clustering evaluation object
<code>plot</code>	Plot clustering evaluation object criterion values



## Definitions

### Calinski-Harabasz Criterion

The Calinski-Harabasz criterion is sometimes called the variance ratio criterion (VRC). The Calinski-Harabasz index is defined as

$$VRC_k = \frac{SS_B}{SS_W} \times \frac{(N - k)}{(k - 1)},$$

, where  $SS_B$  is the overall between-cluster variance,  $SS_W$  is the overall within-cluster variance,  $k$  is the number of clusters, and  $N$  is the number of observations.

The overall between-cluster variance  $SS_B$  is defined as

$$SS_B = \sum_{i=1}^k n_i \|m_i - m\|^2,$$

where  $k$  is the number of clusters,  $m_i$  is the centroid of cluster  $i$ , and  $m$  is the overall mean of the sample data.

The overall within-cluster variance  $SS_W$  is defined as

$$SS_W = \sum_{i=1}^k \sum_{x \in c_i} \|x - m_i\|^2,$$

where  $k$  is the number of clusters,  $x$  is a data point,  $c_i$  is the  $i$ th cluster, and  $m_i$  is the centroid of cluster  $i$ .

Well-defined clusters have a large between-cluster variance ( $SS_B$ ) and a small within-cluster variance ( $SS_W$ ). The larger the  $VRC_k$  ratio, the better the data partition. To determine the optimal number of clusters, maximize  $VRC_k$  with respect to  $k$ . The optimal number of clusters is the solution with the highest Calinski-Harabasz index value.

The Calinski-Harabasz criterion is best suited for  $k$ -means clustering solutions with squared Euclidean distances.

# clustering.evaluation.CalinskiHarabaszEvaluation

---

## Examples

### Evaluate the Clustering Solution Using Calinski-Harabasz Criterion

Evaluate the optimal number of clusters using the Calinski-Harabasz clustering evaluation criterion.

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Evaluate the optimal number of clusters using the Calinski-Harabasz criterion. Cluster the data using kmeans.

```
rng('default'); % For reproducibility  
eva = evalclusters(meas,'kmeans','CalinskiHarabasz','KList',[1:6])
```

```
eva =
```

```
CalinskiHarabaszEvaluation with properties:
```

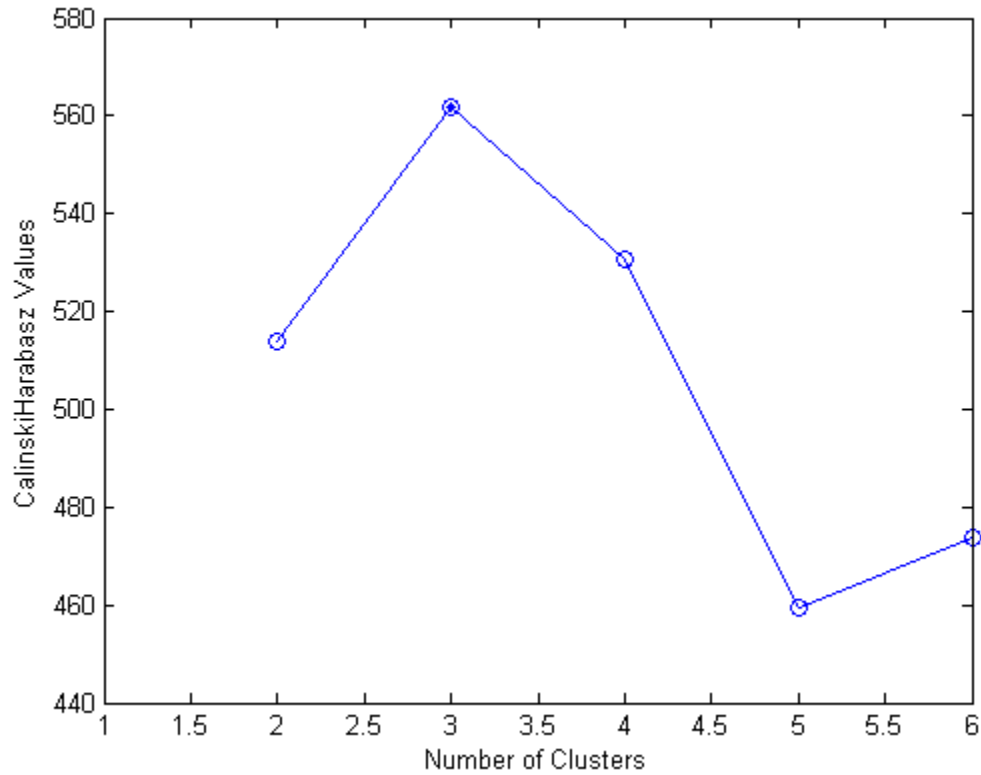
```
NumObservations: 150  
  InspectedK: [1 2 3 4 5 6]  
 CriterionValues: [1x6 double]  
   OptimalK: 3
```

The OptimalK value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

Plot the Calinski-Harabasz criterion values for each number of clusters tested.

```
figure;  
plot(eva);
```

# clustering.evaluation.CalinskiHarabaszEvaluation



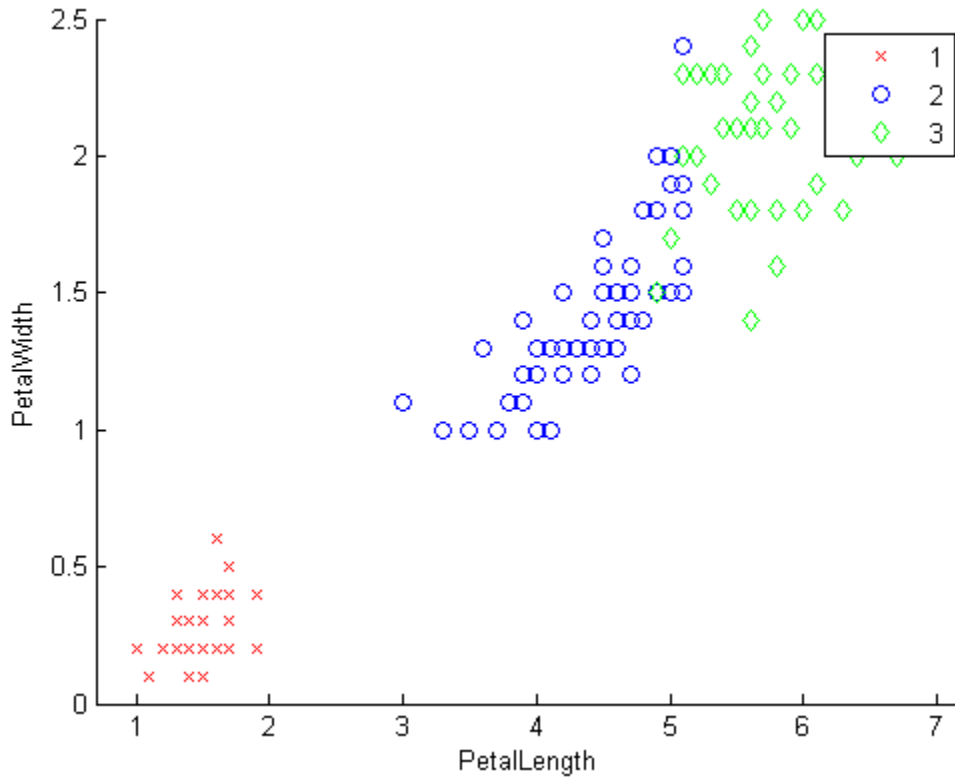
The plot shows that the highest Calinski-Harabasz value occurs at three clusters, suggesting that the optimal number of clusters is three.

Create a grouped scatter plot to examine the relationship between petal length and width. Group the data by suggested clusters.

```
PetalLength = meas(:,3);  
PetalWidth = meas(:,4);  
ClusterGroup = eva.OptimalY;  
figure;
```

# clustering.evaluation.CalinskiHarabaszEvaluation

```
gscatter(PetalLength,PetalWidth,ClusterGroup,'rbg','xod');
```



The plot shows cluster 1 in the lower-left corner, completely separated from the other two clusters. Cluster 1 contains flowers with the smallest petal widths and lengths. Cluster 3 is in the upper-right corner, and contains flowers with the largest petal widths and lengths. Cluster 2 is near the center of the plot, and contains flowers with measurements between these two extremes.

## References

[1] Calinski, T., and J. Harabasz. "A dendrite method for cluster analysis." *Communications in Statistics*. Vol. 3, No. 1, 1974, pp. 1–27.

## See Also

evalclustersclustering.evaluation.DaviesBouldinEvaluation  
| clustering.evaluation.GapEvaluation |  
clustering.evaluation.SilhouetteEvaluation |

## Concepts

- Class Attributes
- Property Attributes

# candexch

---

**Purpose** *D*-optimal design from candidate set using row exchanges

**Syntax**  
`rlist = candexch(C,nrows)`  
`rlist = candexch(C,nrows,Name,Value)`

**Description**  
`rlist = candexch(C,nrows)` uses a row-exchange algorithm to select a *D*-optimal design from the candidate set *C*.  
`rlist = candexch(C,nrows,Name,Value)` generates a *D*-optimal design with additional options specified by one or more *Name,Value* pair arguments.

**Input Arguments**  
**C**  
N-by-P matrix containing the values of P model terms at each of N points.

**nrows**

The desired number of rows in the design.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of *Name,Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as *Name1,Value1,...,NameN,ValueN*.

**'display'**

When 'on', displays iteration number. Disable the display by setting to 'off'.

**Default:** 'on', except when the `UseParallel` option is true

**'init'**

nrows-by-P matrix giving an initial design.

**Default:** A random subset of the rows of *C*

**'maxiter'**

Maximum number of iterations, a positive integer.

**Default:** 10

**'options'**

A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the options structure with `statset`. Option fields:

- `UseParallel` — Set to `true` to compute in parallel. Default is `false`.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. Default is `false`. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`.
- `Streams` — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `candexch` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open Parallel pool
  - `UseParallel` is `true`
  - `UseSubstreams` is `false`In that case, use a cell array the same size as the Parallel pool.

**Default:** []

**'start'**

An `nobs-by-p` matrix of factor settings, specifying a set of `nobs` fixed design points to include in the design. `candexch` finds `nrows` additional rows to add to the `start` design. The parameter provides the same functionality as the `daugment` function, using a row-exchange algorithm rather than a coordinate-exchange algorithm.

**Default:** []

## 'tries'

Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first.

**Default:** 1

## Output Arguments

### rlist

Vector of length nrows listing the selected rows.

## Examples

This example shows how to generate a  $D$ -optimal design when there is a restriction on the candidate set, so the rowexch function isn't appropriate.

```
F = (fullfact([5 5 5])-1)/4; % factor settings in unit cube
T = sum(F,2)<=1.51;        % find rows matching a restriction
F = F(T,:);               % take only those rows
C = [ones(size(F,1),1) F F.^2];
                           % compute model terms including
                           % a constant and all squared terms
R = candexch(C,12);       % find a D-optimal 12-point subset
X = F(R,:);               % get factor settings
```

## Algorithms

candexch selects a starting design  $X$  at random, and uses a row-exchange algorithm to iteratively replace rows of  $X$  by rows of  $C$  in an attempt to improve the determinant of  $X' * X$ .

## Alternatives

The rowexch function also generates  $D$ -optimal designs using a row-exchange algorithm, but it automatically generates a candidate set that is appropriate for a specified model. The daugment function augments a set of fixed design points using a coordinate-exchange algorithm; the 'start' parameter provides the same functionality using the row exchange algorithm.

## See Also

candgen | rowexch | cordexch | daugment | x2fx



**Tutorials**

- “Specify Candidate Sets” on page 18-21

**How To**

- “D-Optimal Designs” on page 18-15

**Purpose** Candidate set generation

**Syntax**

```
dC = candgen(nfactors, 'model')  
[dC,C] = candgen(nfactors, 'model')  
[...] = candgen(nfactors, 'model', 'Name', value)
```

**Description** `dC = candgen(nfactors, 'model')` generates a candidate set `dC` of treatments appropriate for estimating the parameters in the `model` with `nfactors` factors. `dC` has `nfactors` columns and one row for each candidate treatment. `model` is one of the following strings, specified inside single quotes:

- `linear` — Constant and linear terms. This is the default.
- `interaction` — Constant, linear, and interaction terms
- `quadratic` — Constant, linear, interaction, and squared terms
- `purequadratic` — Constant, linear, and squared terms

Alternatively, `model` can be a matrix specifying polynomial terms of arbitrary order. In this case, `model` should have one column for each factor and one row for each term in the model. The entries in any row of `model` are powers for the factors in the columns. For example, if a model has factors `X1`, `X2`, and `X3`, then a row `[0 1 2]` in `model` specifies the term  $(X1.^0) .* (X2.^1) .* (X3.^2)$ . A row of all zeros in `model` specifies a constant term, which can be omitted.

`[dC,C] = candgen(nfactors, 'model')` also returns the design matrix `C` evaluated at the treatments in `dC`. The order of the columns of `C` for a full quadratic model with `n` terms is:

- 1 The constant term
- 2 The linear terms in order 1, 2, ...,  $n$
- 3 The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ )
- 4 The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Pass `C` to `candexch` to generate a  $D$ -optimal design using a coordinate-exchange algorithm.

`[...] = candgen(nfactors, 'model', 'Name', value)` specifies one or more optional name/value pairs for the design. Valid parameters and their values are listed in the following table. Specify *Name* inside single quotes.

Name	Value
bounds	Lower and upper bounds for each factor, specified as a 2-by- <code>nfactors</code> matrix. Alternatively, this value can be a cell array containing <code>nfactors</code> elements, each element specifying the vector of allowable values for the corresponding factor.
categorical	Indices of categorical predictors.
levels	Vector of number of levels for each factor.

---

**Note** The `rowexch` function automatically generates a candidate set using `candgen`, and then creates a  $D$ -optimal design from that candidate set using `candexch`. Call `candexch` separately to specify your own candidate set to the row-exchange algorithm.

---

## Examples

The following example uses `rowexch` to generate a five-run design for a two-factor pure quadratic model using a candidate set that is produced internally:

```
dRE1 = rowexch(2,5,'purequadratic','tries',10)
dRE1 =
    -1     1
     0     0
     1    -1
     1     0
```

# candgen

---

```
1 1
```

The same thing can be done using `candgen` and `candexch` in sequence:

```
[dC,C] = candgen(2,'purequadratic') % Candidate set, C
dC =
-1 -1
 0 -1
 1 -1
-1  0
 0  0
 1  0
-1  1
 0  1
 1  1
C =
 1 -1 -1  1  1
 1  0 -1  0  1
 1  1 -1  1  1
 1 -1  0  1  0
 1  0  0  0  0
 1  1  0  1  0
 1 -1  1  1  1
 1  0  1  0  1
 1  1  1  1  1
treatments = candexch(C,5,'tries',10) % Find D-opt subset
treatments =
 2
 1
 7
 3
 4
dRE2 = dC(treatments,:) % Display design
dRE2 =
 0 -1
-1 -1
-1  1
```

$$\begin{array}{cc} 1 & -1 \\ -1 & 0 \end{array}$$

**See Also**

candexch | rowexch

## Purpose

Canonical correlation

## Syntax

```
[A,B] = canoncorr(X,Y)
[A,B,r] = canoncorr(X,Y)
[A,B,r,U,V] = canoncorr(X,Y)
[A,B,r,U,V,stats] = canoncorr(X,Y)
```

## Description

[A,B] = canoncorr(X,Y) computes the sample canonical coefficients for the n-by-d1 and n-by-d2 data matrices X and Y. X and Y must have the same number of observations (rows) but can have different numbers of variables (columns). A and B are d1-by-d and d2-by-d matrices, where  $d = \min(\text{rank}(X), \text{rank}(Y))$ . The jth columns of A and B contain the canonical coefficients, i.e., the linear combination of variables making up the jth canonical variable for X and Y, respectively. Columns of A and B are scaled to make the covariance matrices of the canonical variables the identity matrix (see U and V below). If X or Y is less than full rank, canoncorr gives a warning and returns zeros in the rows of A or B corresponding to dependent columns of X or Y.

[A,B,r] = canoncorr(X,Y) also returns a 1-by-d vector containing the sample canonical correlations. The jth element of r is the correlation between the jth columns of U and V (see below).

[A,B,r,U,V] = canoncorr(X,Y) also returns the canonical variables, scores. U and V are n-by-d matrices computed as

```
U = (X-repmat(mean(X),N,1))*A
V = (Y-repmat(mean(Y),N,1))*B
```

[A,B,r,U,V,stats] = canoncorr(X,Y) also returns a structure stats containing information relating to the sequence of d null

hypotheses  $H_0^{(k)}$ , that the (k+1)st through dth correlations are all zero, for  $k = 0:(d-1)$ . stats contains seven fields, each a 1-by-d vector with elements corresponding to the values of k, as described in the following table:

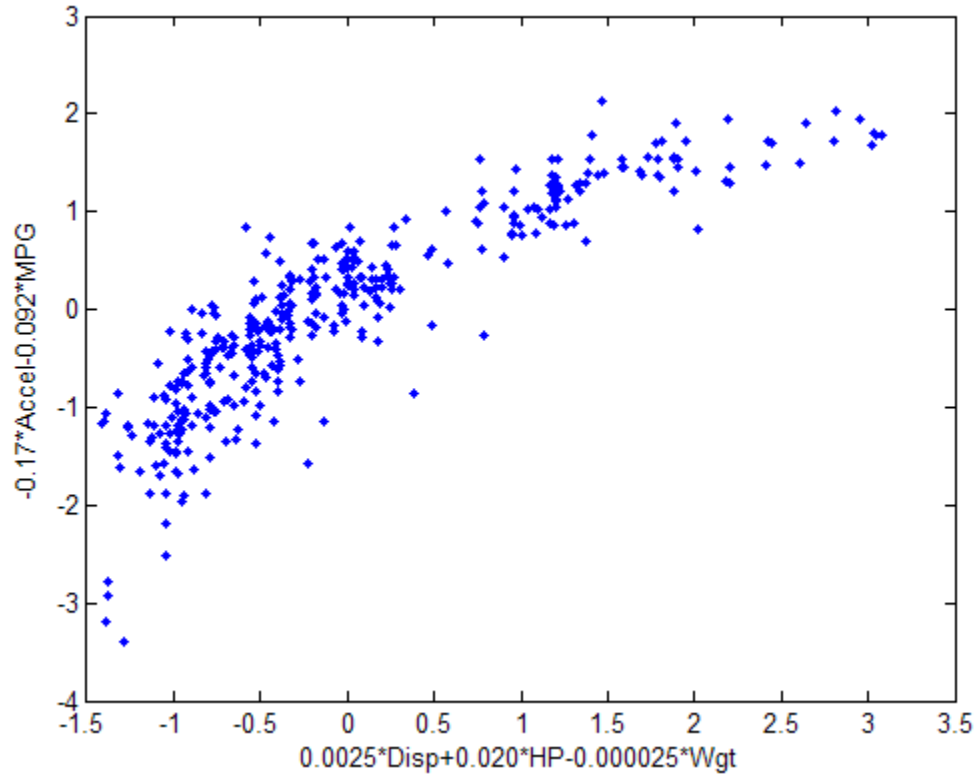
Field	Description
Wilks	Wilks' lambda (likelihood ratio) statistic
df1	Degrees of freedom for the chi-squared statistic, and the numerator degrees of freedom for the $F$ statistic
df2	Denominator degrees of freedom for the $F$ statistic
F	Rao's approximate $F$ statistic for $H_0^{(k)}$
pF	Right-tail significance level for F
chisq	Bartlett's approximate chi-squared statistic for $H_0^{(k)}$ with Lawley's modification
pChisq	Right-tail significance level for chisq

stats has two other fields (dfe and p) which are equal to df1 and pChisq, respectively, and exist for historical reasons.

### Examples

```
load carbig;
X = [Displacement Horsepower Weight Acceleration MPG];
nans = sum(isnan(X),2) > 0;
[A B r U V] = canoncorr(X(~nans,1:3),X(~nans,4:5));

plot(U(:,1),V(:,1),'.')
xlabel('0.0025*Disp+0.020*HP-0.000025*Wgt')
ylabel('-0.17*Accel-0.092*MPG')
```



## References

[1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

[2] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

manova1 | pca



**Purpose** Process capability indices

**Syntax** `S = capability(data,specs)`

**Description** `S = capability(data,specs)` estimates capability indices for measurements in `data` given the specifications in `specs`. `data` can be either a vector or a matrix of measurements. If `data` is a matrix, indices are computed for the columns. `specs` can be either a two-element vector of the form `[L,U]` containing lower and upper specification limits, or (if `data` is a matrix) a two-row matrix with the same number of columns as `data`. If there is no lower bound, use `-Inf` as the first element of `specs`. If there is no upper bound, use `Inf` as the second element of `specs`.

The output `S` is a structure with the following fields:

- `mu` — Sample mean
- `sigma` — Sample standard deviation
- `P` — Estimated probability of being within limits
- `P1` — Estimated probability of being below `L`
- `Pu` — Estimated probability of being above `U`
- `Cp` —  $(U-L)/(6*\text{sigma})$
- `Cp1` —  $(\text{mu}-L)/(3.*\text{sigma})$
- `Cpu` —  $(U-\text{mu})/(3.*\text{sigma})$
- `Cpk` —  $\min(\text{Cp1},\text{Cpu})$

Indices are computed under the assumption that data values are independent samples from a normal population with constant mean and variance.

Indices divide a “specification width” (between specification limits) by a “process width” (between control limits). Higher ratios indicate a process with fewer measurements outside of specification.

## Examples

### Compute Capability Indices

Simulate a sample from a process with a mean of 3 and a standard deviation of 0.005.

```
rng('default') % for reproducibility
data = normrnd(3,0.005,100,1);
```

Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99.

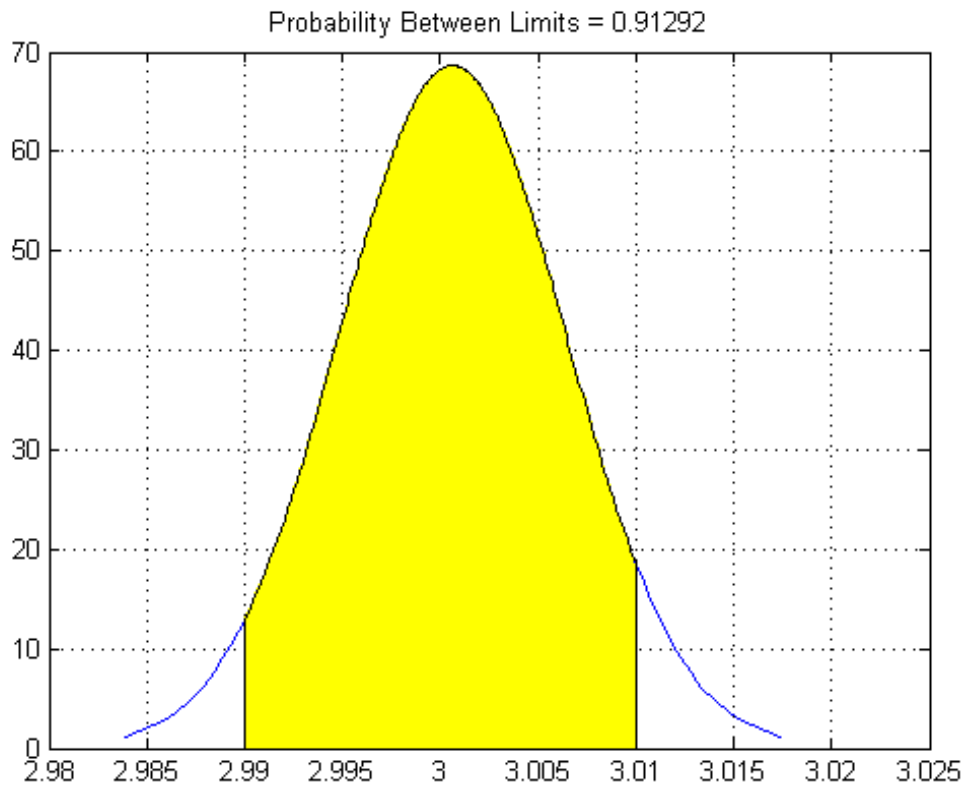
```
S = capability(data,[2.99 3.01])
```

```
S =
```

```
    mu: 3.0006
   sigma: 0.0058
      P: 0.9129
     Pl: 0.0339
     Pu: 0.0532
      Cp: 0.5735
     Cpl: 0.6088
     Cpu: 0.5382
     Cpk: 0.5382
```

Visualize the specification and process widths.

```
capaplot(data,[2.99 3.01]);
grid on
```



**References**

[1] Montgomery, D. *Introduction to Statistical Quality Control*. Hoboken, NJ: John Wiley & Sons, 1991, pp. 369–374.

**See Also**

capaplot | histfit

# capaplot

---

**Purpose** Process capability plot

**Syntax** `p = capaplot(data,specs)`  
`[p,h] = capaplot(data,specs)`

**Description** `p = capaplot(data,specs)` estimates the mean and variance for the observations in input vector `data`, and plots the pdf of the resulting T distribution. The observations in `data` are assumed to be normally distributed. The output, `p`, is the probability that a new observation from the estimated distribution will fall within the range specified by the two-element vector `specs`. The portion of the distribution between the lower and upper bounds specified in `specs` is shaded in the plot.

`[p,h] = capaplot(data,specs)` additionally returns handles to the plot elements in `h`.

`capaplot` treats NaN values in `data` as missing, and ignores them.

**Examples** Simulate a sample from a process with a mean of 3 and a standard deviation of 0.005:

```
data = normrnd(3,0.005,100,1);
```

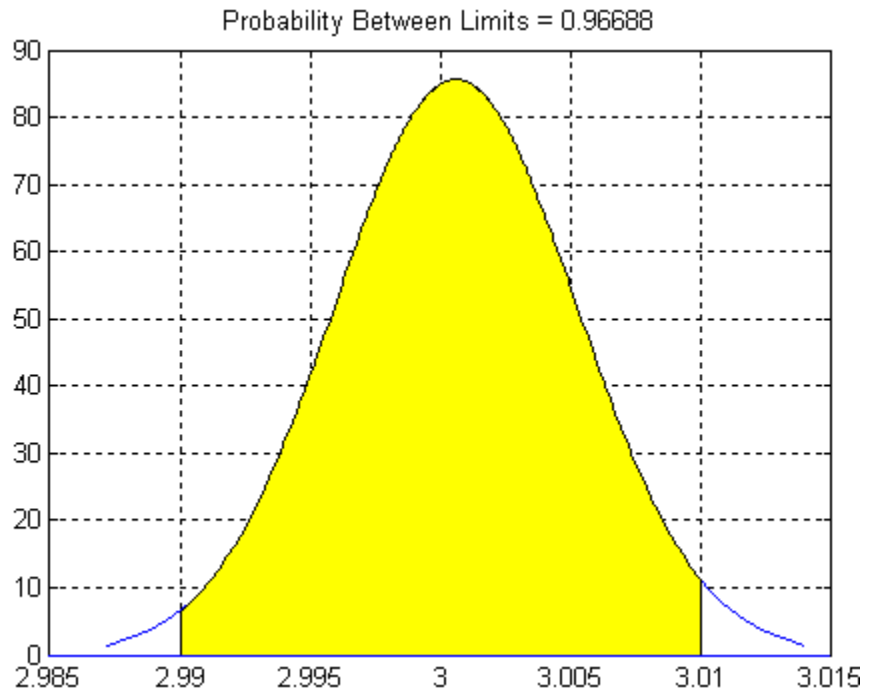
Compute capability indices if the process has an upper specification limit of 3.01 and a lower specification limit of 2.99:

```
S = capability(data,[2.99 3.01])
```

```
S =  
    mu: 3.0006  
   sigma: 0.0047  
    P: 0.9669  
   Pl: 0.0116  
   Pu: 0.0215  
    Cp: 0.7156  
   Cpl: 0.7567  
   Cpu: 0.6744  
   Cpk: 0.6744
```

Visualize the specification and process widths:

```
capaplot(data,[2.99 3.01]);  
grid on
```



### See Also

capability | histfit

# caseread

---

**Purpose** Read case names from file

**Syntax** `names = caseread('filename')`  
`names = caseread`

**Description** `names = caseread('filename')` reads the contents of *filename* and returns a string matrix of names. *filename* is the name of a file in the current folder, or the complete path name of any file elsewhere. `caseread` treats each line as a separate case.

`names = caseread` displays the **Select File to Open** dialog box for interactive selection of the input file.

**Examples** Read the file `months.dat` created using the `casewrite` function.

```
type months.dat
```

```
January  
February  
March  
April  
May
```

```
names = caseread('months.dat')  
names =  
January  
February  
March  
April  
May
```

**See Also** `casewrite` | `gname` | `tdfread` | `tblread`

---

<b>Purpose</b>	Write case names to file
<b>Syntax</b>	<pre>casewrite(strmat, 'filename') casewrite(strmat)</pre>
<b>Description</b>	<p><code>casewrite(strmat, 'filename')</code> writes the contents of string matrix <code>strmat</code> to <code>filename</code>. Each row of <code>strmat</code> represents one case name. <code>filename</code> is the name of a file in the current folder, or the complete path name of any file elsewhere. <code>casewrite</code> writes each name to a separate line in <code>filename</code>.</p> <p><code>casewrite(strmat)</code> displays the <b>Select File to Write</b> dialog box for interactive specification of the output file.</p>
<b>Examples</b>	<pre>strmat = char('January', 'February', ...               'March', 'April', 'May')  strmat = January February March April May  casewrite(strmat, 'months.dat') type months.dat  January February March April May</pre>
<b>See Also</b>	<code>gname</code>   <code>caseread</code>   <code>tblwrite</code>   <code>tdfread</code>

# categorical.cat

---

**Purpose** Concatenate categorical arrays

**Syntax** `c = cat(dim,A,B,...)`

**Description** `c = cat(dim,A,B,...)` concatenates the categorical arrays `A,B,...` along dimension `dim`. All inputs must have the same size except along dimension `dim`. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs, as determined by their labels.

**See Also** `cat` | `horzcat` | `vertcat`



**Purpose** Arrays for categorical data

**Description** In Statistics Toolbox, `categorical` is an abstract class, and you cannot create instances of it directly. You must create `nominal` or `ordinal` arrays. For the MATLAB categorical data type, see `categorical`.

Categorical arrays store data with values in a discrete set of levels. Each level is meant to capture a single, defining characteristic of an observation. If you do not encode ordering in the levels, the data and the array are `nominal`. If you do encode an ordering, the data and the array are `ordinal`.

## Methods

<code>addlevels</code>	Add levels to categorical array
<code>cat</code>	Concatenate categorical arrays
<code>cellstr</code>	Convert categorical array to cell array of strings
<code>char</code>	Convert categorical array to character array
<code>circshift</code>	Shift categorical array circularly
<code>ctranspose</code>	Transpose categorical matrix
<code>disp</code>	Display categorical array
<code>display</code>	Display categorical array
<code>double</code>	Convert categorical array to double array
<code>droplevels</code>	Drop levels
<code>end</code>	Last index in indexing expression for categorical array
<code>flipdim</code>	Flip categorical array along specified dimension

# categorical

---

<code>fliplr</code>	Flip categorical matrix in left/right direction
<code>flipud</code>	Flip categorical matrix in up/down direction
<code>getlabels</code>	Access categorical array labels
<code>getlevels</code>	Get categorical array levels
<code>hist</code>	Plot histogram of categorical data
<code>horzcat</code>	Horizontal concatenation for categorical arrays
<code>int16</code>	Convert categorical array to signed 16-bit integer array
<code>int32</code>	Convert categorical array to signed 32-bit integer array
<code>int64</code>	Convert categorical array to signed 64-bit integer array
<code>int8</code>	Convert categorical array to signed 8-bit integer array
<code>intersect</code>	Set intersection for categorical arrays
<code>ipermute</code>	Inverse permute dimensions of categorical array
<code>isempty</code>	True for empty categorical array
<code>isequal</code>	True if categorical arrays are equal
<code>islevel</code>	Test for levels
<code>ismember</code>	True for elements of categorical array in set
<code>isscalar</code>	True if categorical array is scalar
<code>isundefined</code>	Test for undefined elements

<code>isvector</code>	True if categorical array is vector
<code>length</code>	Length of categorical array
<code>levelcounts</code>	Element counts by level
<code>mergelevels</code>	Merge levels
<code>ndims</code>	Number of dimensions of categorical array
<code>numel</code>	Number of elements in categorical array
<code>permute</code>	Permute dimensions of categorical array
<code>reorderlevels</code>	Reorder levels
<code>repmat</code>	Replicate and tile categorical array
<code>reshape</code>	Resize categorical array
<code>rot90</code>	Rotate categorical matrix 90 degrees
<code>setdiff</code>	Set difference for categorical arrays
<code>setlabels</code>	Label levels
<code>setxor</code>	Set exclusive-or for categorical arrays
<code>shiftdim</code>	Shift dimensions of categorical array
<code>single</code>	Convert categorical array to single array
<code>size</code>	Size of categorical array
<code>squeeze</code>	Squeeze singleton dimensions from categorical array

# categorical

---

<code>subsasgn</code>	Subscripted assignment for categorical array
<code>subsindex</code>	Subscript index for categorical array
<code>subref</code>	Subscripted reference for categorical array
<code>times</code>	Product of categorical arrays
<code>transpose</code>	Transpose categorical matrix
<code>uint16</code>	Convert categorical array to unsigned 16-bit integers
<code>uint32</code>	Convert categorical array to unsigned 32-bit integers
<code>uint64</code>	Convert categorical array to unsigned 64-bit integers
<code>uint8</code>	Convert categorical array to unsigned 8-bit integers
<code>union</code>	Set union for categorical arrays
<code>unique</code>	Unique values in categorical array
<code>vertcat</code>	Vertical concatenation for categorical arrays

## Properties

<code>labels</code>	Text labels for levels
<code>undeflabel</code>	Text label for undefined levels

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

**How To**

- “Categorical Arrays” on page 2-41

# categorical

---

**Purpose** Create categorical array

**Description** In Statistics Toolbox, `categorical` is an abstract class, and you cannot create instances of it directly. You must create `nominal` or `ordinal` arrays. For the MATLAB categorical data type, see `categorical`.

**See Also** `nominal` | `ordinal` | `categorical`

# clustering.evaluation.DaviesBouldinEvaluation

---

**Superclasses** ClusterCriterion

**Purpose** Davies-Bouldin criterion clustering evaluation object

**Description** clustering.evaluation.DaviesBouldinEvaluation is an object consisting of sample data, clustering data, and Davies-Bouldin criterion values used to evaluate the optimal number of clusters. Create a Davies-Bouldin criterion clustering evaluation object using evalclusters.

**Construction** `eva = evalclusters(x,clust,'DaviesBouldin')` creates a Davies-Bouldin criterion clustering evaluation object.

`eva = evalclusters(x,clust,'DaviesBouldin',Name,Value)` creates a Davies-Bouldin criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

### **x - Input data**

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

### **Data Types**

single | double

### **clust - Clustering algorithm**

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

# clustering.evaluation.DaviesBouldinEvaluation

---

'kmeans'	Cluster the data in $X$ using the kmeans clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in $X$ using the clusterdata agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in $X$ using the gmdistribution Gaussian mixture distribution algorithm, with 'SharedCov' set to true and 'Replicates' set to 5.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using the function\_handle (@) operator. The function must be of the form

```
C = clustfun(DATA,K)
```

where DATA is the data to be clustered, and K is the number of clusters. The output of clustfun must be one of the following:

- A vector of integers representing the cluster index for each observation in DATA. There must be K unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify clust as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.



## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'KList',[1:5]` specifies to test 1, 2, 3, 4, and 5 clusters to find the optimal number.

## **KList - List of number of clusters to evaluate**

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of `'KList'` and a vector of positive integer values. You must specify `KList` when `clust` is a clustering algorithm name string or a function handle. When `criterion` is `'gap'`, `clust` must be a string or a function handle, and you must specify `KList`.

**Example:** `'KList',[1:6]`

## **Properties**

### **ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name string or function handle. If the clustering solutions are provided in the input, `ClusteringFunction` is empty.

### **CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name string.

### **CriterionValues**

Criterion values corresponding to each proposed number of clusters in `InspectedK`, stored as a vector of numerical values.

### **InspectedK**

# clustering.evaluation.DaviesBouldinEvaluation

---

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

## Missing

Logical flag for excluded data, stored as a column vector of logical values. If `Missing` equals `true`, then the corresponding value in the data matrix `x` is not used in the clustering solution.

## NumObservations

Number of observations in the data matrix `X`, minus the number of missing (NaN) values in `X`, stored as a positive integer value.

## OptimalK

Optimal number of clusters, stored as a positive integer value.

## OptimalY

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

## X

Data used for clustering, stored as a matrix of numerical values.

## Methods

### Inherited Methods

<code>compact</code>	Compact clustering evaluation object
<code>plot</code>	Plot clustering evaluation object criterion values

## Definitions

### Davies-Bouldin Criterion

The Davies-Bouldin criterion is based on a ratio of within-cluster and between-cluster distances. The Davies-Bouldin index is defined as

$$DB = \frac{1}{k} \sum_{i=1}^k \max_{j \neq i} \{D_{i,j}\},$$

where  $D_{i,j}$  is the within-to-between cluster distance ratio for the  $i$ th and  $j$ th clusters. In mathematical terms,

$$D_{i,j} = \frac{(\bar{d}_i + \bar{d}_j)}{d_{i,j}}.$$

$\bar{d}_i$  is the average distance between each point in the  $i$ th cluster and the centroid of the  $i$ th cluster.  $\bar{d}_j$  is the average distance between each point in the  $i$ th cluster and the centroid of the  $j$ th cluster.  $d_{i,j}$  is the Euclidean distance between the centroids of the  $i$ th and  $j$ th clusters.

The maximum value of  $D_{i,j}$  represents the worst-case within-to-between cluster ratio for cluster  $i$ . The optimal clustering solution has the smallest Davies-Bouldin index value.

## Examples

### Evaluate the Clustering Solution Using Davies-Bouldin Criterion

Evaluate the optimal number of clusters using the Davies-Bouldin clustering evaluation criterion.

Generate sample data containing random numbers from three multivariate distributions with different parameter values.

```
rng('default'); % For reproducibility
mu1 = [2 2];
sigma1 = [0.9 -0.0255; -0.0255 0.9];

mu2 = [5 5];
sigma2 = [0.5 0 ; 0 0.3];

mu3 = [-2, -2];
```

# clustering.evaluation.DaviesBouldinEvaluation

---

```
sigma3 = [1 0 ; 0 0.9];
```

```
N = 200;
```

```
X = [mvnrnd(mu1,sigma1,N);...  
     mvnrnd(mu2,sigma2,N);...  
     mvnrnd(mu3,sigma3,N)];
```

Evaluate the optimal number of clusters using the Davies-Bouldin criterion. Cluster the data using kmeans.

```
E = evalclusters(X,'kmeans','DaviesBouldin','klist',[1:6])
```

```
E =
```

```
DaviesBouldinEvaluation with properties:
```

```
NumObservations: 600
```

```
InspectedK: [1 2 3 4 5 6]
```

```
CriterionValues: [NaN 0.4663 0.4454 0.8300 0.7283 0.9199]
```

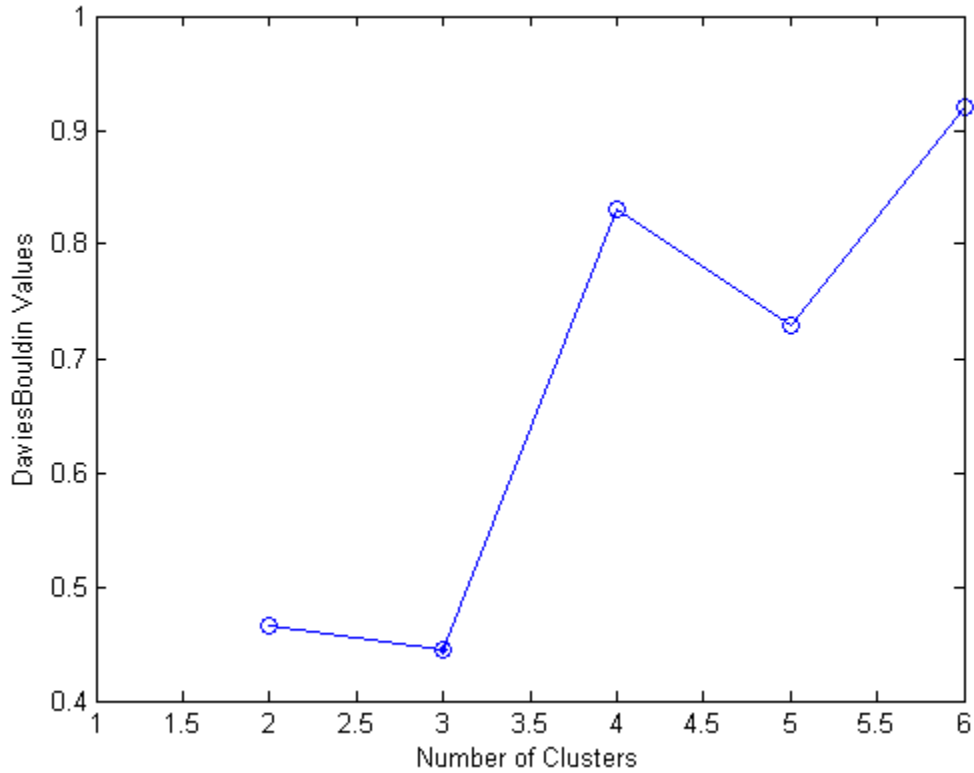
```
OptimalK: 3
```

The `OptimalK` value indicates that, based on the Davies-Bouldin criterion, the optimal number of clusters is three.

Plot the Davies-Bouldin criterion values for each number of clusters tested.

```
figure;  
plot(E)
```

# clustering.evaluation.DaviesBouldinEvaluation



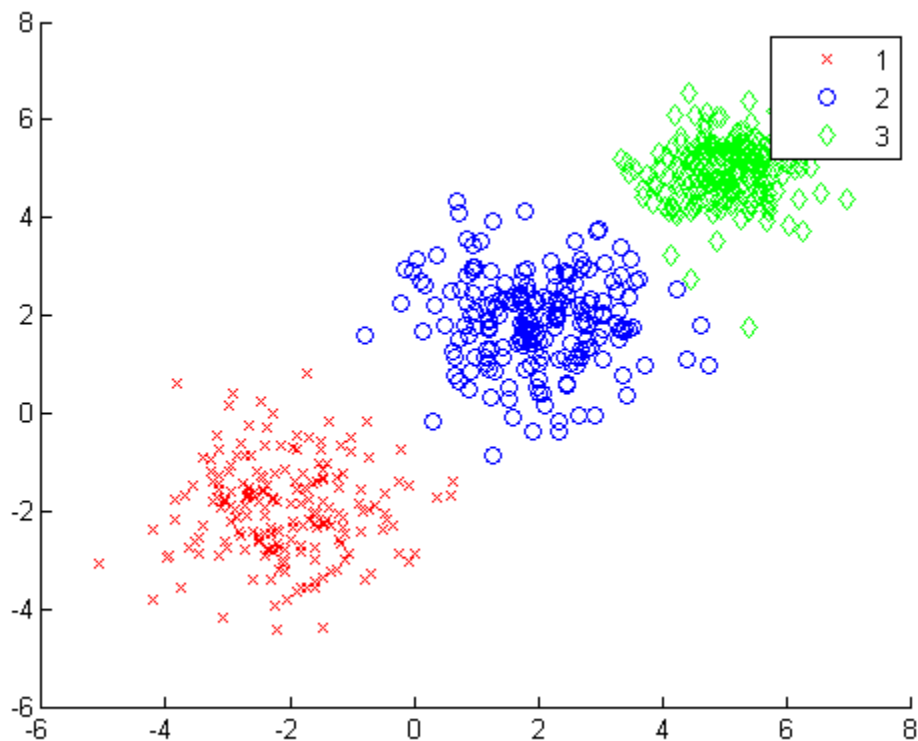
The plot shows that the lowest Davies-Bouldin value occurs at three clusters, suggesting that the optimal number of clusters is three.

Create a grouped scatter plot to visually examine the suggested clusters.

```
figure;  
gscatter(X(:,1),X(:,2),E.OptimalY,'rbg','xod')
```

# clustering.evaluation.DaviesBouldinEvaluation

---



The plot shows three distinct clusters within the data: Cluster 1 is in the lower-left corner, cluster 2 is near the center of the plot, and cluster 3 is in the upper-right corner.

## References

[1] Davies, D. L., and D. W. Bouldin. "A Cluster Separation Measure." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. PAMI-1, No. 2, 1979, pp. 224–227.

## See Also

`evalclusters.clustering.evaluation.CalinskiHarabaszEvaluation`  
| `clustering.evaluation.GapEvaluation` |  
`clustering.evaluation.SilhouetteEvaluation` |

## Concepts

- Class Attributes
- Property Attributes

# dataset.cat

---

**Purpose** Concatenate dataset arrays

**Syntax** `ds = cat(dim, ds1, ds2, ...)`

**Description** `ds = cat(dim, ds1, ds2, ...)` concatenates the dataset arrays `ds1`, `ds2`, ... along dimension `dim` by calling the `dataset/horzcat` or `dataset/vertcat` method. `dim` must be 1 or 2.

**See Also** `horzcat` | `vertcat`



<b>Purpose</b>	Categorical splits used for branches in decision tree
<b>Syntax</b>	<code>v=catsplit(t)</code> <code>v=catsplit(t,j)</code>
<b>Description</b>	<p><code>v=catsplit(t)</code> returns an n-by-2 cell array <code>v</code>. Each row in <code>v</code> gives left and right values for a categorical split. For each branch node <code>j</code> based on a categorical predictor variable <code>z</code>, the left child is chosen if <code>z</code> is in <code>v(j,1)</code> and the right child is chosen if <code>z</code> is in <code>v(j,2)</code>. The splits are in the same order as nodes of the tree. Nodes for these splits can be found by running <code>cuttype</code> and selecting 'categorical' cuts from top to bottom.</p> <p><code>v=catsplit(t,j)</code> takes an array <code>j</code> of rows and returns the splits for the specified rows.</p>
<b>See Also</b>	<code>classregtree</code>

# gmdistribution.cdf

---

**Purpose** Cumulative distribution function for Gaussian mixture distribution

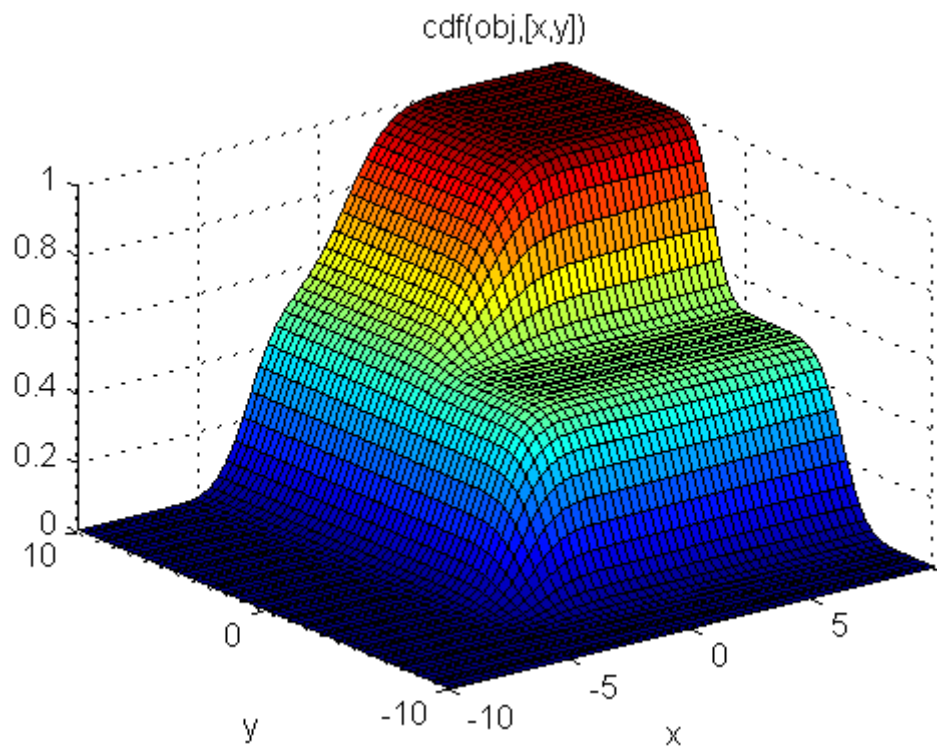
**Syntax** `y = cdf(obj,X)`

**Description** `y = cdf(obj,X)` returns a vector `y` of length  $n$  containing the values of the cumulative distribution function (`cdf`) for the `gmdistribution` object `obj`, evaluated at the  $n$ -by- $d$  data matrix `X`, where  $n$  is the number of observations and  $d$  is the dimension of the data. `obj` is an object created by `gmdistribution` or `fit`. `y(I)` is the cdf of observation `I`.

**Examples** Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2; -3 -5];
SIGMA = cat(3,[2 0; 0 .5],[1 0; 0 1]);
p = ones(1,2)/2;
obj = gmdistribution(MU,SIGMA,p);

ezsurf(@(x,y)cdf(obj,[x y]),[-10 10],[-10 10])
```



## See Also

[gmdistribution](#) | [fit](#) | [pdf](#) | [mvncdf](#)

# ccdesign

**Purpose** Central composite design

**Syntax**  
`dCC = ccdesign(n)`  
`[dCC,blocks] = ccdesign(n)`  
`[...] = ccdesign(n,'Name',value)`

**Description** `dCC = ccdesign(n)` generates a central composite design for  $n$  factors.  $n$  must be an integer 2 or larger. The output matrix `dCC` is  $m$ -by- $n$ , where  $m$  is the number of runs in the design. Each row represents one run, with settings for all factors represented in the columns. Factor values are normalized so that the cube points take values between -1 and 1.

`[dCC,blocks] = ccdesign(n)` requests a blocked design. The output `blocks` is an  $m$ -by-1 vector of block numbers for each run. Blocks indicate runs that are to be measured under similar conditions to minimize the effect of inter-block differences on the parameter estimates.

`[...] = ccdesign(n,'Name',value)` specifies one or more optional name/value pairs for the design. Valid parameters and their values are listed in the following table. Specify *Name* in single quotes.

Parameter	Description	Values	Value Description
center	Number of center points.	Integer	Number of center points to include.
		'uniform'	Select number of center points to give uniform precision.
		'orthogonal'	Select number of center points to give an orthogonal design. This is the default.

Parameter	Description	Values	Value Description
fraction	Fraction of full-factorial cube, expressed as an exponent of 1/2.	0	Whole design. Default when $n \leq 4$ .
		1	1/2 fraction. Default when $4 < n \leq 7$ or $n > 11$ .
		2	1/4 fraction. Default when $7 < n \leq 9$ .
		3	1/8 fraction. Default when $n = 10$ .
		4	1/16 fraction. Default when $n = 11$ .
type	Type of CCD.	'circumscribed'	Circumscribed (CCC). This is the default.
		'inscribed'	Inscribed (CCI).
		'faced'	Faced (CCF).
blocksize	Maximum number of points per block.	Integer	The default is Inf.

### Examples

The following creates a 2-factor CCC:

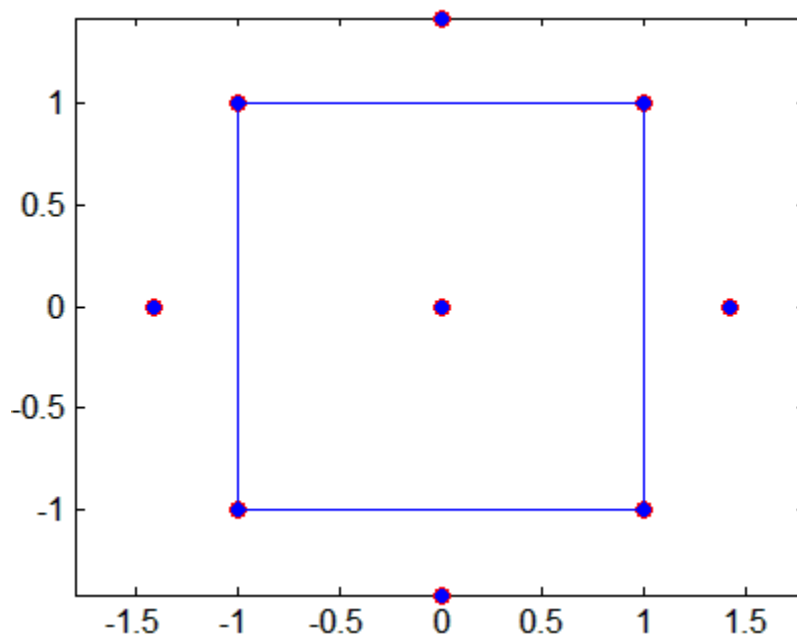
```
dCC = ccdesign(2,'type','circumscribed')
dCC =
    -1.0000    -1.0000
```

```
-1.0000    1.0000
 1.0000   -1.0000
 1.0000    1.0000
-1.4142     0
 1.4142     0
     0   -1.4142
     0    1.4142
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
```

The center point is run 8 times to reduce the correlations among the coefficient estimates.

Visualize the design as follows:

```
plot(dCC(:,1),dCC(:,2),'ro','MarkerFaceColor','b')
X = [1 -1 -1 -1; 1 1 1 -1];
Y = [-1 -1 1 -1; 1 -1 1 1];
line(X,Y,'Color','b')
axis square equal
```



**See Also**

`bbdesign`

**Purpose** Cumulative distribution functions

**Syntax**

```
y = cdf('name',x,A)
y = cdf('name',x,A,B)
y = cdf('name',x,A,B,C)
y = cdf( __ , 'upper' )
```

**Description** `y = cdf('name',x,A)` returns the cumulative distribution function for the one-parameter family of distributions specified by `name`. `a` contains parameter values for the distribution. The cumulative distribution function is evaluated at the values in `x` and its values are returned in `y`. If `x` and `A` are arrays, they must be the same size. If `x` is a scalar, it is expanded to a constant matrix the same size as `A`. If `a` is a scalar, it is expanded to a constant matrix the same size as `x`. `y` is the common size of `x` and `A` after any necessary scalar expansion.

`y = cdf('name',x,A,B)` returns the cumulative distribution function for two-parameter families of distributions, where parameter values are given in `A` and `B`. If `x`, `A`, and `B` are arrays, they must be the same size. If `x` is a scalar, it is expanded to a constant matrix the same size as `A` and `B`. If either `A` or `B` are scalars, they are expanded to constant matrices the same size as `x`. `y` is the common size of `x`, `A`, and `B` after any necessary scalar expansion.

`y = cdf('name',x,A,B,C)` returns the cumulative distribution function for three-parameter families of distributions, where parameter values are given in `A`, `B`, and `C`. If `x`, `A`, `B`, and `C` are arrays, they must be the same size. If `x` is a scalar, it is expanded to a constant matrix the same size as `A`, `B`, and `C`. If any of `A`, `B` or `C` are scalars, they are expanded to constant matrices the same size as `x`. `y` is the common size of `x`, `A`, `B`, and `C` after any necessary scalar expansion.

`y = cdf( __ , 'upper' )` returns the complement of the cumulative distribution function for the specified distribution, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the `'upper'` argument with any of the previous syntaxes.

The following are acceptable strings for `name` (specified in single quotes).



name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'beta' or 'Beta'	“Beta Distribution” on page B-5	a	b	—
'bino' or 'Binomial'	“Binomial Distribution” on page B-9	n: number of trials	p: probability of success for each trial	—
'birnbaumsaunders' or 'Birnbaum-Saunders'	“Birnbaum-Saunders Distribution” on page B-13	$\beta$	$\gamma$	—
'burr' or 'Burr'	“Burr Type XII Distribution” on page B-15	a: scale parameter	c: shape parameter	k: shape parameter
'chi2' or 'Chisquare'	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—
'exp' or 'Exponential'	“Exponential Distribution” on page B-32	$\mu$ : mean	—	—
'ev' or 'Extreme Value'	“Extreme Value Distribution” on page B-35	$\mu$ : location parameter	$\sigma$ : scale parameter	—
'f' or 'F'	“F Distribution” on page B-41	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	—
'gam' or 'Gamma'	“Gamma Distribution” on page B-43	a: shape parameter	b: scale parameter	—
'gev' or 'Generalized Extreme Value'	“Generalized Extreme Value Distribution” on page B-48	k: shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'gp' or 'Generalized Pareto'	“Generalized Pareto Distribution” on page B-53	$k$ : tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter
'geo' or 'Geometric'	“Geometric Distribution” on page B-57	$p$ : probability parameter	—	—
'hyge' or 'Hypergeometric'	“Hypergeometric Distribution” on page B-60	$M$ : size of the population	$K$ : number of items with the desired characteristic in the population	$n$ : number of samples drawn
'inversegaussian'	“Inverse Gaussian Distribution” on page B-63	$\mu$	$\lambda$	—
'logistic'	“Logistic Distribution” on page B-76	$\mu$	$\sigma$	—
'loglogistic'	“Loglogistic Distribution” on page B-78	$\mu$	$\sigma$	—
'logn' or 'Lognormal'	“Lognormal Distribution” on page B-80	$\mu$	$\sigma$	—
'nakagami'	“Nakagami Distribution” on page B-97	$\mu$	$\omega$	—
'nbin' or 'Negative Binomial'	“Negative Binomial Distribution” on page B-98	$r$ : number of successes	$p$ : probability of success in a single trial	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'ncf' or 'Noncentral F'	“Noncentral F Distribution” on page B-106	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	$\delta$ : noncentrality parameter
'nct' or 'Noncentral t'	“Noncentral t Distribution” on page B-108	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—
'ncx2' or 'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-104	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—
'norm' or 'Normal'	“Normal Distribution” on page B-110	$\mu$ : mean	$\sigma$ : standard deviation	—
'poiss' or 'Poisson'	“Poisson Distribution” on page B-118	$\lambda$ : mean	—	—
'rayl' or 'Rayleigh'	“Rayleigh Distribution” on page B-121	$b$ : scale parameter	—	—
'rician'	“Rician Distribution” on page B-123	$s$ : noncentrality parameter	$\sigma$ : scale parameter	—
't' or 'T'	“Student’s t Distribution” on page B-124	$\nu$ : degrees of freedom	—	—
'tlocationscale'	“t Location-Scale Distribution” on page B-126	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter
'unif' or 'Uniform'	“Uniform Distribution (Continuous)” on page B-133	$a$ : lower endpoint (minimum)	$b$ : upper endpoint (maximum)	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'unid' or 'Discrete Uniform'	“Uniform Distribution (Discrete)” on page B-139	N: maximum observable value	—	—
'wbl' or 'Weibull'	“Weibull Distribution” on page B-141	a: scale parameter	b: shape parameter	—

## Examples

### Compute Normal Distribution cdf

Compute the cdf of the normal distribution with mean 0 and standard deviation 1 at inputs -2, -1, 0, 1, 2.

```
p = cdf('Normal', -2:2, 0, 1)
```

```
p =
    0.0228    0.1587    0.5000    0.8413    0.9772
```

The order of the parameters is the same as for `normcdf`.

### Compute Poisson Distribution cdf

Compute the cdfs of Poisson distributions with rate parameters 1, 2, ... 5, at inputs 0, 1, ... 4, respectively.

```
p = cdf('Poisson', 0:4, 1:5)
```

```
p =
    0.3679    0.4060    0.4232    0.4335    0.4405
```

The order of the parameters is the same as for `poisscdf`.

## See Also

`pdf` | `icdf`

<b>Purpose</b>	Cumulative distribution function for piecewise distribution
<b>Syntax</b>	<pre>p= cdf(obj,x) p= cdf(obj,x,'upper')</pre>
<b>Description</b>	<p><code>p= cdf(obj,x)</code> returns an array <code>P</code> of values of the cumulative distribution function for the piecewise distribution object <code>obj</code>, evaluated at the values in the array <code>X</code>.</p> <p><code>p= cdf(obj,x,'upper')</code> returns the complement of the piecewise distribution <code>cdf</code> evaluated at the values in <code>x</code>, using an algorithm that more accurately computes the extreme upper tail probabilities.</p>
<b>Examples</b>	<p><b>Fit Pareto Tails to <i>t</i> Distribution</b></p> <p>Fit Pareto tails to a <i>t</i> distribution at cumulative probabilities 0.1 and 0.9.</p> <pre>t = trnd(3,100,1); obj = paretotails(t,0.1,0.9); [p,q] = boundary(obj)</pre> <pre>p =     0.1000     0.9000 q =    -1.7766     1.8432</pre> <p>Compute the <code>cdf</code> at the values in <code>q</code>.</p> <pre>cdf(obj,q)</pre> <pre>ans =     0.1000     0.9000</pre>
<b>See Also</b>	<code>paretotails</code>   <code>pdf</code>   <code>icdf</code>

# ProbDist.cdf

---

<b>Purpose</b>	Return cumulative distribution function (CDF) for ProbDist object	
<b>Syntax</b>	$Y = \text{cdf}(PD, X)$	
<b>Description</b>	$Y = \text{cdf}(PD, X)$ returns $Y$ , an array containing the cumulative distribution function (CDF) for the ProbDist object $PD$ , evaluated at values in $X$ .	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivParam or ProbDistUnivKernel.
	$X$	A numeric array of values where you want to evaluate the CDF.
<b>Output Arguments</b>	$Y$	An array containing the cumulative distribution function (CDF) for the ProbDist object $PD$ .
<b>See Also</b>	cdf	

<b>Purpose</b>	Cumulative distribution function of probability distribution object
<b>Syntax</b>	<pre>y = cdf(pd,x) y = cdf(pd,x,'upper')</pre>
<b>Description</b>	<p><code>y = cdf(pd,x)</code> returns the cumulative distribution function (cdf) of the probability distribution <code>pd</code> at the values in <code>x</code>.</p> <p><code>y = cdf(pd,x,'upper')</code> returns the complement of the cumulative distribution function (cdf) of the probability distribution <code>pd</code> at the values in <code>x</code>, using an algorithm that more accurately computes the extreme upper tail probabilities.</p>
<b>Input Arguments</b>	<p><b>pd - Probability distribution</b> probability distribution object</p> <p>Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code>. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using <code>fitdist</code> or the Distribution Fitting app.</p> <p><b>x - Values at which to calculate cdf</b> array</p> <p>Values at which to calculate the cdf, specified as an array.</p> <p><b>Data Types</b> single   double</p> <p><b>upper - Upper tail probability flag</b> 'upper'</p> <p>Upper tail probability flag, specified as 'upper'. If you specify 'upper', then <code>cdf</code> returns the complement of the cdf of <code>pd</code>, using an algorithm that more accurately computes the extreme upper tail probabilities.</p>

# prob.TruncatableDistribution.cdf

---

## Output Arguments

### **y - Cumulative distribution function**

array

Cumulative distribution function of the specified probability distribution, evaluated at the values in **x**, returned as a array. **y** has the same dimensions as **x**.

## Examples

### **Plot Standard Normal Distribution cdf**

Create a standard normal distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

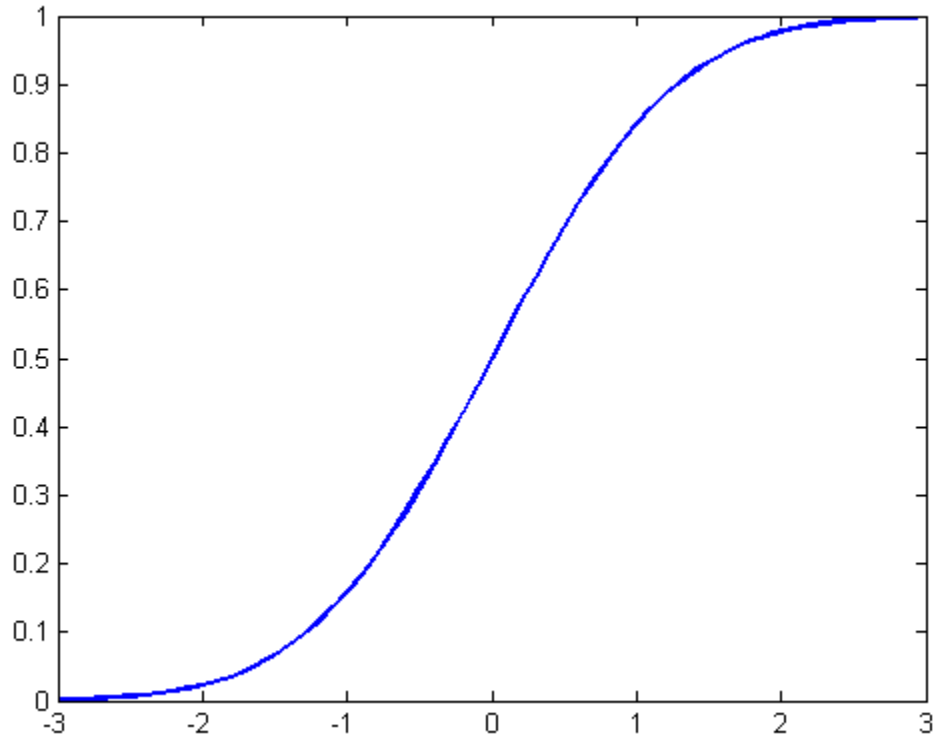
Specify the **x** values and compute the cdf.

```
x = -3:.1:3;  
cdf_normal = cdf(pd,x);
```

Plot the cdf of the standard normal distribution.

```
plot(x,cdf_normal,'LineWidth',2)
```





## Plot Gamma Distribution cdf

Create three gamma distribution objects. The first uses the default parameter values. The second specifies  $a = 1$  and  $b = 2$ . The third specifies  $a = 2$  and  $b = 1$ .

```
pd_gamma = makedist('Gamma')
```

```
pd_gamma =
```

# prob.TruncatableDistribution.cdf

---

```
GammaDistribution
```

```
Gamma distribution
```

```
a = 1
```

```
b = 1
```

```
pd_12 = makedist('Gamma','a',1,'b',2)
```

```
pd_12 =
```

```
GammaDistribution
```

```
Gamma distribution
```

```
a = 1
```

```
b = 2
```

```
pd_21 = makedist('Gamma','a',2,'b',1)
```

```
pd_21 =
```

```
GammaDistribution
```

```
Gamma distribution
```

```
a = 2
```

```
b = 1
```

Specify the x values and compute the cdf for each distribution.

```
x = 0:.1:5;
```

```
cdf_gamma = cdf(pd_gamma,x);
```

```
cdf_12 = cdf(pd_12,x);
```

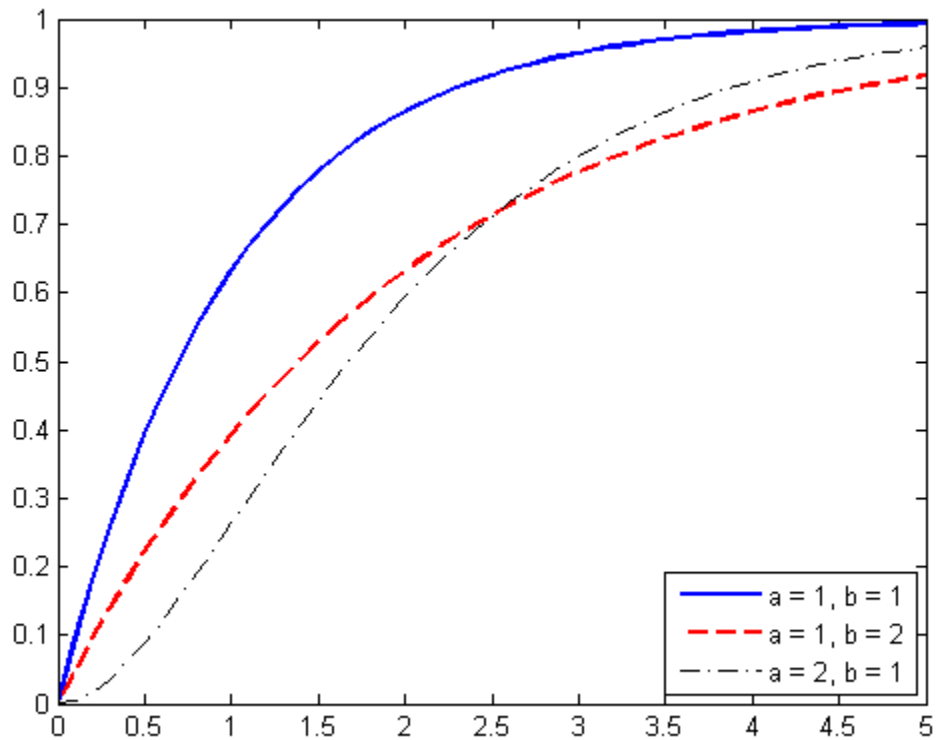
```
cdf_21 = cdf(pd_21,x);
```

Create a plot to visualize how the cdf of the gamma distribution changes when you specify different values for the shape parameters a and b.

```
figure;
```

```
J = plot(x,cdf_gamma)
```

```
hold on;  
K = plot(x,cdf_gamma_12,'r--')  
L = plot(x,cdf_gamma_21,'k-.')  
set(J,'LineWidth',2);  
set(K,'LineWidth',2);  
legend([J K L], 'a = 1, b = 1', 'a = 1, b = 2', 'a = 2, b = 1', 'Location  
hold off;
```



## See Also

[makedist](#) | [fitdist](#) | [cdf](#) | [icdf](#) | [pdf](#) | [dfittool](#)

# cdfplot

---

**Purpose** Empirical cumulative distribution function plot

**Syntax**

```
cdfplot(X)
h = cdfplot(X)
[h,stats] = cdfplot(X)
```

**Description** `cdfplot(X)` displays a plot of the empirical cumulative distribution function (cdf) for the data in the vector `X`. The empirical cdf  $F(x)$  is defined as the proportion of `X` values less than or equal to `x`.

This plot, like those produced by `hist` and `normplot`, is useful for examining the distribution of a sample of data. You can overlay a theoretical cdf on the same plot to compare the empirical distribution of the sample to the theoretical distribution.

The `kstest`, `kstest2`, and `lillietest` functions compute test statistics that are derived from the empirical cdf. You may find the empirical cdf plot produced by `cdfplot` useful in helping you to understand the output from those functions.

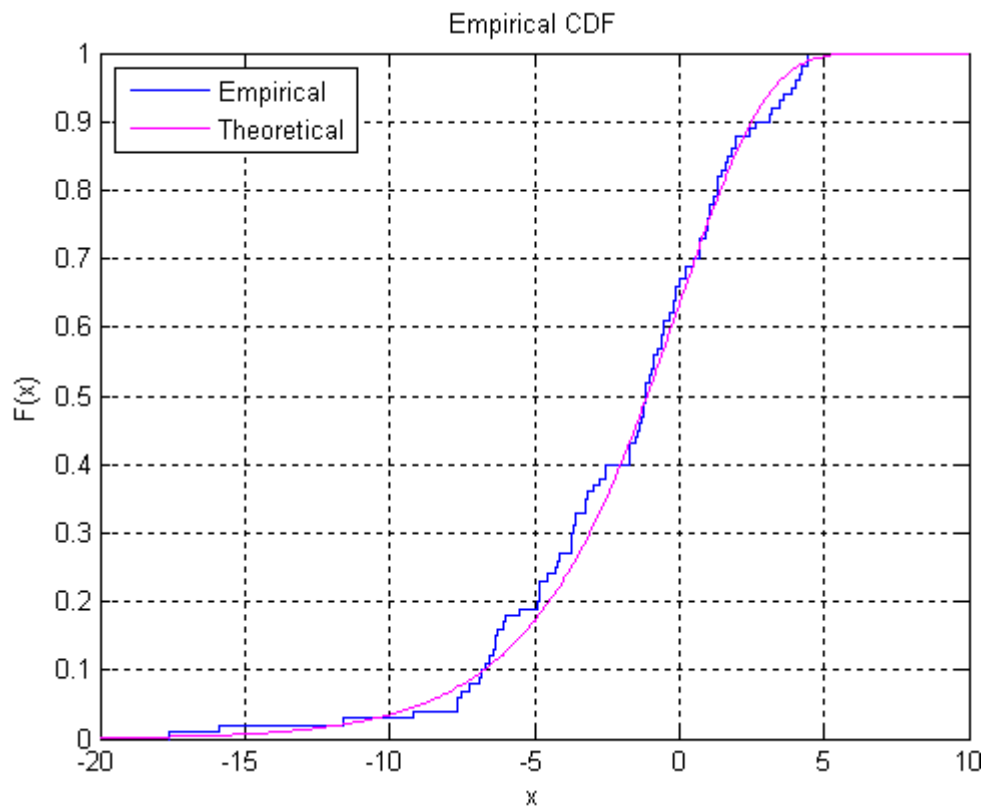
`h = cdfplot(X)` returns a handle to the cdf curve.

`[h,stats] = cdfplot(X)` also returns a `stats` structure with the following fields.

Field	Description
<code>stats.min</code>	Minimum value
<code>stats.max</code>	Maximum value
<code>stats.mean</code>	Sample mean
<code>stats.median</code>	Sample median (50th percentile)
<code>stats.std</code>	Sample standard deviation

**Examples** The following example compares the empirical cdf for a sample from an extreme value distribution with a plot of the cdf for the sampling distribution. In practice, the sampling distribution would be unknown, and would be chosen to match the empirical cdf.

```
y = evrnd(0,3,100,1);  
cdfplot(y)  
hold on  
x = -20:0.1:10;  
f = evcdf(x,0,3);  
plot(x,f,'m')  
legend('Empirical','Theoretical','Location','NW')
```



**See Also** [ecdf](#)

# cell2dataset

---

**Purpose** Convert cell array to dataset array

**Syntax**  
`ds = cell2dataset(C)`  
`ds = cell2dataset(C,Name,Value)`

**Description** `ds = cell2dataset(C)` converts a cell array to a dataset array.

`ds = cell2dataset(C,Name,Value)` performs the conversion using additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **C - Input cell array**

cell array

Input cell array to convert to a dataset array, specified as an  $M$ -by- $N$  cell array. Each column of `C` becomes a variable in the output dataset array, `ds`. By default, `cell2dataset` assumes that the first row of `C` contains variable names.

### **Data Types**

cell

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'ReadVarNames', false, 'ReadObsNames', true` specifies that the first row of the cell array does not contain variable names, but the first column contains observation names.

### **'ReadVarNames' - Indicator for whether or not to read variable names**

true (default) | false

Indicator for whether or not to read variable names from the first row of the input cell array, specified as the comma-separated pair consisting of 'ReadVarNames' and either true or false. The default value is true, unless variable names are specified using the name-value pair argument VarNames. When ReadVarNames is false, cell2dataset creates default variable names if you do not provide any.

**Example:** 'ReadVarNames',false

### **'VarNames' - Variable names for output dataset array**

cell array of strings

Variable names for the output dataset array, specified as the comma-separated pair consisting of 'VarNames' and a cell array of strings. You must provide a variable name for each variable in ds. The names must be valid MATLAB identifiers, and must be unique.

**Example:** 'VarNames',{ 'myVar1', 'myVar2', 'myVar3' }

### **'ReadObsNames' - Indicator for whether or not to read observation names**

false (default) | true

Indicator for whether or not to read observation names from the input cell array, specified as the comma-separated pair consisting of 'ReadObsNames' and either true or false. When ReadObsNames has the value true, cell2dataset creates observation names in ds using the first column of C, and sets ds.Properties.DimNames equal to {C{1,1}, 'Variables'}.

**Example:** 'ReadObsNames',true

### **'ObsNames' - Observation names for output dataset array**

cell array of strings

Observation names for the output dataset array, specified as the comma-separated pair consisting of 'ObsNames' and a cell array of strings. The names do not need to be valid MATLAB identifiers, but they must be unique.

## **'NumCols' - Number of columns for each variable**

vector of nonnegative integers

Number of columns for each variable in `ds`, specified as the comma-separated pair consisting of `'NumCols'` and a vector of nonnegative integers. When the number of columns for a variable is greater than one, `cell2dataset` combines multiple columns in `C` into a single variable in `ds`. The vector you assign to `NumCols` must sum to `size(C,2)`, or `size(C,1)` of `ReadObsNames` is equal to `true`.

For example, to convert a cell array with eight columns into a dataset array with five variables, specify a vector with five elements that sum to eight, such as `'NumCols', [1,1,3,1,2]`.

## **Output Arguments**

### **`ds` - Output dataset array**

dataset array

Output dataset array, returned by default with a variable for each column of `C`, an observation for each row of `C` (except for the first row), and variable names corresponding to the first row of `C`.

- If you set `ReadVarNames` equal to `false` (or specify `VarNames`), then there is an observation in `ds` for each row of `C`, and `cell2dataset` creates default variable names (or uses the names in `VarNames`).
- If you set `ReadObsNames` equal to `true`, then `cell2dataset` uses the first column of `C` as observation names.
- If you specify `NumCols`, then the number of variables in `ds` is equal to the length of the specified vector of column numbers.

## **Examples**

### **Convert Cell Array to Dataset Array**

Convert a cell array to a dataset array using the default options.

Create a cell array to convert.

```
C = {'Name', 'Gender', 'SystolicBP', 'DiastolicBP';  
    'CLARK', 'M', 124, 93;  
    'BROWN', 'F', 122, 80;  
    'MARTIN', 'M', 130, 92}
```



```
C =
      'Name'      'Gender'      'SystolicBP'      'DiastolicBP'
      'CLARK'     'M'          [      124]      [      93]
      'BROWN'     'F'          [      122]      [      80]
      'MARTIN'    'M'          [      130]      [      92]
```

Convert the cell array to a dataset array.

```
ds = cell2dataset(C)
```

```
ds =
      Name          Gender      SystolicBP      DiastolicBP
      'CLARK'      'M'          124              93
      'BROWN'      'F'          122              80
      'MARTIN'     'M'          130              92
```

The first row of `C` become the variable names in the output dataset array, `ds`.

### Create a Dataset Array with Multicolumn Variables

Convert a cell array to a dataset array containing multicolumn variables.

Create a cell array to convert.

```
C = {'Name', 'Gender', 'SystolicBP', 'DiastolicBP';
     'CLARK', 'M', 124, 93;
     'BROWN', 'F', 122, 80;
     'MARTIN', 'M', 130, 92}
```

```
C =
      'Name'      'Gender'      'SystolicBP'      'DiastolicBP'
      'CLARK'     'M'          [      124]      [      93]
      'BROWN'     'F'          [      122]      [      80]
      'MARTIN'    'M'          [      130]      [      92]
```

# cell2dataset

---

Convert the cell array to a dataset array, combining the systolic and diastolic blood pressure measurements into one variable named `BloodPressure`.

```
ds = cell2dataset(C, 'NumCols', [1,1,2]);  
ds.Properties.VarNames{3} = 'BloodPressure';  
ds
```

```
ds =
```

Name	Gender	BloodPressure
'CLARK'	'M'	124 93
'BROWN'	'F'	122 80
'MARTIN'	'M'	130 92

The output dataset array has three observations and three variables.

## See Also

[dataset](#) | [dataset2cell](#) | [struct2dataset](#)

## Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Create a Dataset Array from a File” on page 2-71

## Concepts

- “Dataset Arrays” on page 2-135

**Purpose** Convert categorical array to cell array of strings

**Syntax** `B = cellstr(A)`

**Description** `B = cellstr(A)` converts the categorical array `A` to a cell array of strings. Each element of `B` contains the categorical level label for the corresponding element of `A`.

**See Also** `char` | `getlabels`

# dataset.cellstr

---

**Purpose** Create cell array of strings from dataset array

**Syntax** B = cellstr(A)  
B = cellstr(A, VARS)

**Description** B = cellstr(A) returns the contents of the dataset A, converted to a cell array of strings. The variables in the dataset must support the conversion and must have compatible sizes.

B = cellstr(A, VARS) returns the contents of the dataset variables specified by VARS. VARS is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

**See Also** dataset.double | dataset.replacdata

**Purpose** Convert categorical array to character array

**Syntax** `B = char(A)`

**Description** `B = char(A)` converts the categorical array `A` to a 2-D character matrix. `char` does not preserve the shape of `A`. `B` contains `numel(A)` rows, and each row of `B` contains the categorical level label for the corresponding element of `A(:)`.

**See Also** `cellstr` | `getlabels`

# chi2cdf

---

**Purpose** Chi-square cumulative distribution function

**Syntax**  
`p = chi2cdf(x,v)`  
`p = chi2cdf(x,v,'upper')`

**Description** `p = chi2cdf(x,v)` computes the chi-square cdf at each of the values in `x` using the corresponding degrees of freedom in `v`. `x` and `v` can be vectors, matrices, or multidimensional arrays that have the same size. The degrees of freedom parameters in `v` must be positive integers, and the values in `x` must lie on the interval `[0 Inf]`. A scalar input is expanded to a constant array with the same dimensions as the other input.

`p = chi2cdf(x,v,'upper')` returns the complement of the chi-square cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The  $\chi^2$  cdf for a given value  $x$  and degrees-of-freedom  $v$  is

$$p = F(x | v) = \int_0^x \frac{t^{(v-2)/2} e^{-t/2}}{2^{v/2} \Gamma(v/2)} dt$$

where  $\Gamma(\cdot)$  is the Gamma function.

The chi-square density function with  $v$  degrees-of-freedom is the same as the gamma density function with parameters  $v/2$  and 2.

## Examples

### Compute Chi-Square CDF

```
probability = chi2cdf(5,1:5)
```

```
probability =  
    0.9747    0.9179    0.8282    0.7127    0.5841
```

```
probability = chi2cdf(1:5,1:5)
```

```
probability =  
    0.6827    0.6321    0.6084    0.5940    0.5841
```

**See Also**      `cdf` | `chi2pdf` | `chi2inv` | `chi2stat` | `chi2rnd`

**How To**      • “Chi-Square Distribution” on page B-28

# chi2gof

---

**Purpose** Chi-square goodness-of-fit test

**Syntax**

```
h = chi2gof(x)
h = chi2gof(x,Name,Value)
[h,p] = chi2gof( ___ )
[h,p,stats] = chi2gof( ___ )
```

**Description** `h = chi2gof(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a normal distribution with a mean and variance estimated from `x`, using the chi-square goodness-of-fit test. The alternative hypothesis is that the data does not come from such a distribution. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = chi2gof(x,Name,Value)` returns a test decision for the chi-square goodness-of-fit test with additional options specified by one or more name-value pair arguments. For example, you can test for a distribution other than normal, or change the significance level of the test.

`[h,p] = chi2gof( ___ )` also returns the *p*-value `p` of the hypothesis test, using any of the input arguments from the previous syntaxes.

`[h,p,stats] = chi2gof( ___ )` also returns the structure `stats`, containing information about the test statistic.

## Input Arguments

### **x - Sample data**

vector

Sample data for the hypothesis test, specified as a vector.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can



specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'NBins',8,'Alpha',0.01` pools the data into eight bins and conducts the hypothesis test at the 1% significance level.

### **'NBins' - Number of bins**

10 (default) | positive integer value

Number of bins to use for the data pooling, specified as the comma-separated pair consisting of `'NBins'` and a positive integer value. If you specify a value for `NBins`, do not specify a value for `Ctrs` or `Edges`.

**Example:** `'NBins',8`

### **Data Types**

single | double

### **'Ctrs' - Bin centers**

vector

Bin centers, specified as the comma-separated pair consisting of `'Ctrs'` and a vector of center values for each bin. If you specify a value for `Ctrs`, do not specify a value for `NBins` or `Edges`.

**Example:** `'Ctrs',[1 2 3 4 5]`

### **Data Types**

single | double

### **'Edges' - Bin edges**

vector

Bin edges, specified as the comma-separated pair consisting of `'Edges'` and a vector of edge values for each bin. If you specify a value for `Edges`, do not specify a value for `NBins` or `Ctrs`.

**Example:** `'Edges',[-2.5 -1.5 -0.5 0.5 1.5 2.5]`

### **Data Types**

single | double

## **'CDF' - cdf of hypothesized distribution**

probability distribution object | function handle | cell array

The cdf of the hypothesized distribution, specified as the comma-separated pair consisting of 'CDF' and a probability distribution object, function handle, or cell array.

- If CDF is a probability distribution object, the degrees of freedom account for whether you estimate the parameters using `fitdist` or specify them using `makedist`.
- If CDF is a function handle, the distribution function must take `x` as its only argument.
- If CDF is a cell array, the first element must be a function handle, and the remaining elements must be parameter values, one per cell. The function must take `x` as its first argument, and the other parameters in the array as later arguments.

If you specify a value for CDF, do not specify a value for Expected.

**Example:** 'CDF',pd\_object

### **Data Types**

single | double

## **'Expected' - Expected counts**

vector of nonnegative values

Expected counts for each bin, specified as the comma-separated pair of 'Expected' and a vector of nonnegative values. If Expected depends on estimated parameters, use `NParams` to ensure that `chi2gof` correctly calculates the degrees of freedom. If you specify a value for Expected, do not specify a value for CDF.

**Example:** 'Expected',[19.1446 18.3789 12.3224 8.2432 4.1378]

### **Data Types**

single | double

## **'NParams' - Number of estimated parameters**

positive integer value

Number of estimated parameters used to describe the null distribution, specified as the comma-separated pair consisting of 'NParams' and a positive integer value. This value adjusts the degrees of freedom of the test based on the number of estimated parameters used to compute the cdf or expected counts.

The default value for NParams depends on how you specify the null distribution:

- If you specify CDF as a probability distribution object, NParams is equal to the number of estimated parameters used to create the object.
- If you specify CDF as a function name or handle, the default value of NParams is 0.
- If you specify CDF as a cell array, the default value of NParams is the number of parameters in the array.
- If you specify Expected, the default value of NParams is 0.

**Example:** 'NParams',1

### Data Types

single | double

### 'EMin' - Minimum expected count per bin

5 (default) | nonnegative integer value

Minimum expected count per bin, specified as the comma-separated pair consisting of 'EMin' and a nonnegative integer value. If the bin at the extreme end of either tail has an expected value less than EMin, it is combined with a neighboring bin until the count in each extreme bin is at least 5. If any interior bins have a count less than 5, chi2gof displays a warning, but does not combine the interior bins. In that case, you should use fewer bins, or provide bin centers or edges, to increase the expected counts in all bins. Specify EMin as 0 to prevent the combining of bins.

**Example:** 'EMin',0

## Data Types

single | double

## 'Frequency' - Frequency

vector of nonnegative integer values

Frequency of data values, specified as the comma-separated pair consisting of 'Frequency' and a vector of nonnegative integer values that is the same length as the vector  $x$ .

**Example:** 'Frequency',[20 16 13 10 8]

## Data Types

single | double

## 'Alpha' - Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

## Data Types

single | double

## Output Arguments

### **h** - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** - p-value

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **stats - Test statistics**

structure

Test statistics, returned as a structure containing the following:

- `chi2stat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.
- `edges` — Vector of bin edges after pooling.
- `O` — Vector of observed counts for each bin.
- `E` — Vector of expected counts for each bin.

## **Examples**

### **Test for a Normal Distribution**

Create a standard normal probability distribution object. Generate a data vector `x` using random numbers from the distribution.

```
pd = makedist('Normal');  
rng('default') % for reproducibility  
x = random(pd,100,1);
```

Test the null hypothesis that the data in `x` comes from a population with a normal distribution.

```
h = chi2gof(x)
```

```
h =
```

```
0
```

The returned value `h = 0` indicates that `chi2gof` does not reject the null hypothesis at the default 5% significance level.

## Test the Hypothesis at a Different Significance Level

Create a standard normal probability distribution object. Generate a data vector  $x$  using random numbers from the distribution.

```
pd = makedist('Normal');  
rng('default') % for reproducibility  
x = random(pd,100,1);
```

Test the null hypothesis that the data in  $x$  comes from a population with a normal distribution at the 1% significance level.

```
[h,p] = chi2gof(x,'Alpha',0.01)
```

```
h =
```

```
0
```

```
p =
```

```
0.3775
```

The returned value  $h = 0$  indicates that `chi2gof` does not reject the null hypothesis at the 1% significance level.

## Test for a Weibull Distribution Using a Probability Distribution Object

Navigate to the appropriate folder and load the lightbulb lifetime sample data.

```
cd('matlabroot');  
cd('help/toolbox/stats/examples');  
load lightbulb;
```

Create a vector from the first column of the data matrix, which contains the lifetime in hours of the lightbulbs.

```
x = lightbulb(:,1);
```

Test the null hypothesis that the data in `x` comes from a population with a Weibull distribution. Use `fitdist` to create a probability distribution object with `A` and `B` parameters estimated from the data.

```
pd = fitdist(x, 'Weibull');
h = chi2gof(x, 'CDF', pd)
```

```
h =
```

```
1
```

The returned value `h = 1` indicates that `chi2gof` rejects the null hypothesis at the default 5% significance level.

### Test for a Poisson Distribution

Create six bins, numbered 0 through 5, to use for data pooling.

```
bins = 0:5;
```

Create a vector containing the observed counts for each bin and compute the total number of observations.

```
obsCounts = [6 16 10 12 4 2];
n = sum(obsCounts);
```

Fit a Poisson probability distribution object to the data and compute the expected count for each bin.

```
pd = fitdist(bins, 'Poisson', 'Frequency', obsCounts);
expCounts = n*pdf(pd, bins);
```

Test the null hypothesis that the data in `obsCounts` comes from a Poisson distribution with a `lambda` parameter equal to `lambdaHat`.

```
[h,p,st] = chi2gof(bins, 'Ctrs', bins, ...
                  'Frequency', obsCounts, ...
                  'Expected', expCounts, ...
```

```
                                'NParams',1)

h =

    0

p =

    0.4654

st =

    chi2stat: 2.5550
           df: 3
          edges: [1x6 double]
                0: [6 16 10 12 6]
                E: [7.0429 13.8041 13.5280 8.8383 6.0284]
```

The returned value `h = 0` indicates that `chi2gof` does not reject the null hypothesis at the default 5% significance level. The vector `E` contains the expected counts for each bin under the null hypothesis, and `0` contains the observed counts for each bin.

## Definitions

### Chi-Square Goodness-of-Fit Test

The chi-square goodness-of-fit test determines if a data sample comes from a specified probability distribution, with parameters estimated from the data.

The test groups the data into bins, calculating the observed and expected counts for those bins, and computing the chi-square test statistic

$$\chi^2 = \sum_{i=1}^N (O_i - E_i)^2 / E_i ,$$



where  $O_i$  are the observed counts and  $E_i$  are the expected counts based on the hypothesized distribution. The test statistic has an approximate chi-square distribution when the counts are sufficiently large.

**Algorithms**

`chi2gof` compares the value of the test statistic to a chi-square distribution with degrees of freedom equal to  $n_{bins} - 1 - n_{params}$ , where  $n_{bins}$  is the number of bins used for the data pooling and  $n_{params}$  is the number of estimated parameters used to determine the expected counts. If there are not enough degrees of freedom to conduct the test, `chi2gof` returns the  $p$ -value as NaN.

**See Also**

`kstest` | `lillietest`

**Concepts**

- “Chi-Square Distribution” on page B-28

# chi2inv

---

**Purpose** Chi-square inverse cumulative distribution function

**Syntax** `X = chi2inv(P,V)`

**Description** `X = chi2inv(P,V)` computes the inverse of the chi-square cdf with degrees of freedom specified by `V` for the corresponding probabilities in `P`. `P` and `V` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The degrees of freedom parameters in `V` must be positive integers, and the values in `P` must lie in the interval `[0 1]`.

The inverse chi-square cdf for a given probability  $p$  and  $v$  degrees of freedom is

$$x = F^{-1}(p | v) = \{x : F(x | v) = p\}$$

where

$$p = F(x | v) = \int_0^x \frac{t^{(v-2)/2} e^{-t/2}}{2^{v/2} \Gamma(v/2)} dt$$

and  $\Gamma(\cdot)$  is the Gamma function. Each element of output `X` is the value whose cumulative probability under the chi-square cdf defined by the corresponding degrees of freedom parameter in `V` is specified by the corresponding value in `P`.

**Examples** Find a value that exceeds 95% of the samples from a chi-square distribution with 10 degrees of freedom.

```
x = chi2inv(0.95,10)
x =
    18.3070
```

You would observe values greater than 18.3 only 5% of the time by chance.

**See Also**      `icdf` | `chi2cdf` | `chi2pdf` | `chi2stat` | `chi2rnd`

**How To**      • “Chi-Square Distribution” on page B-28

# chi2pdf

---

**Purpose** Chi-square probability density function

**Syntax** `Y = chi2pdf(X,V)`

**Description** `Y = chi2pdf(X,V)` computes the chi-square pdf at each of the values in `X` using the corresponding degrees of freedom in `V`. `X` and `V` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of the output `Y`. A scalar input is expanded to a constant array with the same dimensions as the other input.

The degrees of freedom parameters in `V` must be positive integers, and the values in `X` must lie on the interval `[0 Inf]`.

The chi-square pdf for a given value  $x$  and  $\nu$  degrees of freedom is

$$y = f(x | \nu) = \frac{x^{(\nu-2)/2} e^{-x/2}}{2^{\nu/2} \Gamma(\nu / 2)}$$

where  $\Gamma(\cdot)$  is the Gamma function.

If  $x$  is standard normal, then  $x^2$  is distributed chi-square with one degree of freedom. If  $x_1, x_2, \dots, x_n$  are  $n$  independent standard normal observations, then the sum of the squares of the  $x$ 's is distributed chi-square with  $n$  degrees of freedom (and is equivalent to the gamma density function with parameters  $\nu/2$  and  $2$ ).

## Examples

```
nu = 1:6;  
x = nu;  
y = chi2pdf(x,nu)  
y =  
    0.2420    0.1839    0.1542    0.1353    0.1220    0.1120
```

The mean of the chi-square distribution is the value of the degrees of freedom parameter, `nu`. The above example shows that the probability density of the mean falls as `nu` increases.

## See Also

`pdf` | `chi2cdf` | `chi2inv` | `chi2stat` | `chi2rnd`

**How To**

- “Chi-Square Distribution” on page B-28

# chi2rnd

---

**Purpose** Chi-square random numbers

**Syntax**  
R = chi2rnd(V)  
R = chi2rnd(V,m,n,...)  
R = chi2rnd(V,[m,n,...])

**Description** R = chi2rnd(V) generates random numbers from the chi-square distribution with degrees of freedom parameters specified by V. V can be a vector, a matrix, or a multidimensional array. R is the same size as V.  
R = chi2rnd(V,m,n,...) or R = chi2rnd(V,[m,n,...]) generates an m-by-n-by-... array containing random numbers from the chi-square distribution with degrees of freedom parameter V. V can be a scalar or an array of the same size as R.

**Examples** Note that the first and third commands are the same, but are different from the second command.

```
r = chi2rnd(1:6)
r =
    0.0037    3.0377    7.8142    0.9021    3.2019    9.0729

r = chi2rnd(6,[1 6])
r =
    6.5249    2.6226   12.2497    3.0388    6.3133    5.0388

r = chi2rnd(1:6,1,6)
r =
    0.7638    6.0955    0.8273    3.2506    1.5469   10.9197
```

**See Also** random | chi2cdf | chi2pdf | chi2inv | chi2stat

**How To** • “Chi-Square Distribution” on page B-28

**Purpose** Chi-square mean and variance

**Syntax** `[M,V] = chi2stat(NU)`

**Description** `[M,V] = chi2stat(NU)` returns the mean of and variance for the chi-square distribution with degrees of freedom parameters specified by NU.

The mean of the chi-square distribution is  $v$ , the degrees of freedom parameter, and the variance is  $2v$ .

### Examples

```

nu = 1:10;
nu = nu'*nu;
[m,v] = chi2stat(nu)
m =
    1     2     3     4     5     6     7     8     9    10
    2     4     6     8    10    12    14    16    18    20
    3     6     9    12    15    18    21    24    27    30
    4     8    12    16    20    24    28    32    36    40
    5    10    15    20    25    30    35    40    45    50
    6    12    18    24    30    36    42    48    54    60
    7    14    21    28    35    42    49    56    63    70
    8    16    24    32    40    48    56    64    72    80
    9    18    27    36    45    54    63    72    81    90
   10    20    30    40    50    60    70    80    90   100

v =
    2     4     6     8    10    12    14    16    18    20
    4     8    12    16    20    24    28    32    36    40
    6    12    18    24    30    36    42    48    54    60
    8    16    24    32    40    48    56    64    72    80
   10    20    30    40    50    60    70    80    90   100
   12    24    36    48    60    72    84    96   108   120
   14    28    42    56    70    84    98   112   126   140
   16    32    48    64    80    96   112   128   144   160
   18    36    54    72    90   108   126   144   162   180
   20    40    60    80   100   120   140   160   180   200

```

# chi2stat

---

**See Also**

[chi2cdf](#) | [chi2pdf](#) | [chi2inv](#) | [chi2rnd](#)

**How To**

- “Chi-Square Distribution” on page B-28



## Purpose

Child nodes

## Syntax

```
C = children(t)
C = children(t,nodes)
```

## Description

`C = children(t)` returns an  $n$ -by-2 array `C` containing the numbers of the child nodes for each node in the tree `t`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

`C = children(t,nodes)` takes a vector `nodes` of node numbers and returns the children for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

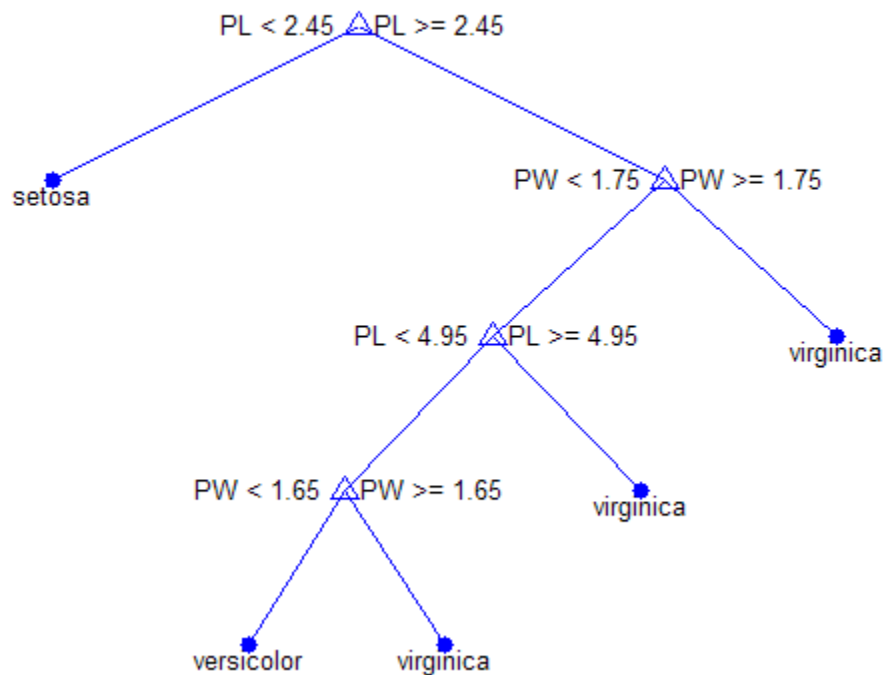
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

# classregtree.children

Click to display:  Magnification:  Pruning level:



```
C = children(t)
```

```
C =
```

```
2    3
0    0
4    5
6    7
0    0
8    9
0    0
0    0
```

0 0

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes` | `parent`

# cholcov

---

**Purpose** Cholesky-like covariance decomposition

**Syntax**  
`T = cholcov(SIGMA)`  
`[T,num] = cholcov(SIGMA)`  
`[T,num] = cholcov(SIGMA,0)`

**Description** `T = cholcov(SIGMA)` computes `T` such that `SIGMA = T'*T`. `SIGMA` must be square, symmetric, and positive semi-definite. If `SIGMA` is positive definite, then `T` is the square, upper triangular Cholesky factor. If `SIGMA` is not positive definite, `T` is computed from an eigenvalue decomposition of `SIGMA`. `T` is not necessarily triangular or square in this case. Any eigenvectors whose corresponding eigenvalue is close to zero (within a small tolerance) are omitted. If any remaining eigenvalues are negative, `T` is empty.

`[T,num] = cholcov(SIGMA)` returns the number `num` of negative eigenvalues of `SIGMA`, and `T` is empty if `num` is positive. If `num` is zero, `SIGMA` is positive semi-definite. If `SIGMA` is not square and symmetric, `num` is NaN and `T` is empty.

`[T,num] = cholcov(SIGMA,0)` returns `num` equal to zero if `SIGMA` is positive definite, and `T` is the Cholesky factor. If `SIGMA` is not positive definite, `num` is a positive integer and `T` is empty. `[...] = cholcov(SIGMA,1)` is equivalent to `[...] = cholcov(SIGMA)`.

## Examples

The following 4-by-4 covariance matrix is rank-deficient:

```
C1 = [2 1 1 2;1 2 1 2;1 1 2 2;2 2 2 3]
```

```
C1 =
```

```
     2     1     1     2
     1     2     1     2
     1     1     2     2
     2     2     2     3
```

```
rank(C1)
```

```
ans =
```

```
     3
```

Use `cholcov` to factor `C1`:

```
T = cholcov(C1)
T =
    -0.2113    0.7887   -0.5774         0
     0.7887   -0.2113   -0.5774         0
     1.1547    1.1547    1.1547    1.7321
```

```
C2 = T'*T
C2 =
    2.0000    1.0000    1.0000    2.0000
    1.0000    2.0000    1.0000    2.0000
    1.0000    1.0000    2.0000    2.0000
    2.0000    2.0000    2.0000    3.0000
```

Use T to generate random data with the specified covariance:

```
C3 = cov(randn(1e6,3)*T)
C3 =
    1.9973    0.9982    0.9995    1.9975
    0.9982    1.9962    0.9969    1.9956
    0.9995    0.9969    1.9980    1.9972
    1.9975    1.9956    1.9972    2.9951
```

## See Also

chol | cov

# categorical.circshift

---

**Purpose** Shift categorical array circularly

**Syntax** `B = circshift(A, shiftsize)`

**Description** `B = circshift(A, shiftsize)` circularly shifts the values in the categorical array `A` by `shiftsize` elements. `shiftsize` is a vector of integer scalars where the `n`-th element specifies the shift amount for the `n`-th dimension of array `A`. If an element in `shiftsize` is positive, the values of `A` are shifted down (or to the right). If it is negative, the values of `A` are shifted up (or to the left).

**See Also** `permute` | `shiftdim`

# NaiveBayes.CIsNonEmpty property

---

**Purpose**

Flag for non-empty classes

**Description**

The `CIsNonEmpty` property is a logical vector of length `NClasses` specifying which classes are not empty. When the grouping variable is categorical, it may contain categorical levels that don't appear in the elements of the grouping variable. Those levels are empty and `NaiveBayes` ignores them for the purposes of training the classifier.

# classregtree.classcount

---

**Purpose** Class counts

**Syntax** `P = classcount(t)`  
`P = classcount(t,nodes)`

**Description** `P = classcount(t)` returns an  $n$ -by- $m$  array  $P$  of class counts for the nodes in the classification tree  $t$ , where  $n$  is the number of nodes and  $m$  is the number of classes. For any node number  $i$ , the class counts  $P(i,:)$  are counts of observations (from the data used in fitting the tree) from each class satisfying the conditions for node  $i$ .

`P = classcount(t,nodes)` takes a vector `nodes` of node numbers and returns the class counts for the specified nodes.

**Examples** Create a classification tree for Fisher's iris data:

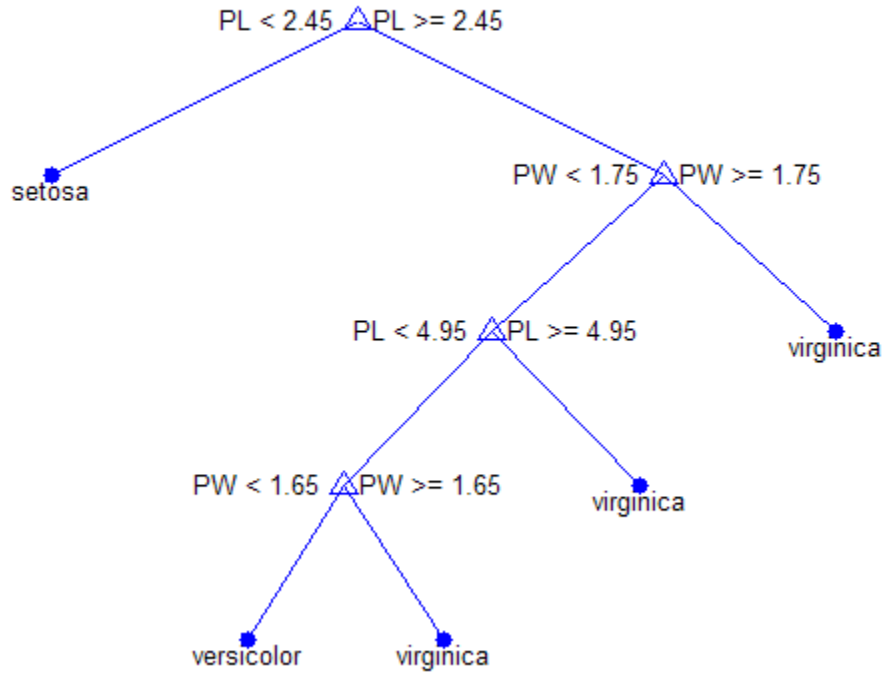
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```



Click to display:  Magnification:  Pruning level:



P = classcount(t)

P =

50	50	50
50	0	0
0	50	50
0	49	5
0	1	45
0	47	1
0	2	4
0	47	0

# classregtree.classcount

---

0 0 1

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree | numnodes

# ClassificationBaggedEnsemble

---

<b>Purpose</b>	Classification ensemble grown by resampling
<b>Description</b>	ClassificationBaggedEnsemble combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners.
<b>Construction</b>	<code>ens = fitensemble(X,Y,'bag',nlearn,learners,'type','classification')</code> creates a bagged classification ensemble. For syntax details, see the <code>fitensemble</code> reference page.
<b>Properties</b>	<p><b>CategoricalPredictors</b></p> <p>List of categorical predictors. <code>CategoricalPredictors</code> is a numeric vector with indices from 1 to <code>p</code>, where <code>p</code> is the number of columns of <code>X</code>.</p> <p><b>CombineWeights</b></p> <p>String describing how <code>ens</code> combines weak learner weights, either <code>'WeightedSum'</code> or <code>'WeightedAverage'</code>.</p> <p><b>FitInfo</b></p> <p>Numeric array of fit information. The <code>FitInfoDescription</code> property describes the content of this array.</p> <p><b>FitInfoDescription</b></p> <p>String describing the meaning of the <code>FitInfo</code> array.</p> <p><b>FResample</b></p> <p>Numeric scalar between 0 and 1. <code>FResample</code> is the fraction of training data <code>fitensemble</code> resampled at random for every weak learner when constructing the ensemble.</p> <p><b>Method</b></p> <p>String describing the method that creates <code>ens</code>.</p> <p><b>ModelParams</b></p>

# ClassificationBaggedEnsemble

---

Parameters used in training `ens`.

## **NTrained**

Number of trained weak learners in `ens`, a scalar.

## **PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in `X`.

## **ReasonForTermination**

String describing the reason `fitensemble` stopped adding weak learners to the ensemble.

## **Replace**

Logical value indicating if the ensemble was trained with replacement (`true`) or without replacement (`false`).

## **ResponseName**

String with the name of the response variable `Y`.

## **ScoreTransform**

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means `@(x)x`. For a list of built-in transformation functions and the syntax of custom transformation functions, see `ClassificationTree.fit`.

Add or change a `ScoreTransform` function by dot addressing:

```
ens.ScoreTransform = 'function'
```

or

```
ens.ScoreTransform = @function
```

## **Trained**

Trained learners, a cell array of compact classification models.

## TrainedWeights

Numeric vector of trained weights for the weak learners in `ens`. `TrainedWeights` has `T` elements, where `T` is the number of weak learners in `learners`.

## UseObsForLearner

Logical matrix of size `N`-by-`NTrained`, where `N` is the number of observations in the training data and `NTrained` is the number of trained weak learners. `UseObsForLearner(I,J)` is `true` if observation `I` was used for training learner `J`, and is `false` otherwise.

## W

Scaled `weights`, a vector with length `n`, the number of rows in `X`. The sum of the elements of `W` is 1.

## X

Matrix of predictor values that trained the ensemble. Each column of `X` represents one variable, and each row represents one observation.

## Y

Numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of `Y` represents the classification of the corresponding row of `X`.

## Methods

<code>oobEdge</code>	Out-of-bag classification edge
<code>oobLoss</code>	Out-of-bag classification error
<code>oobMargin</code>	Out-of-bag classification margins
<code>oobPredict</code>	Predict out-of-bag response of ensemble

# ClassificationBaggedEnsemble

---

## Inherited Methods

<code>compact</code>	Compact classification ensemble
<code>crossval</code>	Cross validate ensemble
<code>resubEdge</code>	Classification edge by resubstitution
<code>resubLoss</code>	Classification error by resubstitution
<code>resubMargin</code>	Classification margins by resubstitution
<code>resubPredict</code>	Predict ensemble response by resubstitution
<code>resume</code>	Resume training ensemble
<code>edge</code>	Classification edge
<code>loss</code>	Classification error
<code>margin</code>	Classification margins
<code>predict</code>	Predict classification
<code>predictorImportance</code>	Estimates of predictor importance
<code>removeLearners</code>	Remove members of compact classification ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Construct a bagged ensemble for the ionosphere data, and examine its resubstitution loss:

```
load ionosphere
rng(0,'twister') % for reproducibility
ens = fitensemble(X,Y,'bag',100,'Tree',...
```

```
        'type', 'classification');  
L = resubLoss(ens)  
  
L =  
    0
```

The ensemble does a perfect job classifying its training data.

## See Also

[ClassificationEnsemble](#) | [fitensemble](#)

## How To

- “Ensemble Methods” on page 16-59

# ClassificationDiscriminant

---

**Superclasses** CompactClassificationDiscriminant

**Purpose** Discriminant analysis classification

**Description** A ClassificationDiscriminant object encapsulates a discriminant analysis classifier, which is a Gaussian mixture model for data generation. A ClassificationDiscriminant object can predict responses for new data using the predict method. The object contains the data used for training, so can compute resubstitution predictions.

**Construction** `obj = ClassificationDiscriminant.fit(X,Y)` creates a discriminant classification object based on the input variables (also known as predictors, features, or attributes) `X` and output (response) `Y`. For syntax details, see `ClassificationDiscriminant.fit`.

`obj = ClassificationDiscriminant.fit(X,Y,Name,Value)` creates a classifier with additional options specified by one or more `Name, Value` pair arguments. If you use one of the following five options, `obj` is of class `ClassificationPartitionedModel`: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Otherwise, `obj` is of class `ClassificationDiscriminant`.

## Input Arguments

### X

Matrix of numeric predictor values. Each column of `X` represents one variable, and each row represents one observation.

NaN values in `X` are considered missing values. Observations with missing values for `X` are not used in the fit.

### Y

Numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of `Y` represents the classification of the corresponding row of `X`. NaN values in `Y` are considered missing values.

Observations with missing values for `Y` are not used in the fit.



## Properties

### BetweenSigma

p-by-p matrix, the between-class covariance, where p is the number of predictors.

### CategoricalPredictors

List of categorical predictors, always empty ([]) for discriminant analysis.

### ClassNames

List of the elements in the training data Y with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. `ClassNames` has the same data type as the data in the argument Y.

### Coeffs

k-by-k structure of coefficient matrices, where k is the number of classes. `Coeffs(i, j)` contains coefficients of the linear or quadratic boundaries between classes i and j. Fields in `Coeffs(i, j)`:

- `DiscrimType`
- `Class1` — `ClassNames(i)`
- `Class2` — `ClassNames(j)`
- `Const` — A scalar
- `Linear` — A vector with p components, where p is the number of columns in X
- `Quadratic` — p-by-p matrix, exists for quadratic `DiscrimType`

The equation of the boundary between class i and class j is

$$\text{Const} + \text{Linear} * x + x' * \text{Quadratic} * x = 0,$$

where x is a column vector of length p.

# ClassificationDiscriminant

---

If `ClassificationDiscriminant.fit` had the `FillCoeffs` name-value pair set to `'off'` when constructing the classifier, `Coeffs` is empty (`[]`).

## Cost

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. `Cost` is `K-by-K`, where `K` is the number of classes.

Change a `Cost` matrix using dot addressing:

```
obj.Cost = costMatrix
```

## Delta

Value of the `Delta` threshold for a linear discriminant model, a nonnegative scalar. If a coefficient of `obj` has magnitude smaller than `Delta`, `obj` sets this coefficient to 0, and so you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Change `Delta` by dot addressing:

```
obj.Delta = newDelta
```

## DeltaPredictor

Row vector of length equal to the number of predictors in `obj`. If `DeltaPredictor(i) < Delta` then coefficient `i` of the model is 0.

If `obj` is a quadratic discriminant model, all elements of `DeltaPredictor` are 0.

## DiscrimType

String specifying the discriminant type. One of:

- `'linear'`
- `'quadratic'`

- 'diagLinear'
- 'diagQuadratic'
- 'pseudoLinear'
- 'pseudoQuadratic'

Change `DiscrimType` using dot addressing:

```
obj.DiscrimType = newDiscrimType
```

You can change between linear types, or between quadratic types, but cannot change between linear and quadratic types.

## **Gamma**

Value of the Gamma regularization parameter, a scalar from 0 to 1. Change Gamma using dot addressing:

```
obj.Gamma = newGamma
```

- If you set 1 for linear discriminant, the discriminant sets its type to 'diagLinear'.
- If you set a value between `MinGamma` and 1 for linear discriminant, the discriminant sets its type to 'linear'.
- You cannot set values below the value of the `MinGamma` property.
- For quadratic discriminant, you can set either 0 (for `DiscrimType 'quadratic'`) or 1 (for `DiscrimType 'diagQuadratic'`).

## **LogDetSigma**

Logarithm of the determinant of the within-class covariance matrix. The type of `LogDetSigma` depends on the discriminant type:

- Scalar for linear discriminant analysis

# ClassificationDiscriminant

---

- Vector of length  $K$  for quadratic discriminant analysis, where  $K$  is the number of classes

## **MinGamma**

Nonnegative scalar, the minimal value of the Gamma parameter so that the correlation matrix is invertible. If the correlation matrix is not singular, MinGamma is 0.

## **ModelParams**

Parameters used in training obj.

## **Mu**

Matrix of class means of size  $K$ -by- $p$ , where  $K$  is the number of classes, and  $p$  is the number of predictors. Each row of Mu represents the mean of the multivariate normal distribution of the corresponding class. The class indices are in the **ClassNames** attribute.

## **NObservations**

Number of observations in the training data, a numeric scalar. NObservations can be less than the number of rows of input data  $X$  when there are missing values in  $X$  or response  $Y$ .

## **PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in the training data  $X$ .

## **Prior**

Prior probabilities for each class. Prior is a numeric vector whose entries relate to the corresponding **ClassNames** property.

Add or change a Prior vector using dot addressing:

```
obj.Prior = priorVector
```

## **ResponseName**

String describing the response variable  $Y$ .

## ScoreTransform

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `ClassificationDiscriminant.fit`.

Add or change a ScoreTransform function by dot addressing:

```
cobj.ScoreTransform = 'function'
```

or

```
cobj.ScoreTransform = @function
```

## Sigma

Within-class covariance matrix or matrices. The dimensions depend on `DiscrimType`:

- 'linear' (default) — Matrix of size p-by-p, where p is the number of predictors
- 'quadratic' — Array of size p-by-p-by-K, where K is the number of classes
- 'diagLinear' — Row vector of length p
- 'diagQuadratic' — Array of size 1-by-p-by-K
- 'pseudoLinear' — Matrix of size p-by-p
- 'pseudoQuadratic' — Array of size p-by-p-by-K

## W

Scaled weights, a vector with length n, the number of rows in X.

## X

Matrix of predictor values. Each column of X represents one predictor (variable), and each row represents one observation.

# ClassificationDiscriminant

---

## **Xcentered**

X data with class means subtracted. If  $Y(i)$  is of class  $j$ ,

$$Xcentered(i,:) = X(i,:) - Mu(j,:),$$

where  $Mu$  is the class mean property.

## **Y**

Numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of  $Y$  represents the classification of the corresponding row of  $X$ .

## **Methods**

compact	Compact discriminant analysis classifier
crossval	Cross-validated discriminant analysis classifier
cvshrink	Cross-validate regularization of linear discriminant
fit	Fit discriminant analysis classifier
make	Construct discriminant analysis classifier from parameters
resubEdge	Classification edge by resubstitution
resubLoss	Classification error by resubstitution
resubMargin	Classification margins by resubstitution

resubPredict	Predict resubstitution response of classifier
template	Discriminant analysis classifier template for ensemble

## Inherited Methods

edge	Classification edge
logP	Log of the unconditional probability density
loss	Classification error
mahal	Mahalanobis distance to class means
margin	Classification margins
nLinearCoeffs	Number of nonzero linear coefficients
predict	Predict classification

## Definitions

### Discriminant Classification

The model for discriminant analysis is:

- Each class ( $Y$ ) generates data ( $X$ ) using a multivariate normal distribution. That is, the model assumes  $X$  has a Gaussian mixture distribution (`gmdistribution`).
  - For linear discriminant analysis, the model has the same covariance matrix for each class, only the means vary.
  - For quadratic discriminant analysis, both means and covariances of each class vary.

`predict` classifies so as to minimize the expected classification cost:

# Classification Discriminant

---

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

For details, see “How the predict Method Classifies” on page 15-6.

## Regularization

Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters,  $\gamma$  and  $\delta$ , that control regularization as follows. `cvshrink` helps you select appropriate values of the parameters.

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be



the correlation matrix of the data  $X$ , and let  $\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = \left[ (x - \mu_0)^T D^{-1/2} \right] \left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square brackets. Each component of the vector

$\left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When  $\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$\left| \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

## Copy Semantics

Value. To learn how value classes affect copy operations, see `Copying Objects` in the MATLAB documentation.

## Examples

Create a discriminant analysis classifier for the Fisher iris data:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species)
```

# ClassificationDiscriminant

---

```
obj =  
ClassificationDiscriminant:  
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}  
  ResponseName: 'Y'  
  ClassNames: {'setosa' 'versicolor' 'virginica'}  
  ScoreTransform: 'none'  
  NObservations: 150  
  DiscrimType: 'linear'  
    Mu: [3x4 double]  
    Coeffs: [3x3 struct]
```

## References

[1] Guo, Y., T. Hastie, and R. Tibshirani. *Regularized linear discriminant analysis and its application in microarrays*. *Biostatistics*, Vol. 8, No. 1, pp. 86–100, 2007.

## See Also

CompactClassificationDiscriminant |  
ClassificationDiscriminant.fit

## How To

• “Discriminant Analysis” on page 15-3

<b>Superclasses</b>	CompactClassificationEnsemble
<b>Purpose</b>	Ensemble classifier
<b>Description</b>	ClassificationEnsemble combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners. It also stores data used for training and can compute resubstitution predictions. It can resume training if desired.
<b>Construction</b>	<p><code>ens = fitensemble(X,Y,method,nlearn,learners)</code> returns an ensemble model that can predict responses to data. The ensemble consists of models listed in <code>learners</code>. For more information on the syntax, see the <code>fitensemble</code> function reference page.</p> <p><code>ens = fitensemble(X,Y,method,nlearn,learners,Name,Value)</code> returns an ensemble model with additional options specified by one or more <code>Name,Value</code> pair arguments. For more information on the syntax, see the <code>fitensemble</code> function reference page.</p>
<b>Properties</b>	<p><b>CategoricalPredictors</b></p> <p>List of categorical predictors. <code>CategoricalPredictors</code> is a numeric vector with indices from 1 to <code>p</code>, where <code>p</code> is the number of columns of <code>X</code>.</p> <p><b>ClassNames</b></p> <p>List of the elements in <code>Y</code> with duplicates removed. <code>ClassNames</code> can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. <code>ClassNames</code> has the same data type as the data in the argument <code>Y</code>.</p> <p><b>CombineWeights</b></p> <p>String describing how <code>ens</code> combines weak learner weights, either <code>'WeightedSum'</code> or <code>'WeightedAverage'</code>.</p> <p><b>Cost</b></p>

# ClassificationEnsemble

---

Square matrix where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ .

## **FitInfo**

Numeric array of fit information. The `FitInfoDescription` property describes the content of this array.

## **FitInfoDescription**

String describing the meaning of the `FitInfo` array.

## **LearnerNames**

Cell array of strings with names of weak learners in the ensemble. The name of each learner appears just once. For example, if you have an ensemble of 100 trees, `LearnerNames` is `{'Tree'}`.

## **Method**

String describing the method that creates `ens`.

## **ModelParams**

Parameters used in training `ens`.

## **NObservations**

Numeric scalar containing the number of observations in the training data.

## **NTrained**

Number of trained weak learners in `ens`, a scalar.

## **PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in  $X$ .

## **Prior**

Prior probabilities for each class. `Prior` is a numeric vector whose entries relate to the corresponding `ClassNames` property.

## **ReasonForTermination**

String describing the reason `fitensemble` stopped adding weak learners to the ensemble.

## ResponseName

String with the name of the response variable `Y`.

## ScoreTransform

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means `@(x)x`. For a list of built-in transformation functions and the syntax of custom transformation functions, see `ClassificationTree.fit`.

Add or change a `ScoreTransform` function by dot addressing:

```
ens.ScoreTransform = 'function'
```

or

```
ens.ScoreTransform = @function
```

## Trained

Trained learners, a cell array of compact classification models.

## TrainedWeights

Numeric vector of trained weights for the weak learners in `ens`. `TrainedWeights` has `T` elements, where `T` is the number of weak learners in `learners`.

## UsePredForLearner

Logical matrix of size `P`-by-`NTrained`, where `P` is the number of predictors (columns) in the training data `X`. `UsePredForLearner(i, j)` is true when learner `j` uses predictor `i`, and is false otherwise. For each learner, the predictors have the same order as the columns in the training data `X`.

If the ensemble is not of type `Subspace`, all entries in `UsePredForLearner` are true.

# ClassificationEnsemble

---

## **W**

Scaled **w**eight**s**, a vector with length  $n$ , the number of rows in  $X$ . The sum of the elements of  $W$  is 1.

## **X**

Matrix of predictor values that trained the ensemble. Each column of  $X$  represents one variable, and each row represents one observation.

## **Y**

Numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of  $Y$  represents the classification of the corresponding row of  $X$ .

## **Methods**

compact	Compact classification ensemble
crossval	Cross validate ensemble
resubEdge	Classification edge by resubstitution
resubLoss	Classification error by resubstitution
resubMargin	Classification margins by resubstitution
resubPredict	Predict ensemble response by resubstitution
resume	Resume training ensemble

## **Inherited Methods**

edge	Classification edge
loss	Classification error

margin	Classification margins
predict	Predict classification
predictorImportance	Estimates of predictor importance
removeLearners	Remove members of compact classification ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Construct a boosted classification ensemble for the ionosphere data, using the `AdaBoostM1` method:

```
load ionosphere
ens = fitensemble(X,Y,'AdaBoostM1',100,'Tree')

ens =
classreg.learning.classif.ClassificationEnsemble:
    PredictorNames: {1x34 cell}
    CategoricalPredictors: []
    ResponseName: 'Response'
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    NObservations: 351
    NTrained: 100
    Method: 'AdaBoostM1'
    LearnerNames: {'Tree'}
    ReasonForTermination: [1x77 char]
    FitInfo: [100x1 double]
    FitInfoDescription: [2x83 char]
```

Predict the classification of the mean of X:

```
ypredict = predict(ens,mean(X))

ypredict =
```

# ClassificationEnsemble

---

'g'

## See Also

ClassificationTree | fitensemble | RegressionEnsemble |  
CompactClassificationEnsemble



<b>Purpose</b>	<i>k</i> -nearest neighbor classification
<b>Description</b>	A nearest-neighbor classification object, where both distance metric (“nearest”) and number of neighbors can be altered. The object classifies new observations using the <code>predict</code> method. The object contains the data used for training, so can compute resubstitution predictions.
<b>Construction</b>	<code>mdl = ClassificationKNN.fit(X,Y)</code> creates a <i>k</i> -nearest neighbor classification model. For details, see <code>ClassificationKNN.fit</code> . <code>mdl = ClassificationKNN.fit(X,Y,Name,Value)</code> creates a classifier with additional options specified by one or more <code>Name,Value</code> pair arguments. For details, see <code>ClassificationKNN.fit</code> .
	<b>Input Arguments</b>
	<b>X</b> Matrix of predictor values. Each column of <i>X</i> represents one variable, and each row represents one observation.
	<b>Y</b> A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings, with the same number of rows as <i>X</i> . Each row of <i>Y</i> represents the classification of the corresponding row of <i>X</i> .
<b>Properties</b>	<b>BreakTies</b> String specifying the method <code>predict</code> uses to break ties if multiple classes have the same smallest cost. By default, ties occur when multiple classes have the same number of nearest points among the <i>K</i> nearest neighbors. <ul style="list-style-type: none"><li>• 'nearest' — Use the class with the nearest neighbor among tied groups.</li><li>• 'random' — Use a random tiebreaker among tied groups.</li></ul>

# ClassificationKNN

---

- 'smallest' — Use the smallest index among tied groups.

'BreakTies' applies when 'IncludeTies' is false.

Change BreakTies using dot addressing:

```
mdl.BreakTies = newBreakTies
```

## CategoricalPredictors

Specification of which predictors are categorical.

- 'all' — All predictors are categorical.
- [] — No predictors are categorical.

## ClassNames

List of elements in the training data Y with duplicates removed. ClassNames can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. ClassNames has the same data type as the data in the argument Y.

Change ClassNames using dot addressing:

```
mdl.ClassNames = newClassNames
```

## Cost

Square matrix, where  $Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . Cost is K-by-K, where K is the number of classes.

Change a Cost matrix using dot addressing:

```
mdl.Cost = costMatrix
```

## Distance

String or function handle specifying the distance metric. The allowable strings depend on the NSMethod parameter, which you

set in `ClassificationKNN.fit`, and which exists as a field in `ModelParams`.

<b>NSMethod</b>	<b>Distance Metric Names</b>
exhaustive	Any distance metric of <code>ExhaustiveSearcher</code>
kdtree	'cityblock', 'chebychev', 'euclidean', or 'minkowski'

For definitions, see “Distance Metrics” on page 16-9.

The distance metrics of `ExhaustiveSearcher`:

<b>Value</b>	<b>Description</b>
'cityblock'	City block distance.
'chebychev'	Chebychev distance (maximum coordinate difference).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.

# ClassificationKNN

Value	Description
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix <b>C</b> . The default value of <b>C</b> is the sample covariance matrix of <b>X</b> , as computed by <code>nancov(X)</code> . To specify a different value for <b>C</b> , use the 'Cov' name-value pair.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' name-value pair.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between <b>X</b> and a query point is scaled, meaning divided by a scale value <b>S</b> . The default value of <b>S</b> is the standard deviation computed from <b>X</b> , <code>S = nanstd(X)</code> . To specify another value for <b>S</b> , use the <code>Scale</code> name-value pair.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	Distance function handle. <i>distfun</i> has the form  <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> where <ul style="list-style-type: none"><li>• <b>ZI</b> is a 1-by-N vector containing one row of <b>X</b> or <b>Y</b>.</li><li>• <b>ZJ</b> is an M2-by-N matrix containing multiple rows of <b>X</b> or <b>Y</b>.</li></ul>

Value	Description
	<ul style="list-style-type: none"> <li>D2 is an M2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ (J, :).</li> </ul>

Change Distance using dot addressing:

```
mdl.Distance = newDistance
```

If NSMethod is kdtree, you can use dot addressing to change Distance only among the types 'cityblock', 'chebychev', 'euclidean', or 'minkowski'.

### DistanceWeight

String or function handle specifying the distance weighting function.

DistanceWeight	Meaning
'equal'	No weighting
'inverse'	Weight is 1/distance
'inversesquared'	Weight is 1/distance <sup>2</sup>
@fcn	<i>fcn</i> is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'inversesquared' is equivalent to @(d)d.^(-2).

Change DistanceWeight using dot addressing:

```
mdl.DistanceWeight = newDistanceWeight
```

### DistParameter

Additional parameter for the distance metric.

# ClassificationKNN

---

Distance Metric	Parameter
'mahalanobis'	Positive definite covariance matrix C.
'minkowski'	Minkowski distance exponent, a positive scalar.
'seuclidean'	Vector of positive scale values with length equal to the number of columns of X.

For values of the distance metric other than those in the table, `DistParameter` must be []. Change `DistParameter` using dot addressing:

```
mdl.DistParameter = newDistParameter
```

## IncludeTies

Logical value indicating whether `predict` includes all the neighbors whose distance values are equal to the Kth smallest distance. If `IncludeTies` is true, `predict` includes all these neighbors. Otherwise, `predict` uses exactly K neighbors (see 'BreakTies').

Change `IncludeTies` using dot addressing:

```
mdl.IncludeTies = newIncludeTies
```

## ModelParams

Parameters used in training `mdl`.

## NObservations

Number of observations used in training `mdl`. This can be less than the number of rows in the training data, because data rows containing NaN values are not part of the fit.

## NumNeighbors

Positive integer specifying the number of nearest neighbors in X to find for classifying each point when predicting. Change `NumNeighbors` using dot addressing:

```
mdl.NumNeighbors = newNumNeighbors
```

## **PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in the training data X. Change PredictorNames using dot addressing:

```
mdl.PredictorNames = newPredictorNames
```

## **Prior**

Prior probabilities for each class. Prior is a numeric vector whose entries relate to the corresponding ClassNames property.

Add or change a Prior vector using dot addressing:

```
obj.Prior = priorVector
```

## **ResponseName**

String describing the response variable Y. Change ResponseName using dot addressing:

```
mdl.ResponseName = newResponseName
```

## **W**

Numeric vector of nonnegative weights with the same number of rows as Y. Each entry in W specifies the relative importance of the corresponding observation in Y. Change W using dot addressing:

```
mdl.W = newW
```

## **X**

Numeric matrix of predictor values. Each column of X represents one predictor (variable), and each row represents one observation.

## **Y**

# ClassificationKNN

---

A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings, with the same number of rows as  $X$ .

$Y$  is of the same type as the passed-in  $Y$  data.

## Methods

crossval	Cross-validated $k$ -nearest neighbor classifier
edge	Edge of $k$ -nearest neighbor classifier
fit	Fit $k$ -nearest neighbor classifier
loss	Loss of $k$ -nearest neighbor classifier
margin	Margin of $k$ -nearest neighbor classifier
predict	Predict $k$ -nearest neighbor classification
resubEdge	Edge of $k$ -nearest neighbor classifier by resubstitution
resubLoss	Loss of $k$ -nearest neighbor classifier by resubstitution
resubMargin	Margin of $k$ -nearest neighbor classifier by resubstitution
resubPredict	Predict resubstitution response of $k$ -nearest neighbor classifier
template	$k$ -nearest neighbor classifier template for ensemble



## Definitions

### Prediction

ClassificationKNN predicts the classification of a point  $X_{new}$  using a procedure equivalent to this:

- 1 Find the NumNeighbors points in the training set  $X$  that are nearest to  $X_{new}$ .
- 2 Find the NumNeighbors response values  $Y$  to those nearest points.
- 3 Assign the classification label  $Y_{new}$  that has smallest expected misclassification cost among the values in  $Y$ .

For details, see “Posterior Probability” on page 21-2608 and “Expected Cost” on page 21-2609 in the predict documentation.

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Construct a KNN Classifier

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',5)
```

```
mdl =
```

```
ClassificationKNN:
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}
  ResponseName: 'Y'
```

# ClassificationKNN

---

```
      ClassNames: {'setosa' 'versicolor' 'virginica'}
ScoreTransform: 'none'
NObservations: 150
      Distance: 'euclidean'
      NumNeighbors: 5
```

## Alternatives

`knnsearch` finds the  $k$ -nearest neighbors of points. `rangesearch` finds all the points within a fixed distance. You can use these functions for classification, as shown in “Classifying Query Data Using `knnsearch`” on page 16-18. If you want to perform classification, `ClassificationKNN` can be more convenient, in that you can construct a classifier in one step and classify in other steps. Also, `ClassificationKNN` has cross-validation options.

## See Also

`ClassificationKNN.fit` | `predict` |

## Related Examples

- “Construct a KNN Classifier” on page 16-27
- “Examine the Quality of a KNN Classifier” on page 16-27
- “Predict Classification Based on a KNN Classifier” on page 16-28
- “Modify a KNN Classifier” on page 16-29

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9

## Purpose

Cross-validated classification ensemble

## Description

`ClassificationPartitionedEnsemble` is a set of classification ensembles trained on cross-validated folds. Estimate the quality of classification by cross validation using one or more “kfold” methods: `kfoldPredict`, `kfoldLoss`, `kfoldMargin`, `kfoldEdge`, and `kfoldfun`.

Every “kfold” method uses models trained on in-fold observations to predict response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, every training fold contains roughly 4/5 of the data and every test fold contains roughly 1/5 of the data. The first model stored in `Trained{1}` was trained on `X` and `Y` with the first 1/5 excluded, the second model stored in `Trained{2}` was trained on `X` and `Y` with the second 1/5 excluded, and so on. When you call `kfoldPredict`, it computes predictions for the first 1/5 of the data using the first model, for the second 1/5 of data using the second model, and so on. In short, response for every observation is computed by `kfoldPredict` using the model trained without this observation.

## Construction

`cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a classification ensemble. For syntax details, see the `crossval` method reference page.

`cvens = fitensemble(X,Y,method,nlearn,learners,name,value)` creates a cross-validated ensemble when `name` is one of `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. For syntax details, see the `fitensemble` function reference page.

## Properties

### CategoricalPredictors

List of categorical predictors. `CategoricalPredictors` is a numeric vector with indices from 1 to `p`, where `p` is the number of columns of `X`.

### ClassNames

List of the elements in `Y` with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables (nominal

# ClassificationPartitionedEnsemble

---

or ordinal), logical vector, character array, or cell array of strings. `ClassNames` has the same data type as the data in the argument `Y`.

## **Combiner**

Cell array of combiners across all folds.

## **Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`.

## **CrossValidatedModel**

Name of the cross-validated model, a string.

## **Kfold**

Number of folds used in a cross-validated ensemble, a positive integer.

## **ModelParams**

Object holding parameters of `cvens`.

## **NObservations**

Number of data points used in training the ensemble, a positive integer.

## **NTrainedPerFold**

Number of data points used in training each fold of the ensemble, a positive integer.

## **Partition**

Partition of class `cvpartition` used in creating the cross-validated ensemble.

## **PredictorNames**

Cell array of names for the predictor variables, in the order in which they appear in `X`.

## **Prior**

Prior probabilities for each class. `Prior` is a numeric vector whose entries relate to the corresponding `ClassNames` property.

## **ResponseName**

Name of the response variable `Y`, a string.

## **ScoreTransform**

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `ClassificationTree.fit`.

Add or change a `ScoreTransform` function by dot addressing:

```
ens.ScoreTransform = 'function'
```

or

```
ens.ScoreTransform = @function
```

## **Trainable**

Cell array of ensembles trained on cross-validation folds. Every ensemble is full, meaning it contains its training data and weights.

## **Trained**

Cell array of compact ensembles trained on cross-validation folds.

## **W**

Scaled `weights`, a vector with length `n`, the number of rows in `X`.

## **X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

## **Y**

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

# ClassificationPartitionedEnsemble

---

## Methods

kfoldEdge	Classification edge for observations not used for training
kfoldLoss	Classification loss for observations not used for training
resume	Resume training learners on cross-validation folds

## Inherited Methods

kfoldEdge	Classification edge for observations not used for training
kfoldfun	Cross validate function
kfoldLoss	Classification loss for observations not used for training
kfoldMargin	Classification margins for observations not used for training
kfoldPredict	Predict response for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Evaluate the k-fold cross-validation error for a classification ensemble that models the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
cvens = crossval(ens);
L = kfoldLoss(cvens)
```

L =  
0.0533

## See Also

[RegressionPartitionedEnsemble](#) |  
[ClassificationPartitionedModel](#) | [ClassificationEnsemble](#)

# ClassificationPartitionedModel

---

**Purpose** Cross-validated classification model

**Description** ClassificationPartitionedModel is a set of classification models trained on cross-validated folds. Estimate the quality of classification by cross validation using one or more “kfold” methods: `kfoldPredict`, `kfoldLoss`, `kfoldMargin`, `kfoldEdge`, and `kfoldfun`.

Every “kfold” method uses models trained on in-fold observations to predict response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, every training fold contains roughly 4/5 of the data and every test fold contains roughly 1/5 of the data. The first model stored in `Trained{1}` was trained on `X` and `Y` with the first 1/5 excluded, the second model stored in `Trained{2}` was trained on `X` and `Y` with the second 1/5 excluded, and so on. When you call `kfoldPredict`, it computes predictions for the first 1/5 of the data using the first model, for the second 1/5 of data using the second model, and so on. In short, response for every observation is computed by `kfoldPredict` using the model trained without this observation.

**Construction** `cvmodel = crossval(obj)` creates a cross-validated classification model from a classification model.

`cvmodel = ClassificationTree.fit(X,Y,name,value)` or `cvmodel = ClassificationDiscriminant.fit(X,Y,name,value)` creates a cross-validated model when `name` is one of `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. For syntax details, see the `ClassificationTree.fit` or `ClassificationDiscriminant.fit` function reference pages.

## Input Arguments

### **obj**

A classification model. `obj` can be a classification tree constructed using `ClassificationTree.fit`, or a discriminant analysis classifier constructed using `ClassificationDiscriminant.fit`.



## Properties

### CategoricalPredictors

List of categorical predictors. `CategoricalPredictors` is a numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .

### ClassNames

List of the elements in  $Y$  with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. `ClassNames` has the same data type as the data in the argument  $Y$ .

### Cost

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ .

If `cvmodel` is a cross-validated `ClassificationDiscriminant` model, you can change a `Cost` matrix by dot addressing:

```
cvmodel.Cost = costMatrix
```

### CrossValidatedModel

Name of the cross-validated model, a string.

### Kfold

Number of folds used in cross-validated model, a positive integer.

### ModelParams

Object holding parameters of `cvmodel`.

### Partition

The partition of class `cvpartition` used in creating the cross-validated model.

### PredictorNames

A cell array of names for the predictor variables, in the order in which they appear in  $X$ .

# ClassificationPartitionedModel

---

## Prior

Prior probabilities for each class. Prior is a numeric vector whose entries relate to the corresponding ClassNames property.

If cvmodel is a cross-validated ClassificationDiscriminant model, you can change a Prior vector by dot addressing:

```
cvmodel.Prior = priorVector
```

## ResponseName

Name of the response variable Y, a string.

## ScoreTransform

Function handle for transforming scores, or string representing a built-in transformation function.

String	Formula
'symmetric'	$2x - 1$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set score for the class with the largest score to 1, and scores for all other classes to 0.
'symmetricismax'	Set score for the class with the largest score to 1, and scores for all other classes to -1.
'none'	$x$
'logit'	$1/(1 + e^{-x})$
'doublelogit'	$1/(1 + e^{-2x})$
'symmetriclogit'	$2/(1 + e^{-x}) - 1$
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$

You can include your own function handle for transforming scores. Your function should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

Add or change a ScoreTransform function by dot addressing:

```
cvmodel.ScoreTransform = 'function'  
or  
cvmodel.ScoreTransform = @function
```

## Trained

The trained learners, a cell array of compact classification models.

## W

The scaled **w**eight, a vector with length  $n$ , the number of rows in  $X$ .

## X

A matrix of predictor values. Each column of  $X$  represents one variable, and each row represents one observation.

## Y

A numeric column vector with the same number of rows as  $X$ . Each entry in  $Y$  is the response to the data in the corresponding row of  $X$ .

## Methods

kfoldEdge	Classification edge for observations not used for training
kfoldfun	Cross validate function
kfoldLoss	Classification loss for observations not used for training

# ClassificationPartitionedModel

---

kfoldMargin	Classification margins for observations not used for training
kfoldPredict	Predict response for observations not used for training

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Evaluate the k-fold cross-validation error for a classification tree model of the Fisher iris data:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
cvtree = crossval(tree);
L = kfoldLoss(cvtree)
```

```
L =
    0.0600
```

## See Also

ClassificationPartitionedEnsemble |  
RegressionPartitionedModel

## How To

- “Cross Validating a Discriminant Analysis Classifier” on page 15-20

**Superclasses** CompactClassificationTree

**Purpose** Binary decision tree for classification

**Description** A decision tree with binary splits for classification. An object of class `ClassificationTree` can predict responses for new data with the `predict` method. The object contains the data used for training, so can compute resubstitution predictions.

**Construction** `tree = ClassificationTree.fit(X,Y)` returns a classification tree based on the input variables (also known as predictors, features, or attributes) `X` and output (response) `Y`. `tree` is a binary tree, where each branching node is split based on the values of a column of `X`.

`tree = ClassificationTree.fit(X,Y,Name,Value)` fits a tree with additional options specified by one or more `Name,Value` pair arguments. If you use one of the following five options, `tree` is of class `ClassificationPartitionedModel`: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Otherwise, `tree` is of class `ClassificationTree`.

## Input Arguments

### X

A matrix of numeric predictor values. Each column of `X` represents one variable, and each row represents one observation.

NaN values in `X` are taken to be missing values. Observations with all missing values for `X` are not used in the fit. Observations with some missing values for `X` are used to find splits on variables for which these observations have valid values.

### Y

A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of `Y` represents the classification of the corresponding

# ClassificationTree

---

row of  $X$ . For numeric  $Y$ , consider using `RegressionTree.fit` instead of `ClassificationTree.fit`.

NaN values in  $Y$  are taken to be missing values. Observations with missing values for  $Y$  are not used in the fit.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## AlgorithmForCategorical

Algorithm to find the best split on a categorical predictor with  $L$  levels for data with  $K \geq 3$  classes. The available algorithms are:

'Exact'	Consider all $2^{L-1} - 1$ combinations
'PullLeft'	Pull left by purity
'PCA'	Principle component-based partition[1]
'OVAbyClass'	One versus all by class

Descriptions of the algorithms are in “Splitting Categorical Predictors” on page 16-56.

**Default:** `ClassificationTree.fit` selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For  $K = 2$  classes, `ClassificationTree.fit` always performs the exact search.

## CategoricalPredictors

List of categorical predictors. Pass `CategoricalPredictors` as one of:

- A numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .
- A logical vector of length  $p$ , where a true entry means that the corresponding column of  $X$  is a categorical variable.
- 'all', meaning all predictors are categorical.
- A cell array of strings, where each element in the array is the name of a predictor variable. The names must match entries in PredictorNames values.
- A character matrix, where each row of the matrix is a name of a predictor variable. The names must match entries in PredictorNames values. Pad the names with extra blanks so each row of the character matrix has the same length.

**Default:** []

## **ClassNames**

Array of class names. Use the data type that exists in  $Y$ .

Use ClassNames to order the classes or to select a subset of classes for training.

**Default:** The class names that exist in  $Y$

## **Cost**

Square matrix, where  $Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . Alternatively, Cost can be a structure  $S$  having two fields:  $S.ClassNames$  containing the group names as a variable of the same type as  $Y$ , and  $S.ClassificationCosts$  containing the cost matrix.

**Default:**  $Cost(i, j) = 1$  if  $i \neq j$ , and  $Cost(i, j) = 0$  if  $i = j$

## **crossval**

# ClassificationTree

---

Flag to grow a cross-validated decision tree. Possible values are 'on' or 'off'.

If 'on', `ClassificationTree.fit` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the 'kfold', 'holdout', 'leaveout', or 'cvpartition' name-value pair arguments. Note that you can only use one of these four options ('kfold', 'holdout', 'leaveout', or 'cvpartition') at a time when creating a cross-validated tree.

Alternatively, cross validate `tree` later using the `crossval` method.

**Default:** 'off'

## **cvpartition**

Partition created with `cvpartition` to use in a cross-validated tree.

Note that if you use 'cvpartition', you cannot use any of the 'kfold', 'holdout', or 'leaveout' name-value pair arguments.

## **holdout**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1.

Note that if you use 'holdout', you cannot use any of the 'cvpartition', 'kfold', or 'leaveout' name-value pair arguments.

## **kfold**

Number of folds to use in a cross-validated tree, a positive integer.

Note that if you use 'kfold', you cannot use any of the 'cvpartition', 'holdout', or 'leaveout' name-value pair arguments.



**Default:** 10

## **leaveout**

Use leave-one-out cross validation by setting to 'on'.

Note that if you use 'leaveout', you cannot use any of the 'cvpartition', 'holdout', or 'kfold' name-value pair arguments.

## **MaxCat**

`ClassificationTree.fit` splits a categorical predictor using the exact search algorithm if the predictor has at most `MaxCat` levels in the split node. Otherwise, `ClassificationTree.fit` finds the best categorical split using one of the inexact algorithms.

Specify `MaxCat` as a numeric nonnegative scalar value. Passing a small value can lead to loss of accuracy and passing a large value can lead to long computation time and memory overload.

**Default:** 10

## **MergeLeaves**

When 'on', `ClassificationTree.fit` merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', `ClassificationTree.fit` does not merge leaves.

**Default:** 'on'

## **MinLeaf**

Each leaf has at least `MinLeaf` observations per tree leaf. If you supply both `MinParent` and `MinLeaf`, `ClassificationTree.fit` uses the setting that gives larger leaves: `MinParent=max(MinParent,2*MinLeaf)`.

**Default:** 1

# ClassificationTree

---

## **MinParent**

Each branch node in the tree has at least `MinParent` observations. If you supply both `MinParent` and `MinLeaf`, `ClassificationTree.fit` uses the setting that gives larger leaves: `MinParent=max(MinParent,2*MinLeaf)`.

**Default:** 10

## **NVarToSample**

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all'

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

**Default:** {'x1', 'x2', ...}

## **prior**

Prior probabilities for each class. Specify as one of:

- A string:
  - 'empirical' determines class probabilities from class frequencies in `Y`. If you pass observation weights, they are used to compute the class probabilities.
  - 'uniform' sets all class probabilities equal.
- A vector (one scalar value for each class)
- A structure `S` with two fields:

- `S.ClassNames` containing the class names as a variable of the same type as `Y`
- `S.ClassProbs` containing a vector of corresponding probabilities

If you set values for both `weights` and `prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

**Default:** `'empirical'`

## **Prune**

When `'on'`, `ClassificationTree.fit` grows the classification tree, and computes the optimal sequence of pruned subtrees. When `'off'` `ClassificationTree.fit` grows the classification tree without pruning.

**Default:** `'on'`

## **PruneCriterion**

String with the pruning criterion, either `'error'` or `'impurity'`.

**Default:** `'error'`

## **ResponseName**

Name of the response variable `Y`, a string.

**Default:** `'Response'`

## **ScoreTransform**

Function handle for transforming scores, or string representing a built-in transformation function.

# ClassificationTree

String	Formula
'symmetric'	$2x - 1$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set score for the class with the largest score to 1, and scores for all other classes to 0.
'symmetricismax'	Set score for the class with the largest score to 1, and scores for all other classes to -1.
'none'	$x$
'logit'	$1/(1 + e^{-x})$
'doublelogit'	$1/(1 + e^{-2x})$
'symmetriclogit'	$2/(1 + e^{-x}) - 1$
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$

You can include your own function handle for transforming scores. Your function should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

**Default:** 'none'

## SplitCriterion

Criterion for choosing a split. One of 'gdi' (Gini's diversity index), 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction (also known as cross entropy).

**Default:** 'gdi'

## Surrogate

String describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', `ClassificationTree.fit` finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, `ClassificationTree.fit` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `ClassificationTree.fit` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## **weights**

Vector of observation weights. The length of `weights` is the number of rows in `X`. `ClassificationTree.fit` normalizes the weights in each class to add up to the value of the prior probability of the class.

**Default:** `ones(size(X,1),1)`

## **Properties**

### **CategoricalPredictors**

List of categorical predictors, a numeric vector with indices from 1 to `p`, where `p` is the number of columns of `X`.

### **CatSplit**

An  $n$ -by-2 cell array, where  $n$  is the number of categorical splits in `tree`. Each row in `CatSplit` gives left and right values for a categorical split. For each branch node with categorical split  $j$

# ClassificationTree

---

based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in `CatSplit(j,1)` and the right child is chosen if  $z$  is in `CatSplit(j,2)`. The splits are in the same order as nodes of the tree. Find the nodes for these splits by selecting 'categorical' cuts from top to bottom in the `CutType` property.

## Children

An  $n$ -by-2 array containing the numbers of the child nodes for each node in `tree`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

## ClassCount

An  $n$ -by- $k$  array of class counts for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class counts `ClassCount(i,:)` are counts of observations (from the data used in fitting the tree) from each class satisfying the conditions for node  $i$ .

## ClassNames

List of the elements in  $Y$  with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. `ClassNames` has the same data type as the data in the argument  $Y$ .

## ClassProb

An  $n$ -by- $k$  array of class probabilities for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class probabilities `ClassProb(i,:)` are the estimated probabilities for each class for a point satisfying the conditions for node  $i$ .

## Cost

Square matrix, where `Cost(i,j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ .

## CutCategories

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `CutCategories{i,1}`, and the right child is chosen if  $x$  is among those listed in `CutCategories{i,2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **CutPoint**

An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if  $x < \text{CutPoint}(i)$  and the right child is chosen if  $x \geq \text{CutPoint}(i)$ . `CutPoint` is NaN for branch nodes based on categorical predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **CutType**

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .
- 'categorical' — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **CutVar**

# ClassificationTree

---

An  $n$ -element cell array of the names of the variables used for branching in each node in `tree`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `CutVar` contains an empty string.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **IsBranch**

An  $n$ -element logical vector that is true for each branch node and false for each leaf node of `tree`.

## **ModelParams**

Parameters used in training `tree`.

## **NObservations**

Number of observations in the training data, a numeric scalar. `NObservations` can be less than the number of rows of input data  $X$  when there are missing values in  $X$  or response  $Y$ .

## **NodeClass**

An  $n$ -element cell array with the names of the most probable classes in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element of this array is a string equal to one of the class names in `ClassNames`.

## **NodeErr**

An  $n$ -element vector of the errors of the nodes in `tree`, where  $n$  is the number of nodes. `NodeErr(i)` is the misclassification probability for node  $i$ .

## **NodeProb**

An  $n$ -element vector of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.



## **NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the measure of impurity (Gini index or deviance) for this node weighted by the node probability. If the tree is grown by twining, the risk for each node is zero.

## **NodeSize**

An  $n$ -element vector of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

## **NumNodes**

The number of nodes in `tree`.

## **Parent**

An  $n$ -element vector containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

## **Prior**

Prior probabilities for each class. `Prior` is a numeric vector whose entries relate to the corresponding `ClassNames` property.

## **PruneAlpha**

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

## **PruneList**

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

## **ResponseName**

String describing the response variable  $Y$ .

## **ScoreTransform**

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ .

Add or change a `ScoreTransform` function by dot addressing:

```
tree.ScoreTransform = 'function'  
or  
tree.ScoreTransform = @function
```

## **SurrCutCategories**

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutCategories{k}` is a cell array. The length of `SurrCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutCategories{k}` is either an empty string for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split, and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutCategories` contains an empty cell.

## **SurrCutFlip**

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutFlip{k}` is a numeric vector. The length of `SurrCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutFlip` contains an empty array.

## **SurrCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutPoint{k}` is a numeric vector. The length of `SurrCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutPoint{k}` is either NaN for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and `SurrCutFlip` for this surrogate split is  $+1$ , or if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables returned by `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutPoint` contains an empty cell.

## **SurrCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

## **SurrCutVar**

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrCutVar` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutVar` contains an empty cell.

## **SurrVarAssoc**

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrVarAssoc{k}` is a numeric vector. The length of `SurrVarAssoc{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrVarAssoc{k}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is the order of variables in `SurrCutVar`.

The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrVarAssoc` contains an empty cell.

## **W**

The scaled `weights`, a vector with length `n`, the number of rows in `X`.

## **X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

## **Y**

A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of `Y` represents the classification of the corresponding row of `X`.

## **Methods**

<code>compact</code>	Compact tree
<code>crossval</code>	Cross-validated decision tree
<code>cvloss</code>	Classification error by cross validation
<code>fit</code>	Fit classification tree
<code>prune</code>	Produce sequence of subtrees by pruning
<code>resubEdge</code>	Classification edge by resubstitution
<code>resubLoss</code>	Classification error by resubstitution
<code>resubMargin</code>	Classification margins by resubstitution

# ClassificationTree

---

resubPredict	Predict resubstitution response of tree
template	Create classification template

## Inherited Methods

edge	Classification edge
loss	Classification error
margin	Classification margins
meanSurrVarAssoc	Mean predictive measure of association for surrogate splits in decision tree
predict	Predict classification
predictorImportance	Estimates of predictor importance
view	View tree

## Definitions

### Impurity and Node Error

`ClassificationTree` splits nodes based on either *impurity* or *node error*. Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a *pure* node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('`deviance`') — With  $p(i)$  defined as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R) \left( \sum_i |L(i) - R(i)| \right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Construct a classification tree for the data in `ionosphere.mat`:

```
load ionosphere
tc = ClassificationTree.fit(X,Y)

tc =

ClassificationTree
```

# ClassificationTree

---

```
PredictorNames: {1x34 cell}
  ResponseName: 'Y'
    ClassNames: {'b' 'g'}
  ScoreTransform: 'none'
CategoricalPredictors: []
NObservations: 351
```

Properties, Methods

## See Also

[RegressionTree](#) | [ClassificationEnsemble](#) |  
[ClassificationTree.fit](#) | [CompactClassificationTree](#) | [predict](#)

## How To

- “Classification Trees and Regression Trees” on page 16-31



**Purpose**

Discriminant analysis

**Syntax**

```
class = classify(sample,training,group)
class = classify(sample,training,group,'type')
class = classify(sample,training,group,'type',prior)
[class,err] = classify(...)
[class,err,POSTERIOR] = classify(...)
[class,err,POSTERIOR,logp] = classify(...)
[class,err,POSTERIOR,logp,coeff] = classify(...)
```

**Description**

`class = classify(sample,training,group)` classifies each row of the data in `sample` into one of the groups in `training`. `sample` and `training` must be matrices with the same number of columns. `group` is a grouping variable for `training`. Its unique values define groups; each element defines the group to which the corresponding row of `training` belongs. `group` can be a categorical variable, a numeric vector, a string array, or a cell array of strings. `training` and `group` must have the same number of rows. `classify` treats NaNs or empty strings in `group` as missing values, and ignores the corresponding rows of `training`. The output `class` indicates the group to which each row of `sample` has been assigned, and is of the same type as `group`.

`class = classify(sample,training,group,'type')` allows you to specify the type of discriminant function. Specify `type` inside single quotes. `type` is one of:

- `linear` — Fits a multivariate normal density to each group, with a pooled estimate of covariance. This is the default.
- `diaglinear` — Similar to `linear`, but with a diagonal covariance matrix estimate (naive Bayes classifiers).
- `quadratic` — Fits multivariate normal densities with covariance estimates stratified by group.
- `diagquadratic` — Similar to `quadratic`, but with a diagonal covariance matrix estimate (naive Bayes classifiers).
- `mahalanobis` — Uses Mahalanobis distances with stratified covariance estimates.

# classify

---

`class = classify(sample, training, group, 'type', prior)` allows you to specify prior probabilities for the groups. *prior* is one of:

- A numeric vector the same length as the number of unique values in `group` (or the number of levels defined for `group`, if `group` is categorical). If `group` is numeric or categorical, the order of *prior* must correspond to the ordered values in `group`, or, if `group` contains strings, to the order of first occurrence of the values in `group`.
- A 1-by-1 structure with fields:
  - `prob` — A numeric vector.
  - `group` — Of the same type as `group`, containing unique values indicating the groups to which the elements of `prob` correspond.

As a structure, *prior* can contain groups that do not appear in `group`. This can be useful if `training` is a subset a larger training set. `classify` ignores any groups that appear in the structure but not in the `group` array.

- The string 'empirical', indicating that group prior probabilities should be estimated from the group relative frequencies in `training`.

*prior* defaults to a numeric vector of equal probabilities, i.e., a uniform distribution. *prior* is not used for discrimination by Mahalanobis distance, except for error rate calculation.

`[class, err] = classify(...)` also returns an estimate `err` of the misclassification error rate based on the `training` data. `classify` returns the apparent error rate, i.e., the percentage of observations in `training` that are misclassified, weighted by the prior probabilities for the groups.

`[class, err, POSTERIOR] = classify(...)` also returns a matrix `POSTERIOR` of estimates of the posterior probabilities that the *j*th training group was the source of the *i*th sample observation, i.e.,  $Pr(\text{group } j | \text{obs } i)$ . `POSTERIOR` is not computed for Mahalanobis discrimination.

`[class, err, POSTERIOR, logp] = classify(...)` also returns a vector `logp` containing estimates of the logarithms of the unconditional

predictive probability density of the sample observations,  $p(\text{obs } i) = \sum p(\text{obs } i | \text{group } j) Pr(\text{group } j)$  over all groups. `logp` is not computed for Mahalanobis discrimination.

`[class,err,POSTERIOR,logp,coeff] = classify(...)` also returns a structure array `coeff` containing coefficients of the boundary curves between pairs of groups. Each element `coeff(I,J)` contains information for comparing group I to group J in the following fields:

- `type` — Type of discriminant function, from the `type` input.
- `name1` — Name of the first group.
- `name2` — Name of the second group.
- `const` — Constant term of the boundary equation (K)
- `linear` — Linear coefficients of the boundary equation (L)
- `quadratic` — Quadratic coefficient matrix of the boundary equation (Q)

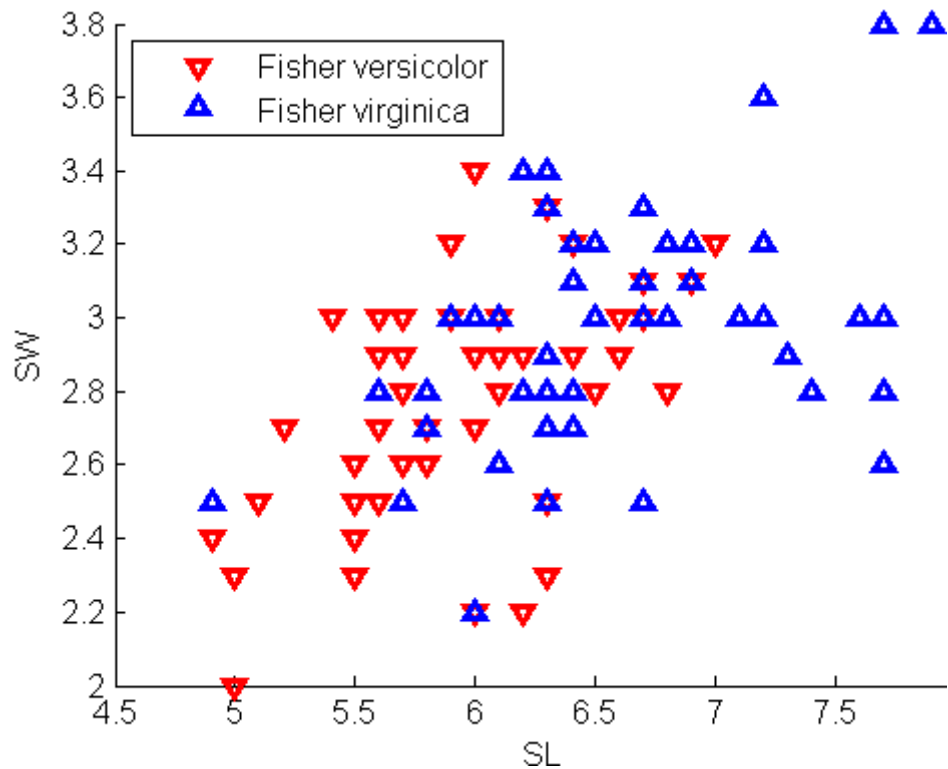
For the `linear` and `diaglinear` types, the `quadratic` field is absent, and a row `x` from the sample array is classified into group I rather than group J if  $0 < K+x*L$ . For the other types, `x` is classified into group I if  $0 < K+x*L+x*Q*x'$ .

## Examples

For training data, use Fisher's sepal measurements for iris versicolor and virginica:

```
load fisheriris
SL = meas(51:end,1);
SW = meas(51:end,2);
group = species(51:end);
h1 = gscatter(SL,SW,group,'rb','v^',[],'off');
set(h1,'LineWidth',2)
legend('Fisher versicolor','Fisher virginica',...
      'Location','NW')
```

# classify



Classify a grid of measurements on the same scale:

```
[X,Y] = meshgrid(linspace(4.5,8),linspace(2,4));  
X = X(:); Y = Y(:);  
[C,err,P,logp,coeff] = classify([X Y],[SL SW],...  
                               group,'quadratic');
```

Visualize the classification:

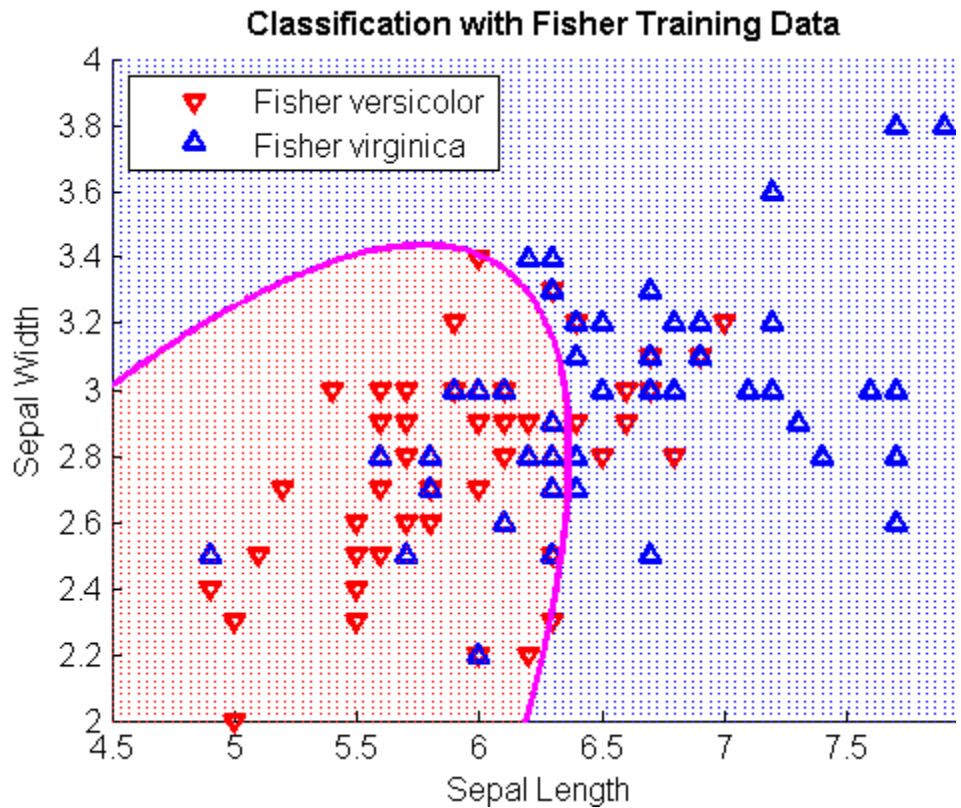
```
hold on;  
gscatter(X,Y,C,'rb','.',1,'off');  
K = coeff(1,2).const;  
L = coeff(1,2).linear;
```

```

Q = coeff(1,2).quadratic;
% Function to compute K + L*v + v'*Q*v for multiple vectors
% v=[x;y]. Accepts x and y as scalars or column vectors.
f = @(x,y) K + [x y]*L + sum(([x y]*Q) .* [x y], 2);

h2 = ezplot(f,[4.5 8 2 4]);
set(h2,'Color','m','LineWidth',2)
axis([4.5 8 2 4])
xlabel('Sepal Length')
ylabel('Sepal Width')
title('\bf Classification with Fisher Training Data')

```



# classify

---

## References

[1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

[2] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

`classregtree` | `mahal` | `NaiveBayes`

## How To

• “Grouping Variables” on page 2-51

- Purpose** Class names for classification decision tree
- Syntax** CNames = classname(T)  
CNames = classname(T,J)
- Description** CNames = classname(T) returns a cell array of strings with class names for this classification decision tree.  
CNames = classname(T,J) takes an array J of class numbers and returns the class names for the specified numbers.
- See Also** classregtree

# TreeBagger.ClassNames property

---

**Purpose** Names of classes

**Description** The `ClassNames` property is a cell array containing the class names for the response variable  $Y$ . This property is empty for regression trees.



## Purpose

Class probabilities

## Syntax

```
P = classprob(t)
P = classprob(t,nodes)
```

## Description

`P = classprob(t)` returns an  $n$ -by- $m$  array  $P$  of class probabilities for the nodes in the classification tree  $t$ , where  $n$  is the number of nodes and  $m$  is the number of classes. For any node number  $i$ , the class probabilities  $P(i, :)$  are the estimated probabilities for each class for a point satisfying the conditions for node  $i$ .

`P = classprob(t,nodes)` takes a vector `nodes` of node numbers and returns the class probabilities for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

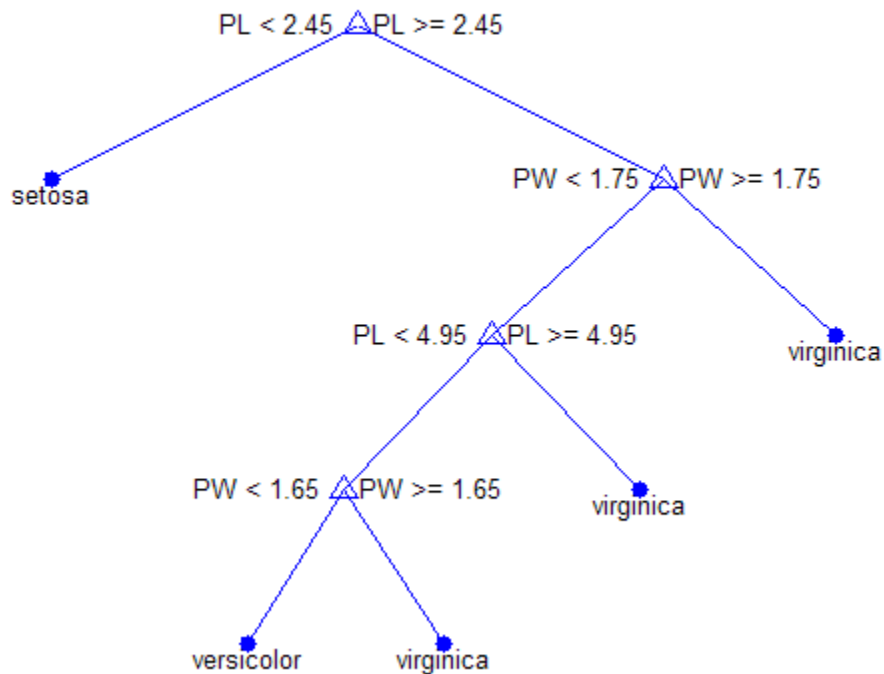
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

# classregtree.classprob

Click to display:  Magnification:  Pruning level:



P = classprob(t)

P =

0.3333	0.3333	0.3333
1.0000	0	0
0	0.5000	0.5000
0	0.9074	0.0926
0	0.0217	0.9783
0	0.9792	0.0208
0	0.3333	0.6667
0	1.0000	0

0 0 1.0000

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree | numnodes

# classregtree

---

**Purpose** Classification and regression trees

---

**Note** Use `ClassificationTree` and `RegressionTree` classes instead. This class is superseded by `ClassificationTree` and `RegressionTree` classes and is maintained only for backwards compatibility.

---

**Construction** `classregtree` Construct classification and regression trees

**Methods**

<code>catsplit</code>	Categorical splits used for branches in decision tree
<code>children</code>	Child nodes
<code>classcount</code>	Class counts
<code>classname</code>	Class names for classification decision tree
<code>classprob</code>	Class probabilities
<code>cutcategories</code>	Cut categories
<code>cutpoint</code>	Decision tree cut point values
<code>cuttype</code>	Cut types
<code>cutvar</code>	Cut variable names
<code>disp</code>	Display <code>classregtree</code> object
<code>display</code>	Display <code>classregtree</code> object
<code>eval</code>	Predicted responses
<code>isbranch</code>	Test node for branch

<code>meansurrvarassoc</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>nodeclass</code>	Class values of nodes of classification tree
<code>nodeerr</code>	Return vector of node errors
<code>nodemean</code>	Mean values of nodes of regression tree
<code>nodeprob</code>	Node probabilities
<code>nodesize</code>	Return node size
<code>numnodes</code>	Number of nodes
<code>parent</code>	Parent node
<code>prune</code>	Prune tree
<code>prunelist</code>	Pruning levels for decision tree nodes
<code>risk</code>	Node risks
<code>subsasgn</code>	Subscripted reference for <code>classregtree</code> object
<code>subsref</code>	Subscripted reference for <code>classregtree</code> object
<code>surrutcategories</code>	Categories used for surrogate splits in decision tree
<code>surrutflip</code>	Numeric cutpoint assignments used for surrogate splits in decision tree
<code>surrutpoint</code>	Cutpoints used for surrogate splits in decision tree
<code>surruttype</code>	Types of surrogate splits used at branches in decision tree

# classregtree

---

surrctvar	Variables used for surrogate splits in decision tree
surrvarassoc	Predictive measure of association for surrogate splits in decision tree
test	Error rate
type	Tree type
varimportance	Compute embedded estimates of input feature importance
view	Plot tree

## Properties

Objects of the `classregtree` class have no properties accessible by dot indexing, `get` methods, or `set` methods. To obtain information about a `classregtree` object, use the appropriate method.

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## How To

- “Ensemble Methods” on page 16-59
- “Classification Trees and Regression Trees” on page 16-31
- “Grouping Variables” on page 2-51

**Purpose** Construct classification and regression trees

---

**Note** Use `ClassificationTree.fit` or `RegressionTree.fit` instead.

---

**Syntax**

```
t = classregtree(X,y)
t = classregtree(X,y, 'Name',value)
```

**Description**

---

**Note** This function is superseded by `ClassificationTree.fit` and `RegressionTree.fit` of `ClassificationTree` and `RegressionTree` classes. It is maintained only for backwards compatibility.

---

`t = classregtree(X,y)` creates a decision tree `t` for predicting the response `y` as a function of the predictors in the columns of `X`. `X` is an  $n$ -by- $m$  matrix of predictor values. If `y` is a vector of  $n$  response values, `classregtree` performs regression. If `y` is a categorical variable, character array, or cell array of strings, `classregtree` performs classification. Either way, `t` is a binary tree where each branching node is split based on the values of a column of `X`. NaN values in `X` or `y` are taken to be missing values. Observations with all missing values for `X` or missing values for `y` are not used in the fit. Observations with some missing values for `X` are used to find splits on variables for which these observations have valid values.

`t = classregtree(X,y, 'Name',value)` specifies one or more optional parameter name/value pairs. Specify *Name* in single quotes. The following options are available:

For all trees:

- `categorical` — Vector of indices of the columns of  $X$  that are to be treated as unordered categorical variables
- `method` — Either 'classification' (default if  $y$  is text or a categorical variable) or 'regression' (default if  $y$  is numeric).
- `names` — A cell array of names for the predictor variables, in the order in which they appear in the  $X$  from which the tree was created.
- `prune` — 'on' (default) to compute the full tree and the optimal sequence of pruned subtrees, or 'off' for the full tree without pruning.
- `minparent` — A number  $k$  such that impure nodes must have  $k$  or more observations to be split (default is 10).
- `minleaf` — A minimal number of observations per tree leaf (default is 1). If you supply both 'minparent' and 'minleaf', `classregtree` uses the setting which results in larger leaves: `minparent = max(minparent, 2*minleaf)`
- `mergeleaves` — 'on' (default) to merge leaves that originate from the same parent node and give the sum of risk values greater or equal to the risk associated with the parent node. If 'off', `classregtree` does not merge leaves.
- `nvartosample` — Number of predictor variables randomly selected for each split. By default all variables are considered for each decision split.
- `stream` — Random number stream. Default is the MATLAB default random number stream.
- `surrogate` — 'on' to find surrogate splits at each branch node. Default is 'off'. If you set this parameter to 'on', `classregtree` can run significantly slower and consume significantly more memory.
- `weights` — Vector of observation weights. By default the weight of every observation is 1. The length of this vector must be equal to the number of rows in  $X$ .



For regression trees only:

- `qetoler` — Defines tolerance on quadratic error per node for regression trees. Splitting nodes stops when quadratic error per node drops below `qetoler*qed`, where `qed` is the quadratic error for the entire data computed before the decision tree is grown: `qed = norm(y-ybar)` with `ybar` estimated as the average of the input array `Y`. Default value is `1e-6`.

For classification trees only:

- `cost` — Square matrix `C`, where `C(i,j)` is the cost of classifying a point into class `j` if its true class is `i` (default has `C(i,j)=1` if `i~j`, and `C(i,j)=0` if `i=j`). Alternatively, this value can be a structure `S` having two fields: `S.group` containing the group names as a categorical variable, character array, or cell array of strings; and `S.cost` containing the cost matrix `C`.
- `splitcriterion` — Criterion for choosing a split. One of `'gdi'` (default) or Gini's diversity index, `'twoing'` for the twoing rule, or `'deviance'` for maximum deviance reduction.
- `priorprob` — Prior probabilities for each class, specified as a string (`'empirical'` or `'equal'`) or as a vector (one value for each distinct group name) or as a structure `S` with two fields:
  - `S.group` containing the group names as a categorical variable, character array, or cell array of strings
  - `S.prob` containing a vector of corresponding probabilities.
 If the input value is `'empirical'` (default), class probabilities are determined from class frequencies in `Y`. If the input value is `'equal'`, all class probabilities are set equal. If both observation weights and class prior probabilities are supplied, the weights are renormalized to add up to the value of the prior probability in the respective class.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;
```

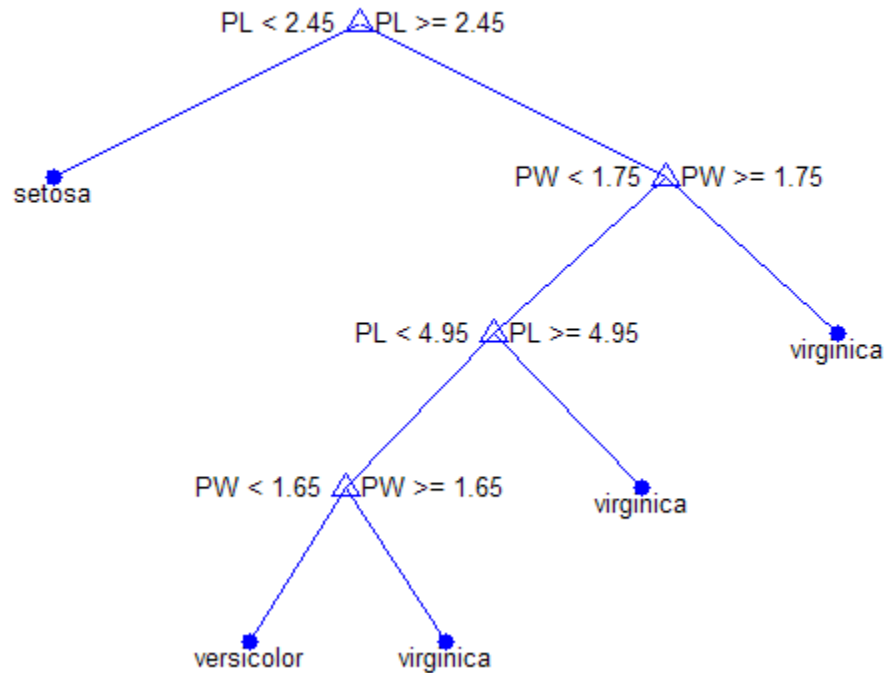
# classregtree

---

```
t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

Click to display:  Magnification:  Pruning level:



## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[eval](#) | [prune](#) | [test](#) | [view](#)

## How To

- “Grouping Variables” on page 2-51
- “Ensemble Methods” on page 16-59

## NaiveBayes.ClassLevels property

---

**Purpose** Class levels

**Description** The `ClassLevels` property is a vector of the same type as the grouping variable, containing the unique levels of the grouping variable.

**Purpose**

Construct agglomerative clusters from linkages

**Syntax**

```
T = cluster(Z, 'cutoff', c)
T = cluster(Z, 'cutoff', c, 'depth', d)
T = cluster(Z, 'cutoff', c, 'criterion', criterion)
T = cluster(Z, 'maxclust', n)
```

**Description**

`T = cluster(Z, 'cutoff', c)` constructs clusters from the agglomerative hierarchical cluster tree, `Z`, as generated by the linkage function. `Z` is a matrix of size  $(m - 1)$ -by-3, where  $m$  is the number of observations in the original data. `c` is a threshold for cutting `Z` into clusters. Clusters are formed when a node and all of its subnodes have inconsistent value less than `c`. All leaves at or below the node are grouped into a cluster. `t` is a vector of size  $m$  containing the cluster assignments of each observation.

If `c` is a vector, `T` is a matrix of cluster assignments with one column per cutoff value.

`T = cluster(Z, 'cutoff', c, 'depth', d)` evaluates inconsistent values by looking to a depth `d` below each node. The default depth is 2.

`T = cluster(Z, 'cutoff', c, 'criterion', criterion)` uses the specified criterion for forming clusters, where `criterion` is one of the strings `'inconsistent'` (default) or `'distance'`. The `'distance'` criterion uses the distance between the two subnodes merged at a node to measure node height. All leaves at or below a node with height less than `c` are grouped into a cluster.

`T = cluster(Z, 'maxclust', n)` constructs a maximum of  $n$  clusters using the `'distance'` criterion. `cluster` finds the smallest height at which a horizontal cut through the tree leaves  $n$  or fewer clusters.

If `n` is a vector, `T` is a matrix of cluster assignments with one column per maximum value.

**Examples**

Compare clusters from Fisher iris data with species:

```
load fisheriris
```

# cluster

---

```
d = pdist(meas);  
Z = linkage(d);  
c = cluster(Z, 'maxclust',3:5);
```

```
crosstab(c(:,1),species)
```

```
ans =  
    0    0    2  
    0   50   48  
   50    0    0
```

```
crosstab(c(:,2),species)
```

```
ans =  
    0    0    1  
    0   50   47  
    0    0    2  
   50    0    0
```

```
crosstab(c(:,3),species)
```

```
ans =  
    0    4    0  
    0   46   47  
    0    0    1  
    0    0    2  
   50    0    0
```

## See Also

[clusterdata](#) | [cophenet](#) | [inconsistent](#) | [linkage](#) | [pdist](#)

**Purpose** Construct clusters from Gaussian mixture distribution

**Syntax**

```
idx = cluster(obj,X)
[idx,nlogl] = cluster(obj,X)
[idx,nlogl,P] = cluster(obj,X)
[idx,nlogl,P,logpdf] = cluster(obj,X)
[idx,nlogl,P,logpdf,M] = cluster(obj,X)
```

**Description** `idx = cluster(obj,X)` partitions data in the  $n$ -by- $d$  matrix  $X$ , where  $n$  is the number of observations and  $d$  is the dimension of the data, into  $k$  clusters determined by the  $k$  components of the Gaussian mixture distribution defined by `obj`. `obj` is an object created by `gmdistribution` or `fit`. `idx` is an  $n$ -by-1 vector, where `idx(I)` is the cluster index of observation  $I$ . The cluster index gives the component with the largest posterior probability for the observation, weighted by the component probability.

---

**Note** The data in  $X$  is typically the same as the data used to create the Gaussian mixture distribution defined by `obj`. Clustering with `cluster` is treated as a separate step, apart from density estimation. For `cluster` to provide meaningful clustering with new data,  $X$  should come from the same population as the data used to create `obj`.

---

`cluster` treats NaN values as missing data. Rows of  $X$  with NaN values are excluded from the partition.

`[idx,nlogl] = cluster(obj,X)` also returns `nlogl`, the negative log-likelihood of the data.

`[idx,nlogl,P] = cluster(obj,X)` also returns the posterior probabilities of each component for each observation in the  $n$ -by- $k$  matrix  $P$ .  $P(I,J)$  is the probability of component  $J$  given observation  $I$ .

`[idx,nlogl,P,logpdf] = cluster(obj,X)` also returns the  $n$ -by-1 vector `logpdf` containing the logarithm of the estimated probability density function for each observation. The density estimate for

## gmdistribution.cluster

---

observation  $I$  is a sum over all components of the component density at  $I$  times the component probability.

`[idx,nlogl,P,logpdf,M] = cluster(obj,X)` also returns an  $n$ -by- $k$  matrix  $M$  containing Mahalanobis distances in squared units.  $M(I,J)$  is the Mahalanobis distance of observation  $I$  from the mean of component  $J$ .

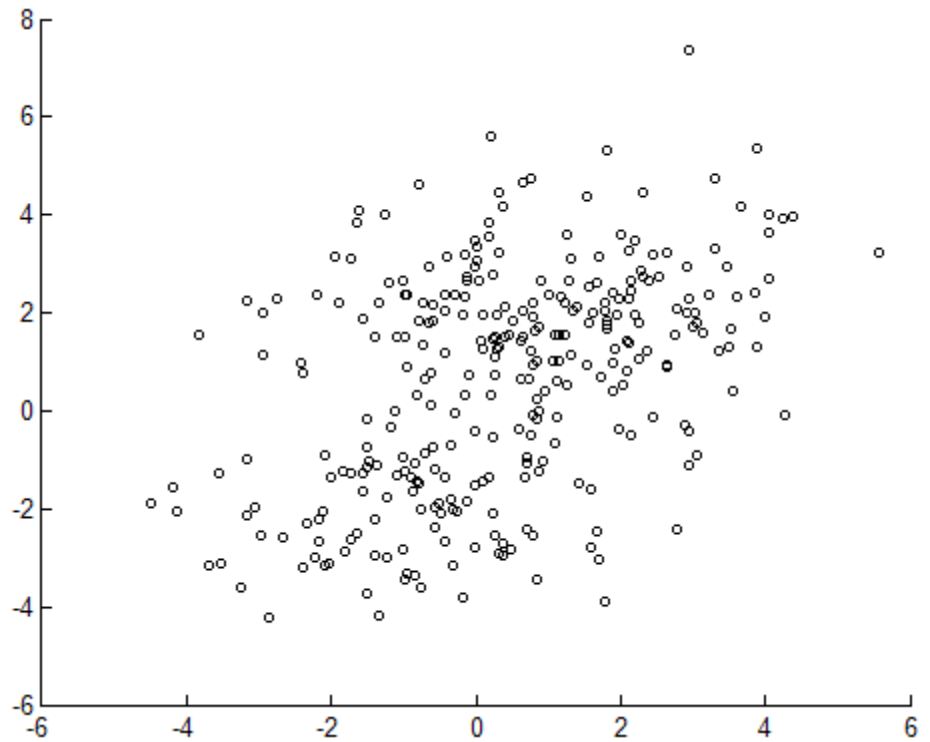
### Examples

Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function:

```
MU1 = [1 2];  
SIGMA1 = [2 0; 0 .5];  
MU2 = [-3 -5];  
SIGMA2 = [1 0; 0 1];  
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];
```

```
scatter(X(:,1),X(:,2),10,'.')  
hold on
```

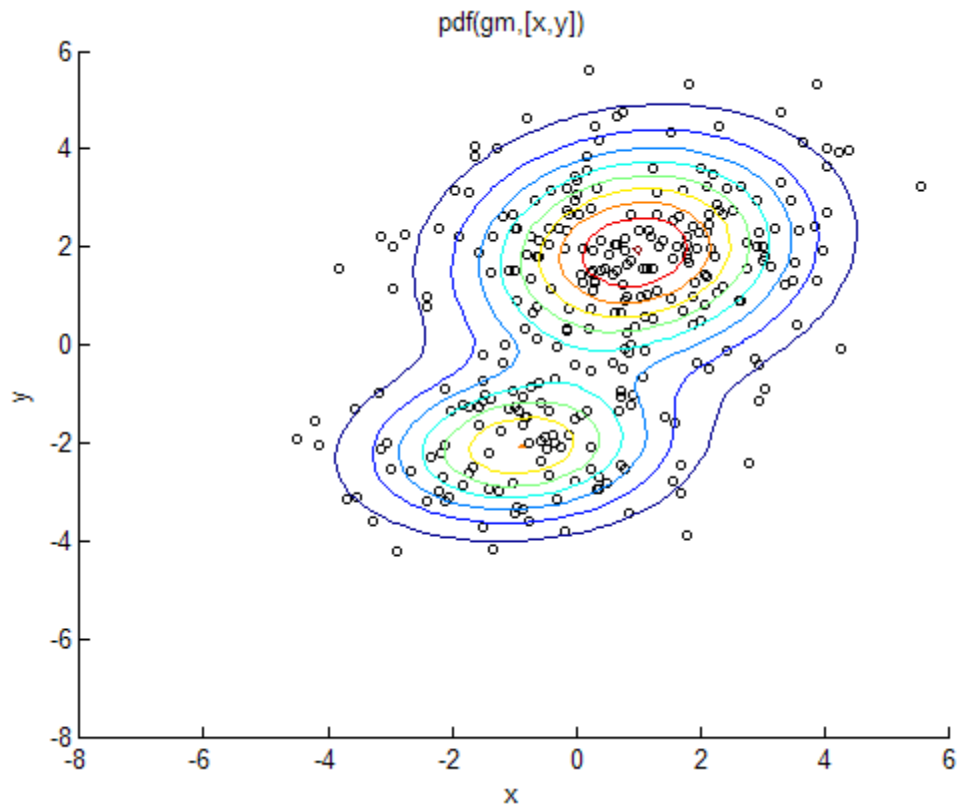




Fit a two-component Gaussian mixture model:

```
obj = gmdistribution.fit(X,2);  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

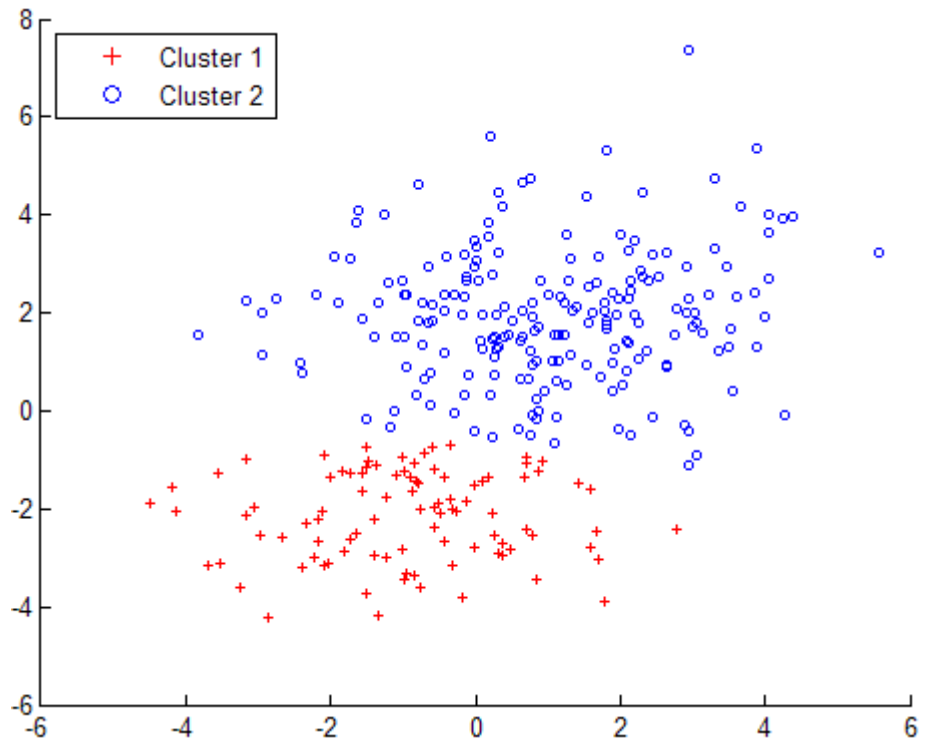
# gmdistribution.cluster



Use the fit to cluster the data:

```
idx = cluster(obj,X);
cluster1 = X(idx == 1,:);
cluster2 = X(idx == 2,:);

delete(h)
h1 = scatter(cluster1(:,1),cluster1(:,2),10,'r. ');
h2 = scatter(cluster2(:,1),cluster2(:,2),10,'g. ');
legend([h1 h2],'Cluster 1','Cluster 2','Location','NW')
```



## See Also

[fit](#) | [gmdistribution](#) | [mahal](#) | [posterior](#)

# clustering.evaluation.ClusterCriterion

---

**Purpose** Clustering evaluation object

**Description** Create a clustering evaluation object using `evalclusters`.

**Properties** **ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name string or function handle. If the clustering solutions are provided in the input, `ClusteringFunction` is empty.

**CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name string.

**CriterionValues**

Criterion values corresponding to each proposed number of clusters in `InspectedK`, stored as a vector of numerical values.

**InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

**Missing**

Logical flag for excluded data, stored as a column vector of logical values. If `Missing` equals `true`, then the corresponding value in the data matrix `x` is not used in the clustering solution.

**NumObservations**

Number of observations in the data matrix `X`, minus the number of missing (NaN) values in `X`, stored as a positive integer value.

**OptimalK**

Optimal number of clusters, stored as a positive integer value.

**OptimalY**

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

## **x**

Data used for clustering, stored as a matrix of numerical values.

## **Methods**

<code>compact</code>	Compact clustering evaluation object
<code>plot</code>	Plot clustering evaluation object criterion values

## **See Also**

`evalclusters`  
`clustering.evaluation.CalinskiHarabaszEvaluation` |  
`clustering.evaluation.DaviesBouldinEvaluation` |  
`clustering.evaluation.GapEvaluation` |  
`clustering.evaluation.SilhouetteEvaluation` |

## **Concepts**

- Class Attributes
- Property Attributes

# clusterdata

---

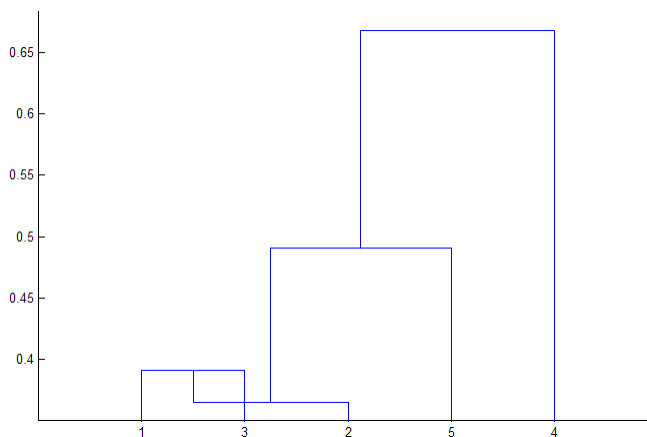
**Purpose** Agglomerative clusters from data

**Syntax**  
`T = clusterdata(X,cutoff)`  
`T = clusterdata(X,Name,Value)`

**Description** `T = clusterdata(X,cutoff)`  
`T = clusterdata(X,Name,Value)` clusters with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- The centroid and median methods can produce a cluster tree that is not monotonic. This occurs when the distance from the union of two clusters,  $r$  and  $s$ , to a third cluster is less than the distance between  $r$  and  $s$ . In this case, in a dendrogram drawn with the default orientation, the path from a leaf to the root node takes some downward steps. To avoid this, use another method. The following image shows a nonmonotonic cluster tree.



In this case, cluster 1 and cluster 3 are joined into a new cluster, while the distance between this new cluster and cluster 2 is less than the distance between cluster 1 and cluster 3. This leads to a nonmonotonic tree.

- You can provide the output  $T$  to other functions including `dendrogram` to display the tree, `cluster` to assign points to clusters, `inconsistent` to compute inconsistent measures, and `cophenet` to compute the cophenetic correlation coefficient.

## Input Arguments

### **X**

Matrix with two or more rows. The rows represent observations, the columns represent categories or dimensions.

### **cutoff**

When  $0 < \text{cutoff} < 2$ , `clusterdata` forms clusters when inconsistent values are greater than `cutoff` (see `inconsistent`). When `cutoff` is an integer  $\geq 2$ , `clusterdata` interprets `cutoff` as the maximum number of clusters to keep in the hierarchical tree generated by `linkage`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **'criterion'**

Either 'inconsistent' or 'distance'.

### **'cutoff'**

Cutoff for inconsistent or distance measure, a positive scalar. When  $0 < \text{cutoff} < 2$ , `clusterdata` forms clusters when inconsistent values are greater than `cutoff` (see `inconsistent`). When `cutoff` is an integer  $\geq 2$ , `clusterdata` interprets `cutoff` as the maximum number of clusters to keep in the hierarchical tree generated by `linkage`.

### **'depth'**

Depth for computing inconsistent values, a positive integer.

## 'distance'

Any of the distance metric names allowed by `pdist` (follow the 'minkowski' option by the value of the exponent `p`):

Metric	Description
'euclidean'	Euclidean distance (default).
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between rows in <code>X</code> is scaled by dividing by the corresponding element of the standard deviation <code>S=nansd(X)</code> . To specify another value for <code>S</code> , use <code>D=pdist(X, 'seuclidean', S)</code> .
'cityblock'	City block metric.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use <code>D = pdist(X, 'minkowski', P)</code> , where <code>P</code> is a scalar
'chebychev'	Chebychev distance (maximum coordinate difference).
'mahalanobis'	Mahalanobis distance, using the sample covariance of <code>X</code> as computed by <code>nancov</code> . To compute the distance with a different covariance, use <code>D = pdist(X, 'mahalanobis', C)</code> , where the matrix <code>C</code> is symmetric and positive definite.
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).



Metric	Description
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
custom distance function	<p>A distance function specified using @:  <math>D = \text{pdist}(X, @\text{distfun})</math></p> <p>A distance function must be of form</p> $d2 = \text{distfun}(XI, XJ)$ <p>taking as arguments a 1-by-<math>n</math> vector <math>XI</math>, corresponding to a single row of <math>X</math>, and an <math>m2</math>-by-<math>n</math> matrix <math>XJ</math>, corresponding to multiple rows of <math>X</math>. <math>\text{distfun}</math> must accept a matrix <math>XJ</math> with an arbitrary number of rows. <math>\text{distfun}</math> must return an <math>m2</math>-by-1 vector of distances <math>d2</math>, whose <math>k</math>th element is the distance between <math>XI</math> and <math>XJ(k, :)</math>.</p>

### 'linkage'

Any of the linkage methods allowed by the `linkage` function:

- 'average'
- 'centroid'
- 'complete'
- 'median'
- 'single'
- 'ward'
- 'weighted'

For details, see the definitions in the `linkage` function reference page.

## 'maxclust'

Maximum number of clusters to form, a positive integer.

## 'savememory'

A string, either 'on' or 'off'. When applicable, the 'on' setting causes clusterdata to construct clusters without computing the distance matrix. savememory is applicable when:

- linkage is 'centroid', 'median', or 'ward'
- distance is 'euclidean' (default)

When savememory is 'on', linkage run time is proportional to the number of dimensions (number of columns of X). When savememory is 'off', linkage memory requirement is proportional to  $N^2$ , where N is the number of observations. So choosing the best (least-time) setting for savememory depends on the problem dimensions, number of observations, and available memory. The default savememory setting is a rough approximation of an optimal setting.

**Default:** 'on' when X has 20 columns or fewer, or the computer does not have enough memory to store the distance matrix; otherwise 'off'

## Output Arguments

### T

T is a vector of size m containing a cluster number for each observation.

- When  $0 < \text{cutoff} < 2$ ,  $T = \text{clusterdata}(X, \text{cutoff})$  is equivalent to:

```
Y = pdist(X, 'euclid');  
Z = linkage(Y, 'single');  
T = cluster(Z, 'cutoff', cutoff);
```

- When cutoff is an integer  $\geq 2$ ,  $T = \text{clusterdata}(X, \text{cutoff})$  is equivalent to:

```
Y = pdist(X, 'euclid');  
Z = linkage(Y, 'single');
```

```
T = cluster(Z, 'maxclust', cutoff);
```

## Examples

### Create Hierarchical Cluster Tree From Sample Data

This example shows how to create a hierarchical cluster tree from sample data, and visualize the clusters using a 3-dimensional scatter plot.

Generate sample data matrices containing random numbers from the standard uniform distribution.

```
rng('default'); % For reproducibility
X = [gallery('uniformdata', [10 3], 12);...
     gallery('uniformdata', [10 3], 13)+1.2;...
     gallery('uniformdata', [10 3], 14)+2.5];
```

Compute the distances between items and create a hierarchical cluster tree from the sample data. List all of the items in cluster 2.

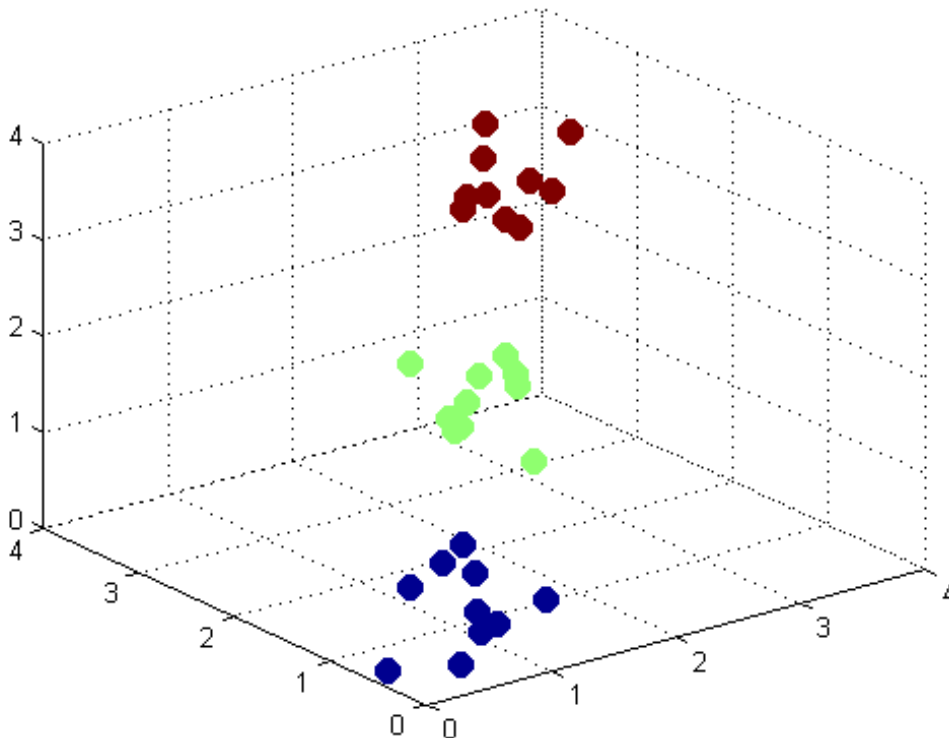
```
T = clusterdata(X, 'maxclust', 3);
find(T==2)
```

```
ans =
```

```
11
12
13
14
15
16
17
18
19
20
```

Plot the data with each cluster shown in a different color.

```
scatter3(X(:,1),X(:,2),X(:,3),100,T,'filled')
```



## Create Hierarchical Cluster Tree Using Ward's Linkage

This example shows how to create a hierarchical cluster tree using Ward's linkage, and visualize the clusters using a 3-dimensional scatter plot.

Create a 20,000-by-3 matrix of sample data generated from the standard uniform distribution.

```
rng('default'); % For reproducibility  
X = rand(20000,3);
```

Create a hierarchical cluster tree from the sample data using Ward's linkage. Set 'savememory' to 'on' to construct clusters without computing the distance matrix.

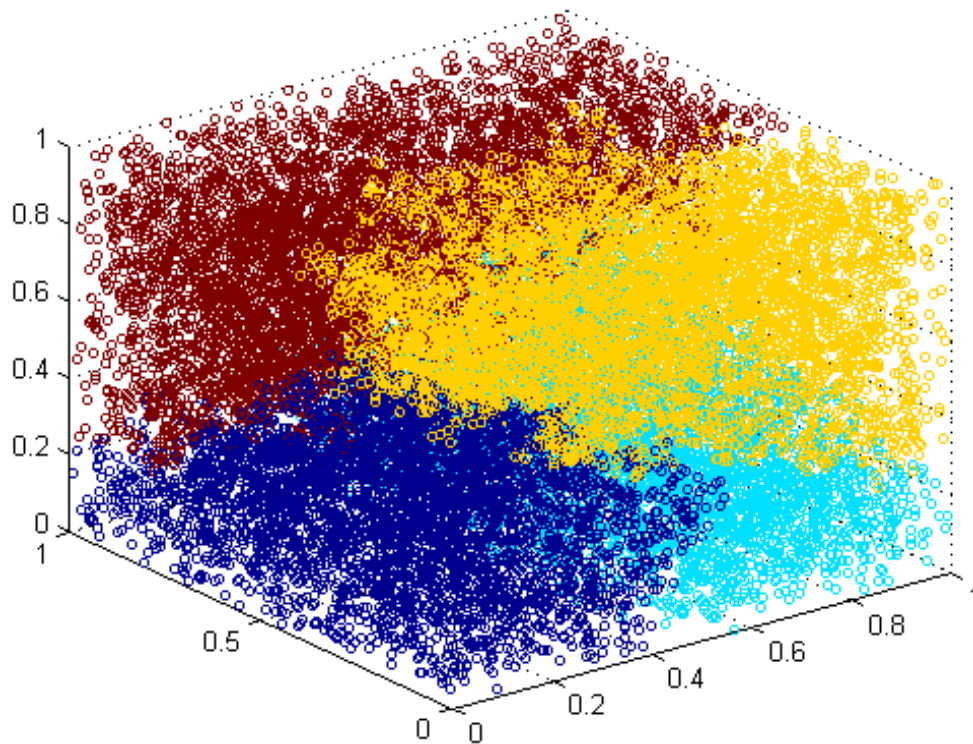
```
c = clusterdata(X,'linkage','ward','savememory','on','maxclust',4);
```

Plot the data with each cluster shown in a different color.

```
scatter3(X(:,1),X(:,2),X(:,3),10,c)
```

# clusterdata

---



## See Also

`cluster` | `inconsistent` | `kmeans` | `linkage` | `pdist`

<b>Purpose</b>	Classical multidimensional scaling
<b>Syntax</b>	$Y = \text{cmdscale}(D)$ $[Y, e] = \text{cmdscale}(D)$
<b>Description</b>	<p><math>Y = \text{cmdscale}(D)</math> takes an <math>n</math>-by-<math>n</math> distance matrix <math>D</math>, and returns an <math>n</math>-by-<math>p</math> configuration matrix <math>Y</math>. Rows of <math>Y</math> are the coordinates of <math>n</math> points in <math>p</math>-dimensional space for some <math>p &lt; n</math>. When <math>D</math> is a Euclidean distance matrix, the distances between those points are given by <math>D</math>. <math>p</math> is the dimension of the smallest space in which the <math>n</math> points whose inter-point distances are given by <math>D</math> can be embedded.</p> <p><math>[Y, e] = \text{cmdscale}(D)</math> also returns the eigenvalues of <math>Y*Y'</math>. When <math>D</math> is Euclidean, the first <math>p</math> elements of <math>e</math> are positive, the rest zero. If the first <math>k</math> elements of <math>e</math> are much larger than the remaining <math>(n-k)</math>, then you can use the first <math>k</math> columns of <math>Y</math> as <math>k</math>-dimensional points whose inter-point distances approximate <math>D</math>. This can provide a useful dimension reduction for visualization, e.g., for <math>k = 2</math>.</p> <p><math>D</math> need not be a Euclidean distance matrix. If it is non-Euclidean or a more general dissimilarity matrix, then some elements of <math>e</math> are negative, and <math>\text{cmdscale}</math> chooses <math>p</math> as the number of positive eigenvalues. In this case, the reduction to <math>p</math> or fewer dimensions provides a reasonable approximation to <math>D</math> only if the negative elements of <math>e</math> are small in magnitude.</p> <p>You can specify <math>D</math> as either a full dissimilarity matrix, or in upper triangle vector form such as is output by <code>pdist</code>. A full dissimilarity matrix must be real and symmetric, and have zeros along the diagonal and positive elements everywhere else. A dissimilarity matrix in upper triangle form must have real, positive entries. You can also specify <math>D</math> as a full similarity matrix, with ones along the diagonal and all other elements less than one. <math>\text{cmdscale}</math> transforms a similarity matrix to a dissimilarity matrix in such a way that distances between the points returned in <math>Y</math> equal or approximate <math>\sqrt{1-D}</math>. To use a different transformation, you must transform the similarities prior to calling <math>\text{cmdscale}</math>.</p>

## Examples

Generate some points in 4-D space, but close to 3-D space, then reduce them to distances only.

```
X = [normrnd(0,1,10,3) normrnd(0,.1,10,1)];  
D = pdist(X,'euclidean');
```

Find a configuration with those inter-point distances.

```
[Y,e] = cmdscale(D);  
  
% Four, but fourth one small  
dim = sum(e > eps^(3/4))  
  
% Poor reconstruction  
maxerr2 = max(abs(pdist(X)-pdist(Y(:,1:2))))  
  
% Good reconstruction  
maxerr3 = max(abs(pdist(X)-pdist(Y(:,1:3))))  
  
% Exact reconstruction  
maxerr4 = max(abs(pdist(X)-pdist(Y)))  
  
% D is now non-Euclidean  
D = pdist(X,'cityblock');  
[Y,e] = cmdscale(D);  
  
% One is large negative  
min(e)  
  
% Poor reconstruction  
maxerr = max(abs(pdist(X)-pdist(Y)))
```

## References

[1] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

mdscale | pdist | procrustes



**Purpose** Confidence intervals of coefficient estimates of generalized linear model

**Syntax**  
`ci = coefCI mdl`  
`ci = coefCI mdl, alpha`

**Description**  
`ci = coefCI mdl` returns confidence intervals for the coefficients in `mdl`.  
`ci = coefCI mdl, alpha` returns confidence intervals with confidence level  $1 - \alpha$ .

**Input Arguments**  
**mdl**  
Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

**alpha**  
Scalar from 0 to 1, the probability that the confidence interval does not contain the true value.

**Default:** 0.05

**Output Arguments**  
**ci**  
k-by-2 matrix of confidence intervals. The *j*th row of `ci` is the confidence interval of coefficient *j* of `mdl`. The name of coefficient *j* of `mdl` is in `mdl.CoefficientNames`.

**Definitions**  
**Confidence Interval**  
Assume that model assumptions hold (data comes from a generalized linear model represented by the formula `mdl`. Formula and the specified link function, and with observations that are independent conditional on the predictor values). Then row *j* of the confidence interval matrix `ci` gives a *confidence interval*  $[ci(j, 1), ci(j, 2)]$  computed such that, with repeated experimentation, a proportion  $1 - \alpha$  of the intervals will contain the true value of the coefficient.

# GeneralizedLinearModel.coefCI

---

## Examples

### Confidence Interval for Coefficients of a Generalized Linear Model

Find confidence intervals for the coefficients of a fitted generalized nonlinear model.

Generate artificial data for the model using Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,...
    'y ~ x1 + x2','distr','poisson')
```

```
mdl =
```

Generalized Linear regression model:

```
log(y) ~ 1 + x1 + x2
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0405	0.022122	47.034	0
x1	0.9968	0.003362	296.49	0
x2	1.987	0.0063433	313.24	0

100 observations, 97 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 2.95e+05, p-value = 0

Find 95% (default) confidence intervals on the coefficients of the model.

```
ci = coefCI mdl
```

```
ci =
```

```
    0.9966    1.0844  
    0.9901    1.0035  
    1.9744    1.9996
```

Find 99% confidence intervals on the coefficients.

```
alpha = .01;  
ci = coefCI(mdl,alpha)
```

```
ci =
```

```
    0.9824    1.0986  
    0.9880    1.0056  
    1.9703    2.0036
```

**See Also** [GeneralizedLinearModel](#) |

## Related Examples

- “Generalized Linear Model Workflow” on page 10-42

# LinearModel.coefCI

---

<b>Purpose</b>	Confidence intervals of coefficient estimates of linear model
<b>Syntax</b>	<code>ci = coefCI mdl</code> <code>ci = coefCI mdl, alpha</code>
<b>Description</b>	<code>ci = coefCI mdl</code> returns confidence intervals for the coefficients in <code>mdl</code> . <code>ci = coefCI mdl, alpha</code> returns confidence intervals with confidence level $1 - \alpha$ .
<b>Input Arguments</b>	<b>mdl</b> Linear model, as constructed by <code>fitlm</code> or <code>stepwiselm</code> . <b>alpha</b> Scalar from 0 to 1, the probability that the confidence interval does not contain the true value. <b>Default:</b> 0.05
<b>Output Arguments</b>	<b>ci</b> k-by-2 matrix of confidence intervals. The <i>j</i> th row of <code>ci</code> is the confidence interval of coefficient <i>j</i> of <code>mdl</code> . The name of coefficient <i>j</i> of <code>mdl</code> is in <code>mdl.CoefficientNames</code> .
<b>Definitions</b>	<b>Confidence Interval</b> Assume that model assumptions hold (data comes from a generalized linear model represented by the formula <code>mdl.Formula</code> , and with observations that are independent conditional on the predictor values). Then row <i>j</i> of the confidence interval matrix <code>ci</code> gives a <i>confidence interval</i> [ <code>ci(j,1)</code> , <code>ci(j,2)</code> ] computed such that, with repeated experimentation, a proportion $1 - \alpha$ of the intervals will contain the true value of the coefficient.

## Examples

### Default Confidence Intervals

Create a linear model for auto mileage based on the carbig data. Then obtain confidence intervals for the resulting model coefficients.

Load the data and create a model.

```
load carbig
Origin = nominal(Origin);
ds = dataset(Horsepower,Weight,MPG,Origin);
modelspec = 'MPG ~ 1 + Horsepower + Weight + Origin';
mdl = fitlm(ds,modelspec);
```

View the names of the coefficients.

```
mdl.CoeffNames
```

```
ans =
Columns 1 through 4
'(Intercept)' 'Horsepower' 'Weight' 'Origin_France'

Columns 5 through 8
'Origin_Germany' 'Origin_Italy' 'Origin_Japan' 'Origin_Sweden'

Column 9
'Origin_USA'
```

Find confidence intervals for the coefficients of the model.

```
ci = coefCI(mdl)
```

```
ci =
    43.361    59.939
   -0.074778   -0.031499
  -0.0058669  -0.0037122
   -17.362   -0.34772
   -15.75    0.74338
   -17.209    0.0613
   -14.511    1.8738
```

```
-18.582    -1.5036
-17.311    -0.96419
```

## Custom Confidence Intervals

Create a linear model for auto mileage based on the `carbig` data. Then obtain confidence intervals for the resulting model coefficients at the 99% level.

Load the data and create a model.

```
load carbig
Origin = nominal(Origin);
ds = dataset(Horsepower,Weight,MPG,Origin);
modelspec = 'MPG ~ 1 + Horsepower + Weight + Origin';
mdl fitlm(ds,modelspec);
```

Find 99% confidence intervals for the coefficients.

```
ci = coefCI(mdl,.01)

ci =
    40.737    62.564
   -0.081629   -0.024647
   -0.006208   -0.0033711
   -20.056     2.3459
   -18.361     3.3546
   -19.943     2.7955
   -17.104     4.4676
   -21.286     1.2002
   -19.899     1.6238
```

The confidence intervals are wider than the default 5% confidence intervals of “Default Confidence Intervals” on page 21-373.

## Alternatives

You can create the intervals from the model coefficients in `mdl.Coefficients.Estimate` and an appropriate multiplier of the standard errors `sqrt(diag(mdl.CoefficientCovariance))`. The multiplier is `tinvs(1-alpha/2,dof)`, where `level` is the confidence

level, and dof is the degrees of freedom (number of data points minus the number of coefficients).

**See Also**

LinearModel

**How To**

- “Linear Regression” on page 9-12

# LinearMixedModel.coefCI

---

**Purpose** Confidence intervals for coefficients of linear mixed-effects model

**Syntax**

```
feCI = coefCI(lme)
feCI = coefCI(lme,Name,Value)
[feCI,reCI] = coefCI( ___ )
```

**Description** `feCI = coefCI(lme)` returns the 95% confidence intervals for the fixed-effects coefficients in the linear mixed-effects model `lme`.

`feCI = coefCI(lme,Name,Value)` returns the 95% confidence intervals for the fixed-effects coefficients in the linear mixed-effects model `lme` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the confidence level or method to compute the degrees of freedom.

`[feCI,reCI] = coefCI( ___ )` also returns the 95% confidence intervals for the random-effects coefficients in the linear mixed-effects model `lme`.

## Input Arguments

### **lme - Linear mixed-effects model**

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'Alpha' - Confidence level**

0.05 (default) | scalar value in the range 0 to 1



Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

**Example:** 'Alpha',0.01

## Data Types

single | double

## 'DFMethod' - Method for computing approximate degrees of freedom

'Residual' (default) | 'Satterthwaite' | 'None'

Method for computing approximate degrees of freedom for confidence interval computation, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'Residual'	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'Satterthwaite'	Satterthwaite approximation.
'None'	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

**Example:** 'DFMethod','Satterthwaite'

## Output Arguments

### **feCI - Fixed-effects confidence intervals**

*p*-by-2 matrix

Fixed-effects confidence intervals, returned as a *p*-by-2 matrix. **feCI** contains the confidence limits that correspond to the *p* fixed-effects estimates in the vector **beta** returned by the **fixedEffects** method. The first column of **feCI** has the lower confidence limits and the second column has the upper confidence limits.

### **reCI - Random-effects confidence intervals**

*q*-by-2 matrix

Random-effects confidence intervals, returned as a *q*-by-2 matrix. **reCI** contains the confidence limits corresponding to the *q* random-effects estimates in the vector **B** returned by the **randomEffects** method. The first column of **reCI** has the lower confidence limits and the second column has the upper confidence limits.

## Examples

### **95% Confidence Intervals for Fixed-Effects Coefficients**

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

**weight** contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define **Subject** and **Program** as categorical variables.

```
ds = dataset(InitialWeight, Program, Subject, Week, y);
ds.Subject = nominal(ds.Subject);
```

```
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(ds, 'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Compute the fixed-effects coefficient estimates.

```
fe = fixedEffects(lme)
```

```
fe =
```

```
    0.6610  
    0.0032  
    0.3608  
   -0.0333  
    0.1132  
    0.1732  
    0.0388  
    0.0305  
    0.0331
```

The first estimate, 0.6610, corresponds to the constant term. The second row, 0.0032, and the third row, 0.3608, are estimates for the coefficient of initial weight and week, respectively. Rows four to six correspond to the indicator variables for programs B-D, and the last three rows correspond to the interaction of programs B-D and week.

Compute the 95% confidence intervals for the fixed-effects coefficients.

```
fecI = coefCI(lme)
```

```
fecI =
```

```
    0.1480    1.1741  
    0.0005    0.0059
```

# LinearMixedModel.coefCI

---

```
0.1004    0.6211
-0.2932   0.2267
-0.1471   0.3734
0.0395    0.3069
-0.1503   0.2278
-0.1585   0.2196
-0.1559   0.2221
```

Some confidence intervals include 0. To obtain specific  $p$ -values for each fixed-effects term, use the `fixedEffects` method. To test for entire terms use the `anova` method.

## Confidence Intervals with Specified Options

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and horsepower, and a potentially correlated random effect for intercept and acceleration grouped by model year. First, store the data in a dataset array.

```
ds = dataset(Acceleration,Horsepower,Model_Year,MPG);
```

Fit the model.

```
lme = fitlme(ds, 'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Y
```

Compute the fixed-effects coefficient estimates.

```
fe = fixedEffects(lme)
```

```
fe =
```

```
50.1325
-0.5833
-0.1695
```

Compute the 99% confidence intervals for fixed-effects coefficients using the residuals method to determine the degrees of freedom. This is the default method.

```
feCI = coefCI(lme, 'Alpha', 0.01)
```

```
feCI =
```

```
    44.2690    55.9961  
   -0.9300   -0.2365  
   -0.1883   -0.1507
```

Compute the 99% confidence intervals for fixed-effects coefficients using the Satterthwaite approximation to compute the degrees of freedom.

```
feCI = coefCI(lme, 'Alpha', 0.01, 'DFMethod', 'Satterthwaite')
```

```
feCI =
```

```
    44.0949    56.1701  
   -0.9640   -0.2025  
   -0.1884   -0.1507
```

The Satterthwaite approximation produces similar confidence intervals to the residual method.

## Compute Confidence Intervals for Random Effects

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during

# LinearMixedModel.coefCI

---

three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);  
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if there is significant difference in the performance according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Compute the estimate of the BLUPs for random effects.

```
randomEffects(lme)
```

```
ans =
```

```
    0.5775  
    1.1757  
   -2.1715  
    2.3655  
   -1.9472
```

Compute the 95% confidence intervals for random effects.

```
[~,reCI] = coefCI(lme)
```

```
reCI =
```

```
   -1.3916    2.5467  
   -0.7934    3.1449  
   -4.1407   -0.2024  
    0.3964    4.3347
```

```
-3.9164    0.0219
```

Compute the 99% confidence intervals for random effects using the residuals method to determine the degrees of freedom. This is the default method.

```
[~,reCI] = coefCI(lme, 'Alpha', 0.01)
```

```
reCI =
```

```
-2.1831    3.3382  
-1.5849    3.9364  
-4.9322    0.5891  
-0.3951    5.1261  
-4.7079    0.8134
```

Compute the 99% confidence intervals for random effects using the Satterthwaite approximation to determine the degrees of freedom.

```
[~,reCI] = coefCI(lme, 'Alpha', 0.01, 'DFMethod', 'Satterthwaite')
```

```
reCI =
```

```
-2.6840    3.8390  
-2.0858    4.4372  
-5.4330    1.0900  
-0.8960    5.6270  
-5.2087    1.3142
```

The Satterthwaite approximation might produce smaller DF values than the residual method. That is why these confidence intervals are larger than the previous ones computed using the residual method.

## See Also

[LinearMixedModel](#) | [coefTest](#) | [fixedEffects](#) | [randomEffects](#) |

# NonLinearModel.coefCI

---

<b>Purpose</b>	Confidence intervals of coefficient estimates of nonlinear regression model
<b>Syntax</b>	<code>ci = coefCI mdl</code> <code>ci = coefCI mdl, alpha</code>
<b>Description</b>	<code>ci = coefCI mdl</code> returns confidence intervals for the coefficients in <code>mdl</code> . <code>ci = coefCI mdl, alpha</code> returns confidence intervals with confidence level $1 - \alpha$ .
<b>Input Arguments</b>	<b>mdl</b> Nonlinear regression model, constructed by <code>fitnlm</code> . <b>alpha</b> Scalar from 0 to 1, the probability that the confidence interval does not contain the true value. <b>Default:</b> 0.05
<b>Output Arguments</b>	<b>ci</b> k-by-2 matrix of confidence intervals. The <i>j</i> th row of <code>ci</code> is the confidence interval of coefficient <i>j</i> of <code>mdl</code> . The name of coefficient <i>j</i> of <code>mdl</code> is in <code>mdl.CoeffNames</code> .
<b>Definitions</b>	<b>Confidence Interval</b> Assume that model assumptions hold (the data comes from a model represented by the formula <code>mdl.Formula</code> , with independent normally distributed errors). Then row <i>j</i> of the confidence interval matrix <code>ci</code> gives a <i>confidence interval</i> [ <code>ci(j,1)</code> , <code>ci(j,2)</code> ] that contains coefficient <i>j</i> with probability $1 - \alpha$ .



## Examples

### Default Confidence Intervals

Create a nonlinear model for auto mileage based on the `carbig` data. Then obtain confidence intervals for the resulting model coefficients.

Load the data and create a nonlinear model.

```
load carbig
ds = dataset(Horsepower,Weight,MPG);
modelfun = @(b,x)b(1) + b(2)*x(:,1) + ...
    b(3)*x(:,2) + b(4)*x(:,1).*x(:,2);
beta0 = [1 1 1 1];
mdl = fitnlm(ds,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
MPG ~ b1 + b2*Horsepower + b3*Weight + b4*Horsepower*Weight
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	63.558	2.3429	27.127	1.2343e-91
b2	-0.25084	0.027279	-9.1952	2.3226e-18
b3	-0.010772	0.00077381	-13.921	5.1372e-36
b4	5.3554e-05	6.6491e-06	8.0542	9.9336e-15

```
Number of observations: 392, Error degrees of freedom: 388
```

```
Root Mean Squared Error: 3.93
```

```
R-Squared: 0.748, Adjusted R-Squared 0.746
```

```
F-statistic vs. constant model: 385, p-value = 7.26e-116
```

All the coefficients have extremely small  $p$ -values. This means a confidence interval around the coefficients will not contain the point 0, unless the confidence level is very high.

Find 95% confidence intervals for the coefficients of the model.

```
ci = coefCI(mdl)
```

```
ci =  
  
    58.9515    68.1644  
   -0.3045   -0.1972  
   -0.0123   -0.0093  
    0.0000    0.0001
```

The confidence interval for `b4` seems to contain 0. Examine it in more detail.

```
ci(4,:)

ans =  
    1.0e-04 *  
    0.4048    0.6663
```

As expected, the confidence interval does not contain the point 0.

## Alternatives

You can create the intervals from the model coefficients in `mdl.Coefficients.Estimate` and an appropriate multiplier of the standard errors `sqrt(diag(mdl.CoefficientCovariance))`. The multiplier is `tinvs(1-alpha/2,dof)`, where `level` is the confidence level, and `dof` is the degrees of freedom (number of data points minus the number of coefficients).

**See Also** `NonLinearModel` |

## Related Examples

- “Nonlinear Regression Workflow” on page 11-16

## Concepts

- “Nonlinear Regression” on page 11-2

## Purpose

Linear hypothesis test on generalized linear regression model coefficients

## Syntax

```
p = coefTest mdl
p = coefTest mdl,H
p = coefTest mdl,H,C
[p,F] = coefTest mdl,...
[p,F,r] = coefTest mdl,...
```

## Description

`p = coefTest(mdl)` computes the  $p$ -value for an  $F$  test that all coefficient estimates in `mdl` are zero, except for the intercept term.

`p = coefTest(mdl,H)` performs an  $F$  test that  $H*B = 0$ , where  $B$  represents the coefficient vector.

`p = coefTest(mdl,H,C)` performs an  $F$  test that  $H*B = C$ .

`[p,F] = coefTest(mdl,...)` returns the  $F$  test statistic.

`[p,F,r] = coefTest(mdl,...)` returns the numerator degrees of freedom for the test.

## Input Arguments

### mdl

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

### H

Numeric matrix having one column for each coefficient in the model. When  $H$  is an input, the output  $p$  is the  $p$ -value for an  $F$  test that  $H*B = 0$ , where  $B$  represents the coefficient vector.

### C

Numeric vector with the same number of rows as  $H$ . When  $C$  is an input, the output  $p$  is the  $p$ -value for an  $F$  test that  $H*B = C$ , where  $B$  represents the coefficient vector.

# GeneralizedLinearModel.coefTest

---

## Output Arguments

**p**

$p$ -value of the  $F$  test (see “Definitions” on page 21-388).

**F**

Value of the test statistic for the  $F$  test (see “Definitions” on page 21-388).

**r**

Numerator degrees of freedom for the  $F$  test (see “Definitions” on page 21-388). The  $F$  statistic has  $r$  degrees of freedom in the numerator and `mdl.DFE` degrees of freedom in the denominator.

## Definitions

### Test Statistics

The  $p$ -value,  $F$  statistic, and numerator degrees of freedom are valid under these assumptions:

- The data comes from a model represented by the formula `mdl.Formula`.
- The observations are independent conditional on the predictor values.

Suppose these assumptions hold. Let  $\beta$  represent the (unknown) coefficient vector of the linear regression. Suppose  $H$  is a full-rank matrix of size  $r$ -by- $s$ , where  $s$  is the number of terms in  $\beta$ . Let  $v$  be a vector the same size as  $\beta$ . The following is a test statistic for the hypothesis that  $H\beta = v$ :

$$F = (H\hat{\beta} - v)'(HCH')^{-1}(H\hat{\beta} - v).$$

Here  $\hat{\beta}$  is the estimate of the coefficient vector  $\beta$  in `mdl.Coeffs`, and  $C$  is the estimated covariance of the coefficient estimates in `mdl.CoeffCov`. When the hypothesis is true, the test statistic  $F$  has an “F Distribution” on page B-41 with  $r$  and  $u$  degrees of freedom.

## Examples

### Test Generalized Linear Model Coefficients

Test a generalized linear model to see if its coefficients differ from zero.

Create a generalized linear regression model of Poisson data.

```
X = 2 + randn(100,1);  
mu = exp(1 + X/2);  
y = poissrnd(mu);  
mdl = fitglm(X,y,'y ~ x1','distr','poisson');
```

Test whether the fitted model has coefficients that differ significantly from zero.

```
p = coefTest(mdl)
```

```
p =
```

```
1.2461e-30
```

There is no doubt that the coefficient of  $x_1$  is nonzero.

## Alternatives

The values of commonly used test statistics are available in the `mdl.Coefficients` dataset array.

## See Also

[GeneralizedLinearModel](#) | [linhpytest](#)

## Related Examples

- “Generalized Linear Model Workflow” on page 10-42

## Concepts

- “Generalized Linear Models” on page 10-12

# LinearModel.coefTest

---

<b>Purpose</b>	Linear hypothesis test on linear regression model coefficients
<b>Syntax</b>	<pre>p = coefTest mdl p = coefTest mdl,H p = coefTest mdl,H,C [p,F] = coefTest mdl,... [p,F,r] = coefTest mdl,...</pre>
<b>Description</b>	<p><code>p = coefTest(mdl)</code> computes the <math>p</math>-value for an <math>F</math> test that all coefficient estimates in <code>mdl</code> are zero, except for the intercept term.</p> <p><code>p = coefTest(mdl,H)</code> performs an <math>F</math> test that <math>H*B = 0</math>, where <math>B</math> represents the coefficient vector.</p> <p><code>p = coefTest(mdl,H,C)</code> performs an <math>F</math> test that <math>H*B = C</math>.</p> <p><code>[p,F] = coefTest(mdl,...)</code> returns the <math>F</math> test statistic.</p> <p><code>[p,F,r] = coefTest(mdl,...)</code> returns the numerator degrees of freedom for the test.</p>
<b>Input Arguments</b>	<p><b>mdl</b> Linear model, as constructed by <code>fitlm</code> or <code>stepwiselm</code>.</p> <p><b>H</b> Numeric matrix having one column for each coefficient in the model. When <math>H</math> is an input, the output <math>p</math> is the <math>p</math>-value for an <math>F</math> test that <math>H*B = 0</math>, where <math>B</math> represents the coefficient vector.</p> <p><b>C</b> Numeric vector with the same number of rows as <math>H</math>. When <math>C</math> is an input, the output <math>p</math> is the <math>p</math>-value for an <math>F</math> test that <math>H*B = C</math>, where <math>B</math> represents the coefficient vector.</p>
<b>Output Arguments</b>	<p><b>p</b> <math>p</math>-value of the <math>F</math> test (see “Definitions” on page 21-391).</p> <p><b>F</b></p>

Value of the test statistic for the  $F$  test (see “Definitions” on page 21-391).

$r$

Numerator degrees of freedom for the  $F$  test (see “Definitions” on page 21-391). The  $F$  statistic has  $r$  degrees of freedom in the numerator and `mdl.DFE` degrees of freedom in the denominator.

## Definitions

### Test Statistics

The  $p$ -value,  $F$  statistic, and numerator degrees of freedom are valid under these assumptions:

- The data comes from a model represented by the formula `mdl.Formula`.
- The observations are independent conditional on the predictor values.

Suppose these assumptions hold. Let  $\beta$  represent the (unknown) coefficient vector of the linear regression. Suppose  $H$  is a full-rank matrix of size  $r$ -by- $s$ , where  $s$  is the number of terms in  $\beta$ . Let  $v$  be a vector the same size as  $\beta$ . The following is a test statistic for the hypothesis that  $H\beta = v$ :

$$F = (H\hat{\beta} - v)'(HCH')^{-1}(H\hat{\beta} - v).$$

Here  $\hat{\beta}$  is the estimate of the coefficient vector  $\beta$  in `mdl.Coeffs`, and  $C$  is the estimated covariance of the coefficient estimates in `mdl.CoeffCov`. When the hypothesis is true, the test statistic  $F$  has an “F Distribution” on page B-41 with  $r$  and  $u$  degrees of freedom.

## Examples

### Test Linear Regression Model

Make a linear model of mileage as a function of the weight, weight squared, and model year from the `carsmall` data set. Test the coefficients to see if all should be zero.

# LinearModel.coefTest

---

Load the data and make a dataset array, where the model year is an ordinal variable.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight + Weight^2');
```

Test the model for significant differences from a constant model.

```
p = coefTest(mdl)
```

```
p =
```

```
5.5208e-41
```

There is no doubt that the model contains more than the intercept term.

## Test a Particular Coefficient

Test the  $\text{Weight}^2$  coefficient in a linear model of mileage as a function of the weight, weight squared, and model year.

Load the data and make a dataset array, where the model year is an ordinal variable.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight + Weight^2');
```

Test the significance of the  $\text{Weight}^2$  coefficient. To do so, find the coefficient corresponding to  $\text{Weight}^2$ .

```
mdl.CoefficientNames
```

```
ans =
```

```
'(Intercept)' 'Weight' 'Year_76' 'Year_82' 'Weight^2'
```



Weight<sup>2</sup> is the fifth (final) coefficient.

Test the significance of the Weight<sup>2</sup> coefficient.

```
p = coefTest mdl, [0 0 0 0 1])
```

```
p =
```

```
0.0022
```

## Alternatives

The values of commonly used test statistics are available in the `mdl.Coefficients` dataset array.

`anova` provides a test for each model term.

## See Also

`anova` | `LinearModel` | `linhpytest`

## How To

- “Linear Regression” on page 9-12

# LinearMixedModel.coefTest

---

**Purpose** Hypothesis test on fixed and random effects of linear mixed-effects model

**Syntax**

```
pVal = coefTest(lme)
pVal = coefTest(lme,H)
pVal = coefTest(lme,H,C)
pVal = coefTest(lme,H,C,'REContrast',K)
pVal = coefTest( __ ,Name,Value)
[pVal,F,DF1,DF2] = coefTest( __ )
```

**Description** `pVal = coefTest(lme)` returns the  $p$ -value for an  $F$ -test that all fixed-effects coefficients except for the intercept are 0.

`pVal = coefTest(lme,H)` returns the  $p$ -value for an  $F$ -test on fixed-effects coefficients of linear mixed-effects model `lme`, using the contrast matrix `H`. It tests the null hypothesis that  $H_0: H\beta = 0$ , where  $\beta$  is the fixed-effects vector.

`pVal = coefTest(lme,H,C)` returns the  $p$ -value for an  $F$ -test on fixed-effects coefficients of the linear mixed-effects model `lme`, using the contrast matrix `H`. It tests the null hypothesis that  $H_0: H\beta = C$ , where  $\beta$  is the fixed-effects vector.

`pVal = coefTest(lme,H,C,'REContrast',K)` returns the  $p$ -value for an  $F$ -test on the fixed- and random-effects coefficients of the linear mixed-effects model `lme`, using the contrast matrices `H` and `K`. It tests the null hypothesis that  $H_0: H\beta + KB = C$ , where  $\beta$  is the fixed-effects vector and  $B$  is the random-effects vector.

`pVal = coefTest( __ ,Name,Value)` returns the  $p$ -value for an  $F$ -test on the fixed- and/or random-effects coefficients of the linear mixed-effects model `lme`, with additional options specified by one or more `Name,Value` pair arguments. It can accept any of the input arguments in the previous syntaxes.

For example, you can specify the method to compute the degrees of freedom.

[pVal,F,DF1,DF2] = coefTest( \_\_\_ ) also returns the  $F$ -statistic  $F$ , and the numerator and denominator degrees of freedom for  $F$ , respectively  $DF1$  and  $DF2$ .

## Input Arguments

### lme - Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

For properties and methods of this object, see LinearMixedModel.

### H - Fixed-effects contrasts

$m$ -by- $p$  matrix

Fixed-effects contrasts, specified as an  $m$ -by- $p$  matrix, where  $p$  is the number of fixed-effects coefficients in `lme`. Each row of `H` represents one contrast. The columns of `H` (left to right) correspond to the rows of the  $p$ -by-1 fixed-effects vector `beta` (top to bottom), returned by the `fixedEffects` method.

### Data Types

single | double

### C - Hypothesized value

$m$ -by-1 vector

Hypothesized value for testing the null hypothesis  $H*beta = C$ , specified as an  $m$ -by-1 matrix. Here, `beta` is the vector of fixed-effects estimates returned by the `fixedEffects` method.

### Data Types

single | double

### K - Random-effects contrasts

$m$ -by- $q$  matrix

Random-effects contrasts, specified as an  $m$ -by- $q$  matrix, where  $q$  is the number of random effects parameters in `lme`. The columns of  $K$  (left to right) correspond to the rows of the random-effects best linear unbiased predictor vector  $B$  (top to bottom), returned by the `randomEffects` method.

## Data Types

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## 'DFMethod' - Method for computing approximate denominator degrees of freedom

'Residual' (default) | 'Satterthwaite' | 'None'

Method for computing the approximate denominator degrees of freedom for the  $F$ -test, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'Residual'	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'Satterthwaite'	Satterthwaite approximation.
'None'	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

**Example:** 'DFMethod', 'Satterthwaite'

## Output Arguments

### **pVal - p-value**

scalar value

$p$ -value for the  $F$ -test on the fixed and/or random-effects coefficients of the linear mixed-effects model `lme`, returned as a scalar value.

### **F - F-statistic**

scalar value

$F$ -statistic, returned as a scalar value.

### **DF1 - Numerator degrees of freedom for F**

scalar value

Numerator degrees of freedom for  $F$ , returned as a scalar value.

- If you test the null hypothesis  $H_0: H\beta = 0$ , or  $H_0: H\beta = C$ , then DF1 is equal to the number of linearly independent rows in  $H$ .
- If you test the null hypothesis  $H_0: H\beta + KB = C$ , then DF1 is equal to the number of linearly independent rows in  $[H, K]$ .

### **DF2 - Denominator degrees of freedom for F**

scalar value

Denominator degrees of freedom for  $F$ , returned as a scalar value. The value of DF2 depends on the option you select for `DFMethod`.

## Examples

### **Randomized Block Design**

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

# LinearMixedModel.coefTest

---

The data shows the absolute deviations from the target quality characteristic measured from the products that five operators manufacture during three different shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if there is significant difference in the performance according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
QCDev ~ 1 + Shift + (1 | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
59.012	62.552	-24.506	49.012

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pVal
'(Intercept)'	3.1196	0.88681	3.5178	12	0
'Shift_Morning'	-0.3868	0.48344	-0.80009	12	0
'Shift_Night'	1.9856	0.48344	4.1072	12	0

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	1.8297

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.76439	0.49315	1.1848

Test if all fixed-effects coefficients except for the intercept are 0.

```
pVal = coefTest(lme)
```

```
pVal =
```

```
7.5956e-04
```

The small  $p$ -value indicates that not all fixed-effects coefficients are 0.

Test the significance of the Shift term using a contrast matrix.

```
H = [0 1 0; 0 0 1];
```

```
pVal = coefTest(lme,H)
```

```
pVal =
```

```
7.5956e-04
```

Test the significance of the Shift term using the anova method.

```
anova(lme)
```

```
ans =
```

# LinearMixedModel.coefTest

---

```
ANOVA marginal tests: DFMethod = 'Residual'
```

Term	FStat	DF1	DF2	pValue
'(Intercept)'	12.375	1	12	0.0042407
'Shift'	13.864	2	12	0.00075956

The  $p$ -value for Shift, 0.00075956, is the same as the  $p$ -value of the previous hypothesis test.

Test if there is any difference between the evening and morning shifts.

```
pVal = coefTest(lme,[0 1 -1])
```

```
pVal =
```

```
3.6147e-04
```

This small  $p$ -value indicates that the performance of the operators are not the same in the morning and the evening shifts.

## Hypothesis Test for Fixed Effects

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);
```



```
ds.Subject = nominal(ds.Subject);  
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(ds, 'y ~ InitialWeight + Program*Week + (Week|Subject)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

```
Formula:
```

```
y ~ 1 + InitialWeight + Program*Week + (1 + Week | Subject)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.66105	0.25892	2.5531	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_B'	0.36079	0.13139	2.746	111
'Program_C'	-0.033263	0.13117	-0.25358	111
'Program_D'	0.11317	0.13132	0.86175	111
'Week'	0.1732	0.067454	2.5677	111
'Program_B:Week'	0.038771	0.095394	0.40644	111
'Program_C:Week'	0.030543	0.095394	0.32018	111

# LinearMixedModel.coefTest

---

```
'Program_D:Week'          0.033114      0.095394      0.34713      111
```

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.18407
'Week'	'(Intercept)'	'corr'	0.66841
'Week'	'Week'	'std'	0.15033

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

Test for the significance of the interaction between Program and Week.

```
H = [0 0 0 0 0 0 1 0 0;  
      0 0 0 0 0 0 0 1 0;  
      0 0 0 0 0 0 0 0 1];  
pVal = coefTest(lme,H)
```

```
pVal =  
  
0.9775
```

The high  $p$ -value indicates that the interaction between Program and Week is not statistically significant.

Now, test whether all coefficients involving Program are 0.

```
H = [0 0 1 0 0 0 0 0 0;  
      0 0 0 1 0 0 0 0 0;  
      0 0 0 0 1 0 0 0 0;  
      0 0 0 0 0 0 1 0 0;  
      0 0 0 0 0 0 0 1 0;  
      0 0 0 0 0 0 0 0 1];  
C = [0;0;0;0;0;0];  
pVal = coefTest(lme,H,C)
```

```
pVal =  
  
    0.0274
```

The  $p$ -value of 0.0274 indicates that not all coefficients involving Program are zero.

## Test for Fixed and Random Effects

Load the sample data.

```
load flu
```

The flu dataset array has a Date variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google® searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and region as the predictor variable, combine the nine columns corresponding to the regions into a tall array. The new dataset array, flu2, must have the response variable, FluRate, the nominal variable, Region, that shows which region each estimate is from, and the grouping variable Date.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...  
    'IndVarName','Region');  
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with fixed effects for the region and a random intercept that varies by Date.

```
lme = fitlme(flu2,'FluRate ~ 1 + Region + (1|Date)')
```

Linear mixed-effects model fit by ML

```
Model information:  
    Number of observations          468  
    Fixed effects coefficients      9
```

# LinearMixedModel.coefTest

```
Random effects coefficients      52
Covariance parameters           2
```

Formula:

```
FluRate ~ 1 + Region + (1|Date)
```

Model fit statistics:

```
AIC      BIC      LogLikelihood  Deviance
318.71   364.35   -148.36      296.71
```

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	p
'(Intercept)'	1.2233	0.096678	12.654	459	
'Region_MidAtl'	0.010192	0.052221	0.19518	459	
'Region_ENCentral'	0.051923	0.052221	0.9943	459	
'Region_WNCentral'	0.23687	0.052221	4.5359	459	7
'Region_SAtl'	0.075481	0.052221	1.4454	459	
'Region_ESCentral'	0.33917	0.052221	6.495	459	2
'Region_WSCentral'	0.069	0.052221	1.3213	459	
'Region_Mtn'	0.046673	0.052221	0.89377	459	
'Region_Pac'	-0.16013	0.052221	-3.0665	459	

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'(Intercept)'	'(Intercept)'	'std'	0.6443		0.

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.26627	0.24878	0.285

Test the hypothesis that the random effects-term for week 10/9/2005 is zero.

```
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics
STATS.Level = nominal(STATS.Level);
K = zeros(length(STATS),1);
```

```
K(STATS.Level == '10/9/2005') = 1;  
pVal = coefTest(lme,[0 0 0 0 0 0 0 0],0,'REContrast',K')
```

```
pVal =  
  
    0.1692
```

Refit the model this time with a random intercept and slope.

```
lme = fitlme(flu2,'FluRate ~ 1 + Region + (1 + Region|Date)');
```

Test the hypothesis that the combined coefficient of region WNCentral for week 10/9/2005 is zero.

```
[~,~,STATS] = randomEffects(lme); STATS.Level = nominal(STATS.Level);  
K = zeros(length(STATS),1);  
K(STATS.Level == '10/9/2005' & flu2.Region == 'WNCentral') = 1;  
pVal = coefTest(lme,[0 0 0 1 0 0 0 0],0,'REContrast',K')
```

```
ans =  
  
    1.4536e-12
```

Also return the  $F$ -statistic with the numerator and denominator degrees of freedom.

```
[pVal,F,DF1,DF2] = coefTest(lme,[0 0 0 1 0 0 0 0],0,'REContrast',K')
```

```
pVal =  
  
    1.4536e-12
```

```
F =  
  
    53.0081
```

```
DF1 =
```

# LinearMixedModel.coefTest

---

1

DF2 =

459

Repeat the test using the Satterthwaite approximation for the denominator degrees of freedom.

```
[pVal,F,DF1,DF2] = coefTest(lme,[0 0 0 1 0 0 0 0 0],0,'REContrast','K','DF
```

pVal =

0.0055

F =

53.0081

DF1 =

1

DF2 =

2.9795

## See Also

[LinearMixedModel](#) | [anova](#) | [coefCI](#) |

**Purpose** Linear hypothesis test on nonlinear regression model coefficients

**Syntax**

```
p = coefTest mdl
p = coefTest mdl,H
p = coefTest mdl,H,C
[p,F] = coefTest mdl,...
[p,F,r] = coefTest mdl,...
```

**Description**

`p = coefTest(mdl)` computes the  $p$ -value for an  $F$  test that all coefficient estimates in `mdl` are zero.

`p = coefTest(mdl,H)` performs an  $F$  test that  $H*B = 0$ , where  $B$  represents the coefficient vector.

`p = coefTest(mdl,H,C)` performs an  $F$  test that  $H*B = C$ .

`[p,F] = coefTest(mdl,...)` returns the  $F$  test statistic.

`[p,F,r] = coefTest(mdl,...)` returns the numerator degrees of freedom for the test.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **H**

Numeric matrix having one column for each coefficient in the model. When  $H$  is an input, the output  $p$  is the  $p$ -value for an  $F$  test that  $H*B = 0$ , where  $B$  represents the coefficient vector.

### **C**

Numeric vector with the same number of rows as  $H$ . When  $C$  is an input, the output  $p$  is the  $p$ -value for an  $F$  test that  $H*B = C$ , where  $B$  represents the coefficient vector.

## Output Arguments

### **p**

$p$ -value of the  $F$  test (see “Definitions” on page 21-408).

### **F**

Value of the test statistic for the  $F$  test (see “Definitions” on page 21-408).

$r$

Numerator degrees of freedom for the  $F$  test (see “Definitions” on page 21-408). The  $F$  statistic has  $r$  degrees of freedom in the numerator and  $mdl.DFE$  degrees of freedom in the denominator.

## Definitions **Test Statistics**

The  $p$ -value,  $F$  statistic, and numerator degrees of freedom are valid under these assumptions:

- The data comes from a normal distribution.
- The entries are independent.

Suppose these assumptions hold. Let  $\beta$  represent the unknown coefficient vector of the linear regression. Suppose  $H$  is a full-rank matrix of size  $r$ -by- $s$ , where  $s$  is the number of terms in  $\beta$ . Let  $v$  be a vector the same size as  $\beta$ . The following is a test statistic for the hypothesis that  $H\beta = v$ :

$$F = (H\hat{\beta} - v)'(HCH')^{-1}(H\hat{\beta} - v).$$

Here  $\hat{\beta}$  is the estimate of the coefficient vector  $\beta$  in `mdl.Coeffs`, and  $C$  is the estimated covariance of the coefficient estimates in `mdl.CoeffCov`. When the hypothesis is true, the test statistic  $F$  has an “F Distribution” on page B-41 with  $r$  and  $u$  degrees of freedom.

## Examples **Test Nonlinear Regression Model Coefficients**

Make a nonlinear model of mileage as a function of the weight from the `carsmall` data set. Test the coefficients to see if all should be zero.



Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
y = MPG;
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';
beta0 = [1 1 1];
mdl = fitnlm(X,y,modelfun,beta0);
```

Test the model for significant differences from a constant model.

```
p = coefTest(mdl)
```

```
p =
```

```
1.3708e-36
```

There is no doubt that the model contains nonzero terms.

## Alternatives

The values of commonly used test statistics are available in the `mdl.Coefficients` dataset array.

## See Also

`NonLinearModel` |

## Concepts

- “Nonlinear Regression” on page 11-2

# CompactTreeBagger.combine

---

**Purpose**            Combine two ensembles

**Syntax**            `B1 = combine(B1,B2)`

**Description**        `B1 = combine(B1,B2)` appends decision trees from ensemble B2 to those stored in B1 and returns ensemble B1. This method requires that the class and variable names be identical in both ensembles.

**See Also**            `TreeBagger.append`

**Purpose** Enumeration of combinations

**Syntax** `C = combnk(v,k)`

**Description** `C = combnk(v,k)` returns all combinations of the  $n$  elements in  $v$  taken  $k$  at a time.

`C = combnk(v,k)` produces a matrix  $C$  with  $k$  columns and  $n!/k!(n-k)!$  rows, where each row contains  $k$  of the elements in the vector  $v$ .

It is not practical to use this function if  $v$  has more than about 15 elements.

**Examples** Combinations of characters from a string.

```
C = combnk('tendrill',4);
last5 = C(31:35,:);
last5 =
tedr
tenl
teni
tenr
tend
```

Combinations of elements from a numeric vector.

```
c = combnk(1:4,2)
c =
 3  4
 2  4
 2  3
 1  4
 1  3
 1  2
```

**See Also** `perms`

# ClassificationDiscriminant.compact

---

<b>Purpose</b>	Compact discriminant analysis classifier
<b>Syntax</b>	<code>cobj = compact(obj)</code>
<b>Description</b>	<code>cobj = compact(obj)</code> creates a compact version of <code>obj</code> .
<b>Input Arguments</b>	<b>obj</b> Discriminant analysis classifier created using <code>ClassificationDiscriminant.fit</code> .
<b>Output Arguments</b>	<b>cobj</b> Compact classifier. <code>cobj</code> has class <code>CompactClassificationDiscriminant</code> . You can predict classifications using <code>cobj</code> exactly as you can using <code>obj</code> . However, since <code>cobj</code> does not contain training data, you cannot perform some actions, such as cross validation.
<b>Examples</b>	Compare the size of the discriminant analysis classifier for Fisher's iris data to the compact version of the classifier:  <pre>load fisheriris fullobj = ClassificationDiscriminant.fit(meas,species); cobj = compact(fullobj); b = whos('fullobj'); % b.bytes = size of fullobj c = whos('cobj'); % c.bytes = size of cobj [b.bytes c.bytes] % shows cobj uses 60% of the memory</pre> <pre>ans =     18578    11498</pre>
<b>See Also</b>	<code>ClassificationDiscriminant</code>
<b>How To</b>	• “Discriminant Analysis” on page 15-3

<b>Purpose</b>	Compact classification ensemble
<b>Syntax</b>	<code>cens = compact(ens)</code>
<b>Description</b>	<code>cens = compact(ens)</code> creates a compact version of <code>ens</code> . You can predict classifications using <code>cens</code> exactly as you can using <code>ens</code> . However, since <code>cens</code> does not contain training data, you cannot perform some actions, such as cross validation.
<b>Input Arguments</b>	<b>ens</b> A classification ensemble created with <code>fitensemble</code> .
<b>Output Arguments</b>	<b>cens</b> A compact classification ensemble. <code>cens</code> has class <code>CompactClassificationEnsemble</code> .
<b>Examples</b>	Compare the size of a classification ensemble for Fisher's iris data to the compact version of the ensemble:  <pre>load fisheriris ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree'); cens = compact(ens); b = whos('ens'); % b.bytes = size of ens c = whos('cens'); % c.bytes = size of ens [b.bytes c.bytes] % shows cens uses less memory</pre> <pre>ans =     571727    532476</pre>
<b>See Also</b>	<code>ClassificationTree</code>   <code>fitensemble</code>
<b>How To</b>	• “Ensemble Methods” on page 16-59

# ClassificationTree.compact

---

<b>Purpose</b>	Compact tree
<b>Syntax</b>	<code>ctree = compact(tree)</code>
<b>Description</b>	<code>ctree = compact(tree)</code> creates a compact version of <code>tree</code> .
<b>Input Arguments</b>	<b>tree</b> A classification tree created using <code>ClassificationTree.fit</code> .
<b>Output Arguments</b>	<b>ctree</b> A compact decision tree. <code>ctree</code> has class <code>CompactClassificationTree</code> . You can predict classifications using <code>ctree</code> exactly as you can using <code>tree</code> . However, since <code>ctree</code> does not contain training data, you cannot perform some actions, such as cross validation.
<b>Examples</b>	<p>Compare the size of the classification tree for Fisher's iris data to the compact version of the tree:</p> <pre>load fisheriris fulltree = ClassificationTree.fit(meas,species); ctree = compact(fulltree); b = whos('fulltree'); % b.bytes = size of fulltree c = whos('ctree'); % c.bytes = size of ctree [b.bytes c.bytes] % shows ctree uses half the memory</pre> <p>ans = 13913            6818</p>
<b>See Also</b>	<code>CompactClassificationTree</code>   <code>ClassificationTree</code>   <code>predict</code>

<b>Purpose</b>	Create compact regression ensemble
<b>Syntax</b>	<code>cens = compact(ens)</code>
<b>Description</b>	<code>cens = compact(ens)</code> creates a compact version of <code>ens</code> . You can predict regressions using <code>cens</code> exactly as you can using <code>ens</code> . However, since <code>cens</code> does not contain training data, you cannot perform some actions, such as cross validation.
<b>Input Arguments</b>	<b>ens</b> A regression ensemble created with <code>fitensemble</code> .
<b>Output Arguments</b>	<b>cens</b> A compact regression ensemble. <code>cens</code> is of class <code>CompactRegressionEnsemble</code> .
<b>Examples</b>	<p>Compare the size of a regression ensemble for the <code>carsmall</code> data to the compact version of the ensemble:</p> <pre>load carsmall X = [Acceleration Cylinders Displacement Horsepower Weight]; ens = fitensemble(X,MPG,'LSBoost',100,'Tree'); cens = compact(ens); b = whos('ens'); % b.bytes = size of ens c = whos('cens'); % c.bytes = size of cens [b.bytes c.bytes] % shows ctree uses less memory</pre> <p>ans =</p> <pre>    311789    287368</pre>
<b>See Also</b>	<code>RegressionEnsemble</code>   <code>CompactRegressionEnsemble</code>

# RegressionTree.compact

---

<b>Purpose</b>	Compact regression tree
<b>Syntax</b>	<code>ctree = compact(tree)</code>
<b>Description</b>	<code>ctree = compact(tree)</code> creates a compact version of <code>tree</code> .
<b>Input Arguments</b>	<b>tree</b> A regression tree created using <code>RegressionTree.fit</code> .
<b>Output Arguments</b>	<b>ctree</b> A compact regression tree. <code>ctree</code> has class <code>CompactRegressionTree</code> . You can predict regressions using <code>ctree</code> exactly as you can using <code>tree</code> . However, since <code>ctree</code> does not contain training data, you cannot perform some actions, such as cross validation.
<b>Examples</b>	Compare the size of a regression tree for the <code>carsmall</code> data to the compact version of the tree:  <pre>load carsmall X = [Acceleration Cylinders Displacement Horsepower Weight]; fulltree = RegressionTree.fit(X,MPG); ctree = compact(fulltree); b = whos('fulltree'); % b.bytes = size of fulltree c = whos('ctree'); % c.bytes = size of ctree [b.bytes c.bytes] % shows ctree uses 2/3 the memory</pre> <pre>ans =     15715    10258</pre>
<b>See Also</b>	<code>CompactRegressionTree</code>   <code>RegressionTree</code>   <code>predict</code>



**Purpose**

Compact ensemble of decision trees

**Description**

Return an object of class `CompactTreeBagger` holding the structure of the trained ensemble. The class is more compact than the full `TreeBagger` class because it does not contain information for growing more trees for the ensemble. In particular, it does not contain `X` and `Y` used for training.

**See Also**

`CompactTreeBagger`

# CompactClassificationDiscriminant

---

**Purpose** Compact discriminant analysis class

**Description** A `CompactClassificationDiscriminant` object is a compact version of a discriminant analysis classifier. The compact version does not include the data for training the classifier. Therefore, you cannot perform some tasks with a compact classifier, such as cross validation. Use a compact classifier for making predictions (classifications) of new data.

**Construction** `cobj = compact(obj)` constructs a compact classifier from a full classifier.

`cobj = ClassificationDiscriminant.make(Mu, Sigma)` constructs a compact discriminant analysis classifier from the class means `Mu` and covariance matrix `Sigma`. For syntax details, see `ClassificationDiscriminant.make`.

## Input Arguments

### **obj**

Discriminant analysis classifier, created with `ClassificationDiscriminant.fit`.

## Properties

### **BetweenSigma**

p-by-p matrix, the between-class covariance, where p is the number of predictors.

### **CategoricalPredictors**

List of categorical predictors, always empty (`[]`) for discriminant analysis.

### **ClassNames**

List of the elements in the training data `Y` with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. `ClassNames` has the same data type as the data in the argument `Y`.

## Coeffs

k-by-k structure of coefficient matrices, where k is the number of classes. `Coeffs(i, j)` contains coefficients of the linear or quadratic boundaries between classes i and j. Fields in `Coeffs(i, j)`:

- `DiscrimType`
- `Class1` — `ClassNames(i)`
- `Class2` — `ClassNames(j)`
- `Const` — A scalar
- `Linear` — A vector with p components, where p is the number of columns in X
- `Quadratic` — p-by-p matrix, exists for quadratic `DiscrimType`

The equation of the boundary between class i and class j is

$$\text{Const} + \text{Linear} * x + x' * \text{Quadratic} * x = 0,$$

where x is a column vector of length p.

If `ClassificationDiscriminant.fit` had the `FillCoeffs` name-value pair set to 'off' when constructing the classifier, `Coeffs` is empty (`[]`).

## Cost

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class j if its true class is i. `Cost` is K-by-K, where K is the number of classes.

Change a `Cost` matrix using dot addressing:

```
obj.Cost = costMatrix
```

## Delta

# CompactClassificationDiscriminant

---

Value of the Delta threshold for a linear discriminant model, a nonnegative scalar. If a coefficient of `obj` has magnitude smaller than `Delta`, `obj` sets this coefficient to 0, and so you can eliminate the corresponding predictor from the model. Set `Delta` to a higher value to eliminate more predictors.

`Delta` must be 0 for quadratic discriminant models.

Change `Delta` by dot addressing:

```
obj.Delta = newDelta
```

## DeltaPredictor

Row vector of length equal to the number of predictors in `obj`. If `DeltaPredictor(i) < Delta` then coefficient `i` of the model is 0.

If `obj` is a quadratic discriminant model, all elements of `DeltaPredictor` are 0.

## DiscrimType

String specifying the discriminant type. One of:

- 'linear'
- 'quadratic'
- 'diagLinear'
- 'diagQuadratic'
- 'pseudoLinear'
- 'pseudoQuadratic'

Change `DiscrimType` using dot addressing:

```
obj.DiscrimType = newDiscrimType
```

You can change between linear types, or between quadratic types, but cannot change between linear and quadratic types.

## Gamma

# CompactClassificationDiscriminant

---

Value of the Gamma regularization parameter, a scalar from 0 to 1. Change Gamma using dot addressing:

```
obj.Gamma = newGamma
```

- If you set 1 for linear discriminant, the discriminant sets its type to 'diagLinear'.
- If you set a value between MinGamma and 1 for linear discriminant, the discriminant sets its type to 'linear'.
- You cannot set values below the value of the MinGamma property.
- For quadratic discriminant, you can set either 0 (for DiscrimType 'quadratic') or 1 (for DiscrimType 'diagQuadratic').

## LogDetSigma

Logarithm of the determinant of the within-class covariance matrix. The type of LogDetSigma depends on the discriminant type:

- Scalar for linear discriminant analysis
- Vector of length K for quadratic discriminant analysis, where K is the number of classes

## MinGamma

Nonnegative scalar, the minimal value of the Gamma parameter so that the correlation matrix is invertible. If the correlation matrix is not singular, MinGamma is 0.

## Mu

Matrix of class means of size K-by-p, where K is the number of classes, and p is the number of predictors. Each row of Mu represents the mean of the multivariate normal distribution of the corresponding class. The class indices are in the ClassNames attribute.

# CompactClassificationDiscriminant

---

## PredictorNames

Cell array of names for the predictor variables, in the order in which they appear in the training data X.

## Prior

Prior probabilities for each class. `Prior` is a numeric vector whose entries relate to the corresponding `ClassNames` property.

Add or change a `Prior` vector using dot addressing:

```
obj.Prior = priorVector
```

## ResponseName

String describing the response variable Y.

## ScoreTransform

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ . For a list of built-in transformation functions and the syntax of custom transformation functions, see `ClassificationDiscriminant.fit`.

Add or change a `ScoreTransform` function by dot addressing:

```
obj.ScoreTransform = 'function'
```

or

```
obj.ScoreTransform = @function
```

## Sigma

Within-class covariance matrix or matrices. The dimensions depend on `DiscrimType`:

- 'linear' (default) — Matrix of size p-by-p, where p is the number of predictors

- 'quadratic' — Array of size  $p$ -by- $p$ -by- $K$ , where  $K$  is the number of classes
- 'diagLinear' — Row vector of length  $p$
- 'diagQuadratic' — Array of size  $1$ -by- $p$ -by- $K$
- 'pseudoLinear' — Matrix of size  $p$ -by- $p$
- 'pseudoQuadratic' — Array of size  $p$ -by- $p$ -by- $K$

## Methods

edge	Classification edge
logP	Log of the unconditional probability density
loss	Classification error
mahal	Mahalanobis distance to class means
margin	Classification margins
nLinearCoeffs	Number of nonzero linear coefficients
predict	Predict classification

## Definitions

### Discriminant Classification

The model for discriminant analysis is:

- Each class ( $Y$ ) generates data ( $X$ ) using a multivariate normal distribution. That is, the model assumes  $X$  has a Gaussian mixture distribution (`gmdistribution`).
  - For linear discriminant analysis, the model has the same covariance matrix for each class, only the means vary.
  - For quadratic discriminant analysis, both means and covariances of each class vary.

# Compact Classification Discriminant

---

predict classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

For details, see “How the predict Method Classifies” on page 15-6.

## Regularization

Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters,  $\gamma$  and  $\delta$ , that control regularization as follows. `cvshrink` helps you select appropriate values of the parameters.

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be



the correlation matrix of the data  $X$ , and let  $\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = \left[ (x - \mu_0)^T D^{-1/2} \right] \left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square brackets. Each component of the vector

$\left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When  $\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$\left| \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

## Copy Semantics

Value. To learn how value classes affect copy operations, see `Copying Objects` in the MATLAB documentation.

## Examples

Construct a compact discriminant analysis classifier for the Fisher iris data, and compare its size to that of the full classifier:

```
load fisheriris
fullobj = ClassificationDiscriminant.fit(meas,species);
```

# CompactClassificationDiscriminant

---

```
cobj = compact(fullobj);
b = whos('fullobj'); % b.bytes = size of fullobj
c = whos('cobj'); % c.bytes = size of cobj
[b.bytes c.bytes] % shows cobj uses 60% of the memory

ans =
      18578      11498
```

---

Construct a compact discriminant analysis classifier from the means and covariances of the Fisher iris data:

```
load fisheriris
mu(1,:) = mean(meas(1:50,:));
mu(2,:) = mean(meas(51:100,:));
mu(3,:) = mean(meas(101:150,:));

mm1 = repmat(mu(1,:),50,1);
mm2 = repmat(mu(2,:),50,1);
mm3 = repmat(mu(3,:),50,1);
cc = meas;
cc(1:50,:) = cc(1:50,:) - mm1;
cc(51:100,:) = cc(51:100,:) - mm2;
cc(101:150,:) = cc(101:150,:) - mm3;
sigstar = cc' * cc / 147;
cpct = ClassificationDiscriminant.make(mu,sigstar,...
    'ClassNames',{'setosa','versicolor','virginica'});
```

## See Also

ClassificationDiscriminant | compact |  
ClassificationDiscriminant.make | predict

## How To

- “Discriminant Analysis” on page 15-3

<b>Purpose</b>	Compact classification ensemble class
<b>Description</b>	Compact version of a classification ensemble (of class <code>ClassificationEnsemble</code> ). The compact version does not include the data for training the classification ensemble. Therefore, you cannot perform some tasks with a compact classification ensemble, such as cross validation. Use a compact classification ensemble for making predictions (classifications) of new data.
<b>Construction</b>	<code>ens = compact(ens)</code> constructs a compact decision ensemble from a full decision ensemble.
	<b>Input Arguments</b>
	<b>ens</b> A classification ensemble created by <code>fitensemble</code> .
<b>Properties</b>	<b>CategoricalPredictors</b> List of categorical predictors. <code>CategoricalPredictors</code> is a numeric vector with indices from 1 to $p$ , where $p$ is the number of columns of $X$ .
	<b>ClassNames</b> List of the elements in $Y$ with duplicates removed. <code>ClassNames</code> can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. <code>ClassNames</code> has the same data type as the data in the argument $Y$ .
	<b>CombineWeights</b> String describing how <code>ens</code> combines weak learner weights, either 'WeightedSum' or 'WeightedAverage'.
	<b>Cost</b> Square matrix where <code>Cost(i, j)</code> is the cost of classifying a point into class $j$ if its true class is $i$ .
	<b>NTrained</b>

# CompactClassificationEnsemble

---

Number of trained weak learners in `cens`, a scalar.

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

## **Prior**

Prior probabilities for each class. `Prior` is a numeric vector whose entries relate to the corresponding `ClassNames` property.

## **ResponseName**

String with the name of the response variable `Y`.

## **ScoreTransform**

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means `@(x)x`. For a list of built-in transformation functions and the syntax of custom transformation functions, see `ClassificationTree.fit`.

Add or change a `ScoreTransform` function by dot addressing:

```
cens.ScoreTransform = 'function'
```

or

```
cens.ScoreTransform = @function
```

## **Trained**

Trained learners, a cell array of compact classification models.

## **TrainedWeights**

Numeric vector of trained weights for the weak learners in `ens`. `TrainedWeights` has `T` elements, where `T` is the number of weak learners in `learners`.

## **UsePredForLearner**

Logical matrix of size P-by-NTrained, where P is the number of predictors (columns) in the training data X. UsePredForLearner(i, j) is true when learner j uses predictor i, and is false otherwise. For each learner, the predictors have the same order as the columns in the training data X.

If the ensemble is not of type Subspace, all entries in UsePredForLearner are true.

## Methods

edge	Classification edge
loss	Classification error
margin	Classification margins
predict	Predict classification
predictorImportance	Estimates of predictor importance
removeLearners	Remove members of compact classification ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Create a compact classification ensemble for the ionosphere data:

```
load ionosphere
ens = fitensemble(X,Y,'AdaBoostM1',100,'Tree');
cens = compact(ens)

cens =
classreg.learning.classif.CompactClassificationEnsemble:
    PredictorNames: {1x34 cell}
    CategoricalPredictors: []
    ResponseName: 'Y'
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
```

# CompactClassificationEnsemble

---

NTrained: 100

## See Also

`fitensemble` | `ClassificationEnsemble` | `predict` | `compact`

<b>Purpose</b>	Compact classification tree
<b>Description</b>	Compact version of a classification tree (of class <code>ClassificationTree</code> ). The compact version does not include the data for training the classification tree. Therefore, you cannot perform some tasks with a compact classification tree, such as cross validation. Use a compact classification tree for making predictions (classifications) of new data.
<b>Construction</b>	<code>ctree = compact(tree)</code> constructs a compact decision tree from a full decision tree.
	<b>Input Arguments</b>
	<b>tree</b> A decision tree constructed by <code>ClassificationTree.fit</code> .
<b>Properties</b>	<b>CategoricalPredictors</b> List of categorical predictors. <code>CategoricalPredictors</code> is a numeric vector with indices from 1 to $p$ , where $p$ is the number of columns of $X$ .
	<b>CatSplit</b> An $n$ -by-2 cell array, where $n$ is the number of categorical splits in <code>tree</code> . Each row in <code>CatSplit</code> gives left and right values for a categorical split. For each branch node with categorical split $j$ based on a categorical predictor variable $z$ , the left child is chosen if $z$ is in <code>CatSplit(j,1)</code> and the right child is chosen if $z$ is in <code>CatSplit(j,2)</code> . The splits are in the same order as nodes of the tree. Find the nodes for these splits by selecting 'categorical' cuts from top to bottom in the <code>CutType</code> property.
	<b>Children</b> An $n$ -by-2 array containing the numbers of the child nodes for each node in <code>tree</code> , where $n$ is the number of nodes. Leaf nodes have child node 0.
	<b>ClassCount</b>

# CompactClassificationTree

---

An  $n$ -by- $k$  array of class counts for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class counts `ClassCount(i, :)` are counts of observations (from the data used in fitting the tree) from each class satisfying the conditions for node  $i$ .

## **ClassNames**

List of the elements in `Y` with duplicates removed. `ClassNames` can be a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. `ClassNames` has the same data type as the data in the argument `Y`.

## **ClassProb**

An  $n$ -by- $k$  array of class probabilities for the nodes in `tree`, where  $n$  is the number of nodes and  $k$  is the number of classes. For any node number  $i$ , the class probabilities `ClassProb(i, :)` are the estimated probabilities for each class for a point satisfying the conditions for node  $i$ .

## **Cost**

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class  $j$  if its true class is  $i$ .

## **CutCategories**

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `CutCategories{i, 1}`, and the right child is chosen if  $x$  is among those listed in `CutCategories{i, 2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **CutPoint**



An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if  $x < \text{CutPoint}(i)$  and the right child is chosen if  $x \geq \text{CutPoint}(i)$ . `CutPoint` is NaN for branch nodes based on categorical predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## CutType

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .
- 'categorical' — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## CutVar

An  $n$ -element cell array of the names of the variables used for branching in each node in `tree`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `CutVar` contains an empty string.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## IsBranch

An  $n$ -element logical vector that is true for each branch node and false for each leaf node of `tree`.

# CompactClassificationTree

---

## **NodeClass**

An  $n$ -element cell array with the names of the most probable classes in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element of this array is a string equal to one of the class names in `ClassNames`.

## **NodeErr**

An  $n$ -element vector of the errors of the nodes in `tree`, where  $n$  is the number of nodes. `NodeErr(i)` is the misclassification probability for node  $i$ .

## **NodeProb**

An  $n$ -element vector of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.

## **NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the measure of impurity (Gini index or deviance) for this node weighted by the node probability. If the tree is grown by twining, the risk for each node is zero.

## **NodeSize**

An  $n$ -element vector of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

## **NumNodes**

The number of nodes in `tree`.

## **Parent**

An  $n$ -element vector containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

## PredictorNames

A cell array of names for the predictor variables, in the order in which they appear in `X`.

## Prior

Prior probabilities for each class. `Prior` is a numeric vector whose entries relate to the corresponding `ClassNames` property.

## PruneAlpha

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

## PruneList

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

## ResponseName

String describing the response variable `Y`.

## ScoreTransform

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means `@(x)x`. For a list of built-in transformation functions and the syntax of custom transformation functions, see `ClassificationTree.fit`.

Add or change a `ScoreTransform` function by dot addressing:

```
ctree.ScoreTransform = 'function'  
or
```

# CompactClassificationTree

---

```
ctree.ScoreTransform = @function
```

## SurrCutCategories

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutCategories{k}` is a cell array. The length of `SurrCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutCategories{k}` is either an empty string for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutCategories` contains an empty cell.

## SurrCutFlip

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutFlip{k}` is a numeric vector. The length of `SurrCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this

node does not appear. For nonbranch (leaf) nodes, `SurrCutFlip` contains an empty array.

## **SurrCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutPoint{k}` is a numeric vector. The length of `SurrCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutPoint{k}` is either NaN for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and `SurrCutFlip` for this surrogate split is  $+1$ , or if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables returned by `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutPoint` contains an empty cell.

## **SurrCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

# CompactClassificationTree

---

## SurrCutVar

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrCutVar` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutVar` contains an empty cell.

## SurrVarAssoc

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrVarAssoc{k}` is a numeric vector. The length of `SurrVarAssoc{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrVarAssoc{k}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrVarAssoc` contains an empty cell.

## Methods

<code>edge</code>	Classification edge
<code>loss</code>	Classification error
<code>margin</code>	Classification margins
<code>meanSurrVarAssoc</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>predict</code>	Predict classification
<code>predictorImportance</code>	Estimates of predictor importance
<code>view</code>	View tree

## Definitions

### Impurity and Node Error

ClassificationTree splits nodes based on either *impurity* or *node error*. Impurity means one of several things, depending on your choice of the SplitCriterion name-value pair:

- Gini's Diversity Index (gdi) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a *pure* node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('deviance') — With  $p(i)$  defined as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('twoing') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R) \left( \sum_i |L(i) - R(i)| \right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

# CompactClassificationTree

---

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Construct a compact classification tree for the Fisher iris data, and compare the size of the resulting tree to that of the original tree:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
ctree = compact(tree);
t = whos('tree'); % t.bytes = size of tree in bytes
c = whos('ctree'); % c.bytes = size of ctree in bytes
[c.bytes t.bytes]
```

```
ans =
      6818      13913
```

## See Also

[ClassificationTree.fit](#) | [ClassificationTree](#) | [compact](#)



<b>Purpose</b>	Compact regression ensemble class
<b>Description</b>	Compact version of a regression ensemble (of class <code>RegressionEnsemble</code> ). The compact version does not include the data for training the regression ensemble. Therefore, you cannot perform some tasks with a compact regression ensemble, such as cross validation. Use a compact regression ensemble for making predictions (regressions) of new data.
<b>Construction</b>	<code>ens = compact(ens)</code> constructs a compact decision ensemble from a full decision ensemble.
	<b>Input Arguments</b>
	<b>ens</b> A regression ensemble created by <code>fitensemble</code> .
<b>Properties</b>	<b>CategoricalPredictors</b> List of categorical predictors. <code>CategoricalPredictors</code> is a numeric vector with indices from 1 to <code>p</code> , where <code>p</code> is the number of columns of <code>X</code> .
	<b>CombineWeights</b> A string describing how the ensemble combines learner predictions.
	<b>NTrained</b> Number of trained learners in the ensemble, a positive scalar.
	<b>PredictorNames</b> A cell array of names for the predictor variables, in the order in which they appear in <code>X</code> .
	<b>ResponseName</b> A string with the name of the response variable <code>Y</code> .
	<b>ResponseTransform</b>

# CompactRegressionEnsemble

---

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ .

Add or change a ResponseTransform function by dot addressing:

```
cens.ResponseTransform = @function
```

## Trained

The trained learners, a cell array of compact regression models.

## TrainedWeights

A numeric vector of weights the ensemble assigns to its learners. The ensemble computes predicted response by aggregating weighted predictions from its learners.

## Methods

loss	Regression error
predict	Predict response of ensemble
predictorImportance	Estimates of predictor importance
removeLearners	Remove members of compact regression ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Construct a regression ensemble for the carsmall data. Make a compact version of the ensemble, and compare its size to that of the full ensemble:

```
load carsmall
learner = RegressionTree.template('MinParent',20);
ens = fitensemble([Weight, Cylinders],MPG,...
    'LSBoost',100,learner,'PredictorNames',{'W','C'},...
    'categoricalpredictors',2);
```

```
cens = compact(ens);  
ee = whos('ens'); % ee.bytes = size of ensemble in bytes  
cee = whos('cens');  
[ee.bytes cee.bytes]
```

```
ans =  
    606903    587096
```

## See Also

[fitensemble](#) | [RegressionEnsemble](#) | [predict](#) | [compact](#)

# CompactRegressionTree

---

**Purpose** Compact regression tree

**Description** Compact version of a regression tree (of class `RegressionTree`). The compact version does not include the data for training the regression tree. Therefore, you cannot perform some tasks with a compact regression tree, such as cross validation. Use a compact regression tree for making predictions (regressions) of new data.

**Construction** `ctree = compact(tree)` constructs a compact decision tree from a full decision tree.

## Input Arguments

### **tree**

A decision tree constructed by `RegressionTree.fit`.

## Properties

### **CategoricalPredictors**

List of categorical predictors. `CategoricalPredictors` is a numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .

### **CatSplit**

An  $n$ -by-2 cell array, where  $n$  is the number of categorical splits in `tree`. Each row in `CatSplit` gives left and right values for a categorical split. For each branch node with categorical split  $j$  based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in `CatSplit(j,1)` and the right child is chosen if  $z$  is in `CatSplit(j,2)`. The splits are in the same order as nodes of the tree. Nodes for these splits can be found by running `cuttype` and selecting 'categorical' cuts from top to bottom.

### **Children**

An  $n$ -by-2 array containing the numbers of the child nodes for each node in `tree`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

### **CutCategories**

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `CutCategories{i,1}`, and the right child is chosen if  $x$  is among those listed in `CutCategories{i,2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **CutPoint**

An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if `CutPoint < v(i)` and the right child is chosen if `x >= CutPoint(i)`. `CutPoint` is NaN for branch nodes based on categorical predictors and for leaf nodes.

## **CutType**

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .
- 'categorical' — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **CutVar**

An  $n$ -element cell array of the names of the variables used for branching in each node in `tree`, where  $n$  is the number of nodes.

# CompactRegressionTree

---

These variables are sometimes known as *cut variables*. For leaf nodes, `CutVar` contains an empty string.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

## **IsBranch**

An  $n$ -element logical vector `ib` that is `true` for each branch node and `false` for each leaf node of `tree`.

## **NodeErr**

An  $n$ -element vector `e` of the errors of the nodes in `tree`, where  $n$  is the number of nodes. `e(i)` is the misclassification probability for node  $i$ .

## **NodeMean**

An  $n$ -element numeric array with mean values in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element in `NodeMean` is the average of the true  $Y$  values over all observations in the node.

## **NodeProb**

An  $n$ -element vector `p` of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.

## **NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the node error weighted by the node probability.

## **NodeSize**

An  $n$ -element vector `sizes` of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as

the number of observations from the data used to create the tree that satisfy the conditions for the node.

## **NumNodes**

The number of nodes  $n$  in `tree`.

## **Parent**

An  $n$ -element vector `p` containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

## **PruneAlpha**

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

## **PruneList**

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

## **ResponseName**

Name of the response variable `Y`, a string.

## **ResponseTransform**

Function handle for transforming the raw response values (mean squared error). The function handle should accept a matrix of response values and return a matrix of the same size. The default string 'none' means `@(x)x`, or no transformation.

Add or change a `ResponseTransform` function by dot addressing:

```
ctree.ResponseTransform = @function
```

## **SurrCutCategories**

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutCategories{k}` is a cell array. The length of `SurrCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutCategories{k}` is either an empty string for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split, and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutCategories` contains an empty cell.

## **SurrCutFlip**

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutFlip{k}` is a numeric vector. The length of `SurrCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this



node does not appear. For nonbranch (leaf) nodes, `SurrCutFlip` contains an empty array.

## **SurrCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutPoint{k}` is a numeric vector. The length of `SurrCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutPoint{k}` is either NaN for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and `SurrCutFlip` for this surrogate split is  $+1$ , or if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables returned by `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutPoint` contains an empty cell.

## **SurrCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

# CompactRegressionTree

---

## SurrCutVar

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrCutVar` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutVar` contains an empty cell.

## SurrVarAssoc

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrVarAssoc{k}` is a numeric vector. The length of `SurrVarAssoc{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrVarAssoc{k}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrVarAssoc` contains an empty cell.

## Methods

<code>loss</code>	Regression error
<code>meanSurrVarAssoc</code>	Mean predictive measure of association for surrogate splits in decision tree
<code>predict</code>	Predict response of regression tree
<code>predictorImportance</code>	Estimates of predictor importance
<code>view</code>	View tree

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Construct a regression tree for the `carsmall` data. Make a compact version of the tree, and compare its size to that of the full tree:

```
load carsmall
tree = RegressionTree.fit([Weight, Cylinders],MPG,...
    'MinParent',20,...
    'PredictorNames',{'W','C'});
ctree = compact(tree);
t = whos('tree'); % t.bytes = size of tree in bytes
c = whos('ctree'); % c.bytes = size of ctree in bytes
[c.bytes t.bytes]
```

```
ans =
      4972      8173
```

## See Also

[RegressionTree.fit](#) | [RegressionTree](#) | [compact](#)

# CompactTreeBagger

---

**Purpose** Compact ensemble of decision trees grown by bootstrap aggregation

**Description** CompactTreeBagger class is a lightweight class that contains the trees grown using TreeBagger. CompactTreeBagger does not preserve any information about how TreeBagger grew the decision trees. It does not contain the input data used for growing trees, nor does it contain training parameters such as minimal leaf size or number of variables sampled for each decision split at random. You can only use CompactTreeBagger for predicting the response of the trained ensemble given new data X, and other related functions.

CompactTreeBagger lets you save the trained ensemble to disk, or use it in any other way, while discarding training data and various parameters of the training configuration irrelevant for predicting response of the fully grown ensemble. This reduces storage and memory requirements, especially for ensembles trained on large data sets.

**Construction** CompactTreeBagger Create CompactTreeBagger object

**Methods**

combine	Combine two ensembles
error	Error (misclassification probability or MSE)
margin	Classification margin
mdsProx	Multidimensional scaling of proximity matrix
meanMargin	Mean classification margin
outlierMeasure	Outlier measure for data
predict	Predict response

proximity	Proximity matrix for data
setDefaultYfit	Set default value for predict

## Properties

### ClassNames

The `ClassNames` property is a cell array containing the class names for the response variable `Y` supplied to `TreeBagger`. This property is empty for regression trees.

### DeltaCritDecisionSplit

The `DeltaCritDecisionSplit` property is a numeric array of size 1-by-`Nvars` of changes in the split criterion summed over splits on each variable, averaged across the entire ensemble of grown trees.

See also `TreeBagger.DeltaCritDecisionSplit`, `ClassificationTree.predictorImportance`, and `RegressionTree.predictorImportance`

### DefaultYfit

The `DefaultYfit` property controls what predicted value `CompactTreeBagger` returns when no prediction is possible, for example when the `predict` method needs to predict for an observation which has only false values in the matrix supplied through `'useifort'` argument.

For classification, you can set this property to either `''` or `'MostPopular'`. If you choose `'MostPopular'` (default), the property value becomes the name of the most probable class in the training data.

For regression, you can set this property to any numeric scalar. The default is the mean of the response for the training data.

See also `predict`, `setDefaultYfit`, `TreeBagger.DefaultYfit`.

### Method

The `Method` property is `'classification'` for classification ensembles and `'regression'` for regression ensembles.

# CompactTreeBagger

---

## **NTrees**

The `NTrees` property is a scalar equal to the number of decision trees in the ensemble.

## **NVarSplit**

The `NVarSplit` property is a numeric array of size 1-by-*Nvars*, where every element gives a number of splits on this predictor summed over all trees.

## **Trees**

The `Trees` property is a cell array of size `NTrees`-by-1 containing the trees in the ensemble.

## **VarAssoc**

The `VarAssoc` property is a matrix of size *Nvars*-by-*Nvars* with predictive measures of variable association, averaged across the entire ensemble of grown trees. If you grew the ensemble setting 'surrogate' to 'on', this matrix for each tree is filled with predictive measures of association averaged over the surrogate splits. If you grew the ensemble setting 'surrogate' to 'off' (default), `VarAssoc` is diagonal.

See also `ClassificationTree.meanSurrVarAssoc`, `RegressionTree.meanSurrVarAssoc`.

## **VarNames**

The `VarNames` property is a cell array containing the names of the predictor variables (features). These names are taken from the optional 'names' parameter that supplied to `TreeBagger`. The default names are 'x1', 'x2', etc.

## **Copy Semantics**

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## **See Also**

`TreeBagger` | `TreeBagger.compact` | `ClassificationTree` | `RegressionTree`

## **How To**

- “Ensemble Methods” on page 16-59
- “Classification Trees and Regression Trees” on page 16-31
- “Grouping Variables” on page 2-51

# CompactTreeBagger

---

**Purpose** Create CompactTreeBagger object

**Description** When you use the TreeBagger constructor to grow trees, it creates a CompactTreeBagger object. You can obtain the compact object from the full TreeBagger object using the TreeBagger/compact method. You do not create an instance of CompactTreeBagger directly.

**See Also** TreeBagger

**How To**

- “Grouping Variables” on page 2-51
- “Ensemble Methods” on page 16-59



**Purpose** Compare linear mixed-effects models

## Syntax

```
results = compare(lme,altlme)
results = compare(__,Name,Value)

[results,siminfo] = compare(lme,altlme,'NSim',nsim)
[results,siminfo] = compare(__,Name,Value)
```

## Description

`results = compare(lme,altlme)` returns the results of a likelihood ratio test that compares the linear mixed-effects models `lme` and `altlme`. Both models must use the same response vector in the fit and `lme` must be nested in `altlme` for a valid theoretical likelihood ratio test. Always input the smaller model first, and the larger model second.

`compare` tests the following null and alternate hypotheses:

$H_0$ : Observed response vector is generated by `lme`.

$H_1$ : Observed response vector is generated by model `altlme`.

It is recommended that you fit `lme` and `altlme` using the maximum likelihood (ML) method prior to model comparison. If you use the restricted maximum likelihood (REML) method, then both models must have the same fixed-effects design matrix.

To test for fixed effects, use `compare` with the simulated likelihood ratio test when `lme` and `altlme` are fit using ML or use the `fixedEffects`, `anova`, or `coefTest` methods.

`results = compare(__,Name,Value)` also returns the results of a likelihood ratio test that compares linear mixed-effects models `lme` and `altlme` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can check if the first input model is nested in the second input model.

# LinearMixedModel.compare

---

`[results,siminfo] = compare(lme,altlme,'NSim',nsim)` returns the results of a simulated likelihood ratio test that compares linear mixed-effects models `lme` and `altlme`.

You can fit `lme` and `altlme` using ML or REML. Also, `lme` does not have to be nested in `altlme`. If you use the restricted maximum likelihood (REML) method to fit the models, then both models must have the same fixed-effects design matrix.

`[results,siminfo] = compare(___,Name,Value)` also returns the results of a simulated likelihood ratio test that compares linear mixed-effects models `lme` and `altlme` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can change the options for performing the simulated likelihood ratio test, or change the confidence level of the confidence interval for the  $p$ -value.

## Input Arguments

### **lme** - Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **altlme** - Alternative linear mixed-effects model

`LinearMixedModel` object

Alternative linear mixed-effects model fit to the same response vector but with different model specifications, specified as a `LinearMixedModel` object. `lme` must be nested in `altlme`, that is, `lme` should be obtained from `altlme` by setting some parameters to fixed values, such as 0. You can create a linear mixed-effects object using `fitlme` or `fitlmematrix`.

### **nsim** - Number of replications for simulations

positive integer number

Number of replications for simulations in the simulated likelihood ratio test, specified as a positive integer number. You must specify `nsim` to do a simulated likelihood ratio test.

**Example:** `'NSim',1000`

## Data Types

double | single

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## 'Alpha' - Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

**Example:** `'Alpha',0.01`

## Data Types

single | double

## 'Options' - Options for performing simulated likelihood ratio test

structure

Options for performing the simulated likelihood ratio test, specified as the comma-separated pair consisting of `'Options'`, and a structure created by `statset('LinearMixedModel')`.

`compare` uses the following fields.

# LinearMixedModel.compare

---

'UseParallel'

- False for serial computation. Default.
- True for parallel computation.

'UseSubstreams'

- False for not using a separate substream of the random number generator for each iteration. Default.
- True for using a separate substream of the random number generator for each iteration. You can only use this option with random stream types that support substreams.

'Streams'

- If 'UseSubstreams' is True, then 'Streams' must be a single random number stream, or a scalar cell array containing a single stream.
- If 'UseSubstreams' is False and
  - 'UseParallel' is False, then 'Streams' must be a single random number stream, or a scalar cell array containing a single stream.
  - 'UseParallel' is True, then 'Streams' must be equal to the number of processors used. If a parallel pool is open, then

the 'Streams' is the same length as the size of the parallel pool. If 'UseParallel' is True, a parallel pool might open up for you. But since 'Streams' must be equal to the number of processors used, it is best to open a pool explicitly using the `parpool` command, before calling `compare` with the 'UseParallel', 'True' option.

For information on parallel statistical computing at the command line, enter

```
help parallelstats
```

## Data Types

struct

### 'CheckNesting' - Indicator to check nesting between two models

false (default) | true

Indicator to check nesting between two models, specified as the comma-separated pair consisting of 'CheckNesting' and one of the following.

false

Default. No checks.

true

compare checks if the smaller model `lme` is nested in the bigger model `alllme`.

# LinearMixedModel.compare

---

`lme` must be nested in the alternate model `altlme` for a valid theoretical likelihood ratio test. `compare` returns an error message if the nesting requirements are not satisfied.

Although valid for both tests, the nesting requirements are weaker for the simulated likelihood ratio test.

**Example:** `'CheckNesting',true`

## Data Types

`single` | `double`

## Output Arguments

### **results** - Results of likelihood ratio test or simulated likelihood ratio test

`dataset array`

Results of the likelihood ratio test or simulated likelihood ratio test, returned as a dataset array with two rows. The first row is for `lme`, and the second row is for `altlme`. The columns of results depend on whether the test is a likelihood ratio or a simulated likelihood ratio test.

- If you use the likelihood ratio test, then `results` contains the following columns.

<code>Model</code>	Name of the model
<code>DF</code>	Degrees of freedom, that is, the number of free parameters in the model
<code>AIC</code>	Akaike information criterion for the model
<code>BIC</code>	Bayesian information criterion for the model
<code>LogLik</code>	Maximized log likelihood for the model

LRStat	Likelihood ratio test statistic for comparing altlme versus lme
deltaDF	DF for altlme minus DF for lme
pValue	<i>p</i> -value for the likelihood ratio test

- If you use the simulated likelihood ratio test, then results contains the following columns.

Model	Name of the model
DF	Degrees of freedom, that is, the number of free parameters in the model
LogLik	Maximized log likelihood for the model
LRStat	Likelihood ratio test statistic for comparing altlme versus lme
pValue	<i>p</i> -value for the likelihood ratio test
Lower	Lower limit of the confidence interval for pValue
Upper	Upper limit of the confidence interval for pValue

## **siminfo - Simulation output**

structure

Simulation output, returned as a structure with the following fields.

# LinearMixedModel.compare

---

<code>nsim</code>	Value set for <code>nsim</code> .
<code>alpha</code>	Value set for 'Alpha'.
<code>pValueSim</code>	Simulation-based $p$ -value.
<code>pValueSimCI</code>	Confidence interval for <code>pValueSim</code> . The first element of the vector is the lower limit and the second element of the vector contains the upper limit.
<code>deltaDF</code>	The number of free parameters in <code>altlme</code> minus the number of free parameters in <code>lme</code> . DF for <code>altlme</code> minus DF for <code>lme</code> .
<code>THO</code>	A vector of simulated likelihood ratio test statistics under the null hypothesis that the model <code>lme</code> generated the observed response vector <code>y</code> .

## Examples

### Test for Random Effects

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and region as the predictor variable, combine the nine columns corresponding to the regions into a tall array. The new dataset array, `flu2`, must have the response variable,



FluRate, the nominal variable, Region, that shows which region each estimate is from, and the grouping variable Date.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...  
            'IndVarName','Region');  
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model, with a varying intercept and varying slope for each region, grouped by Date.

```
altlme = fitlme(flu2,'FluRate ~ 1 + Region + (1 + Region|Date)');
```

Fit a linear mixed-effects model with fixed effects for the region and a random intercept that varies by Date.

```
lme = fitlme(flu2,'FluRate ~ 1 + Region + (1|Date)');
```

Compare the two models. Also check if lme2 is nested in lme.

```
compare(lme,altlme,'CheckNesting',true)
```

```
ans =
```

## Theoretical Likelihood Ratio Test

Model	DF	AIC	BIC	LogLik	LRStat	deltaDF
lme	11	318.71	364.35	-148.36		
altlme	55	-305.51	-77.346	207.76	712.22	44

The small  $p$ -value of 0 indicates that model altlme is not significantly better than the simpler model lme.

## Test for Fixed and Random Effects

Navigate to a folder containing sample data.

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

# LinearMixedModel.compare

---

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently.

```
lmeBig = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term `(1 | Soil)`.

```
lmeSmall = fitlme(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato)');
```

Compare the two models using the simulated likelihood ratio test with 1000 replications. You must use this test to test for both fixed- and random-effect terms. Note that both models are fit using the default fitting method, ML. That's why, there is no restriction on the fixed-effects design matrices. If you use restricted maximum likelihood (REML) method, both models must have identical fixed-effects design matrices.

```
[table,siminfo] = compare(lmeSmall,lmeBig,'nsim',1000)
```

```
table =
```

```
Simulated Likelihood Ratio Test: Nsim = 1000, Alpha = 0.05
```

Model	DF	AIC	BIC	LogLik	LRStat	pValue
lme2	10	511.06	532	-245.53		
lme	23	522.57	570.74	-238.29	14.491	0.54845

```
siminfo =
```

```
    nsim: 1000  
    alpha: 0.0500  
    pvalueSim: 0.5485  
    pvalueSimCI: [0.5170 0.5796]  
    deltaDF: 13  
    TH0: [1000x1 double]
```

The high  $p$ -value 0.5485 suggests that the larger model, `lme` is not significantly better than the smaller model, `lme2`. The smaller values of AIC and BIC for `lme2` also support this.

## Models with Correlated and Uncorrelated Random Effects

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and the cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

First, prepare the design matrices.

```
X = [ones(406,1) Acceleration Horsepower];
```

# LinearMixedModel.compare

---

```
Z = [ones(406,1) Acceleration];
Model_Year = nominal(Model_Year);
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'});
```

Refit the model with uncorrelated random effects for intercept and acceleration. First prepare the random effects design and the random effects grouping variables.

```
Z = {ones(406,1),Acceleration};
G = {Model_Year,Model_Year};
```

```
lme2 = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept'},{'Acceleration'}},'RandomEffectGroups',...
{'Model_Year','Model_Year'});
```

Compare `lme` and `lme2` using the simulated likelihood ratio test.

```
compare(lme2,lme,'CheckNesting',true,'NSim',1000)
```

```
ans =
```

```
Simulated Likelihood Ratio Test: Nsim = 1000, Alpha = 0.05
```

Model	DF	AIC	BIC	LogLik	LRStat	pValue	
lme2	6	2194.5	2218.3	-1091.3			L
lme	7	2193.5	2221.3	-1089.7	3.0323	0.095904	C

The high  $p$ -value of 0.46807 indicates that `lme2` is not a significantly better fit than `lme`.

## Simulated Likelihood Ratio Test Using Parallel Computing

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term `(1|Soil)`.

```
lme2 = fitlme(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato)');
```

# LinearMixedModel.compare

---

Create the options structure for LinearMixedModel.

```
opt = statset('LinearMixedModel')
```

```
opt =
```

```
    Display: 'off'  
MaxFunEvals: []  
    MaxIter: 10000  
    TolBnd: []  
    TolFun: 1.0000e-06  
TolTypeFun: []  
    TolX: 1.0000e-12  
TolTypeX: []  
    GradObj: []  
    Jacobian: []  
    DerivStep: []  
FunValCheck: []  
    Robust: []  
RobustWgtFun: []  
    WgtFun: []  
    Tune: []  
UseParallel: []  
UseSubstreams: []  
    Streams: {}  
OutputFcn: []
```

Change the options for parallel testing.

```
opt.UseParallel = true;
```

Start a parallel environment.

```
mypool = parpool();
```

```
Starting parpool using the 'local' profile ... connected to 2 workers.
```

```
mypool =
```

Pool with properties:

```
AttachedFiles: {0x1 cell}
NumWorkers: 2
Cluster: [1x1 parallel_cluster.Local]
SpmEnabled: 1
```

Compare `lme2` and `lme` using the simulated likelihood ratio test with 1000 replications and parallel computing.

```
compare(lme2,lme,'nsim',1000,'Options',opt)
```

```
ans =
```

```
Simulated Likelihood Ratio Test: Nsim = 1000, Alpha = 0.05
```

Model	DF	AIC	BIC	LogLik	LRStat	pValue
<code>lme2</code>	10	511.06	532	-245.53		
<code>lme</code>	23	522.57	570.74	-238.29	14.491	0.54845

The high  $p$ -value, 0.5485 suggests that the larger model, `lme` is not significantly better than the smaller model, `lme2`. The smaller values of AIC and BIC for `lme2` also support this.

## Definitions

### Likelihood Ratio Test

Under the null hypothesis  $H_0$ , the observed likelihood ratio test statistic has an approximate chi-squared reference distribution with degrees of freedom `deltaDF`. When comparing two models, `compare` computes the  $p$ -value for the likelihood ratio test by comparing the observed likelihood ratio test statistic with this chi-squared reference distribution.

The  $p$ -values obtained using the likelihood ratio test can be conservative when testing for the presence or absence of random-effects terms and anticonservative when testing for the presence or absence of fixed-effects

terms. Hence, use the `fixedEffects`, `anova`, or `coefTest` method or the simulated likelihood ratio test while testing for fixed effects.

## Simulated Likelihood Ratio Test

To perform the simulated likelihood ratio test, `compare` first generates the reference distribution of the likelihood ratio test statistic under the null hypothesis. Then, it assesses the statistical significance of the alternate model by comparing the observed likelihood ratio test statistic to this reference distribution.

`compare` produces the simulated reference distribution of the likelihood ratio test statistic under the null hypothesis as follows:

- Generate random data `ysim` from the fitted model `lme`.
- Fit the model specified in `lme` and alternate model `altlme` to the simulated data `ysim`.
- Calculate the likelihood ratio test statistic using results from step 2 and store the value.
- Repeat step 1 to 3 `nsim` times.

Then, `compare` computes the  $p$ -value for the simulated likelihood ratio test by comparing the observed likelihood ratio test statistic with the simulated reference distribution. The  $p$ -value estimate is the ratio of the number of times the simulated likelihood ratio test statistic is equal to or exceeds the observed value plus one, to the number of replications plus one.

Suppose the observed likelihood ratio statistic is  $T$ , and the simulated reference distribution is stored in vector  $T_{H_0}$ . Then,

$$p\text{-value} = \frac{\left[ \sum_{j=1}^{nsim} I(T_{H_0}(j) \geq T) \right] + 1}{nsim + 1}.$$



To account for the uncertainty in the simulated reference distribution, `compare` computes a  $100 \times (1 - \alpha)\%$  confidence interval for the true  $p$ -value.

You can use the simulated likelihood ratio test to compare arbitrary linear mixed-effects models. That is, when you are using the simulated likelihood ratio test, `lme` does not have to be nested within `altlme`, and you can fit `lme` and `altlme` using either maximum likelihood (ML) or restricted maximum likelihood (REML) methods. If you use the restricted maximum likelihood (REML) method to fit the models, then both models must have the same fixed-effects design matrix.

## Nesting Requirements

The `'CheckNesting', 'True'` name-value pair argument checks the following requirements.

For a simulated likelihood ratio test:

- You must use the same method to fit both models (`lme` and `altlme`). `compare` cannot compare a model fit using ML to a model fit using REML.
- You must fit both models to the same response vector.
- If you use REML to fit `lme` and `altlme`, then both models must have the same fixed-effects design matrix.
- The maximized log likelihood or restricted log likelihood of the bigger model (`altlme`) must be greater than or equal to that of the smaller model (`lme`).

For a theoretical test, `'CheckNesting', 'True'` checks all the requirements listed for a simulated likelihood ratio test and the following:

- Weight vectors you use to fit `lme` and `altlme` must be identical.
- If you use ML to fit `lme` and `altlme`, the fixed-effects design matrix of the bigger model (`altlme`) must contain that of the smaller model (`lme`).

- The random-effects design matrix of the bigger model (`alt1me`) must contain that of the smaller model (`1me`).

## Akaike and Bayesian Information Criteria

Akaike information criterion (AIC) is  $AIC = -2 * \log L_M + 2 * (nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the model, and  $nc + p + 1$  is the number of parameters estimated in the model.  $p$  is the number of fixed-effects coefficients, and  $nc$  is the total number of parameters in the random-effects covariance excluding the residual variance.

Bayesian information criterion (BIC) is  $BIC = -2 * \log L_M + \ln(n_{eff}) * (nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the model,  $n_{eff}$  is the effective number of observations, and  $(nc + p + 1)$  is the number of parameters estimated in the model.

- If the fitting method is maximum likelihood (ML), then  $n_{eff} = n$ , where  $n$  is the number of observations.
- If the fitting method is restricted maximum likelihood (REML), then  $n_{eff} = n - p$ .

A lower value of deviance indicates a better fit. As the value of deviance decreases, both AIC and BIC tend to decrease. Both AIC and BIC also include penalty terms based on the number of parameters estimated,  $p$ . So, when the number of parameters increase, the values of AIC and BIC tend to increase as well. When comparing different models, the model with the lowest AIC or BIC value is considered as the best fitting model.

## Deviance

`LinearMixedModel` computes the deviance of model  $M$  as minus two times the loglikelihood of that model. Let  $L_M$  denote the maximum value of the likelihood function for model  $M$ . Then, the deviance of model  $M$  is

$$-2 * \log L_M.$$

A lower value of deviance indicates a better fit. Suppose  $M_1$  and  $M_2$  are two different models, where  $M_1$  is nested in  $M_2$ . Then, the fit of the models can be assessed by comparing the deviances  $Dev_1$  and  $Dev_2$  of these models. The difference of the deviances is

$$Dev = Dev_1 - Dev_2 = 2(\log LM_2 - \log LM_1).$$

Usually, the asymptotic distribution of this difference has a chi-square distribution with degrees of freedom  $v$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using  $1 - \text{chi2cdf}(Dev, v)$ , where  $Dev = Dev_2 - Dev_1$ .

However, in mixed-effects models, when some variance components fall on the boundary of the parameter space, the asymptotic distribution of this difference is more complicated. For example, consider the hypotheses

$$H_0: D = \begin{pmatrix} D_{11} & 0 \\ 0 & 0 \end{pmatrix}, \text{ } D \text{ is a } q\text{-by-}q \text{ symmetric positive semidefinite matrix.}$$

$$H_1: D \text{ is a } (q+1)\text{-by-}(q+1) \text{ symmetric positive semidefinite matrix.}$$

That is,  $H_1$  states that the last row and column of  $D$  are different from zero. Here, the bigger model  $M_2$  has  $q + 1$  parameters and the smaller model  $M_1$  has  $q$  parameters. And  $Dev$  has a 50:50 mixture of  $\chi^2_q$  and  $\chi^2_{(q+1)}$  distributions (Stram and Lee, 1994).

## References

- [1] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc., 2002.

# LinearMixedModel.compare

---

[2] Stram D. O. and J. W. Lee. "Variance components testing in the longitudinal mixed-effects model". *Biometrics*, Vol. 50, 4, 1994, pp. 1171–1177.

## See Also

[LinearMixedModel](#) | [fitlme](#) | [fitlmematrixanova](#) | [fixedEffects](#)  
| [randomEffects](#) | [covarianceParameters](#) |

# TreeBagger.ComputeOOBPrediction property

---

## Purpose

Flag to compute out-of-bag predictions

## Description

The ComputeOOBPrediction property is a logical flag specifying whether out-of-bag predictions for training observations should be computed. The default is `false`.

If this flag is true, the following properties are available:

- `OOBIndices`
- `OOBInstanceWeight`

If this flag is true, the following methods can be called:

- `oobError`
- `oobMargin`
- `oobMeanMargin`

## See Also

`oobError` | `OOBIndices` | `OOBInstanceWeight` | `oobMargin` | `oobMeanMargin`

# TreeBagger.ComputeOOBVarImp property

---

**Purpose** Flag to compute out-of-bag variable importance

**Description** The ComputeOOBVarImp property is a logical flag specifying whether TreeBagger should compute out-of-bag estimates of variable importance. The default is false.

If this flag is true, the following properties are available:

- OOBPermutedVarDeltaError
- OOBPermutedVarDeltaMeanMargin
- OOBPermutedVarCountRaiseMargin

**See Also** ComputeOOBPrediction | OOBPermutedVarDeltaError | OOBPermutedVarDeltaMeanMargin | OOBPermutedVarCountRaiseMargin | oobMeanMargin | TreeBagger

## Purpose

Confusion matrix

## Syntax

```
C = confusionmat(group,grouphat)
C = confusionmat(group,grouphat,'order',groupporder)
[C,order] = confusionmat(...)
```

## Description

`C = confusionmat(group,grouphat)` returns the confusion matrix `C` determined by the known and predicted groups in `group` and `grouphat`, respectively. `group` and `grouphat` are grouping variables with the same number of observations, as described in “Grouping Variables” on page 2-51. Input vectors must be of the same type. `C` is a square matrix with size equal to the total number of distinct elements in `group` and `grouphat`. `C(i,j)` is a count of observations known to be in group `i` but predicted to be in group `j`. Group indices and their order are the same for the rows and columns of `C`, computed by `grp2idx` using `grp2idx(group;grouphat)`. NaN, empty, or 'undefined' groups are not counted.

`C = confusionmat(group,grouphat,'order',groupporder)` uses `groupporder` to order the rows and columns of `C`. `groupporder` is a grouping variable containing all of the distinct elements in `group` and `grouphat`. If `groupporder` contains elements that are not in `group` or `grouphat`, the corresponding entries in `C` will be 0.

`[C,order] = confusionmat(...)` also returns the order of the rows and columns of `C` in a variable `order` the same type as `group` and `grouphat`.

## Examples

### Example 1

Display the confusion matrix for data with two misclassifications and one missing classification:

```
g1 = [1 1 2 2 3 3]'; % Known groups
g2 = [1 1 2 3 4 NaN]'; % Predicted groups
```

```
[C,order] = confusionmat(g1,g2)
C =
```

```
    2    0    0    0
    0    1    1    0
    0    0    0    1
    0    0    0    0
order =
    1
    2
    3
    4
```

## Example 2

Randomize the measurements and groups in Fisher's iris data:

```
load fisheriris
numObs = length(species);
p = randperm(numObs);
meas = meas(p,:);
species = species(p);
```

Use `classify` to classify measurements in the second half of the data, using the first half of the data for training:

```
half = floor(numObs/2);
training = meas(1:half,:);
trainingSpecies = species(1:half);
sample = meas(half+1:end,:);
grouphat = classify(sample,training,trainingSpecies);
```

Display the confusion matrix for the resulting classification:

```
group = species(half+1:end);
[C,order] = confusionmat(group,grouphat)
C =
    22    0    0
     2   22    0
     0    0   29
order =
    'virginica'
```



```
'versicolor'  
'setosa'
```

## See Also

`crosstab` | `grp2idx`

## How To

- “Grouping Variables” on page 2-51

# controlchart

---

## Purpose

Shewhart control charts

## Syntax

```
controlchart(X)
controlchart(x,group)
controlchart(X,group)
[stats,plotdata] = controlchart(x,[group])
controlchart(x,group,'name',value)
```

## Description

`controlchart(X)` produces an xbar chart of the measurements in matrix *X*. Each row of *X* is considered to be a subgroup of measurements containing replicate observations taken at the same time. The rows should be in time order. If *X* is a time series object, the time samples should contain replicate observations.

The chart plots the means of the subgroups in time order, a center line (CL) at the average of the means, and upper and lower control limits (UCL, LCL) at three standard errors from the center line. The standard error is the estimated process standard deviation divided by the square root of the subgroup size. Process standard deviation is estimated from the average of the subgroup standard deviations. Out of control measurements are marked as violations and drawn with a red circle. Data cursor mode is enabled, so clicking any data point displays information about that point.

`controlchart(x,group)` accepts a grouping variable `group` for a vector of measurements `x`. `group` is a categorical variable, vector, string array, or cell array of strings the same length as `x`. Consecutive measurements `x(n)` sharing the same value of `group(n)` for  $1 \leq n \leq \text{length}(x)$  are defined to be a subgroup. Subgroups can have different numbers of observations.

`controlchart(X,group)` accepts a grouping variable `group` for a matrix of measurements in *X*. In this case, `group` is only used to label the time axis; it does not change the default grouping by rows.

`[stats,plotdata] = controlchart(x,[group])` returns a structure `stats` of subgroup statistics and parameter estimates, and a structure `plotdata` of plotted values. `plotdata` contains one record for each chart.

The fields in `stats` and `plotdata` depend on the chart type.

The fields in `stats` are selected from the following:

- `mean` — Subgroup means
- `std` — Subgroup standard deviations
- `range` — Subgroup ranges
- `n` — Subgroup size, or total inspection size or area
- `i` — Individual data values
- `ma` — Moving averages
- `mr` — Moving ranges
- `count` — Count of defects or defective items
- `mu` — Estimated process mean
- `sigma` — Estimated process standard deviation
- `p` — Estimated proportion defective
- `m` — Estimated mean defects per unit

The fields in `plotdata` are the following:

- `pts` — Plotted point values
- `cl` — Center line
- `lcl` — Lower control limit
- `ucl` — Upper control limit
- `se` — Standard error of plotted point
- `n` — Subgroup size
- `ooc` — Logical that is true for points that are out of control

`controlchart(x,group,'name',value)` specifies one or more of the following optional parameter name/value pairs, with *name* in single quotes:

# controlchart

---

- `charttype` — The name of a chart type chosen from among the following:
  - `'xbar'` — Xbar or mean
  - `'s'` — Standard deviation
  - `'r'` — Range
  - `'ewma'` — Exponentially weighted moving average
  - `'i'` — Individual observation
  - `'mr'` — Moving range of individual observations
  - `'ma'` — Moving average of individual observations
  - `'p'` — Proportion defective
  - `'np'` — Number of defectives
  - `'u'` — Defects per unit
  - `'c'` — Count of defects

Alternatively, a parameter can be a cell array listing multiple compatible chart types. There are four sets of compatible types:

- `'xbar', 's', 'r', and 'ewma'`
  - `'i', 'mr', and 'ma'`
  - `'p' and 'np'`
  - `'u' and 'c'`
- `display` — Either `'on'` (default) to display the control chart, or `'off'` to omit the display
  - `label` — A string array or cell array of strings, one per subgroup. This label is displayed as part of the data cursor for a point on the plot.
  - `lambda` — A parameter between 0 and 1 controlling how much the current prediction is influenced by past observations in an EWMA

plot. Higher values of 'lambda' give less weight to past observations and more weight to the current observation. The default is 0.4.

- **limits** — A three-element vector specifying the values of the lower control limit, center line, and upper control limits. Default is to estimate the center line and to compute control limits based on the estimated value of sigma. Not permitted if there are multiple chart types.
- **mean** — Value for the process mean, or an empty value (default) to estimate the mean from X. This is the p parameter for p and np charts, the mean defects per unit for u and c charts, and the normal mu parameter for other charts.
- **nsigma** — The number of sigma multiples from the center line to a control limit. Default is 3.
- **parent** — The handle of the axes to receive the control chart plot. Default is to create axes in a new figure. Not permitted if there are multiple chart types.
- **rules** — The name of a control rule, or a cell array containing multiple control rule names. These rules, together with the control limits, determine if a point is marked as out of control. The default is to apply no control rules, and to use only the control limits to decide if a point is out of control. See `controlrules` for more information. Control rules are applied to charts that measure the process level (xbar, i, c, u, p, and np) rather than the variability (r, s), and they are not applied to charts based on moving statistics (ma, mr, ewma).
- **sigma** — Either a value for sigma, or a method of estimating sigma chosen from among 'std' (the default) to use the average within-subgroup standard deviation, 'range' to use the average subgroup range, and 'variance' to use the square root of the pooled variance. When creating i, mr, or ma charts for data not in subgroups, the estimate is always based on a moving range.
- **specs** — A vector specifying specification limits. Typically this is a two-element vector of lower and upper specification limits. Since specification limits typically apply to individual measurements, this

# controlchart

---

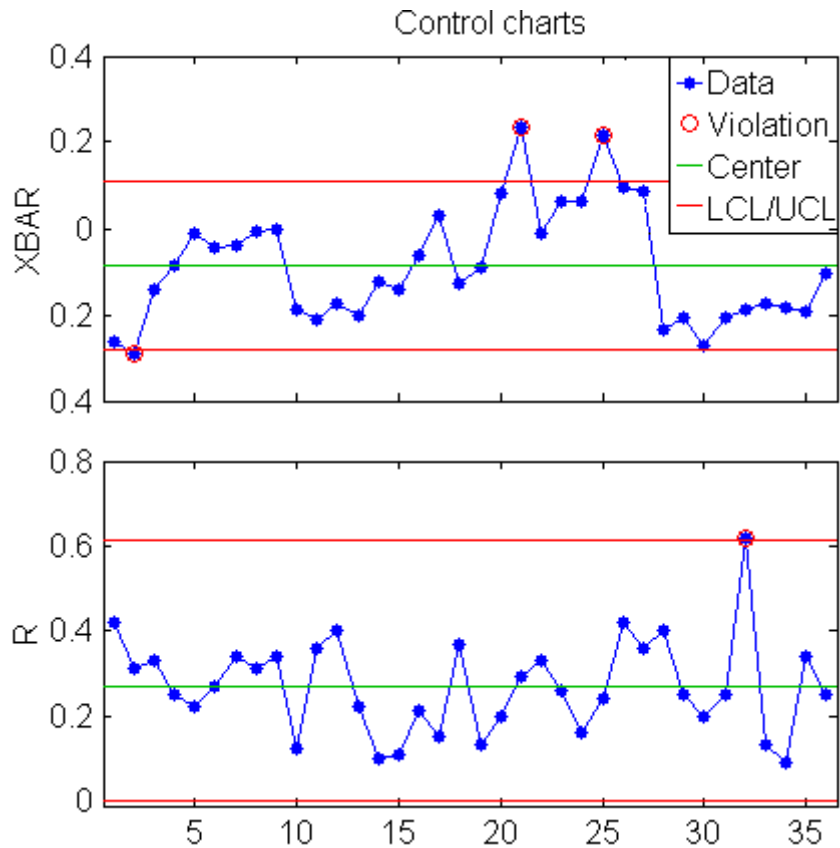
parameter is primarily suitable for *i* charts. These limits are not plotted on *r*, *s*, or *mr* charts.

- `unit` — The total number of inspected items for *p* and *np* charts, and the size of the inspected unit for *u* and *c* charts. In both cases *X* must be the count of the number of defects or defectives found. Default is 1 for *u* and *c* charts. This argument is required (no default) for *p* and *np* charts.
- `width` — The width of the window used for computing the moving ranges and averages in *mr* and *ma* charts, and for computing the sigma estimate in *i*, *mr*, and *ma* charts. Default is 5.

## Examples

Create *xbar* and *r* control charts for the data in `parts.mat`:

```
load parts
st = controlchart(runout, 'chart', {'xbar' 'r'});
```



Display the process mean and standard deviation:

```
fprintf('Parameter estimates: mu = %g, sigma = %g\n',st.mu,st.sigma);
Parameter estimates: mu = -0.0863889, sigma = 0.130215
```

## See Also

controlrules

## How To

- “Grouping Variables” on page 2-51

# controlrules

---

**Purpose** Western Electric and Nelson control rules

**Syntax**  
`R = controlrules('rules',x,c1,se)`  
`[R,RULES] = controlrules('rules',x,c1,se)`

**Description** `R = controlrules('rules',x,c1,se)` determines which points in the vector `x` violate the control rules in `rules`. `c1` is a vector of center-line values. `se` is a vector of standard errors. (Typically, control limits on a control chart are at the values `c1 - 3*se` and `c1 + 3*se`.) `rules` is the name of a control rule, or a cell array containing multiple control rule names, from the list below. If `x` has  $n$  values and `rules` contains  $m$  rules, then `R` is an  $n$ -by- $m$  logical array, with `R(i, j)` assigned the value 1 if point  $i$  violates rule  $j$ , 0 if it does not.

The following are accepted values for `rules` (specified inside single quotes):

- `we1` — 1 point above `c1 + 3*se`
- `we2` — 2 of 3 above `c1 + 2*se`
- `we3` — 4 of 5 above `c1 + se`
- `we4` — 8 of 8 above `c1`
- `we5` — 1 below `c1 - 3*se`
- `we6` — 2 of 3 below `c1 - 2*se`
- `we7` — 4 of 5 below `c1 - se`
- `we8` — 8 of 8 below `c1`
- `we9` — 15 of 15 between `c1 - se` and `c1 + se`
- `we10` — 8 of 8 below `c1 - se` or above `c1 + se`
- `n1` — 1 point below `c1 - 3*se` or above `c1 + 3*se`
- `n2` — 9 of 9 on the same side of `c1`
- `n3` — 6 of 6 increasing or decreasing
- `n4` — 14 alternating up/down



- n5 — 2 of 3 below  $c1 - 2*se$  or above  $c1 + 2*se$ , same side
- n6 — 4 of 5 below  $c1 - se$  or above  $c1 + se$ , same side
- n7 — 15 of 15 between  $c1 - se$  and  $c1 + se$
- n8 — 8 of 8 below  $c1 - se$  or above  $c1 + se$ , either side
- we — All Western Electric rules
- n — All Nelson rules

For multi-point rules, a rule violation at point  $i$  indicates that the set of points ending at point  $i$  triggered the rule. Point  $i$  is considered to have violated the rule only if it is one of the points violating the rule's condition.

Any points with NaN as their  $x$ ,  $c1$ , or  $se$  values are not considered to have violated rules, and are not counted in the rules for other points.

Control rules can be specified in the `controlchart` function as values for the 'rules' parameter.

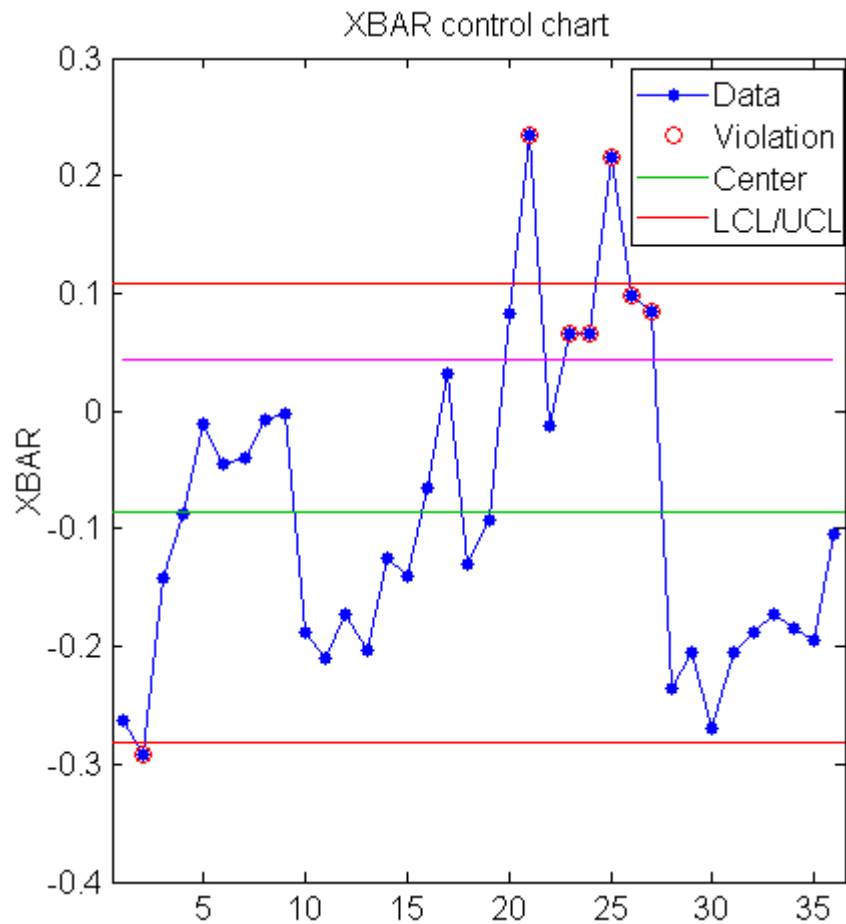
`[R,RULES] = controlrules('rules',x,c1,se)` returns a cell array of text strings RULES listing the rules applied.

## Examples

Create an xbar chart using the `we2` rule to mark out of control measurements:

```
load parts;
st = controlchart(runout, 'rules', 'we2');
x = st.mean;
c1 = st.mu;
se = st.sigma./sqrt(st.n);
hold on
plot(c1+2*se, 'm')
```

# controlrules



Use controlrules to identify the measurements that violate the control rule:

```
R = controlrules('we2',x,c1,se);  
I = find(R)  
I =  
    21  
    23
```

24  
25  
26  
27

**See Also**      controlchart

# gmdistribution.Converged property

---

**Purpose** Determine if algorithm converged

**Description** Logical true if the algorithm has converged; logical false if the algorithm has not converged.

---

**Note** This property applies only to gmdistribution objects constructed with `fit`.

---

**Purpose** Cophenetic correlation coefficient

**Syntax**  
`c = cophenet(Z,Y)`  
`[c,d] = cophenet(Z,Y)`

**Description** `c = cophenet(Z,Y)` computes the cophenetic correlation coefficient for the hierarchical cluster tree represented by `Z`. `Z` is the output of the linkage function. `Y` contains the distances or dissimilarities used to construct `Z`, as output by the `pdist` function. `Z` is a matrix of size  $(m-1)$ -by-3, with distance information in the third column. `Y` is a vector of size  $m*(m-1)/2$ .

`[c,d] = cophenet(Z,Y)` returns the cophenetic distances `d` in the same lower triangular distance vector format as `Y`.

The cophenetic correlation for a cluster tree is defined as the linear correlation coefficient between the cophenetic distances obtained from the tree, and the original distances (or dissimilarities) used to construct the tree. Thus, it is a measure of how faithfully the tree represents the dissimilarities among observations.

The cophenetic distance between two observations is represented in a dendrogram by the height of the link at which those two observations are first joined. That height is the distance between the two subclusters that are merged by that link.

The output value, `c`, is the cophenetic correlation coefficient. The magnitude of this value should be very close to 1 for a high-quality solution. This measure can be used to compare alternative cluster solutions obtained using different algorithms.

The cophenetic correlation between `Z(:,3)` and `Y` is defined as

$$c = \frac{\sum_{i < j} (Y_{ij} - y)(Z_{ij} - z)}{\sqrt{\sum_{i < j} (Y_{ij} - y)^2 \sum_{i < j} (Z_{ij} - z)^2}}$$

where:

- $Y_{ij}$  is the distance between objects  $i$  and  $j$  in `Y`.

# cophenet

---

- $Z_{ij}$  is the cophenetic distance between objects  $i$  and  $j$ , from  $Z(:,3)$ .
- $y$  and  $z$  are the average of  $Y$  and  $Z(:,3)$ , respectively.

## Examples

```
X = [rand(10,3); rand(10,3)+1; rand(10,3)+2];
Y = pdist(X);
Z = linkage(Y, 'average');

% Compute Spearman's rank correlation between the
% dissimilarities and the cophenetic distances
[c,D] = cophenet(Z,Y);
r = corr(Y',D', 'type', 'spearman')
r =
    0.8279
```

## See Also

[cluster](#) | [dendrogram](#) | [inconsistent](#) | [linkage](#) | [pdist](#) | [squareform](#)

**Purpose** Copula cumulative distribution function

**Syntax**

```
Y = copulacdf('Gaussian',U,rho)
Y = copulacdf('t',U,rho,NU)
Y = copulacdf('family',U,alpha)
```

**Description**

`Y = copulacdf('Gaussian',U,rho)` returns the cumulative probability of the Gaussian copula with linear correlation parameters `rho`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`, representing `n` points in the `p`-dimensional unit hypercube. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-2 matrix, `rho` may also be a scalar correlation coefficient.

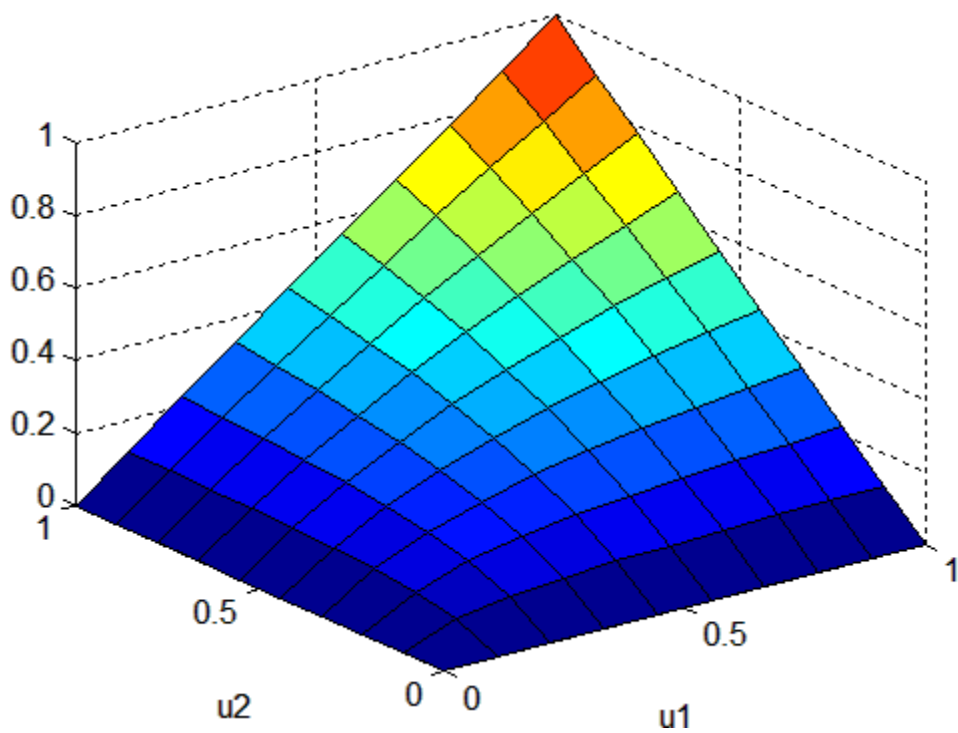
`Y = copulacdf('t',U,rho,NU)` returns the cumulative probability of the `t` copula with linear correlation parameters `rho` and degrees of freedom parameter `NU`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-2 matrix, `rho` may also be a scalar correlation coefficient.

`Y = copulacdf('family',U,alpha)` returns the cumulative probability of the bivariate Archimedean copula determined by `family`, with scalar parameter `alpha`, evaluated at the points in `U`. `family` is Clayton, Frank, or Gumbel. `U` is an `n`-by-2 matrix of values in `[0,1]`.

## Examples

```
u = linspace(0,1,10);
[U1,U2] = meshgrid(u,u);
F = copulacdf('Clayton',[U1(:) U2(:)],1);
surf(U1,U2,reshape(F,10,10))
xlabel('u1')
ylabel('u2')
```

# copulacdf



## See Also

[copulapdf](#) | [copularnd](#) | [copulastat](#) | [copulaparam](#)



## Purpose

Fit copula to data

## Syntax

```
RHOHAT = copulafit('Gaussian',U)
[RHOHAT,nuhat] = copulafit('t',U)
[RHOHAT,nuhat,nuci] = copulafit('t',U)
paramhat = copulafit('family',U)
[paramhat,paramci] = copulafit('family',U)
[...] = copulafit(...,'alpha',alpha)
[...] = copulafit('t',U,'Method','ApproximateML')
[...] = copulafit(...,'Options',options)
```

## Description

`RHOHAT = copulafit('Gaussian',U)` returns an estimate `RHOHAT` of the matrix of linear correlation parameters for a Gaussian copula, given data in `U`. `U` is an  $n$ -by- $p$  matrix of values in the open interval  $(0,1)$  representing  $n$  points in the  $p$ -dimensional unit hypercube.

`[RHOHAT,nuhat] = copulafit('t',U)` returns an estimate `RHOHAT` of the matrix of linear correlation parameters for a  $t$  copula and an estimate `nuhat` of the degrees of freedom parameter, given data in `U`. `U` is an  $n$ -by- $p$  matrix of values in the open interval  $(0,1)$  representing  $n$  points in the  $p$ -dimensional unit hypercube.

`[RHOHAT,nuhat,nuci] = copulafit('t',U)` also returns an approximate 95% confidence interval `nuci` for the degrees of freedom parameter estimated in `nuhat`.

`paramhat = copulafit('family',U)` returns an estimate `paramhat` of the copula parameter for an Archimedean copula specified by `family`, given data in `U`. `U` is an  $n$ -by-2 matrix of values in the open interval  $(0,1)$  representing  $n$  points in the unit square. `family` is one of Clayton, Frank, or Gumbel.

`[paramhat,paramci] = copulafit('family',U)` also returns an approximate 95% confidence interval `paramci` for the copula parameter estimated in `paramhat`.

`[...] = copulafit(...,'alpha',alpha)` returns approximate  $100 \cdot (1 - \alpha)\%$  confidence intervals in `nuci` or `paramci`.

---

**Note** By default, `copulafit` uses maximum likelihood to fit a copula to `U`. When `U` contains data transformed to the unit hypercube by parametric estimates of their marginal cumulative distribution functions, this is known as the *Inference Functions for Margins (IFM)* method. When `U` contains data transformed by the empirical cdf (see `ecdf`), this is known as *Canonical Maximum Likelihood (CML)*.

---

`[...] = copulafit('t',U,'Method','ApproximateML')` fits a  $t$  copula for large samples `U` by maximizing an objective function that approximates the profile log-likelihood for the degrees of freedom parameter (see [1]). This method can be significantly faster than maximum likelihood, but the estimates and confidence limits may not be accurate for small to moderate sample sizes.

`[...] = copulafit(...,'Options',options)` specifies control parameters for the iterative parameter estimation algorithm using an options structure `options` as created by `statset`. Type `statset('copulafit')` at the command prompt for fields and default values used by `copulafit`. This argument is not applicable to the 'Gaussian' family.

## References

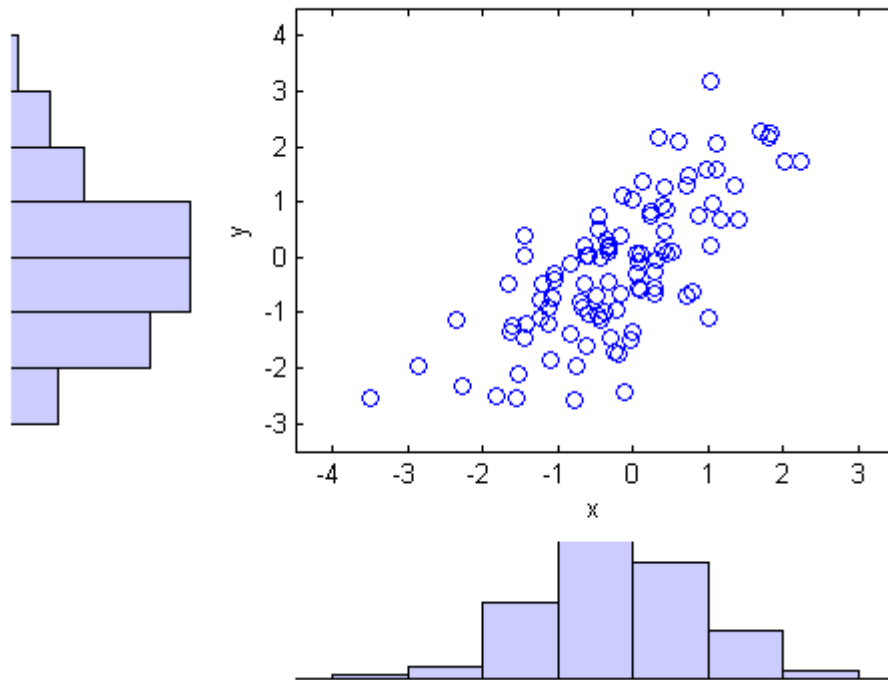
[1] Bouye, E., V. Durrleman, A. Nikeghbali, G. Riboulet, and T. Roncalli. "Copulas for Finance: A Reading Guide and Some Applications." Working Paper. Groupe de Recherche Operationnelle, Credit Lyonnais, 2000.

## Examples

Load and plot simulated stock return data:

```
load stockreturns
x = stocks(:,1);
y = stocks(:,2);

scatterhist(x,y)
```

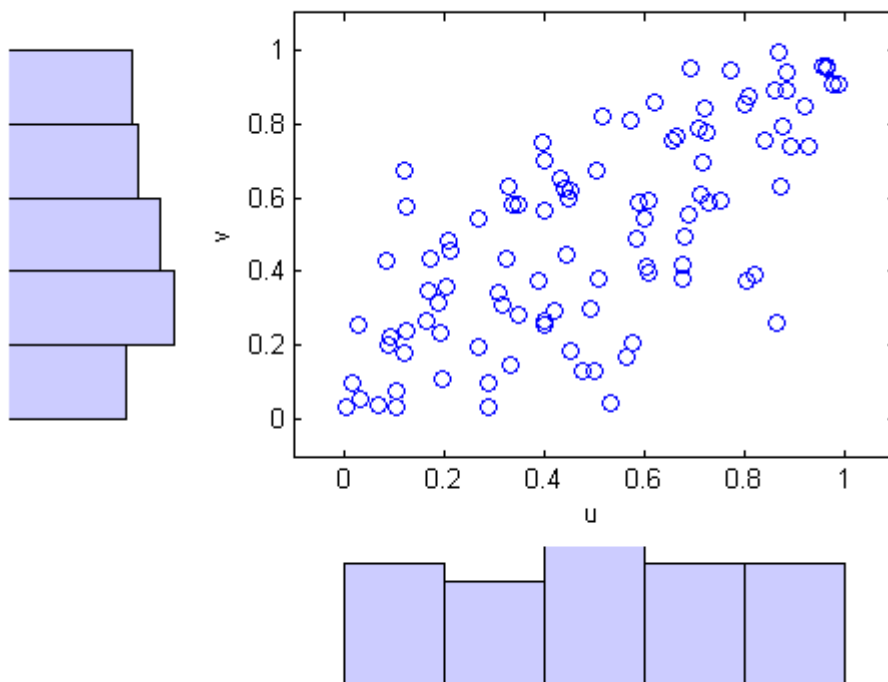


Transform the data to the copula scale (unit square) using a kernel estimator of the cumulative distribution function:

```
u = ksdensity(x,x,'function','cdf');  
v = ksdensity(y,y,'function','cdf');  
  
scatterhist(u,v)  
xlabel('u')  
ylabel('v')
```

# copulafit

---



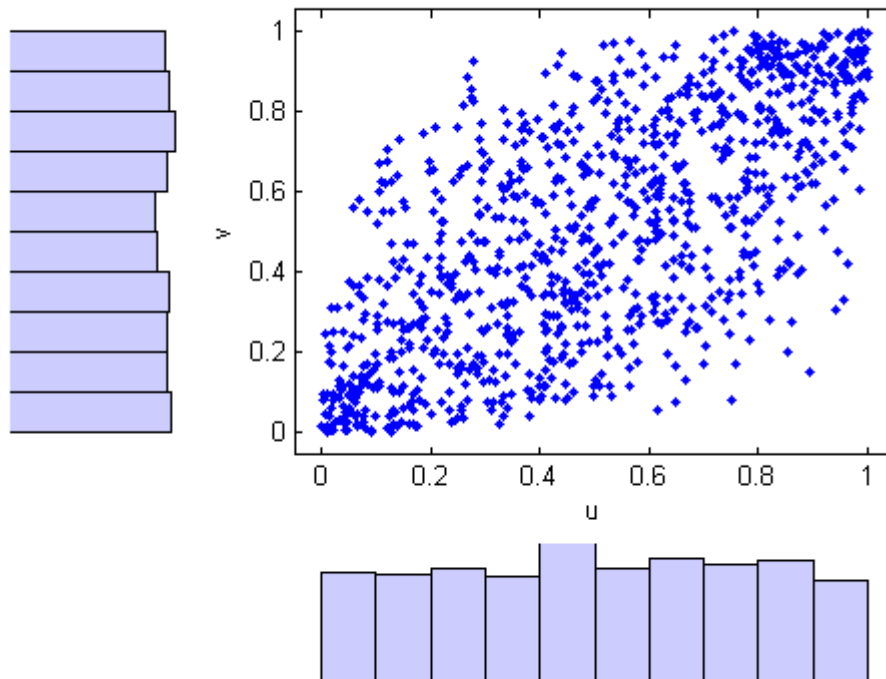
Fit a  $t$  copula:

```
[Rho,nu] = copulafit('t',[u v],'Method','ApproximateML')  
Rho =  
    1.0000    0.7220  
    0.7220    1.0000  
nu =  
    2.8934e+006
```

Generate a random sample from the  $t$  copula:

```
r = copularnd('t',Rho,nu,1000);
```

```
u1 = r(:,1);  
v1 = r(:,2);  
  
scatterhist(u1,v1)  
xlabel('u')  
ylabel('v')  
set(get(gca,'children'),'marker','.')
```



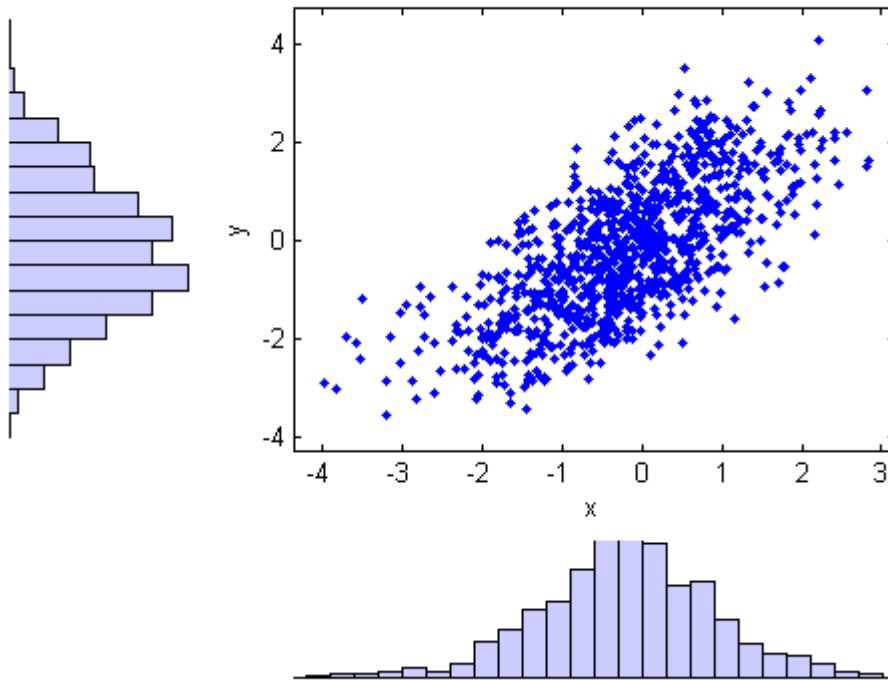
Transform the random sample back to the original scale of the data:

```
x1 = ksdensity(x,u1,'function','icdf');
```

# copulafit

---

```
y1 = ksdensity(y,v1,'function','icdf');  
  
scatterhist(x1,y1)  
set(get(gca,'children'),'marker','.')
```



## See Also

[ecdf](#) | [copulacdf](#) | [copulaparam](#) | [copulapdf](#) | [copularnd](#) | [copulastat](#)

**Purpose** Copula parameters as function of rank correlation

**Syntax**

```
rho = copulaparam('Gaussian',R)
rho = copulaparam('t',R,NU)
alpha = copulaparam(family,R)
[...] = copulaparam(...,'type',type)
```

**Description** `rho = copulaparam('Gaussian',R)` returns the linear correlation parameters `rho` corresponding to a Gaussian copula having Kendall's rank correlation `R`. If `R` is a scalar correlation coefficient, `rho` is a scalar correlation coefficient corresponding to a bivariate copula. If `R` is a `p`-by-`p` correlation matrix, `rho` is a `p`-by-`p` correlation matrix.

`rho = copulaparam('t',R,NU)` returns the linear correlation parameters `rho` corresponding to a `t` copula having Kendall's rank correlation `R` and degrees of freedom `NU`. If `R` is a scalar correlation coefficient, `rho` is a scalar correlation coefficient corresponding to a bivariate copula. If `R` is a `p`-by-`p` correlation matrix, `rho` is a `p`-by-`p` correlation matrix.

`alpha = copulaparam(family,R)` returns the copula parameter `alpha` corresponding to a bivariate Archimedean copula having Kendall's rank correlation `R`. `R` is a scalar. `family` is one of 'Clayton', 'Frank', or 'Gumbel'.

`[...] = copulaparam(...,'type',type)` assumes `R` is the specified type of rank correlation. `type` is 'Kendall' for Kendall's tau or 'Spearman' for Spearman's rho.

`copulaparam` uses an approximation to Spearman's rank correlation for copula families when no analytic formula exists. The approximation is based on a smooth fit to values computed at discrete values of the copula parameters. For a `t` copula, the approximation is accurate for degrees of freedom larger than 0.05.

**Examples** Get the linear correlation coefficient corresponding to a bivariate Gaussian copula having a rank correlation of -0.5.

```
tau = -0.5
```

# copulaparam

---

```
rho = copulaparam('gaussian',tau)
rho =
    -0.7071

% Generate dependent beta random values using that copula
u = copularnd('gaussian',rho,100);
b = betainv(u,2,2);

% Verify that the sample has a rank correlation
% approximately equal to tau
tau_sample = corr(b,'type','k')
tau_sample =
    1.0000    -0.4638
   -0.4638    1.0000
```

## See Also

[copulacdf](#) | [copulapdf](#) | [copularnd](#) | [copulastat](#)



**Purpose**

Copula probability density function

**Syntax**

```
Y = copulapdf('Gaussian',U,rho)
Y = copulapdf('t',U,rho,NU)
Y = copulapdf('family',U,alpha)
```

**Description**

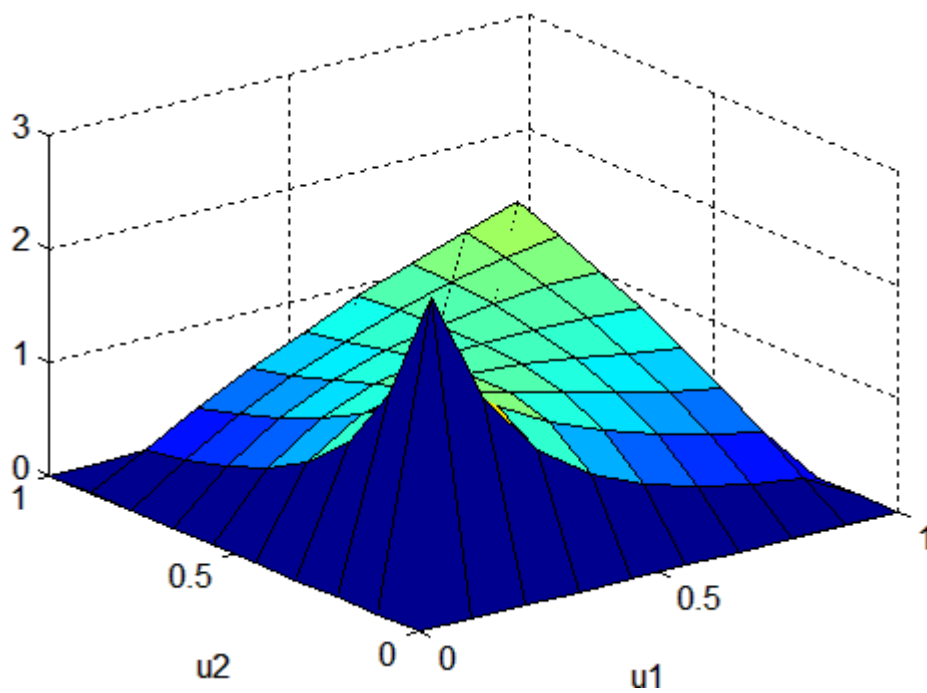
`Y = copulapdf('Gaussian',U,rho)` returns the probability density of the Gaussian copula with linear correlation parameters `rho`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`, representing `n` points in the `p`-dimensional unit hypercube. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-2 matrix, `rho` may also be a scalar correlation coefficient.

`Y = copulapdf('t',U,rho,NU)` returns the probability density of the `t` copula with linear correlation parameters `rho` and degrees of freedom parameter `NU`, evaluated at the points in `U`. `U` is an `n`-by-`p` matrix of values in `[0,1]`. `rho` is a `p`-by-`p` correlation matrix. If `U` is an `n`-by-2 matrix, `rho` may also be a scalar correlation coefficient.

`Y = copulapdf('family',U,alpha)` returns the probability density of the bivariate Archimedean copula determined by `family`, with scalar parameter `alpha`, evaluated at the points in `U`. `family` is Clayton, Frank, or Gumbel. `U` is an `n`-by-2 matrix of values in `[0,1]`.

**Examples**

```
u = linspace(0,1,10);
[U1,U2] = meshgrid(u,u);
F = copulapdf('Clayton',[U1(:) U2(:)],1);
surf(U1,U2,reshape(F,10,10))
xlabel('u1')
ylabel('u2')
```



**See Also**

[copulacdf](#) | [copulaparam](#) | [copularnd](#) | [copulastat](#)

## Purpose

Copula rank correlation

## Syntax

```
R = copulastat('Gaussian',rho)
R = copulastat('t',rho,NU)
R = copulastat(family,alpha)
R = copulastat(...,'type',type)
```

## Description

`R = copulastat('Gaussian',rho)` returns the Kendall's rank correlation  $R$  that corresponds to a Gaussian copula having linear correlation parameters  $\rho$ . If  $\rho$  is a scalar correlation coefficient,  $R$  is a scalar correlation coefficient corresponding to a bivariate copula. If  $\rho$  is a  $p$ -by- $p$  correlation matrix,  $R$  is a  $p$ -by- $p$  correlation matrix.

`R = copulastat('t',rho,NU)` returns the Kendall's rank correlation  $R$  that corresponds to a  $t$  copula having linear correlation parameters  $\rho$  and degrees of freedom  $NU$ . If  $\rho$  is a scalar correlation coefficient,  $R$  is a scalar correlation coefficient corresponding to a bivariate copula. If  $\rho$  is a  $p$ -by- $p$  correlation matrix,  $R$  is a  $p$ -by- $p$  correlation matrix.

`R = copulastat(family,alpha)` returns the Kendall's rank correlation  $R$  that corresponds to a bivariate Archimedean copula with scalar parameter  $\alpha$ . `family` is one of 'Clayton', 'Frank', or 'Gumbel'.

`R = copulastat(...,'type',type)` returns the specified type of rank correlation. `type` is 'Kendall' to compute Kendall's tau, or 'Spearman' to compute Spearman's rho.

`copulastat` uses an approximation to Spearman's rank correlation for copula families when no analytic formula exists. The approximation is based on a smooth fit to values computed at discrete values of the copula parameters. For a  $t$  copula, the approximation is accurate for degrees of freedom larger than 0.05.

## Examples

Get the theoretical rank correlation coefficient for a bivariate.

```
% Gaussian copula with linear correlation parameter rho
rho = -.7071;
tau = copulastat('gaussian',rho)
tau =
```

```
-0.5000
```

```
% Generate dependent beta random values using that copula  
u = copularnd('gaussian',rho,100);  
b = betainv(u,2,2);
```

```
% Verify that the sample has a rank correlation  
% approximately equal to tau  
tau_sample = corr(b,'type','k')  
tau_sample =  
    1.0000   -0.5265  
   -0.5265    1.0000
```

## See Also

[copulacdf](#) | [copulaparam](#) | [copulapdf](#) | [copularnd](#)

<b>Purpose</b>	Copula random numbers
<b>Syntax</b>	<pre>U = copularnd('Gaussian',rho,N) U = copularnd('t',rho,NU,N) U = copularnd('family',alpha,N)</pre>
<b>Description</b>	<p><code>U = copularnd('Gaussian',rho,N)</code> returns <math>N</math> random vectors generated from a Gaussian copula with linear correlation parameters <code>rho</code>. If <code>rho</code> is a <math>p</math>-by-<math>p</math> correlation matrix, <code>U</code> is an <math>n</math>-by-<math>p</math> matrix. If <code>rho</code> is a scalar correlation coefficient, <code>copularnd</code> generates <code>U</code> from a bivariate Gaussian copula. Each column of <code>U</code> is a sample from a <code>Uniform(0,1)</code> marginal distribution.</p> <p><code>U = copularnd('t',rho,NU,N)</code> returns <math>N</math> random vectors generated from a <code>t</code> copula with linear correlation parameters <code>rho</code> and degrees of freedom <code>NU</code>. If <code>rho</code> is a <math>p</math>-by-<math>p</math> correlation matrix, <code>U</code> is an <math>n</math>-by-<math>p</math> matrix. If <code>rho</code> is a scalar correlation coefficient, <code>copularnd</code> generates <code>U</code> from a bivariate <code>t</code> copula. Each column of <code>U</code> is a sample from a <code>Uniform(0,1)</code> marginal distribution.</p> <p><code>U = copularnd('family',alpha,N)</code> returns <math>N</math> random vectors generated from the bivariate Archimedean copula determined by <code>family</code>, with scalar parameter <code>alpha</code>. <code>family</code> is Clayton, Frank, or Gumbel. <code>U</code> is an <math>n</math>-by-2 matrix. Each column of <code>U</code> is a sample from a <code>Uniform(0,1)</code> marginal distribution.</p>
<b>Examples</b>	<p>Determine the linear correlation parameter corresponding to a bivariate Gaussian copula having a rank correlation of <code>-0.5</code>.</p> <pre>tau = -0.5 rho = copulaparam('gaussian',tau) rho =     -0.7071</pre> <p><code>% Generate dependent beta random values using that copula</code></p> <pre>u = copularnd('gaussian',rho,100); b = betainv(u,2,2);</pre>

# copularnd

---

```
% Verify that the sample has a rank correlation
% approximately equal to tau
tau_sample = corr(b,'type','kendall')
tau_sample =
    1.0000   -0.4537
   -0.4537    1.0000
```

## See Also

[copulacdf](#) | [copulaparam](#) | [copulapdf](#) | [copulastat](#)

**Purpose**

Coordinate exchange

**Syntax**

```
dCE = cordexch(nfactors,nruns)
[dCE,X] = cordexch(nfactors,nruns)
[dCE,X] = cordexch(nfactors,nruns,'model')
[dCE,X] = cordexch(...,'name',value)
```

**Description**

`dCE = cordexch(nfactors,nruns)` uses a coordinate-exchange algorithm to generate a  $D$ -optimal design `dCE` with `nruns` runs (the rows of `dCE`) for a linear additive model with `nfactors` factors (the columns of `dCE`). The model includes a constant term.

`[dCE,X] = cordexch(nfactors,nruns)` also returns the associated design matrix `X`, whose columns are the model terms evaluated at each treatment (row) of `dCE`.

`[dCE,X] = cordexch(nfactors,nruns,'model')` uses the linear regression model specified in `model`. `model` is one of the following strings, specified inside single quotes:

- `linear` — Constant and linear terms. This is the default.
- `interaction` — Constant, linear, and interaction terms
- `quadratic` — Constant, linear, interaction, and squared terms
- `purequadratic` — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with  $n$  terms is:

- 1** The constant term
- 2** The linear terms in order 1, 2, ...,  $n$
- 3** The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ )
- 4** The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors X1, X2, and X3, then a row [0 1 2] in *model* specifies the term  $(X1.^0) \cdot (X2.^1) \cdot (X3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

[dCE,X] = cordexch(..., 'name', value) specifies one or more optional name/value pairs for the design. Valid parameters and their values are listed in the following table. Specify *name* inside single quotes.

name	Value
bounds	Lower and upper bounds for each factor, specified as a 2-by-nfactors matrix. Alternatively, this value can be a cell array containing nfactors elements, each element specifying the vector of allowable values for the corresponding factor.
categorical	Indices of categorical predictors.
display	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
excludefun	Handle to a function that excludes undesirable runs. If the function is <i>f</i> , it must support the syntax $b = f(S)$ , where <i>S</i> is a matrix of treatments with nfactors columns and <i>b</i> is a vector of Boolean values with the same number of rows as <i>S</i> . <i>b</i> ( <i>i</i> ) is true if the method should exclude <i>i</i> th row <i>S</i> .
init	Initial design as a nruns-by-nfactors matrix. The default is a randomly selected set of points.
levels	Vector of number of levels for each factor. Not used when bounds is specified as a cell array
maxiter	Maximum number of iterations. The default is 10.



name	Value
tries	Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.
options	<p>A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the options structure with <code>statset</code>. Option fields:</p> <ul style="list-style-type: none"> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>cordexch</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>▪ You have an open Parallel pool</li> <li>▪ <code>UseParallel</code> is <code>true</code></li> <li>▪ <code>UseSubstreams</code> is <code>false</code></li> </ul> <p>In that case, use a cell array the same size as the Parallel pool.</p> </li> </ul>

## Algorithms

Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix  $X$  to increase  $D = |X^T X|$  at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally,  $D$ -optimal designs. Run each algorithm multiple times

and select the best result for your final design. Both functions have a 'tries' parameter that automates this repetition and comparison.

Unlike the row-exchange algorithm used by rowexch, cordexch does not use a candidate set. (Or rather, the candidate set is the entire design space.) At each step, the coordinate-exchange algorithm exchanges a single element of  $X$  with a new element evaluated at a neighboring point in design space. The absence of a candidate set reduces demands on memory, but the smaller scale of the search means that the coordinate-exchange algorithm is more likely to become trapped in a local minimum.

## Examples

Suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_3 + \beta_{12}x_1x_2 + \beta_{13}x_1x_3 + \beta_{23}x_2x_3 + \varepsilon$$

Use cordexch to generate a  $D$ -optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dCE,X] = cordexch(nfactors,nruns,'interaction','tries',10)
dCE =
    -1     1     1
    -1    -1    -1
     1     1     1
    -1     1    -1
     1    -1     1
     1    -1    -1
    -1    -1     1
X =
     1    -1     1     1    -1    -1     1
     1    -1    -1    -1     1     1     1
     1     1     1     1     1     1     1
     1    -1     1    -1    -1     1    -1
     1     1    -1     1    -1     1    -1
     1     1    -1    -1    -1    -1     1
     1    -1    -1     1     1    -1    -1
```

Columns of the design matrix  $X$  are the model terms evaluated at each row of the design  $dCE$ . The terms appear in order from left to right: constant term, linear terms (1, 2, 3), interaction terms (12, 13, 23). Use  $X$  to fit the model, as described in “Linear Regression” on page 9-12, to response data measured at the design points in  $dCE$ .

**See Also**

rowexch | daugment | dcovary

**Purpose** Linear or rank correlation

**Syntax**

```
RHO = corr(X)
RHO = corr(X,Y)
[RHO,PVAL] = corr(X,Y)
[RHO,PVAL] = corr(X,Y, 'name', value)
```

**Description** RHO = corr(X) returns a  $p$ -by- $p$  matrix containing the pairwise linear correlation coefficient between each pair of columns in the  $n$ -by- $p$  matrix X.

RHO = corr(X,Y) returns a  $p1$ -by- $p2$  matrix containing the pairwise correlation coefficient between each pair of columns in the  $n$ -by- $p1$  and  $n$ -by- $p2$  matrices X and Y.

The difference between corr(X,Y) and the MATLAB function corrcoef(X,Y) is that corrcoef(X,Y) returns a matrix of correlation coefficients for the two column vectors X and Y. If X and Y are not column vectors, corrcoef(X,Y) converts them to column vectors.

[RHO,PVAL] = corr(X,Y) also returns PVAL, a matrix of  $p$ -values for testing the hypothesis of no correlation against the alternative that there is a nonzero correlation. Each element of PVAL is the  $p$  value for the corresponding element of RHO. If PVAL(i,j) is small, say less than 0.05, then the correlation RHO(i,j) is significantly different from zero.

[RHO,PVAL] = corr(X,Y, 'name', value) specifies one or more optional name/value pairs. Specify *name* inside single quotes. The following table lists valid parameters and their values.

Parameter	Values
type	<ul style="list-style-type: none"> <li>'Pearson' (the default) computes Pearson's linear correlation coefficient</li> <li>'Kendall' computes Kendall's tau</li> <li>'Spearman' computes Spearman's rho</li> </ul>
rows	<ul style="list-style-type: none"> <li>'all' (the default) uses all rows regardless of missing values (NaNs)</li> <li>'complete' uses only rows with no missing values</li> <li>'pairwise' computes <math>RHO(i, j)</math> using rows with no missing values in column <math>i</math> or <math>j</math></li> </ul>
tail — The alternative hypothesis against which to compute $p$ -values for testing the hypothesis of no correlation	<ul style="list-style-type: none"> <li>'both' — Correlation is not zero (the default)</li> <li>'right' — Correlation is greater than zero</li> <li>'left' — Correlation is less than zero</li> </ul>

Using the 'pairwise' option for the rows parameter may return a matrix that is not positive definite. The 'complete' option always returns a positive definite matrix, but in general the estimates are based on fewer observations.

corr computes  $p$ -values for Pearson's correlation using a Student's  $t$  distribution for a transformation of the correlation. This correlation is exact when  $X$  and  $Y$  are normal. corr computes  $p$ -values for Kendall's tau and Spearman's rho using either the exact permutation distributions (for small sample sizes), or large-sample approximations.

corr computes  $p$ -values for the two-tailed test by doubling the more significant of the two one-tailed  $p$ -values.

**Example****Find Correlation Between Two Matrices**

Find the correlation between two matrices and compare to the correlation between two column vectors.

Generate sample data.

```
rng('default')
x = randn(30,4);
y = randn(30,4);
y(:,4) = sum(x,2); % introduce correlation
```

Calculate the correlation between columns of X and Y.

```
[r,p] = corr(x,y)
```

r =

```
-0.1686    -0.0363     0.2278     0.6901
 0.3022     0.0332    -0.0866     0.2617
-0.3632    -0.0987    -0.0200     0.3504
-0.1365    -0.1804     0.0853     0.4908
```

p =

```
0.3731     0.8489     0.2260     0.0000
0.1045     0.8619     0.6491     0.1624
0.0485     0.6039     0.9166     0.0577
0.4721     0.3400     0.6539     0.0059
```

Calculate the correlation between X and Y using corrcoef.

```
[r,p] = corrcoef(x,y)
```

r =

```
1.0000     0.1252
0.1252     1.0000
```

p =

```
    1.0000    0.1729
    0.1729    1.0000
```

MATLAB function `corrcoef` converts X and Y into column vectors before computing the correlation between them.

## References

- [1] Gibbons, J.D. (1985) Nonparametric Statistical Inference, 2nd ed., M. Dekker.
- [2] Hollander, M. and D.A. Wolfe (1973) Nonparametric Statistical Methods, Wiley.
- [3] Kendall, M.G. (1970) Rank Correlation Methods, Griffin.
- [4] Best, D.J. and D.E. Roberts (1975) "Algorithm AS 89: The Upper Tail Probabilities of Spearman's rho", Applied Statistics, 24:377-379.

## See Also

`corrcoef` | `partialcorr` | `corrcoef` | `tiedrank`

# corrcoef

---

**Purpose** Convert covariance matrix to correlation matrix

**Syntax** `R = corrcoef(C)`  
`[R,sigma] = corrcoef(C)`

**Description** `R = corrcoef(C)` computes the correlation matrix `R` corresponding to the covariance matrix `C`. `C` must be square, symmetric, and positive semi-definite.

`[R,sigma] = corrcoef(C)` also computes the vector of standard deviations `sigma`.

**Examples** Use `cov` and `corrcoef` to compute covariances and correlations, respectively, for sample data on weight and blood pressure (systolic, diastolic) in `hospital.mat`:

```
load hospital
X = [hospital.Weight hospital.BloodPressure];
C = cov(X)
C =
    706.0404    27.7879    41.0202
    27.7879    45.0622    23.8194
    41.0202    23.8194    48.0590
R = corrcoef(X)
R =
    1.0000    0.1558    0.2227
    0.1558    1.0000    0.5118
    0.2227    0.5118    1.0000
```

Compare `R` with the correlation matrix computed from `C` by `corrcoef`:

```
corrcoef(C)
ans =
    1.0000    0.1558    0.2227
    0.1558    1.0000    0.5118
    0.2227    0.5118    1.0000
```

**See Also** `cov` | `corrcoef` | `corr` | `cholcov`



<b>Purpose</b>	Misclassification costs
<b>Description</b>	The Cost property is a matrix with misclassification costs. This property is empty for ensembles of regression trees.
<b>See Also</b>	<code>classregtree</code>

# LinearMixedModel.covarianceParameters

---

<b>Purpose</b>	Extract covariance parameters of linear mixed-effects model
<b>Syntax</b>	<pre>psi = covarianceParameters(lme) [psi,mse] = covarianceParameters(lme) [psi,mse,stats] = covarianceParameters(lme) [psi,mse,stats] = covarianceParameters(lme,Name,Value)</pre>
<b>Description</b>	<p><code>psi = covarianceParameters(lme)</code> returns the estimated covariance parameters that parameterize the prior covariance of random effects.</p> <p><code>[psi,mse] = covarianceParameters(lme)</code> also returns an estimate of the residual variance.</p> <p><code>[psi,mse,stats] = covarianceParameters(lme)</code> also returns a cell array, <code>stats</code>, containing the covariance parameters and related statistics.</p> <p><code>[psi,mse,stats] = covarianceParameters(lme,Name,Value)</code> returns the covariance parameters and related statistics in <code>stats</code> with additional options specified by one or more <code>Name,Value</code> pair arguments.</p> <p>For example, you can specify the confidence level for the confidence limits of covariance parameters.</p>
<b>Input Arguments</b>	<p><b>lme - Linear mixed-effects model</b> <code>LinearMixedModel</code> object</p> <p>Linear mixed-effects model, returned as a <code>LinearMixedModel</code> object.</p> <p>For properties and methods of this object, see <code>LinearMixedModel</code>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding</p>

value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

## 'Alpha' - Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

**Example:** 'Alpha', 0.01

## Data Types

single | double

## Output Arguments

### psi - Estimate of covariance parameters

cell array

Estimate of covariance parameters that parameterize the prior covariance of the random effects, returned as a cell array of length  $R$ , such that  $\text{psi}\{r\}$  contains the covariance matrix of random effects associated with grouping variable  $g_r$ ,  $r = 1, 2, \dots, R$ . The order of grouping variables is the same order you enter when you fit the model.

### mse - Residual variance estimate

scalar value

Residual variance estimate, returned as a scalar value.

### stats - Covariance parameter estimates and related statistics

cell array

Covariance parameter estimates and related statistics, returned as a cell array of length  $(R + 1)$  containing dataset arrays with the following columns.

# LinearMixedModel.covarianceParameters

---

Group	Grouping variable name
Name1	Name of the first predictor variable
Name2	Name of the second predictor variable
Type	std (standard deviation), if Name1 and Name2 are the same corr (correlation), if Name1 and Name2 are different
Estimate	Standard deviation of the random effect associated with predictor Name1 or Name2, if Name1 and Name2 are the same Correlation between the random effects associated with predictors Name1 and Name2, if Name1 and Name2 are different
Lower	Lower limit of a 95% confidence interval for the covariance parameter
Upper	Upper limit of a 95% confidence interval for the covariance parameter

stats{r} is a dataset array containing statistics on covariance parameters for the  $r$ th grouping variable,  $r = 1, 2, \dots, R$ .  
stats{R+1} contains statistics on the residual standard deviation.  
The dataset array for the residual error has the fields Group, Name, Estimate, Lower, and Upper.

## Examples      **Two Random-Effects Terms for Intercept**

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` is the fixed-effects variable, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently. This model corresponds to

$$y_{ijk} = \beta_0 + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + b_{0k} S_k + b_{0jk} (S^* T)_{jk} + \varepsilon_{ijk},$$

where  $i = 1, 2, \dots, 60$  corresponds to the observations,  $j = 2, \dots, 5$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $(S^* T)_{jk}$  represents the  $j$ th tomato type nested in the  $k$ th soil type.  $I[T]_{ij}$  is the dummy variable representing the level  $j$  of the tomato type.

# LinearMixedModel.covarianceParameters

---

The random effects and observation error have the following prior distributions:  $b_{0k} \sim N(0, \sigma^2_S)$ ,  $b_{0jk} \sim N(0, \sigma^2_{S^*T})$ , and  $\varepsilon_{ijk} \sim N(0, \sigma^2)$ .

```
lme = fitlme(ds, 'Yield ~ Fertilizer + (1|Soil) + (1|Soil:Tomato)');
```

Compute the covariance parameter estimates (estimates of  $\sigma^2_S$  and  $\sigma^2_{S^*T}$ ) of the random-effects terms.

```
psi = covarianceParameters(lme)
```

```
psi =
```

```
    [ 4.4026e-17]
    [ 352.8481]
```

Compute the residual variance ( $\sigma^2$ ).

```
[~,mse] = covarianceParameters(lme)
```

```
mse =
```

```
    151.9007
```

## Potentially Correlated Random-Effects Terms

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

# LinearMixedModel.covarianceParameters

```
ds = dataset(InitialWeight,Program,Subject,Week,y);
ds.Subject = nominal(ds.Subject);
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

For 'reference' dummy variable coding, `fitlme` uses Program A as reference and creates the necessary dummy variables  $I[.]$ . This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 Week_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i + \beta_5 I[PD]_i \\ + b_{0m} + b_{1m} Week_{im} + \varepsilon_{im},$$

where  $i$  corresponds to the observation number,  $i = 1, 2, \dots, 120$ , and  $m$  corresponds to the subject number,  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[.]$  is a dummy variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing Program B.

The random effects and observation error have the following prior

distributions: 
$$\begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix} \right)$$
 and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
lme = fitlme(ds, 'y ~ InitialWeight + Program + (Week|Subject)');
```

Compute the estimates of covariance parameters for the random effects.

```
[psi,mse,stats] = covarianceParameters(lme)
```

```
psi =
```

```
[2x2 double]
```

# LinearMixedModel.covarianceParameters

---

```
mse =
```

```
0.0105
```

```
stats =
```

```
[3x7 classreg.regr.lmeutils.titledataset]  
[1x5 dataset]
```

mse is the estimated residual variance. It is the estimate for  $\sigma^2$ .

To see the covariance parameters estimates for the random-effects terms ( $\sigma^2_0$ ,  $\sigma^2_1$ , and  $\sigma^2_{0,1}$ ), index into `psi`.

```
psi{1}
```

```
ans =
```

```
0.0572    0.0490  
0.0490    0.0624
```

The estimate of the variance of the random effects term for the intercept,  $\sigma^2_0$ , is 0.0572. The estimate of the variance of the random effects term for week,  $\sigma^2_1$ , is 0.0624. The estimate for the covariance of the random effects terms for the intercept and week,  $\sigma_{0,1}$ , is 0.0490.

`stats` is a 2-by-1 cell array. The first cell of `stats` contains the confidence intervals for the standard deviation of the random effects and the correlation between the random effects for intercept and week. To display them, index into `stats`.

```
stats{1}
```

```
ans =
```

```
Covariance Type: FullCholesky
```



# LinearMixedModel.covarianceParameters

Group	Name1	Name2	Type
Subject	'(Intercept)'	'(Intercept)'	'std'
Subject	'Week'	'(Intercept)'	'corr'
Subject	'Week'	'Week'	'std'

The display shows the name of the grouping parameter (`Group`), the random-effects variables (`Name1`, `Name2`), the type of the covariance parameters (`Type`), the estimate (`Estimate`) for each parameter, and the 95% confidence intervals for the parameters (`Lower`, `Upper`). The estimates in this table are related to the estimates in `psi` as follows.

The standard deviation of the random-effects term for intercept is  $0.23927 = \sqrt{0.0527}$ . Likewise, the standard deviation of the random effects term for week is  $0.2497 = \sqrt{0.0624}$ . Finally, the correlation between the random-effects terms of intercept and week is  $0.81971 = 0.0490 / (0.23927 * 0.2497)$ .

Note that this display also shows which covariance pattern you use when fitting the model. In this case, the covariance pattern is `FullCholesky`. To change the covariance pattern for the random-effects terms, you must use the `'CovariancePattern'` name-value pair argument when fitting the model.

The second cell of `stats` includes similar statistics for the residual standard deviation. Display the contents of the second cell.

```
stats{2}
```

```
ans =
```

Group	Name	Estimate	Lower	Upper
Error	'Res Std'	0.10261	0.087882	0.11981

The estimate for residual standard deviation is the square root of `mse`,  $0.10261 = \sqrt{0.0105}$ .

## Two Grouping Variables

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and weight, a potentially correlated random effect for intercept and acceleration grouped by model year, and an independent random effect for weight, grouped by the origin of the car. This model corresponds to

$$MPG_{imk} = \beta_0 + \beta_1 Acc_i + \beta_2 Weight_i + b_{10m} + b_{11m} Acc_i + b_{21k} Weight_i + \varepsilon_{imk}, \\ m = 1, 2, \dots, 13, \quad k = 1, 2, \dots, 8,$$

where  $m$  represents the levels for the variable `Model_Year`, and  $k$  represents the levels for the variable `Origin`.  $MPG_{imk}$  is the miles per gallon for the  $i$ th observation,  $m$ th model year, and  $k$ th origin that correspond to the  $i$ th observation. The random-effects terms and the observation error have the following prior distributions:

$$b_{1m} = \begin{pmatrix} b_{10m} \\ b_{11m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_{10}^2 & \sigma_{10,11} \\ \sigma_{10,11} & \sigma_{11}^2 \end{pmatrix} \right),$$

$$b_{2k} \sim N(0, \sigma_2^2),$$

$$\varepsilon_{imk} \sim N(0, \sigma^2).$$

Here, the random-effects term  $b_{1m}$  represents the first random effect at level  $m$  of the first grouping variable. The random-effects term  $b_{10m}$  corresponds to the first random effects term (1), for the intercept (0), at the  $m$ th level ( $m$ ) of the first grouping variable. Likewise  $b_{11m}$  is the level  $m$  for the first predictor (1) in the first random-effects term (1).

Similarly,  $b_{2k}$  stands for the second random effects-term at level  $k$  of the second grouping variable.

# LinearMixedModel.covarianceParameters

---

$\sigma_{10}^2$  is the variance of the random-effects term for the intercept,  $\sigma_{11}^2$  is the variance of the random effects term for the predictor acceleration, and  $\sigma_{10,11}$  is the covariance of the random-effects terms for the intercept and the predictor acceleration.  $\sigma_2^2$  is the variance of the second random-effects term, and  $\sigma^2$  is the residual variance.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Weight];
Z = {[ones(406,1) Acceleration],[Weight]};
Model_Year = nominal(Model_Year);
Origin = nominal(Origin);
G = {Model_Year,Origin};
```

Fit the model using the design matrices.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Weight'],'RandomEffectPredictors',...
{'Intercept','Acceleration'},{'Weight'}},'RandomEffectGroups',{'Model_Year','Origin'});
```

Compute the estimates of covariance parameters for the random effects.

```
[psi,mse,stats] = covarianceParameters(lme)
```

```
psi =
```

```
    [2x2 double]
    [6.7989e-08]
```

```
mse =
```

```
    9.0755
```

```
stats =
```

```
    [3x7 classreg.regr.lmeutils.titledataset]
```

# LinearMixedModel.covarianceParameters

---

```
[1x7 classreg.regr.lmeutils.titledataset]
[1x5 dataset                               ]
```

The residual variance `mse` is 9.0755. `psi` is a 2-by-1 cell array, and `stats` is a 3-by-1 cell array. To see the contents, you must index into these cell arrays.

First, index into the first cell of `psi`.

```
psi{1}
```

```
ans =
```

```
    8.5160   -0.8387
   -0.8387    0.1087
```

The first cell of `psi` contains the covariance parameters for the correlated random effects for intercept  $\sigma_{10}^2$  as 8.5160, and for acceleration  $\sigma_{11}^2$  as 0.1087. The estimate for the covariance of the random-effects terms for the intercept and acceleration  $\sigma_{10,11}$  is -0.8387.

Now, index into the second cell of `psi`.

```
psi{2}
```

```
ans =
```

```
6.7989e-08
```

The second cell of `psi` contains the estimate for the variance of the random-effects term for weight  $\sigma_2^2$ .

Index into the first cell of `stats`.

```
stats{1}
```

```
ans =
```

# LinearMixedModel.covarianceParameters

Covariance Type: FullCholesky

Group	Name1	Name2	Type
Model_Year	'Intercept'	'Intercept'	'std'
Model_Year	'Acceleration'	'Intercept'	'corr'
Model_Year	'Acceleration'	'Acceleration'	'std'

This table shows the standard deviation estimates for the random-effects terms for intercept and acceleration. Note that the standard deviations estimates are the square roots of the diagonal elements in the first cell of `psi`. Specifically,  $2.9182 = \sqrt{8.5160}$  and  $0.32968 = \sqrt{0.1087}$ . The correlation is a function of the covariance of intercept and acceleration, and the standard deviations of intercept and acceleration. The covariance of intercept and acceleration is the off-diagonal value in the first cell of `psi`,  $-0.8387$ . So, the correlation is  $-0.8387 / (0.32968 * 2.92182) = -0.87$ .

The grouping variable for intercept and acceleration is `Model_Year`.

Index into the second cell of `stats`.

```
stats{2}
```

```
ans =
```

Covariance Type: FullCholesky

Group	Name1	Name2	Type	Estimate
Origin	'Weight'	'Weight'	'std'	0.00026075

The second cell of `stats` has the standard deviation estimate and the 95% confidence limits for the standard deviation of the random-effects term for `Weight`. The grouping variable is `Origin`.

Index into the third cell of `stats`.

```
stats{3}
```

# LinearMixedModel.covarianceParameters

---

```
ans =
```

Group	Name	Estimate	Lower	Upper
Error	'Res Std'	3.0126	2.8028	3.238

The third cell of `stats` contains the estimate for residual standard deviation and the 95% confidence limits. The estimate for residual standard deviation is the square root of `mse`,  $\text{sqrt}(9.0755) = 3.0126$ .

Construct 99% confidence intervals for the covariance parameters.

```
[~,~,stats] = covarianceParameters(lme, 'Alpha', 0.01);  
stats{1}
```

```
ans =
```

```
Covariance Type: FullCholesky
```

Group	Name1	Name2	Type
Model_Year	'Intercept'	'Intercept'	'std'
Model_Year	'Acceleration'	'Intercept'	'corr'
Model_Year	'Acceleration'	'Acceleration'	'std'

```
stats{2}
```

```
ans =
```

```
Covariance Type: FullCholesky
```

Group	Name1	Name2	Type	Estimate
Origin	'Weight'	'Weight'	'std'	0.00026075

```
stats{3}
```

```
ans =
```

# LinearMixedModel.covarianceParameters

---

Group	Name	Estimate	Lower	Upper
Error	'Res Std'	3.0126	2.74	3.3123

## See Also

[LinearMixedModel](#) | [compare](#) | [fixedEffects](#) | [randomEffects](#) |

# gmdistribution.CovType property

---

**Purpose** Type of covariance matrices

**Description** The string 'diagonal' if the covariance matrices are restricted to be diagonal; the string 'full' otherwise.



**Purpose**

Cox proportional hazards regression

**Syntax**

```
b = coxphfit(X,T)
b = coxphfit(X,T,Name,Value)
[b,logl,H,stats] = coxphfit( ___ )
```

**Description**

`b = coxphfit(X,T)` returns a  $p$ -by-1 vector, `b`, of coefficient estimates for a Cox proportional hazards regression of the observed responses in an  $n$ -by-1 vector, `T`, on the predictors in an  $n$ -by- $p$  matrix `X`.

The model does not include a constant term, and `X` cannot contain a column of 1s.

`b = coxphfit(X,T,Name,Value)` returns a vector of coefficient estimates, with additional options specified by one or more `Name,Value` pair arguments.

`[b,logl,H,stats] = coxphfit( ___ )` also returns the loglikelihood, `logl`, a structure, `stats`, that contains additional statistics, and a two-column matrix, `H`, that contains the `T` values in the first column and the estimated baseline cumulative hazard, in the second column. You can use any of the input arguments in the previous syntaxes.

**Input Arguments**

**X - Observations on predictor variables**

matrix

Observations on predictor variables, specified as an  $n$ -by- $p$  matrix of  $p$  predictors for each of  $n$  observations.

The model does not include a constant term, thus `X` cannot contain a column of 1s.

**Data Types**

single | double

**T - Time-to-event data**

vector

Time-to-event data, specified as an  $n$ -by-1 vector.

## Data Types

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

### Example:

`'baseline',0,'censoring',censoreddata,'frequency',freq` specifies that `coxphfit` calculates the baseline hazard rate relative to 0, considering the censoring information in the vector `censoreddata`, and the frequency of observations on **T** and **X** given in the vector `freq`.

### 'baseline' - X values at which to compute the baseline hazard

`mean(X)` (default) | scalar value

**X** values at which to compute the baseline hazard, specified as the comma-separated pair consisting of 'baseline' and a scalar value.

The default is `mean(X)`, so the hazard rate at **X** is  $h(t) \cdot \exp((X - \text{mean}(X)) \cdot b)$ . Enter 0 to compute the baseline relative to 0, so the hazard rate at **X** is  $h(t) \cdot \exp(X \cdot b)$ . Changing the baseline does not affect the coefficient estimates, but the hazard ratio changes.

**Example:** `'baseline',0`

## Data Types

single | double

### 'censoring' - Indicator for censoring

array of 0s (default) | array of 0s and 1s

Indicator for censoring, specified as the comma-separated pair consisting of 'censoring' and a Boolean array of the same size as **T**.

Use 1 for observations that are right censored and 0 for observations that are fully observed. The default is all observations are fully observed.

**Example:** 'censoring',cens

### Data Types

logical

### 'frequency' - Frequency of observations

array of 1s (default) | vector of nonnegative integer counts

Frequency of observations, specified as the comma-separated pair consisting of 'frequency' and an array that is the same size as T containing nonnegative integer counts.

The  $j^{\text{th}}$  element of this vector gives the number of times the method observes the  $j^{\text{th}}$  element of T and the  $j^{\text{th}}$  row of X. The default is one observation per row of X and T.

**Example:** 'frequency',freq

### Data Types

single | double

### 'init' - Initial values for estimated coefficients

vector

Initial values for estimated coefficients, specified as the comma-separated pair consisting of 'init' and a vector containing the coefficient initial values.

**Example:** 'init',initcoef

### Data Types

single | double

### 'options' - Algorithm control parameters

structure

Algorithm control parameters for the iterative algorithm used to estimate b, specified as the comma-separated pair consisting of

'options' and a structure. A call to `statset` creates this argument. For parameter names and default values, type `statset('coxphfit')`. You can set the options under a new name and use that in the name-value pair argument.

**Example:** `'options',statset('coxphfit')`

## Data Types

char

## Output Arguments

### **b** - Coefficient estimates

vector

Coefficient estimates for a Cox proportional hazards regression, returned as a  $p$ -by-1 vector.

### **logl** - Loglikelihood

scalar

Loglikelihood of the fitted model, returned as a scalar.

You can use log likelihood values to compare different models and assess the significance of effects of terms in the model.

### **H** - Estimated baseline cumulative hazard

two-column matrix

Estimated baseline cumulative hazard rate evaluated at T values, returned as a two-column matrix. The first column of the matrix contains T values, and the second column contains cumulative hazard rate estimates.

### **stats** - Coefficient statistics

structure

Coefficient statistics, returned as a structure that contains the following fields.

beta	Coefficient estimates (same as <b>b</b> )
se	Standard errors of coefficient estimates, <b>b</b>
z	<i>z</i> -statistics for <b>b</b> (that is, <b>b</b> divided by standard error)
p	<i>p</i> -values for <b>b</b>
covb	Estimated covariance matrix for <b>b</b>

## Examples

### Lifetime of Light Bulbs

Navigate to a folder containing sample data.

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

Load the sample data.

```
load lightbulb
```

The first column of the light bulb data has the lifetime (in hours) of two different types of bulbs. The second column has the binary variable indicating whether the bulb is fluorescent or incandescent. 0 indicates that the bulb is incandescent, and 1 indicates that it is fluorescent. The third column contains the censorship information, where 0 indicates the bulb was observed until failure, and 1 indicates the bulb was censored.

Fit a Cox proportional hazards model for the lifetime of the light bulbs, also accounting for censoring. The predictor variable is the type of bulb.

```
b = coxphfit(lightbulb(:,2),lightbulb(:,1),...
'censoring',lightbulb(:,3))
```

```
b =
```

```
4.7262
```

The estimate of the hazard ratio is  $\exp(b) = 112.8646$ . This means that the hazard for the incandescent bulbs is 112.86 times the hazard for the fluorescent bulbs.

## Change the Algorithm Parameters

Navigate to a folder containing sample data.

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

Load the sample data.

```
load lightbulb
```

The first column of the data has the lifetime (in hours) of two types of bulbs. The second column has the binary variable indicating whether the bulb is fluorescent or incandescent. 0 indicates that the bulb is incandescent, and 1 indicates that it is fluorescent. The third column contains the censorship information, where 0 indicates the bulb is observed until failure, and 1 indicates the item (bulb) is censored.

Fit a Cox proportional hazards model, also accounting for censoring. The predictor variable is the type of bulb.

```
b = coxphfit(lightbulb(:,2),lightbulb(:,1),...
'censoring',lightbulb(:,3))
```

```
b =
```

```
4.7262
```

Display the default control parameters for the algorithm `coxphfit` uses to estimate the coefficients.

```
statset('coxphfit')
```

```
ans =
```

```
Display: 'off'
```

```

MaxFunEvals: 200
  MaxIter: 100
    TolBnd: 1.0000e-06
    TolFun: 1.0000e-08
  TolTypeFun: []
    TolX: 1.0000e-08
  TolTypeX: []
  GradObj: []
  Jacobian: []
  DerivStep: []
  FunValCheck: []
  Robust: []
  RobustWgtFun: []
  WgtFun: []
  Tune: []
  UseParallel: []
  UseSubstreams: []
  Streams: {}
  OutputFcn: []

```

Save the options under a different name and change how the results will be displayed and the maximum number of iterations, `Display` and `MaxIter`.

```

coxphopt = statset('coxphfit');
coxphopt.Display = 'final';
coxphopt.MaxIter = 50;

```

Run `coxphfit` with the new algorithm parameters.

```

b = coxphfit(lightbulb(:,2),lightbulb(:,1),...
'censoring',lightbulb(:,3),'options',coxphopt)

```

Successful convergence: Norm of gradient less than `OPTIONS.TolFun`

b =

```

4.7262

```

coxphfit displays a report on the final iteration. Changing the maximum number of iterations did not affect the coefficient estimate.

## Fit and Compare Cox and Weibull Survivor Functions

Generate Weibull data depending on predictor X.

```
rng('default') % for reproducibility
X = 4*rand(100,1);
A = 50*exp(-0.5*X);
B = 2;
y = wblrnd(A,B);
```

The response values are generated from a Weibull distribution with a shape parameter depending on the predictor variable X and a scale parameter of 2.

Fit a Cox proportional hazards model.

```
[b,logL,H,stats] = coxphfit(X,y);
[b logL]
```

```
ans =

    0.9409 -331.1479
```

The coefficient estimate is 0.9409 and the log likelihood value is -331.1479.

Request the model statistics.

```
stats

stats =

    covb: 0.0158
    beta: 0.9409
    se: 0.1256
    z: 7.4889
```

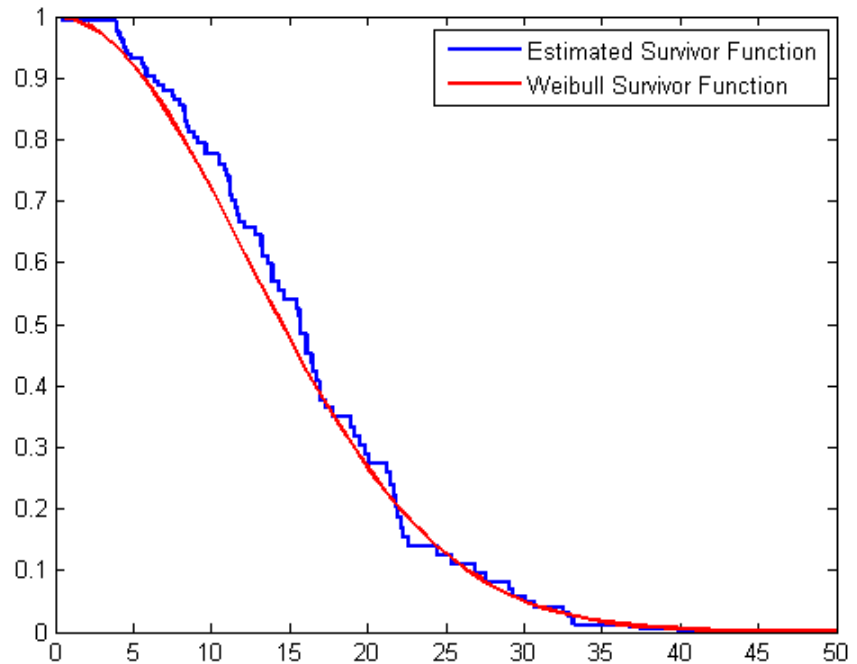


p: 6.9462e-14

The covariance matrix of the coefficient estimates, `covb`, contains only one value, which is equal to the variance of the coefficient estimate in this example. The coefficient estimate, `beta`, is the same as `b` and is equal to 0.9409. The standard error of the coefficient estimate, `se`, is 0.1256, which is the square root of the variance 0.0158. The z-statistic, `z`, is  $\text{beta}/\text{se} = 0.9409/0.1256 = 7.4880$ . The p-value, `p`, indicates that the effect of `X` is significant.

Plot the Cox estimate of the baseline survivor function together with the known Weibull function.

```
stairs(H(:,1),exp(-H(:,2)),'LineWidth',2)
xx = linspace(0,100);
line(xx,1-wblcdf(xx,50*exp(-0.5*mean(X)),B),'color','r','LineWidth',2)
xlim([0,50])
legend('Estimated Survivor Function','Weibull Survivor Function')
```



The fitted model gives a close estimate to the survivor function of the actual distribution.

## Definitions

### Cox Proportional Hazards Regression

Cox proportional hazards regression is a semiparametric method for adjusting survival rate estimates to remove the effect of confounding variables and to quantify the effect of predictor variables. The method represents the effects of explanatory and confounding variables as a multiplier of a common baseline hazard function,  $h_0(t)$ .

For a baseline relative to 0, this model corresponds to

$$h_X(t) = h_0(t)e^{\sum_i X_i b_i},$$

where  $h_X(t)$  is the hazard rate at X and  $h_0(t)$  is the baseline hazard rate function. The baseline hazard function is the nonparametric part of the Cox proportional hazards regression function, whereas the impact of the predictor variables is a loglinear regression. The assumption is that the baseline hazard function depends on time,  $t$ , but the predictor variables do not depend on time.

## References

- [1] Cox, D.R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [3] Kleinbaum, D. G., and M. Klein. *Survival Analysis*. Statistics for Biology and Health. 2nd edition. Springer, 2005.

## See Also

ecdf | statset | wblfit

## Related Examples

- “Hazard and Survivor Functions for Different Groups” on page 12-19
- “Survivor Functions for Two Groups” on page 12-26
- “Cox Proportional Hazards Model for Censored Data” on page 12-34

## Concepts

- “What Is Survival Analysis?” on page 12-2
- “Kaplan-Meier Method” on page 12-11
- “Cox Proportional Hazards Regression” on page 12-31

# NaiveBayes.Prior property

---

**Purpose**            Class priors

**Description**        The Prior property is a vector of length NClasses containing the class priors. The priors for empty classes are zero.

<b>Purpose</b>	Create object to use in <i>k</i> -nearest neighbors search
<b>Syntax</b>	<pre>NS = createns(X) NS = createns(X, 'Name', Value)</pre>
<b>Description</b>	<p>NS = createns(X) uses the data observations in an <i>m</i><i>x</i>-by-<i>n</i> matrix X to create an object NS. Rows of X correspond to observations and columns correspond to variables. NS is either an ExhaustiveSearcher or a KDTreeSearcher object which you can use to find nearest neighbors in X for desired query points. When NS is an ExhaustiveSearcher object, knnsearch uses the exhaustive search algorithm to find nearest neighbors. When NS is a KDTreeSearcher, createns creates and saves a <i>kd</i>-tree based on X in NS. knnsearch uses the <i>kd</i>-tree to find nearest neighbors. For information on these search methods, see “<i>k</i>-Nearest Neighbor Search and Radius Search” on page 16-12.</p> <p>NS = createns(X, 'Name', Value) accepts one or more optional name/value pairs. Specify <i>Name</i> inside single quotes. Specify NSMethod to determine which type of object to create. The object's properties save the information when you specify other arguments. For more information on the objects' properties, see ExhaustiveSearcher and KDTreeSearcher.</p>
<b>Input Arguments</b>	<b>Name-Value Pair Arguments</b>  <b>'NSMethod'</b>  Nearest neighbors search method, used to define the type of object created. Value is either: <ul style="list-style-type: none"><li>• 'kdtree' — Create a KDTreeSearcher object. If you do not specify NSMethod, this is the default value when the number of columns of X is less than 10, X is not sparse, and the distance measure is one of the following measures:<ul style="list-style-type: none"><li>▪ 'euclidean' (default)</li><li>▪ 'cityblock'</li></ul></li></ul>

- 'minkowski'
- 'chebychev'
- 'exhaustive' — Create an ExhaustiveSearcher object. If you do not specify NSMethod, this is the default value when the default criteria for 'kdtree' do not apply.

## 'Distance'

A string or a function handle specifying the default distance metric used when you call the `knnsearch` method to find nearest neighbors for future query points. If you specify a distance metric but not an NSMethod, this input determines the type of object `createns` creates, according to the default values described in NSMethod.

For both `KDTreeSearcher` and `ExhaustiveSearcher` objects, the following options apply:

- 'euclidean' (default) — Euclidean distance.
- 'cityblock' — City block distance.
- 'chebychev' — Chebychev distance (maximum coordinate difference).
- 'minkowski' — Minkowski distance.

The following options apply only to `ExhaustiveSearcher` objects:

- 'seuclidean' — Standardized Euclidean distance. Each coordinate difference between rows in  $X$  and the query matrix is scaled by dividing by the corresponding element of the standard deviation computed from  $X$ ,  $S = \text{nanstd}(X)$ . To specify another value for  $S$ , use the `Scale` argument.
- 'mahalanobis' — Mahalanobis distance, which is computed using a positive definite covariance matrix  $C$ . The default value of  $C$  is the sample covariance matrix of  $X$ , as computed by `nancov(X)`. To change the value of  $C$ , use the `Cov` parameter.
- 'cosine' — One minus the cosine of the included angle between observations (treated as vectors).

- 'correlation' — One minus the sample linear correlation between observations (treated as sequences of values).
- 'spearman' — One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
- 'hamming' — Hamming distance, which is percentage of coordinates that differ.
- 'jaccard' — One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
- custom distance function — A distance function specified using @ (for example, @distfun). A distance function must be of the form function D2 = distfun(ZI, ZJ), taking as arguments a 1-by- $n$  vector ZI containing a single row from X or from the query points Y, and an  $m$ -by- $n$  matrix ZJ containing multiple rows of X or Y, and returning an  $m$ -by-1 vector of distances  $d2$ , whose  $j$ th element is the distance between the observations ZI and ZJ( $j$ , :).

**'p'**

A positive scalar,  $p$ , indicating the exponent of the Minkowski distance. This parameter is only valid when Distance is 'minkowski'. Default is 2.

**'Cov'**

A positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This parameter is only valid when Distance is 'mahalanobis'. Default is nancov(X).

**'Scale'**

A vector S with the length equal to the number of columns in X. Each coordinate of X and each query point is scaled by the corresponding element of S when computing the standardized Euclidean distance. This parameter is only valid when Distance is 'seuclidean'. Default is nanstd(X).

## 'BucketSize'

A positive integer, indicating the maximum number of data points in each leaf node of the *kd*-tree. This argument is only meaningful when using the *kd*-tree search method. Default is 50.

## Examples

Create a *kd*-tree with a Minkowski distance metric and a P value of 5:

```
load fisheriris
x = meas(:,3:4);
% Since x has only two columns and the Distance is Minkowski,
% createns creates a KDTreeSearcher object by default:
knnobj = createns(x, 'Distance', 'minkowski', 'P', 5)

knnobj =

KDTreeSearcher

Properties:
    BucketSize: 50
              X: [150x2 double]
              Distance: 'minkowski'
              DistParameter: 5
```

## See Also

[ExhaustiveSearcher.knnsearch](#) | [KDTreeSearcher.knnsearch](#) | [ExhaustiveSearcher](#) | [KDTreeSearcher](#)

## How To

- “*k*-Nearest Neighbor Search and Radius Search” on page 16-12



**Purpose**

Cross-tabulation

**Syntax**

```
table = crosstab(x1,x2)
table = crosstab(x1,x2,...,xn)
[table,chi2,p] = crosstab(x1,...,xn)
[table,chi2,p,labels] = crosstab(x1,...,xn)
```

**Description**

`table = crosstab(x1,x2)` returns a cross-tabulation table of two vectors of the same length `x1` and `x2`. `table` is *m*-by-*n*, where *m* is the number of distinct values in `x1` and *n* is the number of distinct values in `x2`.

`x1` and `x2` are grouping variables, as described in “Grouping Variables” on page 2-51. `crosstab` uses `grp2idx` to assign a positive integer to each distinct value. `table(i,j)` is a count of indices where `grp2idx(x1)` is *i* and `grp2idx(x2)` is *j*. The numerical order of `grp2idx(x1)` and `grp2idx(x2)` order rows and columns of `table`, respectively.

`table = crosstab(x1,x2,...,xn)` returns a multi-dimensional table where `table(i,j,...,n)` is a count of indices where `grp2idx(x1)` is *i*, `grp2idx(x2)` is *j*, `grp2idx(x3)` is *k*, and so on.

`[table,chi2,p] = crosstab(x1,...,xn)` also returns the chi-square statistic `chi2` and its *p* value `p` for a test that `table` is independent in each dimension. The null hypothesis is that the proportion in any entry of `table` is the product of the proportions in each dimension.

`[table,chi2,p,labels] = crosstab(x1,...,xn)` also returns a cell array `labels` with one column for each input argument. The entries in the first column are labels for the rows of `table`, the entries in the second column are labels for the columns, and so on, for a multi-dimensional table.

**Examples****Cross-Tabulate Two Data Vectors**

This example shows how to cross-tabulate two data vectors.

Create two sample data vectors, containing three and four distinct values, respectively.

```
x = [1 1 2 3 1];  
y = [1 2 5 3 1];
```

Cross-tabulate x and y.

```
table = crosstab(x,y)
```

```
table =  
  
     2     1     0     0  
     0     0     0     1  
     0     0     1     0
```

## Cross-Tabulate Independent Data Vectors

This example shows how to cross-tabulate two independent data vectors.

Generate two independent vectors, each containing 50 discrete uniform random numbers in the range 1:3.

```
rng('default'); % For reproducibility  
x1 = unidrnd(3,50,1);  
x2 = unidrnd(3,50,1);
```

Cross-tabulate x1 and x2.

```
[table,chi2,p] = crosstab(x1,x2)
```

```
table =  
  
     1     6     7  
     5     5     2  
    11     7     6
```

```
chi2 =  
      7.5449
```

```
p =  
    0.1097
```

At the 95% confidence level, the  $p$  value fails to reject the null hypothesis that `table` is independent in each dimension.

### **Cross-Tabulate Grouped Data**

This example shows how to cross-tabulate grouped sample data.

Load the sample data, which contains measurements of large model cars during the years 1970-1982.

```
load carbig
```

Cross-tabulate the data of four-cylinder cars (`cy14`) based on model year (`when`) and country of origin (`org`).

```
[table,chi2,p,labels] = crosstab(cy14,when,org)
```

```
table(:, :, 1) =  
      82   75   25  
      12   22   38
```

```
table(:, :, 2) =  
      0    4    3  
      23  26  17
```

# crosstab

---

```
table(:, :, 3) =  
  
      3      3      4  
     12     25     32  
  
chi2 =  
  
     207.7689  
  
p =  
  
     8.0973e-38  
  
labels =  
  
      'Other'      'Early'      'USA'  
      'Four'      'Mid'      'Europe'  
      []      'Late'      'Japan'
```

The results in `table` and `labels` together show that the number of four-cylinder cars made in the USA during the late period of the data is located in `table(2,3,1)`, and equals 38 cars.

## See Also

`grp2idx` | `tabulate`

## How To

- “Grouping Variables” on page 2-51

**Purpose**

Loss estimate using cross validation

**Syntax**

```
vals = crossval(fun,X)
vals = crossval(fun,X,Y,...)
mse = crossval('mse',X,y,'Predfun',predfun)
mcr = crossval('mcr',X,y,'Predfun',predfun)
val = crossval(criterion,X1,X2,...,y,'Predfun',predfun)
vals = crossval(...,'name',value)
```

**Description**

`vals = crossval(fun,X)` performs 10-fold cross validation for the function `fun`, applied to the data in `X`.

`fun` is a function handle to a function with two inputs, the training subset of `X`, `XTRAIN`, and the test subset of `X`, `XTEST`, as follows:

```
testval = fun(XTRAIN,XTEST)
```

Each time it is called, `fun` should use `XTRAIN` to fit a model, then return some criterion `testval` computed on `XTEST` using that fitted model.

`X` can be a column vector or a matrix. Rows of `X` correspond to observations; columns correspond to variables or features. Each row of `vals` contains the result of applying `fun` to one test set. If `testval` is a non-scalar value, `crossval` converts it to a row vector using linear indexing and stored in one row of `vals`.

`vals = crossval(fun,X,Y,...)` is used when data are stored in separate variables `X`, `Y`, ... . All variables (column vectors, matrices, or arrays) must have the same number of rows. `fun` is called with the training subsets of `X`, `Y`, ... , followed by the test subsets of `X`, `Y`, ... , as follows:

```
testvals = fun(XTRAIN,YTRAIN,...,XTEST,YTEST,...)
```

`mse = crossval('mse',X,y,'Predfun',predfun)` returns `mse`, a scalar containing a 10-fold cross validation estimate of mean-squared error for the function `predfun`. `X` can be a column vector, matrix, or array of predictors. `y` is a column vector of response values. `X` and `y` must have the same number of rows.

`predfun` is a function handle called with the training subset of  $X$ , the training subset of  $y$ , and the test subset of  $X$  as follows:

```
yfit = predfun(XTRAIN,ytrain,XTEST)
```

Each time it is called, `predfun` should use `XTRAIN` and `ytrain` to fit a regression model and then return fitted values in a column vector `yfit`. Each row of `yfit` contains the predicted values for the corresponding row of `XTEST`. `crossval` computes the squared errors between `yfit` and the corresponding response test set, and returns the overall mean across all test sets.

`mcr = crossval('mcr',X,y,'Predfun',predfun)` returns `mcr`, a scalar containing a 10-fold cross validation estimate of misclassification rate (the proportion of misclassified samples) for the function `predfun`. The matrix  $X$  contains predictor values and the vector  $y$  contains class labels. `predfun` should use `XTRAIN` and `YTRAIN` to fit a classification model and return `yfit` as the predicted class labels for `XTEST`. `crossval` computes the number of misclassifications between `yfit` and the corresponding response test set, and returns the overall misclassification rate across all test sets.

`val = crossval(criterion,X1,X2,...,y,'Predfun',predfun)`, where *criterion* is 'mse' or 'mcr', returns a cross validation estimate of mean-squared error (for a regression model) or misclassification rate (for a classification model) with predictor values in  $X1$ ,  $X2$ , ... and, respectively, response values or class labels in  $y$ .  $X1$ ,  $X2$ , ... and  $y$  must have the same number of rows. `predfun` is a function handle called with the training subsets of  $X1$ ,  $X2$ , ..., the training subset of  $y$ , and the test subsets of  $X1$ ,  $X2$ , ..., as follows:

```
yfit=predfun(X1TRAIN,X2TRAIN,...,ytrain,X1TEST,X2TEST,...)
```

`yfit` should be a column vector containing the fitted values.

`vals = crossval(...,'name',value)` specifies one or more optional parameter name/value pairs from the following table. Specify *name* inside single quotes.

Name	Value
holdout	A scalar specifying the ratio or the number of observations $p$ for holdout cross validation. When $0 < p < 1$ , approximately $p \cdot n$ observations for the test set are randomly selected. When $p$ is an integer, $p$ observations for the test set are randomly selected.
kfold	A scalar specifying the number of folds $k$ for $k$ -fold cross validation.
leaveout	Specifies leave-one-out cross validation. The value must be 1.
mcreps	A positive integer specifying the number of Monte-Carlo repetitions for validation. If the first input of <code>crossval</code> is 'mse' or 'mcr', <code>crossval</code> returns the mean of mean-squared error or misclassification rate across all of the Monte-Carlo repetitions. Otherwise, <code>crossval</code> concatenates the values <code>vals</code> from all of the Monte-Carlo repetitions along the first dimension.
partition	An object <code>c</code> of the <code>cvpartition</code> class, specifying the cross validation type and partition.
stratify	A column vector <code>group</code> specifying groups for stratification. Both training and test sets have roughly the same class proportions as in <code>group</code> . NaNs or empty strings in <code>group</code> are treated as missing values, and the corresponding rows of the data are ignored.
options	A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the <code>options</code> structure with <code>statset</code> . Option fields:

Name	Value
	<ul style="list-style-type: none"><li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li><li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li><li>• <code>Streams</code> — A <code>RandStream</code> object or cell array consisting of one such object. If you do not specify <code>Streams</code>, <code>crossval</code> uses the default stream.</li></ul>

Only one of `kfold`, `holdout`, `leaveout`, or `partition` can be specified, and `partition` cannot be specified with `stratify`. If both `partition` and `mcreps` are specified, the first Monte-Carlo repetition uses the partition information in the `cvpartition` object, and the `repartition` method is called to generate new partitions for each of the remaining repetitions. If no cross validation type is specified, the default is 10-fold cross validation.

---

**Note** When using cross validation with classification algorithms, stratification is preferred. Otherwise, some test sets may not include observations from all classes.

---

## Examples

### Example 1

Compute mean-squared error for regression using 10-fold cross validation:

```
load('fisheriris');  
y = meas(:,1);  
X = [ones(size(y,1),1),meas(:,2:4)];
```



```
regf=@(XTRAIN,ytrain,XTEST)(XTEST*regress(ytrain,XTRAIN));  
  
cvMse = crossval('mse',X,y,'predfun',regf)  
cvMse =  
    0.1015
```

### Example 2

Compute misclassification rate using stratified 10-fold cross validation:

```
load('fisheriris');  
y = species;  
X = meas;  
cp = cvpartition(y,'k',10); % Stratified cross-validation  
  
classf = @(XTRAIN, ytrain,XTEST)(classify(XTEST,XTRAIN,...  
ytrain));  
  
cvMCR = crossval('mcr',X,y,'predfun',classf,'partition',cp)  
cvMCR =  
    0.0200
```

### Example 3

Compute the confusion matrix using stratified 10-fold cross validation:

```
load('fisheriris');  
y = species;  
X = meas;  
order = unique(y); % Order of the group labels  
cp = cvpartition(y,'k',10); % Stratified cross-validation  
  
f = @(xtr,ytr,xte,yte)confusionmat(yte,...  
classify(xte,xtr,ytr),'order',order);  
  
cfMat = crossval(f,X,y,'partition',cp);  
cfMat = reshape(sum(cfMat),3,3)  
cfMat =
```

50	0	0
0	48	2
0	1	49

cfMat is the summation of 10 confusion matrices from 10 test sets.

## References

[1] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. New York: Springer, 2001.

## See Also

cvpartition

## How To

• “Grouping Variables” on page 2-51

**Purpose** Cross-validated discriminant analysis classifier

**Syntax**  
`cvmodel = crossval(obj)`  
`cvmodel = crossval(obj,Name,Value)`

**Description**  
`cvmodel = crossval(obj)` creates a partitioned model from `obj`, a fitted discriminant analysis classifier. By default, `crossval` uses 10-fold cross validation on the training data to create `cvmodel`.  
`cvmodel = crossval(obj,Name,Value)` creates a partitioned model with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- Assess the predictive performance of `obj` on cross-validated data using the “kfold” methods and properties of `cvmodel`, such as `kfoldLoss`.

**Input Arguments**

**obj**  
Discriminant analysis classifier, produced using `ClassificationDiscriminant.fit`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### 'cvpartition'

Object of class `cvpartition`, created by the `cvpartition` function. `crossval` splits the data into subsets with `cvpartition`.

Use only one of these options at a time: `'cvpartition'`, `'holdout'`, `'kfold'`, or `'leaveout'`.

**Default:** []

# ClassificationDiscriminant.crossval

---

## 'holdout'

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. Use only one of these options at a time: 'cvpartition', 'holdout', 'kfold', or 'leaveout'.

## 'kfold'

Number of folds to use in a cross-validated classifier, a positive integer.

Use only one of these options at a time: 'cvpartition', 'holdout', 'kfold', or 'leaveout'.

**Default:** 10

## 'leaveout'

Set to 'on' for leave-one-out cross validation.

Use only one of these options at a time: 'cvpartition', 'holdout', 'kfold', or 'leaveout'.

## Examples

Create a classification model for the Fisher iris data, and then create a cross-validation model. Evaluate the quality the model using `kfoldLoss`.

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
cvmodel = crossval(obj);
L = kfoldLoss(cvmodel)

L =
    0.0200
```

## Alternatives

You can create a cross-validation classifier directly from the data, instead of creating a discriminant analysis classifier followed by a cross-validation classifier. To do so, include one of these options in

ClassificationDiscriminant.fit: 'crossval', 'cvpartition',  
'holdout', 'kfold', or 'leaveout'.

## See Also

ClassificationDiscriminant.fit | crossval | kfoldEdge |  
kfoldfun | kfoldLoss | kfoldMargin | kfoldPredict

## How To

- “Discriminant Analysis” on page 15-3

# ClassificationEnsemble.crossval

---

**Purpose** Cross validate ensemble

**Syntax**  
`cvens = crossval(ens)`  
`cvens = crossval(ens,Name,Value)`

**Description**  
`cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a classification ensemble. Default is 10-fold cross validation.  
`cvens = crossval(ens,Name,Value)` creates a cross-validated ensemble with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

## Input Arguments

**ens**  
A classification ensemble created with `fitensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1, . . . ,NameN,ValueN`.

#### 'cvpartition'

A partition of class `cvpartition`. Sets the partition for cross validation.

Use no more than one of the name-value pairs `cvpartition`, `holdout`, `kfold`, or `leaveout`.

#### 'holdout'

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

## 'kfold'

Number of folds for cross validation, a numeric positive scalar.

Use no more than one of the name-value pairs 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## 'leaveout'

If 'on', use leave-one-out cross validation.

Use no more than one of the name-value pairs 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## 'nprint'

Printout frequency, a positive integer scalar. Use this parameter to observe the training of cross-validation folds.

**Default:** 'off', meaning no printout

## Output Arguments

### **cvens**

A cross-validated classification ensemble of class ClassificationPartitionedEnsemble.

## Alternatives

You can create a cross-validation ensemble directly from the data, instead of creating an ensemble followed by a cross-validation ensemble. To do so, include one of these five options in fitensemble: 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## Examples

Create a cross-validated classification model for the Fisher iris data, and assess its quality using the kfoldLoss method.

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
cvens = crossval(ens);
L = kfoldLoss(cvens)
```

```
L =
    0.0467
```

# ClassificationEnsemble.crossval

---

## See Also

ClassificationPartitionedEnsemble | cvpartition



<b>Purpose</b>	Cross-validated $k$ -nearest neighbor classifier
<b>Syntax</b>	<pre>cvmodel = crossval mdl cvmodel = crossval mdl, Name, Value</pre>
<b>Description</b>	<p><code>cvmodel = crossval(mdl)</code> creates a partitioned model from <code>mdl</code>, a fitted KNN classification model. By default, <code>crossval</code> uses 10-fold cross validation on the training data to create <code>cvmodel</code>.</p> <p><code>cvmodel = crossval(mdl, Name, Value)</code> creates a partitioned model with additional options specified by one or more <code>Name, Value</code> pair arguments.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>Assess the predictive performance of <code>mdl</code> on cross-validated data using the “kfold” methods and properties of <code>cvmodel</code>, such as <code>kfoldLoss</code>.</li></ul>
<b>Input Arguments</b>	<p><b>mdl</b></p> <p><math>k</math>-nearest neighbor classifier, created by <code>ClassificationKNN.fit</code>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name, Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1, Value1, ..., NameN, ValueN</code>.</p> <p><b>'cvpartition'</b></p> <p>Object of class <code>cvpartition</code>, created by the <code>cvpartition</code> function. <code>crossval</code> splits the data into subsets with <code>cvpartition</code>.</p> <p>Use only one of these four options at a time: <code>'kfold'</code>, <code>'holdout'</code>, <code>'leaveout'</code>, or <code>'cvpartition'</code>.</p> <p><b>'holdout'</b></p>

# ClassificationKNN.crossval

---

Holdout validation tests the specified fraction of the data, and uses the remaining data for training. Specify a numeric scalar from 0 to 1. Use only one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## 'kfold'

Number of folds to use in a cross-validated tree, a positive integer.

Use only one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

**Default:** 10

## 'leaveout'

Set to 'on' for leave-one-out cross validation.

Use only one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## Output Arguments

### cvmodel

Partitioned model of class ClassificationPartitionedModel.

## Examples

### Cross-Validated $K$ -Nearest Neighbor Model

Construct a cross-validated  $k$ -nearest neighbor model, and assess classification performance using the model.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y);
```

Construct a cross-validated classifier.

```
cvmdl = crossval(md1)

cvmdl =

classreg.learning.partition.ClassificationPartitionedModel:
  CrossValidatedModel: 'KNN'
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}
  CategoricalPredictors: []
  ResponseName: 'Y'
  NObservations: 150
  KFold: 10
  Partition: [1x1 cvpartition]
  ClassNames: {'setosa' 'versicolor' 'virginica'}
  ScoreTransform: 'none'
```

Find the cross-validated loss of the classifier.

```
cvmdlloss = kfoldLoss(cvmdl)

cvmdlloss =

    0.0400
```

The cross-validated loss is less than 5%. You can expect md1 to have a similar error rate.

## Alternatives

You can create a cross-validated model directly from the data, instead of creating a model followed by a cross-validated model. To do so, include one of these options in `ClassificationKNN.fit`: 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## See Also

[ClassificationKNN](#) | [ClassificationPartitionedModel](#) | [crossvalkfoldEdge](#) | [kfoldfun](#) | [kfoldLoss](#) | [kfoldMargin](#) | [kfoldPredict](#) |

## Related Examples

- “Examine the Quality of a KNN Classifier” on page 16-27
- “Modify a KNN Classifier” on page 16-29

# ClassificationKNN.crossval

---

## **Concepts**

- “Classification Using Nearest Neighbors” on page 16-9

<b>Purpose</b>	Cross-validated decision tree
<b>Syntax</b>	<pre>cvmodel = crossval(model) cvmodel = crossval(model,Name,Value)</pre>
<b>Description</b>	<p><code>cvmodel = crossval(model)</code> creates a partitioned model from <code>model</code>, a fitted classification tree. By default, <code>crossval</code> uses 10-fold cross validation on the training data to create <code>cvmodel</code>.</p> <p><code>cvmodel = crossval(model,Name,Value)</code> creates a partitioned model with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>Assess the predictive performance of <code>model</code> on cross-validated data using the “kfold” methods and properties of <code>cvmodel</code>, such as <code>kfoldLoss</code>.</li></ul>
<b>Input Arguments</b>	<p><b>model</b></p> <p>A classification model, produced using <code>ClassificationTree.fit</code>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>'cvpartition'</b></p> <p>Object of class <code>cvpartition</code>, created by the <code>cvpartition</code> function. <code>crossval</code> splits the data into subsets with <code>cvpartition</code>.</p> <p>Use only one of these four options at a time: <code>'kfold'</code>, <code>'holdout'</code>, <code>'leaveout'</code>, or <code>'cvpartition'</code>.</p> <p><b>'holdout'</b></p>

# ClassificationTree.crossval

---

Holdout validation tests the specified fraction of the data, and uses the remaining data for training. Specify a numeric scalar from 0 to 1. Use only one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## 'kfold'

Number of folds to use in a cross-validated tree, a positive integer.

Use only one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

**Default:** 10

## 'leaveout'

Set to 'on' for leave-one-out cross validation.

Use only one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## Output Arguments

### cvmodel

Partitioned model of class ClassificationPartitionedModel.

## Examples

Create a classification model for the ionosphere data, then create a cross-validation model. Evaluate the quality the model using kfoldLoss.

```
load ionosphere
tree = ClassificationTree.fit(X,Y);
cvmodel = crossval(tree);
L = kfoldLoss(cvmodel)
```

```
L =
    0.1168
```

## Alternatives

You can create a cross-validation tree directly from the data, instead of creating a decision tree followed by a cross-validation tree. To do

so, include one of these five options in `ClassificationTree.fit`:  
'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## See Also

`ClassificationTree.fit` | `crossval`

# RegressionEnsemble.crossval

---

**Purpose** Cross validate ensemble

**Syntax**  
`cvens = crossval(ens)`  
`cvens = crossval(ens,Name,Value)`

**Description** `cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a regression ensemble. Default is 10-fold cross validation.

`cvens = crossval(ens,Name,Value)` creates a cross-validated ensemble with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

**ens**  
A regression ensemble created with `fitensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

#### 'cvpartition'

A partition of class `cvpartition`. Sets the partition for cross validation.

Use no more than one of the name-value pairs `cvpartition`, `holdout`, `kfold`, and `leaveout`.

#### 'holdout'

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.



## 'kfold'

Number of folds for cross validation, a numeric positive scalar.

Use no more than one of the name-value pairs 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## 'leaveout'

If 'on', use leave-one-out cross-validation.

Use no more than one of the name-value pairs 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## 'nprint'

Printout frequency, a positive integer scalar. Use this parameter to observe the training of cross-validation folds.

**Default:** 'off', meaning no printout

## Output Arguments

### **cvens**

A cross-validated classification ensemble of class RegressionPartitionedEnsemble.

## Alternatives

You can create a cross-validation ensemble directly from the data, instead of creating an ensemble followed by a cross-validation ensemble. To do so, include one of these five options in `fitensemble`: 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## Examples

Create a cross-validated classification model for the `carsmall` data, and assess its quality using the `kfoldLoss` method:

```
X = [Acceleration Displacement Horsepower Weight];  
rens = fitensemble(X,MPG,'LSBoost',100,'Tree');  
cvens = crossval(rens);  
L = kfoldLoss(cvens)
```

```
L =  
    21.9868
```

# RegressionEnsemble.crossval

---

## **See Also**

`RegressionPartitionedEnsemble` | `cvpartition`

<b>Purpose</b>	Cross-validated decision tree
<b>Syntax</b>	<pre>cvmodel = crossval(model) cvmodel = crossval(model,Name,Value)</pre>
<b>Description</b>	<p><code>cvmodel = crossval(model)</code> creates a partitioned model from <code>model</code>, a fitted regression tree. By default, <code>crossval</code> uses 10-fold cross validation on the training data to create <code>cvmodel</code>.</p> <p><code>cvmodel = crossval(model,Name,Value)</code> creates a partitioned model with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>Assess the predictive performance of <code>model</code> on cross-validated data using the “kfold” methods and properties of <code>cvmodel</code>, such as <code>kfoldLoss</code>.</li></ul>
<b>Input Arguments</b>	<p><b>model</b></p> <p>A regression model, produced using <code>RegressionTree.fit</code>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>'cvpartition'</b></p> <p>Object of class <code>cvpartition</code>, created by the <code>cvpartition</code> function. <code>crossval</code> splits the data into subsets with <code>cvpartition</code>.</p> <p>Use only one of these four options at a time: <code>'kfold'</code>, <code>'holdout'</code>, <code>'leaveout'</code>, or <code>'cvpartition'</code>.</p> <p><b>Default:</b> []</p>

# RegressionTree.crossval

---

## 'holdout'

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## 'kfold'

Number of folds to use in a cross-validated tree, a positive integer.

Use only one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

**Default:** 10

## 'leaveout'

Set to 'on' for leave-one-out cross-validation.

## Output Arguments

### cvmodel

A partitioned model of class RegressionPartitionedModel.

## Examples

Create a regression model of the carsmall data, and assess its accuracy with kfoldLoss:

```
load carsmall
X = [Acceleration Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
cvtree = crossval(tree);
L = kfoldLoss(cvtree)
```

```
L =
    25.2432
```

## Alternatives

You can create a cross-validation tree directly from the data, instead of creating a decision tree followed by a cross-validation tree. To do so,

include one of these five options in `RegressionTree.fit`: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

## See Also

`RegressionTree.fit` | `crossval`

# categorical.ctranspose

---

**Purpose** Transpose categorical matrix

**Syntax** `B = ctranspose(A)`

**Description** `B = ctranspose(A)` returns the transpose of the 2-D categorical matrix A. Note that `ctranspose` is identical to `transpose` for categorical arrays.

**See Also** `transpose` | `permute`

**Purpose** Cut categories

**Syntax** `C = cutcategories(t)`  
`C = cutcategories(t,nodes)`

**Description** `C = cutcategories(t)` returns an  $n$ -by-2 cell array `C` of the categories used at branches in the decision tree `t`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `C{i,1}`, and the right child is chosen if  $x$  is among those listed in `C{i,2}`. Both columns of `C` are empty for branch nodes based on continuous predictors and for leaf nodes.

`C = cutcategories(t,nodes)` takes a vector `nodes` of node numbers and returns the categories for the specified nodes.

**Examples** Create a classification tree for car data:

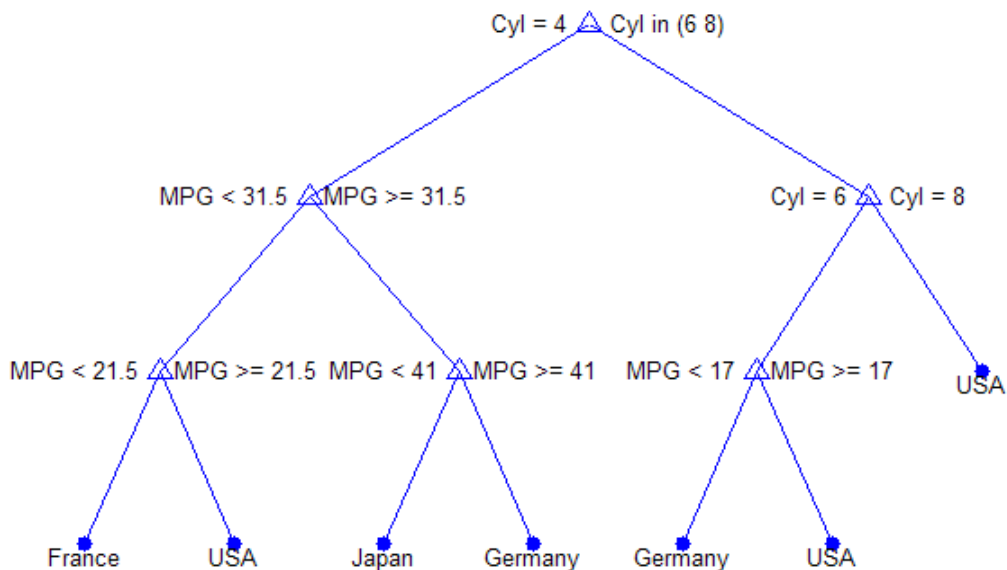
```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                'names',{'MPG' 'Cyl'},'cat',2)
t =
Decision tree for classification
 1  if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
 2  if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
 3  if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
 4  if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
 5  if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
 6  if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
 7  class = USA
 8  class = France
 9  class = USA
10  class = Japan
11  class = Germany
12  class = Germany
13  class = USA
```

# classregtree.cutcategories

view(t)

Click to display:  Magnification:  Pruning level:



```
C = cutcategories(t)
C =
    [4]    [1x2 double]
    []
    [6]    [      8]
    []
    []
    []
    []
    []
    []
```



```
      []      []
      []      []
      []      []
      []      []
      []      []
C{1,2}
ans =
     6     8
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree | cutpoint | cuttype | cutvar

# classregtree.cutpoint

---

**Purpose** Decision tree cut point values

**Syntax** `v = cutpoint(t)`  
`v = cutpoint(t,nodes)`

**Description** `v = cutpoint(t)` returns an  $n$ -element vector  $v$  of the values used as cut points in the decision tree  $t$ , where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if  $x < v(i)$  and the right child is chosen if  $x \geq v(i)$ .  $v$  is NaN for branch nodes based on categorical predictors and for leaf nodes.

`v = cutpoint(t,nodes)` takes a vector `nodes` of node numbers and returns the cut points for the specified nodes.

**Examples** Create a classification tree for car data:

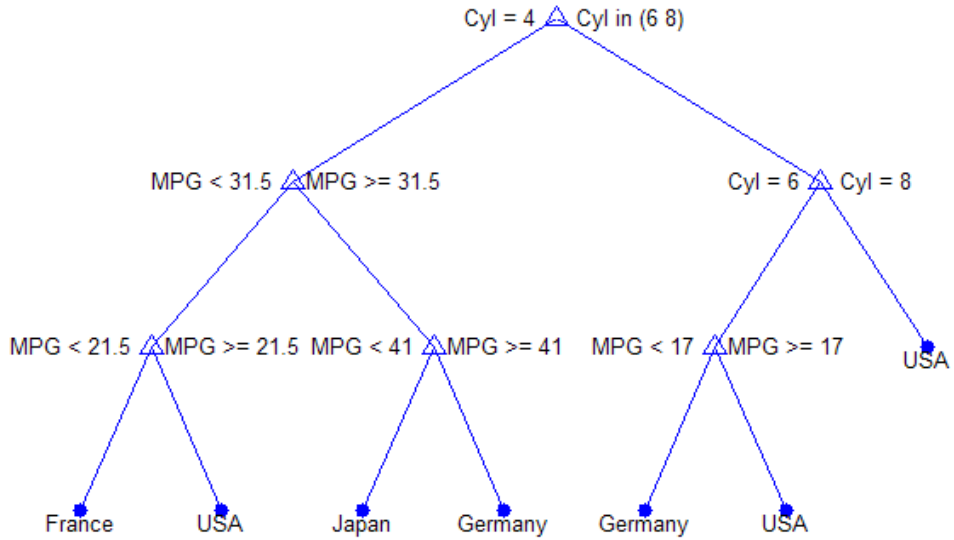
```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                'names',{'MPG' 'Cyl'},'cat',2)

t =
Decision tree for classification
 1  if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
 2  if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
 3  if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
 4  if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
 5  if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
 6  if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
 7  class = USA
 8  class = France
 9  class = USA
10  class = Japan
11  class = Germany
12  class = Germany
13  class = USA
```

view(t)

Click to display:  Magnification:  Pruning level:



```
v = cutpoint(t)
```

```
v =
```

```
NaN  
31.5000  
NaN  
21.5000  
41.0000  
17.0000  
NaN  
NaN  
NaN  
NaN
```

# classregtree.cutpoint

---

NaN

NaN

NaN

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[classregtree](#) | [cutcategories](#) | [cuttype](#) | [cutvar](#)

**Purpose**

Cut types

**Syntax**

```
c = cuttype(t)
c = cuttype(t,nodes)
```

**Description**

`c = cuttype(t)` returns an  $n$ -element cell array `c` indicating the type of cut at each node in the tree `t`, where  $n$  is the number of nodes. For each node  $i$ , `c{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .
- 'categorical' — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

`cutvar` returns the cut points for 'continuous' cuts, and `cutcategories` returns the set of categories.

`c = cuttype(t,nodes)` takes a vector `nodes` of node numbers and returns the cut types for the specified nodes.

**Examples**

Create a classification tree for car data:

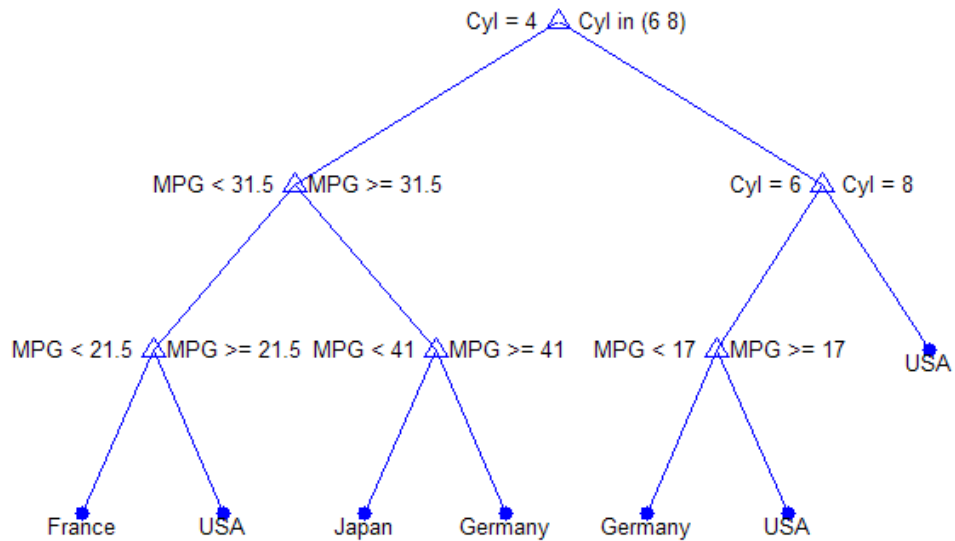
```
load carsmall

t = classregtree([MPG Cylinders],Origin,...
                'names',{'MPG' 'Cyl'},'cat',2)
t =
Decision tree for classification
 1 if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
 2 if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
 3 if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
 4 if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
 5 if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
 6 if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
 7 class = USA
```

# classregtree.cuttype

```
8 class = France
9 class = USA
10 class = Japan
11 class = Germany
12 class = Germany
13 class = USA
view(t)
```

Click to display:  Magnification:  Pruning level:



```
c = cuttype(t)
c =
'categorical'
'continuous'
'categorical'
'continuous'
```

```
'continuous'  
'continuous'  
''  
''  
''  
''  
''  
''  
''  
''
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes` | `cutvar` | `cutcategories`

# classregtree.cutvar

---

**Purpose** Cut variable names

**Syntax**  
`v = cutvar(t)`  
`v = cutvar(t,nodes)`  
`[v,num] = cutvar(...)`

**Description** `v = cutvar(t)` returns an  $n$ -element cell array `v` of the names of the variables used for branching in each node of the tree `t`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, `v` contains an empty string.

`v = cutvar(t,nodes)` takes a vector `nodes` of node numbers and returns the cut variables for the specified nodes.

`[v,num] = cutvar(...)` also returns a vector `num` containing the number of each variable.

**Examples** Create a classification tree for car data:

```
load carsmall

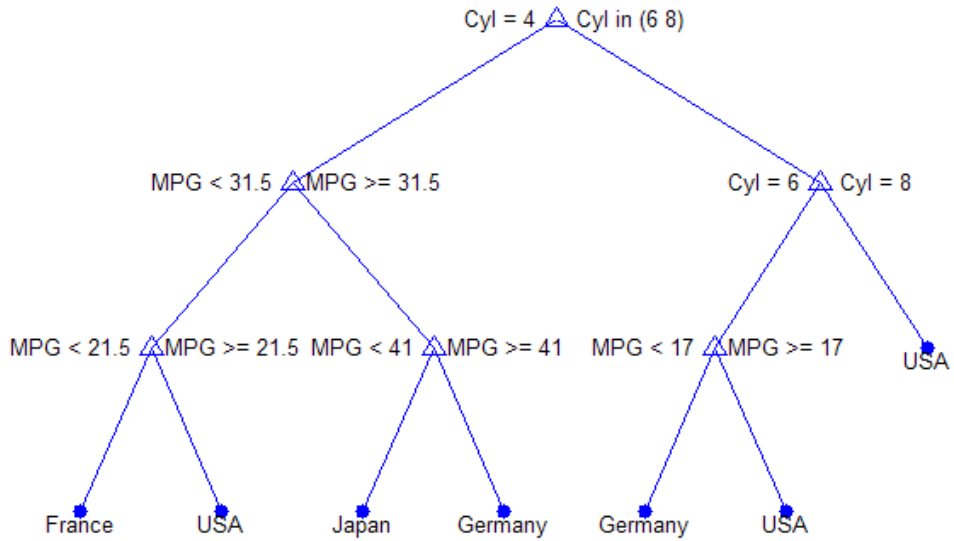
t = classregtree([MPG Cylinders],Origin,...
                'names',{'MPG' 'Cyl'},'cat',2)

t =
Decision tree for classification
 1 if Cyl=4 then node 2 elseif Cyl in {6 8} then node 3 else USA
 2 if MPG<31.5 then node 4 elseif MPG>=31.5 then node 5 else USA
 3 if Cyl=6 then node 6 elseif Cyl=8 then node 7 else USA
 4 if MPG<21.5 then node 8 elseif MPG>=21.5 then node 9 else USA
 5 if MPG<41 then node 10 elseif MPG>=41 then node 11 else Japan
 6 if MPG<17 then node 12 elseif MPG>=17 then node 13 else USA
 7 class = USA
 8 class = France
 9 class = USA
10 class = Japan
11 class = Germany
12 class = Germany
13 class = USA
```



view(t)

Click to display:  Magnification:  Pruning level:



```
[v,num] = cutvar(t)
```

```
v =
```

```

'Cyl'
'MPG'
'Cyl'
'MPG'
'MPG'
'MPG'
''
''
''

```

# classregtree.cutvar

---

```
    ''
    ''
    ''
    ''
num =
    2
    1
    2
    1
    1
    1
    0
    0
    0
    0
    0
    0
    0
    0
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[classregtree](#) | [numnodes](#) | [children](#)

## Purpose

Classification error by cross validation

## Syntax

```
E = cvLoss(tree)
[E,SE] = cvLoss(tree)
[E,SE,Nleaf] = cvLoss(tree)
[E,SE,Nleaf,BestLevel] = cvLoss(tree)
[E,...] = cvLoss(tree,Name,Value)
```

## Description

`E = cvLoss(tree)` returns the cross-validated classification error (loss) for `tree`, a classification tree.

`[E,SE] = cvLoss(tree)` returns the standard error of `E`.

`[E,SE,Nleaf] = cvLoss(tree)` returns the number of leaves of `tree`.

`[E,SE,Nleaf,BestLevel] = cvLoss(tree)` returns the optimal pruning level for `tree`.

`[E,...] = cvLoss(tree,Name,Value)` cross validates with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### `tree`

A classification tree produced by `ClassificationTree.fit`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### 'subtrees'

Vector of pruning levels, with 0 representing the full, unpruned tree. `tree` must include a pruning sequence as created either by `ClassificationTree.fit` with 'prune' set to 'on', or by the

# ClassificationTree.cvLoss

---

prune method. The returned `E`, `SE`, and `Nleaf` are vectors of the same length as `subtrees`; `BestLevel` is a scalar. If you set `subtrees` to `'all'`, `cvLoss` uses the entire pruning sequence.

**Default:** 0

## **'treesize'**

One of the following strings:

- `'se'` — `cvLoss` uses the smallest tree whose cost is within one standard error of the minimum cost.
- `'min'` — `cvLoss` uses the minimal cost tree.

**Default:** `'se'`

## **'kfold'**

Number of cross-validation samples, a positive integer.

**Default:** 10

## **Output Arguments**

### **E**

The cross-validation classification error (loss). A vector or scalar depending on the setting of the `subtrees` name-value pair.

### **SE**

The standard error of `E`. A vector or scalar depending on the setting of the `subtrees` name-value pair.

### **Nleaf**

Number of leaf nodes in `tree`. Leaf nodes are terminal nodes, which give classifications, not splits. A vector or scalar depending on the setting of the `subtrees` name-value pair.

### **BestLevel**

By default, a scalar representing the largest pruning level that achieves a value of  $E$  within  $SE$  of the minimum error. If you set `treeSize` to 'min', `BestLevel` is the smallest value in subtrees.

## Examples

Compute the cross-validation error for the default classification tree for the ionosphere data:

```
load ionosphere
tree = ClassificationTree.fit(X,Y);
[E,SE,Nleaf,BestLevel] = cvLoss(tree)
```

```
E =
    0.1282
```

```
SE =
    0.0178
```

```
Nleaf =
    19
```

```
BestLevel =
    0
```

Find the best level by using the subtrees name-value pair:

```
[~,~,~,BestLevel] = cvLoss(tree,'subtrees','all')
```

```
BestLevel =
    6
```

## Alternatives

You can construct a cross-validated tree model with `crossval`, and call `kfoldLoss` instead of `cvLoss`. The alternative can save time if you are going to examine the cross-validated tree more than once.

However, unlike `cvLoss`, `kfoldLoss` does not return `SE`, `Nleaf`, or `BestLevel`. `kfoldLoss` also does not allow you to examine any error other than classification error.

# ClassificationTree.cvLoss

---

## See Also

`ClassificationTree.fit` | `crossval` | `loss` | `kfoldLoss`

## Purpose

Regression error by cross validation

## Syntax

```
E = cvLoss(tree)
[E,SE] = cvLoss(tree)
[E,SE,Nleaf] = cvLoss(tree)
[E,SE,Nleaf,BestLevel] = cvLoss(tree)
[E,...] = cvLoss(tree,Name,Value)
```

## Description

`E = cvLoss(tree)` returns the cross-validated regression error (loss) for `tree`, a regression tree.

`[E,SE] = cvLoss(tree)` returns the standard error of `E`.

`[E,SE,Nleaf] = cvLoss(tree)` returns the number of leaves (terminal nodes) in `tree`.

`[E,SE,Nleaf,BestLevel] = cvLoss(tree)` returns the optimal pruning level for `tree`.

`[E,...] = cvLoss(tree,Name,Value)` cross validates with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### `tree`

A regression tree produced by `RegressionTree.fit`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### 'subtrees'

Vector of pruning levels, with 0 representing the full, unpruned tree. `tree` must include a pruning sequence as created either by

# RegressionTree.cvLoss

---

ClassificationTree.fit with 'prune' set to 'on', or by the prune method. The returned E, SE, and Nleaf are vectors of the same length as subtrees; BestLevel is a scalar. If you set subtrees to 'all', cvLoss uses the entire pruning sequence.

**Default:** 0

## 'treesize'

One of the following strings:

- 'se' — cvLoss uses the smallest tree whose cost is within one standard error of the minimum cost.
- 'min' — cvLoss uses the minimal cost tree.

**Default:** 'se'

## 'kfold'

Number of cross-validation samples, a positive integer.

**Default:** 10

## Output Arguments

### E

The cross-validation mean squared error (loss). A vector or scalar depending on the setting of the subtrees name-value pair.

### SE

The standard error of E. A vector or scalar depending on the setting of the subtrees name-value pair.

### Nleaf

Number of leaf nodes in tree. Leaf nodes are terminal nodes, which give responses, not splits. A vector or scalar depending on the setting of the subtrees name-value pair.

### BestLevel



By default, a scalar representing the largest pruning level that achieves a value of `E` within `SE` of the minimum error. If you set `treecsize` to `'min'`, `BestLevel` is the smallest value in subtrees.

## Examples

Compute the cross-validation error for the default classification tree for the `carsmall` data:

```
load carsmall
X = [Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
[E,SE,Nleaf,BestLevel] = cvLoss(tree)
```

```
E =
    30.7558
```

```
SE =
    6.0651
```

```
Nleaf =
    19
```

```
BestLevel =
    0
```

Find the best level by using the subtrees name-value pair:

```
[~,~,~,BestLevel] = cvLoss(tree,'subtrees','all')
```

```
BestLevel =
    15
```

## Alternatives

You can construct a cross-validated tree model with `crossval`, and call `kfoldLoss` instead of `cvLoss`. The alternative can save time if you are going to examine the cross-validated tree more than once.

However, unlike `cvLoss`, `kfoldLoss` does not return `SE`, `Nleaf`, or `BestLevel`.

# RegressionTree.cvLoss

---

## See Also

[crossval](#) | [kfoldLoss](#) | [RegressionTree.fit](#) | [loss](#)

<b>Purpose</b>	Data partitions for cross validation	
<b>Description</b>	An object of the <code>cvpartition</code> class defines a random partition on a set of data of a specified size. Use this partition to define test and training sets for validating a statistical model using cross validation.	
<b>Construction</b>	<code>cvpartition</code>	Create cross validation partition for data
<b>Methods</b>	<code>disp</code>	Display <code>cvpartition</code> object
	<code>display</code>	Display <code>cvpartition</code> object
	<code>repartition</code>	Repartition data for cross-validation
	<code>test</code>	Test indices for cross-validation
	<code>training</code>	Training indices for cross-validation
<b>Properties</b>	<code>N</code>	Number of observations (including observations with missing group values)
	<code>NumTestSets</code>	Number of test sets
	<code>TestSize</code>	Size of each test set
	<code>TrainSize</code>	Size of each training set
	<code>Type</code>	Type of partition
<b>Copy Semantics</b>	Value. To learn how this affects your use of the class, see <a href="#">Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation</a> .	

# cvpartition

---

## Examples

Use a 10-fold stratified cross validation to compute the misclassification error for `classify` on iris data.

```
load('fisheriris');
CVO = cvpartition(species,'k',10);
err = zeros(CVO.NumTestSets,1);
for i = 1:CVO.NumTestSets
    trIdx = CVO.training(i);
    teIdx = CVO.test(i);
    ytest = classify(meas(teIdx,:),meas(trIdx,:),...
        species(trIdx,:));
    err(i) = sum(~strcmp(ytest,species(teIdx)));
end
cvErr = sum(err)/sum(CVO.TestSize);
```

## See Also

`crossval`

## How To

- “Grouping Variables” on page 2-51

**Purpose**

Create cross validation partition for data

**Syntax**

```
c = cvpartition(n, 'kfold', k)
c = cvpartition(group, 'kfold', k)
c = cvpartition(n, 'holdout', p)
c = cvpartition(group, 'holdout', p)
c = cvpartition(n, 'leaveout')
c = cvpartition(n, 'resubstitution')
```

**Description**

`c = cvpartition(n, 'kfold', k)` constructs an object `c` of the `cvpartition` class defining a random partition for `k`-fold cross validation on `n` observations. The partition divides the observations into `k` disjoint subsamples (or *fold*s), chosen randomly but with roughly equal size. The default value of `k` is 10.

`c = cvpartition(group, 'kfold', k)` creates a random partition for a stratified `k`-fold cross validation. `group` is a numeric vector, categorical array, string array, or cell array of strings indicating the class of each observation. Each subsample has roughly equal size and roughly the same class proportions as in `group`. `cvpartition` treats NaNs or empty strings in `group` as missing values.

`c = cvpartition(n, 'holdout', p)` creates a random partition for holdout validation on `n` observations. This partition divides the observations into a training set and a test (or *holdout*) set. The parameter `p` must be a scalar. When  $0 < p < 1$ , `cvpartition` randomly selects approximately  $p*n$  observations for the test set. When `p` is an integer, `cvpartition` randomly selects `p` observations for the test set. The default value of `p` is 1/10.

`c = cvpartition(group, 'holdout', p)` randomly partitions observations into a training set and a test set with stratification, using the class information in `group`; that is, both training and test sets have roughly the same class proportions as in `group`.

`c = cvpartition(n, 'leaveout')` creates a random partition for leave-one-out cross validation on `n` observations. Leave-one-out is a special case of 'kfold', in which the number of folds equals the number of observations.

# cvpartition

---

`c = cvpartition(n, 'resubstitution')` creates an object `c` that does not partition the data. Both the training set and the test set contain all of the original `n` observations.

## Examples

Use stratified 10-fold cross validation to compute misclassification rate:

```
load fisheriris;
y = species;
c = cvpartition(y, 'k', 10);

fun = @(xT,yT,xt,yt)(sum(~strcmp(yt,classify(xt,xT,yT))));

rate = sum(crossval(fun,meas,y,'partition',c)...
           /sum(c.TestSize)
rate =
    0.0200
```

## See Also

`crossval` | `repartition`

## How To

- “Grouping Variables” on page 2-51

## Purpose

Cross-validate regularization of linear discriminant

## Syntax

```
err = cvshrink(obj)
[err,gamma] = cvshrink(obj)
[err,gamma,delta] = cvshrink(obj)
[err,gamma,delta,numpred] = cvshrink(obj)
[err,...] = cvshrink(obj,Name,Value)
```

## Description

`err = cvshrink(obj)` returns a vector of cross-validated classification error values for differing values of the regularization parameter Gamma.

`[err,gamma] = cvshrink(obj)` also returns the vector of Gamma values.

`[err,gamma,delta] = cvshrink(obj)` also returns the vector of Delta values.

`[err,gamma,delta,numpred] = cvshrink(obj)` returns the vector of number of nonzero predictors for each setting of the parameters Gamma and Delta.

`[err,...] = cvshrink(obj,Name,Value)` cross validates with additional options specified by one or more Name,Value pair arguments.

## Tips

- Examine the `err` and `numpred` outputs to see the tradeoff between cross-validated error and number of predictors. When you find a satisfactory point, set the corresponding `gamma` and `delta` properties in the model by dot addressing. For example, if  $(i, j)$  is the location of the satisfactory point, set

```
obj.Gamma = gamma(i);
obj.Delta = delta(i,j);
```

## Input Arguments

### **obj**

Discriminant analysis classifier, produced using `ClassificationDiscriminant.fit`.

# ClassificationDiscriminant.cvshrink

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### 'delta'

- **Scalar delta** — `cvshrink` uses this value of `delta` with every value of `gamma` for regularization.
- **Row vector delta** — For each `i` and `j`, `cvshrink` uses `delta(j)` with `gamma(i)` for regularization.
- **Matrix delta** — The number of rows of `delta` must equal the number of elements in `gamma`. For each `i` and `j`, `cvshrink` uses `delta(i, j)` with `gamma(i)` for regularization.

**Default:** 0

### 'gamma'

Vector of Gamma values for cross-validation.

**Default:** 0:0.1:1

### 'NumDelta'

Number of Delta intervals for cross-validation. For every value of Gamma, `cvshrink` cross-validates the discriminant using `NumDelta + 1` values of Delta, uniformly spaced from zero to the maximal Delta at which all predictors are eliminated for this value of Gamma. If you set `delta`, `cvshrink` ignores `NumDelta`.

**Default:** 0

### 'NumGamma'



Number of Gamma intervals for cross-validation. `cvshrink` cross-validates the discriminant using `NumGamma + 1` values of Gamma, uniformly spaced from `MinGamma` to 1. If you set `gamma`, `cvshrink` ignores `NumGamma`.

**Default:** 10

## **'verbose'**

Verbosity level, an integer from 0 to 2. Higher values give more progress messages.

**Default:** 0

## **Output Arguments**

### **err**

Numeric vector or matrix of errors. `err` is the misclassification error rate, meaning the average fraction of misclassified data over all folds.

- If `delta` is a scalar (default), `err(i)` is the misclassification error rate for `obj` regularized with `gamma(i)`.
- If `delta` is a vector, `err(i,j)` is the misclassification error rate for `obj` regularized with `gamma(i)` and `delta(j)`.
- If `delta` is a matrix, `err(i,j)` is the misclassification error rate for `obj` regularized with `gamma(i)` and `delta(i,j)`.

### **gamma**

Vector of Gamma values used for regularization. See “Gamma and Delta” on page 21-610.

### **delta**

Vector or matrix of Delta values used for regularization. See “Gamma and Delta” on page 21-610.

# ClassificationDiscriminant.cvshrink

---

- If you give a scalar for the `delta` name-value pair, the output `delta` is a row vector the same size as `gamma`, with entries equal to the input scalar.
- If you give a row vector for the `delta` name-value pair, the output `delta` is a matrix with the same number of columns as the row vector, and with the number of rows equal to the number of elements of `gamma`. The output `delta(i, j)` is equal to the input `delta(j)`.
- If you give a matrix for the `delta` name-value pair, the output `delta` is the same as the input matrix. The number of rows of `delta` must equal the number of elements in `gamma`.

## **numpred**

Numeric vector or matrix containing the number of predictors in the model at various regularizations. `numpred` has the same size as `err`.

- If `delta` is a scalar (default), `numpred(i)` is the number of predictors for `obj` regularized with `gamma(i)` and `delta`.
- If `delta` is a vector, `numpred(i, j)` is the number of predictors for `obj` regularized with `gamma(i)` and `delta(j)`.
- If `delta` is a matrix, `numpred(i, j)` is the number of predictors for `obj` regularized with `gamma(i)` and `delta(i, j)`.

## **Definitions**

### **Gamma and Delta**

Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters,  $\gamma$  and  $\delta$ , that control regularization as follows. `cvshrink` helps you select appropriate values of the parameters.

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be the correlation matrix of the data  $X$ , and let  $\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = \left[ (x - \mu_0)^T D^{-1/2} \right] \left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square brackets. Each component of the vector

$\left[ \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When  $\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$\left| \tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0) \right| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

# ClassificationDiscriminant.cvshrink

---

## Examples

### Regularize Data with Many Predictors

Regularize a discriminant analysis classifier, and view the tradeoff between the number of predictors in the model and the classification accuracy.

Create a linear discriminant analysis classifier for the ovariancancer data. Set the `SaveMemory` and `FillCoeffs` options to keep the resulting model reasonably small.

```
load ovariancancer
obj = ClassificationDiscriminant.fit(obs,grp,...
    'SaveMemory','on','FillCoeffs','off');
```

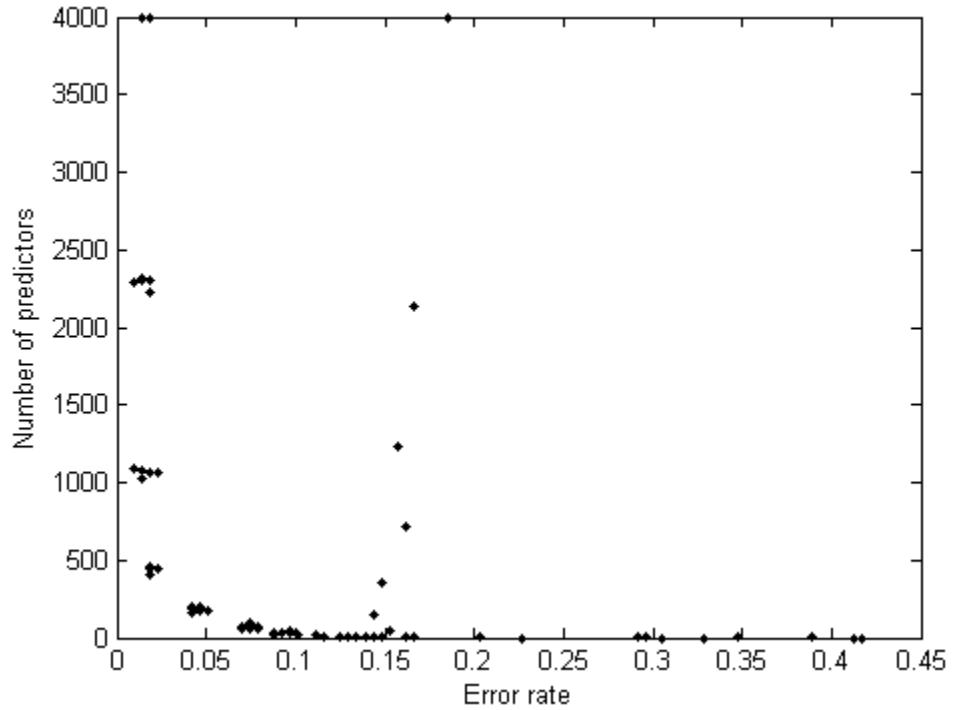
Use 10 levels of `Gamma` and 10 levels of `Delta` to search for good parameters. This search is time-consuming. Set `Verbose` to 1 to view the progress.

```
rng('default') % for reproducibility
[err,gamma,delta,numpred] = cvshrink(obj,...
    'NumGamma',9,'NumDelta',9,'Verbose',1);
```

```
Done building cross-validated model.
Processing Gamma step 1 out of 10.
Processing Gamma step 2 out of 10.
Processing Gamma step 3 out of 10.
Processing Gamma step 4 out of 10.
Processing Gamma step 5 out of 10.
Processing Gamma step 6 out of 10.
Processing Gamma step 7 out of 10.
Processing Gamma step 8 out of 10.
Processing Gamma step 9 out of 10.
Processing Gamma step 10 out of 10.
```

Plot the classification error rate against the number of predictors.

```
plot(err,numpred,'k.')
xlabel('Error rate');
ylabel('Number of predictors');
```



## See Also

[ClassificationDiscriminant](#) | [ClassificationDiscriminant.fit](#) |

## Related Examples

- “Regularize a Discriminant Analysis Classifier” on page 15-23

## Concepts

- “Discriminant Analysis” on page 15-3

# RegressionEnsemble.cvshrink

---

**Purpose** Cross validate shrinking (pruning) ensemble

**Syntax**

```
vals = cvshrink(ens)
[vals,nlearn] = cvshrink(ens)
[vals,nlearn] = cvshrink(ens,Name,Value)
```

**Description** `vals = cvshrink(ens)` returns an L-by-T matrix with cross-validated values of the mean squared error. L is the number of `lambda` values in the `ens.Regularization` structure. T is the number of `threshold` values on weak learner weights. If `ens` does not have a `Regularization` property filled in by the `regularize` method, pass a `lambda` name-value pair.

`[vals,nlearn] = cvshrink(ens)` returns an L-by-T matrix of the mean number of learners in the cross-validated ensemble.

`[vals,nlearn] = cvshrink(ens,Name,Value)` cross validates with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

**ens**

A regression ensemble, created with `fitensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

#### 'cvpartition'

A partition created with `cvpartition` to use in a cross-validated tree. You can only use one of these four options at a time: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

#### 'holdout'

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated tree: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## **'kfold'**

Number of folds to use in a cross-validated tree, a positive integer. If you do not supply a cross-validation method, cvshrink uses 10-fold cross validation. You can only use one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

**Default:** 10

## **'lambda'**

Vector of nonnegative regularization parameter values for lasso. If empty, cvshrink does not perform cross validation.

**Default:** []

## **'leaveout'**

Use leave-one-out cross validation by setting to 'on'. You can only use one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## **'threshold'**

Numeric vector with lower cutoffs on weights for weak learners. cvshrink discards learners with weights below threshold in its cross-validation calculation.

**Default:** 0

## **Output Arguments**

### **vals**

L-by-T matrix with cross-validated values of the mean squared error. L is the number of values of the regularization parameter

# RegressionEnsemble.cvshrink

---

'lambda', and T is the number of 'threshold' values on weak learner weights.

## nlearn

L-by-T matrix with cross-validated values of the mean number of learners in the cross-validated ensemble. L is the number of values of the regularization parameter 'lambda', and T is the number of 'threshold' values on weak learner weights.

## Examples

Create a regression ensemble for predicting mileage from the carsmall data. Cross validate the ensemble for three values each of lambda and threshold.

```
load carsmall
X = [Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'bag',100,'Tree',...
    'type','regression');
[vals nlearn] = cvshrink(ens,'lambda',[.01 .1 1],...
    'threshold',[0 .01 .1])

vals =
    20.0949    19.9007    131.6316
    20.0924    19.8431    128.0989
    19.9759    19.7987    119.5574

nlearn =
    13.3000    11.6000     3.5000
    13.2000    11.5000     3.6000
    13.4000    11.4000     3.9000
```

Clearly, setting a threshold of 0.1 leads to unacceptable errors, while a threshold of 0.01 gives similar errors to a threshold of 0. The mean number of learners with a threshold of 0.1 is about 11.5, whereas the mean number is about 13.2 when the threshold is 0.

## See Also

[regularize](#) | [shrink](#)



<b>Purpose</b>	Randomly sample from data, with or without replacement
<b>Syntax</b>	<pre> y = datasample(data,k) y = datasample(data,k,dim) [y,idx] = datasample(data,k,...) [y,...] = datasample(s,data,k,...) [y,...] = datasample(data,k,Name,Value) [y,...] = datasample(data,k,dim,Name,Value) </pre>
<b>Description</b>	<p><code>y = datasample(data,k)</code> returns <code>k</code> observations sampled uniformly at random, with replacement, from the data in <code>data</code>.</p> <p><code>y = datasample(data,k,dim)</code> returns a sample taken along dimension <code>dim</code> of data.</p> <p><code>[y,idx] = datasample(data,k,...)</code> returns an index vector indicating which values <code>datasample</code> sampled from <code>data</code>.</p> <p><code>[y,...] = datasample(s,data,k,...)</code> uses the random number stream <code>s</code> to generate random numbers.</p> <p><code>[y,...] = datasample(data,k,Name,Value)</code> or <code>[y,...] = datasample(data,k,dim,Name,Value)</code> samples with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Tips</b>	<ul style="list-style-type: none"> <li>• To sample random integers with replacement from a range, use <code>randi</code>.</li> <li>• To sample random integers without replacement, use <code>randperm</code> or <code>datasample</code>.</li> <li>• To randomly sample from data, with or without replacement, use <code>datasample</code>.</li> </ul>
<b>Input Arguments</b>	<p><b>data</b></p> <p>Vector, matrix, <math>N</math>-dimensional array, or dataset array representing the data from which to sample. By default, <code>datasample</code> regards the rows of a data matrix, or the first nonsingleton dimension of a data array, as data elements. Change this behavior with the <code>dim</code> argument.</p>

## **k**

Positive integer, the number of samples.

## **dim**

Integer specifying the dimension on which to take samples. For example, if `data` is a matrix and `dim` is 2, `y` contains a selection of columns in `data`. If `data` is a dataset array and `dim` is 2, `y` contains a selection of variables in `data`. Use `dim` to ensure sampling along a specific dimension regardless of whether `data` is a vector, matrix or  $N$ -dimensional array.

**Default:** 1

## **s**

Random number stream. Create `s` using `rng` or `RandStream`.

**Default:** The global random number stream

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## **'Replace'**

Select the sample with replacement if `Replace` is true, or without replacement if `Replace` is false. If `Replace` is false, `k` must not be larger than the number of data elements in `data`.

**Default:** true

## **'Weights'**

Vector with the same number of elements as data elements in `data`, and with nonnegative elements. Sample with probability proportional to the elements of `Weights`.

**Default:** `ones(datasize,1)`, where `datasize` is the number of data elements in `data`

## Output Arguments

### `y`

- If `data` is a vector, `y` is a vector containing `k` elements selected from `data`.
- If `data` is a matrix, `y` is a matrix containing `k` rows selected from `data`. Or, if `dim = 2`, `y` is a matrix containing `k` columns selected from `data`
- If `data` is an  $N$ -dimensional array, `datasample` samples along its first non-singleton dimension. Or, if you give a `dim` name-value pair, `datasample` samples along the dimension `dim`.

When the sample is taken with replacement (default), `y` can contain repeated observations from `data`. Set the `Replace` name-value pair to `false` to sample without replacement.

### `idx`

Vector of indices indicating which elements `datasample` chose from `data` to create `y`. For example:

- If `data` is a vector, `y = data(idx)`.
- If `data` is a matrix, `y = data(idx,:)`.

## Examples

Draw five unique values from the integers `1:10`.

```
y = datasample(1:10,5,'Replace',false)
```

```
y =  
     6     3     7     8     5
```

---

# datasample

---

Generate a random sequence of the characters ACGT, with replacement, according to specified probabilities.

```
seq = datasample('ACGT',48,'Weights',[0.15 0.35 0.35 0.15])

seq =
CTTCGACTGTGAGTGGGCGCGACAAGGCTACCGGCCCGGGCGGCACTC
```

---

Select a random subset of columns from a data matrix.

```
X = randn(10,1000);
Y = datasample(X,5,2,'Replace',false)

Y =
    0.7007    0.3382    2.1298   -0.1891    0.5026
    0.6520   -0.6693   -0.1961   -0.9915    1.9107
    0.1785    0.6640    2.3247   -1.1735   -1.0020
    1.6760    2.6102   -0.8902   -0.7735    1.8676
   -0.3251   -0.6415   -0.2572   -0.1629   -1.0523
    0.1011    0.9323   -1.3088   -0.4477    0.8036
   -0.5767   -0.5778   -0.8556    0.8672   -0.0727
   -0.0615   -0.9084    0.9020   -0.4185   -1.9520
    0.7256   -1.1228    0.7558    1.2691    2.4997
   -1.2273    0.5754   -0.8755   -0.8224   -1.2066
```

---

Resample observations from a dataset array to create a bootstrap replicate dataset.

```
load hospital
y = datasample(hospital,size(hospital,1));
```

---

Use the second output to sample "in parallel" from two data vectors.

```
x1 = randn(100,1);
```

```
x2 = randn(100,1);  
[y1,idx] = datasample(x1,10);  
y2 = x2(idx);
```

## Algorithms

`datasample` uses `randperm`, `rand`, or `randi` to generate random values. Therefore, `datasample` changes the state of the MATLAB global random number generator. Control the random number generator using `rng`.

For selecting weighted samples without replacement, `datasample` uses the algorithm of Wong and Easton [1].

## References

[1] Wong, C. K. and M. C. Easton. *An Efficient Method for Weighted Sampling Without Replacement*. SIAM Journal of Computing 9(1), pp. 111–113, 1980.

## Alternatives

You can use `randi` or `randperm` to generate indices for random sampling with or without replacement, respectively. However, `datasample` can be more convenient because it samples directly from your data. `datasample` also allows weighted sampling.

## See Also

`rand` | `randi` | `randperm` | `RandStream` | `rng`

# dataset

---

**Purpose** Arrays for statistical data

**Description** Dataset arrays are used to collect heterogeneous data and metadata including variable and observation names into a single container variable. Dataset arrays are suitable for storing column-oriented or tabular data that are often stored as columns in a text file or in a spreadsheet, and can accommodate variables of different types, sizes, units, etc.

Dataset arrays can contain different kinds of variables, including numeric, logical, character, categorical, and cell. However, a dataset array is a different class than the variables that it contains. For example, even a dataset array that contains only variables that are double arrays cannot be operated on as if it were itself a double array. However, using dot subscripting, you can operate on variable in a dataset array as if it were a workspace variable.

You can subscript dataset arrays using parentheses much like ordinary numeric arrays, but in addition to numeric and logical indices, you can use variable and observation names as indices.

**Construction** Use the dataset constructor to create a dataset array from variables in the MATLAB workspace. You can also create a dataset array by reading data from a text or spreadsheet file. You can access each variable in a dataset array much like fields in a structure, using dot subscripting. See the following section for a list of operations available for dataset arrays.

dataset Construct dataset array

**Methods** cat Concatenate dataset arrays

cellstr Create cell array of strings from dataset array

dataset2cell Convert dataset array to cell array

---

dataset2struct	Convert dataset array to structure
datasetfun	Apply function to dataset array variables
disp	Display dataset array
display	Display dataset array
double	Convert dataset variables to double array
end	Last index in indexing expression for dataset array
export	Write dataset array to file
get	Access dataset array properties
horzcat	Horizontal concatenation for dataset arrays
intersect	Set intersection for dataset array observations
isempty	True for empty dataset array
ismember	Dataset array elements that are members of set
ismissing	Find dataset array elements with missing values
join	Merge observations
length	Length of dataset array
ndims	Number of dimensions of dataset array
numel	Number of elements in dataset array
replacedata	Replace dataset variables

# dataset

---

<code>replaceWithMissing</code>	Insert missing data indicators into a dataset array
<code>set</code>	Set and display properties
<code>setdiff</code>	Set difference for dataset array observations
<code>setxor</code>	Set exclusive or for dataset array observations
<code>single</code>	Convert dataset variables to single array
<code>size</code>	Size of dataset array
<code>sortrows</code>	Sort rows of dataset array
<code>stack</code>	Stack data from multiple variables into single variable
<code>subsasgn</code>	Subscripted assignment to dataset array
<code>subsref</code>	Subscripted reference for dataset array
<code>summary</code>	Print summary of dataset array
<code>union</code>	Set union for dataset array observations
<code>unique</code>	Unique observations in dataset array
<code>unstack</code>	Unstack data from single variable into multiple variables
<code>vertcat</code>	Vertical concatenation for dataset arrays

## Properties

A dataset array `D` has properties that store metadata (information about your data). Access or assign to a property using `P =`



`D.Properties.PropName` or `D.Properties.PropName = P`, where `PropName` is one of the following:

<code>Description</code>	String describing data set
<code>DimNames</code>	Two-element cell array of strings giving names of dimensions of data set
<code>ObsNames</code>	Cell array of nonempty, distinct strings giving names of observations in data set
<code>Units</code>	Units of variables in data set
<code>UserData</code>	Variable containing additional information associated with data set
<code>VarDescription</code>	Cell array of strings giving descriptions of variables in data set
<code>VarNames</code>	Cell array giving names of variables in data set

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## Examples

Load a dataset array from a `.mat` file and create some simple subsets:

```
load hospital
h1 = hospital(1:10,:)
h2 = hospital(:, {'LastName' 'Age' 'Sex' 'Smoker'})

% Access and modify metadata
hospital.Properties.Description
hospital.Properties.VarNames{4} = 'Wgt'
```

# dataset

---

```
% Create a new dataset variable from an existing one
hospital.AtRisk = hospital.Smoker | (hospital.Age > 40)

% Use individual variables to explore the data
boxplot(hospital.Age,hospital.Sex)
h3 = hospital(hospital.Age<30,...
    {'LastName' 'Age' 'Sex' 'Smoker'})

% Sort the observations based on two variables
h4 = sortrows(hospital,{'Sex','Age'})
```

## See Also

genvarname | tdfread | textscan | xlsread

## How To

- “Dataset Arrays” on page 2-135

**Purpose**

Construct dataset array

**Syntax**

```
A = dataset(varspec, 'ParamName', Value)  
A = dataset('File', filename, 'ParamName', Value)  
A = dataset('XLSFile', filename, 'ParamName', Value)  
A = dataset('XPTFile', xptfilename, 'ParamName', Value)
```

**Description**

`A = dataset(varspec, 'ParamName', Value)` creates dataset array `A` using the workspace variable input method `varspec` and one or more optional name/value pairs (see Parameter Name/Value Pairs).

The input method `varspec` can be one or more of the following:

- `VAR` — a workspace variable. `dataset` uses the workspace name for the variable name in `A`. To include multiple variables, specify `VAR_1,VAR_2,...,VAR_N`. Variables can be arrays of any size, but all variables must have the same number of rows. `VAR` can also be an expression. In this case, `dataset` creates a default name automatically.
- `{VAR,name}` — a workspace variable, `VAR` and a variable name, `name`. `dataset` uses `name` as the variable name. To include multiple variables and names, specify `{VAR_1,name_1}, {VAR_2,name_2},..., {VAR_N,name_N}`.
- `{VAR,name_1,...,name_m}` — an `m`-columned workspace variable, `VAR`. `dataset` uses the names `name_1, . . . , name_m` as variable names. You must include a name for every column in `VAR`. Each column becomes a separate variable in `A`.

You can combine these input methods to include as many variables and names as needed. Names must be valid, unique MATLAB identifier strings. For example input combinations, see Examples. For optional name/value pairs see Inputs.

To convert numeric arrays, cell arrays, or structure arrays to dataset arrays, you can also use (respectively):

- `mat2dataset`

- `cell2dataset`
- `struct2dataset`

---

**Note** Dataset arrays may contain built-in types or array objects as variables. Array objects must implement each of the following:

- Standard MATLAB parenthesis indexing of the form `var(i,...)`, where `i` is a numeric or logical vector corresponding to rows of the variable
  - A `size` method with a `dim` argument
  - A `vertcat` method
- 

`A = dataset('File', filename, 'ParamName', Value)` creates dataset array `A` from column-oriented data in the text file specified by the string `filename`. Variables in `A` are of type `double` if data in the corresponding column of the file, following the column header, are entirely numeric; otherwise the variables in `A` are cell arrays of strings. `dataset` converts empty fields to either `NaN` (for a numeric variable) or the empty string (for a string-valued variable). `dataset` ignores insignificant white space in the file. You cannot specify both a file and workspace variables as input. See Name/Value Pairs for more information.

`A = dataset('XLSFile', filename, 'ParamName', Value)` creates dataset array `A` from column-oriented data in the Excel spreadsheet specified by the string `filename`. Variables in `A` are of type `double` if data in the corresponding column of the spreadsheet, following the column header, are entirely numeric; otherwise the variables in `A` are cell arrays of strings. See Name/Value Pairs for more information.

`A = dataset('XPTFile', xptfilename, 'ParamName', Value)` creates a dataset array from a SAS® XPORT format file. Variable names from the XPORT format file are preserved. Numeric data types in the XPORT format file are preserved but all other data types are converted to cell

arrays of strings. The XPORT format allows for 28 missing data types. `dataset` represents these in the file by an upper case letter, '.' or '\_'. `dataset` converts all missing data to NaN values in A. However, if you need the specific missing types you can use the `xptread` function to recover the information. See Name/Value Pairs for more information.

## Parameter Name/Value Pairs

Specify one or more of the following name/value pairs when constructing a dataset:

### VarNames

A cell array `{name_1, ..., name_m}` naming the `m` variables in A with the specified variable names. Names must be valid, unique MATLAB identifier strings. The number of names must equal the number of variables in A. You cannot use the `VarNames` parameter if you provide names for individual variables using `{VAR, name}` pairs. To specify `VarNames` when using a file as input, set `ReadVarNames` to `false`.

### ObsNames

A cell array `{name_1, ..., name_n}` naming the `n` observations in A with the specified observation names. The names need not be valid MATLAB identifier strings, but must be unique. The number of names must equal the number of observations (rows) in A. To specify `ObsNames` when using a file as input, set `ReadObsNames` to `false`.

### Name/value pairs available when using text files as inputs:

#### Delimiter

A string indicating the character separating columns in the file. Values are

- '\t' (tab, the default when no format is specified)
- ' ' (space, the default when a format is specified)
- ',' (comma)

# dataset

---

- ';' (semicolon)
- '|' (bar)

## **Format**

A format string, as accepted by `textscan`. `dataset` reads the file using `textscan`, and creates variables in `A` according to the conversion specifiers in the format string. You may also provide any name/value pairs accepted by `textscan`. Using the `Format` parameter is much faster for large files. If `ReadObsNames` is `true`, the format string should include a format specifier for the first column of the file.

## **HeaderLines**

Numeric value indicating the number of lines to skip at the beginning of a file.

**Default:** 0

## **TreatAsEmpty**

Specifies strings to treat as the empty string in a numeric column. Values may be a character string or a cell array of strings. The parameter applies only to numeric columns in the file; `dataset` does not accept numeric literals such as `'-99'`.

## **Name/value pairs available when using text files or Excel spreadsheets as inputs:**

### **ReadVarNames**

A logical value indicating whether (`true`) or not (`false`) to read variable names from the first row of the file. The default is `true`. If `ReadVarNames` is `true`, variable names in the column headers of the file or range (if using an Excel spreadsheet) cannot be empty.

### **ReadObsNames**

A logical value indicating whether (`true`) or not (`false`) to read observation names from the first column of the file or range (if using an Excel spreadsheet). The default is `false`. If `ReadObsNames` and `ReadVarNames` are both `true`, `dataset` saves the header of the first column in the file or range as the name of the first dimension in `A.Properties.DimNames`.

When reading from an XPT format file, the `ReadObsNames` parameter name/value pair determines whether or not to try to use the first variable in the file as observation names. Specify as a logical value (default `false`). If the contents of the first variable are not valid observation names then `dataset` reads the variable into a variable of the dataset array and does not set the observation names.

### **Name/value pairs available when using Excel spreadsheets as input:**

#### **Sheet**

A positive scalar value of type `double` indicating the sheet number, or a quoted string indicating the sheet name.

#### **Range**

A string of the form `'C1:C2'` where `C1` and `C2` are the names of cells at opposing corners of a rectangular region to be read, as for `xlsread`. By default, the rectangular region extends to the right-most column containing data. If the spreadsheet contains empty columns between columns of data, or if the spreadsheet contains figures or other non-tabular information, specify a range that contains only data.

## **Examples**

Create a dataset array from workspace variables, including observation names:

```
load cereal
cereal = dataset(Calories,Protein,Fat,Sodium,Fiber,Carbo,...
    Sugars,'ObsNames',Name)
```

# dataset

---

```
cereal.Properties.VarDescription = Variables(4:10,2);
```

---

Create a dataset array from a single, multi-columned workspace variable, designating variable names for each column:

```
load cities
categories = cellstr(categories);
cities = dataset({ratings, categories{:}}, ...
    'ObsNames', cellstr(names))
```

---

Load data from a text or spreadsheet file

```
patients = dataset('File', 'hospital.dat', ...
    'Delimiter', ',', 'ReadObsNames', true)
patients2 = dataset('XLSFile', 'hospital.xls', ...
    'ReadObsNames', true)
```

---

- 1 Load patient data from the CSV file `hospital.dat` and store the information in a `dataset` array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file', 'hospital.dat', ...
    'format', '%S%S%S%f%f%f%f%f%f%f', ...
    'Delimiter', ',', 'ReadObsNames', true);
```

You can also load the data without specifying a format string. `dataset` will automatically create dataset variables that are either double arrays or cell arrays of strings, depending on the contents of the file:

```
patients = dataset('file', 'hospital.dat', ...
    'delimiter', ',', ...
    'ReadObsNames', true);
```



- 2 Make the {0,1}-valued variable `smoke` nominal, and change the labels to 'No' and 'Yes':

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

- 3 Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...  
                           {'0-5 Years','5-10 Years','LongTerm'});
```

- 4 Assuming the nonsmokers have never smoked, relabel the 'No' level:

```
patients.smoke = setlabels(patients.smoke,'Never','No');
```

- 5 Drop the undifferentiated 'Yes' level from `smoke`:

```
patients.smoke = droplevels(patients.smoke,'Yes');
```

Warning: OLDLEVELS contains categorical levels that were present in A, caused some array elements to have undefined levels.

Note that smokers now have an undefined level.

- 6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

## See Also

[cell2dataset](#) | [mat2dataset](#) | [struct2dataset](#) | [tdfread](#) | [textscan](#) | [xlsread](#)

## Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Create a Dataset Array from a File” on page 2-71
- “Dataset Arrays in the Variables Editor” on page 2-122

## Concepts

- “Dataset Arrays” on page 2-135

# dataset.dataset2cell

---

**Purpose** Convert dataset array to cell array

**Syntax** `C = dataset2cell(D)`

**Description** `C = dataset2cell(D)` converts the dataset array `D` to a cell array `C`. Each variable of `D` becomes a column in `C`. If `D` is an `M`-by-`N` array, then `C` is `(M+1)`-by-`N`, with the variable names of `D` in the first row. If `D` contains observation names, then `C` is `(M+1)`-by-`(N+1)`, with the observation names in the first column.

**See Also** `dataset` | `cell2dataset` | `dataset.export`

**Concepts**

- “Dataset Arrays” on page 2-135

<b>Purpose</b>	Convert dataset array to structure
<b>Syntax</b>	<pre>S = dataset2struct(D) S = dataset2struct(D, 'AsScalar', true)</pre>
<b>Description</b>	<p><code>S = dataset2struct(D)</code> converts a dataset array to a structure array. Each variable of <code>D</code> becomes a field in <code>S</code>. If <code>D</code> is an <math>M</math>-by-<math>N</math> dataset array, then <code>S</code> is <math>M</math>-by-1 and has <math>N</math> fields. If <code>D</code> contains observation names, then <code>S</code> contains those names in the additional field <code>ObsNames</code>.</p> <p><code>S = dataset2struct(D, 'AsScalar', true)</code> converts a dataset array to a scalar structure. Each variable of <code>D</code> becomes a field in <code>S</code>. If <code>D</code> is an <math>M</math>-by-<math>N</math> dataset array, then <code>S</code> has <math>N</math> fields, each of which as <math>M</math> rows. If <code>D</code> contains observation names, then <code>S</code> contains those names in the additional field <code>ObsNames</code>.</p>
<b>Input Arguments</b>	<p><b>D</b></p> <p><math>M</math>-by-<math>N</math> dataset array.</p>
<b>Output Arguments</b>	<p><b>s</b></p> <p><math>M</math>-by-1 structure array, with <math>N</math> fields. If the input dataset array contains observation names, then <code>S</code> has an additional field <code>ObsNames</code>.</p> <p>If you specify <code>'AsScalar', true</code>, then <code>S</code> is a scalar structure, with <math>N</math> fields, each with <math>M</math> rows.</p>
<b>Examples</b>	<p><b>Convert Dataset Array to Structure Array</b></p> <p>Load sample dataset array.</p> <pre>load('hospital')</pre> <p>Create a dataset array, <code>D</code>, that has only a subset of the observations and variables.</p> <pre>D = hospital(1:8, {'LastName', 'Sex', 'Age'});</pre>

# dataset.dataset2struct

---

```
size(D)
```

```
ans =
```

```
      8      3
```

The dataset array `D` has 8 observations and 3 variables.

Convert `D` to a structure array.

```
S = dataset2struct(D)
```

```
S =
```

```
8x1 struct array with fields:
```

```
  ObsNames
```

```
  LastName
```

```
  Sex
```

```
  Age
```

The structure is `8x1`, corresponding to the 8 observations in the dataset array. `S` also has the field `ObsNames`, since `D` had observation names.

Display the field data for the first element of `S`.

```
S(1)
```

```
ans =
```

```
  ObsNames: 'YPL-320'
```

```
  LastName: 'SMITH'
```

```
    Sex: [1x1 nominal]
```

```
    Age: 38
```

This information corresponds to the first observation (row) of the dataset array.

## Convert Dataset Array to Scalar Structure

Load sample dataset array.

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

Create a dataset array, D, that has only a subset of the observations and variables.

```
D = hospital(1:8, {'LastName', 'Sex', 'Age'});  
size(D)
```

```
ans =
```

```
      8      3
```

The dataset array D has 8 observations and 3 variables.

Convert D to a scalar structure array.

```
S = dataset2struct(D, 'AsScalar', true)
```

```
S =
```

```
    ObsNames: {8x1 cell}  
    LastName: {8x1 cell}  
           Sex: [8x1 nominal]  
           Age: [8x1 double]
```

The data in the fields of the scalar structure is 8x1, corresponding to the 8 observations in the dataset array. S also has the field `ObsNames`, since D had observation names.

Display the data for the field `LastName`.

```
S.LastName
```

```
ans =
```

# dataset.dataset2struct

---

```
'SMITH'  
'JOHNSON'  
'WILLIAMS'  
'JONES'  
'BROWN'  
'DAVIS'  
'MILLER'  
'WILSON'
```

The structure field `LastName` contains all of the data that was in the original dataset array variable, `LastName`.

## See Also

`dataset` | `dataset2cell` | `struct2dataset`

## Concepts

- “Dataset Arrays” on page 2-135

**Purpose** Convert dataset array to table

**Syntax** `t = dataset2table(ds)`

**Description** `t = dataset2table(ds)` converts a dataset array to a table.

**Input Arguments** **ds - Input dataset array**  
dataset array

Input dataset array to convert to a table, specified as a dataset array. Each variable in `ds` becomes a variable in the output table `t`.

**Output Arguments** **t - Output table**  
table

Output table, returned as a table. The table can store metadata such as descriptions, variable units, variable names, and row names. For more information, see [Table Properties](#).

**Examples** **Convert a Dataset Array to a Table**

Load the sample data, which contains nutritional information for 77 cereals.

```
load cereal;
```

Create a dataset array containing the calorie, protein, fat, and name data for the first five cereals. Label the variables.

```
Calories = Calories(1:5);  
Protein = Protein(1:5);  
Fat = Fat(1:5);  
Name = Name(1:5);
```

```
cereal = dataset(Calories,Protein,Fat,'ObsNames',Name)  
cereal.Properties.VarDescription = Variables(4:6,2);
```

```
cereal =
```

# dataset2table

---

	Calories	Protein	Fat
100% Bran	70	4	1
100% Natural Bran	120	3	5
All-Bran	70	4	1
All-Bran with Extra Fiber	50	4	0
Almond Delight	110	2	2

Convert the dataset array to a table.

```
t = dataset2table(cereal)
```

```
t =
```

	Calories	Protein	Fat
	-----	-----	---
100% Bran	70	4	1
100% Natural Bran	120	3	5
All-Bran	70	4	1
All-Bran with Extra Fiber	50	4	0
Almond Delight	110	2	2

## See Also

[table](#) | [dataset](#)

## Concepts

- “Array Dimensions”
- “Dataset Arrays” on page 2-135



## Purpose

Apply function to dataset array variables

## Syntax

```
b = datasetfun(fun,A)
[b,c,...] = datasetfun(fun,A)
[b,...] = datasetfun(fun,A,...,'UniformOutput',false)
[b,...] = datasetfun(fun,A,...,'DatasetOutput',true)
[b,...] = datasetfun(fun,A,...,'DataVars',vars)
[b,...] = datasetfun(fun,A,...,'ObsNames',obsnames)
[b,...] = datasetfun(fun,A,...,'ErrorHandler',efun)
```

## Description

`b = datasetfun(fun,A)` applies the function specified by `fun` to each variable of the dataset array `A`, and returns the results in the vector `b`. The  $i$ th element of `b` is equal to `fun` applied to the  $i$ th dataset variable of `A`. `fun` is a function handle to a function that takes one input argument and returns a scalar value. `fun` must return values of the same class each time it is called, and `datasetfun` concatenates them into the vector `b`. The outputs from `fun` must be one of the following types: numeric, logical, character, structure, or cell.

To apply functions that return results that are nonscalar or of different sizes and types, use the `'UniformOutput'` or `'DatasetOutput'` parameters described below.

Do not rely on the order in which `datasetfun` computes the elements of `b`, which is unspecified.

If `fun` is bound to more than one built-in function or file, (that is, if it represents a set of overloaded functions), `datasetfun` follows MATLAB dispatching rules in calling the function. (See “Function Precedence Order”.)

`[b,c,...] = datasetfun(fun,A)`, where `fun` is a function handle to a function that returns multiple outputs, returns vectors `b`, `c`, ..., each corresponding to one of the output arguments of `fun`. `datasetfun` calls `fun` each time with as many outputs as there are in the call to `datasetfun`. `fun` may return output arguments having different classes, but the class of each output must be the same each time `fun` is called.

# dataset.datasetfun

---

`[b,...] = datasetfun(fun,A,...,'UniformOutput',false)` allows you to specify a function `fun` that returns values of different sizes or types. `datasetfun` returns a cell array (or multiple cell arrays), where the  $i$ th cell contains the value of `fun` applied to the  $i$ th dataset variable of `A`. Setting `'UniformOutput'` to `true` is equivalent to the default behavior.

`[b,...] = datasetfun(fun,A,...,'DatasetOutput',true)` specifies that the output(s) of `fun` are returned as variables in a dataset array (or multiple dataset arrays). `fun` must return values with the same number of rows each time it is called, but it may return values of any type. The variables in the output dataset array(s) have the same names as the variables in the input. Setting `'DatasetOutput'` to `false` (the default) specifies that the type of the output(s) from `datasetfun` is determined by `'UniformOutput'`.

`[b,...] = datasetfun(fun,A,...,'DataVars',vars)` allows you to apply `fun` only to the dataset variables in `A` specified by `vars`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

`[b,...] = datasetfun(fun,A,...,'ObsNames',obsnames)` specifies observation names for the dataset output when `'DatasetOutput'` is `true`.

`[b,...] = datasetfun(fun,A,...,'ErrorHandler',efun)`, where `efun` is a function handle, specifies the MATLAB function to call if the call to `fun` fails. The error-handling function is called with the following input arguments:

- A structure with the fields `identifier`, `message`, and `index`, respectively containing the identifier of the error that occurred, the text of the error message, and the linear index into the input array(s) at which the error occurred
- The set of input arguments at which the call to the function failed

The error-handling function should either re-throw an error, or return the same number of outputs as `fun`. These outputs are then returned as

the outputs of `datasetfun`. If `'UniformOutput'` is true, the outputs of the error handler must also be scalars of the same type as the outputs of `fun`. For example, the following code could be saved in a file as the error-handling function:

```
function [A,B] = errorFunc(S,varargin)

warning(S.identifier,S.message);
A = NaN;
B = NaN;
```

If an error-handling function is not specified, the error from the call to `fun` is rethrown.

## Examples

Compute statistics on selected variables in the `hospital` dataset array:

```
load hospital

stats = ...
    datasetfun(@mean,hospital,...
               'DataVars',{'Weight','BloodPressure'},...
               'UniformOutput',false)

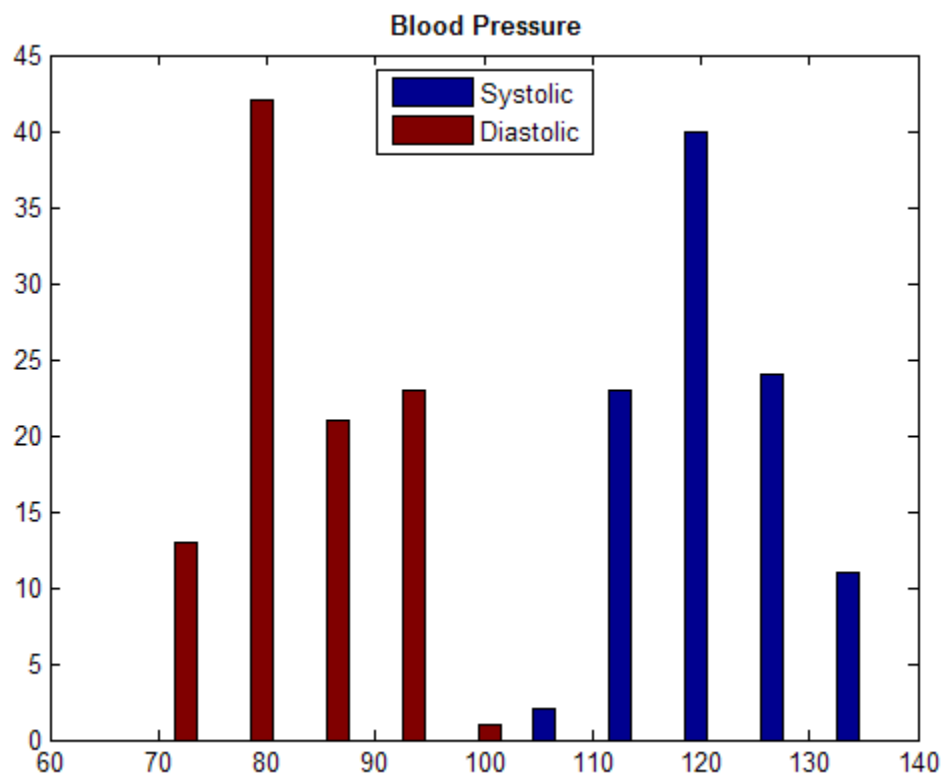
stats =
    [154]    [1x2 double]
stats{2}
ans =
    122.7800    82.9600
```

Display the blood pressure variable:

```
datasetfun(@hist,hospital,...
           'DataVars','BloodPressure',...
           'UniformOutput',false);
title({'\bf Blood Pressure'})
legend('Systolic','Diastolic','Location','N')
```

## dataset.datasetfun

---



**See Also** `grpstats`

**Purpose**

*D*-optimal augmentation

**Syntax**

```
dCE2 = daugment(dCE,mruns)
[dCE2,X] = daugment(dCE,mruns)
[dCE2,X] = daugment(dCE,mruns,model)
[dCE2,X] = daugment(...,param1,val1,param2,val2,...)
```

**Description**

`dCE2 = daugment(dCE,mruns)` uses a coordinate-exchange algorithm to *D*-optimally add `mruns` runs to an existing experimental design `dCE` for a linear additive model.

`[dCE2,X] = daugment(dCE,mruns)` also returns the design matrix `X` associated with the augmented design.

`[dCE2,X] = daugment(dCE,mruns,model)` uses the linear regression model specified in `model`. `model` is one of the following strings:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with  $n$  terms is:

- 1 The constant term
- 2 The linear terms in order 1, 2, ...,  $n$
- 3 The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ )
- 4 The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Alternatively, `model` can be a matrix specifying polynomial terms of arbitrary order. In this case, `model` should have one column for each factor and one row for each term in the model. The entries in any row of `model` are powers for the factors in the columns. For example, if a

# daugment

model has factors X1, X2, and X3, then a row [0 1 2] in *model* specifies the term  $(X1.^0) \cdot (X2.^1) \cdot (X3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

[dCE2,X] = `daugment(...,param1,val1,param2,val2,...)` specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'bounds'	Lower and upper bounds for each factor, specified as a 2-by-nfactors matrix, where nfactors is the number of factors. Alternatively, this value can be a cell array containing nfactors elements, each element specifying the vector of allowable values for the corresponding factor.
'categorical'	Indices of categorical predictors.
'display'	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
'excldefun'	Handle to a function that excludes undesirable runs. If the function is <i>f</i> , it must support the syntax $b = f(S)$ , where <i>S</i> is a matrix of treatments with nfactors columns, where nfactors is the number of factors, and <i>b</i> is a vector of Boolean values with the same number of rows as <i>S</i> . <i>b</i> ( <i>i</i> ) is true if the <i>i</i> th row <i>S</i> should be excluded.
'init'	Initial design as an mruns-by-nfactors matrix, where nfactors is the number of factors. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.
'maxiter'	Maximum number of iterations. The default is 10.

Parameter	Value
'options'	<p>The value is a structure that contains options specifying whether to compute multiple tries in parallel, and specifying how to use random numbers when generating the starting points for the tries. Create the options structure with <code>statset</code>. Applicable <code>statset</code> parameters are:</p> <ul style="list-style-type: none"> <li>• 'UseParallel' — If true and if a parpool of the Parallel Computing Toolbox is open, compute in parallel. If the Parallel Computing Toolbox is not installed, or a parpool is not open, computation occurs in serial mode. Default is false, meaning serial computation.</li> <li>• UseSubstreams — Set to true to compute in parallel in a reproducible fashion. Default is false. To compute reproducibly, set Streams to a type allowing substreams: 'mlfg6331_64' or 'mrg32k3a'.</li> <li>• Streams — A RandStream object or cell array of such objects. If you do not specify Streams, daugment uses the default stream or streams. If you choose to specify Streams, use a single object except in the case             <ul style="list-style-type: none"> <li>▪ You have an open Parallel pool</li> <li>▪ UseParallel is true</li> <li>▪ UseSubstreams is false</li> </ul>             In that case, use a cell array the same size as the Parallel pool.           </li> </ul>
'tries'	<p>Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.</p>

---

**Note** The `daugment` function augments an existing design using a coordinate-exchange algorithm; the `'start'` parameter of the `candexch` function provides the same functionality using a row-exchange algorithm.

---

## Examples

The following eight-run design is adequate for estimating main effects in a four-factor model:

```
dCEmain = cordexch(4,8)
dCEmain =
    1    -1    -1     1
   -1    -1     1     1
   -1     1    -1     1
    1     1     1    -1
    1     1     1     1
   -1     1    -1    -1
    1    -1    -1    -1
   -1    -1     1    -1
```

To estimate the six interaction terms in the model, augment the design with eight additional runs:

```
dCEinteraction = daugment(dCEmain,8,'interaction')
dCEinteraction =
    1    -1    -1     1
   -1    -1     1     1
   -1     1    -1     1
    1     1     1    -1
    1     1     1     1
   -1     1    -1    -1
    1    -1    -1    -1
   -1    -1     1    -1
   -1     1     1     1
   -1    -1    -1    -1
    1    -1     1    -1
```



1	1	-1	1
-1	1	1	-1
1	1	-1	-1
1	-1	1	1
1	1	1	-1

The augmented design is full factorial, with the original eight runs in the first eight rows.

## See Also

[dcovary](#) | [cordexch](#) | [candexch](#)

**Purpose** *D*-optimal design with fixed covariates

**Syntax**

```
dCV = dcovary(nfactors, fixed)
[dCV, X] = dcovary(nfactors, fixed)
[dCV, X] = dcovary(nfactors, fixed, model)
[dCV, X] = daugment(..., param1, val1, param2, val2, ...)
```

**Description** `dCV = dcovary(nfactors, fixed)` uses a coordinate-exchange algorithm to generate a *D*-optimal design for a linear additive model with `nfactors` factors, subject to the constraint that the model include the fixed covariate factors in `fixed`. The number of runs in the design is the number of rows in `fixed`. The design `dCV` augments `fixed` with initial columns for treatments of the model terms.

`[dCV, X] = dcovary(nfactors, fixed)` also returns the design matrix `X` associated with the design.

`[dCV, X] = dcovary(nfactors, fixed, model)` uses the linear regression model specified in `model`. `model` is one of the following strings:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of `X` for a full quadratic model with  $n$  terms is:

- 1** The constant term
- 2** The linear terms in order 1, 2, ...,  $n$
- 3** The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ )
- 4** The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.

Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors *X1*, *X2*, and *X3*, then a row [0 1 2] in *model* specifies the term  $(X1.^0) \cdot (X2.^1) \cdot (X3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

[dCV,X] = daugment(...,param1,val1,param2,val2,...) specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'bounds'	Lower and upper bounds for each factor, specified as a 2-by-nfactors matrix. Alternatively, this value can be a cell array containing nfactors elements, each element specifying the vector of allowable values for the corresponding factor.
'categorical'	Indices of categorical predictors.
'display'	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
'excludefun'	Handle to a function that excludes undesirable runs. If the function is <i>f</i> , it must support the syntax $b = f(S)$ , where <i>S</i> is a matrix of treatments with nfactors columns and <i>b</i> is a vector of Boolean values with the same number of rows as <i>S</i> . <i>b</i> ( <i>i</i> ) is true if the <i>i</i> th row <i>S</i> should be excluded.
'init'	Initial design as an mruns-by-nfactors matrix. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.
'maxiter'	Maximum number of iterations. The default is 10.

Parameter	Value
'options'	<p>The value is a structure that contains options specifying whether to compute multiple tries in parallel, and specifying how to use random numbers when generating the starting points for the tries. Create the options structure with <code>statset</code>. Applicable <code>statset</code> parameters are:</p> <ul style="list-style-type: none"> <li>• <code>'UseParallel'</code> — If true and if a parpool of the Parallel Computing Toolbox is open, compute in parallel. If the Parallel Computing Toolbox is not installed, or a parpool is not open, computation occurs in serial mode. Default is false, meaning serial computation.</li> <li>• <code>UseSubstreams</code> — Set to true to compute in parallel in a reproducible fashion. Default is false. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>dcovary</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>▪ You have an open Parallel pool</li> <li>▪ <code>UseParallel</code> is true</li> <li>▪ <code>UseSubstreams</code> is false</li> </ul> In that case, use a cell array the same size as the Parallel pool.</li> </ul>
'tries'	<p>Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.</p>

## Examples

### Example 1

Suppose you want a design to estimate the parameters in a three-factor linear additive model, with eight runs that necessarily occur at different times. If the process experiences temporal linear drift, you may want to include the run time as a variable in the model. Produce the design as follows:

```
time = linspace(-1,1,8)';
[dCV1,X] = dcovary(3,time,'linear')
dCV1 =
-1.0000    1.0000    1.0000   -1.0000
 1.0000   -1.0000   -1.0000   -0.7143
-1.0000   -1.0000   -1.0000   -0.4286
 1.0000   -1.0000    1.0000   -0.1429
 1.0000    1.0000   -1.0000    0.1429
-1.0000    1.0000   -1.0000    0.4286
 1.0000    1.0000    1.0000    0.7143
-1.0000   -1.0000    1.0000    1.0000
X =
 1.0000   -1.0000    1.0000    1.0000   -1.0000
 1.0000    1.0000   -1.0000   -1.0000   -0.7143
 1.0000   -1.0000   -1.0000   -1.0000   -0.4286
 1.0000    1.0000   -1.0000    1.0000   -0.1429
 1.0000    1.0000    1.0000   -1.0000    0.1429
 1.0000   -1.0000    1.0000   -1.0000    0.4286
 1.0000    1.0000    1.0000    1.0000    0.7143
 1.0000   -1.0000   -1.0000    1.0000    1.0000
```

The column vector `time` is a fixed factor, normalized to values between  $\pm 1$ . The number of rows in the fixed factor specifies the number of runs in the design. The resulting design `dCV` gives factor settings for the three controlled model factors at each time.

### Example 2

The following example uses the `dummyvar` function to block an eight-run experiment into 4 blocks of size 2 for estimating a linear additive model with two factors:

# dcovary

---

```
fixed = dummyvar([1 1 2 2 3 3 4 4]);
dCV2 = dcovary(2, fixed(:, 1:3), 'linear')
dCV2 =
     1     1     1     0     0
    -1    -1     1     0     0
    -1     1     0     1     0
     1    -1     0     1     0
     1     1     0     0     1
    -1    -1     0     0     1
    -1     1     0     0     0
     1    -1     0     0     0
```

The first two columns of dCV2 contain the settings for the two factors; the last three columns are dummy variable codings for the four blocks.

## See Also

[daugment](#) | [cordexch](#) | [dummyvar](#)

# TreeBagger.DefaultYfit property

---

## Purpose

Default value returned by `predict` and `oobPredict`

## Description

The `DefaultYfit` property controls what predicted value `TreeBagger` returns when no prediction is possible, for example when the `oobPredict` method needs to predict for an observation that is in-bag for all trees in the ensemble.

For classification, you can set this property to either `' '` or `'MostPopular'`. If you choose `'MostPopular'` (default), the property value becomes the name of the most probable class in the training data.

For regression, you can set this property to any numeric scalar. The default is the mean of the response for the training data.

If you set this property to `' '` for classification or `NaN` for regression, `TreeBagger` excludes the in-bag observations from computation of the out-of-bag error and margin.

## See Also

`oobPredict` | `Predict` | `OOBIndices`

# grandstream.delete

---

**Purpose** Delete handle object

**Syntax** `delete(h)`

**Description** `delete(h)` deletes the handle object `h`, where `h` is a scalar handle. The `delete` method deletes a handle object but does not clear the handle from the workspace. A deleted handle is no longer valid.

**See Also** `clear` | `isvalid` | `grandstream`



# TreeBagger.DeltaCritDecisionSplit property

---

- Purpose** Split criterion contributions for each predictor
- Description** The DeltaCritDecisionSplit property is a numeric array of size 1-by-Nvars of changes in the split criterion summed over splits on each variable, averaged across the entire ensemble of grown trees.
- See Also** `classregtree.varimportance`

# dendrogram

---

## Purpose

Dendrogram plot

## Syntax

```
dendrogram(tree)
dendrogram(tree,Name,Value)
```

```
dendrogram(tree,P)
dendrogram(tree,P,Name,Value)
```

```
H = dendrogram( ___ )
[H,T,outperm] = dendrogram( ___ )
```

## Description

`dendrogram(tree)` generates a dendrogram plot of the hierarchical binary cluster tree. A dendrogram consists of many *U*-shaped lines that connect data points in a hierarchical tree. The height of each *U* represents the distance between the two data points being connected.

- If there are 30 or fewer data points in the original data set, then each leaf in the dendrogram corresponds to one data point.
- If there are more than 30 data points, then `dendrogram` collapses lower branches so that there are 30 leaf nodes. As a result, some leaves in the plot correspond to more than one data point.

`dendrogram(tree,Name,Value)` uses additional options specified by one or more name-value pair arguments.

`dendrogram(tree,P)` generates a dendrogram plot with no more than *P* leaf nodes. If there are more than *P* data points in the original data set, then `dendrogram` collapses the lower branches of the tree. As a result, some leaves in the plot correspond to more than one data point.

`dendrogram(tree,P,Name,Value)` uses additional options specified by one or more name-value pair arguments.

`H = dendrogram( ___ )` generates a dendrogram plot and returns a vector of line handles. You can use any of the input arguments from the previous syntaxes.

`[H,T,outperm] = dendrogram( ___ )` also returns a vector containing the leaf node number for each object in the original data set, `T`, and a vector giving the order of the node labels of the leaves as shown in the dendrogram, `outperm`.

- It is useful to return `T` when the number of leaf nodes, `P`, is less than the total number of data points, so that some leaf nodes in the display correspond to multiple data points.
- The order of the node labels given in `outperm` is from left to right for a horizontal dendrogram, and from bottom to top for a vertical dendrogram.

## Input Arguments

### **tree - Hierarchical binary cluster tree**

matrix returned by `linkage`

Hierarchical binary cluster tree, specified as an  $(M - 1)$ -by-3 matrix that you generate using `linkage`, where  $M$  is the number of data points in the original data set.

### **P - Maximum number of leaf nodes**

30 (default) | positive integer value

Maximum number of leaf nodes to include in the dendrogram plot, specified as a positive integer value.

- If there are `P` or fewer data points in the original data set, then each leaf in the dendrogram corresponds to one data point.
- If there are more than `P` data points, then `dendrogram` collapses lower branches so that there are `P` leaf nodes. As a result, some leaves in the plot correspond to more than one data point.

If you do not specify `P`, then `dendrogram` uses 30 as the maximum number of leaf nodes. To display the complete tree, set `P` equal to 0.

# dendrogram

---

## Data Types

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'Orientation','left','Reorder',myOrder`, specifies a vertical dendrogram with leaves in the order specified by `myOrder`.

## 'Reorder' - Order of leaf nodes

vector

Order of leaf nodes in the dendrogram plot, specified as the comma-separated pair consisting of `'Reorder'` and a vector giving the order of nodes in the complete tree. The order vector must be a permutation of the vector `1:M`, where  $M$  is the number of data points in the original data set. Specify the order from left to right for horizontal dendrograms, and from bottom to top for vertical dendrograms.

If  $M$  is greater than the number of leaf nodes in the dendrogram plot, `P` (by default, `P` is 30), then you can only specify a permutation vector that does not separate the groups of leaves that correspond to collapsed nodes.

**Example:**

## Data Types

single | double

## 'CheckCrossing' - Indicator for whether to check for crossing branches

true (default) | false

Indicator for whether to check for crossing branches in the dendrogram plot, specified as the comma-separated pair consisting of

'CheckCrossing' and either true or false. This option is only useful when you specify a value for Reorder.

When CheckCrossing has the value true, dendrogram issues a warning if the order of the leaf nodes causes crossing branches in the plot. If the dendrogram plot does not show a complete tree (because the number of data points in the original data set is greater than P), dendrogram only issues a warning when the order of the leaf nodes causes branch to cross in the dendrogram as shown in the plot. That is, there is no warning if the order causes crossing branches in the complete tree but not in the dendrogram as shown in the plot.

## Data Types

logical

## 'ColorThreshold' - Threshold for unique colors

'default' | scalar value in the range (0,max(tree(:,3)))

Threshold for unique colors in the dendrogram plot, specified as the comma-separated pair consisting of 'ColorThreshold' and either the string 'default' or a scalar value in the range (0,max(tree(:,3))). If ColorThreshold has the value  $T$ , then dendrogram assigns a unique color to each group of nodes in the dendrogram whose linkage is less than  $T$ .

- If ColorThreshold has the value 'default', then the threshold,  $T$ , is 70% of the maximum linkage,  $0.7*\max(\text{tree}(:,3))$ .
- If you do not specify a value for ColorThreshold, or if you specify a threshold outside the range (0,max(tree(:,3))), then dendrogram uses only one color for the dendrogram plot.

## 'Orientation' - Orientation of dendrogram

'top' (default) | 'bottom' | 'left' | 'right'

Orientation of the dendrogram in the figure window, specified as the comma-separated pair consisting of 'Orientation' and one of these strings:

# dendrogram

---

'top'	Top to bottom
'bottom'	Bottom to top
'left'	Left to right
'right'	Right to left

## Data Types

char

## 'Labels' - Label for each data point

character array | cell array of strings

Label for each data point in the original data set, specified as the comma-separated pair consisting of 'Labels' and a character array or cell array of strings. `dendrogram` labels any leaves in the dendrogram plot containing a single data point with that data point's label.

## Data Types

char | cell

## Output Arguments

### H - Handles to lines

vector

Handles to lines in the dendrogram plot, returned as a vector.

### T - Leaf node numbers

column vector

Leaf node numbers for each data point in the original data set, returned as a column vector of length  $M$ , where  $M$  is the number of data points in the original data set.

When there are fewer than  $P$  data points in the original data ( $P$  is 30, by default), all data points are displayed in the dendrogram, with each node containing a single data point. In this case,  $T$  is the identity map,  $T = (1:M)'$ .

$T$  is useful when  $P$  is less than the total number of data points. That is, when some leaf nodes in the dendrogram display correspond to multiple

data points. For example, to find out which data points are contained in leaf node  $k$  of the dendrogram plot, use `find(T==k)`.

## **outperm - Permutation of node labels**

vector

Permutation of the node labels of the leaves of the dendrogram as shown in the plot, returned as a row vector. `outperm` gives the order from left to right for a horizontal dendrogram, and from bottom to top for a vertical dendrogram. If there are  $P$  leaves in the dendrogram plot, `outperm` is a permutation of the vector `1:P`.

## **Examples**

### **Plot Dendrogram**

Generate sample data.

```
rng('default') % For reproducibility
X = rand(10,3);
```

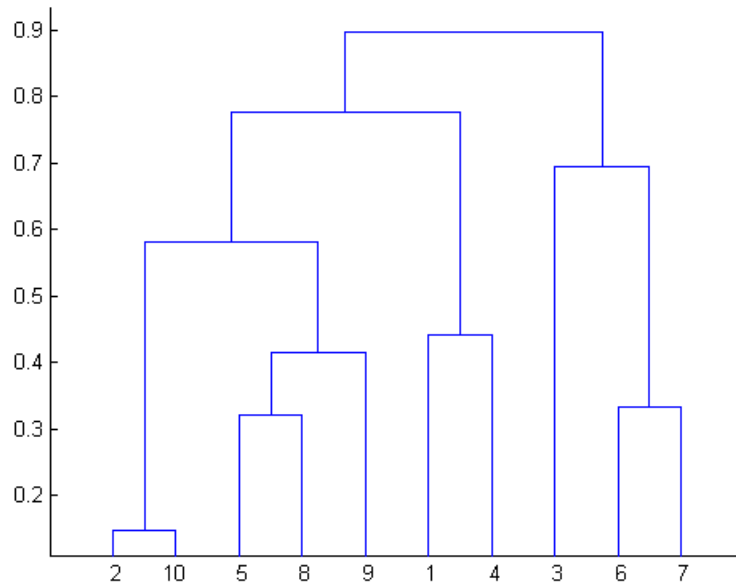
Create a hierarchical binary cluster tree using `linkage`. Then, plot the dendrogram using the default options.

```
tree = linkage(X,'average');
```

```
figure()
dendrogram(tree)
```

# dendrogram

---



## Specify Dendrogram Leaf Node Order

Generate sample data.

```
rng('default') % For reproducibility  
X = rand(10,3);
```

Create a hierarchical binary cluster tree using linkage.

```
tree = linkage(X,'average');
```

```
D = pdist(X);  
leafOrder = optimalleaforder(tree,D)
```

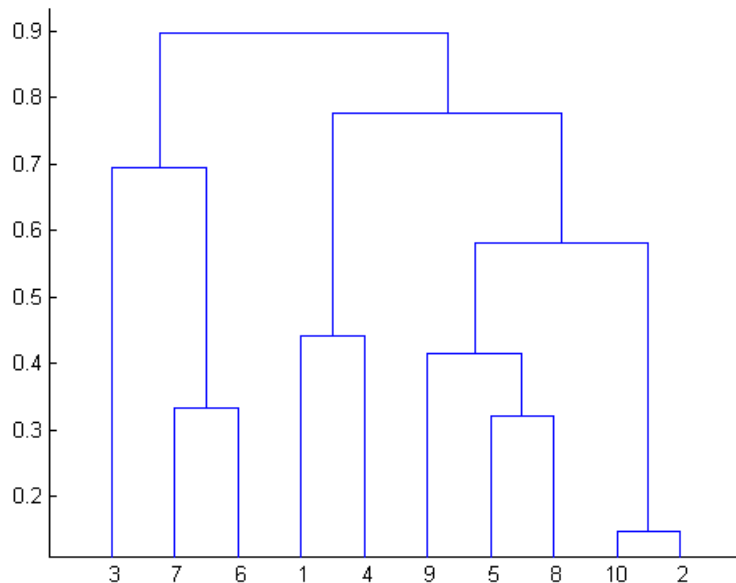
```
leafOrder =
```



```
3 7 6 1 4 9 5 8 10 2
```

Plot the dendrogram using an optimal leaf order.

```
figure()  
dendrogram(tree, 'Reorder', leafOrder)
```



The order of the leaf nodes in the dendrogram plot corresponds—from left to right—to the permutation in `leafOrder`.

## Specify Number of Nodes in Dendrogram Plot

Generate sample data.

```
rng('default') % For reproducibility  
X = rand(100,2);
```

# dendrogram

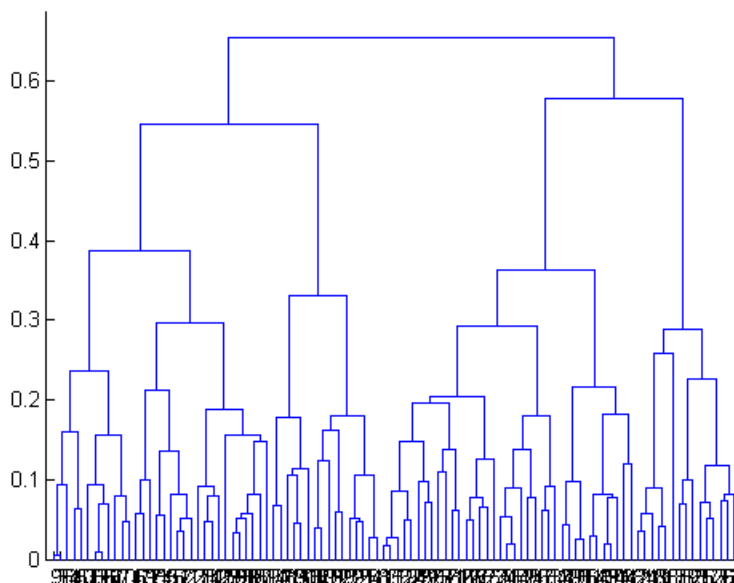
---

There are 100 data points in the original data set, X.

Create a hierarchical binary cluster tree using `linkage`. Then, plot the dendrogram for the complete tree (100 leaf nodes) by setting the input argument `P` equal to 0.

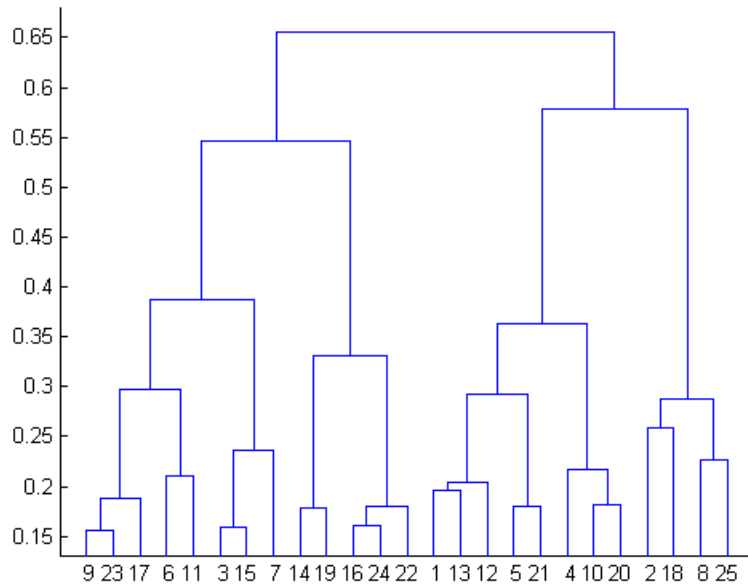
```
tree = linkage(X, 'average');
```

```
figure()  
dendrogram(tree, 0)
```



Now, plot the dendrogram with only 25 leaf nodes. Return the mapping of the original data points to the leaf nodes shown in the plot.

```
figure()  
[~,T] = dendrogram(tree, 25);
```



List the original data points that are in leaf node 7 of the dendrogram plot.

```
find(T==7)
```

```
ans =
```

```
7  
33  
60  
70  
74  
76  
86
```

# dendrogram

---

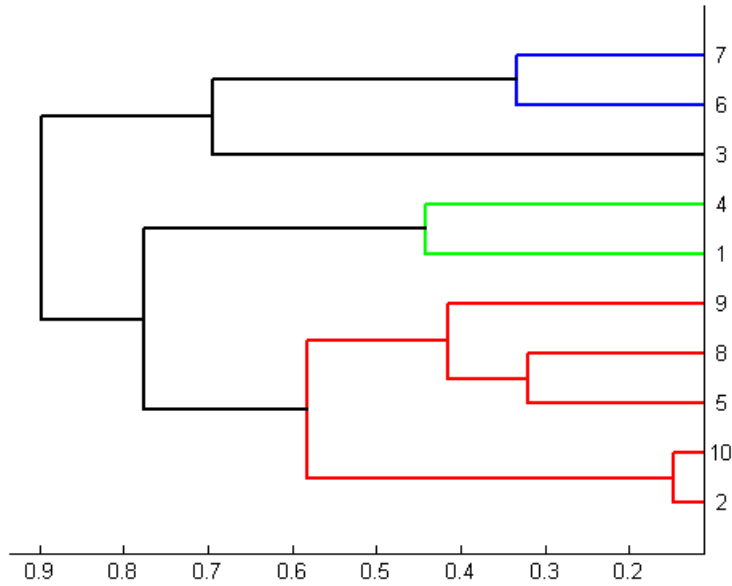
## Change Dendrogram Orientation and Line Width

Generate sample data.

```
rng('default') % For reproducibility  
X = rand(10,3);
```

Create a hierarchical binary cluster tree using linkage. Then, plot the dendrogram with a vertical orientation, using the default color threshold. Return handles to the lines so you can change the dendrogram line widths.

```
tree = linkage(X,'average');  
  
figure()  
H = dendrogram(tree,'Orientation','left','ColorThreshold','default');  
set(H,'LineWidth',2)
```



## See Also

[cluster](#) | [clusterdata](#) | [cophenet](#) | [inconsistent](#) | [linkage](#) | [pdist](#) | [silhouette](#)

# dataset.Description property

---

**Purpose** String describing data set

**Description** Description is a string describing the data set. The default is an empty string.

# GeneralizedLinearModel.devianceTest

---

<b>Purpose</b>	Analysis of deviance				
<b>Syntax</b>	<code>tbl = devianceTest(md1)</code>				
<b>Description</b>	<code>tbl = devianceTest(md1)</code> returns an analysis of deviance table for the <code>md1</code> generalized linear model. <code>tbl</code> gives the result of a test of whether the fitted model fits significantly better than a constant model.				
<b>Input Arguments</b>	<b>md1</b> Generalized linear model, as constructed by <code>fitglm</code> or <code>stepwiseglm</code> .				
<b>Output Arguments</b>	<b>tbl</b> Dataset array containing two rows and four columns. <ul style="list-style-type: none"><li>• The first row relates to a constant model.</li><li>• The second row relates to the full model in <code>md1</code>.</li><li>• The columns are:<table><tr><td>Deviance</td><td>Deviance is twice the difference between the log likelihoods of the corresponding model (<code>md1</code> or constant) and the saturated model. The test statistic for the deviance test is twice the difference between the log likelihoods of the tested model <code>md1</code> and the constant model. For more information, see Deviance.</td></tr><tr><td>DFE</td><td>Error degrees of freedom. It is the number of observations minus the</td></tr></table></li></ul>	Deviance	Deviance is twice the difference between the log likelihoods of the corresponding model ( <code>md1</code> or constant) and the saturated model. The test statistic for the deviance test is twice the difference between the log likelihoods of the tested model <code>md1</code> and the constant model. For more information, see Deviance.	DFE	Error degrees of freedom. It is the number of observations minus the
Deviance	Deviance is twice the difference between the log likelihoods of the corresponding model ( <code>md1</code> or constant) and the saturated model. The test statistic for the deviance test is twice the difference between the log likelihoods of the tested model <code>md1</code> and the constant model. For more information, see Deviance.				
DFE	Error degrees of freedom. It is the number of observations minus the				

# GeneralizedLinearModel.devianceTest

---

chi2Stat

number of parameters in the corresponding model.

F statistic or Chi-squared statistic, depending on whether the dispersion is estimated (F statistic) or not (Chi-squared statistic)

- *Chi-squared statistic* is the difference between the deviance of the constant model and the deviance of the full model.

- *F statistic* is the difference between the deviance of the constant model and the deviance of the full model, divided by the estimated dispersion.

pValue

*p*-value associated with the test. It is the Chi-squared statistic with (number of coefficients in the model minus one) degrees of freedom, or *F* statistic with (number of coefficients in the model minus one) numerator degrees of freedom, and DFE denominator degrees of freedom.

## Definitions

### Deviance

Deviance of a model  $M_1$  is twice the difference between the loglikelihood of that model and the saturated model,  $M_S$ . The saturated model is the model with the maximum number of parameters that can be estimated.



For example, if there are  $n$  observations  $y_i, i = 1, 2, \dots, n$ , with potentially different values for  $X_i^T\beta$ , then you can define a saturated model with  $n$  parameters. Let  $L(b,y)$  denote the maximum value of the likelihood function for a model. Then the deviance of model  $M_1$  is

$$-2(\log L(b_1, y) - \log L(b_S, y)),$$

where  $b_1$  are the estimated parameters for model  $M_1$  and  $b_S$  are the estimated parameters for the saturated model. The deviance has a chi-square distribution with  $n - p$  degrees of freedom, where  $n$  is the number of parameters in the saturated model and  $p$  is the number of parameters in model  $M_1$ .

If  $M_1$  and  $M_2$  are two different generalized linear models, then the fit of the models can be assessed by comparing the deviances  $D_1$  and  $D_2$  of these models. The difference of the deviances is

$$\begin{aligned} D = D_2 - D_1 &= -2(\log L(b_2, y) - \log L(b_S, y)) + 2(\log L(b_1, y) - \log L(b_S, y)) \\ &= -2(\log L(b_2, y) - \log L(b_1, y)). \end{aligned}$$

Asymptotically, this difference has a chi-square distribution with degrees of freedom  $v$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using `1 - chi2cdf(D, V)`, where  $D = D_2 - D_1$ .

## Examples

### Deviance Test

Perform a deviance test on a generalized linear model.

Construct a generalized linear model.

```
rng('default') % for reproducibility
X = randn(100,5);
mu = exp(X(:, [1 4 5]) * [.4; .2; .3]);
y = poissrnd(mu);
```

# GeneralizedLinearModel.devianceTest

---

```
mdl = fitglm(X,y,'linear','Distribution','poisson');
```

Test whether the model differs from a constant in a statistically significant way.

```
tbl = devianceTest(mdl)
```

```
tbl =
```

	Deviance	DFE	chi2Stat	pValue
log(y) ~ 1	128.58	99		
log(y) ~ 1 + x1 + x2 + x3 + x4 + x5	83.726	94	44.858	1.5502e-08

The  $p$ -value is very small, indicating that the model significantly differs from a constant.

**See Also** [GeneralizedLinearModel](#) |

**Concepts**

- “Generalized Linear Models” on page 10-12

## Purpose

Fixed- and random-effects design matrices

## Syntax

```
D = designMatrix(lme)
```

```
D = designMatrix(lme, 'Fixed')
```

```
D = designMatrix(lme, 'Random')
```

```
Dsub = designMatrix(lme, 'Random', gnumbers)
```

```
[Dsub, gnames] = designMatrix(lme, 'Random', gnumbers)
```

## Description

`D = designMatrix(lme)` or `D = designMatrix(lme, 'Fixed')` returns the fixed-effects design matrix for the linear mixed-effects model `lme`.

`D = designMatrix(lme, 'Random')` returns the random-effects design matrix for the linear mixed-effects model `lme`.

`Dsub = designMatrix(lme, 'Random', gnumbers)` returns a subset of the random-effects design matrix for the linear mixed-effects model `lme` corresponding to the grouping variables indicated by the integers in `gnumbers`.

`[Dsub, gnames] = designMatrix(lme, 'Random', gnumbers)` also returns the grouping variable names corresponding to the integers in `gnumbers`.

## Input Arguments

### **lme - Linear mixed-effects model**

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

For properties and methods of this object, see LinearMixedModel.

### **gnumbers - Grouping variable numbers**

integer array

# LinearMixedModel.designMatrix

---

Grouping variable numbers, specified as an integer array, where  $R$  is the length of the cell array that contains the grouping variables for the linear mixed-effects model `lme`.

For example, you can specify the grouping variables  $g_1$ ,  $g_3$ , and  $g_r$  as follows.

**Example:** `[1,3,r]`

## Data Types

`double` | `single`

## Output Arguments

### D - Design matrix

`matrix`

Design matrix of a linear mixed-effects model `lme` returned as one of the following:

- Fixed-effects design matrix —  $n$ -by- $p$  matrix consisting of the fixed-effects design of `lme`, where  $n$  is the number of observations and  $p$  is the number of fixed-effects terms.
- Random-effects design matrix —  $n$ -by- $k$  matrix, consisting of the random-effects design matrix of `lme`. Here,  $k$  is equal to `length(B)`, where **B** is the random-effects coefficients vector of linear mixed-effects model `lme`.

If `lme` has  $R$  grouping variables  $g_1, g_2, \dots, g_R$ , with levels  $m_1, m_2, \dots, m_R$ , respectively, and if  $q_1, q_2, \dots, q_R$  are the lengths of the random-effects vectors that are associated with  $g_1, g_2, \dots, g_R$ , respectively, then **B** is a column vector of length  $q_1 * m_1 + q_2 * m_2 + \dots + q_R * m_R$ .

**B** is made by concatenating the best linear unbiased predictors of random-effects vectors corresponding to each level of each grouping variable as `[g1_level1; g1_level2; ...; g1_level_m1; g2_level1; g2_level2; ...; g2_level_m2; ...; gR_level1; gR_level2; ...; gR_level_mR]`'.

## Data Types

single | double

## Dsub - Submatrix of random-effects design matrix

matrix

Submatrix of random-effects design matrix corresponding to the grouping variables indicated by the integers in `gnumbers`, returned as an  $n$ -by- $k$  matrix, where  $k$  is length of the column vector `Bsub`.

`Bsub` contains the concatenated best linear unbiased predictors (BLUPs) of random-effects vectors, corresponding to each level of the grouping variables, specified by `gnumbers`.

If, for example, `gnumbers` is `[1,3,r]`, this corresponds to the grouping variables  $g_1$ ,  $g_3$ , and  $g_r$ . Then, `Bsub` contains the concatenated BLUPs of random-effects vectors corresponding to each level of the grouping variables  $g_1$ ,  $g_3$ , and  $g_r$ , such as

$$[g_{1,level_1}; g_{1,level_2}; \dots; g_{1,level_{m_1}}; g_{3,level_1}; g_{3,level_2}; \dots; g_{3,level_{m_3}}; g_{r,level_1}; g_{r,level_2}; \dots; g_{r,level_{m_r}}]'$$

Thus, `Dsub*Bsub` represents the contribution of all random effects corresponding to grouping variables  $g_1$ ,  $g_3$ , and  $g_r$  to the response of `lme`.

If `gnumbers` is empty, then `Dsub` is the full random-effects design matrix.

## Data Types

single | double

## gnames - Names of grouping variables

$k$ -by-1 cell array

Names of grouping variables corresponding to the integers in `gnumbers` if the design type is 'Random', returned as a  $k$ -by-1 cell array. If the design type is 'Fixed', then `gnames` is an empty matrix `[]`.

# LinearMixedModel.designMatrix

---

## Data Types

cell

## Examples

### Display Fixed- and Random-Effects Design Matrices

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The data shows the deviations from the target quality characteristic measured from the products that 5 operators manufacture during three different shifts, morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Display the fixed-effects design matrix.

```
designMatrix(lme)
```

```
ans =
```

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

The column of 1s represents the constant term in the model. `fitlme` takes the evening shift as the reference group and creates two dummy variables to represent the morning and night shifts, respectively.

Display the random-effects design matrix.

```
designMatrix(lme, 'random')
```

```
ans =
```

```
(1,1)      1
(2,1)      1
(3,1)      1
(4,2)      1
(5,2)      1
(6,2)      1
(7,3)      1
(8,3)      1
(9,3)      1
(10,4)     1
(11,4)     1
(12,4)     1
```

# LinearMixedModel.designMatrix

---

```
(13,5)      1
(14,5)      1
(15,5)      1
```

The first number,  $i$ , in the  $(i,j)$  indices corresponds to the observation number, and  $j$  corresponds to the level of the grouping variable, Operator, i.e., the operator number.

Show the full display of the random-effects design matrix.

```
full(designMatrix(lme, 'random'))
```

```
ans =
```

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

Each column corresponds to a level of the grouping variable, Operator.

## Random-Effects Design Matrix of Multiple Grouping Variables

Navigate to a folder containing sample data.

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



Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Store and examine the full random-effects design matrix.

```
D = full(designMatrix(lme, 'random'));
```

The first three columns of matrix `D` contain the indicator variables `fitlme` creates for the three levels (Loamy, Silty, Sandy, respectively) of the first grouping variable, `Soil`. The next 15 columns contain the indicator variables created for the second grouping variable, `Tomato` nested under `Soil`. These are basically the elementwise products of the dummy variables representing the levels of `Soil` (Loamy, Silty, and Sandy, respectively) and the levels of `Tomato` (Cherry, Grape, Heirloom, Plum, Vine, respectively).

# LinearMixedModel.designMatrix

---

## Subset of the Random-Effects Design Matrix

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds,'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)')
```

Compute the random-effects design matrix for the second grouping variable, and display the first 12 rows.

```
[Dsub,gname] = designMatrix(lme,'random',2);
full(Dsub(1:12,:))
```

```
ans =  
  
      0      0      0      0      0      0      0      0      0      0      1  
      0      0      0      0      0      0      0      0      0      0      1  
      0      0      0      0      0      0      0      0      0      0      1  
      0      0      0      0      0      0      0      0      0      0      1  
      0      1      0      0      0      0      0      0      0      0      0  
      0      1      0      0      0      0      0      0      0      0      0  
      0      1      0      0      0      0      0      0      0      0      0  
      0      0      0      0      0      0      0      0      0      0      0  
      0      0      0      0      0      0      0      1      0      0      0  
      0      0      0      0      0      0      0      1      0      0      0  
      0      0      0      0      0      0      0      1      0      0      0  
      0      0      0      0      0      0      0      1      0      0      0
```

Dsub contains the dummy variables created for the second grouping variable, that is, tomato nested under soil. These are the elementwise products of the dummy variables representing the levels of Soil (Loamy, Silty, Sandy, respectively) and the levels of Tomato (Cherry, Grape, Heirloom, Plum, Vine, respectively).

Display the name of the grouping variable.

```
gname
```

```
gname =
```

```
'Soil:Tomato'
```

## See Also

[LinearMixedModel](#) | [fitted](#) | [fitlmematrix](#)

# dfittool

---

**Purpose** Open Distribution Fitting app

**Syntax**

```
dfittool
dfittool(y)
dfittool(y,cens)
dfittool(y,cens,freq)
dfittool(y,cens,freq,dsname)
```

**Description** `dfittool` opens the Distribution Fitting app or brings focus to the app if it is already open. Use the Distribution Fitting app to import data from the workspace, fit distributions to the data, and display the fitted distribution over plots of the empirical distributions.

`dfittool(y)` displays the Distribution Fitting app and creates a data set with data specified by the vector `y`.

`dfittool(y,cens)` uses the vector `cens` to specify whether the observation `y(j)` is censored, (`cens(j)==1`) and/or observed, exactly (`cens(j)==0`). If `cens` is omitted or empty, no `y` values are censored.

`dfittool(y,cens,freq)` uses the vector `freq` to specify the frequency of each element of `y`. If `freq` is omitted or empty, all `y` values have a frequency of 1.

`dfittool(y,cens,freq,dsname)` creates a data set with the name `dsname` using the data vector `y`, censoring indicator `cens`, and frequency vector `freq`.

**See Also** `mle` | `randtool` | `disttool`

**Purpose**

Number of dimensions

**Description**

Number of dimensions in the point set. The `Dimensions` property of a point set contains a positive integer that indicates the number of dimensions for which the points have values. For example, a point set with `Dimensions=5` produces points that each have five values.

Set this property by specifying the number of dimensions when constructing a new point set. After construction, you cannot change the value. The default number of dimensions is 2.

## dataset.DimNames property

---

**Purpose** Two-element cell array of strings giving names of dimensions of data set

**Description** A two-element cell array of strings giving the names of the two dimensions of the data set. The default is {'Observations' 'Variables'}.

**Purpose** Display categorical array

**Syntax** `disp(A)`

**Description** `disp(A)` prints the categorical array `A` without printing the array name. In all other ways it's the same as leaving the semicolon off an expression, except that empty arrays don't display.

**See Also** `categorical` | `display`

# classregtree.disp

---

<b>Purpose</b>	Display classregtree object
<b>Syntax</b>	<code>display(t)</code>
<b>Description</b>	<code>display(t)</code> prints the classregtree object <code>t</code> .
<b>See Also</b>	<code>classregtree</code>   <code>view</code>



<b>Purpose</b>	Display cvpartition object
<b>Syntax</b>	<code>disp(c)</code>
<b>Description</b>	<code>disp(c)</code> prints the cvpartition object <code>c</code> .
<b>See Also</b>	<code>cvpartition</code>

# dataset disp

---

**Purpose** Display dataset array

**Syntax** `disp(ds)`

**Description** `disp(ds)` prints the dataset array `ds`, including variable names and observation names (if present), without printing the dataset name. In all other ways it's the same as leaving the semicolon off an expression.

For numeric or categorical variables that are 2-D and have three or fewer columns, `disp` prints the actual data using either short `g`, long `g`, or bank format, depending on the current command line setting. Otherwise, `disp` prints the size and type of each dataset element.

For character variables that are 2-D and 10 or fewer characters wide, `disp` prints quoted strings. Otherwise, `disp` prints the size and type of each dataset element.

For cell variables that are 2-D and have three or fewer columns, `disp` prints the contents of each cell (or its size and type if too large). Otherwise, `disp` prints the size of each dataset element.

For time series variables, `disp` prints columns for both the time and the data. If the variable is 2-D and has three or fewer columns, `disp` prints the actual data. Otherwise, `disp` prints the size and type of each dataset element.

For other types of variables, `disp` prints the size and type of each dataset element.

**See Also** `dataset` | `display` | `format`

**Purpose** Display generalized linear regression model

**Syntax** `disp mdl`

**Description** `disp mdl` displays the `mdl` linear model.

**Input Arguments** **mdl**  
Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

## Examples **Display a Generalized Linear Regression Model**

Create and display a generalized linear regression model.

Create a generalized linear regression model of Poisson data.

```
X = 2 + randn(100,1);  
mu = exp(1 + X/2);  
y = poissrnd(mu);  
mdl = fitglm(X,y,...  
    'y ~ x1','distr','poisson');
```

Display the model.

```
disp(mdl)
```

```
Generalized Linear regression model:
```

```
log(y) ~ 1 + x1  
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	0.9581	0.090294	10.611	2.6519e-26
x1	0.51027	0.033738	15.124	1.1179e-51

```
100 observations, 98 error degrees of freedom
```

# GeneralizedLinearModel.disp

---

Dispersion: 1  
Chi^2-statistic vs. constant model: 221, p-value = 5.77e-50

**Alternatives** Enter *mdl* at the command line to obtain a display, where *mdl* is the name of your model.

**See Also** [GeneralizedLinearModel](#) |

**Concepts**

- “Generalized Linear Models” on page 10-12

<b>Purpose</b>	Display Gaussian mixture distribution object
<b>Syntax</b>	<code>disp(obj)</code>
<b>Description</b>	<code>disp(obj)</code> prints a text representation of the <code>gmdistribution</code> object, <code>obj</code> , without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.
<b>See Also</b>	<code>gmdistribution</code>   <code>display</code>

# LinearModel disp

---

**Purpose** Display linear regression model

**Syntax** `display mdl`

**Description** `display mdl` displays the `mdl` linear model.

**Input Arguments** **mdl**  
Linear model, as constructed by `fitlm` or `stepwiselm`.

## Examples **Display a Linear Regression Model**

Create and display a linear regression model.

Create a linear regression model.

```
X = randn(100,5);  
y = X*[1;2;3;4;5] + 6 + randn(100,1);  
mdl = fitlm(X,y);
```

Display the model.

```
disp(mdl)
```

```
Linear regression model:  
y ~ 1 + x1 + x2 + x3 + x4 + x5
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	6.0806	0.11393	53.371	4.2509e-72
x1	1.1181	0.1154	9.6894	8.4046e-16
x2	2.0903	0.11378	18.372	7.2209e-33
x3	3.0926	0.10725	28.836	1.7967e-48
x4	3.9343	0.11489	34.244	6.9609e-55
x5	4.9538	0.10799	45.873	3.8195e-66

Number of observations: 100, Error degrees of freedom: 94

Root Mean Squared Error: 1.08  
R-squared: 0.979, Adjusted R-Squared 0.978  
F-statistic vs. constant model: 891, p-value = 1.59e-77

## Alternatives

Enter *mdl* at the command line to obtain a display, where *mdl* is the name of your model.

## See Also

LinearModel

## How To

- “Linear Regression” on page 9-12
- “Stepwise Regression” on page 9-119
- “Robust Regression — Reduce Outlier Effects” on page 9-125

# LinearMixedModel.disp

---

**Purpose** Display linear mixed-effects model

**Syntax** `display(lme)`

**Description** `display(lme)` displays the fitted linear mixed-effects model `lme`.

**Input Arguments** **lme - Linear mixed-effects model**

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

**Examples** **Randomized Block Design**

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The dataset array shows the absolute deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts, morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);
shift.Operator = nominal(shift.Operator);
```



Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Display the model.

```
disp(lme)
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

Formula:

```
QCDev ~ 1 + Shift + (1 | Operator)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
59.012	62.552	-24.506	49.012

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pV
'(Intercept)'	3.1196	0.88681	3.5178	12	0
'Shift_Morning'	-0.3868	0.48344	-0.80009	12	0
'Shift_Night'	1.9856	0.48344	4.1072	12	0

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	1.8297

Group: Error

Name	Estimate	Lower	Upper
------	----------	-------	-------

# LinearMixedModel.disp

---

'Res Std'                    0.76439                    0.49315                    1.1848

This display includes the model performance statistics, Akaike information criterion (AIC), Bayesian information criterion (BIC), loglikelihood, and deviance.

The fixed-effects coefficients table includes the names and estimates of the coefficients in the first two columns. The third column SE shows the standard errors of the coefficients. The column tStat includes the *t*-statistic values that correspond to each coefficient. DF is the residual degrees of freedom, and the pValue is the *p*-value that corresponds to the corresponding *t*-statistic value. The columns Lower and Upper display the lower and upper limits of a 95% confidence interval for each fixed-effects coefficient.

The first table for the random effects shows the types and the estimates of the random effects covariance parameters, with the lower and upper limits of a 95% confidence interval for each parameter. The display also shows the name of the grouping variable, operator, and the total number of levels, 5.

The second table for the random effects shows the estimate of the observation error, with the lower and upper limits of a 95% confidence interval.

## Definitions

### Akaike and Bayesian Information Criteria

Akaike information criterion (AIC) is  $AIC = -2*\log L_M + 2*(nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the model, and  $nc + p + 1$  is the number of parameters estimated in the model.  $p$  is the number of fixed-effects coefficients, and  $nc$  is the total number of parameters in the random-effects covariance excluding the residual variance.

Bayesian information criterion (BIC) is  $BIC = -2*\log L_M + \ln(n_{eff})*(nc + p + 1)$ , where  $\log L_M$  is the maximized log likelihood (or maximized restricted log likelihood) of the model,  $n_{eff}$  is the effective number of observations, and  $(nc + p + 1)$  is the number of parameters estimated in the model.

- If the fitting method is maximum likelihood (ML), then  $n_{eff} = n$ , where  $n$  is the number of observations.
- If the fitting method is restricted maximum likelihood (REML), then  $n_{eff} = n - p$ .

A lower value of deviance indicates a better fit. As the value of deviance decreases, both AIC and BIC tend to decrease. Both AIC and BIC also include penalty terms based on the number of parameters estimated,  $p$ . So, when the number of parameters increase, the values of AIC and BIC tend to increase as well. When comparing different models, the model with the lowest AIC or BIC value is considered as the best fitting model.

## Deviance

`LinearMixedModel` computes the deviance of model  $M$  as minus two times the loglikelihood of that model. Let  $L_M$  denote the maximum value of the likelihood function for model  $M$ . Then, the deviance of model  $M$  is

$$-2 * \log L_M.$$

A lower value of deviance indicates a better fit. Suppose  $M_1$  and  $M_2$  are two different models, where  $M_1$  is nested in  $M_2$ . Then, the fit of the models can be assessed by comparing the deviances  $Dev_1$  and  $Dev_2$  of these models. The difference of the deviances is

$$Dev = Dev_1 - Dev_2 = 2(\log LM_2 - \log LM_1).$$

Usually, the asymptotic distribution of this difference has a chi-square distribution with degrees of freedom  $v$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using `1 - chi2cdf(Dev, V)`, where  $Dev = Dev_2 - Dev_1$ .

However, in mixed-effects models, when some variance components fall on the boundary of the parameter space, the asymptotic distribution of this difference is more complicated. For example, consider the hypotheses

# LinearMixedModel.disp

---

$H_0: D = \begin{pmatrix} D_{11} & 0 \\ 0 & 0 \end{pmatrix}$ ,  $D$  is a  $q$ -by- $q$  symmetric positive semidefinite matrix.  
 $H_1: D$  is a  $(q+1)$ -by- $(q+1)$  symmetric positive semidefinite matrix.

That is,  $H_1$  states that the last row and column of  $D$  are different from zero. Here, the bigger model  $M_2$  has  $q + 1$  parameters and the smaller model  $M_1$  has  $q$  parameters. And  $Dev$  has a 50:50 mixture of  $\chi^2_q$  and  $\chi^2_{(q+1)}$  distributions (Stram and Lee, 1994).

## References

[1] Hox, J. *Multilevel Analysis, Techniques and Applications*. Lawrence Erlbaum Associates, Inc., 2002.

[2] Stram D. O. and J. W. Lee. "Variance components testing in the longitudinal mixed-effects model". *Biometrics*, Vol. 50, 4, 1994, pp. 1171–1177.

## See Also

`LinearMixedModel` | `fitlme` | `fitlmematrix`

<b>Purpose</b>	Display NaiveBayes classifier object
<b>Syntax</b>	<code>disp(nb)</code>
<b>Description</b>	<code>disp(nb)</code> prints a text representation of the NaiveBayes object <code>nb</code> , without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.
<b>See Also</b>	<code>NaiveBayes</code>   <code>display</code>

# NonLinearModel.disp

---

**Purpose** Display nonlinear regression model

**Syntax** `disp(md1)`

**Description** `disp(md1)` displays the `md1` nonlinear model at the command line.

**Input Arguments** **md1**  
Nonlinear regression model, constructed by `fitnlm`.

## Examples **Display a Nonlinear Regression Model**

Create and display a nonlinear regression model.

Load the reaction data, and specify both a model function and starting values for the iterations.

```
load reaction
modelfun = 'rate~(b1*x2-x3/b5)/(1+b2*x1+b3*x2+b4*x3)';
beta0 = [1 .05 .02 .1 2];
```

Create a model of the data.

```
mdl = fitnlm(reactants,rate,modelfun,beta0);
```

Display the model.

```
disp(mdl)
```

```
Nonlinear regression model:
    rate ~ (b1*x2 - x3/b5)/(1 + b2*x1 + b3*x2 + b4*x3)
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 0.193  
R-Squared: 0.999, Adjusted R-Squared 0.998  
F-statistic vs. constant model: 1.81e+03, p-value = 7.36e-12

**Alternatives** Enter *mdl* at the command line to obtain a display, where *mdl* is the name of your model.

**See Also** NonLinearModel |

**Concepts**

- “Nonlinear Regression” on page 11-2

# piecwisedistribution.disp

---

<b>Purpose</b>	Display piecwisedistribution object
<b>Syntax</b>	<code>disp(A)</code>
<b>Description</b>	<code>disp(A)</code> prints a text representation of the piecwisedistribution object A, without printing the object name. In all other ways it's the same as leaving the semicolon off an expression.
<b>See Also</b>	<code>piecwisedistribution</code>



**Purpose** Display grandset object

**Syntax** `disp(p)`

**Description** `disp(p)` displays the properties of the quasi-random point set `s`, without printing the variable name. `disp` prints out the number of dimensions and points in the point-set, and follows this with the list of all property values for the object.

**See Also** `grandset`

# grandstream.disp

---

**Purpose**            Display grandstream object

**Syntax**            `disp(q)`

**Description**        `disp(q)` displays the quasi-random stream `q`, without printing the variable name. `disp` prints the type and number of dimensions in the stream, and follows it with the list of point set properties.

**See Also**            `grandstream`

**Purpose** Display categorical array

**Syntax** `display(A)`

**Description** `display(A)` prints the categorical array `A`. `categorical` calls `display` when a you do not use a semicolon to terminate a statement.

**See Also** `categorical` | `disp`

# classregtree.display

---

<b>Purpose</b>	Display classregtree object
<b>Syntax</b>	<code>display(t)</code> <code>display(A)</code>
<b>Description</b>	<code>display(t)</code> prints the classregtree object <code>t</code> . <code>classregtree</code> calls <code>display</code> when a you do not use a semicolon to terminate a statement.  <code>display(A)</code> prints the categorical array <code>A</code> . <code>categorical</code> calls <code>display</code> when a you do not use a semicolon to terminate a statement.
<b>See Also</b>	<code>classregtree</code>   <code>eval</code>   <code>prune</code>   <code>test</code>

<b>Purpose</b>	Display cvpartition object
<b>Syntax</b>	<code>display(c)</code>
<b>Description</b>	<code>display(c)</code> prints the cvpartition object <code>c</code> . <code>cvpartition</code> calls <code>display</code> when a you do not use a semicolon to terminate a statement.
<b>See Also</b>	<code>cvpartition</code>

# dataset.display

---

**Purpose** Display dataset array

**Syntax** `display(ds)`

**Description** `display(ds)` prints the dataset array `ds`, including variable names and observation names (if present). `dataset` calls `display` when a you do not use a semicolon to terminate a statement

For numeric or categorical variables that are 2-D and have three or fewer columns, `display` prints the actual data. Otherwise, `display` prints the size and type of each dataset element.

For character variables that are 2-D and 10 or fewer characters wide, `display` prints quoted strings. Otherwise, `display` prints the size and type of each dataset element.

For cell variables that are 2-D and have three or fewer columns, `display` prints the contents of each cell (or its size and type if too large). Otherwise, `display` prints the size of each dataset element.

For time series variables, `display` prints columns for both the time and the data. If the variable is 2-D and has three or fewer columns, `display` prints the actual data. Otherwise, `display` prints the size and type of each dataset element.

For other types of variables, `display` prints the size and type of each dataset element.

**See Also** `dataset` | `display` | `format`

<b>Purpose</b>	Display Gaussian mixture distribution object
<b>Syntax</b>	<code>display(obj)</code>
<b>Description</b>	<code>display(obj)</code> prints a text representation of the <code>gmdistribution</code> object <code>obj</code> . <code>gmdistribution</code> calls <code>display</code> when a you do not use a semicolon to terminate a statement.
<b>See Also</b>	<code>gmdistribution</code>   <code>disp</code>

# NaiveBayes.display

---

**Purpose**            Display NaiveBayes classifier object

**Syntax**            `display(nb)`

**Description**        `display(nb)` prints a text representation of the NaiveBayes object `nb`. NaiveBayes calls `display` when a you do not use a semicolon to terminate a statement.

**See Also**            NaiveBayes | display



<b>Purpose</b>	Display piecwisedistribution object
<b>Syntax</b>	<code>display(A)</code>
<b>Description</b>	<code>display(A)</code> prints a text representation of the piecwisedistribution object <code>A</code> , without printing the object name. <code>piecwisedistribution</code> calls <code>display</code> when a you do not use a semicolon to terminate a statement.
<b>See Also</b>	<code>piecwisedistribution</code>

# ProbDist.DistName property

---

**Purpose** Read-only string containing probability distribution name of ProbDist object

**Description** DistName is a read-only property of the ProbDist class. DistName is a string containing the type of distribution used to create the object.

**Values** Possible values are:

- 'kernel'
- 'beta'
- 'binomial'
- 'birnbaumsaunders'
- 'exponential'
- 'extreme value'
- 'gamma'
- 'generalized extreme value'
- 'generalized pareto'
- 'inversegaussian'
- 'logistic'
- 'loglogistic'
- 'lognormal'
- 'nakagami'
- 'negative binomial'
- 'normal'
- 'poisson'
- 'rayleigh'
- 'rician'

## ProbDist.DistName property

---

- 'tlocationscale'
- 'weibull'

Use this information to view and compare the type of distribution used to create distribution objects.

# NaiveBayes.Dist property

---

**Purpose** Distribution names

**Description** The `Dist` property is a string or a 1-by-NDims cell array of strings indicating the types of distributions for all the features. If all the features use the same type of distribution, `Dist` is a single string. Otherwise `Dist(j)` indicates the distribution type used for the `j`th feature.

The valid strings for this property are the following:

'normal'	Normal distribution.
'kernel'	Kernel smoothing density estimate.
'mvmn'	Multivariate multinomial distribution.
'mn'	Multinomial bag-of-tokens model.

## gmdistribution.DistName property

---

**Purpose** Type of distribution

**Description** The string 'gaussian mixture distribution'.

# disttool

---

<b>Purpose</b>	Interactive density and distribution plots
<b>Syntax</b>	<code>disttool</code>
<b>Description</b>	<code>disttool</code> is a graphical interface for exploring the effects of changing parameters on the plot of a cdf or pdf.
<b>See Also</b>	<code>randtool</code>   <code>dfittool</code>

**Purpose** Convert categorical array to double array

**Syntax** `B = double(A)`

**Description** `B = double(A)` converts the categorical array `A` to a double array. Each element of `B` contains the internal categorical level code for the corresponding element of `A`.

**See Also** `single`

# dataset.double

---

**Purpose** Convert dataset variables to double array

**Syntax** `b = double(A)`  
`b = double(a,vars)`

**Description** `b = double(A)` returns the contents of the dataset A, converted to one double array. The classes of the variables in the dataset must support the conversion.

`b = double(a,vars)` returns the contents of the dataset variables specified by vars. vars is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

**See Also** `dataset` | `single` | `replacedata`



## Purpose

Drop levels

## Syntax

```
B = droplevels(A)
B = droplevels(A,oldlevels)
```

## Description

`B = droplevels(A)` removes unused levels from the categorical array `A`. `B` is a categorical array with the same size and values as `A`, but with a list of potential levels that includes only those present in some element of `A`.

`B = droplevels(A,oldlevels)` removes specified levels from the categorical array `A`. `oldlevels` is a cell array of strings or a 2-D character matrix specifying the levels to be removed.

`droplevels` removes levels, but does not remove elements. Elements of `B` that correspond to elements of `A` having levels in `oldlevels` all have an undefined level.

## Examples

### Example 1

Drop unused age levels from the data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);
AgeGroup = droplevels(AgeGroup);
getlabels(AgeGroup)
ans =
    '20s'    '30s'    '40s'    '50s'
```

### Example 2

- 1 Load patient data from the CSV file `hospital.dat` and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
                  'delimiter',';',...)
```

## categorical.droplevels

---

```
'ReadObsNames', true);
```

- 2 Make the {0,1}-valued variable `smoke` nominal, and change the labels to 'No' and 'Yes':

```
patients.smoke = nominal(patients.smoke, {'No', 'Yes'});
```

- 3 Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke, ...  
                           {'0-5 Years', '5-10 Years', 'LongTerm'});
```

- 4 Assuming the nonsmokers have never smoked, relabel the 'No' level:

```
patients.smoke = setlabels(patients.smoke, 'Never', 'No');
```

- 5 Drop the undifferentiated 'Yes' level from `smoke`:

```
patients.smoke = droplevels(patients.smoke, 'Yes');
```

Warning: OLDLEVELS contains categorical levels that were present in A, caused some array elements to have undefined levels.

Note that smokers now have an undefined level.

- 6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

### See Also

`addlevels` | `getlabels` | `islevel` | `mergelevels` | `reorderlevels`

<b>Purpose</b>	Create dummy variables
<b>Syntax</b>	<code>D = dummyvar(group)</code>
<b>Description</b>	<p><code>D = dummyvar(group)</code> returns a matrix <code>D</code> containing zeros and ones, whose columns are dummy variables for the grouping variable <code>group</code>. Columns of <code>group</code> represent categorical predictor variables, with values indicating categorical levels. Rows of <code>group</code> represent observations across variables.</p> <p><code>group</code> can be a numeric vector or categorical column vector representing levels within a single variable, a cell array containing one or more grouping variables, or a numeric matrix or cell array of categorical column vectors representing levels within multiple variables. If <code>group</code> is a numeric vector or matrix, values in any column must be positive integers in the range from 1 to the number of levels for the corresponding variable. In this case, <code>dummyvars</code> treats each column as a separate numeric grouping variable. With multiple grouping variables, the sets of dummy variable columns are in the same order as the grouping variables in <code>group</code>.</p> <p>The order of the dummy variable columns in <code>D</code> matches the order of the groups defined by <code>group</code>. When <code>group</code> is a categorical vector, the groups and their order match the output of the <code>getlabels(group)</code> method. When <code>group</code> is a numeric vector, <code>dummyvar</code> assumes that the groups and their order are <code>1:max(group)</code>. In this respect, <code>dummyvars</code> treats a numeric grouping variable differently than <code>grp2idx</code>.</p> <p>If <code>group</code> is <math>n</math>-by-<math>p</math>, <code>D</code> is <math>n</math>-by-<math>S</math>, where <math>S</math> is the sum of the number of levels in each of the columns of <code>group</code>. The number of levels <math>s</math> in any column of <code>group</code> is the maximum positive integer in the column or the number of categorical levels. Levels are considered distinct if they appear in different columns of <code>group</code>, even if they have the same value. Columns of <code>D</code> are, from left to right, dummy variables created from the first column of <code>group</code>, followed by dummy variables created from the second column of <code>group</code>, etc.</p> <p><code>dummyvar</code> treats NaN values or undefined categorical levels in <code>group</code> as missing data and returns NaN values in <code>D</code>.</p>

# dummyvar

---

Dummy variables are used in regression analysis and ANOVA to indicate values of categorical predictors.

---

**Note** If a column of 1s is introduced in the matrix  $D$ , the resulting matrix  $X = [\text{ones}(\text{size}(D,1),1) \ D]$  will be rank deficient. The matrix  $D$  itself will be rank deficient if `group` has multiple columns. This is because dummy variables produced from any column of `group` always sum to a column of 1s. Regression and ANOVA calculations often address this issue by eliminating one dummy variable (implicitly setting the coefficients for dropped columns to zero) from each group of dummy variables produced by a column of `group`.

---

## Examples

Suppose you are studying the effects of two machines and three operators on a process. Use `group` to organize predictor data on machine-operator combinations:

```
machine = [1 1 1 1 2 2 2 2]';
operator = [1 2 3 1 2 3 1 2]';
group = [machine operator]
group =
     1     1
     1     2
     1     3
     1     1
     2     2
     2     3
     2     1
     2     2
```

Use `dummyvar` to create dummy variables for a regression or ANOVA calculation:

```
D = dummyvar(group)
D =
     1     0     1     0     0
```

1	0	0	1	0
1	0	0	0	1
1	0	1	0	0
0	1	0	1	0
0	1	0	0	1
0	1	1	0	0
0	1	0	1	0

The first two columns of D represent observations of machine 1 and machine 2, respectively; the remaining columns represent observations of the three operators.

## See Also

`regress` | `anova1`

## How To

- “Grouping Variables” on page 2-51
- “Dummy Indicator Variables” on page 2-55
- “Regression with Categorical Covariates” on page 2-59

# dwtest

---

## Purpose

Durbin-Watson test

## Syntax

```
p = dwtest(r,x)
p = dwtest(r,x,Name,Value)
[p,dw] = dwtest( ___ )
```

## Description

`p = dwtest(r,x)` returns the  $p$ -value for the Durbin-Watson test of the null hypothesis that the residuals from a linear regression are uncorrelated. The alternative hypothesis is that there is autocorrelation among the residuals.

`p = dwtest(r,x,Name,Value)` returns the  $p$ -value for the Durbin-Watson test with additional options specified by one or more name-value pair arguments. For example, you can conduct a one-sided test or calculate the  $p$ -value using a normal approximation.

`[p,dw] = dwtest( ___ )` also returns the Durbin-Watson test statistic, `dw`, using any of the input arguments from the previous syntaxes.

## Input Arguments

### **x** - Design matrix

matrix

Design matrix for a linear regression, specified as a matrix. Include a column of 1 values in the design matrix so the model contains a constant term.

### **Data Types**

single | double

### **r** - Regression residuals

vector

Regression residuals, specified as a vector. Obtain `r` by performing a linear regression using a function such as `regress`, or by using the backslash operator.

## Data Types

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'Tail', 'right', 'Method', 'approximate'` specifies a right-tailed hypothesis test and calculates the  $p$ -value using a normal approximation.

## 'Method' - Computation method for $p$ -value

`'exact'` | `'approximate'`

Computation method for the  $p$ -value, specified as the comma-separated pair consisting of `'Method'` and one of the following.

`'exact'` Calculate an exact  $p$ -value using the Pan algorithm. This is the default if the sample size is less than 400.

`'approximate'` Calculate the  $p$ -value using a normal approximation. This is the default if the sample size is 400 or larger.

**Example:** `'Method', 'exact'`

## 'Tail' - Type of alternative hypothesis

`'both'` (default) | `'right'` | `'left'`

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of `'Tail'` and one of the following.

'both'	Test the alternate hypothesis that autocorrelation among the residuals is not zero.
'right'	Test the alternative hypothesis that autocorrelation among the residuals is greater than zero.
'left'	Test the alternative hypothesis that autocorrelation among the residuals is less than zero.

Example: 'Tail', 'right'

## Output Arguments

### **p - p-value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **dw - Test statistic**

nonnegative scalar value

Test statistic of the hypothesis test, returned as a nonnegative scalar value.

## Examples

### **Test Residuals for Correlation**

Load the sample census data.

```
load census;
```

Create a design matrix using the census date (cdate) as the predictor. Add a column of 1 values to include a constant term.

```
n = length(cdate);  
x = [ones(n,1),cdate];
```

Fit a linear regression to the data.



```
[b,bint,r] = regress(pop,x);
```

Test the null hypothesis that there is no autocorrelation among the residuals,  $r$ .

```
[p,dw] = dwtest(r,x)
```

```
p =  
    0
```

```
dw =  
    0.1308
```

The returned value  $p = 0$  indicates rejection of the null hypothesis at the 5% significance level.

### One-Sided Hypothesis Test

Load the sample census data.

```
load census;
```

Create a design matrix using the census date (`cdate`) as the predictor. Add a column of 1 values to include a constant term.

```
n = length(cdate);  
x = [ones(n,1),cdate];
```

Fit a linear regression to the data.

```
[b,bint,r] = regress(pop,x);
```

Test the null hypothesis that there is no autocorrelation among regression residuals, against the alternative hypothesis that the autocorrelation is greater than zero.

```
[p,dw] = dwtest(r,x,'Tail','right')
```

```
p =  
    0
```

dw =  
0.1308

The returned value  $p = 0$  indicates rejection of the null hypothesis at the 5% significance level, in favor of the alternative hypothesis that the autocorrelation among residuals is greater than zero.

## Definitions **Durbin-Watson Test**

The Durbin-Watson test is used to test if linear regression residuals are uncorrelated, against the alternative that autocorrelation exists.

The test statistic is

$$d = \frac{\sum_{t=2}^T (e_t - e_{t-1})^2}{\sum_{t=1}^T e_t^2},$$

where  $T$  is the number of observations, and  $e_t$  is the residual at time  $t$ .

The  $p$ -value can be calculated exactly using the Pan algorithm. Alternatively, the  $p$ -value can be estimated using a normal approximation. Significantly small  $p$ -values indicate correlation among residuals.

**See Also** regress

**Purpose**

Durbin-Watson test of linear model

**Syntax**

```
P = dwtest mdl
[P,DW] = dwtest mdl
[P,DW] = dwtest mdl,method
[P,DW] = dwtest mdl,method,tail
```

**Description**

`P = dwtest(mdl)` returns the  $p$ -value of the Durbin-Watson test on the `mdl` linear model.

`[P,DW] = dwtest(mdl)` returns the Durbin-Watson statistic.

`[P,DW] = dwtest(mdl,method)` specifies the method `dwtest` uses to compute the  $p$ -value.

`[P,DW] = dwtest(mdl,method,tail)` specifies the alternative hypothesis.

**Input Arguments****mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

**method**

Algorithm for computing the  $p$ -value:

- 'exact' — Calculates an exact  $p$ -value using Pan's algorithm.
- 'approximate' — Calculates the  $p$ -value using a normal approximation.

**Default:** 'exact' when the sample size is less than 400, 'approximate' otherwise

**tail**

`dwtest` tests whether `mdl` has no serial correlation against one of these alternative hypotheses:

# LinearModel.dwtest

---

Tail	Alternative Hypothesis
'both'	Serial correlation is not 0.
'right'	Serial correlation is greater than 0 (right-tailed test).
'left'	Serial correlation is less than 0 (left-tailed test).

**Default:** 'both'

## Output Arguments

**P**

*p*-value of the test, a scalar. `dwtest` tests if the residuals are uncorrelated, against the alternative that there is autocorrelation among them. Small values of **P** indicate that the residuals are correlated.

**DW**

Value of the Durbin-Watson statistic, a scalar.

## Definitions

### Durbin-Watson Statistic

Let  $r$  be the vector of residuals (in `mdl.residuals.response`). The Durbin-Watson statistic is

$$DW = \frac{\sum_{i=1}^{n-1} (r_{i+1} - r_i)^2}{\sum_{i=1}^n r_i^2}.$$

## Examples

### Test Residuals for Autocorrelation

Examine whether the residuals from a fitted model of census data over time have autocorrelated residuals.

Load the census data and create a linear model.

```
load census
mdl = fitlm(cdate,pop);
```

Find the  $p$ -value of the Durbin-Watson autocorrelation test.

```
P = dwtest(mdl)
```

```
P =
```

```
0
```

There is significant autocorrelation in the residuals.

## Algorithms

Approximate calculation of the  $p$ -value uses a normal approximation [1]. Exact calculation uses Pan's algorithm [2].

## References

- [1] Durbin, J., and G. S. Watson. *Testing for Serial Correlation in Least Squares Regression I*. *Biometrika* 37, pp. 409–428, 1950.
- [2] Farebrother, R. W. *Pan's Procedure for the Tail Probabilities of the Durbin-Watson Statistic*. *Applied Statistics* 29, pp. 224–227, 1980.

## See Also

LinearModel

## How To

- “Linear Regression” on page 9-12

**Purpose** Empirical cumulative distribution function

**Syntax**  
`[f,x] = ecdf(y)`  
`[f,x] = ecdf(y,Name,Value)`  
`[f,x,flo,fup] = ecdf( ___ )`

`ecdf( ___ )`  
`ecdf(ax, ___ )`

**Description** `[f,x] = ecdf(y)` returns the empirical cumulative distribution function (cdf), `f`, evaluated at the points in `x`, using the data in the vector `y`.

In survival and reliability analysis, this empirical cdf is called the Kaplan-Meier estimate. And the data might correspond to survival or failure times.

`[f,x] = ecdf(y,Name,Value)` returns the empirical function values, `f`, evaluated at the points in `x`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the type of function to evaluate or which data is censored.

`[f,x,flo,fup] = ecdf( ___ )` also returns the 95% lower and upper confidence bounds for the evaluated function values. You can use any of the input arguments in the previous syntaxes.

`ecdf` computes the confidence bounds using Greenwood's formula. They are not simultaneous confidence bounds.

`ecdf( ___ )` plots the evaluated function.

`ecdf(ax, ___ )` plots the evaluated function using axes with the handle, `ax`, instead of the current axes returned by `gca`.

## Input Arguments

### **y - Input data**

column vector

Input data, specified as a column vector. For example, in survival or reliability analysis, data might be survival or failure times for each item or individual.

### **Data Types**

single | double

### **ax - Axes handle**

handle

Axes handle for the figure ecdf plots to, specified as a handle.

For instance, if `h` is a handle for a figure, then `ecdf` can plot to that figure as follows.

**Example:** `ecdf(h,x)`

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'censoring',c,'function','cumulative hazard','alpha',0.025,'bounds','on'` specifies that `ecdf` returns the cumulative hazard function and plots the 97.5% confidence bounds, accounting for the censored data specified by vector `c`.

### **'censoring' - Indicator of censored data**

array of 0s (default) | vector of 0s and 1s

Indicator of censored data, specified as the comma-separated pair including `'censoring'` and a Boolean array of the same size as `x`. Enter 1 for observations that are right-censored and 0 for observations that are fully observed. Default is all observations are fully observed.

For instance, if vector `cdata` stores the censored data information, you can enter the censoring information as follows.

**Example:** `'censoring',cdata`

### **Data Types**

logical

### **'frequency' - Frequency of observations**

array of 1s (default) | vector of nonnegative scalars

Frequency of observations, specified as the comma-separated pair consisting of `'frequency'` and a vector containing nonnegative integer counts. This vector is the same size as the vector `x`. The  $j$ th element of this vector gives the number of times the  $j$ th element of `x` was observed. Default is one observation per element of `x`.

For instance, if `failurefreq` is a vector of frequencies, then you can enter it as follows.

**Example:** `'frequency',failurefreq`

### **Data Types**

single | double

### **'alpha' - Confidence level**

0.05 (default) | scalar value in the range (0,1)

Confidence level for the confidence interval of the evaluated function, specified as the comma-separated pair consisting of `'alpha'` and a scalar value between in the range (0,1). Default is 0.05 for 95% confidence. For a given value `alpha`, the confidence level is  $100(1 - \alpha)\%$ .

For instance, for a 99% confidence interval, you can specify the `alpha` value as follows.

**Example:** `'alpha',0.01`

### **Data Types**

single | double

### **'function' - Type of function returned**



'cdf' (default) | 'survivor' | 'cumulative hazard'

Type of function that `ecdf` evaluates and returns, specified as the comma-separated pair consisting of 'function' and one of the following.

'cdf'	Default. Cumulative distribution function.
'survivor'	Survivor function.
'cumulative hazard'	Cumulative hazard function.

**Example:** 'function', 'cumulative hazard'

### Data Types

char

### 'bounds' - Indicator for including bounds

'off' (default) | 'on'

Indicator for including bounds, specified as the comma-separated pair consisting of 'bounds' and one of the following.

'off'	Default. Specify to omit bounds.
'on'	Specify to include bounds.

---

**Note** This name-value argument is used only for plotting.

---

**Example:** 'bounds', 'on'

### Data Types

char

## Output Arguments

### **f** - Function values

column vector

Function values evaluated at the points in **x**, returned as a column vector.

### **x** - Distinct observed points

column vector

Distinct observed points in data vector **y**, returned as a column vector.

### **flo** - Lower confidence bound

column vector

Lower confidence bound for the evaluated function, returned as a column vector. `ecdf` computes the confidence bounds using Greenwood's formula. They are not simultaneous confidence bounds.

### **fup** - Upper confidence bound

column vector

Upper confidence bound for the evaluated function, returned as a column vector. `ecdf` computes the confidence bounds using Greenwood's formula. They are not simultaneous confidence bounds.

## Examples

### **Compute Empirical Cumulative Distribution Function**

Compute the Kaplan-Meier estimate of the cumulative distribution function (cdf) for simulated survival data.

Generate survival data from a Weibull distribution with parameters 3 and 1.

```
rng('default') % for reproducibility
failuretime = random('wbl',3,1,15,1);
```

Compute the Kaplan-Meier estimate of the cdf for survival data.

```
[f,x] = ecdf(failuretime);
```

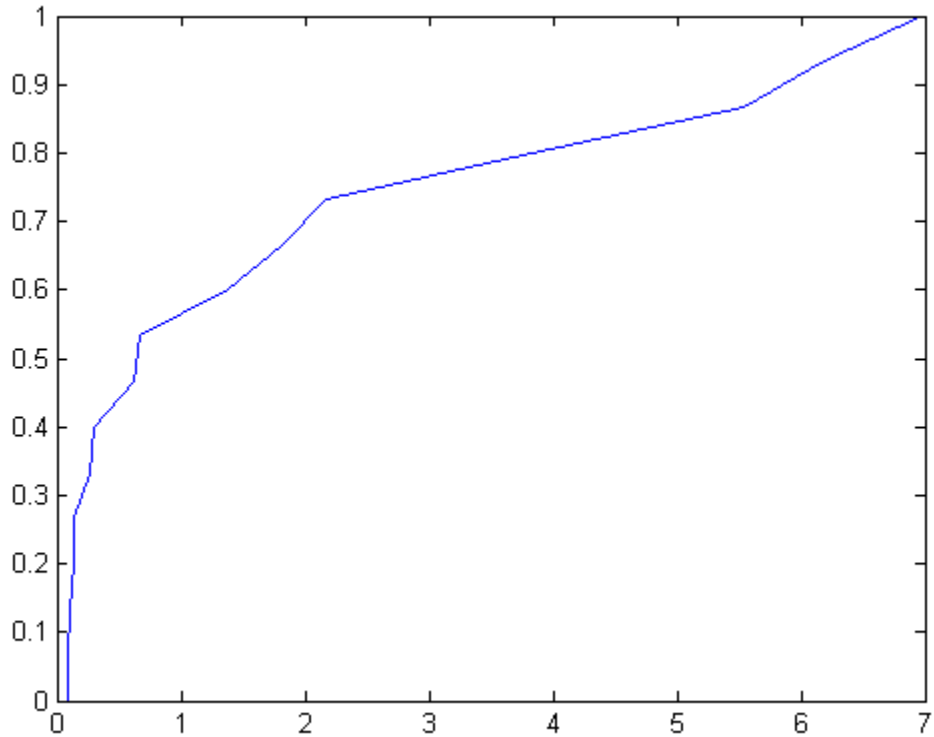
```
[f,x]
```

```
ans =
```

0	0.0895
0.0667	0.0895
0.1333	0.1072
0.2000	0.1303
0.2667	0.1313
0.3333	0.2718
0.4000	0.2968
0.4667	0.6147
0.5333	0.6684
0.6000	1.3749
0.6667	1.8106
0.7333	2.1685
0.8000	3.8350
0.8667	5.5428
0.9333	6.1910
1.0000	6.9825

Plot the estimated cdf.

```
figure()  
plot(x,f)
```



## Empirical Hazard Function of Right-Censored Data

Compute and plot the hazard function of simulated right-censored survival data.

Generate failure times from a Birnbaum-Saunders distribution.

```
rng('default') % for reproducibility  
failuretime = random('birnbaumsaunders',0.3,1,100,1);
```

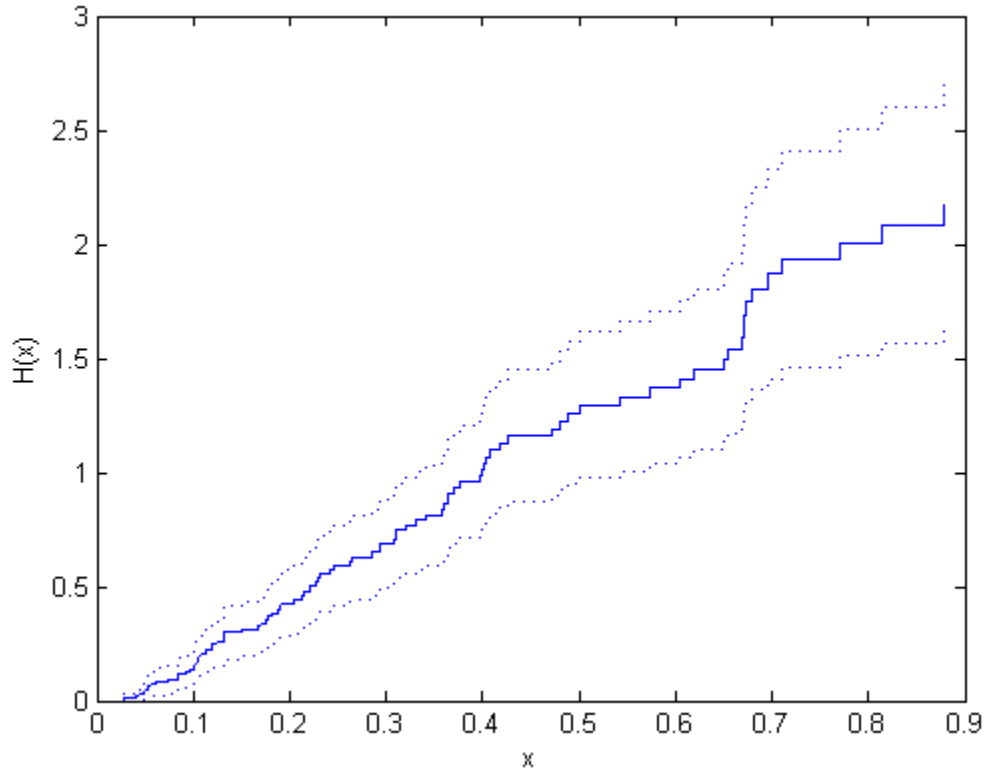
---

Assuming that the end of the study is at time 0.9, generate a logical array that indicates simulated failure times that are larger than 0.9 as censored data, and store this information in a vector.

```
T = 0.9;  
cens = (failuretime>T);
```

Plot the empirical hazard function for the data.

```
ecdf(failuretime,'function','cumulative hazard',...  
'censoring',cens,'bounds','on');
```



## Compare Empirical Cumulative Distribution Function (CDF) with Known CDF

Generate right-censored survival data and compare the empirical cumulative distribution function (cdf) with the known cdf.

Generate failure times from an exponential distribution with mean failure time of 15.

```
rng('default')  
y = exprnd(15,75,1);
```

Generate drop-out times from an exponential distribution with mean failure time of 30.

```
d = exprnd(30,75,1);
```

Generate the observed failure times. They are the minimum of the generated failure times and the drop-out times.

```
t = min(y,d);
```

Create a logical array that indicates generated failure times that are larger than the drop-out times. The data for which this is true are censored.

```
censored = (y>d);
```

Compute the empirical cdf and confidence bounds.

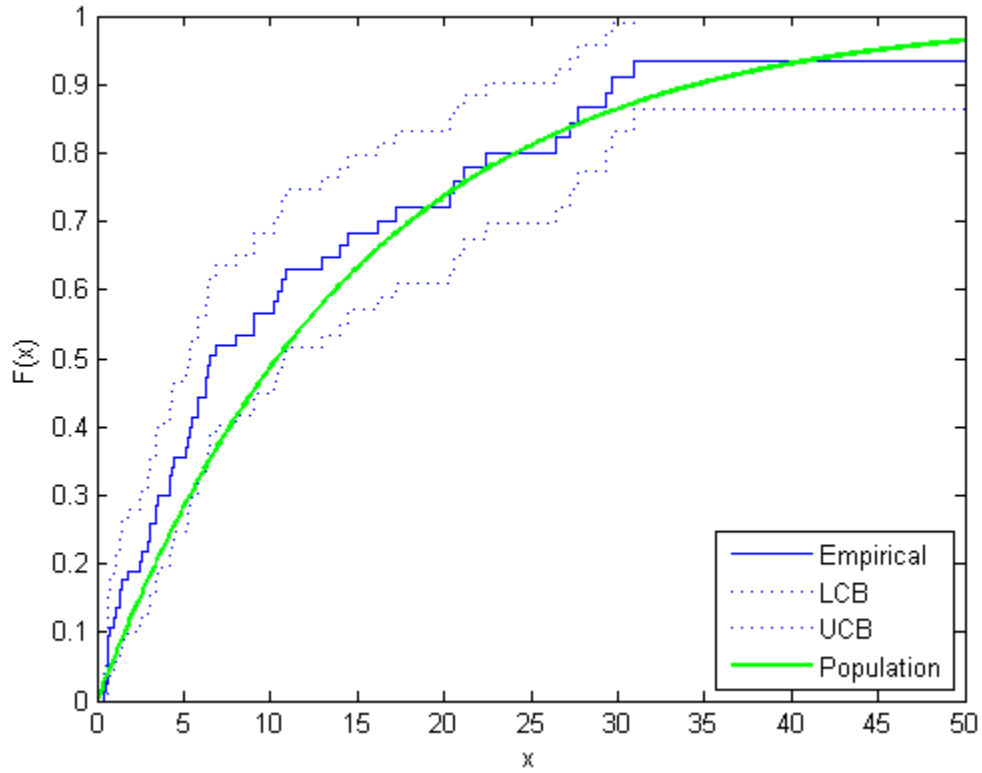
```
[f,x,flo,fup] = ecdf(t,'censoring',censored);
```

Plot the cdf and confidence bounds.

```
figure()  
ecdf(t,'censoring',censored,'bounds','on');  
hold on
```

Superimpose a plot of the known population cdf.

```
xx = 0:.1:max(t);  
yy = 1-exp(-xx/15);  
plot(xx,yy,'g-','LineWidth',2)  
axis([0 50 0 1])  
legend('Empirical','LCB','UCB','Population',...  
       'Location','SE')  
hold off
```



## Empirical Survivor Function with 99% Confidence Bounds

Generate survival data and plot the empirical survivor function with 99% confidence bounds.

Generate lifetime data from a Weibull distribution with parameters 100 and 2.

```
rng('default') % for reproducibility  
R = wblrnd(100,2,100,1);
```

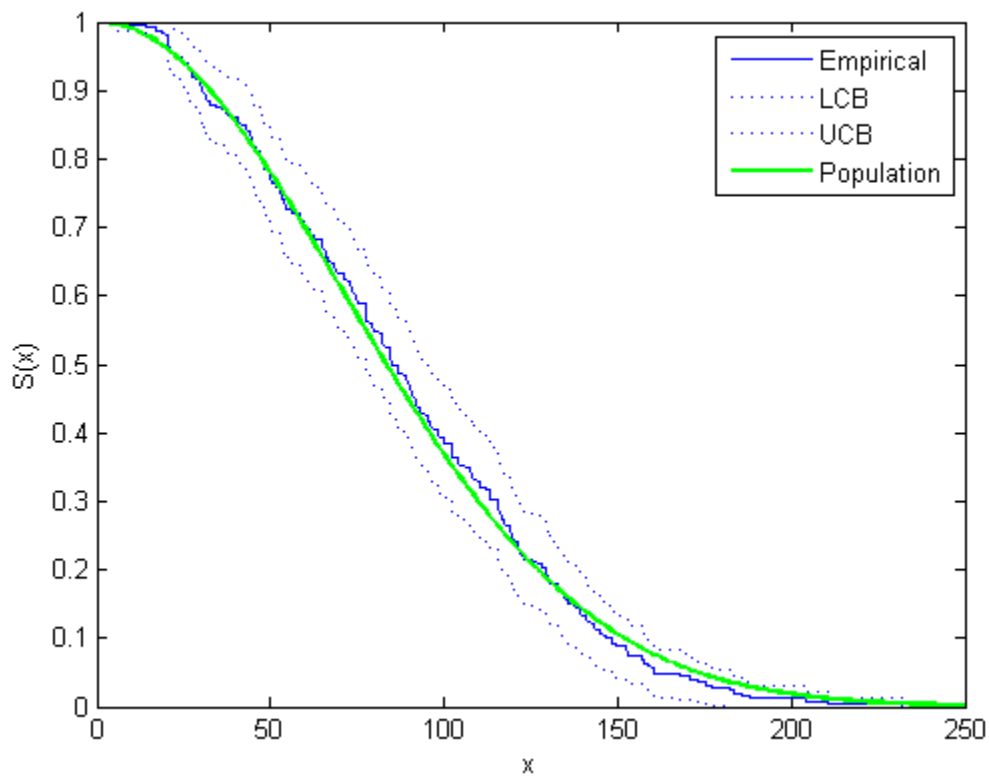


Plot the survivor function for the data with 99% confidence bounds.

```
ecdf(R, 'function', 'survivor', 'alpha', 0.01, 'bounds', 'on')  
hold on
```

Fit the Weibull survivor function.

```
x = 1:1:250;  
wblsurv = 1-cdf('weibull',x,100,2);  
plot(x,wblsurv,'g-', 'LineWidth',2)  
legend('Empirical','LCB','UCB','Population',...  
      'Location','NE')
```



The survivor function based on the actual distribution is within the confidence bounds.

## Definitions

### Greenwood's Formula

Approximation for the variance of Kaplan-Meier estimator.

The variance estimate is given by

$$V(S(t)) = S^2(t) \sum_{t_i < T} \frac{d_i}{r_i(r_i - d_i)},$$

where  $r_i$  is the number at risk at time  $t_i$ , and  $d_i$  is the number of failures at time  $t_i$ .

## References

- [1] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [2] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2003.

## See Also

`cdfplot` | `ecdfhist`

# ecdfhist

---

**Purpose** Histogram based on empirical cumulative distribution function

**Syntax** `[n,c] = ecdfhist(f,x)`  
`[n,c] = ecdfhist(f,x,m)`

`n = ecdfhist(f,x,centers)`

`ecdfhist( ___ )`

**Description** `[n,c] = ecdfhist(f,x)` returns the heights, `n`, of histogram bars for 10 equally spaced bins and the position of the bin centers, `c`.

`ecdfhist` computes the bar heights from the increases in the empirical cumulative distribution function, `f`, at evaluation points, `x`. It normalizes the bar heights so that the area of the histogram is equal to 1. In contrast, `hist` produces bars with heights representing bin counts.

`[n,c] = ecdfhist(f,x,m)` returns the histogram bars using `m` bins.

`n = ecdfhist(f,x,centers)` returns the heights of the histogram bars with bin centers specified by `centers`.

`ecdfhist( ___ )` plots the histogram bars.

## Input Arguments

### **f** - Empirical cdf values

vector

Empirical cdf values at given evaluation points, `x`, specified as a vector.

For instance, you can use `ecdf` to obtain the empirical cdf values and enter them in `ecdfhist` as follows.

**Example:** `[f,x] = ecdf(failure); ecdfhist(f,x);`

### **Data Types**

single | double

**x - Evaluation points**

vector

Evaluation points at which empirical cdf values, `f`, are calculated, specified as a vector.

For instance, you can use `ecdf` to obtain the empirical cdf values and enter them in `ecdfhist` as follows.

**Example:** `[f,x] = ecdf(failure); ecdfhist(f,x);`

**Data Types**

single | double

**m - Number of bins**

scalar

Number of bins, specified as a scalar.

For instance, you can draw a histogram with 8 bins as follows.

**Example:** `ecdfhist(f,x,8)`

**Data Types**

single | double

**centers - Center points of bins**

vector

Center points of bins, specified as a vector.

**Example:** `centers = 2:2:10; ecdfhist(f,x,centers);`

**Data Types**

single | double

**Output Arguments****n - Heights of histogram bars**

row vector

Heights of histogram bars `ecdfhist` calculates based on the empirical cdf values, returned as a row vector.

**c - Position of bin centers**

row vector

Position of bin centers, returned as a row vector.

## Examples

### Return Histogram Bar Heights and Bin Centers

Compute the histogram bar heights based on the empirical cumulative distribution function.

Generate failure times from a Birnbaum-Saunders distribution.

```
rng('default') % for reproducibility
failuretime = random('birnbaumsaunders',0.3,1,100,1);
```

Assuming that the end of the study is at time 0.9, mark the generated failure times that are larger than 0.9 as censored data and store that information in a vector.

```
T = 0.9;
cens = (failuretime>T);
```

Compute the empirical cumulative distribution function for the data.

```
[f,x] = ecdf(failuretime,'censoring',cens);
```

Now, find the bar heights of the histogram using the cumulative distribution function estimate.

```
n = ecdfhist(f,x);
[n' c']
```

```
ans =
```

```
2.3529    0.0715
1.7647    0.1565
1.4117    0.2415
1.5294    0.3265
1.0588    0.4115
0.4706    0.4965
```

```

0.4706    0.5815
0.9412    0.6665
0.2353    0.7515
0.2353    0.8365

```

### Return Bar Heights and Bin Centers for a Given Number of Bins

Compute the bar heights for six bins using the empirical cumulative distribution function and also return the bin centers.

Generate failure times from a Birnbaum-Saunders distribution.

```

rng('default') % for reproducibility
failuretime = random('birnbaumsaunders',0.3,1,100,1);

```

Assuming that the end of the study is at time 0.9, mark the generated failure times that are larger than 0.9 as censored data and store that information in a vector.

```

T = 0.9;
cens = (failuretime>T);

```

First, compute the empirical cumulative distribution function for the data.

```

[f,x] = ecdf(failuretime,'censoring',cens);

```

Now, estimate the histogram with six bins using the cumulative distribution function estimate.

```

[n,c] = ecdfhist(f,x,6);
[n' c']

```

```

ans =

```

```

1.9764    0.0998
1.7647    0.2415
1.1294    0.3831

```

```
0.4235    0.5248
0.7764    0.6665
0.2118    0.8081
```

## Draw Histogram for Given Bin Centers

Draw the histogram of the empirical cumulative distribution histogram for specified bin centers.

Generate failure times from a Birnbaum-Saunders distribution.

```
rng('default') % for reproducibility
failuretime = random('birnbaumsaunders',0.3,1,100,1);
```

Assuming that the end of the study is at time 0.9, mark the generated failure times that are larger than 0.9 as censored data and store that information in a vector.

```
T = 0.9;
cens = (failuretime>T);
```

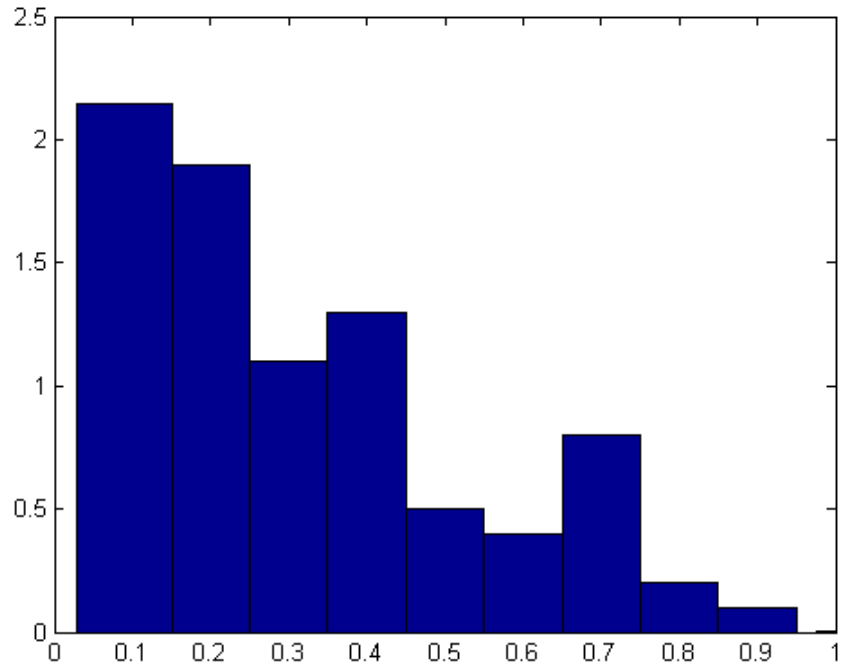
Define bin centers.

```
centers = 0.1:0.1:1;
```

Compute the empirical cumulative distribution function for the data and draw the histogram for specified bin centers.

```
[f,x] = ecdf(failuretime,'censoring',cens);
ecdfhist(f,x,centers)
axis([0 1 0 2.5])
```





### Compare Histogram with Known Probability Distribution Function

Generate right-censored survival data and compare the histogram from cumulative distribution function with the known probability distribution function.

Generate failure times from an exponential distribution with mean failure time of 15.

```
rng('default')  
y = exprnd(15,75,1);
```

## ecdfhist

---

Generate drop-out times from an exponential distribution with mean failure time of 30.

```
d = exprnd(30,75,1);
```

Record the minimum of these times as the observed failure times.

```
t = min(y,d);
```

Generate censoring by finding the generated failure times that are greater than the drop-out times.

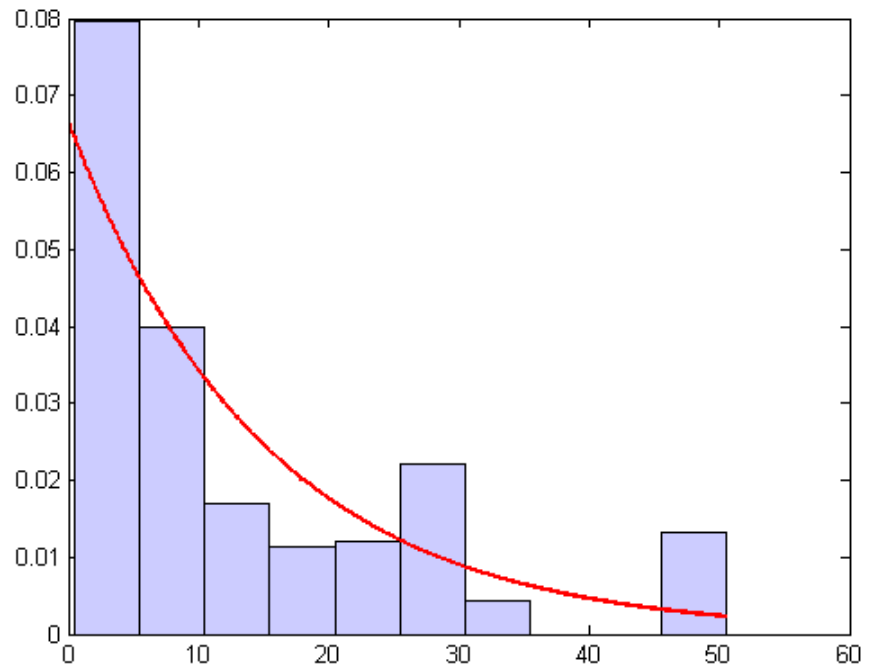
```
censored = (y>d);
```

Calculate the empirical cdf and plot a histogram using the empirical cumulative distribution function.

```
[f,x] = ecdf(t,'censoring',censored);  
ecdfhist(f,x)  
set(get(gca,'Children'),'FaceColor',[.8 .8 1])  
hold on
```

Superimpose a plot of the known population pdf.

```
xx = 0:.1:max(t);  
yy = exp(-xx/15)/15;  
plot(xx,yy,'r-','LineWidth',2)  
hold off
```



**See Also** [ecdf](#) | [hist](#) | [histc](#)

# ClassificationKNN.edge

---

<b>Purpose</b>	Edge of $k$ -nearest neighbor classifier
<b>Syntax</b>	<code>E = edge(md1,X,Y)</code> <code>E = edge(md1,X,Y,Name,Value)</code>
<b>Description</b>	<code>E = edge(md1,X,Y)</code> returns the classification edge for <code>md1</code> with data <code>X</code> and classification <code>Y</code> . <code>E = edge(md1,X,Y,Name,Value)</code> computes the edge with additional options specified by one or more <code>Name,Value</code> pair arguments.
<b>Input Arguments</b>	<b>md1</b> <i>k</i> -nearest neighbor classifier, created by <code>ClassificationKNN.fit</code> . <b>X</b> Matrix of predictor values. Each column of <code>X</code> represents one variable, and each row represents one observation. <b>Y</b> A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings, with the same number of rows as <code>X</code> . Each row of <code>Y</code> represents the classification of the corresponding row of <code>X</code> . <b>Name-Value Pair Arguments</b> Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> . <b>'weights'</b> Observation weights, a numeric vector of length <code>size(X,1)</code> . If you supply <code>weights</code> , <code>edge</code> computes weighted classification edge. <b>Default:</b> <code>ones(size(X,1))</code>

## Output Arguments

**E**

Classification edge, a scalar that is the mean classification margin (see “Margin” on page 21-757).

## Definitions

### Edge

The *edge* is the mean value of the classification *margin*.

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes.

Margin is a column vector with the same number of rows as *X*.

### Score

The *score* of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 21-2608.

## Examples

### Edge Calculation

Construct a *k*-nearest neighbor classifier for the Fisher iris data, where *k* = 5.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for five-nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',5);
```

# ClassificationKNN.edge

---

Examine the edge of the classifier for minimum, mean, and maximum observations classified 'setosa', 'versicolor', and 'virginica' respectively.

```
NewX = [min(X);mean(X);max(X)];  
Y = {'setosa';'versicolor';'virginica'};  
E = edge mdl,NewX,Y
```

```
E =
```

```
1
```

The classifier has no doubt that the Y entries are correct classifications (all five nearest neighbors of each NewX point classify as the corresponding Y entry).

## See Also

[ClassificationKNN](#) | [ClassificationKNN.fit](#) | [loss](#) | [margin](#) |

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9

## Purpose

Classification edge

## Syntax

`E = edge(obj,X,Y)`  
`E = edge(obj,X,Y,Name,Value)`

## Description

`E = edge(obj,X,Y)` returns the classification edge for `obj` with data `X` and classification `Y`.

`E = edge(obj,X,Y,Name,Value)` computes the edge with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **obj**

Discriminant analysis classifier of class `ClassificationDiscriminant` or `CompactClassificationDiscriminant`, typically constructed with `ClassificationDiscriminant.fit`.

### **X**

Matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `obj`.

### **Y**

Class labels, with the same data type as exists in `obj`. The number of elements of `Y` must equal the number of rows of `X`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'weights'**

Observation weights, a numeric vector of length `size(X,1)`. If you supply weights, `edge` computes the weighted classification edge.

# CompactClassificationDiscriminant.edge

---

**Default:** ones(size(X,1))

## Output Arguments

**E**

Edge, a scalar representing the weighted average value of the margin.

## Definitions

### Edge

The *edge* is the weighted mean value of the classification *margin*. The weights are the class probabilities in `Obj.Prior`. If you supply weights in the `weights` name-value pair, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average.

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `X`. A high value of margin indicates a more reliable prediction than a low value.

### Score (discriminant analysis)

For discriminant analysis, the *score* of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 15-7.

## Examples

Compute the classification edge and margin for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries:

```
load fisheriris
X = meas(:,1:2);
obj = ClassificationDiscriminant.fit(X,species);
E = edge(obj,X,species)
```

```
E =
    0.4980
```



```
M = margin(obj,X,species);  
M(end-10:end)
```

```
ans =  
    0.6551  
    0.4838  
    0.6551  
   -0.5127  
    0.5659  
    0.4611  
    0.4949  
    0.1024  
    0.2787  
   -0.1439  
   -0.4444
```

The classifier trained on all the data is better:

```
obj = ClassificationDiscriminant.fit(meas,species);  
E = edge(obj,meas,species)
```

```
E =  
    0.9454
```

```
M = margin(obj,meas,species);  
M(end-10:end)
```

```
ans =  
    0.9983  
    1.0000  
    0.9991  
    0.9978  
    1.0000  
    1.0000  
    0.9999  
    0.9882
```

# CompactClassificationDiscriminant.edge

---

0.9937  
1.0000  
0.9649

## See Also

ClassificationDiscriminant | loss | margin | predict

## How To

- “Discriminant Analysis” on page 15-3

## Purpose

Classification edge

## Syntax

```
E = edge(ens,X,Y)
E = edge(ens,X,Y,Name,Value)
```

## Description

`E = edge(ens,X,Y)` returns the classification edge for `ens` with data `X` and classification `Y`.

`E = edge(ens,X,Y,Name,Value)` computes the edge with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A classification ensemble constructed with `fitensemble`, or a compact classification ensemble constructed with `compact`.

### **X**

A matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `ens`.

### **Y**

Class labels, with the same data type as exists in `ens`. The number of elements of `Y` must equal the number of rows of `X`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'learners'**

Indices of weak learners in the ensemble ranging from 1 to `ens.NTrained`. `edge` uses only these learners for calculating loss.

**Default:** 1:NTrained

# CompactClassificationEnsemble.edge

---

## 'mode'

String representing the meaning of the output E:

- 'ensemble' — E is a scalar value, the edge for the entire ensemble.
- 'individual' — E is a vector with one element per trained learner.
- 'cumulative' — E is a vector in which element J is obtained by using learners 1:J from the input list of learners.

**Default:** 'ensemble'

## 'UseObsForLearner'

A logical matrix of size N-by-T, where:

- N is the number of rows of X.
- T is the number of weak learners in `ens`.

When `UseObsForLearner(i,j)` is true, learner j is used in predicting the class of row i of X.

**Default:** `true(N,T)`

## 'weights'

Observation weights, a numeric vector of length `size(X,1)`. If you supply weights, `edge` computes weighted classification edge.

**Default:** `ones(size(X,1))`

## Output Arguments

### E

The classification edge, a vector or scalar depending on the setting of the mode name-value pair. Classification edge is weighted average classification margin.

## Definitions

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix  $X$ .

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

### Edge

The *edge* is the weighted mean value of the classification margin. The weights are the class probabilities in `ens.Prior`. If you supply weights in the `weights` name-value pair, those weights are used instead of class probabilities.

## Examples

Make a boosted ensemble classifier for the `ionosphere` data, and find the classification edge for the last few rows:

```
load ionosphere
ens = fitensemble(X,Y,'AdaboostM1',100,'Tree');
E = edge(ens,X(end-10:end,:),Y(end-10:end))
```

```
E =
    8.3310
```

## See Also

`margin` | `edge`

# CompactClassificationTree.edge

---

<b>Purpose</b>	Classification edge
<b>Syntax</b>	<code>E = edge(tree,X,Y)</code> <code>E = edge(tree,X,Y,Name,Value)</code>
<b>Description</b>	<code>E = edge(tree,X,Y)</code> returns the classification edge for <code>tree</code> with data <code>X</code> and classification <code>Y</code> . <code>E = edge(tree,X,Y,Name,Value)</code> computes the edge with additional options specified by one or more <code>Name,Value</code> pair arguments.
<b>Input Arguments</b>	<b>tree</b> A classification tree created by <code>ClassificationTree.fit</code> , or a compact classification tree created by <code>compact</code> . <b>X</b> A matrix where each row represents an observation, and each column represents a predictor. The number of columns in <code>X</code> must equal the number of predictors in <code>tree</code> . <b>Y</b> Class labels, with the same data type as exists in <code>tree</code> . The number of elements of <code>Y</code> must equal the number of rows of <code>X</code> . <b>Name-Value Pair Arguments</b> Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> . <b>'weights'</b> Observation weights, a numeric vector of length <code>size(X,1)</code> . If you supply <code>weights</code> , <code>edge</code> computes weighted classification edge. <b>Default:</b> <code>ones(size(X,1))</code>

## Output Arguments

**E**

The edge, a scalar representing the weighted average value of the margin.

## Definitions

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as the matrix  $X$ .

### Score (tree)

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

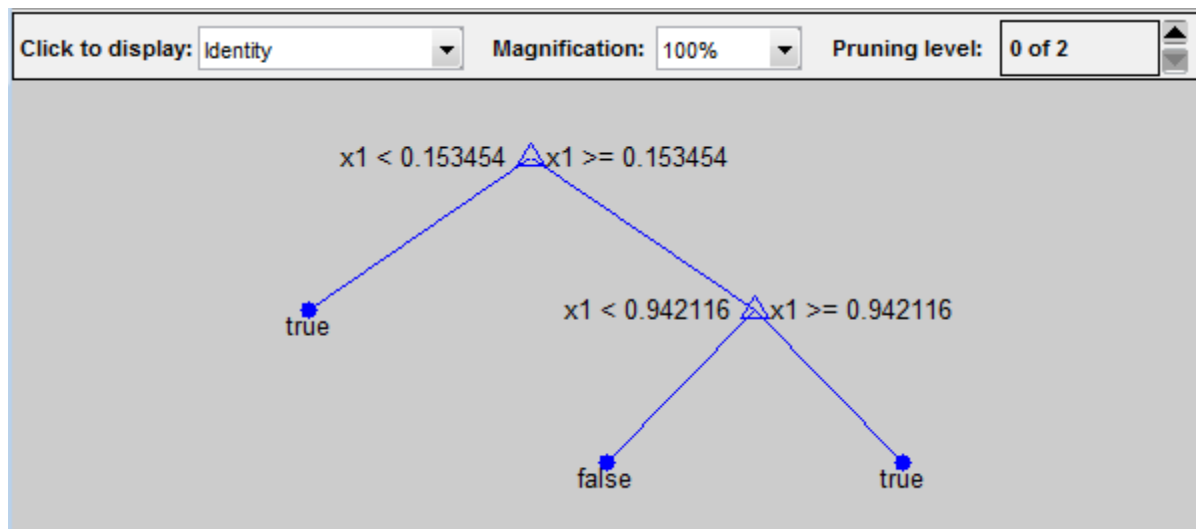
For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

**1**

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```

# CompactClassificationTree.edge

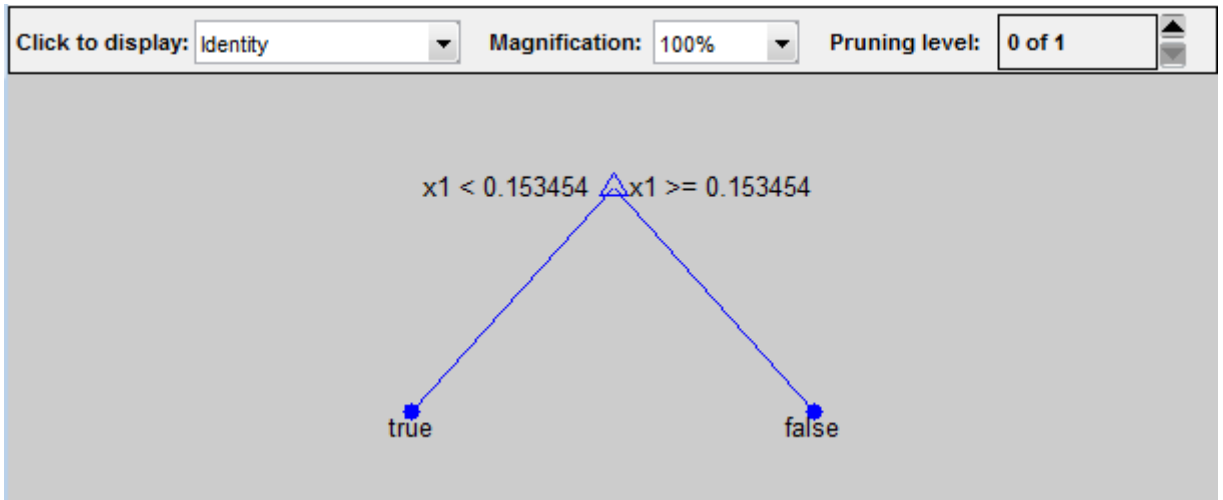


**2**

Prune the tree:

```
tree1 = prune(tree, 'level', 1);
view(tree1, 'mode', 'graph')
```





The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =
    0.9059    0.0941    0.8147
    0.9059    0.0941    0.9058
         0     1.0000    0.1270
    0.9059    0.0941    0.9134
    0.9059    0.0941    0.6324
         0     1.0000    0.0975
    0.9059    0.0941    0.2785
```

# CompactClassificationTree.edge

---

```
0.9059    0.0941    0.5469
0.9059    0.0941    0.9575
0.9059    0.0941    0.9649
```

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range (.95, 1) instead of the expected 5 observations.

## Edge

The *edge* is the weighted mean value of the classification margin. The weights are the class probabilities in `tree.Prior`. If you supply weights in the `weights` name-value pair, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average.

## Examples

Compute the classification margin and edge for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries:

```
load fisheriris
X = meas(:,1:2);
tree = ClassificationTree.fit(X,species);
E = edge(tree,X,species)

E =
    0.6299

M = margin(tree,X,species);
M(end-10:end)

ans =
    0.1111
    0.1111
    0.1111
   -0.2857
```

```
0.6364
0.6364
0.1111
0.7500
1.0000
0.6364
0.2000
```

The classification tree trained on all the data is better:

```
tree = ClassificationTree.fit(meas,species);
E = edge(tree,meas,species)
```

```
E =
    0.9384
```

```
M = margin(tree,meas,species);
M(end-10:end)
```

```
ans =
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
    0.9565
```

## See Also

[margin](#) | [loss](#) | [predict](#)

# categorical.end

---

<b>Purpose</b>	Last index in indexing expression for categorical array
<b>Syntax</b>	<code>end(A,k,n)</code>
<b>Description</b>	<code>end(A,k,n)</code> indexes expressions involving the categorical array <code>A</code> when <code>end</code> is part of the <code>k</code> -th index out of <code>n</code> indices. For example, the expression <code>A(end-1,:)</code> calls <code>A</code> 's <code>end</code> method with <code>end(A,1,2)</code> .
<b>See Also</b>	<code>single</code>

<b>Purpose</b>	Last index in indexing expression for dataset array
<b>Syntax</b>	<code>end(A,k,n)</code>
<b>Description</b>	<code>end(A,k,n)</code> is called for indexing expressions involving the dataset <code>A</code> when <code>end</code> is part of the <code>k</code> -th index out of <code>n</code> indices. For example, the expression <code>A(end-1,:)</code> calls <code>A</code> 's <code>end</code> method with <code>end(A,1,2)</code> .
<b>See Also</b>	<code>size</code>

# grandset.end

---

**Purpose** Last index in indexing expression for point set

**Syntax** `end(p,k,n)`

**Description** `end(p,k,n)` is called for indexing expressions involving the point set `p` when `end` is part of the `k`-th index out of `n` indices. For example, the expression `p(end-1,:)` calls `p`'s `end` method with `end(p,1,2)`.

**See Also** `grandset`

## Purpose

Extreme value cumulative distribution function

## Syntax

```
p = evcdf(x,mu,sigma)
[p,plo,pup] = evcdf(x,mu,sigma,pcov,alpha)
[p,plo,pup] = evcdf( __ , 'upper' )
```

## Description

`p = evcdf(x,mu,sigma)` returns the cumulative distribution function (cdf) for the type 1 extreme value distribution, with location parameter `mu` and scale parameter `sigma`, at each of the values in `x`. `x`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are 0 and 1, respectively.

`[p,plo,pup] = evcdf(x,mu,sigma,pcov,alpha)` returns confidence bounds for `p` when the input parameters `mu` and `sigma` are estimates. `pcov` is a 2-by-2 covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1 - `alpha`)% confidence bounds. `plo` and `pup` are arrays of the same size as `p`, containing the lower and upper confidence bounds.

`[p,plo,pup] = evcdf( __ , 'upper' )` returns the complement of the type 1 extreme value distribution cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the 'upper' argument with any of the previous syntaxes.

The function `evcdf` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\frac{X - \hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `P`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating  $X$ . See “Extreme Value Distribution” on page B-35 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## See Also

`cdf` | `evpdf` | `evinv` | `evstat` | `evfit` | `evlike` | `evrnd`

## How To

- “Extreme Value Distribution” on page B-35



**Purpose**

Extreme value parameter estimates

**Syntax**

```
parmhat = evfit(data)
[parmhat,parmci] = evfit(data)
[parmhat,parmci] = evfit(data,alpha)
[...] = evfit(data,alpha,censoring)
[...] = evfit(data,alpha,censoring,freq)
[...] = evfit(data,alpha,censoring,freq,options)
```

**Description**

`parmhat = evfit(data)` returns maximum likelihood estimates of the parameters of the type 1 extreme value distribution given the data in the vector `data`. `parmhat(1)` is the location parameter,  $\mu$ , and `parmhat(2)` is the scale parameter,  $\sigma$ .

`[parmhat,parmci] = evfit(data)` returns 95% confidence intervals for the parameter estimates on the  $\mu$  and  $\sigma$  parameters in the 2-by-2 matrix `parmci`. The first column of the matrix of the extreme value fit contains the lower and upper confidence bounds for the parameter  $\mu$ , and the second column contains the confidence bounds for the parameter  $\sigma$ .

`[parmhat,parmci] = evfit(data,alpha)` returns  $100(1 - \text{alpha})\%$  confidence intervals for the parameter estimates, where `alpha` is a value in the range  $[0 \ 1]$  specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals.

`[...] = evfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = evfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq` of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = evfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood

estimates. You can create options using the function `statset`. Enter `statset('evfit')` to see the names and default values of the parameters that `evfit` accepts in the options structure. See the reference page for `statset` for more information about these options.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating  $x$ . See “Extreme Value Distribution” on page B-35 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

## See Also

`mle` | `evlike` | `evpdf` | `evcdf` | `evinv` | `evstat` | `evrnd`

## How To

- “Extreme Value Distribution” on page B-35

**Purpose**

Extreme value inverse cumulative distribution function

**Syntax**

`X = evinv(P,mu,sigma)`  
`[X,XLO,XUP] = evinv(P,mu,sigma,pcov,alpha)`

**Description**

`X = evinv(P,mu,sigma)` returns the inverse cumulative distribution function (cdf) for a type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`, evaluated at the values in `P`. `P`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `mu` and `sigma` are 0 and 1, respectively.

`[X,XLO,XUP] = evinv(P,mu,sigma,pcov,alpha)` produces confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` is a scalar that specifies  $100(1 - \alpha)\%$  confidence bounds for the estimated parameters, and has a default value of 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `evinv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from an extreme value distribution with parameters  $\mu = 0$  and  $\sigma = 1$ . The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating `X`. See “Extreme Value Distribution” on page B-35 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

# evinv

---

## **See Also**

`icdf` | `evcdf` | `evpdf` | `evstat` | `evfit` | `evlike` | `evrnd`

**Purpose** Test handle equality

**Syntax**  
`h1 == h2`  
`tf = eq(h1, h2)`

**Description** `h1 == h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise equality result. If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = eq(h1, h2)` stores the result in a logical array of the same dimensions.

**See Also** `grandstream` | `ge` | `gt` | `le` | `lt` | `ne`

# CompactTreeBagger.error

---

**Purpose** Error (misclassification probability or MSE)

**Syntax**  
`err = error(B,X,Y)`  
`err = error(B,X,Y, 'param1',val1, 'param2',val2, ...)`

**Description** `err = error(B,X,Y)` computes the misclassification probability (for classification trees) or mean squared error (MSE, for regression trees) for each tree, for predictors `X` given true response `Y`. For classification, `Y` can be either a numeric vector, character matrix, cell array of strings, categorical vector or logical vector. For regression, `Y` must be a numeric vector. `err` is a vector with one error measure for each of the `NTrees` trees in the ensemble `B`.

`err = error(B,X,Y, 'param1',val1, 'param2',val2, ...)` specifies optional parameter name/value pairs:

`'mode'` String indicating how the method computes errors. If set to `'cumulative'` (default), error computes cumulative errors and `err` is a vector of length `NTrees`, where the first element gives error from `trees(1)`, second element gives error from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to `'individual'`, `err` is a vector of length `NTrees`, where each element is an error from each tree in the ensemble. If set to `'ensemble'`, `err` is a scalar showing the cumulative error for the entire ensemble.

`'trees'` Vector of indices indicating what trees to include in this calculation. By default, this argument is set to `'all'` and the method uses all trees. If `'trees'` is a numeric vector, the method returns a vector of length `NTrees` for `'cumulative'` and `'individual'` modes, where `NTrees` is the number of elements in the input vector, and a scalar for `'ensemble'` mode. For example, in the `'cumulative'` mode, the first element gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.

- 'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
- 'useifort' Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

### See Also

`TreeBagger.error`

# TreeBagger.error

---

**Purpose** Error (misclassification probability or MSE)

**Syntax**  
`err = error(B,X,Y)`  
`err = error(B,X,Y, 'param1',val1, 'param2',val2, ...)`

**Description** `err = error(B,X,Y)` computes the misclassification probability for classification trees or mean squared error (MSE) for regression trees for each tree, for predictors X given true response Y. For classification, Y can be either a numeric vector, character matrix, cell array of strings, categorical vector or logical vector. For regression, Y must be a numeric vector. `err` is a vector with one error measure for each of the `NTrees` trees in the ensemble B.

`err = error(B,X,Y, 'param1',val1, 'param2',val2, ...)` specifies optional parameter name/value pairs:

`'mode'` String indicating how the method computes errors. If set to `'cumulative'` (default), error computes cumulative errors and `err` is a vector of length `NTrees`, where the first element gives error from `trees(1)`, second element gives error from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to `'individual'`, `err` is a vector of length `NTrees`, where each element is an error from each tree in the ensemble. If set to `'ensemble'`, `err` is a scalar showing the cumulative error for the entire ensemble.

`'trees'` Vector of indices indicating what trees to include in this calculation. By default, this argument is set to `'all'` and the method uses all trees. If `'trees'` is a numeric vector, the method returns a vector of length `NTrees` for `'cumulative'` and `'individual'` modes, where `NTrees` is the number of elements in the input vector, and a scalar for `'ensemble'` mode. For example, in the `'cumulative'` mode, the first element gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.



- 'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
- 'useifort' Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

### See Also

`CompactTreeBagger.error`

# classregtree.eval

---

**Purpose** Predicted responses

**Syntax**

```
yfit = eval(t,X)
yfit = eval(t,X,s)
[yfit,nodes] = eval(...)
[yfit,nodes,cnums] = eval(...)
[...] = t(X)
[...] = t(X,s)
```

**Description** `yfit = eval(t,X)` takes a classification or regression tree `t` and a matrix `X` of predictors, and produces a vector `yfit` of predicted response values. For a regression tree, `yfit(i)` is the fitted response value for a point having the predictor values `X(i,:)`. For a classification tree, `yfit(i)` is the class into which the tree assigns the point with data `X(i,:)`.

`yfit = eval(t,X,s)` takes an additional vector `s` of pruning levels, with 0 representing the full, unpruned tree. `t` must include a pruning sequence as created by `classregtree` or by `prune`. If `s` has  $k$  elements and `X` has  $n$  rows, the output `yfit` is an  $n$ -by- $k$  matrix, with the  $j$ th column containing the fitted values produced by the `s(j)` subtree. `s` must be sorted in ascending order.

To compute fitted values for a tree that is not part of the optimal pruning sequence, first use `prune` to prune the tree.

`[yfit,nodes] = eval(...)` also returns a vector `nodes` the same size as `yfit` containing the node number assigned to each row of `X`. Use `view` to display the node numbers for any node you select.

`[yfit,nodes,cnums] = eval(...)` is valid only for classification trees. It returns a vector `cnum` containing the predicted class numbers.

NaN values in `X` are treated as missing. If `eval` encounters a missing value when it attempts to evaluate the split rule at a branch node, it cannot determine whether to proceed to the left or right child node. Instead, it sets the corresponding fitted value equal to the fitted value assigned to the branch node.

`[...] = t(X)` or `[...] = t(X,s)` also invoke `eval`.

## Examples

Create a classification tree for Fisher's iris data:

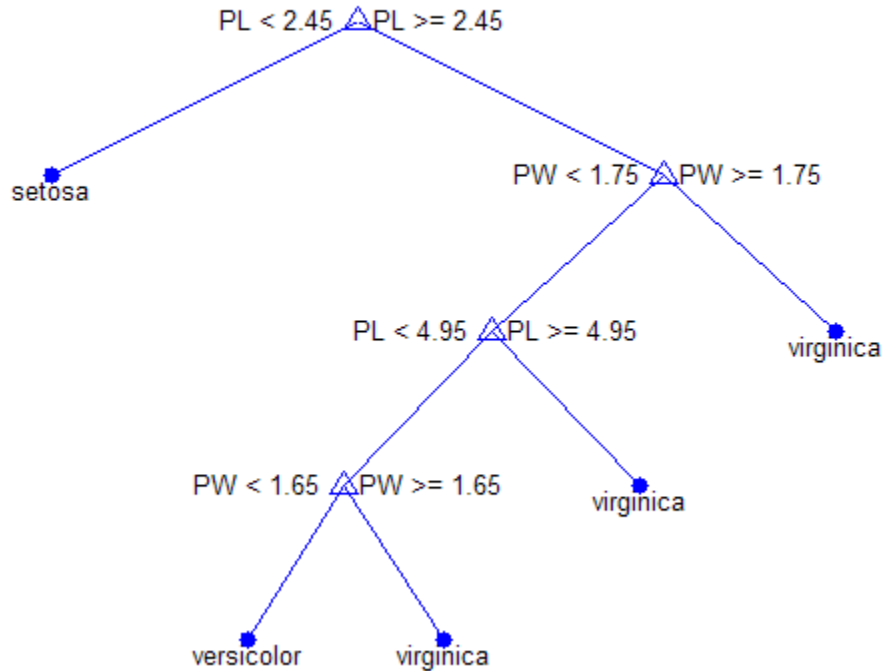
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

# classregtree.eval

Click to display:  Magnification:  Pruning level:



Find assigned class names:

```
sfit = eval(t,meas);
```

Compute that proportion is correctly classified:

```
pct = mean(strcmp(sfit,species))  
pct =  
    0.9800
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `prune` | `test` | `view`

# evalclusters

---

## Purpose

Evaluate clustering solutions

## Syntax

```
eva = evalclusters(x,clust,criterion)
eva = evalclusters(x,clust,criterion,Name,Value)
```

## Description

`eva = evalclusters(x,clust,criterion)` creates a clustering evaluation object containing data used to evaluate the optimal number of data clusters.

`eva = evalclusters(x,clust,criterion,Name,Value)` creates a clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

### **x - Input data**

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

### **Data Types**

single | double

### **clust - Clustering algorithm**

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

'kmeans'	Cluster the data in <code>x</code> using the <code>kmeans</code> clustering algorithm, with <code>'EmptyAction'</code> set to <code>'singleton'</code> and <code>'Replicates'</code> set to 5.
'linkage'	Cluster the data in <code>x</code> using the <code>clusterdata</code> agglomerative clustering algorithm, with <code>'Linkage'</code> set to <code>'ward'</code> .
'gmdistribution'	Cluster the data in <code>x</code> using the <code>gmdistribution</code> Gaussian mixture distribution algorithm, with <code>'SharedCov'</code> set to <code>true</code> and <code>'Replicates'</code> set to 5.

If `Criterion` is `'CalinskHarabasz'`, `'DaviesBouldin'`, or `'silhouette'`, you can specify a clustering algorithm using the `function_handle` (`@`) operator. The function must be of the form

```
C = clustfun(DATA,K)
```

where `DATA` is the data to be clustered, and `K` is the number of clusters. The output of `clustfun` must be one of the following:

- A vector of integers representing the cluster index for each observation in `DATA`. There must be `K` unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If `Criterion` is `'CalinskHarabasz'`, `'DaviesBouldin'`, or `'silhouette'`, you can also specify `clust` as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

### **criterion - Clustering evaluation criterion**

```
'CalinskiHarabasz' | 'DaviesBouldin' | 'gap' | 'silhouette'
```

Clustering evaluation criterion, specified as one of the following.

'CalinskiHarabasz'	Create a <code>CalinskiHarabaszEvaluation</code> clustering evaluation object containing Calinski-Harabasz index values.
'DaviesBouldin'	Create a <code>DaviesBouldinEvaluation</code> cluster evaluation object containing Davies-Bouldin index values.
'gap'	Create a <code>GapEvaluation</code> cluster evaluation object containing gap criterion values.
'silhouette'	Create a <code>SilhouetteEvaluation</code> cluster evaluation object containing silhouette values.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'KList', [1:5], 'Distance', 'cityblock'` specifies to test 1, 2, 3, 4, and 5 clusters using the sum of absolute differences distance measure.

## For All Criteria

### 'KList' - List of number of clusters to evaluate

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of 'KList' and a vector of positive integer values. You must specify `KList` when `clust` is a clustering algorithm name string



or a function handle. When `criterion` is 'gap', `clust` must be a string or a function handle, and you must specify `KList`.

**Example:** 'KList',[1:6]

## For Silhouette and Gap

### 'Distance' - Distance metric

'sqEuclidean' (default) | 'Euclidean' | 'cityblock' | vector | function | ...

Distance metric used for computing the criterion values, specified as the comma-separated pair consisting of 'Distance' and one of the following.

'sqEuclidean'	Squared Euclidean distance
'Euclidean'	Euclidean distance
'cityblock'	Sum of absolute differences
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values)
'Hamming'	Percentage of coordinates that differ
'Jaccard'	Percentage of nonzero coordinates that differ

For detailed information about each distance metric, see `pdist`.

You can also specify a function for the distance metric by using the `function_handle` (@) operator. The distance function must be of the form

```
d2 = distfun(XI,XJ),
```

where `XI` is a 1-by- $n$  vector corresponding to a single row of the input matrix `X`, and `XJ` is an  $m_2$ -by- $n$  matrix corresponding to multiple rows of

X. `distfun` must return an  $m_2$ -by-1 vector of distances `d2`, whose  $k$ th element is the distance between `XI` and `XJ(k, :)`.

If `Criterion` is `'silhouette'`, you can also specify `Distance` as the output vector output created by the function `pdist`.

When `Clust` a string representing a built-in clustering algorithm, `evalclusters` uses the distance metric specified for `Distance` to cluster the data, except for the following:

- If `Clust` is `'linkage'`, and `Distance` is either `'sqEuclidean'` or `'Euclidean'`, then the clustering algorithm uses Euclidean distance and Ward linkage.
- If `Clust` is `'linkage'` and `Distance` is any other metric, then the clustering algorithm uses the specified distance metric and average linkage.

In all other cases, the distance metric specified for `Distance` must match the distance metric used in the clustering algorithm to obtain meaningful results.

**Example:** `'Distance', 'Euclidean'`

## For Silhouette Only

### **'ClusterPriors' - Prior probabilities for each cluster**

`'empirical'` (default) | `'equal'`

Prior probabilities for each cluster, specified as the comma-separated pair consisting of `'ClusterPriors'` and one of the following.

'empirical'	Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points. Each cluster contributes to the overall silhouette value proportionally to its size.
'equal'	Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points within each cluster, and then averaging those values across all clusters. Each cluster contributes equally to the overall silhouette value, regardless of its size.

**Example:** 'ClusterPriors', 'empirical'

### For Gap Only

#### 'B' - Number of reference data sets

100 (default) | positive integer value

Number of reference data sets generated from the reference distribution `ReferenceDistribution`, specified as the comma-separated pair consisting of 'B' and a positive integer value.

**Example:** 'B', 150

#### 'ReferenceDistribution' - Reference data generation method

'PCA' (default) | 'uniform'

Reference data generation method, specified as the comma-separated pair consisting of 'ReferenceDistributions' and one of the following.

'PCA'	Generate reference data from a uniform distribution over a box aligned with the principal components of the data matrix $x$ .
'uniform'	Generate reference data uniformly over the range of each feature in the data matrix $x$ .

**Example:** 'ReferenceDistribution', 'uniform'

## 'SearchMethod' - Method for selecting optimal number of clusters

'firstmaxse' | 'globalmaxse'

Method for selecting the optimal number of clusters, specified as the comma-separated pair consisting of 'SearchMethod' and one of the following.

'firstmaxse'

Evaluate each proposed number of clusters in `KList` and select the smallest number of clusters satisfying

$$\text{Gap}(K) \geq \text{Gap}(K + 1) - \text{SE}(K + 1),$$

where  $K$  is the number of clusters,  $\text{Gap}(K)$  is the gap value for the clustering solution with  $K$  clusters, and  $\text{SE}(K + 1)$  is the standard error of the clustering solution with  $K + 1$  clusters. This is the default if the sorted `KList` contains consecutive integer values.

'globalmaxse'

Evaluate each proposed number of clusters in `KList` and select the smallest number of clusters satisfying

$$\text{Gap}(K) \geq \text{GAPMAX} - \text{SE}(\text{GAPMAX}),$$

where  $K$  is the number of clusters,  $\text{Gap}(K)$  is the gap value for the clustering solution with  $K$  clusters,  $\text{GAPMAX}$  is the largest gap value, and  $\text{SE}(\text{GAPMAX})$  is the standard error corresponding to the largest gap value. This is the default if `KList` contains only one value, or contains any nonconsecutive integer values.

**Example:** 'SearchMethod', 'globalmaxse'

**Output Arguments****eva - Clustering evaluation data**

clustering evaluation object

Clustering evaluation data, returned as a clustering evaluation object.

**Examples****Evaluate the Clustering Solution Using Calinski-Harabasz Criterion**

Evaluate the optimal number of clusters using the Calinski-Harabasz clustering evaluation criterion.

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Evaluate the optimal number of clusters using the Calinski-Harabasz criterion. Cluster the data using kmeans.

```
rng('default'); % For reproducibility
eva = evalclusters(meas,'kmeans','CalinskiHarabasz','KList',[1:6])
```

```
eva =
```

```
CalinskiHarabaszEvaluation with properties:
```

```
NumObservations: 150
InspectecedK: [1 2 3 4 5 6]
CriterionValues: [1x6 double]
OptimalK: 3
```

The OptimalK value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

**Evaluate a Matrix of Clustering Solutions**

Use an input matrix of proposed clustering solutions to evaluate the optimal number of clusters.

## evalclusters

---

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Use `kmeans` to create an input matrix of proposed clustering solutions for the sepal length measurements, using 1, 2, 3, 4, 5, and 6 clusters.

```
clust = zeros(size(meas,1),6);  
for i=1:6  
    clust(:,i) = kmeans(meas,i,'emptyaction','singleton',...  
        'replicate',5);  
end
```

Each row of `clust` corresponds to one sepal length measurement. Each of the six columns corresponds to a clustering solution containing 1 to 6 clusters.

Evaluate the optimal number of clusters using the Calinski-Harabasz criterion.

```
eva = evalclusters(meas,clust,'CalinskiHarabasz')
```

```
eva =
```

```
CalinskiHarabaszEvaluation with properties:
```

```
NumObservations: 150  
InspectedK: [1 2 3 4 5 6]  
CriterionValues: [NaN 513.9245 561.6278 530.7658 459.5058 473.6577]  
OptimalK: 3
```

The `OptimalK` value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

## Specify Clustering Algorithm with a Function Handle

Use a function handle to specify the clustering algorithm, then evaluate the optimal number of clusters.

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Use a function handle to specify the clustering algorithm.

```
myfunc = @(X,K)(kmeans(X, K, 'emptyaction','singleton',...  
    'replicate',5));
```

Evaluate the optimal number of clusters for the sepal length data using the Calinski-Harabasz criterion.

```
eva = evalclusters(meas,myfunc,'CalinskiHarabasz',...  
    'klist',[1:6])
```

```
eva =
```

```
CalinskiHarabaszEvaluation with properties:
```

```
NumObservations: 150
```

```
InspectedK: [1 2 3 4 5 6]
```

```
CriterionValues: [NaN 513.9245 561.6278 530.7658 459.5058 473.6577]
```

```
OptimalK: 3
```

The OptimalK value indicates that, based on the Calinski-Harabasz criterion, the optimal number of clusters is three.

## See Also

```
clustering.evaluation.CalinskiHarabaszEvaluation  
| clustering.evaluation.DaviesBouldinEvaluation  
| clustering.evaluation.GapEvaluation |  
clustering.evaluation.SilhouetteEvaluation |
```

## **Concepts**

- “k-Means Clustering” on page 14-22
- “Gaussian Mixture Models” on page 14-30
- “Hierarchical Clustering” on page 14-3



---

<b>Purpose</b>	Extreme value negative log-likelihood
<b>Syntax</b>	<pre>nlogL = evlike(params,data) [nlogL,AVAR] = evlike(params,data) [...] = evlike(params,data,censoring) [...] = evlike(params,data,censoring,freq)</pre>
<b>Description</b>	<p><code>nlogL = evlike(params,data)</code> returns the negative of the log-likelihood for the type 1 extreme value distribution. <code>params(1)</code> is the tail location parameter, <code>mu</code>, and <code>params(2)</code> is the scale parameter, <code>sigma</code>. <code>nlogL</code> is a scalar.</p> <p><code>[nlogL,AVAR] = evlike(params,data)</code> returns the inverse of Fisher's information matrix, <code>AVAR</code>. If the input parameter values in <code>params</code> are the maximum likelihood estimates, the diagonal elements of <code>AVAR</code> are their asymptotic variances. <code>AVAR</code> is based on the observed Fisher's information, not the expected information.</p> <p><code>[...] = evlike(params,data,censoring)</code> accepts a Boolean vector of the same size as <code>data</code>, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.</p> <p><code>[...] = evlike(params,data,censoring,freq)</code> accepts a frequency vector of the same size as <code>data</code>. <code>freq</code> typically contains integer frequencies for the corresponding elements in <code>data</code>, but can contain any nonnegative values. Pass in <code>[]</code> for <code>censoring</code> to use its default value.</p> <p>The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating data. See “Extreme Value Distribution” on page B-35 for more details. If <math>x</math> has a Weibull distribution, then <math>X = \log(x)</math> has the type 1 extreme value distribution.</p>
<b>See Also</b>	<code>evfit</code>   <code>evpdf</code>   <code>evcdf</code>   <code>evinv</code>   <code>evstat</code>   <code>evrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Extreme Value Distribution” on page B-35</li></ul>

# evpdf

---

**Purpose** Extreme value probability density function

**Syntax**  $Y = \text{evpdf}(X, \mu, \sigma)$

**Description**  $Y = \text{evpdf}(X, \mu, \sigma)$  returns the pdf of the type 1 extreme value distribution with location parameter  $\mu$  and scale parameter  $\sigma$ , evaluated at the values in  $X$ .  $X$ ,  $\mu$ , and  $\sigma$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for  $\mu$  and  $\sigma$  are 0 and 1, respectively.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating  $X$ . See “Extreme Value Distribution” on page B-35 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

**See Also** pdf | evcdf | evinv | evstat | evfit | evlike | evrnd

**How To** • “Extreme Value Distribution” on page B-35

---

<b>Purpose</b>	Extreme value random numbers
<b>Syntax</b>	<pre>R = evrnd(mu,sigma) R = evrnd(mu,sigma,m,n,...) R = evrnd(mu,sigma,[m,n,...])</pre>
<b>Description</b>	<p><code>R = evrnd(mu,sigma)</code> generates random numbers from the extreme value distribution with parameters specified by location parameter <code>mu</code> and scale parameter <code>sigma</code>. <code>mu</code> and <code>sigma</code> can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of <code>R</code>. A scalar input for <code>mu</code> or <code>sigma</code> is expanded to a constant array with the same dimensions as the other input.</p> <p><code>R = evrnd(mu,sigma,m,n,...)</code> or <code>R = evrnd(mu,sigma,[m,n,...])</code> generates an <code>m-by-n-by-...</code> array containing random numbers from the extreme value distribution with parameters <code>mu</code> and <code>sigma</code>. <code>mu</code> and <code>sigma</code> can each be scalars or arrays of the same size as <code>R</code>.</p> <p>The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima by negating <code>R</code>. See “Extreme Value Distribution” on page B-35 for more details. If <math>x</math> has a Weibull distribution, then <math>X = \log(x)</math> has the type 1 extreme value distribution.</p>
<b>See Also</b>	<code>random</code>   <code>evpdf</code>   <code>evcdf</code>   <code>evinv</code>   <code>evstat</code>   <code>evfit</code>   <code>evlike</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Extreme Value Distribution” on page B-35</li></ul>

# evstat

---

**Purpose** Extreme value mean and variance

**Syntax** `[M,V] = evstat(mu,sigma)`

**Description** `[M,V] = evstat(mu,sigma)` returns the mean of and variance for the type 1 extreme value distribution with location parameter `mu` and scale parameter `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other input. The default values for `mu` and `sigma` are 0 and 1, respectively.

The type 1 extreme value distribution is also known as the Gumbel distribution. The version used here is suitable for modeling minima; the mirror image of this distribution can be used to model maxima. See “Extreme Value Distribution” on page B-35 for more details. If  $x$  has a Weibull distribution, then  $X = \log(x)$  has the type 1 extreme value distribution.

**See Also** `evpdf` | `evcdf` | `evinv` | `evfit` | `evlike` | `evrnd`

**How To**

- “Extreme Value Distribution” on page B-35

<b>Superclasses</b>	ToolboxFittableParametricDistribution
<b>Purpose</b>	Exponential probability distribution object
<b>Description</b>	<p>prob.ExponentialDistribution is an object consisting of parameters, a model description, and sample data for an exponential probability distribution.</p> <p>Create a probability distribution object with specified parameter values using <code>makedist</code>. Alternatively, fit a distribution to data using <code>fitdist</code> or the Distribution Fitting app.</p>
<b>Construction</b>	<p><code>pd = makedist('Exponential')</code> creates an exponential probability distribution object using the default parameter values.</p> <p><code>pd = makedist('Exponential','mu',mu)</code> creates an exponential probability distribution object using the specified parameter value.</p>
	<b>Input Arguments</b>
	<b>mu - Mean</b> 1 (default)   positive scalar value Mean of the exponential distribution, specified as a positive scalar value.
	<b>Data Types</b> single   double
<b>Properties</b>	<b>mu</b> Mean of the exponential distribution, stored as a positive scalar value.
	<b>Data Types</b> single   double
	<b>DistributionName</b> Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

# prob.ExponentialDistribution

---

## Data Types

char

## InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the

estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

**Data Types**

single | double

**ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

**Data Types**

char

**ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

**Data Types**

logical

**ParameterNames**

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

**Data Types**

char

**ParameterValues**

Values of distribution parameters, stored as a vector. This property is read-only.

**Data Types**

single | double

**Truncation**

# prob.ExponentialDistribution

---

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

<code>cdf</code>	Cumulative distribution function of probability distribution object
<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
<code>iqr</code>	Interquartile range of probability distribution object
<code>median</code>	Median of probability distribution object
<code>pdf</code>	Probability density function of probability distribution object
<code>random</code>	Generate random numbers from probability distribution object
<code>truncate</code>	Truncate probability distribution object
<code>mean</code>	Mean of probability distribution object
<code>negloglik</code>	Negative loglikelihood of probability distribution object
<code>paramci</code>	Confidence intervals for probability distribution parameters



proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Exponential Distribution

The exponential distribution is used to model events that occur randomly over time, and its main application area is studies of lifetimes. It is a special case of the gamma distribution with the shape parameter  $\alpha = 1$ .

The exponential distribution uses the following parameters.

Parameter	Description	Support
mu	Mean	$\mu > 0$

The probability density function (pdf) is

$$f(x | \mu) = \frac{1}{\mu} e^{-\frac{x}{\mu}} ; x \geq 0.$$

## Examples

### Create an Exponential Distribution Object Using Default Parameters

Create an exponential distribution object using the default parameter values.

```
pd = makedist('Exponential')
```

```
pd =
```

# prob.ExponentialDistribution

---

```
ExponentialDistribution
```

```
Exponential distribution  
mu = 1
```

## Create an Exponential Distribution Object Using Specified Parameters

Create an exponential distribution object by specifying the parameter values.

```
pd = makedist('Exponential','mu',2)
```

```
pd =
```

```
ExponentialDistribution
```

```
Exponential distribution  
mu = 2
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =
```

```
4
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Exponential Distribution” on page B-32
- Class Attributes
- Property Attributes

**Purpose** Exponential cumulative distribution function

**Syntax**

```
p = expcdf(x,mu)
[p,plo,pup] = expcdf(x,mu,pcov,alpha)
[p,plo,pup] = expcdf( __ , 'upper' )
```

**Description** `p = expcdf(x,mu)` computes the exponential cdf at each of the values in `x` using the corresponding mean parameter `mu`. `x` and `mu` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive.

`[p,plo,pup] = expcdf(x,mu,pcov,alpha)` produces confidence bounds for `Pp` when the input mean parameter `mu` is an estimate. `pcov` is the variance of the estimated `mu`. `alpha` specifies `100(1 - alpha)%` confidence bounds. The default value of `alpha` is `0.05`. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds. The bounds are based on a normal approximation for the distribution of the log of the estimate of `mu`. If you estimate `mu` from a set of data, you can get a more accurate set of bounds by applying `expfit` to the data to get a confidence interval for `mu`, and then evaluating `expinv` at the lower and upper endpoints of that interval.

`[p,plo,pup] = expcdf( __ , 'upper' )` returns the complement of the exponential cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the `'upper'` argument with any of the prior syntaxes.

The exponential cdf is

$$p = F(x | u) = \int_0^x \frac{1}{\mu} e^{-\frac{t}{\mu}} dt = 1 - e^{-\frac{x}{\mu}}$$

The result, `p`, is the probability that a single observation from an exponential distribution will fall in the interval `[0 x]`.

## Examples

### Compute the Exponential CDF

The following code shows that the median of the exponential distribution is  $\mu \log(2)$ .

```
mu = 10:10:60;  
p = expcdf(log(2)*mu,mu)
```

```
p =  
0.5000 0.5000 0.5000 0.5000 0.5000 0.5000
```

What is the probability that an exponential random variable is less than or equal to the mean,  $\mu$ ?

```
mu = 1:6;  
x = mu;  
p = expcdf(x,mu)
```

```
p =  
0.6321 0.6321 0.6321 0.6321 0.6321 0.6321
```

## See Also

[cdf](#) | [exppdf](#) | [expinv](#) | [expstat](#) | [expfit](#) | [explike](#) | [exprnd](#)

## How To

- “Exponential Distribution” on page B-32

**Purpose**

Exponential parameter estimates

**Syntax**

```
muhat = expfit(data)
[muhat,muci] = expfit(data)
[muhat,muci] = expfit(data,alpha)
[...] = expfit(data,alpha,censoring)
[...] = expfit(data,alpha,censoring,freq)
```

**Description**

`muhat = expfit(data)` estimates the mean of an exponentially distributed sample data. Each entry of `muhat` corresponds to the data in a column of data.

`[muhat,muci] = expfit(data)` returns 95% confidence intervals for the mean parameter estimates in matrix `muci`. The first row of `muci` contains the lower bounds of the confidence intervals, and the second row contains the upper bounds.

`[muhat,muci] = expfit(data,alpha)` returns  $100(1 - \alpha)\%$  confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals.

`[...] = expfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly. `data` must be a vector in order to pass in the argument `censoring`.

`[...] = expfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq` of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

**Examples**

The following estimates the mean  $\mu$  of exponentially distributed data, and returns a 95% confidence interval for the estimate:

```
mu = 3;
data = exprnd(mu,100,1); % Simulated data
```

# expfit

---

```
[muhat,muci] = expfit(data)
muhat =
    2.7511
muci =
    2.2826
    3.3813
```

## See Also

mle | explike | exppdf | expcdf | expinv | expstat | exprnd

**Superclasses** NeighborSearcher

**Purpose** Nearest neighbors search using exhaustive search

**Description** An ExhaustiveSearcher object performs *k*NN (*k*-nearest neighbor) search using exhaustive search. Search objects store information about the data used, and the distance metric and parameters. The search performance for this object, compared with the KDTreeSearcher object, tends to be better for larger dimensions (10 or more) and worse for smaller dimensions. For more information on search objects, see “What Are Search Objects?” on page 16-17.

**Construction** `NS = ExhaustiveSearcher(X, 'Name', Value)` constructs an ExhaustiveSearcher object based on X, where rows of X correspond to observations and columns correspond to variables, using one or more optional name/value pairs. You can then use this object to find neighbors in X nearest to the query points.

`NS = createns(X, 'NSMethod', 'exhaustive', 'Name', Value)` creates an ExhaustiveSearcher object based on X using createns, where rows of X correspond to observations and columns correspond to variables, using one or more optional name/value pairs. You can use this object to find neighbors in X nearest to the query points.

### Name-Value Pair Arguments

Both the ExhaustiveSearcher and the createns functions accept one or more of the following optional name/value pairs as input:

#### ‘Distance’

A string or function handle specifying the default distance metric used when you call the knnsearch method.

- 'euclidean' — Euclidean distance (default).
- 'seuclidean' — Standardized Euclidean distance. Each coordinate difference between rows in X and the query matrix is scaled by dividing by the corresponding element of the standard

# ExhaustiveSearcher

---

deviation computed from  $X$ ,  $S = \text{nanstd}(X)$ . To specify another value for  $S$ , use the `Scale` argument.

- `'cityblock'` — City block distance.
- `'chebychev'` — Chebychev distance (maximum coordinate difference).
- `'minkowski'` — Minkowski distance.
- `'mahalanobis'` — Mahalanobis distance, computed using a positive definite covariance matrix  $C$ . The default value of  $C$  is  $\text{nancov}(X)$ . To change the value of  $C$ , use the `Cov` parameter.
- `'cosine'` — One minus the cosine of the included angle between observations (treated as vectors).
- `'correlation'` — One minus the sample linear correlation between observations (treated as sequences of values).
- `'spearman'` — One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
- `'hamming'` — Hamming distance, percentage of coordinates that differ.
- `'jaccard'` — One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
- `custom distance function` — A distance function specified using `@` (for example, `@distfun`). A distance function must be of the form `function D2 = distfun(ZI, ZJ)`, taking as arguments a 1-by- $n$  vector  $ZI$  containing a single row from  $X$  or from the query points  $Y$ , an  $m2$ -by- $n$  matrix  $ZJ$  containing multiple rows of  $X$  or  $Y$ , and returning an  $m2$ -by-1 vector of distances  $D2$ , whose  $j$ th element is the distance between the observations  $ZI$  and  $ZJ(j,:)$ .

For more information on these distance metrics, see “Distance Metrics” on page 16-9.



## 'P'

A positive scalar indicating the exponent of Minkowski distance. This parameter is only valid when Distance is 'minkowski'. Default is 2.

## 'Cov'

A positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This parameter is only valid when Distance is 'mahalanobis'. Default is `nancov(X)`.

## 'Scale'

A vector **S** with the length equal to the number of columns in **X**. Each coordinate of **X** and each query point is scaled by the corresponding element of **S** when computing the standardized Euclidean distance. This parameter is only valid when Distance is 'seuclidean'. Default is `nanstd(X)`.

## Properties

### **X**

A matrix used to create the object

### **Distance**

A string specifying a built-in distance metric or a function handle that you provide when you create the object. This property is the default distance metric used when you call the `knnsearch` method to find nearest neighbors for future query points.

### **DistParameter**

Specifies the additional parameter for the chosen distance metric. The value is:

- If 'Distance' is 'minkowski': A positive scalar indicating the exponent of the Minkowski distance.
- If 'Distance' is 'mahalanobis': A positive definite matrix representing the covariance matrix used for computing the Mahalanobis distance.

# ExhaustiveSearcher

---

- If 'Distance' is 'seuclidean': A vector representing the scale value of the data when computing the 'seuclidean' distance.
- Otherwise: Empty.

## Methods

knnsearch	Find $k$ -nearest neighbors using ExhaustiveSearcher object
rangesearch	Find all neighbors within specified distance using object

## Examples

Create an ExhaustiveSearcher object using the constructor:

```
load fisheriris
x = meas(:,3:4);
NS = ExhaustiveSearcher(x,'distance','minkowski')

NS =

    ExhaustiveSearcher

    Properties:
        X: [150x2 double]
        Distance: 'minkowski'
        DistParameter: 2
```

---

Create an ExhaustiveSearcher object using createns:

```
load fisheriris
x = meas(:,3:4);
NS = createns(x,'NsMethod','exhaustive',...
             'distance','minkowski')
```

NS =

ExhaustiveSearcher

Properties:

X: [150x2 double]  
Distance: 'minkowski'  
DistParameter: 2

For more in-depth examples using the `knnsearch` method, see the method reference page or see “Classifying Query Data Using `knnsearch`” on page 16-18.

## References

[1] Friedman, J. H., Bentley, J. and Finkel, R. A. (1977). *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, ACM Transactions on Mathematical Software 3, 209.

## See Also

[createns](#) | [KDTreeSearcher](#) | [NeighborSearcher](#)

## How To

- “*k*-Nearest Neighbor Search and Radius Search” on page 16-12
- “Distance Metrics” on page 16-9

# expinv

---

**Purpose** Exponential inverse cumulative distribution function

**Syntax**  
`X = expinv(P,mu)`  
`[X,XLO,XUP] = expinv(X,mu,pcov,alpha)`

**Description** `X = expinv(P,mu)` computes the inverse of the exponential cdf with parameters specified by mean parameter `mu` for the corresponding probabilities in `P`. `P` and `mu` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive and the values in `P` must lie on the interval `[0 1]`.

`[X,XLO,XUP] = expinv(X,mu,pcov,alpha)` produces confidence bounds for `X` when the input mean parameter `mu` is an estimate. `pcov` is the variance of the estimated `mu`. `alpha` specifies `100(1 - alpha)%` confidence bounds. The default value of `alpha` is `0.05`. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds. The bounds are based on a normal approximation for the distribution of the log of the estimate of `mu`. If you estimate `mu` from a set of data, you can get a more accurate set of bounds by applying `expfit` to the data to get a confidence interval for `mu`, and then evaluating `expinv` at the lower and upper end points of that interval.

The inverse of the exponential cdf is

$$x = F^{-1}(p | \mu) = -\mu \ln(1 - p)$$

The result, `x`, is the value such that an observation from an exponential distribution with parameter `μ` will fall in the range `[0 x]` with probability `p`.

**Examples** Let the lifetime of light bulbs be exponentially distributed with `μ = 700` hours. What is the median lifetime of a bulb?

```
expinv(0.50,700)
ans =
```

485.2030

Suppose you buy a box of “700 hour” light bulbs. If 700 hours is the mean life of the bulbs, half of them will burn out in less than 500 hours.

**See Also**

`icdf` | `expcdf` | `exppdf` | `expstat` | `expfit` | `explike` | `expnrd`

**How To**

- “Exponential Distribution” on page B-32

# explike

---

**Purpose** Exponential negative log-likelihood

**Syntax**  
`nlogL = explike(param,data)`  
`[nlogL,avar] = explike(param,data)`  
`[...] = explike(param,data,censoring)`  
`[...] = explike(param,data,censoring,freq)`

**Description** `nlogL = explike(param,data)` returns the negative of the log-likelihood for the exponential distribution. `param` is the mean parameter, `mu`. `nlogL` is a scalar.

`[nlogL,avar] = explike(param,data)` returns the inverse of Fisher's information, `avar`, a scalar. If the input parameter value in `param` is the maximum likelihood estimate, `avar` is its asymptotic variance. `avar` is based on the observed Fisher's information, not the expected information.

`[...] = explike(param,data,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = explike(param,data,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

**See Also** `expcdf` | `exppdf` | `expstat` | `expfit` | `expinv` | `exprnd`

**How To** • “Exponential Distribution” on page B-32

**Purpose** Write dataset array to file

**Syntax**

```
export(DS, 'file', filename)
export(DS)
export(DS, 'file', filename, 'Delimiter', delim)
export(DS, 'XLSfile', filename)
export(DS, 'XPTfile', filename)
export(DS, ..., 'WriteVarNames', false)
export(DS, ..., 'WriteObsNames', false)
```

**Description** `export(DS, 'file', filename)` writes the dataset array `DS` to a tab-delimited text file, including variable names and observation names, if present. If the observation names exist, the name in the first column of the first line of the file is the first dimension name for the dataset (by default, 'Observations'). `export` overwrites any existing file named `filename`.

`export(DS)` writes to a text file whose default name is the name of the dataset array `DS` appended by `'.txt'`. If `export` cannot construct the file name from the dataset array input, it writes to the file `'dataset.txt'`. `export` overwrites any existing file.

`export(DS, 'file', filename, 'Delimiter', delim)` writes the dataset array `DS` to a text file using the delimiter `delim`. `delim` must be one of the following:

- ' ' or 'space'
- '\t' or 'tab'
- ',' or 'comma'
- ';' or 'semi'
- '|' or 'bar'

`export(DS, 'XLSfile', filename)` writes the dataset array `DS` to a Microsoft® Excel spreadsheet file, including variable names and observation names (if present). You can specify the `'Sheet'` and

'Range' parameter name/value pairs, with parameter values as accepted by the `xlsread` function.

`export(DS, 'XPTFile', filename)` writes the dataset array `DS` to a SAS XPORT format file. When writing to an XPORT format file, variables must be scalar valued. `export` saves observation names to a variable called `obsnames`, unless the `WriteObsNames` parameter described below is `false`. The XPORT format restricts the length of variable names to eight characters; longer variable names are truncated.

`export(DS, ..., 'WriteVarNames', false)` does not write the variable names to the text file. `export(DS, ..., 'WriteVarNames', true)` is the default, writing the names as column headings in the first line of the file.

`export(DS, ..., 'WriteObsNames', false)` does not write the observation names to the text file. `export(DS, ..., 'WriteObsNames', true)` is the default, writing the names as the first column of the file.

In some cases, `export` creates a text file that does not represent `A` exactly, as described below. If you use `dataset` to read the file back into MATLAB, the new dataset array may not have exactly the same contents as the original dataset array. Save `A` as a MAT-file if you need to import it again as a dataset array.

`export` writes out numeric variables using long `g` format, and categorical or character variables as unquoted strings.

For non-character variables with more than one column, `export` writes out multiple delimiter-separated fields on each line, and constructs suitable column headings for the first line of the file.

`export` writes out variables that have more than two dimensions as a single empty field in each line of the file.

For cell-valued variables, `export` writes out the contents of each cell only when the cell contains a single row, and writes out a single empty field otherwise.



In some cases, `export` creates a file that cannot be read back into MATLAB using `dataset`. Writing a dataset array that contains a cell-valued variable whose cell contents are not scalars results in a mismatch in the file between the number of fields on each line and the number of column headings on the first line. Writing a dataset array that contains a cell-valued variable whose cell contents are not all the same length results in a different number of fields on each line in the file. Therefore, if you might need to import a dataset array again, save it as a `.mat` file.

## Examples

Move data between external text files and dataset arrays in the MATLAB workspace:

```
A = dataset('file','sat2.dat','delimiter',' ','')
A =
    Test          Gender          Score
    'Verbal'      'Male'          470
    'Verbal'      'Female'        530
    'Quantitative' 'Male'          520
    'Quantitative' 'Female'        480

export(A(A.Score > 500,:), 'file', 'HighScores.txt')

B = dataset('file', 'HighScores.txt', 'delimiter', '\t')
B =
    Test          Gender          Score
    'Verbal'      'Female'        530
    'Quantitative' 'Male'          520
```

## See Also

`dataset`

# exp pdf

---

**Purpose** Exponential probability density function

**Syntax** `Y = exp pdf(X, mu)`

**Description** `Y = exp pdf(X, mu)` returns the pdf of the exponential distribution with mean parameter `mu`, evaluated at the values in `X`. `X` and `mu` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `mu` must be positive.

The exponential pdf is

$$y = f(x | \mu) = \frac{1}{\mu} e^{-\frac{x}{\mu}}$$

The exponential pdf is the gamma pdf with its first parameter equal to 1.

The exponential distribution is appropriate for modeling waiting times when the probability of waiting an additional period of time is independent of how long you have already waited. For example, the probability that a light bulb will burn out in its next minute of use is relatively independent of how many minutes it has already burned.

## Examples

```
y = exp pdf(5, 1:5)
y =
    0.0067    0.0410    0.0630    0.0716    0.0736
```

```
y = exp pdf(1:5, 1:5)
y =
    0.3679    0.1839    0.1226    0.0920    0.0736
```

**See Also** `pdf` | `expcdf` | `expinv` | `expstat` | `expfit` | `explike` | `expnd`

**How To** • “Exponential Distribution” on page B-32

**Purpose** Exponential random numbers

**Syntax**  
`R = exprnd(mu)`  
`R = exprnd(mu,m,n,...)`  
`R = exprnd(mu,[m,n,...])`

**Description** `R = exprnd(mu)` generates random numbers from the exponential distribution with mean parameter `mu`. `mu` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `mu`.

`R = exprnd(mu,m,n,...)` or `R = exprnd(mu,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the exponential distribution with mean parameter `mu`. `mu` can be a scalar or an array of the same size as `R`.

**Examples**

```
n1 = exprnd(5:10)
n1 =
    7.5943    18.3400    2.7113    3.0936    0.6078    9.5841

n2 = exprnd(5:10,[1 6])
n2 =
    3.2752    1.1110    23.5530    23.4303    5.7190    3.9876

n3 = exprnd(5,2,3)
n3 =
    24.3339    13.5271    1.8788
     4.7932     4.3675    2.6468
```

**See Also** `random` | `expcdf` | `exppdf` | `expstat` | `expfit` | `explike` | `expinv`

**How To**

- “Exponential Distribution” on page B-32

# expstat

---

**Purpose** Exponential mean and variance

**Syntax** `[m,v] = expstat(mu)`

**Description** `[m,v] = expstat(mu)` returns the mean of and variance for the exponential distribution with parameters `mu`. `mu` can be a vectors, matrix, or multidimensional array. The mean of the exponential distribution is  $\mu$ , and the variance is  $\mu^2$ .

**Examples**

```
[m,v] = expstat([1 10 100 1000])  
m =  
    1    10    100    1000  
v =  
    1    100   10000  1000000
```

**See Also** `expinv` | `expcdf` | `exppdf` | `expstat` | `expfit` | `explike` | `exprnd`

**How To**

- “Exponential Distribution” on page B-32

<b>Superclasses</b>	ToolboxFittableParametricDistribution
<b>Purpose</b>	Extreme value probability distribution object
<b>Description</b>	<p>prob.ExtremeValueDistribution is an object consisting of parameters, a model description, and sample data for an extreme value probability distribution.</p> <p>Create a probability distribution object with specified parameter values using <code>makedist</code>. Alternatively, fit a distribution to data using <code>fitdist</code> or the Distribution Fitting app.</p>
<b>Construction</b>	<p><code>pd = makedist('ExtremeValue')</code> creates an extreme value probability distribution object using the default parameter values.</p> <p><code>pd = makedist('ExtremeValue','mu',mu,'sigma',sigma)</code> creates an extreme value probability distribution object using the specified parameter values.</p>
	<b>Input Arguments</b>
	<b>mu - Location parameter</b> 0 (default)   scalar value
	Location parameter of the extreme value distribution, specified as a scalar value.
	<b>Data Types</b> single   double
	<b>sigma - Scale parameter</b> 1 (default)   nonnegative scalar value
	Scale parameter of the extreme value distribution, specified as a nonnegative scalar value.
	<b>Data Types</b> single   double

# prob.ExtremeValueDistribution

---

## Properties

### **mu**

Location parameter of the extreme value distribution, stored as a scalar value.

### **Data Types**

single | double

### **sigma**

Scale parameter of the extreme value distribution, stored as a nonnegative scalar value.

### **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### **Data Types**

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

### **Data Types**

single | double

### **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

# prob.ExtremeValueDistribution

---

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object



pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Extreme Value Distribution

The extreme value distribution is appropriate for modeling the smallest value from a distribution whose tails decay exponentially fast, for example, the normal distribution. It can also model the largest value from a distribution, such as the normal or exponential distributions, by using the negative of the original values.

The extreme value distribution uses the following parameters.

# prob.ExtremeValueDistribution

---

Parameter	Description	Support
mu	Location parameter	$-\infty < \mu < \infty$
sigma	Scale parameter	$\sigma \geq 0$

The probability density function (pdf) is

$$f(x | \mu, \sigma) = \sigma^{-1} \exp\left(\frac{x - \mu}{\sigma}\right) \exp\left(-\exp\left(\frac{x - \mu}{\sigma}\right)\right) ; \quad -\infty < x < \infty.$$

This form of the probability density function is suitable for modeling the minimum value. To model the maximum value, use the negative of the original values.

## Examples

### Create an Extreme Value Distribution Object Using Default Parameters

Create an extreme value distribution object using the default parameter values.

```
pd = makedist('ExtremeValue')
```

```
pd =
```

```
ExtremeValueDistribution
```

```
Extreme Value distribution
```

```
mu = 0
```

```
sigma = 1
```

### Create an Extreme Value Distribution Object Using Specified Parameters

Create an extreme value distribution object by specifying the parameter values.

```
pd = makedist('ExtremeValue', 'mu', -1, 'sigma', 2)
```

```
pd =  
  
ExtremeValueDistribution  
  
Extreme Value distribution  
    mu = -1  
    sigma = 2
```

Compute the standard deviation for the distribution.

```
s = std(pd)
```

```
s =  
  
    2.5651
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Extreme Value Distribution” on page B-35
- Class Attributes
- Property Attributes

# factoran

---

**Purpose** Factor analysis

**Syntax**

```
lambda = factoran(X,m)
[lambda,psi] = factoran(X,m)
[lambda,psi,T] = factoran(X,m)
[lambda,psi,T,stats] = factoran(X,m)
[lambda,psi,T,stats,F] = factoran(X,m)
[...] = factoran(...,param1,val1,param2,val2,...)
```

**Definitions** factoran computes the maximum likelihood estimate (MLE) of the factor loadings matrix  $\Lambda$  in the factor analysis model

$$x = \mu + \Lambda f + e$$

where  $x$  is a vector of observed variables,  $\mu$  is a constant vector of means,  $\Lambda$  is a constant  $d$ -by- $m$  matrix of factor loadings,  $f$  is a vector of independent, standardized common factors, and  $e$  is a vector of independent specific factors.  $x$ ,  $\mu$ , and  $e$  are of length  $d$ .  $f$  is of length  $m$ .

Alternatively, the factor analysis model can be specified as

$$\text{cov}(x) = \Lambda\Lambda^T + \Psi$$

where  $\Psi = \text{cov}(e)$  is a  $d$ -by- $d$  diagonal matrix of specific variances.

**Description** `lambda = factoran(X,m)` returns the maximum likelihood estimate, `lambda`, of the factor loadings matrix, in a common factor analysis model with  $m$  common factors.  $X$  is an  $n$ -by- $d$  matrix where each row is an observation of  $d$  variables. The  $(i, j)$ th element of the  $d$ -by- $m$  matrix `lambda` is the coefficient, or loading, of the  $j$ th factor for the  $i$ th variable. By default, `factoran` calls the function `rotatefactors` to rotate the estimated factor loadings using the 'varimax' option.

`[lambda,psi] = factoran(X,m)` also returns maximum likelihood estimates of the specific variances as a column vector `psi` of length  $d$ .

`[lambda,psi,T] = factoran(X,m)` also returns the  $m$ -by- $m$  factor loadings rotation matrix `T`.

[lambda,psi,T,stats] = factoran(X,m) also returns a structure stats containing information relating to the null hypothesis,  $H_0$ , that the number of common factors is m. stats includes the following fields:

Field	Description
loglike	Maximized log-likelihood value
dfe	Error degrees of freedom = $((d-m)^2 - (d+m))/2$
chisq	Approximate chi-squared statistic for the null hypothesis
p	Right-tail significance level for the null hypothesis

factoran does not compute the chisq and p fields unless dfe is positive and all the specific variance estimates in psi are positive (see “Heywood Case” on page 21-843 below). If X is a covariance matrix, then you must also specify the 'nobs' parameter if you want factoran to compute the chisq and p fields.

[lambda,psi,T,stats,F] = factoran(X,m) also returns, in F, predictions of the common factors, known as factor scores. F is an n-by-m matrix where each row is a prediction of m common factors. If X is a covariance matrix, factoran cannot compute F. factoran rotates F using the same criterion as for lambda.

[...] = factoran(...,param1,va11,param2,va12,...) enables you to specify optional parameter name/value pairs to control the model fit and the outputs. The following are the valid parameter/value pairs.

Parameter	Value	
'xtype'	Type of input in the matrix X. 'xtype' can be one of:	
	'data'	Raw data (default)
	'covariance'	Positive definite covariance or correlation matrix

# factoran

Parameter	Value	
'scores'	Method for predicting factor scores. 'scores' is ignored if X is not raw data.	
	'wls' 'Bartlett'	Synonyms for a weighted least-squares estimate that treats F as fixed (default)
	'regression' 'Thomson'	Synonyms for a minimum mean squared error prediction that is equivalent to a ridge regression
'start'	Starting point for the specific variances $\psi$ in the maximum likelihood optimization. Can be specified as:	
	'random'	Chooses $d$ uniformly distributed values on the interval [0,1].
	'Rsquared'	Chooses the starting vector as a scale factor times $\text{diag}(\text{inv}(\text{corrcoef}(X)))$ (default). For examples, see Jöreskog [2].
	Positive integer	Performs the given number of maximum likelihood fits, each initialized as with 'random'. factoran returns the fit with the highest likelihood.
	Matrix	Performs one maximum likelihood fit for each column of the specified matrix. The $i$ th optimization is initialized with the values from the $i$ th column. The matrix must have $d$ rows.

Parameter	Value	
'rotate'	Method used to rotate factor loadings and scores. 'rotate' can have the same values as the 'Method' parameter of rotatefactors. See the reference page for rotatefactors for a full description of the available methods.	
	'none'	Performs no rotation.
	'equamax'	Special case of the orthomax rotation. Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.
	'orthomax'	Orthogonal rotation that maximizes a criterion based on the variance of the loadings.  Use the 'coeff', 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.
	'parsimax'	Special case of the orthomax rotation (default). Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.
	'pattern'	Performs either an oblique rotation (the default) or an orthogonal rotation to best match a specified pattern matrix. Use the 'type' parameter to choose the type of rotation. Use the 'target' parameter to specify the pattern matrix.

# factoran

---

Parameter	Value
	<p>'procrustes'</p> <p>Performs either an oblique (the default) or an orthogonal rotation to best match a specified target matrix in the least squares sense.</p> <p>Use the 'type' parameter to choose the type of rotation. Use 'target' to specify the target matrix.</p>
	<p>'promax'</p> <p>Performs an oblique procrustes rotation to a target matrix determined by factoran as a function of an orthomax solution.</p> <p>Use the 'power' parameter to specify the exponent for creating the target matrix. Because 'promax' uses 'orthomax' internally, you can also specify the parameters that apply to 'orthomax'.</p>
	<p>'quartimax'</p> <p>Special case of the orthomax rotation (default). Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.</p>
	<p>'varimax'</p> <p>Special case of the orthomax rotation (default). Use the 'normalize', 'reltol', and 'maxit' parameters to control the details of the rotation.</p>



Parameter	Value
	<p>Function</p> <p>Function handle to rotation function of the form</p> $[B, T] = \text{myrotation}(A, \dots)$ <p>where A is a d-by-m matrix of unrotated factor loadings, B is a d-by-m matrix of rotated loadings, and T is the corresponding m-by-m rotation matrix.</p> <p>Use the <code>factoran</code> parameter <code>'userargs'</code> to pass additional arguments to this rotation function. See “Example 4” on page 21-848.</p>
<code>'coeff'</code>	Coefficient, often denoted as $\gamma$ , defining the specific <code>'orthomax'</code> criterion. Must be from 0 to 1. The value 0 corresponds to <code>quartimax</code> , and 1 corresponds to <code>varimax</code> . Default is 1.
<code>'normalize'</code>	Flag indicating whether the loading matrix should be row-normalized (1) or left unnormalized (0) for <code>'orthomax'</code> or <code>'varimax'</code> rotation. Default is 1.
<code>'reltol'</code>	Relative convergence tolerance for <code>'orthomax'</code> or <code>'varimax'</code> rotation. Default is <code>sqrt(eps)</code> .
<code>'maxit'</code>	Iteration limit for <code>'orthomax'</code> or <code>'varimax'</code> rotation. Default is 250.
<code>'target'</code>	Target factor loading matrix for <code>'procrustes'</code> rotation. Required for <code>'procrustes'</code> rotation. No default value.
<code>'type'</code>	Type of <code>'procrustes'</code> rotation. Can be <code>'oblique'</code> (default) or <code>'orthogonal'</code> .

Parameter	Value
'power'	Exponent for creating the target matrix in the 'promax' rotation. Must be $\geq 1$ . Default is 4.
'userargs'	Denotes the beginning of additional input values for a user-defined rotation function. <code>factoran</code> appends all subsequent values, in order and without processing, to the rotation function argument list, following the unrotated factor loadings matrix A. See “Example 4” on page 21-848.
'nobs'	If X is a covariance or correlation matrix, indicates the number of observations that were used in its estimation. This allows calculation of significance for the null hypothesis even when the original data are not available. There is no default. 'nobs' is ignored if X is raw data.
'delta'	Lower bound for the specific variances $\psi_i$ during the maximum likelihood optimization. Default is 0.005.
'optimopts'	Structure that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. Create this structure with the function <code>statset</code> . Enter <code>statset('factoran')</code> to see the names and default values of the parameters that <code>factoran</code> accepts in the <code>options</code> structure. See the reference page for <code>statset</code> for more information about these options.

## Tips

### Observed Data Variables

The variables in the observed data matrix X must be linearly independent, i.e.,  $\text{cov}(X)$  must have full rank, for maximum likelihood estimation to succeed. `factoran` reduces both raw data and a covariance matrix to a correlation matrix before performing the fit.

factoran standardizes the observed data  $X$  to zero mean and unit variance before estimating the loadings  $\lambda$ . This does not affect the model fit, because MLEs in this model are invariant to scale. However,  $\lambda$  and  $\psi$  are returned in terms of the standardized variables, i.e.,  $\lambda\lambda' + \text{diag}(\psi)$  is an estimate of the correlation matrix of the original data  $X$  (although not after an oblique rotation). See “Example 1” on page 21-843 and “Example 3” on page 21-845.

### Heywood Case

If elements of  $\psi$  are equal to the value of the 'delta' parameter (i.e., they are essentially zero), the fit is known as a Heywood case, and interpretation of the resulting estimates is problematic. In particular, there can be multiple local maxima of the likelihood, each with different estimates of the loadings and the specific variances. Heywood cases can indicate overfitting (i.e.,  $m$  is too large), but can also be the result of underfitting.

### Rotation of Factor Loadings and Scores

Unless you explicitly specify no rotation using the 'rotate' parameter, factoran rotates the estimated factor loadings,  $\lambda$ , and the factor scores,  $F$ . The output matrix  $T$  is used to rotate the loadings, i.e.,  $\lambda = \lambda_0 T$ , where  $\lambda_0$  is the initial (unrotated) MLE of the loadings.  $T$  is an orthogonal matrix for orthogonal rotations, and the identity matrix for no rotation. The inverse of  $T$  is known as the primary axis rotation matrix, while  $T$  itself is related to the reference axis rotation matrix. For orthogonal rotations, the two are identical.

factoran computes factor scores that have been rotated by  $\text{inv}(T')$ , i.e.,  $F = F_0 * \text{inv}(T')$ , where  $F_0$  contains the unrotated predictions. The estimated covariance of  $F$  is  $\text{inv}(T' * T)$ , which, for orthogonal or no rotation, is the identity matrix. Rotation of factor loadings and scores is an attempt to create a more easily interpretable structure in the loadings matrix after maximum likelihood estimation.

## Examples

### Example 1

Load the carbig data, and fit the default model with two factors.

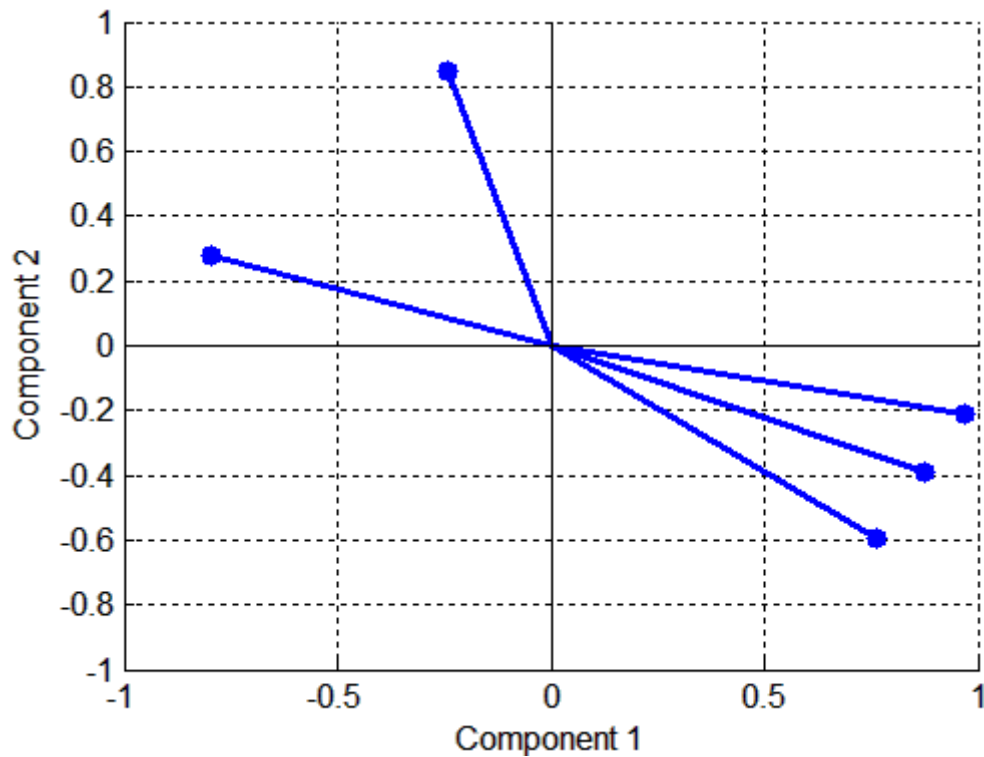
# factoran

---

```
load carbig

X = [Acceleration Displacement Horsepower MPG Weight];
X = X(all(~isnan(X),2),:);
[Lambda,Psi,T,stats,F] = factoran(X,2,...
                                'scores','regression');
inv(T'*T) % Estimated correlation matrix of F, == eye(2)
Lambda*Lambda'+diag(Psi) % Estimated correlation matrix
Lambda*inv(T)           % Unrotate the loadings
F*T'                    % Unrotate the factor scores

biplot(Lambda,...         % Create biplot of two factors
       'LineWidth',2,...
       'MarkerSize',20)
```



### Example 2

Although the estimates are the same, the use of a covariance matrix rather than raw data doesn't let you request scores or significance level:

```
[Lambda,Psi,T] = factoran(cov(X),2,'xtype','cov')
[Lambda,Psi,T] = factoran(corrcoef(X),2,'xtype','cov')
```

### Example 3

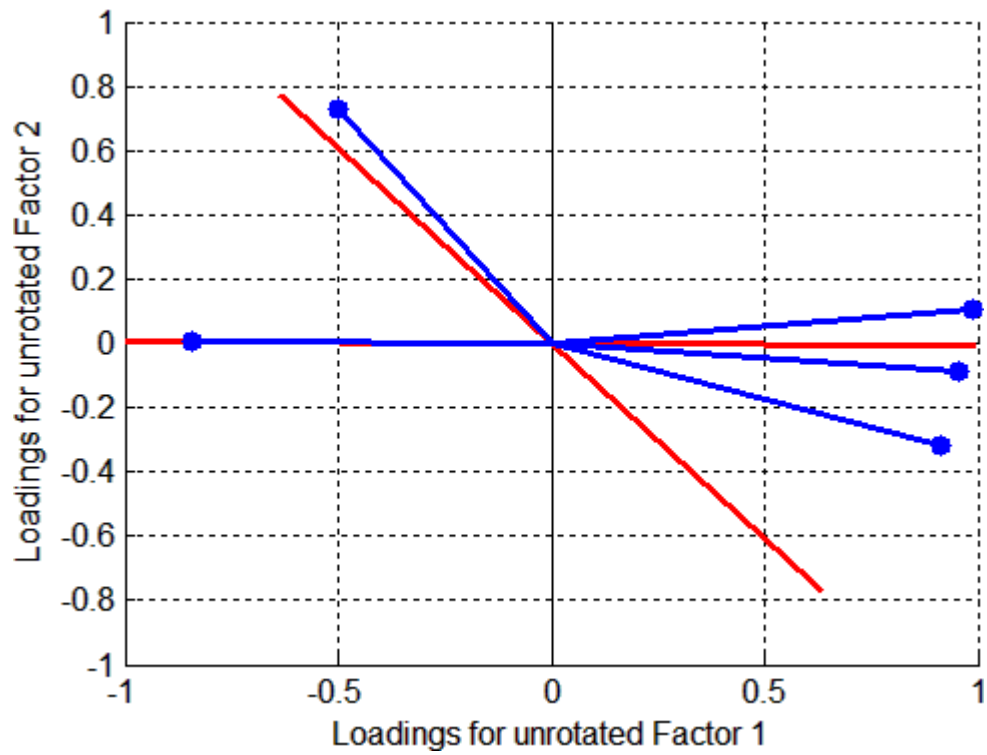
Use promax rotation:

```
[Lambda,Psi,T,stats,F] = factoran(X,2,'rotate','promax',...
    'powerpm',4);
```

```
inv(T'*T) % Est'd corr of F,  
          % no longer eye(2)  
Lambda*inv(T'*T)*Lambda'+diag(Psi) % Est'd corr of X
```

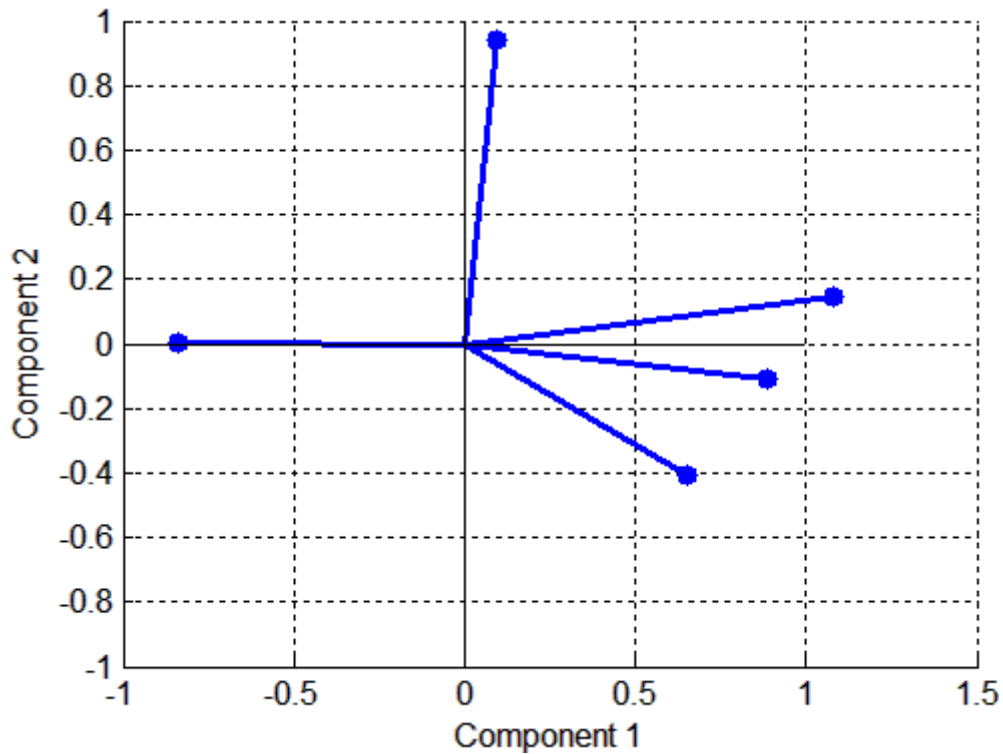
Plot the unrotated variables with oblique axes superimposed.

```
invT = inv(T)  
Lambda0 = Lambda*invT  
  
line([-invT(1,1) invT(1,1) NaN -invT(2,1) invT(2,1)], ...  
      [-invT(1,2) invT(1,2) NaN -invT(2,2) invT(2,2)], ...  
      'Color','r','linewidth',2)  
hold on  
biplot(Lambda0,...  
       'LineWidth',2,...  
       'MarkerSize',20)  
xlabel('Loadings for unrotated Factor 1')  
ylabel('Loadings for unrotated Factor 2')
```



Plot the rotated variables against the oblique axes:

```
biplot(Lambda, 'LineWidth', 2, 'MarkerSize', 20)
```



## Example 4

Syntax for passing additional arguments to a user-defined rotation function:

```
[Lambda,Psi,T] = ...  
    factoran(X,2,'rotate',@myrotation,'userargs',1,'two');
```

## References

- [1] Harman, H. H. *Modern Factor Analysis*. 3rd Ed. Chicago: University of Chicago Press, 1976.
- [2] Jöreskog, K. G. "Some Contributions to Maximum Likelihood Factor Analysis." *Psychometrika*. Vol. 32, Issue 4, 1967, pp. 443–482.



[3] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd Ed. New York: American Elsevier Publishing Co., 1971.

**See Also**

[biplot](#) | [pca](#) | [procrustes](#) | [pcacov](#) | [rotatefactors](#) | [statset](#)

# TreeBagger.FBoot property

---

**Purpose** Fraction of in-bag observations

**Description** The FBoot property is the fraction of observations to be randomly selected with replacement for each bootstrap replica. The size of each replica is given by  $n \cdot \text{FBoot}$ , where  $n$  is the number of observations in the training set. The default value is 1.

**Purpose**

$F$  cumulative distribution function

**Syntax**

```
p = fcdf(x,v1,v2)
p = fcdf(x,v1,vv2,'upper')
```

**Description**

`p = fcdf(x,v1,v2)` computes the  $F$  cdf at each of the values in  $x$  using the corresponding numerator degrees of freedom  $v_1$  and denominator degrees of freedom  $v_2$ .  $x$ ,  $v_1$ , and  $v_2$  can be vectors, matrices, or multidimensional arrays that are all the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.  $v_1$  and  $v_2$  parameters must contain real positive values.

`p = fcdf(x,v1,vv2,'upper')` returns the complement of the  $F$  cdf at each value in  $x$ , using an algorithm that more accurately computes the extreme upper tail probabilities.

The  $F$  cdf is

$$p = F(x | v_1, v_2) = \int_0^x \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{t^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)t\right]^{\frac{v_1+v_2}{2}}} dt$$

The result,  $p$ , is the probability that a single observation from an  $F$  distribution with parameters  $v_1$  and  $v_2$  will fall in the interval  $[0, x]$ .

**Examples****Compute  $F$  Distribution CDF**

The following illustrates a useful mathematical identity for the  $F$  distribution.

```
nu1 = 1:5;
nu2 = 6:10;
x = 2:6;
```

```
F1 = fcdf(x,nu1,nu2)
```

# fcdf

---

```
F1 =  
  0.7930  0.8854  0.9481  0.9788  0.9919
```

```
F2 = 1 - fcdf(1./x,nu2,nu1)
```

```
F2 =  
  0.7930  0.8854  0.9481  0.9788  0.9919
```

## See Also

[cdf](#) | [fpdf](#) | [finv](#) | [fstat](#) | [frnd](#)

## How To

- “F Distribution” on page B-41

<b>Purpose</b>	Evaluate generalized linear regression model prediction
<b>Syntax</b>	<code>ypred = feval mdl, Xnew1, Xnew2, ..., Xnewn</code>
<b>Description</b>	<code>ypred = feval(mdl, Xnew1, Xnew2, ..., Xnewn)</code> returns the predicted response of <code>mdl</code> to the input <code>[Xnew1, Xnew_2, ..., Xnewn]</code> .
<b>Tips</b>	<ul style="list-style-type: none"><li>• <code>feval</code> allows you to easily evaluate predictions of a model when the model was fitted using a dataset array. <code>predict</code> requires a dataset array with the same predictor names, but you can use simple arrays of scalars with <code>feval</code>.</li></ul>
<b>Input Arguments</b>	<p><b>mdl</b></p> <p>Generalized linear model, as constructed by <code>fitglm</code> or <code>stepwiseglm</code>.</p> <p><b>Xnew1, Xnew2, ..., Xnewn</b></p> <p>Predictor components. <code>Xnewi</code> can be one of:</p> <ul style="list-style-type: none"><li>• Scalar</li><li>• Vector</li><li>• Array</li></ul> <p>Each nonscalar component must have the same size (number of elements in each dimension).</p> <p>If you pass just one <code>Xnew</code> array, <code>Xnew</code> can be a dataset array or an array of doubles, where each column of the array represents one predictor.</p>
<b>Output Arguments</b>	<p><b>ypred</b></p> <p>Predicted mean values at <code>Xnew</code>. <code>ypred</code> is the same size as each component of <code>Xnew</code>.</p>

# GeneralizedLinearModel.feval

---

For binomial models, `feval` uses 1 as the `BinomialSize` parameter, so `ypred` is predicted probabilities.

For models with an offset, `feval` uses 0 as the offset value.

## Examples

### Predict Responses Using `feval`

Generate a generalized linear model, and plot its responses to a range of input data.

Generate artificial data for the model, Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

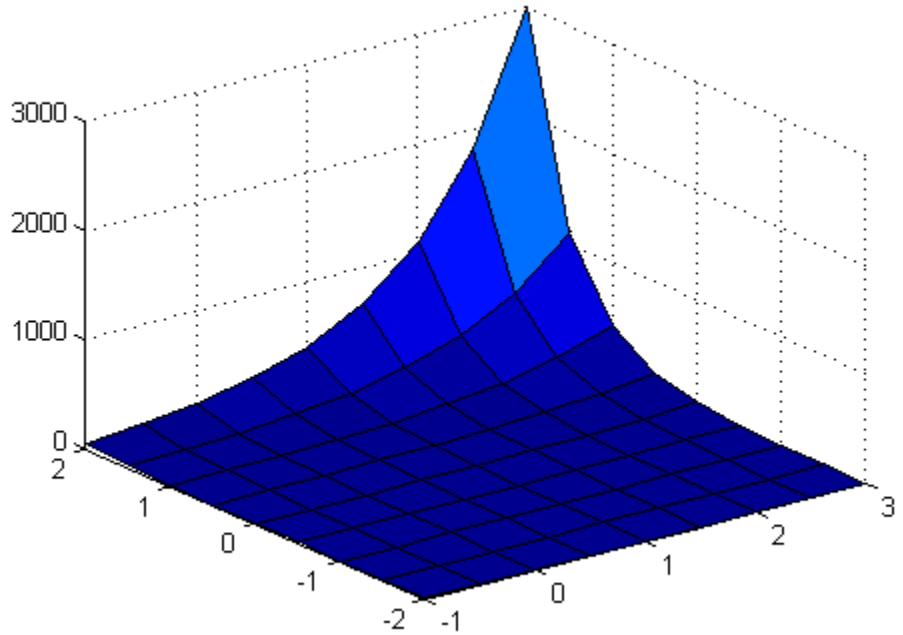
```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Generate a range of values for  $X(1)$  and  $X(2)$ , and plot the model predictions at those values.

```
[Xtest1 Xtest2] = meshgrid(-1:.5:3,-2:.5:2);
Z = feval(mdl,Xtest1,Xtest2);
surf(Xtest1,Xtest2,Z)
```



## Alternatives

`predict` gives the same predictions, but uses a single input array with one observation in each row, rather than one component in each input argument.

`random` predicts with added noise.

## See Also

`GeneralizedLinearModel` | `predict` | `random` |

## Related Examples

- “feval” on page 10-38

## Concepts

- “Generalized Linear Models” on page 10-12

# LinearModel.feval

---

**Purpose** Evaluate linear regression model prediction

**Syntax** `ypred = feval mdl, Xnew1, Xnew2, ..., Xnewn`

**Description** `ypred = feval(mdl, Xnew1, Xnew2, ..., Xnewn)` returns the predicted response of `mdl` to the input `[Xnew1, Xnew2, ..., Xnewn]`.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **Xnew1, Xnew2, ..., Xnewn**

Predictor components. `Xnewi` can be one of:

- Scalar
- Vector
- Array

Each nonscalar component must have the same size (number of elements in each dimension).

If you pass just one `Xnew` array, `Xnew` can be a dataset array or an array of doubles, where each column of the array represents one predictor.

## Output Arguments

### **ypred**

Predicted mean values at `Xnew`. `ypred` is the same size as each component of `Xnew`.

For models with an offset, `feval` uses 0 as the offset value.

## Examples

### **Plot Different Categorical Levels**

Fit a mileage model to the `smallcar` data, including the `Year` categorical predictor. Superimpose fitted curves on a scatter plot of the data.

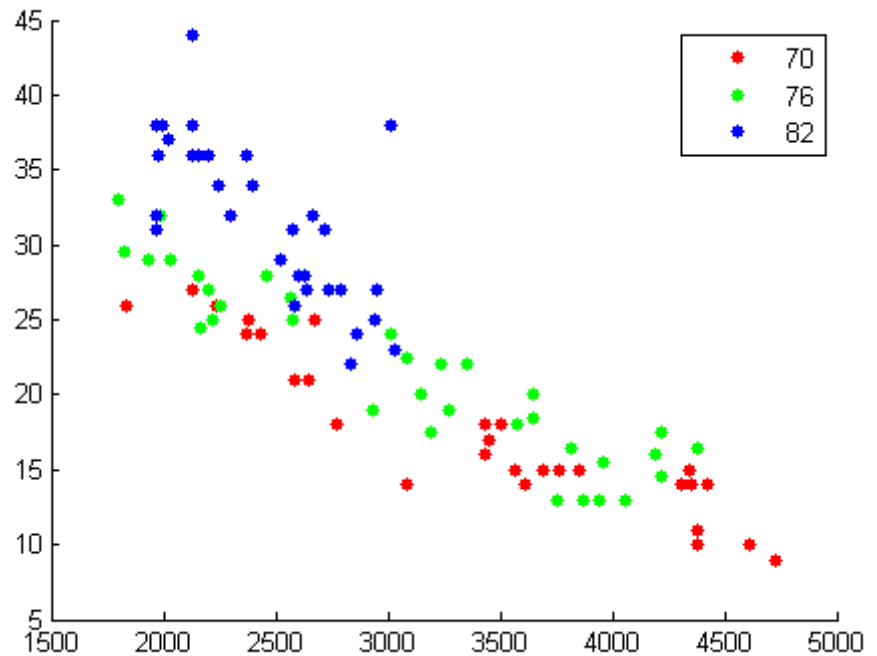
Load the data and fit a model.



```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Create a scatter plot of the mileage versus weight.

```
gscatter(ds.Weight,ds.MPG,ds.Year);
```

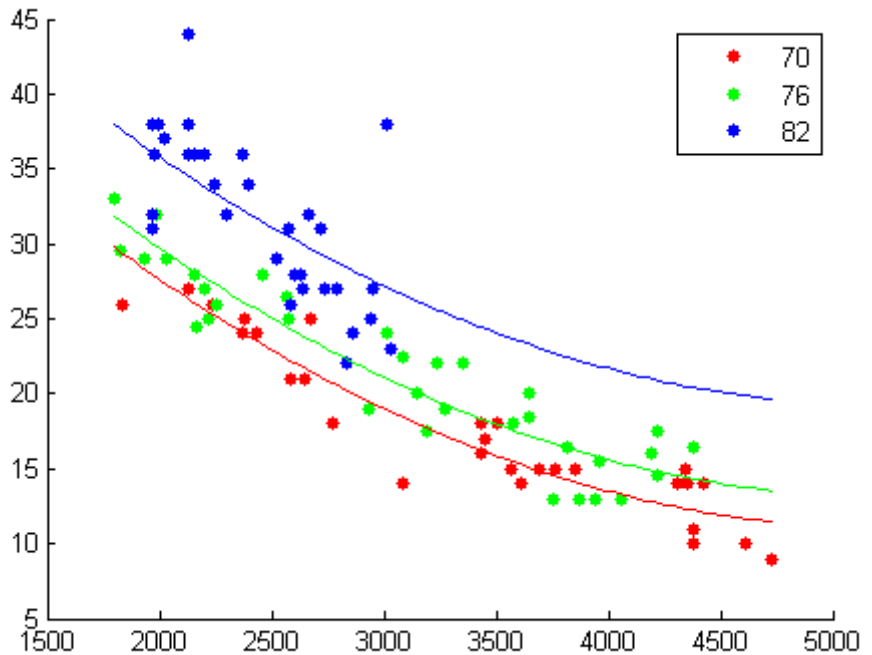


Use feval to plot curves of the model predictions for the various years and weights.

```
w = linspace(min(ds.Weight),max(ds.Weight))';
line(w,feval(mdl,w,'70'),'Color','r')
```

# LinearModel.feval

```
line(w, feval mdl, w, '76', 'Color', 'g')  
line(w, feval mdl, w, '82', 'Color', 'b')
```



## Alternatives

`predict` gives the same predictions, but uses a single input array with one observation in each row, rather than one component in each input argument. `predict` also gives confidence intervals on its predictions.

`random` predicts with added noise.

## See Also

`LinearModel` | `predict` | `random`

## Tutorials

- “feval” on page 9-41

## How To

- “Linear Regression Workflow” on page 9-44
- “Linear Regression” on page 9-12

# NonLinearModel.feval

---

**Purpose** Evaluate nonlinear regression model prediction

**Syntax** `ypred = feval mdl, Xnew1, Xnew2, ..., Xnewn`

**Description** `ypred = feval(mdl, Xnew1, Xnew2, ..., Xnewn)` returns the predicted response of `mdl` to the input `[Xnew1, Xnew2, ..., Xnewn]`.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **Xnew1, Xnew2, ..., Xnewn**

Predictor components. `Xnewi` can be one of:

- Scalar
- Vector
- Array

Each nonscalar component must have the same size (number of elements in each dimension).

If you pass just one `Xnew` array, `Xnew` can be a dataset array or an array of doubles, where each column of the array represents one predictor.

## Output Arguments

### **ypred**

Predicted mean values at `Xnew`. `ypred` is the same size as each component of `Xnew`.

## Examples

### **Predict a Nonlinear Model from a Dataset Array**

Create a nonlinear model for auto mileage based on the `carbig` data. Predict the mileage of an average automobile.

Load the data and create a nonlinear model.

```
load carbig
```

```
ds = dataset(Horsepower,Weight,MPG);
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = fitnlm(ds,modelfun,beta0);
```

Find the predicted mileage of an average auto. The data contain some observations with NaN, so compute the mean using `nanmean`.

```
Xnew = nanmean([Horsepower Weight]);
MPGnew = feval(mdl,Xnew)
```

```
MPGnew =

    21.8073
```

## Alternatives

`predict` gives the same predictions, but uses a single input array with one observation in each row, rather than one component in each input argument. `predict` also gives confidence intervals on its predictions.

`random` predicts with added noise.

## See Also

`NonLinearModel` | `predict` | `random` |

## Related Examples

- “`feval`” on page 11-15

## Concepts

- “Nonlinear Regression” on page 11-2

# ff2n

---

**Purpose** Two-level full factorial design

**Syntax** `dFF2 = ff2n(n)`

**Description** `dFF2 = ff2n(n)` gives factor settings `dFF2` for a two-level full factorial design with `n` factors. `dFF2` is  $m$ -by- $n$ , where  $m$  is the number of treatments in the full-factorial design. Each row of `dFF2` corresponds to a single treatment. Each column contains the settings for a single factor, with values of 0 and 1 for the two levels.

**Examples**

```
dFF2 = ff2n(3)
dFF2 =
  0  0  0
  0  0  1
  0  1  0
  0  1  1
  1  0  0
  1  0  1
  1  1  0
  1  1  1
```

**See Also** `fullfact`

**Purpose**

Proximity matrix for training data

**Syntax**

```
B = fillProximities(B)
B = fillProximities(B, 'param1', val1, 'param2', val2, ...)
```

**Description**

`B = fillProximities(B)` computes a proximity matrix for the training data and stores it in the `Properties` field of `B`.

`B = fillProximities(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

`'trees'`        Either `'all'` or a vector of indices of the trees in the ensemble to be used in computing the proximity matrix. Default is `'all'`.

`'nprint'`        Number of training cycles (grown trees) after which `TreeBagger` displays a diagnostic message showing training progress. Default is no diagnostic messages.

**See Also**

`CompactTreeBagger.outlierMeasure` |  
`CompactTreeBagger.proximity`

# grandstream.findobj

---

**Purpose** Find objects matching specified conditions

**Syntax** `hm = findobj(h, 'conditions')`

**Description** The `findobj` method of the `handle` class follows the same syntax as the MATLAB `findobj` command, except that the first argument must be an array of handles to objects.

`hm = findobj(h, 'conditions')` searches the handle object array `h` and returns an array of handle objects matching the specified conditions. Only the public members of the objects of `h` are considered when evaluating the conditions.

**See Also** `findobj` | `grandstream`



**Purpose** Find property of MATLAB handle object

**Syntax** `p = findprop(h,'propname')`

**Description** `p = findprop(h,'propname')` finds and returns the `meta.property` object associated with property name `propname` of scalar handle object `h`. `propname` must be a string. It can be the name of a property defined by the class of `h` or a dynamic property added to scalar object `h`.

If no property named `propname` exists for object `h`, an empty `meta.property` array is returned.

**See Also** `dynamicprops` | `findobj` | `meta.property` | `grandstream`

**Purpose**  $F$  inverse cumulative distribution function

**Syntax**  $X = \text{finv}(P, V1, V2)$

**Description**  $X = \text{finv}(P, V1, V2)$  computes the inverse of the  $F$  cdf with numerator degrees of freedom  $V1$  and denominator degrees of freedom  $V2$  for the corresponding probabilities in  $P$ .  $P$ ,  $V1$ , and  $V2$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

$V1$  and  $V2$  parameters must contain real positive values, and the values in  $P$  must lie on the interval  $[0\ 1]$ .

The  $F$  inverse function is defined in terms of the  $F$  cdf as

$$x = F^{-1}(p | v_1, v_2) = \{x : F(x | v_1, v_2) = p\}$$

where

$$p = F(x | v_1, v_2) = \int_0^x \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{t^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)t\right]^{\frac{v_1+v_2}{2}}} dt$$

## Examples

Find a value that should exceed 95% of the samples from an  $F$  distribution with 5 degrees of freedom in the numerator and 10 degrees of freedom in the denominator.

```
x = finv(0.95,5,10)
x =
    3.3258
```

You would observe values greater than 3.3258 only 5% of the time by chance.

**See Also**      `icdf | fcdf | fpdf | fstat | frnd`

**How To**      • “F Distribution” on page B-41

# ClassificationDiscriminant.fit

---

**Purpose** Fit discriminant analysis classifier

**Syntax**  
`obj = ClassificationDiscriminant.fit(X,Y)`  
`obj = ClassificationDiscriminant.fit(X,Y,Name,Value)`

**Description**  
`obj = ClassificationDiscriminant.fit(X,Y)` returns a discriminant analysis classifier based on the input variables (also known as predictors, features, or attributes) `X` and output (response) `Y`.  
`obj = ClassificationDiscriminant.fit(X,Y,Name,Value)` fits a classifier with additional options specified by one or more `Name, Value` pair arguments. If you use one of the following five options, `obj` is of class `ClassificationPartitionedModel`: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Otherwise, `obj` is of class `ClassificationDiscriminant`.

## Input Arguments

### X

Matrix of numeric predictor values. Each column of `X` represents one variable, and each row represents one observation.

`ClassificationDiscriminant.fit` considers NaN values in `X` as missing values. `ClassificationDiscriminant.fit` does not use observations with missing values for `X` in the fit.

### Y

Numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of `Y` represents the classification of the corresponding row of `X`.

`ClassificationDiscriminant.fit` considers NaN values in `Y` to be missing values. `ClassificationDiscriminant.fit` does not use observations with missing values for `Y` in the fit.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding

value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

## 'ClassNames'

Array of class names. Use the data type that exists in Y.

Use ClassNames to order the classes or to select a subset of classes for training.

**Default:** Class names that exist in Y

## 'Cost'

Square matrix, where  $Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . Alternatively, Cost can be a structure S having two fields: S.ClassNames containing the group names as a variable of the same type as Y, and S.ClassificationCosts containing the cost matrix.

**Default:**  $Cost(i, j)=1$  if  $i \neq j$ , and  $Cost(i, j)=0$  if  $i=j$

## 'crossval'

If 'on', creates a cross-validated classifier with 10 folds. You can use 'kfold', 'holdout', 'leaveout', or 'cvpartition' parameters to override this cross-validation setting. You can only use one of these options at a time for creating a cross-validated classifier: 'cvpartition', 'holdout', 'kfold', or 'leaveout'.

Alternatively, cross validate obj later using the crossval method.

**Default:** 'off'

## 'cvpartition'

Partition created with cvpartition to use in a cross-validated classifier. You can only use one of these options at a time for

# ClassificationDiscriminant.fit

---

creating a cross-validated classifier: 'cvpartition', 'holdout', 'kfold', or 'leaveout'.

## 'delta'

Threshold on linear coefficients, a nonnegative scalar. If a coefficient of `obj` has magnitude smaller than `delta`, `obj` sets this coefficient to 0, and so you can eliminate the corresponding predictor from the model. Set `delta` to a higher value to eliminate more predictors.

`delta` must be 0 for quadratic discriminant models.

**Default:** 0

## 'discrimType'

String specifying the discriminant type. Case-insensitive. One of:

- 'linear'
- 'quadratic'
- 'diagLinear'
- 'diagQuadratic'
- 'pseudoLinear'
- 'pseudoQuadratic'

**Default:** 'linear'

## 'fillCoeffs'

String, either 'on' or 'off', specifying whether to populate the `Coeffs` property in the classifier object. Setting to 'on' can be computationally intensive, especially when cross validating.

**Default:** 'on', except 'off' when cross validating

## 'gamma'

Parameter for regularizing the correlation matrix of predictors.

- Linear discriminant — Scalar from 0 to 1.
  - If you pass a value strictly between 0 and 1, `ClassificationDiscriminant.fit` sets the discriminant type to `'Linear'`.
  - If you pass 0 for gamma and `'Linear'` for `DiscrimType`, and if the correlation matrix is singular, `ClassificationDiscriminant.fit` sets gamma to the minimal value required for inverting the covariance matrix.
  - If you set gamma to 1, `ClassificationDiscriminant.fit` sets the discriminant type to `'diagLinear'`.
- Quadratic discriminant — Either 0 or 1.
  - If you pass 0 for gamma and `'Quadratic'` for `DiscrimType`, and if one of the classes has a singular covariance matrix, `ClassificationDiscriminant.fit` errors.
  - If you set gamma to 1, `ClassificationDiscriminant.fit` sets the discriminant type to `'diagQuadratic'`.

## **'holdout'**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1. You can only use one of these options at a time for creating a cross-validated classifier: `'cvpartition'`, `'holdout'`, `'kfold'`, or `'leaveout'`.

## **'kfold'**

Number of folds to use in a cross-validated classifier, a positive integer. You can only use one of these options at a time for creating a cross-validated classifier: `'cvpartition'`, `'holdout'`, `'kfold'`, or `'leaveout'`.

# ClassificationDiscriminant.fit

---

**Default:** 10

## **'leaveout'**

Use leave-one-out cross validation by setting to 'on'. You can only use one of these options at a time for creating a cross-validated classifier: 'cvpartition', 'holdout', 'kfold', or 'leaveout'.

## **'PredictorNames'**

Cell array of names for the predictor variables, in the order in which they appear in X.

**Default:** {'x1', 'x2', ...}

## **'prior'**

Prior probabilities for each class. Specify as one of:

- A string:
  - 'empirical' determines class probabilities from class frequencies in Y. If you pass observation weights, they are used to compute the class probabilities.
  - 'uniform' sets all class probabilities equal.
- A vector (one scalar value for each class)
- A structure S with two fields:
  - S.ClassNames containing the class names as a variable of the same type as Y
  - S.ClassProbs containing a vector of corresponding probabilities

**Default:** 'empirical'

## **'ResponseName'**

Name of the response variable Y, a string.



**Default:** 'Y'

## 'SaveMemory'

When 'on', `ClassificationDiscriminant.fit` does not store the full covariance matrix, but instead stores enough information to compute the matrix. The `predict` method computes the full covariance matrix for prediction, and does not store the matrix. When 'off', `ClassificationDiscriminant.fit` computes and stores the full covariance matrix in `obj`.

Set `SaveMemory` to 'on' when `X` has thousands of predictors.

**Default:** 'off'

## 'ScoreTransform'

Function handle for transforming scores, or string representing a built-in transformation function.

String	Formula
'symmetric'	$2x - 1$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set score for the class with the largest score to 1, and scores for all other classes to 0.
'symmetricismax'	Set score for the class with the largest score to 1, and scores for all other classes to -1.
'none'	$x$
'logit'	$1/(1 + e^{-x})$
'doublelogit'	$1/(1 + e^{-2x})$

# ClassificationDiscriminant.fit

---

String	Formula
'symmetriclogit'	$2/(1 + e^{-x}) - 1$
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$

You can include your own function handle for transforming scores. Your function should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

**Default:** 'none'

## 'weights'

Vector of observation weights. The length of weights is the number of rows in X. ClassificationDiscriminant.fit normalizes the weights to sum to 1.

**Default:** ones(size(X,1),1)

## Output Arguments

### obj

Discriminant analysis classifier. You can use obj to predict the response of new data using the predict method.

## Definitions

### Discriminant Classification

The model for discriminant analysis is:

- Each class (Y) generates data (X) using a multivariate normal distribution. That is, the model assumes X has a Gaussian mixture distribution (gmdistribution).
  - For linear discriminant analysis, the model has the same covariance matrix for each class, only the means vary.
  - For quadratic discriminant analysis, both means and covariances of each class vary.

predict classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

For details, see “How the predict Method Classifies” on page 15-6.

## Examples

Construct a discriminant analysis classifier for the Fisher iris data:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas, species)

obj =
ClassificationDiscriminant:
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}
  ResponseName: 'Y'
  ClassNames: {'setosa' 'versicolor' 'virginica'}
  ScoreTransform: 'none'
  NObservations: 150
  DiscrimType: 'linear'
  Mu: [3x4 double]
  Coeffs: [3x3 struct]
```

## Alternatives

The classify function also performs discriminant analysis. classify is usually more awkward to use:

# ClassificationDiscriminant.fit

---

- `classify` requires you to fit the classifier every time you make a new prediction.
- `classify` does not perform cross validation.
- `classify` requires you to fit the classifier when changing prior probabilities.

## See Also

`ClassificationDiscriminant` | `ClassificationTree.fit`

## How To

- “Discriminant Analysis” on page 15-3

**Purpose** Fit  $k$ -nearest neighbor classifier

**Syntax** `mdl = ClassificationKNN.fit(X,Y)`  
`mdl = ClassificationKNN.fit(X,Y,Name,Value)`

**Description** `mdl = ClassificationKNN.fit(X,Y)` returns a classification model based on the input variables (also known as predictors, features, or attributes)  $X$  and output (response)  $Y$ .

`mdl = ClassificationKNN.fit(X,Y,Name,Value)` fits a model with additional options specified by one or more `Name,Value` pair arguments.

If you use one of these options, `mdl` is of class `ClassificationPartitionedModel`: 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'. Otherwise, `mdl` is of class `ClassificationKNN`.

## Input Arguments

### **X**

Matrix of predictor values. Each column of  $X$  represents one variable, and each row represents one observation.

### **Y**

A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings, with the same number of rows as  $X$ . Each row of  $Y$  represents the classification of the corresponding row of  $X$ .

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'BreakTies'**

# ClassificationKNN.fit

---

String specifying the method `predict` uses to break ties if multiple classes have the same smallest cost. By default, ties occur when multiple classes have the same number of nearest points among the  $K$  nearest neighbors.

- 'nearest' — Use the class with the nearest neighbor among tied groups.
- 'random' — Use a random tiebreaker among tied groups.
- 'smallest' — Use the smallest index among tied groups.

**Default:** 'smallest'

## 'BucketSize'

Maximum number of data points in the leaf node of the *kd*-tree. This argument is meaningful only when `NSMethod` is 'kdtree'.

**Default:** 50

## 'CategoricalPredictors'

Specification of which predictors are categorical:

- 'all' — All predictors are categorical.
- [] — No predictors are categorical.

When you set `CategoricalPredictors` to 'all', the default `Distance` is 'hamming'.

**Default:** []

## 'ClassNames'

Array of class names. Use the data type that exists in  $Y$ .

Use `ClassNames` to order the classes or to select a subset of classes for training.

**Default:** Class names in `Y`

## **'Cost'**

Square matrix, where  $\text{Cost}(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . Alternatively, `Cost` can be a structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `Y`, and `S.ClassificationCosts` containing the cost matrix.

**Default:**  $\text{Cost}(i, j) = 1$  if  $i \neq j$ , and  $\text{Cost}(i, j) = 0$  if  $i = j$

## **'Cov'**

Positive definite matrix, the covariance matrix when computing the Mahalanobis distance. This argument is only valid when `'Distance'` is `'mahalanobis'`.

**Default:** `nancov(X)`

## **'crossval'**

If `'on'`, create a cross-validated model with 10 folds. Use the `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'` parameters to override this cross-validation setting. You can only use one parameter at a time to create a cross-validated model.

Alternatively, cross validate `model` later using the `crossval` method.

**Default:** `'off'`

## **'cvpartition'**

Partition created with `cvpartition` to use in a cross-validated model. You can only use one of these four options at a time for creating a cross-validated model: `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`.

## **'Distance'**

# ClassificationKNN.fit

---

String or function handle specifying the distance metric. The allowable strings depend on the `NSMethod` parameter, which you set in `ClassificationKNN.fit`, and which exists as a field in `ModelParams`.

<b>NSMethod</b>	<b>Distance Metric Names</b>
exhaustive	Any distance metric of <code>ExhaustiveSearcher</code>
kdtree	'cityblock', 'chebychev', 'euclidean', or 'minkowski'

For definitions, see “Distance Metrics” on page 16-9.

The distance metrics of `ExhaustiveSearcher`:

<b>Value</b>	<b>Description</b>
'cityblock'	City block distance.
'chebychev'	Chebychev distance (maximum coordinate difference).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.



Value	Description
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' name-value pair.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , use the Scale name-value pair.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	Distance function handle. <i>distfun</i> has the form  <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-N vector containing one row of X or Y.</li> <li>• ZJ is an M2-by-N matrix containing multiple rows of X or Y.</li> </ul>

# ClassificationKNN.fit

Value	Description
	<ul style="list-style-type: none"><li>• D2 is an M2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(J, :).</li></ul>

**Default:** 'euclidean', except the default is 'hamming' when CategoricalPredictors is 'all'

## 'DistanceWeight'

String or function handle specifying the distance weighting function.

DistanceWeight	Meaning
'equal'	No weighting
'inverse'	Weight is 1/distance
'squaredinverse'	Weight is 1/distance <sup>2</sup>
@fcn	<i>fcn</i> is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'squaredinverse' is equivalent to @(d)d.^(-2).

**Default:** 'equal'

## 'Exponent'

Positive scalar specifying the exponent of Minkowski distance. This argument is only valid when 'Distance' is 'minkowski'.

**Default:** 2

## 'holdout'

Holdout validation tests the specified fraction of the data, and uses the remaining data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated model: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## **'IncludeTies'**

Logical value indicating whether `predict` includes all the neighbors whose distance values are equal to the *K*th smallest distance. If `IncludeTies` is `true`, `predict` includes all these neighbors. Otherwise, `predict` uses exactly *K* neighbors.

**Default:** `false`

## **'kfold'**

Number of folds to use in a cross-validated model, a positive integer. You can only use one of these four options at a time for creating a cross-validated model: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

**Default:** 10

## **'leaveout'**

Use leave-one-out cross validation by setting to 'on'. You can only use one of these four options at a time for creating a cross-validated model: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## **'NSMethod'**

String specifying the nearest neighbor search method:

- 'kdtree' — Create and use a *kd*-tree to find nearest neighbors. 'kdtree' is valid when the distance metric is one of the following:
  - 'euclidean'

# ClassificationKNN.fit

---

- 'cityblock'
- 'minkowski'
- 'chebyshev'
- 'exhaustive' — Use the exhaustive search algorithm. The distance values from all points in X to each point in Y are computed to find nearest neighbors.

**Default:** 'kdtree' when X has 10 or fewer columns, X is not sparse, and the distance metric is a 'kdtree' type; otherwise, 'exhaustive'

## 'NumNeighbors'

Positive integer specifying the number of nearest neighbors in X to find for classifying each point when predicting.

**Default:** 1

## 'PredictorNames'

Cell array of names for the predictor variables, in the order in which they appear in X.

**Default:** {'x1', 'x2', ...}

## 'prior'

Prior probabilities for each class. Specify as one of:

- A string:
  - 'empirical' determines class probabilities from class frequencies in Y. If you pass observation weights, they are used to compute the class probabilities.
  - 'uniform' sets all class probabilities equal.
- A vector (one scalar value for each class)

- A structure S with two fields:
  - S.ClassNames containing the class names as a variable of the same type as Y
  - S.ClassProbs containing a vector of corresponding probabilities

If you set values for both `weights` and `prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

**Default:** 'empirical'

### **'ResponseName'**

Name of the response variable Y, a string.

**Default:** 'Response'

### **'Scale'**

Vector containing nonnegative values, with length equal to the number of columns in X. Each coordinate difference between X and a query point is scaled by the corresponding element of Scale. This argument is only valid when 'Distance' is 'seuclidean'.

**Default:** `nanstd(X)`

### **'weights'**

Vector of observation weights. The length of `weights` is the number of rows in X. `ClassificationKNN.fit` normalizes the weights in each class to add up to the value of the prior probability of the class.

**Default:** `ones(size(X,1),1)`

# ClassificationKNN.fit

---

## Output Arguments

**mdl**

*k*-nearest neighbor classifier model.

## Definitions

### Prediction

ClassificationKNN predicts the classification of a point  $X_{new}$  using a procedure equivalent to this:

- 1 Find the NumNeighbors points in the training set  $X$  that are nearest to  $X_{new}$ .
- 2 Find the NumNeighbors response values  $Y$  to those nearest points.
- 3 Assign the classification label  $Y_{new}$  that has largest posterior probability among the values in  $Y$ .

For details, see “Posterior Probability” on page 21-2608 in the predict documentation.

## Examples

### Construct a KNN Classifier

Construct a *k*-nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',5)
```

```
mdl =
```

```
ClassificationKNN:
  PredictorNames: {'x1' 'x2' 'x3' 'x4'}
```

```
ResponseName: 'Y'  
ClassNames: {'setosa' 'versicolor' 'virginica'}  
ScoreTransform: 'none'  
NObservations: 150  
Distance: 'euclidean'  
NumNeighbors: 5
```

## Use a Different Metric

To use the Minkowski metric, you must use an exhaustive searcher.

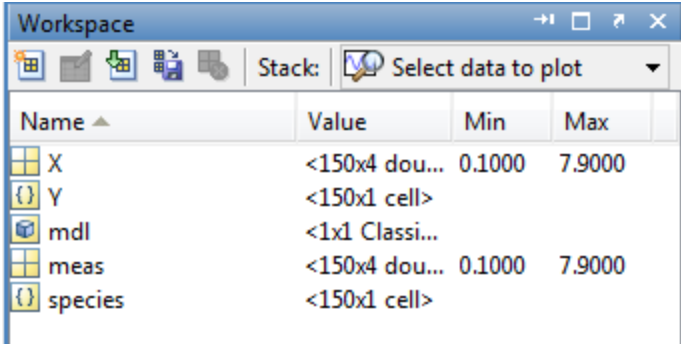
Load the data.

```
load fisheriris  
X = meas;  
Y = species;
```

Construct a classifier for 3-nearest neighbors with the Minkowski metric.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',3,...  
    'NSMethod','exhaustive','Distance','minkowski');
```

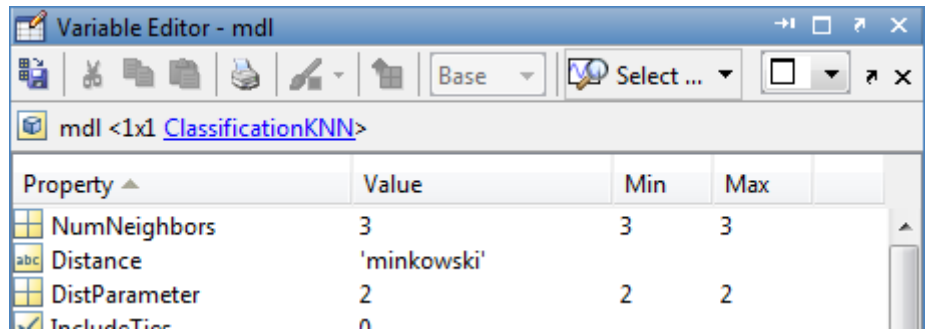
To examine mdl properties, double-click mdl in the Workspace window to open the Variable Editor.



The screenshot shows the MATLAB Workspace window with a table of variables. The table has columns for Name, Value, Min, and Max. The variables listed are X, Y, mdl, meas, and species.

Name	Value	Min	Max
X	<150x4 dou...	0.1000	7.9000
Y	<150x1 cell>		
mdl	<1x1 Classi...		
meas	<150x4 dou...	0.1000	7.9000
species	<150x1 cell>		

# ClassificationKNN.fit



## See Also

ClassificationKNN | predict |

## Related Examples

- “Construct a KNN Classifier” on page 16-27
- “Modify a KNN Classifier” on page 16-29

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9



## Purpose

Fit classification tree

## Syntax

```
tree = ClassificationTree.fit(X,Y)
tree = ClassificationTree.fit(X,Y,Name,Value)
```

## Description

`tree = ClassificationTree.fit(X,Y)` returns a classification tree based on the input variables (also known as predictors, features, or attributes) `X` and output (response) `Y`. `tree` is a binary tree, where each branching node is split based on the values of a column of `X`.

`tree = ClassificationTree.fit(X,Y,Name,Value)` fits a tree with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

Note that using the `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'` options results in a tree of class `ClassificationPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `tree` is of class `ClassificationTree`, and you can use the `predict` method to make predictions.

## Input Arguments

### X

A matrix of floating-point predictor values. Each column of `X` represents one variable, and each row represents one observation.

`ClassificationTree.fit` considers NaN values in `X` as missing values. `ClassificationTree.fit` does not use observations with all missing values for `X` in the fit. `ClassificationTree.fit` uses observations with some missing values for `X` to find splits on variables for which these observations have valid values.

### Y

A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings. Each row of `Y` represents the classification of the corresponding

# ClassificationTree.fit

---

row of  $X$ . For numeric  $Y$ , consider using `RegressionTree.fit` instead of `ClassificationTree.fit`.

`ClassificationTree.fit` considers NaN values in  $Y$  to be missing values. `ClassificationTree.fit` does not use observations with missing values for  $Y$  in the fit.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## 'AlgorithmForCategorical'

Algorithm to find the best split on a categorical predictor with  $L$  levels for data with  $K \geq 3$  classes. The available algorithms are:

'Exact'	Consider all $2^{L-1} - 1$ combinations
'PullLeft'	Pull left by purity
'PCA'	Principle component-based partition[1]
'OVAbyClass'	One versus all by class

Descriptions of the algorithms are in “Splitting Categorical Predictors” on page 16-56.

**Default:** `ClassificationTree.fit` selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For  $K = 2$  classes, `ClassificationTree.fit` always performs the exact search.

## 'CategoricalPredictors'

List of categorical predictors. Pass `CategoricalPredictors` as one of:

- A numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .
- A logical vector of length  $p$ , where a true entry means that the corresponding column of  $X$  is a categorical variable.
- 'all', meaning all predictors are categorical.
- A cell array of strings, where each element in the array is the name of a predictor variable. The names must match entries in `PredictorNames` values.
- A character matrix, where each row of the matrix is a name of a predictor variable. The names must match entries in `PredictorNames` values. Pad the names with extra blanks so each row of the character matrix has the same length.

**Default:** []

## 'ClassNames'

Array of class names. Use the data type that exists in  $Y$ .

Use `ClassNames` to order the classes or to select a subset of classes for training.

**Default:** The class names that exist in  $Y$

## 'Cost'

Square matrix, where  $Cost(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . Alternatively, `Cost` can be a structure  $S$  having two fields: `S.ClassNames` containing the group names as a variable of the same type as  $Y$ , and `S.ClassificationCosts` containing the cost matrix.

**Default:**  $Cost(i, j) = 1$  if  $i \neq j$ , and  $Cost(i, j) = 0$  if  $i = j$

# ClassificationTree.fit

---

## **'crossval'**

Flag to grow a cross-validated decision tree. Possible values are 'on' or 'off'.

If 'on', `ClassificationTree.fit` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the 'kfold', 'holdout', 'leaveout', or 'cvpartition' name-value pair arguments. Note that you can only use one of these four options ('kfold', 'holdout', 'leaveout', or 'cvpartition') at a time when creating a cross-validated tree.

Alternatively, cross validate `tree` later using the `crossval` method.

**Default:** 'off'

## **'cvpartition'**

Partition created with `cvpartition` to use in a cross-validated tree.

Note that if you use 'cvpartition', you cannot use any of the 'kfold', 'holdout', or 'leaveout' name-value pair arguments.

## **'holdout'**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1.

Note that if you use 'holdout', you cannot use any of the 'cvpartition', 'kfold', or 'leaveout' name-value pair arguments.

## **'kfold'**

Number of folds to use in a cross-validated tree, a positive integer.

Note that if you use 'kfold', you cannot use any of the 'cvpartition', 'holdout', or 'leaveout' name-value pair arguments.

**Default:** 10

## **'leaveout'**

Use leave-one-out cross validation by setting to 'on'.

Note that if you use 'leaveout', you cannot use any of the 'cvpartition', 'holdout', or 'kfold' name-value pair arguments.

## **'MaxCat'**

ClassificationTree.fit splits a categorical predictor using the exact search algorithm if the predictor has at most MaxCat levels in the split node. Otherwise, ClassificationTree.fit finds the best categorical split using one of the inexact algorithms.

Specify MaxCat as a numeric nonnegative scalar value. Passing a small value can lead to loss of accuracy and passing a large value can lead to long computation time and memory overload.

**Default:** 10

## **'MergeLeaves'**

When 'on', ClassificationTree.fit merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', ClassificationTree.fit does not merge leaves.

**Default:** 'on'

## **'MinLeaf'**

Each leaf has at least MinLeaf observations per tree leaf. If you supply both MinParent and MinLeaf,

# ClassificationTree.fit

---

ClassificationTree.fit uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** 1

## 'MinParent'

Each branch node in the tree has at least MinParent observations. If you supply both MinParent and MinLeaf, ClassificationTree.fit uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** 10

## 'NVarToSample'

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all'

## 'PredictorNames'

A cell array of names for the predictor variables, in the order in which they appear in X.

**Default:** {'x1', 'x2', ...}

## 'prior'

Prior probabilities for each class. Specify as one of:

- A string:
  - 'empirical' determines class probabilities from class frequencies in Y. If you pass observation weights, they are used to compute the class probabilities.
  - 'uniform' sets all class probabilities equal.

- A vector (one scalar value for each class)
- A structure S with two fields:
  - S.ClassNames containing the class names as a variable of the same type as Y
  - S.ClassProbs containing a vector of corresponding probabilities

If you set values for both `weights` and `prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

**Default:** 'empirical'

## **'Prune'**

When 'on', `ClassificationTree.fit` grows the classification tree, and computes the optimal sequence of pruned subtrees. When 'off' `ClassificationTree.fit` grows the classification tree without pruning.

**Default:** 'on'

## **'PruneCriterion'**

String with the pruning criterion, either 'error' or 'impurity'.

**Default:** 'error'

## **'ResponseName'**

Name of the response variable Y, a string.

**Default:** 'Response'

## **'ScoreTransform'**

# ClassificationTree.fit

Function handle for transforming scores, or string representing a built-in transformation function.

String	Formula
'symmetric'	$2x - 1$
'invlogit'	$\log(x / (1-x))$
'ismax'	Set score for the class with the largest score to 1, and scores for all other classes to 0.
'symmetricismax'	Set score for the class with the largest score to 1, and scores for all other classes to -1.
'none'	$x$
'logit'	$1/(1 + e^{-x})$
'doublelogit'	$1/(1 + e^{-2x})$
'symmetriclogit'	$2/(1 + e^{-x}) - 1$
'sign'	-1 for $x < 0$ 0 for $x = 0$ 1 for $x > 0$

You can include your own function handle for transforming scores. Your function should accept a matrix (the original scores) and return a matrix of the same size (the transformed scores).

**Default:** 'none'

## 'SplitCriterion'

Criterion for choosing a split. One of 'gdi' (Gini's diversity index), 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction (also known as cross entropy).

**Default:** 'gdi'



## 'Surrogate'

String describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', `ClassificationTree.fit` finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, `ClassificationTree.fit` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `ClassificationTree.fit` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## 'weights'

Vector of observation weights. The length of `weights` is the number of rows in `X`. `ClassificationTree.fit` normalizes the weights in each class to add up to the value of the prior probability of the class.

**Default:** `ones(size(X,1),1)`

## Output Arguments

### `tree`

A classification tree object.

Note that using the 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition' options results in a tree of class `ClassificationPartitionedModel`. You cannot use a partitioned

tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, tree is of class `ClassificationTree`, and you can use the `predict` method to make predictions.

## Definitions

### Impurity and Node Error

`ClassificationTree` splits nodes based on either *impurity* or *node error*. Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a *pure* node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('`deviance`') — With  $p(i)$  defined as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('`twoing`') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R)\left(\sum_i |L(i) - R(i)|\right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Examples

Construct a classification tree for the data in `ionosphere.mat`:

```
load ionosphere
tc = ClassificationTree.fit(X,Y)

tc =

ClassificationTree
    PredictorNames: {1x34 cell}
    ResponseName: 'Y'
    ClassNames: {'b' 'g'}
    ScoreTransform: 'none'
    CategoricalPredictors: []
    NObservations: 351
```

Properties, Methods

# ClassificationTree.fit

---

## References

[1] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. “Partitioning Nominal Attributes in Decision Trees.” *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.

## See Also

`predict` | `ClassificationTree`

<b>Purpose</b>	Create generalized linear regression model
<b>Syntax</b>	<pre>mdl = GeneralizedLinearModel.fit(ds) mdl = GeneralizedLinearModel.fit(X,y) mdl = GeneralizedLinearModel.fit(...,modelspec) mdl = GeneralizedLinearModel.fit(...,Name,Value) mdl = GeneralizedLinearModel.fit(...,modelspec,Name,Value)</pre>
<b>Description</b>	<p><code>mdl = GeneralizedLinearModel.fit(ds)</code> creates a generalized linear model of a dataset array <code>ds</code>.</p> <p><code>mdl = GeneralizedLinearModel.fit(X,y)</code> creates a generalized linear model of the responses <code>y</code> to a data matrix <code>X</code>.</p> <p><code>mdl = GeneralizedLinearModel.fit(...,modelspec)</code> creates a generalized linear model as specified by <code>modelspec</code>.</p> <p><code>mdl = GeneralizedLinearModel.fit(...,Name,Value)</code> or <code>mdl = GeneralizedLinearModel.fit(...,modelspec,Name,Value)</code> creates a generalized linear model with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• Use the function <code>fitglm</code> instead of <code>GeneralizedLinearModel.fit</code>.</li><li>• The generalized linear model <code>mdl</code> is a standard linear model unless you specify otherwise with the <code>Distribution</code> name-value pair.</li><li>• For other methods such as <code>devianceTest</code>, or properties of the <code>GeneralizedLinearModel</code> object, see <code>GeneralizedLinearModel</code>.</li></ul>
<b>Input Arguments</b>	<p><b>ds - Input data</b> dataset array</p> <p>Input data, specified as a dataset array. When <code>modelspec</code> is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.</p>

# GeneralizedLinearModel.fit

---

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

## Data Types

single | double | logical

## X - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of X represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in X.

## Data Types

single | double | logical

## y - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in y is the response for the corresponding row of X.

## Data Types

single | double

## modelspec - Model specification

string specifying the model |  $t$ -by- $(p+1)$  terms matrix | string of the form 'Y ~ terms'

Model specification, which is the starting model for `stepwiseglm`, specified as one of the following:

- String specifying the type of model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p+1)$  matrix, namely terms matrix, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- String representing a formula in the form

`'Y ~ terms'`,

where the terms are in “Wilkinson Notation” on page 21-915.

**Example:** `'quadratic'`

# GeneralizedLinearModel.fit

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## 'BinomialSize' - Number of trials for binomial distribution

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the response. This is the parameter  $n$  for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

## Data Types

single | double

## 'CategoricalVars' - Categorical variables

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.



- If data is in matrix X, then the default value of this name-value pair argument is an empty matrix []. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** 'CategoricalVars',[2,3]

**Example:** 'CategoricalVars',logical([0 1 1 0 0 0])

## Data Types

single | double | logical

### 'DispersionFlag' - Indicator to compute dispersion parameter

false for 'binomial' and 'poisson' distributions (default) | true

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

true	Estimate a dispersion parameter when computing standard errors
false	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

**Example:** 'DispersionFlag',true

### 'Distribution' - Distribution of the response variable

'normal' (default) | 'binomial' | 'poisson' | 'gamma' | 'inverse gaussian'

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

# GeneralizedLinearModel.fit

---

'normal'	Normal distribution
'binomial'	Binomial distribution
'poisson'	Poisson distribution
'gamma'	Gamma distribution
'inverse gaussian'	Inverse Gaussian distribution

**Example:** 'Distribution','gamma'

## **'Exclude' - Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

## **Data Types**

single | double | logical

## **'Intercept' - Indicator for constant term**

true (default) | false

Indicator the for constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

**Example:** 'Intercept',false

## **'Link' - Link function**

The canonical link function (default) | scalar value | structure

Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
'identity'	$f(\mu) = \mu$	$\mu = Xb$
'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'probit'	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$
'complloglog'	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
'reciprocal'	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
p (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$
S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size: <ul style="list-style-type: none"> <li>• S.Link — The link function</li> <li>• S.Inverse — The inverse link function</li> <li>• S.Derivative — The derivative of the link function</li> </ul>	$f(\mu) = S.Link(\mu)$	$\mu = S.Inverse(Xb)$

The link function defines the relationship  $f(\mu) = X*b$  between the mean response  $\mu$  and the linear combination of predictors  $X*b$ .

# GeneralizedLinearModel.fit

---

For more information on the canonical link functions, see [Definitions](#).

**Example:** 'Link', 'probit'

## 'Offset' - Offset variable

[] (default) | vector | string

Offset variable in the fit, specified as the comma-separated pair consisting of 'Offset' and a vector or name of a variable with the same length as the response.

`fitglm` and `stepwiseglm` use `Offset` as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the `Offset` predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor `A`. By using the log link function and by specifying `log(A)` as an offset, you can force the model to satisfy this theoretical constraint.

## Data Types

single | double | char

## 'PredictorVars' - Predictor variables

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array `ds`, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the `ds` dataset array, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in X, or all variables in ds except for ResponseVar.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** 'PredictorVars',[2,3]

**Example:** 'PredictorVars',logical([0 1 1 0 0 0])

## Data Types

single | double | logical | cell

### 'ResponseVar' - Response variable

last column in dataset array ds (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a string of the variable name in the dataset array ds, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a dataset array ds.

For example, you can specify the fourth variable, say yield, as the response out of six variables, in one of the following ways.

**Example:** 'ResponseVar','yield'

**Example:** 'ResponseVar',[4]

**Example:** 'ResponseVar',logical([0 0 0 1 0 0])

## Data Types

single | double | logical | char

### 'VarNames' - Names of variables in fit

{ 'x1', 'x2', ..., 'xn', 'y' } (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of strings including the names for the columns of X first, and the name for the response variable y last.

# GeneralizedLinearModel.fit

---

'VarNames' is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

## Example:

```
'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}
```

## Data Types

cell

## 'Weights' - Observation weights

ones(n,1) (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

## Data Types

single | double

## Output Arguments

### mdl - Generalized linear model

GeneralizedLinearModel object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a GeneralizedLinearModel object.

For properties and methods of the generalized linear model object, mdl, see the GeneralizedLinearModel class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

# GeneralizedLinearModel.fit

---

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
    0    0    0
    1    0    0
    0    1    0
    1    1    0
    0    2    0
```



Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

$$y \sim 1 + x1*x2 + x2^2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.000196
x1	0.54418	0.57125	0.95261	0.343
x2	-0.012781	0.0060312	-2.1192	0.0368
x1:x2	-0.00010892	0.00017925	-0.6076	0.5
x2^2	9.7518e-07	7.5389e-07	1.2935	0.199

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

```
mdl =
```

Linear regression model:

$$y \sim 1 + x2$$

# GeneralizedLinearModel.fit

---

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a string of the form

`'Y ~ terms'`,

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include - 1 in the formula.

For example,

'Y ~ A + B + C' means a three-variable linear model with intercept.

'Y ~ A + B + C - 1' is a three-variable linear model without intercept.

'Y ~ A + B + C + B^2' is a three-variable model with intercept and a B^2 term.

'Y ~ A + B^2 + C' is the same as the previous example because B^2 includes a B term.

'Y ~ A + B + C + A:B' includes an A\*B term.

'Y ~ A\*B + C' is the same as the previous example because  $A*B = A + B + A:B$ .

'Y ~ A\*B\*C - A:B:C' has all interactions among A, B, and C, except the three-way interaction.

'Y ~ A\*(B + C + D)' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
A^k, where k is a positive integer	A, A <sup>2</sup> , ..., A <sup>k</sup>
A + B	A, B
A*B	A, B, A*B
A:B	A*B only

# GeneralizedLinearModel.fit

Wilkinson Notation	Factors in Standard Notation
-B	Do not include B
A*B + C	A, B, C, A*B
A + B + C + A:B	A, B, C, A*B
A*B*C - A:B:C	A, B, C, A*B, A*C, B*C
A*(B + C)	A, B, C, A*B, A*C

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using -1.

## Canonical Link Function

The default link function for a generalized linear model is the *canonical link function*.

## Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

## Examples

### Fit a Generalized Linear Model

Make a logistic binomial model of the probability of smoking as a function of age, weight, and sex, using a two-way interactions model.

Load the hospital dataset array.

```
load hospital
```

```
ds = hospital; % just to use the ds name
```

Specify the model using a formula that allows up to two-way interactions.

```
modelspec = 'Smoker ~ Age*Weight*Sex - Age:Weight:Sex';
```

Create the generalized linear model.

```
mdl = fitglm(ds,modelspec,'Distribution','binomial')
```

```
mdl =
```

Generalized Linear regression model:

```
logit(Smoker) ~ 1 + Sex*Age + Sex*Weight + Age*Weight  
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-6.0492	19.749	-0.3063	0.75938
Sex_Male	-2.2859	12.424	-0.18399	0.85402
Age	0.11691	0.50977	0.22934	0.81861
Weight	0.031109	0.15208	0.20455	0.83792
Sex_Male:Age	0.020734	0.20681	0.10025	0.92014
Sex_Male:Weight	0.01216	0.053168	0.22871	0.8191
Age:Weight	-0.00071959	0.0038964	-0.18468	0.85348

100 observations, 93 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 5.07, p-value = 0.535

The large  $p$ -value indicates the model might not differ statistically from a constant.

## Alternatives

You can also construct a generalized linear model using `fitglm`.

# GeneralizedLinearModel.fit

---

Use `stepwiseglm` to select a model specification automatically. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

## References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## See Also

`GeneralizedLinearModel` | `GeneralizedLinearModel.stepwise` |

## Related Examples

- “Generalized Linear Model Workflow” on page 10-42

## Concepts

- “Generalized Linear Models” on page 10-12

**Purpose** Gaussian mixture parameter estimates

**Syntax**

```
obj = gmdistribution.fit(X,k)
obj = gmdistribution.fit(...,param1,val1,param2,val2,...)
```

**Description** `obj = gmdistribution.fit(X,k)` uses an Expectation Maximization (EM) algorithm to construct an object `obj` of the `gmdistribution` class containing maximum likelihood estimates of the parameters in a Gaussian mixture model with `k` components for data in the  $n$ -by- $d$  matrix `X`, where  $n$  is the number of observations and  $d$  is the dimension of the data.

`gmdistribution` treats NaN values as missing data. Rows of `X` with NaN values are excluded from the fit.

`obj = gmdistribution.fit(...,param1,val1,param2,val2,...)` provides control over the iterative EM algorithm. Parameters and values are listed below.

Parameter	Value
'Start'	<p>Method used to choose initial component parameters. One of the following:</p> <ul style="list-style-type: none"> <li>'randSample' — To select <code>k</code> observations from <code>X</code> at random as initial component means. The mixing proportions are uniform. The initial covariance matrices for all components are diagonal, where the element <code>j</code> on the diagonal is the variance of <code>X(:,j)</code>. This is the default.</li> <li><code>S</code> — A structure array with fields <code>mu</code>, <code>Sigma</code>, and <code>PComponents</code>. See <code>gmdistribution</code> for descriptions of values.</li> <li><code>s</code> — A vector of length <math>n</math> containing an initial guess of the component index for each point.</li> </ul>
'Replicates'	A positive integer giving the number of times to repeat the EM algorithm, each time with a new set of

# gmdistribution.fit

---

Parameter	Value
	parameters. The solution with the largest likelihood is returned. A value larger than 1 requires the 'randSample' start method. The default is 1.
'CovType'	'diagonal' if the covariance matrices are restricted to be diagonal; 'full' otherwise. The default is 'full'.
'SharedCov'	Logical true if all the covariance matrices are restricted to be the same (pooled estimate); logical false otherwise.
'Regularize'	A nonnegative regularization number added to the diagonal of covariance matrices to make them positive-definite. The default is 0.
'Options'	Options structure for the iterative EM algorithm, as created by <code>statset</code> . <code>gmdistribution.fit</code> uses the parameters 'Display' with a default value of 'off', 'MaxIter' with a default value of 100, and 'TolFun' with a default value of $1e-6$ .

In some cases, `gmdistribution` may converge to a solution where one or more of the components has an ill-conditioned or singular covariance matrix.

The following issues may result in an ill-conditioned covariance matrix:

- The number of dimension of your data is relatively high and there are not enough observations.
- Some of the features (variables) of your data are highly correlated.
- Some or all the features are discrete.
- You tried to fit the data to too many components.

In general, you can avoid getting ill-conditioned covariance matrices by using one of the following precautions:



- Pre-process your data to remove correlated features.
- Set 'SharedCov' to true to use an equal covariance matrix for every component.
- Set 'CovType' to 'diagonal'.
- Use 'Regularize' to add a very small positive number to the diagonal of every covariance matrix.
- Try another set of initial values.

In other cases `gmdistribution` may pass through an intermediate step where one or more of the components has an ill-conditioned covariance matrix. Trying another set of initial values may avoid this issue without altering your data or model.

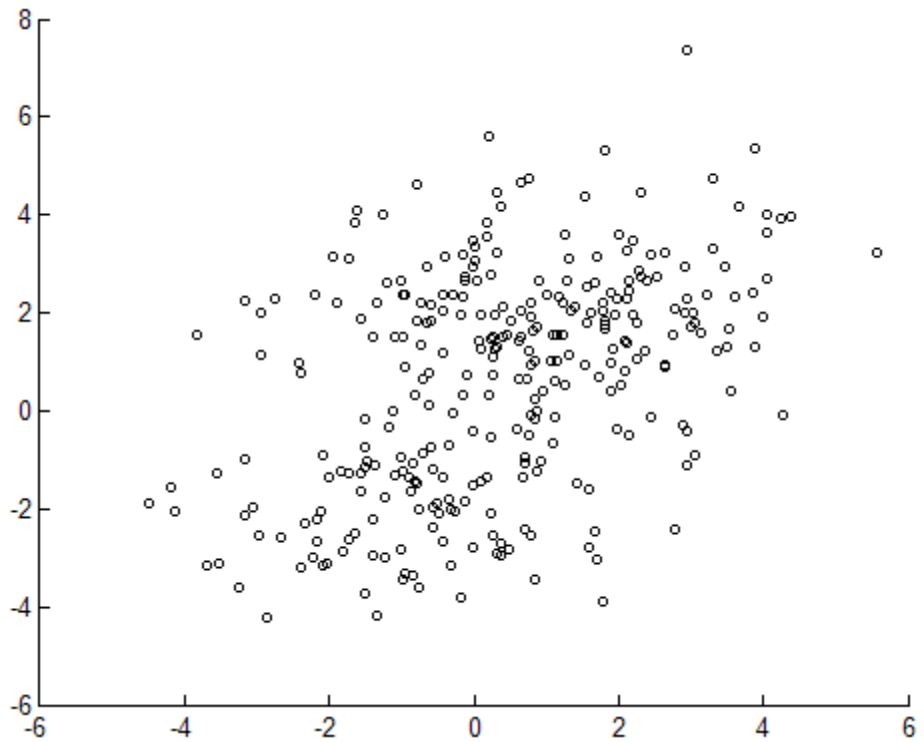
## Examples

Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function:

```
MU1 = [1 2];  
SIGMA1 = [2 0; 0 .5];  
MU2 = [-3 -5];  
SIGMA2 = [1 0; 0 1];  
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];  
  
scatter(X(:,1),X(:,2),10,'.')  
hold on
```

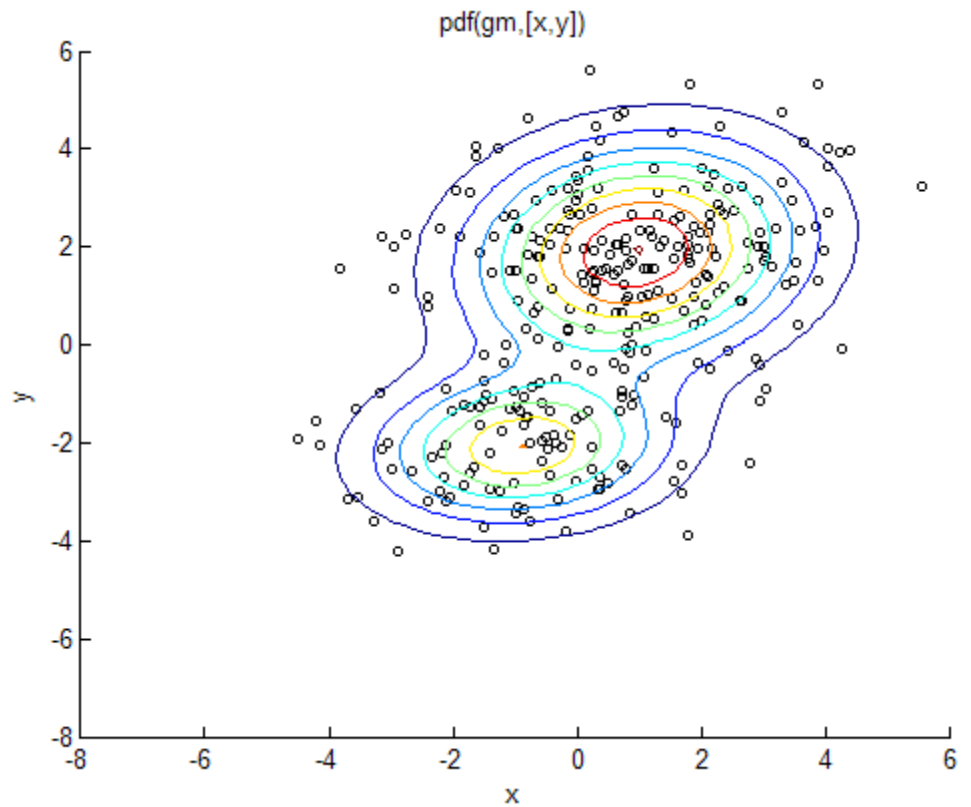
# gmdistribution.fit

---



Next, fit a two-component Gaussian mixture model:

```
options = statset('Display','final');  
obj = gmdistribution.fit(X,2,'Options',options);  
10 iterations, log-likelihood = -7046.78  
  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```



Among the properties of the fit are the parameter estimates:

```
ComponentMeans = obj.mu
```

```
ComponentMeans =
```

```
    0.9391    2.0322
   -2.9823   -4.9737
```

```
ComponentCovariances = obj.Sigma
```

```
ComponentCovariances(:, :, 1) =
```

```
    1.7786   -0.0528
   -0.0528    0.5312
```

# gmdistribution.fit

---

```
ComponentCovariances(:,:,2) =  
    1.0491   -0.0150  
   -0.0150    0.9816  
  
MixtureProportions = obj.PComponents  
MixtureProportions =  
    0.5000    0.5000
```

The Akaike information is minimized by the two-component model:

```
AIC = zeros(1,4);  
obj = cell(1,4);  
for k = 1:4  
    obj{k} = gmdistribution.fit(X,k);  
    AIC(k) = obj{k}.AIC;  
end  
  
[minAIC,numComponents] = min(AIC);  
numComponents  
numComponents =  
    2  
  
model = obj{2}  
model =  
Gaussian mixture distribution  
with 2 components in 2 dimensions  
Component 1:  
Mixing proportion: 0.500000  
Mean:    0.9391    2.0322  
Component 2:  
Mixing proportion: 0.500000  
Mean:   -2.9823   -4.9737
```

Both the Akaike and Bayes information are negative log-likelihoods for the data with penalty terms for the number of estimated parameters. They are often used to determine an appropriate number of components for a model when the number of components is unspecified.

## References

[1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

## See Also

`gmdistribution` | `cluster`

# LinearModel.fit

---

**Purpose** Create linear regression model

**Syntax**

```
mdl = LinearModel.fit(ds)
mdl = LinearModel.fit(X,y)
mdl = LinearModel.fit(...,modelspec)
mdl = LinearModel.fit(...,Name,Value)
mdl = LinearModel.fit(...,modelspec,Name,Value)
```

**Description**

`mdl = LinearModel.fit(ds)` creates a linear model of a dataset array `ds`.

`mdl = LinearModel.fit(X,y)` creates a linear model of the responses `y` to a data matrix `X`.

`mdl = LinearModel.fit(...,modelspec)` creates a linear model of the specified type.

`mdl = LinearModel.fit(...,Name,Value)` or `mdl = LinearModel.fit(...,modelspec,Name,Value)` creates a linear model with additional options specified by one or more `Name,Value` pair arguments.

- Tips**
- Use the function `fitlm` instead of `LinearModel.fit`.
  - Use robust fitting (`RobustOpts` name-value pair) to reduce the effect of outliers automatically.
  - Do not use robust fitting when you want to subsequently adjust a model using `step`.
  - For other methods or properties of the `LinearModel` object, see `LinearModel`.

**Input Arguments**

**ds - Input data**  
dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and

response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### Data Types

single | double | logical

### X - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

### Data Types

single | double | logical

### y - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### Data Types

single | double

### modelspec - Model specification

string naming the model |  $t$ -by- $(p + 1)$  terms matrix | string of the form 'Y ~ terms'

# LinearModel.fit

---

Model specification, specified as one of the following. The choice is the starting model for `stepwiselm`.

- A string naming the model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p + 1)$  matrix, namely terms matrix, specifying terms to include in the model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus 1 is for the response variable.
- A string representing a formula in the form

`'Y ~ terms'`,



where the terms are in “Wilkinson Notation” on page 21-939.

**Example:** 'quadratic'

**Example:** 'y ~ X1 + X2^2 + X1:X2'

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'CategoricalVars' - Categorical variables**

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of 'CategoricalVars' and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.
- If data is in matrix `X`, then the default value of this name-value pair argument is an empty matrix `[]`. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** 'CategoricalVars',[2,3]

**Example:** 'CategoricalVars',logical([0 1 1 0 0 0])

### **Data Types**

single | double | logical

## 'Exclude' - Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

## Data Types

single | double | logical

## 'Intercept' - Indicator for constant term

true (default) | false

Indicator the for constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

**Example:** 'Intercept',false

## 'PredictorVars' - Predictor variables

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array ds, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the ds dataset array, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in X, or all variables in ds except for ResponseVar.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** 'PredictorVars',[2,3]

**Example:** 'PredictorVars',logical([0 1 1 0 0 0])

### Data Types

single | double | logical | cell

### 'ResponseVar' - Response variable

last column in dataset array ds (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a string of the variable name in the dataset array ds, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a dataset array ds.

For example, you can specify the fourth variable, say yield, as the response out of six variables, in one of the following ways.

**Example:** 'ResponseVar','yield'

**Example:** 'ResponseVar',[4]

**Example:** 'ResponseVar',logical([0 0 0 1 0 0])

### Data Types

single | double | logical | char

### 'RobustOpts' - Indicator of robust fitting type

'off' (default) | 'on' | string | structure with string or function handle

Indicator of the robust fitting type to use, specified as the comma-separated pair consisting of 'RobustOpts' and one of the following.

# LinearModel.fit

---

- 'off' — No robust fitting. `fitlm` uses ordinary least squares.
- 'on' — Robust fitting. When you use robust fitting, 'bisquare' weight function is the default.
- String — Name of the robust fitting weight function from the following table. `fitlm` uses the corresponding default tuning constant in the table.
- Structure with the string `RobustWgtFun` containing the name of the robust fitting weight function from the following table and optional scalar `Tune` fields — `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. You can choose the name of the robust fitting weight function from this table. If you do not supply a `Tune` field, the fitting function uses the corresponding default tuning constant.

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) \cdot \sin(r) \cdot / r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) \cdot (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 \cdot / (1 + r.^2)$	2.385
'fair'	$w = 1 \cdot / (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 \cdot / \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) \cdot / r$	1.205
'ols'	Ordinary least squares (no weighting function)	None
'talwar'	$w = 1 \cdot * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985

The value  $r$  in the weight functions is

$$r = \text{resid} / (\text{tune} * s * \sqrt{1-h}),$$

where `resid` is the vector of residuals from the previous iteration, `h` is the vector of leverage values from a least-squares fit, and `s` is an estimate of the standard deviation of the error term given by

$$s = \text{MAD} / 0.6745.$$

MAD is the median absolute deviation of the residuals from their median. The constant 0.6745 makes the estimate unbiased for the normal distribution. If there are  $p$  columns in  $X$ , the smallest  $p$  absolute deviations are excluded when computing the median.

Default tuning constants give coefficient estimates that are approximately 95% as statistically efficient as the ordinary least-squares estimates, provided the response has a normal distribution with no outliers. Decreasing the tuning constant increases the downweight assigned to large residuals; increasing the tuning constant decreases the downweight assigned to large residuals.

- Structure with the function handle `RobustWgtFun` and optional scalar `Tune` fields — You can specify a custom weight function. `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. Specify `RobustWgtFun` as a function handle that accepts a vector of residuals, and returns a vector of weights the same size. The fitting function scales the residuals, dividing by the tuning constant (default 1) and by an estimate of the error standard deviation before it calls the weight function.

**Example:** `'RobustOpts', 'andrews'`

### 'VarNames' - Names of variables in fit

{ 'x1', 'x2', ..., 'xn', 'y' } (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings including the

# LinearModel.fit

---

names for the columns of X first, and the name for the response variable y last.

'VarNames' is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

## Example:

```
'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}
```

## Data Types

cell

## 'Weights' - Observation weights

ones(n,1) (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

## Data Types

single | double

## Output Arguments

### mdl - Linear model

LinearModel object

Linear model representing a least-squares fit of the response to the data, returned as a LinearModel object.

If the value of the 'RobustOpts' name-value pair is not [] or 'ols', the model is not a least-squares fit, but uses the robust fitting function.

For properties and methods of the linear model object, mdl, see the LinearModel class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

T =

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

# LinearModel.fit

---

```
0 0 0 1
```

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...  
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall  
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```



```
0 0 0
1 0 0
0 1 0
1 1 0
0 2 0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

```
y ~ 1 + x1*x2 + x2^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.000196
x1	0.54418	0.57125	0.95261	0.343
x2	-0.012781	0.0060312	-2.1192	0.0368
x1:x2	-0.00010892	0.00017925	-0.6076	0.5
x2^2	9.7518e-07	7.5389e-07	1.2935	0.199

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

# LinearModel.fit

---

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

mdl =

Linear regression model:

$$y \sim 1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a string of the form

`'Y ~ terms'`,

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**

- $\wedge$  raises the predictor to a power, exactly as in  $*$  repeated, so  $\wedge$  includes lower order terms as well
- $()$  groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include  $-1$  in the formula.

---

For example,

' $Y \sim A + B + C$ ' means a three-variable linear model with intercept.

' $Y \sim A + B + C - 1$ ' is a three-variable linear model without intercept.

' $Y \sim A + B + C + B^2$ ' is a three-variable model with intercept and a  $B^2$  term.

' $Y \sim A + B^2 + C$ ' is the same as the previous example because  $B^2$  includes a  $B$  term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of  $A$  with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$

# LinearModel.fit

---

Wilkinson Notation	Factors in Standard Notation
A + B	A, B
A*B	A, B, A*B
A:B	A*B only
-B	Do not include B
A*B + C	A, B, C, A*B
A + B + C + A:B	A, B, C, A*B
A*B*C - A:B:C	A, B, C, A*B, A*C, B*C
A*(B + C)	A, B, C, A*B, A*C

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using -1.

## Examples

### Linear Regression Model of Matrix Data

Fit a linear model of the Hald data.

Load the data.

```
load hald
X = ingredients; % predictor variables
y = heat; % response
```

Fit a default linear model to the data.

```
mdl = LinearModel.fit(X,y)
```

```
mdl =
```

```
Linear regression model:
y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	62.405	70.071	0.8906	0.39913

x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 2.45  
R-squared: 0.982, Adjusted R-Squared 0.974  
F-statistic vs. constant model: 111, p-value = 4.76e-07

## Linear Regression with Categorical Predictor and Nonlinear Model

Fit a model of a dataset array that contains a categorical predictor.  
Use a nonlinear response formula.

Load the carsmall data.

```
load carsmall
```

Construct a dataset containing continuous predictor variable `Weight`,  
nominal predictor variable `Year`, and response variable `MPG`.

```
ds = dataset(MPG,Weight);  
ds.Year = nominal(Model_Year);
```

Create a fitted model of `MPG` as a function of `Year`, `Weight`, and `Weight2`.  
(You don't have to include `Weight` explicitly in your formula because it  
is a lower-order term of `Weight2`. For details, see "Formula" on page  
21-938.)

```
mdl = fitlm(ds,'MPG ~ Year + Weight^2')
```

```
mdl =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Year + Weight^2
```

```
Estimated Coefficients:
```

# LinearModel.fit

---

	Estimate	SE	tStat	pValue
(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 2.78

R-squared: 0.885, Adjusted R-Squared 0.88

F-statistic vs. constant model: 172, p-value = 5.52e-41

`fitlm` creates two dummy (indicator) variables for the nominal variate, `Year`. The dummy variable `Year_76` takes the value 1 if model year is 1976 and takes the value 0 if it is not. The dummy variable `Year_82` takes the value 1 if model year is 1982 and takes the value 0 if it is not. And the year 1970 is the reference year (for more information on dummy variables see “Dummy Indicator Variables” on page 2-55). The corresponding model is

$$\hat{MPG} = 54.206 - 0.0164Weight + 2.0887(Year\_76) + 8.1864(Year\_82) + 1.557e-06Weight^2$$

## Simultaneously Specify the Variables and Use Formula

Simultaneously identify response and predictor variables and specify the model using formula in linear regression.

Load sample data.

```
load hospital
```

Fit a linear model with interaction terms to the data.

```
mdl = LinearModel.fit(hospital, 'Weight~1+Age*Sex*Smoker-Age:Sex:Smoker', 'MPG')
```

```
mdl =
```

Linear regression model:

```
Weight ~ 1 + Sex*Age + Sex*Smoker + Age*Smoker
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	118.7	7.0718	16.785	6.821e-30
Sex_Male	68.336	9.7153	7.0339	3.3386e-10
Age	0.31068	0.18531	1.6765	0.09699
Smoker_1	3.0425	10.446	0.29127	0.7714
Sex_Male:Age	-0.49094	0.24764	-1.9825	0.05037
Sex_Male:Smoker_1	0.9509	3.8031	0.25003	0.8031
Age:Smoker_1	-0.07288	0.26275	-0.27737	0.7821

Number of observations: 100, Error degrees of freedom: 93

Root Mean Squared Error: 8.75

R-squared: 0.898, Adjusted R-Squared 0.892

F-statistic vs. constant model: 137, p-value = 6.91e-44

The weight of the patients do not seem to differ significantly according to age, or the status of smoking, or interaction of these factors with gender at the 5% significance level.

## Robust Linear Regression Model

Fit a linear regression model of the Hald data using robust fitting.

Load the data.

```
load hald
X = ingredients; % predictor variables
y = heat; % response
```

Fit a robust linear model to the data.

```
mdl = fitlm(X,y,'linear','RobustOpts','on')
```

```
mdl =
```

Linear regression model (robust fit):

# LinearModel.fit

---

$$y \sim 1 + x1 + x2 + x3 + x4$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	60.09	75.818	0.79256	0.4509
x1	1.5753	0.80585	1.9548	0.086346
x2	0.5322	0.78315	0.67957	0.51596
x3	0.13346	0.8166	0.16343	0.87424
x4	-0.12052	0.7672	-0.15709	0.87906

Number of observations: 13, Error degrees of freedom: 8

Root Mean Squared Error: 2.65

R-squared: 0.979, Adjusted R-Squared 0.969

F-statistic vs. constant model: 94.6, p-value = 9.03e-07

## Algorithms

The main fitting algorithm is QR decomposition. For robust fitting, the algorithm is `robustfit`.

## Alternatives

You can also construct a linear model using `fitlm`.

You can construct a model in a range of possible models using `stepwiselm`. However, you cannot use robust regression and stepwise regression together.

## See Also

`LinearModel` | `predict` | `stepwiselm`

## Tutorials

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Predict or Simulate Responses to New Data” on page 9-40
- “Linear Regression Workflow” on page 9-44
- “Regression with Categorical Covariates” on page 2-59

## How To

- “Linear Regression” on page 9-12



## Purpose

Fit linear mixed-effects model using dataset arrays

## Syntax

```
lme = LinearMixedModel.fit(ds, formula)
lme = LinearMixedModel.fit(ds, formula, Name, Value)
```

## Description

`lme = LinearMixedModel.fit(ds, formula)` returns a linear mixed-effects model, specified by `formula`, fitted to the variables in the dataset array `ds`.

`lme = LinearMixedModel.fit(ds, formula, Name, Value)` returns a linear mixed-effects model with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the covariance pattern of the random-effects terms, the method to use in estimating the parameters, or options for the optimization algorithm.

## Tips

- Use the function `fitlme` instead of `LinearMixedModel.fit`.
- If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix`.

## Input Arguments

**ds - Input data**  
dataset array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a dataset array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-51). You must specify the model for the variables using `formula`.

### Data Types

single | double | char | cell

### formula - Formula for model specification

string of the form 'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'

Formula for model specification, specified as a string of the form 'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'. For full description, see “Formula” on page 21-966.

**Example:** 'y ~ treatment +(1|block)'

## Data Types

char

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## 'CovariancePattern' - Pattern of covariance matrix

'FullCholesky' (default) | string | square symmetric logical matrix | cell array of strings or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of 'CovariancePattern' and a string, a square symmetric logical matrix, or a cell array of strings or logical matrices.

If there are  $R$  random-effects terms, then the value of 'CovariancePattern' must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

'FullCholesky'

Default. Full covariance matrix using the Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Full'

Full covariance matrix, using the log-Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Diagonal'

Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$

where  $\sigma_b^2$  is the common variance of the random-effects terms.

'CompSymm'

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$$

where  $\sigma_{b1}^2$  is the common variance of the random-effects terms and  $\sigma_{b1,b2}$  is the common covariance between any two random-effects term .

PAT

Square symmetric logical matrix. If 'CovariancePattern' is defined by the matrix PAT, and if  $\text{PAT}(a,b) = \text{false}$ , then the (a,b) element of

the corresponding covariance matrix is constrained to be 0.

**Example:** `'CovariancePattern','Diagonal'`

**Example:** `'CovariancePattern',{'Full','Diagonal'}`

## 'FitMethod' - Method for estimating parameters

`'ML'` (default) | `'REML'`

Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of `'FitMethod'` and either of the following.

<code>'ML'</code>	Default. Maximum likelihood estimation
<code>'REML'</code>	Restricted maximum likelihood estimation

**Example:** `'FitMethod','REML'`

## 'Weights' - Observation weights

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and a vector of length  $n$ , where  $n$  is the number of observations.

## Data Types

`single` | `double`

## 'Exclude' - Indices for rows to exclude

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of `'Exclude'` and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

**Example:** `'Exclude',[13,67]`

# LinearMixedModel.fit

---

## Data Types

single | double | logical

## 'DummyVarCoding' - Coding to use for dummy variables

'reference' (default) | 'effects' | 'full'

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

**Example:** 'DummyVarCoding','effects'

## 'Optimizer' - Optimization algorithm

'quasinevton' (default) | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.

'quasinevton'	Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using <code>statset('LinearMixedModel')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>statset('LinearMixedModel')</code> .
'fminunc'	You must have Optimization Toolbox to specify this option. Change the options of the algorithm using <code>optimoptions('fminunc')</code> . If

you don't specify the options, then `LinearMixedModel` uses the default options of `optimoptions('fminunc')` with `'Algorithm'` set to `'quasi-newton'`.

**Example:** `'Optimizer','fminunc'`

## **'OptimizerOptions' - Options for optimization algorithm**

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizerOptions'` and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If `'Optimizer'` is `'fminunc'`, then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options `'fminunc'` uses. If `'Optimizer'` is `'fminunc'` and you do not supply `'OptimizerOptions'`, then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with `'Algorithm'` set to `'quasi-newton'`.
- If `'Optimizer'` is `'quasinevton'`, then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The `'quasinevton'` optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

## **'TolFun' - Relative tolerance on gradient of objective function**

1e-6 (default) | positive scalar value

# LinearMixedModel.fit

---

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

## **'TolX' - Absolute tolerance on step size**

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

## **'MaxIter' - Maximum number of iterations allowed**

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

## **'Display' - Level of display**

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

## **'StartMethod' - Method to start iterative optimization**

'default' (default) | 'random'

Method to start iterative optimization, specified as the comma-separated pair consisting of 'StartMethod' and either of the following.

'default'                      Default. An internally defined default value.

'random'                        A random initial value.

**Example:** 'StartMethod', 'random'

## **'Verbose' - Indicator to display optimization process on screen**

false (default) | true

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of 'Verbose' and either false or true. Default is false.



The setting for 'Verbose' overrides the field 'Display' in 'OptimizerOptions'.

**Example:** 'Verbose', true

## 'CheckHessian' - Indicator to check positive definiteness of Hessian

false (default) | true

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of 'CheckHessian' and either false or true. Default is false.

Specify 'CheckHessian' as true to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

**Example:** 'CheckHessian', true

## Output Arguments

### lme - Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

For properties and methods of this object, see LinearMixedModel.

## Examples

### Random-Intercept Model

Load the sample data.

```
load flu
```

The flu dataset array has a Date variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses, combine the nine columns

# LinearMixedModel.fit

---

corresponding to the regions into a tall array. The new dataset array, `flu2`, must have the response variable `FluRate`, the nominal variable `Region` that shows which region each estimate is from, the nationwide estimate `WtdILI`, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...
            'IndVarName','Region');
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with the nationwide a random intercept that varies by `Date`. The model corresponds to

$$y_{im} = \beta_0 + \beta_1 WtdILI_i + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable `Date`.  $b_{0m}$  is the random effect for level  $m$  of the grouping variable `Date` and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b \sim N(0, \sigma_b^2)$  and the error term has the distribution,  $\varepsilon \sim N(0, \sigma^2)$ .

```
lme = LinearMixedModel.fit(flu2,'FluRate ~ 1 + WtdILI + (1|Date)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

```
Formula:
```

```
FluRate ~ 1 + WtdILI + (1 | Date)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
-----	-----	---------------	----------

286.24      302.83      -139.12                      278.24

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.16385	0.057525	2.8484	466	0.0046
'WtdILI'	0.7236	0.032219	22.459	466	3.0e-46

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.17146

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

The confidence limits for the standard deviation of the random-effects term,  $\sigma^2_b$ , do not include 0 (0.13227, 0.22226), which indicates that the random-effects term is significant. You can also test the significance of the random-effects terms using the `compare` method.

The estimated value of an observation is the sum of the fixed effects and the random-effect value at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28 is

$$\begin{aligned} \hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 WtdILI_{28} + \hat{b}_{10/30/2005} \\ &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\ &= 1.46749, \end{aligned}$$

where  $\hat{b}$  is the BLUP of the random effects for the intercept. You can compute this value in the following way.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level);
```

```
y_hat =  
    1.4674
```

You can display the fitted value using the `fitted` method.

```
F = fitted(lme);  
F(28)  
  
ans =  
    1.4674
```

## Randomized-Block Design

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The dataset array shows the absolute deviations from the target quality characteristic measured from the products each of five operators manufacture over three different shifts, morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviations of the quality characteristics from the target value. This is simulated data.

Fit a linear mixed-effects model with a random intercept grouped by operator, to assess if there is significant difference in the performance according to the time of the shift. Use the restricted maximum likelihood method and 'effects' contrasts.

'effects' contrasts mean that the coefficients sum to 0, and `LinearMixedModel.fit` creates a matrix called a 'fixed effects design

matrix' to describe the effect of Shift. This matrix has two columns, *Shift\_Evening* and *Shift\_Morning*, where

$$Shift\_Evening = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases} \quad \text{and} \quad Shift\_Morning = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}.$$

The model corresponds to

Morning Shift:  $QCDev_{im} = \beta_0 + \beta_2 Shift - Morning_i + b_{0m} + \varepsilon_{im}$ ,  $m = 1, 2, \dots, 5$ ,

Evening Shift:  $QCDev_{im} = \beta_0 + \beta_1 Shift - Evening_i + b_{0m} + \varepsilon_{im}$ ,

Night Shift:  $QCDev_{im} = \beta_0 - \beta_1 Shift - Evening_i - \beta_2 Shift - Morning_i + b_{0m} + \varepsilon_{im}$ .

where  $b \sim N(0, \sigma_b^2)$  and  $\varepsilon \sim N(0, \sigma^2)$ .

```
lme = LinearMixedModel.fit(shift, 'QCDev ~ Shift + (1|Operator)', ...
'FitMethod', 'REML', 'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
QCDev ~ 1 + Shift + (1 | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

```
Fixed effects coefficients (95% CIs):
```

# LinearMixedModel.fit

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.6525	0.94109	3.8812	12	0.0021
'Shift_Evening'	-0.53293	0.31206	-1.7078	12	0.11
'Shift_Morning'	-0.91973	0.31206	-2.9473	12	0.012

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate	Lo
'(Intercept)'	'(Intercept)'	'std'	2.0457	0.

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.85462	0.52357	1.395

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

```
B = randomEffects(lme)
```

```
B =
```

```
0.5775  
1.1757  
-2.1715  
2.3655  
-1.9472
```

The estimated absolute deviation from the target quality characteristics for the third operator working in the evening shift is

$$\begin{aligned}\hat{y}_{Evening,Operator3} &= \hat{\beta}_0 + \hat{\beta}_1 Shift\_Evening + \hat{b}_{03} \\ &= 3.6525 - 0.53293 - 2.1715 \\ &= 0.94807.\end{aligned}$$

You can also display this value as follows.

```
F = fitted(lme);
```

```
F(shift.Shift=='Evening' & shift.Operator=='3')  
  
ans =  
  
    0.9481
```

Similarly, you can calculate the estimated absolute deviation from the target quality characteristics for the third operator working in the morning shift is

$$\begin{aligned}\hat{y}_{Morning,Operator3} &= \hat{\beta}_0 + \hat{\beta}_2 Shift\_Morning + \hat{b}_{03} \\ &= 3.6525 - 0.91973 - 2.1715 \\ &= 0.56127.\end{aligned}$$

You can also display this value in the following way.

```
F(shift.Shift=='Morning' & shift.Operator=='3')  
  
ans =  
  
    0.5613
```

The operator tends to make a smaller magnitude of error in the morning shift.

## Split-Plot Experiment

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and

# LinearMixedModel.fit

loamy. Each block is divided into five plots, where five types of tomato plants, (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. Then, the tomato plants in the plots are divided into subplots, where each subplot is treated by one of the four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently.

This model corresponds to

$$y_{imjk} = \beta_0 + \sum_{m=2}^4 \beta_{1m} I[F]_{im} + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + \sum_{j=2}^5 \sum_{m=2}^4 \beta_{3mj} I[F]_{im} I[T]_{ij} + b_{0k} S_k + b_{0jk} (S * T)_{jk} + \varepsilon_{imjk},$$

where  $i = 1, 2, \dots, 60$ , the index  $m$  corresponds to the fertilizer types,  $j$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $(S * T)_{jk}$  represents the  $j$ th tomato type nested in the  $k$ th soil type.  $I[F]_{im}$  is the dummy variable representing level  $m$  of the fertilizer. Similarly,  $I[T]_{ij}$  is the dummy variable representing the level  $j$  of the tomato type.

The random effects and observation error have the following prior distributions:  $b_{0k} \sim N(0, \sigma^2_S)$ ,  $b_{0jk} \sim N(0, \sigma^2_{S * T})$ , and  $\varepsilon_{imjk} \sim N(0, \sigma^2)$ .

```
lme = LinearMixedModel.fit(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (
```

```
lme =
```



Linear mixed-effects model fit by ML

Model information:

Number of observations	60
Fixed effects coefficients	20
Random effects coefficients	18
Covariance parameters	3

Formula:

Yield ~ 1 + Tomato\*Fertilizer + (1 | Soil) + (1 | Soil:Tomato)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
522.57	570.74	-238.29	476.57

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat
'(Intercept)'	77	8.5836	8.97
'Tomato_Grape'	-16	11.966	-1.33
'Tomato_Heirloom'	-6.6667	11.966	-0.557
'Tomato_Plum'	32.333	11.966	2.70
'Tomato_Vine'	-13	11.966	-1.08
'Fertilizer_2'	34.667	8.572	4.04
'Fertilizer_3'	33.667	8.572	3.92
'Fertilizer_4'	47.667	8.572	5.56
'Tomato_Grape:Fertilizer_2'	-2.6667	12.123	-0.219
'Tomato_Heirloom:Fertilizer_2'	-8	12.123	-0.659
'Tomato_Plum:Fertilizer_2'	-15	12.123	-1.23
'Tomato_Vine:Fertilizer_2'	-16	12.123	-1.33
'Tomato_Grape:Fertilizer_3'	16.667	12.123	1.37
'Tomato_Heirloom:Fertilizer_3'	3.3333	12.123	0.274
'Tomato_Plum:Fertilizer_3'	3.6667	12.123	0.302
'Tomato_Vine:Fertilizer_3'	3	12.123	0.247
'Tomato_Grape:Fertilizer_4'	13.333	12.123	1.09
'Tomato_Heirloom:Fertilizer_4'	-19	12.123	-1.56

# LinearMixedModel.fit

```
'Tomato_Plum:Fertilizer_4'      -2.6667      12.123      -0.21997
'Tomato_Vine:Fertilizer_4'      8.6667       12.123       0.71492
```

Random effects covariance parameters (95% CIs):

Group: Soil (3 Levels)

```
      Name1          Name2          Type      Estimate      L
      '(Intercept)'      '(Intercept)'      'std'      2.5028      0.
```

Group: Soil:Tomato (15 Levels)

```
      Name1          Name2          Type      Estimate      L
      '(Intercept)'      '(Intercept)'      'std'      10.225      6.
```

Group: Error

```
      Name          Estimate      Lower      Upper
      'Res Std'      10.499      8.5389     12.908
```

The  $p$ -values corresponding to the last 12 rows in the fixed-effects coefficients display (0.82701 to 0.47881) indicate that interaction coefficients between the tomato and fertilizer types are not significant. To test for the overall interaction between tomato and fertilizer, use the anova method after refitting the model using 'effects' contrasts.

The confidence interval for the standard deviations of the random-effects terms ( $\sigma^2_{\epsilon}$ ), where the intercept is grouped by soil is very large. This term does not appear significant.

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term `(1 | Soil)`.

```
lme = LinearMixedModel.fit(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

```
      Number of observations      60
```

```
Fixed effects coefficients      8
Random effects coefficients    15
Covariance parameters         2
```

Formula:

```
Yield ~ 1 + Tomato + Fertilizer + (1 | Soil:Tomato)
```

Model fit statistics:

```
AIC      BIC      LogLikelihood      Deviance
511.06   532      -245.53      491.06
```

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	77.733	7.3293	10.606	52
'Tomato_Grape'	-9.1667	9.6045	-0.95441	52
'Tomato_Heirloom'	-12.583	9.6045	-1.3102	52
'Tomato_Plum'	28.833	9.6045	3.0021	52
'Tomato_Vine'	-14.083	9.6045	-1.4663	52
'Fertilizer_2'	26.333	4.5004	5.8514	52
'Fertilizer_3'	39	4.5004	8.6659	52
'Fertilizer_4'	47.733	4.5004	10.607	52

Random effects covariance parameters (95% CIs):

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	10.02

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	12.325	10.024	15.153

You can compare the two models using the `compare` method with the simulated likelihood ratio test since both a fixed-effect and a random-effect term will be tested.

## Longitudinal Study with a Covariate

Navigate to a folder containing sample data.

# LinearMixedModel.fit

---

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs (A, B, C, D), and their weight loss is recorded over six two-week time periods. This is simulated data.

Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);
ds.Subject = nominal(ds.Subject);
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

`LinearMixedModel.fit` uses Program A as a reference and creates the necessary dummy variables  $I[.]$ . Since the model already has an intercept, `LinearMixedModel.fit` only creates dummy variables for program types B, C, D. This is also known as the 'reference' method of coding dummy variables. This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 Week_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i + \beta_5 I[PD]_i \\ + \beta_6 (Week_i * I[PB]_i) + \beta_7 (Week_i * I[PC]_i) + \beta_8 (Week_i * I[PD]_i) \\ + b_{0m} + b_{1m} Week_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{1m}$  and  $b_{0m}$  are random effects.  $IW$  stands for initial weight and  $I[.]$  is a dummy variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing program type B. The random effects and observation error have the

following prior distributions:  $b_{0m} \sim N(0, \sigma_0^2)$ ,  $b_{1m} \sim N(0, \sigma_1^2)$ , and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
lme = LinearMixedModel.fit(ds, 'y ~ InitialWeight + Program*Week + (Week | Subject)')
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

Formula:

```
y ~ 1 + InitialWeight + Program*Week + (1 + Week | Subject)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.66105	0.25892	2.5531	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_B'	0.36079	0.13139	2.746	111
'Program_C'	-0.033263	0.13117	-0.25358	111
'Program_D'	0.11317	0.13132	0.86175	111
'Week'	0.1732	0.067454	2.5677	111
'Program_B:Week'	0.038771	0.095394	0.40644	111
'Program_C:Week'	0.030543	0.095394	0.32018	111
'Program_D:Week'	0.033114	0.095394	0.34713	111

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

# LinearMixedModel.fit

---

Name1	Name2	Type	Estimate	
'(Intercept)'	'(Intercept)'	'std'	0.18407	L
'Week'	'(Intercept)'	'corr'	0.66841	C
'Week'	'Week'	'std'	0.15033	C

Group: Error			
Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The  $p$ -values 0.022863 and 0.011567 indicate significant effects of subject initial weights and time in the amount of weight lost. The weight loss of subjects who are in Program B is significantly different relative to the weight loss of subjects who are in Program A. The lower and upper limits of the covariance parameters for the random effects do not include 0, thus they are significant. You can also test the significance of the random effects using the `compare` method.

## Definitions

### Formula

In general, a formula for model specification is a string of the form `'y ~ terms'`. For the linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` and `random` contain the fixed-effects and the random-effects terms. Suppose a dataset array `ds` contains the following:

- A response variable,  $y$
- Predictor variables,  $X_j$ , which can be continuous or grouping variables
- Grouping variables,  $g_1, g_2, \dots, g_R$ ,

where the grouping variables in  $X_j$  and  $g_r$  can be categorical, logical, character arrays, or cell arrays of strings.

Then, in a formula of the form, `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`, the term `fixed` corresponds to a specification of the fixed-effects design matrix  $X$ , `random1` is a specification of the random-effects design matrix  $Z_1$  corresponding to grouping variable  $g_1$ ,

and similarly  $\text{random}_R$  is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable  $g_R$ . You can express the fixed and random terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1.*X2$ (elementwise multiplication of $X1$ and $X2$ )
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ . Here are some examples for linear mixed-effects model specification.

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, $X1$ and $X2$ . This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	Fixed effects for $X1$ and $X2$ . The implicit intercept term is suppressed by including $-1$ .

# LinearMixedModel.fit

Formula	Description
'y ~ 1 + (1   g1)'	Intercept plus random effect for each level of the grouping variable g1.
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1)'	Random intercept and slope, with possible correlation between them.
'y ~ X1 + (1   g1) + (-1 + X1   g1)'	Independent random intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

## Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.

$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q^*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ . For example, if



$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$

## Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## Alternatives

You can also construct a linear mixed-effects model using `fitlme`. If your data is in matrix format, then use `fitlmematrix`.

# LinearMixedModel.fit

---

## See Also

[LinearMixedModel](#) | [fitlmematrix](#) | [fitlmecompare](#) | [anova](#) |

## Purpose

Create Naive Bayes classifier object by fitting training data

## Syntax

```
nb = NaiveBayes.fit(training, class)
nb = NaiveBayes.fit(..., 'param1',val1, 'param2',val2, ...)
```

## Description

`nb = NaiveBayes.fit(training, class)` builds a `NaiveBayes` classifier object `nb`. `training` is an N-by-D numeric matrix of training data. Rows of `training` correspond to observations; columns correspond to features. `class` is a classing variable for `training` taking K distinct levels. Each element of `class` defines which class the corresponding row of `training` belongs to. `training` and `class` must have the same number of rows.

`nb = NaiveBayes.fit(..., 'param1',val1, 'param2',val2, ...)` specifies one or more of the following name/value pairs:

- `'Distribution'` – a string or a 1-by-D cell vector of strings, specifying which distributions `fit` uses to model the data. If the value is a string, `fit` models all the features using one type of distribution. `fit` can also model different features using different types of distributions. If the value is a cell vector, its `j`th element specifies the distribution `fit` uses for the `j`th feature. The available types of distributions are:

`'normal'`            Normal (Gaussian) distribution.  
(default)

`'kernel'`            Kernel smoothing density estimate.

# NaiveBayes.fit

---

- |        |   |
|--------|---|
| 'mvmn' | Multivariate multinomial distribution for discrete data. <code>fit</code> assumes each individual feature follows a multinomial model within a class. The parameters for a feature include the probabilities of all possible values that the corresponding feature can take.  |
| 'mn'   | Multinomial distribution for classifying the count-based data such as the bag-of-tokens model. In the bag-of-tokens model, the value of the $j$ th feature is the number of occurrences of the $j$ th token in this observation, so it must be a non-negative integer. When 'mn' is used, <code>fit</code> considers each observation as multiple trials of a multinomial distribution, and considers each occurrence of a token as one trial. The number of categories (bins) in this multinomial model is the number of distinct tokens, i.e., the number of columns of training. |
- 'Prior' – The prior probabilities for the classes, specified as one of the following:

'empirical' (default)	<code>fit</code> estimates the prior probabilities from the relative frequencies of the classes in training.
'uniform'	The prior probabilities are equal for all classes.

vector	A numeric vector of length $K$ specifying the prior probabilities in the class order of <code>class</code> .
structure	A structure <code>S</code> containing class levels and their prior probabilities. <code>S</code> must have two fields: <ul style="list-style-type: none"><li>• <code>S.prob</code>: A numeric vector of prior probabilities.</li><li>▪ <code>S.class</code>: A vector of the same type as <code>class</code>, containing unique class levels indicating the class for the corresponding element of <code>prob</code>. <code>S.class</code> must contain all the <math>K</math> levels in <code>class</code>. It can also contain classes that do not appear in <code>class</code>. This can be useful if <code>training</code> is a subset of a larger training set. <code>fit</code> ignores any classes that appear in <code>S.class</code> but not in <code>class</code>.</li></ul>

If the prior probabilities don't sum to one, `fit` will normalize them.

- `'KSWidth'` – The bandwidth of the kernel smoothing window. The default is to select a default bandwidth automatically for each combination of feature and class, using a value that is optimal for a Gaussian distribution. You can specify the value as one of the following:

scalar	Width for all features in all classes.
row vector	1-by- $D$ vector where the $j$ th element is the bandwidth for the $j$ th feature in all classes.
column vector	$K$ -by-1 vector where the $i$ th element specifies the bandwidth for all features in the $i$ th class. $K$ represents the number of class levels.

# NaiveBayes.fit

---

**matrix** K-by-D matrix  $M$  where  $M(i, j)$  specifies the bandwidth for the  $j$ th feature in the  $i$ th class.

**structure** A structure  $S$  containing class levels and their bandwidths.  $S$  must have two fields:

- **S.width** – A numeric array of bandwidths specified as a row vector, or a matrix with  $D$  columns.
- **S.class** – A vector of the same type as **class**, containing unique class levels indicating the class for the corresponding row of **width**.
- **'KSSupport'** – The regions where the density can be applied. It can be a string, a two-element vector as shown below, or a 1-by- $D$  cell array of these values:

<b>'unbounded'</b> (default)	The density can extend over the whole real line.
<b>'positive'</b>	The density is restricted to positive values.
<b>[L,U]</b>	A two-element vector specifying the finite lower bound $L$ and upper bound $U$ for the support of the density.
- **'KSType'** – The type of kernel smoother to use. It can be a string or a 1-by- $D$  cell array of strings. Each string can be **'normal'** (default), **'box'**, **'triangle'**, or **'epanechnikov'**.

## How To

- “Naive Bayes Classification” on page 15-38
- “Grouping Variables” on page 2-51

## Purpose

Fit nonlinear regression model

## Syntax

```
mdl = NonLinearModel.fit(ds,modelfun,beta0)
mdl = NonLinearModel.fit(X,y,modelfun,beta0)
mdl = NonLinearModel.fit(...,modelfun,beta0,Name,Value)
```

## Description

`mdl = NonLinearModel.fit(ds,modelfun,beta0)` fits the model specified by `modelfun` to variables in the dataset array `ds`, and returns the nonlinear model `mdl`. `NonLinearModel.fit` estimates model coefficients using an iterative procedure starting from the initial values in `beta0`.

`mdl = NonLinearModel.fit(X,y,modelfun,beta0)` fits a nonlinear regression model using the column vector `y` as a response variable and the columns of the matrix `X` as predictor variables.

`mdl = NonLinearModel.fit(...,modelfun,beta0,Name,Value)` fits a nonlinear regression model with additional options specified by one or more `Name, Value` pair arguments.

## Tips

- Use the function `fitnlm` instead of `NonLinearModel.fit`.

## Input Arguments

### **ds** - Input data

dataset array

Input data, specified as a dataset array. If you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables and response variable must be numeric.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### Data Types

single | double | logical

### **X** - Predictor variables

# NonLinearModel.fit

---

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

## Data Types

single | double | logical

## **y** - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

## Data Types

single | double

## **modelfun** - Functional form of the model

function handle | string of the form 'y ~ f(b1,b2,...,bj,x1,x2,...,xk)'

Functional form of the model, specified as either of the following.

- Function handle `@modelfun` or `@(b,x)modelfun`, where
  - $b$  is a coefficient vector with the same number of elements as `beta0`.
  - $x$  is a matrix with the same number of columns as  $X$  or the number of predictor variable columns of  $ds$ .

`modelfun(b,x)` returns a column vector that contains the same number of rows as  $x$ . Each row of the vector is the result of evaluating `modelfun` on the corresponding row of  $x$ . In other words, `modelfun` is a vectorized function, one that operates on all data rows and returns all evaluations in one function call.



- String of the form  $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ , where  $f$  represents a scalar function of the scalar coefficient variables  $b_1, \dots, b_j$  and the scalar data variables  $x_1, \dots, x_k$ .

## beta0 - Coefficients

numeric vector

Coefficients for the nonlinear model, specified as a numeric vector. NonLinearModel starts its search for optimal coefficients from beta0.

## Data Types

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

## 'CoefficientNames' - Names of the model coefficients

{'b1', 'b2', ..., 'bk'} (default) | cell array of strings

Names of the model coefficients, specified as a cell array of strings.

## Data Types

char

## 'ErrorModel' - Form of the error variance model

'constant' (default) | 'proportional' | 'combined'

Form of the error variance model, specified as one of the following. Each model defines the error using a standard mean-zero and unit-variance variable  $e$  in combination with independent components: the function value  $f$ , and one or two parameters  $a$  and  $b$

# NonLinearModel.fit

---

'constant' (default)	$y = f + ae$
'proportional'	$y = f + bfe$
'combined'	$y = f + (a + b f )e$

The only allowed error model when using `Weights` is 'constant'.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than 'constant'.

---

**Example:** 'ErrorModel', 'proportional'

## 'ErrorParameters' - Initial estimates of the error model parameters

numeric array

Initial estimates of the error model parameters for the chosen `ErrorModel`, specified as a numeric array.

Error Model	Parameters	Default Values
'constant'	$a$	1
'proportional'	$b$	1
'combined'	$a, b$	[1,1]

You can only use the 'constant' error model when using `Weights`.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than 'constant'.

---

For example, if 'ErrorModel' has the value 'combined', you can specify the starting value 1 for  $a$  and the starting value 2 for  $b$  as follows.

**Example:** 'ErrorParameters',[1,2]

## Data Types

char

## 'Exclude' - Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

## Data Types

single | double | logical

## 'Options' - Options for controlling the iterative fitting procedure

[] (default) | structure

Options for controlling the iterative fitting procedure, specified as a structure created by `statset`. The relevant fields are the nonempty fields in the structure returned by the call `statset('nlinfit')`.

# NonLinearModel.fit

Option	Meaning	Default
DerivStep	Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics Toolbox function using the options structure.	$\text{eps}^{(1/3)}$
Display	Amount of information displayed by the fitting algorithm. <ul style="list-style-type: none"><li>• 'off' — Displays no information.</li><li>• 'final' — Displays the final output.</li><li>• 'iter' — Displays iterative output to the Command Window.</li></ul>	'off'
FunValCheck	String indicating to check for invalid values, such as NaN or Inf, from the model function.	'on'
MaxIter	Maximum number of iterations allowed. Positive integer.	200
RobustWgtFun	Weight function for robust fitting. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. If you use a function handle, give a Tune constant. See “Robust Options” on page 21-1143	[]
Tune	Tuning constant used in robust fitting to normalize the residuals before applying the weight function. A positive scalar. Required if the weight function is specified as a function handle.	See “Robust Options” on page 21-1143 for the default, which depends on RobustWgtFun.

Option	Meaning	Default
ToIFun	Termination tolerance for the objective function value. Positive scalar.	1e-8
ToIX	Termination tolerance for the parameters. Positive scalar.	1e-8

## Data Types

struct

### 'PredictorVars' - Predictor variables

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array ds, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the ds dataset array, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in X, or all variables in ds except for ResponseVar.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** 'PredictorVars',[2,3]

**Example:** 'PredictorVars',logical([0 1 1 0 0 0])

## Data Types

single | double | logical | cell

### 'ResponseVar' - Response variable

last column in dataset array ds (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a

# NonLinearModel.fit

---

string of the variable name in the dataset array `ds`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a dataset array `ds`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

**Example:** `'ResponseVar','yield'`

**Example:** `'ResponseVar',[4]`

**Example:** `'ResponseVar',logical([0 0 0 1 0 0])`

## Data Types

`single` | `double` | `logical` | `char`

## 'VarNames' - Names of variables in fit

`{'x1','x2',...,'xn','y'}` (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings including the names for the columns of `X` first, and the name for the response variable `y` last.

`'VarNames'` is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

## Example:

`'VarNames',{'Horsepower','Acceleration','Model_Year','MPG'}`

## Data Types

`cell`

## 'Weights' - Observation weights

`ones(n,1)` (default) | vector of nonnegative scalar values | function handle

Observation weights, specified as a vector of nonnegative scalar values or function handle.

- If you specify a vector, then it must have  $n$  elements, where  $n$  is the number of rows in `ds` or `y`.
- If you specify a function handle, then the function must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights, `W`, `NonLinearModel` estimates the error variance at observation `i` by  $MSE * (1/W(i))$ , where `MSE` is the mean squared error.

### Data Types

`single` | `double` | `function_handle`

## Output Arguments

### `mdl` - Nonlinear model

`NonLinearModel` object

Nonlinear model representing a least-squares fit of the response to the data, returned as a `NonLinearModel` object.

If the `Options` structure contains a nonempty `RobustWgtFun` field, the model is not a least-squares fit, but uses the `RobustWgtFun` robust fitting function.

For properties and methods of the nonlinear model object, `mdl`, see the `NonLinearModel` class page.

# NonLinearModel.fit

## Definitions

## Robust Options

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) \cdot \sin(r) \cdot / r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) \cdot (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 \cdot / (1 + r.^2)$	2.385
'fair'	$w = 1 \cdot / (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 \cdot / \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) \cdot / r$	1.205
'talwar'	$w = 1 \cdot * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985
[]	No robust fitting	—

## Examples

### Nonlinear Model from a Dataset Array

Create a nonlinear model for auto mileage based on the carbig data.

Load the data and create a nonlinear model.

```
load carbig
ds = dataset(Horsepower,Weight,MPG);
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = NonLinearModel.fit(ds,modelfun,beta0)
```

```
mdl =
```

Nonlinear regression model:



```
MPG ~ b1 + b2*Horsepower^b3 + b4*Weight^b5
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

Number of observations: 392, Error degrees of freedom: 387

Root Mean Squared Error: 3.96

R-Squared: 0.745, Adjusted R-Squared 0.743

F-statistic vs. constant model: 283, p-value = 1.79e-113

## Nonlinear Model from Matrix Data

Create a nonlinear model for auto mileage based on the carbig data.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = NonLinearModel.fit(X,y,modelfun,beta0)
```

```
mdl =
```

Nonlinear regression model:

```
y ~ b1 + b2*x1^b3 + b4*x2^b5
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
--	----------	----	-------	--------

# NonLinearModel.fit

---

b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

Number of observations: 392, Error degrees of freedom: 387  
Root Mean Squared Error: 3.96  
R-Squared: 0.745, Adjusted R-Squared 0.743  
F-statistic vs. constant model: 283, p-value = 1.79e-113

## Adjust Fitting Options in the Nonlinear Model

Create a nonlinear model for auto mileage based on the `carbig` data. Strive for more accuracy by lowering the `TolFun` option, and observe the iterations by setting the `Display` option.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
```

Create options to lower `TolFun` and to report iterative display, and create a model using the options.

```
opts = statset('Display','iter','TolFun',1e-10);
mdl = NonLinearModel.fit(X,y,modelfun,beta0,'Options',opts);
```

Iteration	SSE	Norm of Gradient	Norm of Step
0	1.82248e+06		
1	678600	788810	1691.07
2	616716	6.12739e+06	45.4738

```
%%%      Many iterations deleted      %%%  
  
      122      6068.48      1.56393      0.629325  
      123      6068.48      1.13809      0.432543  
      124      6068.48      0.295962      0.297511  
  
Iterations terminated: relative change in SSE less than OPTIONS.TolFun
```

## Specify Nonlinear Regression Using Model String Syntax

Specify a nonlinear regression model for estimation using a function handle or model string syntax.

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Use a function handle to specify the Hougen-Watson model for the rate data.

```
mdl = NonLinearModel.fit(X,y,@hougen,beta0)
```

```
mdl =
```

```
Nonlinear regression model:  
y ~ hougen(b,X)
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

# NonLinearModel.fit

---

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 0.193  
R-Squared: 0.999, Adjusted R-Squared 0.998  
F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

Alternatively, you can use a string expression to specify the Hougén-Watson model for the rate data.

```
myfun = 'y~(b1*x2-x3/b5)/(1+b2*x1+b3*x2+b4*x3)';  
mdl2 = NonLinearModel.fit(X,y,myfun,beta0)
```

```
mdl2 =
```

Nonlinear regression model:

$$y \sim (b_1 \cdot x_2 - x_3 / b_5) / (1 + b_2 \cdot x_1 + b_3 \cdot x_2 + b_4 \cdot x_3)$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 0.193  
R-Squared: 0.999, Adjusted R-Squared 0.998  
F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

## Estimate Nonlinear Regression Using Robust Fitting Options

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp\{-b_3 x\} + \varepsilon,$$

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x)(b(1)+b(2)*exp(-b(3)*x));
```

```
rng('default') % for reproducibility
```

```
b = [1;3;2];
```

```
x = exprnd(2,100,1);
```

```
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

Set robust fitting options.

```
opts = statset('nlinfit');
```

```
opts.RobustWgtFun = 'bisquare';
```

Fit the nonlinear model using the robust fitting options. Here, use a string expression to specify the model.

```
b0 = [2;2;2];
```

```
modelstr = 'y ~ b1 + b2*exp(-b3*x)';
```

```
mdl = NonLinearModel.fit(x,y,modelstr,b0,'Options',opts)
```

```
mdl =
```

```
Nonlinear regression model (robust fit):
```

```
y ~ b1 + b2*exp( - b3*x)
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	1.0218	0.07202	14.188	2.1344e-25
b2	3.6619	0.25429	14.401	7.974e-26
b3	2.9732	0.38496	7.7232	1.0346e-11

```
Number of observations: 100, Error degrees of freedom: 97
```

```
Root Mean Squared Error: 0.501
```

# NonLinearModel.fit

---

R-Squared: 0.807, Adjusted R-Squared 0.803  
F-statistic vs. constant model: 203, p-value = 2.34e-35

## Fit Nonlinear Regression Model Using Weights Function Handle

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Specify a function handle for observation weights. The function accepts the model fitted values as input, and returns a vector of weights.

```
a = 1; b = 1;  
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
```

Fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
mdl = NonLinearModel.fit(X,y,@hougen,beta0,'Weights',weights)
```

```
mdl =
```

Nonlinear regression model:

$y \sim \text{hougen}(b,X)$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	0.83085	0.58224	1.427	0.19142
b2	0.04095	0.029663	1.3805	0.20477
b3	0.025063	0.019673	1.274	0.23842
b4	0.080053	0.057812	1.3847	0.20353
b5	1.8261	1.281	1.4256	0.19183

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 0.037  
R-Squared: 0.998, Adjusted R-Squared 0.998  
F-statistic vs. zero model: 1.14e+03, p-value = 3.49e-11

## Nonlinear Regression Model Using Nonconstant Error Model

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error variance model.

```
mdl = NonLinearModel.fit(X,y,@hougen,beta0,'ErrorModel','combined')
```

```
mdl =
```

```
Nonlinear regression model:  
y ~ hougen(b,X)
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86702	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075158	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 1.27  
R-Squared: 0.999, Adjusted R-Squared 0.998  
F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

# NonLinearModel.fit

---

**Algorithms** NonLinearModel.fit uses the same fitting algorithm as nlinfit.

**Alternatives** You can also construct a nonlinear model using fitnlm.

## References

[1] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

[2] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.

[3] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.

**See Also** NonLinearModel | nlinfit

**Related Examples**

- "Examine Quality and Adjust the Fitted Model" on page 11-8
- "Predict or Simulate Responses to New Data" on page 11-12
- "Nonlinear Regression Workflow" on page 11-16

**Concepts**

- "Nonlinear Regression" on page 11-2



## Purpose

Binary decision tree for regression

## Syntax

```
tree = RegressionTree.fit(X,Y)
tree = RegressionTree.fit(X,Y,Name,Value)
```

## Description

`tree = RegressionTree.fit(X,Y)` returns a regression tree based on the input variables (also known as predictors, features, or attributes) `X` and output (response) `Y`. `tree` is a binary tree where each branching node is split based on the values of a column of `X`.

`tree = RegressionTree.fit(X,Y,Name,Value)` fits a tree with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

Note that using the `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'` options results in a tree of class `RegressionPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `tree` is of class `RegressionTree`, and you can use the `predict` method to make predictions.

## Input Arguments

### X

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

`RegressionTree.fit` considers NaN values in `X` as missing values. `RegressionTree.fit` does not use observations with all missing values for `X` the fit. `RegressionTree.fit` uses observations with some missing values for `X` to find splits on variables for which these observations have valid values.

### Y

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

# RegressionTree.fit

---

`RegressionTree.fit` considers NaN values in `Y` to be missing values. `RegressionTree.fit` does not use observations with missing values for `Y` in the fit.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## **'CategoricalPredictors'**

List of categorical predictors. Pass `CategoricalPredictors` as one of:

- A numeric vector with indices from 1 to `p`, where `p` is the number of columns of `X`.
- A logical vector of length `p`, where a `true` entry means that the corresponding column of `X` is a categorical variable.
- `'all'`, meaning all predictors are categorical.
- A cell array of strings, where each element in the array is the name of a predictor variable. The names must match entries in the `PredictorNames` property.
- A character matrix, where each row of the matrix is a name of a predictor variable. Pad the names with extra blanks so each row of the character matrix has the same length.

**Default:** []

## **'crossval'**

Flag to grow a cross-validated decision tree. Possible values are `'on'` or `'off'`.

If 'on', `RegressionTree.fit` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the 'kfold', 'holdout', 'leaveout', or 'cvpartition' name-value pair arguments. Note that you can only use one of these four options ('kfold', 'holdout', 'leaveout', or 'cvpartition') at a time when creating a cross-validated tree.

Alternatively, cross-validate `tree` later using the `crossval` method.

**Default:** 'off'

## **'cvpartition'**

Partition created with `cvpartition` to use in cross-validated tree.

Note that if you use 'cvpartition', you cannot use any of the 'kfold', 'holdout', or 'leaveout' name-value pair arguments.

## **'holdout'**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1.

Note that if you use 'holdout', you cannot use any of the 'cvpartition', 'kfold', or 'leaveout' name-value pair arguments.

## **'kfold'**

Number of folds to use in a cross-validated tree, a positive integer.

Note that if you use 'kfold', you cannot use any of the 'cvpartition', 'holdout', or 'leaveout' name-value pair arguments.

**Default:** 10

## **'leaveout'**

# RegressionTree.fit

---

Use leave-one-out cross validation by setting to 'on'.

Note that if you use 'leaveout', you cannot use any of the 'cvpartition', 'holdout', or 'kfold' name-value pair arguments.

## **'MergeLeaves'**

When 'on', RegressionTree merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', RegressionTree does not merge leaves.

**Default:** 'on'

## **'MinLeaf'**

Each leaf has at least MinLeaf observations per tree leaf. If you supply both MinParent and MinLeaf, RegressionTree uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** 1

## **'MinParent'**

Each branch node in the tree has at least MinParent observations. If you supply both MinParent and MinLeaf, RegressionTree uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** 10

## **'NVarToSample'**

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all'

## **'PredictorNames'**

A cell array of names for the predictor variables, in the order in which they appear in X.

**Default:** { 'x1', 'x2', ... }

## **'Prune'**

When 'on', RegressionTree computes the full tree and the optimal sequence of pruned subtrees. When 'off' RegressionTree computes the full tree without pruning.

**Default:** 'on'

## **'PruneCriterion'**

String with the pruning criterion, always 'error'.

**Default:** 'error'

## **'QEToler'**

Defines tolerance on quadratic error per node for regression trees. Splitting nodes stops when quadratic error per node drops below  $QEToler * QED$ , where QED is the quadratic error for the entire data computed before the decision tree is grown.

**Default:** 1e-6

## **'ResponseName'**

Name of the response variable Y, a string.

**Default:** 'Y'

## **'ResponseTransform'**

Function handle for transforming the raw response values. The function handle should accept a matrix of response values and

# RegressionTree.fit

---

return a matrix of the same size. The default string 'none' means  $@(x)x$ , or no transformation.

Add or change a ResponseTransform function by dot addressing:

```
tree.ResponseTransform = @function
```

**Default:** 'none'

## 'SplitCriterion'

Criterion for choosing a split, always the string 'MSE', meaning mean squared error.

**Default:** 'MSE'

## 'Surrogate'

String describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', `RegressionTree.fit` finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, `RegressionTree.fit` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `RegressionTree.fit` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## 'Weights'

Vector of observation weights. The length of weights is the number of rows in X.

**Default:** `ones(size(X,1),1)`

## Output Arguments

**tree**

A regression tree object.

Note that using the 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition' options results in a tree of class `RegressionPartitionedModel`. You cannot use a partitioned tree for prediction, so this kind of tree does not have a `predict` method.

Otherwise, `tree` is of class `RegressionTree`, and you can use the `predict` method to make predictions.

## Examples

Load the data in `carsmall.mat`, and make a regression tree to predict the mileage of cars based on their weights and numbers of cylinders:

```
load carsmall
tree = RegressionTree.fit([Weight, Cylinders],MPG,...
    'MinParent',20,...
    'PredictorNames',{'W','C'})
```

```
tree =
```

```
RegressionTree
    PredictorNames: {'W' 'C'}
    ResponseName: 'Y'
    ResponseTransform: 'none'
    CategoricalPredictors: []
    NObservations: 94
```

Properties, Methods

# RegressionTree.fit

---

Predict the mileage of a car that weighs 2200 lbs and has four cylinders:

```
predict(tree, [2200, 4])
```

```
ans =  
    29.6111
```

## See Also

`predict`

## How To

- “Splitting Categorical Predictors” on page 16-56



**Purpose** Create generalized linear regression model

**Syntax**

```
mdl = fitglm(ds)
mdl = fitglm(X,y)
mdl = fitglm( ___,modelspec)
mdl = fitglm( ___,Name,Value)
```

**Description** `mdl = fitglm(ds)` returns a generalized linear model fit to variables in the dataset array `ds`. By default, `fitglm` takes the last variable as the response variable.

`mdl = fitglm(X,y)` returns a generalized linear model of the responses `y`, fit to the data matrix `X`.

`mdl = fitglm( ___,modelspec)` returns a generalized linear model of the type you specify in `modelspec`.

`mdl = fitglm( ___,Name,Value)` returns a generalized linear model with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify which variables are categorical, the distribution of the response variable, and the link function to use.

## Input Arguments

### **ds - Input data**

dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

## Data Types

single | double | logical

## X - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

## Data Types

single | double | logical

## y - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

## Data Types

single | double

## modelspec - Model specification

string specifying the model |  $t$ -by- $(p+1)$  terms matrix | string of the form 'Y ~ terms'

Model specification, which is the starting model for `stepwiseglm`, specified as one of the following:

- String specifying the type of model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p+1)$  matrix, namely “Terms Matrix” on page 21-1013, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- String representing a “Formula” on page 21-1016 in the form

`'Y ~ terms'`,

where the terms are in “Wilkinson Notation” on page 21-1017.

**Example:** `'quadratic'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### Example:

```
'Distribution','normal','link','probit','Exclude',[23,59]
```

specifies that the distribution of the response is normal, and instructs `fitglm` to use the probit link function and exclude the 23rd and 59th observations from the fit.

## 'BinomialSize' - Number of trials for binomial distribution

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the response. This is the parameter `n` for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

## Data Types

single | double

## 'CategoricalVars' - Categorical variables

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.
- If data is in matrix `X`, then the default value of this name-value pair argument is an empty matrix `[]`. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** `'CategoricalVars',[2,3]`

**Example:** `'CategoricalVars',logical([0 1 1 0 0 0])`

### Data Types

single | double | logical

### 'DispersionFlag' - Indicator to compute dispersion parameter

false for 'binomial' and 'poisson' distributions (default) | true

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

true	Estimate a dispersion parameter when computing standard errors
false	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

**Example:** `'DispersionFlag',true`

### 'Distribution' - Distribution of the response variable

'normal' (default) | 'binomial' | 'poisson' | 'gamma' | 'inverse gaussian'

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

'normal'	Normal distribution
'binomial'	Binomial distribution
'poisson'	Poisson distribution
'gamma'	Gamma distribution
'inverse gaussian'	Inverse Gaussian distribution

**Example:** 'Distribution','gamma'

### **'Exclude' - Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

### **Data Types**

single | double | logical

### **'Intercept' - Indicator for constant term**

true (default) | false

Indicator the for constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

Example: 'Intercept', false

### 'Link' - Link function

The canonical link function (default) | scalar value | structure

Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
'identity'	$f(\mu) = \mu$	$\mu = Xb$
'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'probit'	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$
'comloglog'	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
'reciprocal'	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
p (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$
S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size: <ul style="list-style-type: none"> <li>• S.Link — The link function</li> <li>• S.Inverse — The inverse link function</li> <li>• S.Derivative — The derivative of the link function</li> </ul>	$f(\mu) = S.Link(\mu)$	$\mu = S.Inverse(Xb)$

The link function defines the relationship  $f(\mu) = X^*b$  between the mean response  $\mu$  and the linear combination of predictors  $X^*b$ .

For more information on the canonical link functions, see [Definitions](#).

**Example:** 'Link', 'probit'

## 'Offset' - Offset variable

[] (default) | vector | string

Offset variable in the fit, specified as the comma-separated pair consisting of 'Offset' and a vector or name of a variable with the same length as the response.

`fitglm` and `stepwiseglm` use `Offset` as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the `Offset` predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor `A`. By using the log link function and by specifying `log(A)` as an offset, you can force the model to satisfy this theoretical constraint.

## Data Types

single | double | char

## 'PredictorVars' - Predictor variables

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array `ds`, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the `ds` dataset array, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `ds` except for `ResponseVar`.



For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** `'PredictorVars',[2,3]`

**Example:** `'PredictorVars',logical([0 1 1 0 0 0])`

### Data Types

`single | double | logical | cell`

### 'ResponseVar' - Response variable

last column in dataset array `ds` (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a string of the variable name in the dataset array `ds`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a dataset array `ds`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

**Example:** `'ResponseVar','yield'`

**Example:** `'ResponseVar',[4]`

**Example:** `'ResponseVar',logical([0 0 0 1 0 0])`

### Data Types

`single | double | logical | char`

### 'VarNames' - Names of variables in fit

`{'x1','x2',...,'xn','y'}` (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of strings including the names for the columns of `X` first, and the name for the response variable `y` last.

'VarNames' is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

**Example:**

```
'VarNames', {'Horsepower', 'Acceleration', 'Model_Year', 'MPG'}
```

**Data Types**

cell

**'Weights' - Observation weights**

ones(n,1) (default) | *n*-by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an *n*-by-1 vector of nonnegative scalar values, where *n* is the number of observations.

**Data Types**

single | double

## Output Arguments

**mdl - Generalized linear model**

GeneralizedLinearModel object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a GeneralizedLinearModel object.

For properties and methods of the generalized linear model object, mdl, see the GeneralizedLinearModel class page.

## Examples

**Fit a Logistic Regression Model**

Make a logistic binomial model of the probability of smoking as a function of age, weight, and sex, using a two-way interactions model.

Load the hospital dataset array.

```
load hospital  
ds = hospital; % just to use the ds name
```

Specify the model using a formula that allows up to two-way interactions between the variables age, weight, and sex. Smoker is the response variable.

```
modelspec = 'Smoker ~ Age*Weight*Sex - Age:Weight:Sex';
```

Fit a logistic binomial model.

```
mdl = fitglm(ds,modelspec,'Distribution','binomial')
```

```
mdl =
```

Generalized Linear regression model:

```
logit(Smoker) ~ 1 + Sex*Age + Sex*Weight + Age*Weight
Distribution = Binomial
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	-6.0492	19.749	-0.3063	0.75938
Sex_Male	-2.2859	12.424	-0.18399	0.85402
Age	0.11691	0.50977	0.22934	0.81861
Weight	0.031109	0.15208	0.20455	0.83792
Sex_Male:Age	0.020734	0.20681	0.10025	0.92014
Sex_Male:Weight	0.01216	0.053168	0.22871	0.8191
Age:Weight	-0.00071959	0.0038964	-0.18468	0.85348

100 observations, 93 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 5.07, p-value = 0.535

All of the p-values (under pValue) are large. This means none of the coefficients are significant. The large  $p$ -value for the test of the model, 0.535, indicates that this model might not differ statistically from a constant model.

## GLM for Poisson Response

Create sample data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng('default') % for reproducibility
X = randn(100,7);
mu = exp(X(:,[1 3 6]))* [.4;.2;.3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = fitglm(X,y,'linear','Distribution','poisson')
```

```
mdl =
```

```
Generalized Linear regression model:
```

```
log(y) ~ 1 + x1 + x2 + x3 + x4 + x5 + x6 + x7
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	0.88723	0.070969	12.502	7.3149e-36
x1	0.44413	0.052337	8.4858	2.1416e-17
x2	0.0083388	0.056527	0.14752	0.88272
x3	0.21518	0.063416	3.3932	0.00069087
x4	-0.058386	0.065503	-0.89135	0.37274
x5	-0.060824	0.073441	-0.8282	0.40756
x6	0.34267	0.056778	6.0352	1.5878e-09
x7	0.04316	0.06146	0.70225	0.48252

```
100 observations, 92 error degrees of freedom
```

```
Dispersion: 1
```

```
Chi^2-statistic vs. constant model: 119, p-value = 1.55e-22
```

The  $p$ -values of  $2.14e-17$ ,  $0.00069$ , and  $1.58e-09$  indicate that the coefficients of the variables  $x_1$ ,  $x_3$ , and  $x_6$  are statistically significant.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

T =

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

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

T =

```

    0    0    0
    1    0    0
    0    1    0
    1    1    0
    0    2    0

```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

mdl =

Linear regression model:

$$y \sim 1 + x1*x2 + x2^2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.000196
x1	0.54418	0.57125	0.95261	0.343
x2	-0.012781	0.0060312	-2.1192	0.0368
x1:x2	-0.00010892	0.00017925	-0.6076	0.5
x2^2	9.7518e-07	7.5389e-07	1.2935	0.199

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];  
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

mdl =

Linear regression model:  
y ~ 1 + x2

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of fitlm in the previous step.

## Formula

A formula for model specification is a string of the form

```
'Y ~ terms',
```

where

- Y is the response name.
- *terms* contains
  - Variable names
  - + means include the next variable
  - - means do not include the next variable



- `:` defines an interaction, a product of terms
- `*` defines an interaction **and all lower-order terms**
- `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
- `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

' $Y \sim A + B + C$ ' means a three-variable linear model with intercept.

' $Y \sim A + B + C - 1$ ' is a three-variable linear model without intercept.

' $Y \sim A + B + C + B^2$ ' is a three-variable model with intercept and a  $B^2$  term.

' $Y \sim A + B^2 + C$ ' is the same as the previous example because  $B^2$  includes a  $B$  term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of  $A$  with each of the other variables.

### Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
A^k, where k is a positive integer	A, A <sup>2</sup> , ..., A <sup>k</sup>
A + B	A, B
A*B	A, B, A*B
A:B	A*B only
-B	Do not include B
A*B + C	A, B, C, A*B
A + B + C + A:B	A, B, C, A*B
A*B*C - A:B:C	A, B, C, A*B, A*C, B*C
A*(B + C)	A, B, C, A*B, A*C

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using -1.

## Canonical Link Function

The default link function for a generalized linear model is the *canonical link function*.

## Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

**Tips**

- The generalized linear model `mdl` is a standard linear model unless you specify otherwise with the `Distribution` name-value pair.
- For methods such as `plotResiduals` or `devianceTest`, or properties of the `GeneralizedLinearModel` object, see `GeneralizedLinearModel`.

**Alternative Functionality**

Use `stepwiseglm` to select a model specification automatically. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

**References**

[1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.

[2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.

[3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

**See Also**

`GeneralizedLinearModel` | `stepwiseglm`

**Related Examples**

- “Generalized Linear Model Workflow” on page 10-42

**Concepts**

- “Generalized Linear Models” on page 10-12

# fitlm

---

**Purpose** Create linear regression model

**Syntax**

```
mdl = fitlm(ds)
mdl = fitlm(ds,modelspec)

mdl = fitlm(X,y)
mdl = fitlm(X,y,modelspec)

mdl = fitlm( __ ,Name,Value)
```

**Description** `mdl = fitlm(ds)` returns a linear model fit to variables in the dataset array `ds`. By default, `fitlm` takes the last variable as the response variable.

`mdl = fitlm(ds,modelspec)` returns a linear model of the type you specify in `modelspec` fit to variables in the dataset array `ds`.

`mdl = fitlm(X,y)` returns a linear model of the responses `y`, fit to the data matrix `X`.

`mdl = fitlm(X,y,modelspec)` returns a linear model of the type you specify in `modelspec` for the responses `y`, fit to the data matrix `X`.

`mdl = fitlm( __ ,Name,Value)` returns a linear model with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify which variables are categorical, perform robust regression, or use observation weights.

## **Input Arguments**

**ds - Input data**  
dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response

variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### Data Types

single | double | logical

### X - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

### Data Types

single | double | logical

### y - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### Data Types

single | double

### modelspec - Model specification

string naming the model |  $t$ -by- $(p + 1)$  terms matrix | string of the form 'Y ~ terms'

Model specification, specified as one of the following. The choice is the starting model for `stepwiselm`.

- A string naming the model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p + 1)$  matrix, namely terms matrix, specifying terms to include in the model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus 1 is for the response variable.
- A string representing a formula in the form

'Y ~ terms',

where the terms are in Wilkinson Notation.

**Example:** 'quadratic'

**Example:** `'y ~ X1 + X2^2 + X1:X2'`

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

#### **Example:**

`'Intercept',false,'PredictorVars',[1,3],'ResponseVar',5,'RobustOpts'`, specifies a robust regression model with no constant term, where the algorithm uses the logistic weighting function with the default tuning constant, first and third variables are the predictor variables, and fifth variable is the response variable.

### **'CategoricalVars' - Categorical variables**

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.
- If data is in matrix `X`, then the default value of this name-value pair argument is an empty matrix `[]`. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** `'CategoricalVars',[2,3]`

**Example:** `'CategoricalVars',logical([0 1 1 0 0 0])`

### **Data Types**

single | double | logical

## **'Exclude' - Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

## **Data Types**

single | double | logical

## **'Intercept' - Indicator for constant term**

true (default) | false

Indicator for the constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

**Example:** 'Intercept',false

## **'PredictorVars' - Predictor variables**

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array ds, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the ds dataset array, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in X, or all variables in ds except for ResponseVar.



For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** 'PredictorVars',[2,3]

**Example:** 'PredictorVars',logical([0 1 1 0 0 0])

### Data Types

single | double | logical | cell

### 'ResponseVar' - Response variable

last column in dataset array `ds` (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a string of the variable name in the dataset array `ds`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a dataset array `ds`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

**Example:** 'ResponseVar','yield'

**Example:** 'ResponseVar',[4]

**Example:** 'ResponseVar',logical([0 0 0 1 0 0])

### Data Types

single | double | logical | char

### 'RobustOpts' - Indicator of robust fitting type

'off' (default) | 'on' | string | structure with string or function handle

Indicator of the robust fitting type to use, specified as the comma-separated pair consisting of 'RobustOpts' and one of the following.

- 'off' — No robust fitting. `fitlm` uses ordinary least squares.

- 'on' — Robust fitting. When you use robust fitting, 'bisquare' weight function is the default.
- String — Name of the robust fitting weight function from the following table. `fitlm` uses the corresponding default tuning constant in the table.
- Structure with the string `RobustWgtFun` containing the name of the robust fitting weight function from the following table and optional scalar `Tune` fields — `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. You can choose the name of the robust fitting weight function from this table. If you do not supply a `Tune` field, the fitting function uses the corresponding default tuning constant.

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'ols'	Ordinary least squares (no weighting function)	None
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985

The value  $r$  in the weight functions is

$$r = \text{resid} / (\text{tune} * s * \sqrt{1-h}),$$

where `resid` is the vector of residuals from the previous iteration, `h` is the vector of leverage values from a least-squares fit, and `s` is an estimate of the standard deviation of the error term given by

$$s = \text{MAD}/0.6745.$$

MAD is the median absolute deviation of the residuals from their median. The constant 0.6745 makes the estimate unbiased for the normal distribution. If there are  $p$  columns in  $X$ , the smallest  $p$  absolute deviations are excluded when computing the median.

Default tuning constants give coefficient estimates that are approximately 95% as statistically efficient as the ordinary least-squares estimates, provided the response has a normal distribution with no outliers. Decreasing the tuning constant increases the downweight assigned to large residuals; increasing the tuning constant decreases the downweight assigned to large residuals.

- Structure with the function handle `RobustWgtFun` and optional scalar `Tune` fields — You can specify a custom weight function. `fitlm` uses the `RobustWgtFun` weight function and `Tune` tuning constant from the structure. Specify `RobustWgtFun` as a function handle that accepts a vector of residuals, and returns a vector of weights the same size. The fitting function scales the residuals, dividing by the tuning constant (default 1) and by an estimate of the error standard deviation before it calls the weight function.

**Example:** `'RobustOpts', 'andrews'`

### 'VarNames' - Names of variables in fit

`{ 'x1', 'x2', ..., 'xn', 'y' }` (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings including the names for the columns of  $X$  first, and the name for the response variable  $y$  last.

`'VarNames'` is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

**Example:**

```
'VarNames',{ 'Horsepower','Acceleration','Model_Year','MPG' }
```

**Data Types**

cell

**'Weights' - Observation weights**

ones(n,1) (default) | *n*-by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an *n*-by-1 vector of nonnegative scalar values, where *n* is the number of observations.

**Data Types**

single | double

## Output Arguments

**mdl - Linear model**

LinearModel object

Linear model representing a least-squares fit of the response to the data, returned as a LinearModel object.

If the value of the 'RobustOpts' name-value pair is not [] or 'ols', the model is not a least-squares fit, but uses the robust fitting function.

For properties and methods of the linear model object, mdl, see the LinearModel class page.

## Examples

**Linear Regression with Categorical Predictor and Quadratic Term**

Fit a quadratic linear regression model to variables in a dataset array. The data includes continuous and categorical predictor variables.

Load the sample data.

```
load carsmall
```

Construct a dataset array containing continuous predictor variable `Weight` and response variable `MPG`. Add the nominal predictor variable `Year`.

```
ds = dataset(MPG,Weight);
ds.Year = nominal(Model_Year);
```

Create a fitted model of `MPG` as a function of `Year`, `Weight`, and `Weight2`. You don't have to include `Weight` explicitly in your formula because it is a lower-order term of `Weight2`. For details, see “Formula” on page 21-1035.

```
mdl = fitlm(ds,'MPG ~ Year + Weight^2')
```

```
mdl =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Year + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 2.78
```

```
R-squared: 0.885, Adjusted R-Squared 0.88
```

```
F-statistic vs. constant model: 172, p-value = 5.52e-41
```

`fitlm` creates two dummy (indicator) variables for the nominal variables, `Year`. The dummy variable `Year_76` takes the value 1 if the model year is 1976 and takes the value 0 if it is not. The dummy variable `Year_82` takes the value 1 if model year is 1982 and takes the value 0 if it is not. 1970 is the reference year (for details on dummy variables, see “Dummy Indicator Variables” on page 2-55). The fitted model is

$$\hat{MPG} = 54.206 - 0.0164Weight + 2.0887(Year\_76) + 8.1864(Year\_82) + 1.557e - 06Weight^2$$

## Simultaneously Specify the Variables and Use Formula

Simultaneously identify response and predictor variables and specify the model using a formula in linear regression.

Load sample data.

```
load hospital
```

Fit a linear model with interaction terms to the data. Indicate which variable is the response variable and identify the continuous and categorical predictors using name-value pair arguments.

```
mdl = fitlm(hospital, 'Weight~1+Age*Sex*Smoker-Age:Sex:Smoker', 'ResponseVariable');
```

```
mdl =
```

Linear regression model:

```
Weight ~ 1 + Sex*Age + Sex*Smoker + Age*Smoker
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	118.7	7.0718	16.785	6.821e-30
Sex_Male	68.336	9.7153	7.0339	3.3386e-10
Age	0.31068	0.18531	1.6765	0.096991
Smoker_1	3.0425	10.446	0.29127	0.77149
Sex_Male:Age	-0.49094	0.24764	-1.9825	0.050377
Sex_Male:Smoker_1	0.9509	3.8031	0.25003	0.80312
Age:Smoker_1	-0.07288	0.26275	-0.27737	0.78211

Number of observations: 100, Error degrees of freedom: 93

Root Mean Squared Error: 8.75

R-squared: 0.898, Adjusted R-Squared 0.892

F-statistic vs. constant model: 137, p-value = 6.91e-44

The  $t$ -statistics in `tStat` and corresponding  $p$ -values in `pValue` indicate that patient weights do not seem to differ significantly according to age, or the status of smoking, or the interaction of these factors with gender at the 5% significance level.

### Robust Linear Regression Model

Fit a linear regression model of the Hald data using robust fitting.

Load the data.

```
load hald
X = ingredients; % Predictor variables
y = heat; % Response
```

Fit a robust linear model to the data.

```
mdl = fitlm(X,y,'linear','RobustOpts','on')
```

```
mdl =
```

```
Linear regression model (robust fit):
```

```
y ~ 1 + x1 + x2 + x3 + x4
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	60.09	75.818	0.79256	0.4509
x1	1.5753	0.80585	1.9548	0.086346
x2	0.5322	0.78315	0.67957	0.51596
x3	0.13346	0.8166	0.16343	0.87424
x4	-0.12052	0.7672	-0.15709	0.87906

Number of observations: 13, Error degrees of freedom: 8

Root Mean Squared Error: 2.65

R-squared: 0.979, Adjusted R-Squared 0.969

F-statistic vs. constant model: 94.6, p-value = 9.03e-07

## Definitions

## Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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



```
0 0 0 1
```

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

T =

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

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

T =

```
0 0 0
1 0 0
0 1 0
1 1 0
0 2 0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

```
y ~ 1 + x1*x2 + x2^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
mdl = stepwiselm(X,MPG,[0 0 0], 'upper', T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

mdl =

Linear regression model:

$$y \sim 1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of fitlm in the previous step.

### Formula

A formula for model specification is a string of the form

`'Y ~ terms'`,

where

- Y is the response name.
- *terms* contains
  - Variable names
  - + means include the next variable
  - - means do not include the next variable
  - : defines an interaction, a product of terms
  - \* defines an interaction **and all lower-order terms**

- $\wedge$  raises the predictor to a power, exactly as in  $*$  repeated, so  $\wedge$  includes lower order terms as well
- $()$  groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include  $-1$  in the formula.

---

For example,

' $Y \sim A + B + C$ ' means a three-variable linear model with intercept.

' $Y \sim A + B + C - 1$ ' is a three-variable linear model without intercept.

' $Y \sim A + B + C + B^2$ ' is a three-variable model with intercept and a  $B^2$  term.

' $Y \sim A + B^2 + C$ ' is the same as the previous example because  $B^2$  includes a  $B$  term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of  $A$  with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$

Wilkinson Notation	Factors in Standard Notation
$A + B$	A, B
$A*B$	A, B, $A*B$
$A:B$	$A*B$ only
$-B$	Do not include B
$A*B + C$	A, B, C, $A*B$
$A + B + C + A:B$	A, B, C, $A*B$
$A*B*C - A:B:C$	A, B, C, $A*B$ , $A*C$ , $B*C$
$A*(B + C)$	A, B, C, $A*B$ , $A*C$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

## See Also

`LinearModel` | `predict` | `stepwiselm`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 9-21
- “Predict or Simulate Responses to New Data” on page 9-40
- “Linear Regression Workflow” on page 9-44
- “Regression with Categorical Covariates” on page 2-59

## Concepts

- “Linear Regression” on page 9-12

# fitlme

---

**Purpose** Fit linear mixed-effects model

**Syntax**  
`lme = fitlme(ds,formula)`  
`lme = fitlme(ds,formula,Name,Value)`

**Description** `lme = fitlme(ds,formula)` returns a linear mixed-effects model, specified by `formula`, fitted to the variables in the dataset array `ds`.

`lme = fitlme(ds,formula,Name,Value)` returns a linear mixed-effects model with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the covariance pattern of the random-effects terms, the method to use in estimating the parameters, or options for the optimization algorithm.

## Input Arguments

### **ds - Input data**

dataset array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a dataset array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-51). You must specify the model for the variables using `formula`.

### **Data Types**

single | double | char | cell

### **formula - Formula for model specification**

string of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`

Formula for model specification, specified as a string of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`. For a full description, see “Formula” on page 21-1059.

**Example:** `'y ~ treatment + (1|block)'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### Example:

```
'CovariancePattern', 'Diagonal', 'Optimizer', 'fminunc', 'OptimizerOptions'
```

specifies a model, where the random-effects terms have a diagonal covariance matrix structure, and `fitlme` uses the `fminunc` optimization algorithm with the custom optimization parameters defined in variable `opt`.

### 'CovariancePattern' - Pattern of covariance matrix

```
'FullCholesky' (default) | string | square symmetric logical matrix |  
cell array of strings or logical matrices
```

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of `'CovariancePattern'` and a string, a square symmetric logical matrix, or a cell array of strings or logical matrices.

If there are  $R$  random-effects terms, then the value of `'CovariancePattern'` must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

'FullCholesky'

Default. Full covariance matrix using the Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Full'

Full covariance matrix, using the log-Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Diagonal'

Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$



'CompSymm'

where  $\sigma^2_{b_1}$  is the common variance of the random-effects terms.

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma^2_{b_1} & \sigma_{b_1,b_2} & \sigma_{b_1,b_2} \\ \sigma_{b_1,b_2} & \sigma^2_{b_1} & \sigma_{b_1,b_2} \\ \sigma_{b_1,b_2} & \sigma_{b_1,b_2} & \sigma^2_{b_1} \end{pmatrix}$$

where  $\sigma^2_{b_1}$  is the common variance of the random-effects terms and  $\sigma_{b_1,b_2}$  is the common covariance between any two random-effects term .

PAT

Square symmetric logical matrix. If 'CovariancePattern' is defined by the matrix PAT, and if  $\text{PAT}(a,b) = \text{false}$ , then the (a,b) element of the corresponding covariance matrix is constrained to be 0.

**Example:** 'CovariancePattern', 'Diagonal'

**Example:** 'CovariancePattern', {'Full', 'Diagonal'}

### 'FitMethod' - Method for estimating parameters

'ML' (default) | 'REML'

Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of 'FitMethod' and either of the following.

'ML'	Default. Maximum likelihood estimation
'REML'	Restricted maximum likelihood estimation

**Example:** 'FitMethod','REML'

### **'Weights' - Observation weights**

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$ , where  $n$  is the number of observations.

#### **Data Types**

single | double

### **'Exclude' - Indices for rows to exclude**

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

**Example:** 'Exclude',[13,67]

#### **Data Types**

single | double | logical

### **'DummyVarCoding' - Coding to use for dummy variables**

'reference' (default) | 'effects' | 'full'

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

**Example:** 'DummyVarCoding','effects'

### 'Optimizer' - Optimization algorithm

'quasinevton' (default) | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.

'quasinevton'	Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using <code>statset('LinearMixedModel')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>statset('LinearMixedModel')</code> .
'fminunc'	You must have Optimization Toolbox to specify this option. Change the options of the algorithm using <code>optimoptions('fminunc')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>optimoptions('fminunc')</code>

with 'Algorithm' set to  
'quasi-newton'.

**Example:** 'Optimizer','fminunc'

## **'OptimizerOptions' - Options for optimization algorithm**

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of 'OptimizerOptions' and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If 'Optimizer' is 'fminunc', then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options 'fminunc' uses. If 'Optimizer' is 'fminunc' and you do not supply 'OptimizerOptions', then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with 'Algorithm' set to 'quasi-newton'.
- If 'Optimizer' is 'quasinewton', then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The 'quasinewton' optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

## **'TolFun' - Relative tolerance on gradient of objective function**

1e-6 (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

## **'TolX' - Absolute tolerance on step size**

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

**'MaxIter' - Maximum number of iterations allowed**

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

**'Display' - Level of display**

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

**'StartMethod' - Method to start iterative optimization**

'default' (default) | 'random'

Method to start iterative optimization, specified as the comma-separated pair consisting of 'StartMethod' and either of the following.

'default'	Default. An internally defined default value.
-----------	---

'random'	A random initial value.
----------	-------------------------

**Example:** 'StartMethod', 'random'

**'Verbose' - Indicator to display optimization process on screen**

false (default) | true

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of 'Verbose' and either false or true. Default is false.

The setting for 'Verbose' overrides the field 'Display' in 'OptimizerOptions'.

**Example:** 'Verbose', true

**'CheckHessian' - Indicator to check positive definiteness of Hessian**

false (default) | true

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of 'CheckHessian' and either `false` or `true`. Default is `false`.

Specify 'CheckHessian' as `true` to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

**Example:** 'CheckHessian',true

## Output Arguments

### lme - Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

For properties and methods of this object, see LinearMixedModel.

## Examples

### Random Intercept Model

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google searches, plus a nationwide estimate from the Centers for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses, combine the nine columns corresponding to the regions into a tall array. The new dataset array, `flu2`, must have the response variable `FluRate`, the nominal variable `Region` that shows which region each estimate is from, the nationwide estimate `WtdILI`, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...  
            'IndVarName','Region');  
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with a fixed-effects term for the nationwide estimate, `WtdILI`, and a random intercept that varies by `Date`. The model corresponds to

$$y_{im} = \beta_0 + \beta_1 \text{WtdILI}_i + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable `Date`.  $b_{0m}$  is the random effect for level  $m$  of the grouping variable `Date` and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b \sim N(0, \sigma_b^2)$  and the error term has the distribution,  $\varepsilon \sim N(0, \sigma^2)$ .

```
lme = fitlme(flu2, 'FluRate ~ 1 + WtdILI + (1|Date)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

```
Formula:
```

```
FluRate ~ 1 + WtdILI + (1 | Date)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
286.24	302.83	-139.12	278.24

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	0.16385	0.057525	2.8484	466	0.0045
'WtdILI'	0.7236	0.032219	22.459	466	3.05e-45

```
Random effects covariance parameters (95% CIs):
```

```

Group: Date (52 Levels)
      Name1              Name2              Type      Estimate      Lower      Upper
      '(Intercept)'      '(Intercept)'      'std'      0.17146      0.13227    0.22226
Group: Error
      Name              Estimate      Lower      Upper
      'Res Std'        0.30201      0.28217    0.32324

```

The confidence limits for the standard deviation of the random-effects term,  $\sigma_b^2$ , do not include 0 (0.13227, 0.22226), which indicates that the random-effects term is significant. You can also test the significance of the random-effects terms using the `compare` method.

The estimated value of an observation is the sum of the fixed effects and the random-effect value at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28 is

$$\begin{aligned}
 \hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 WtdILI_{28} + \hat{b}_{10/30/2005} \\
 &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\
 &= 1.46749,
 \end{aligned}$$

where  $\hat{b}$  is the best linear unbiased predictor (BLUP) of the random effects for the intercept. You can compute this value as follows.

```

beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level=='10/30/2005');

y_hat =

    1.4674

```

You can display the fitted value using the `fitted` method.

```

F = fitted(lme);
F(28)

```



```
ans =
    1.4674
```

## Randomized Block Design

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The data shows the absolute deviations from the target quality characteristic measured from the products each of five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the absolute deviations of the quality characteristics from the target value. This is simulated data.

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift. Use the restricted maximum likelihood method and 'effects' contrasts.

'effects' contrasts mean that the coefficients sum to 0, and fitlme creates a matrix called a “fixed effects design matrix” to describe the effect of shift. This matrix has two columns, *Shift\_Evening* and *Shift\_Morning*, where

$$Shift\_Evening = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases} \quad \text{and} \quad Shift\_Morning = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases} .$$

The model corresponds to

Morning Shift:  $QCDev_{im} = \beta_0 + \beta_2 Shift - Morning_i + b_{0m} + \varepsilon_{im}$ ,  $m = 1, 2, \dots, 5$ ,  
 Evening Shift:  $QCDev_{im} = \beta_0 + \beta_1 Shift - Evening_i + b_{0m} + \varepsilon_{im}$ ,  
 Night Shift:  $QCDev_{im} = \beta_0 - \beta_1 Shift - Evening_i - \beta_2 Shift - Morning_i + b_{0m} + \varepsilon_{im}$ ,

where  $b \sim N(0, \sigma^2_b)$  and  $\varepsilon \sim N(0, \sigma^2)$ .

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)', ...
'FitMethod', 'REML', 'DummyVarCoding', 'effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
QCDev ~ 1 + Shift + (1 | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.6525	0.94109	3.8812	12	0.0021
'Shift_Evening'	-0.53293	0.31206	-1.7078	12	0.11
'Shift_Morning'	-0.91973	0.31206	-2.9473	12	0.012

```
Random effects covariance parameters (95% CIs):
```

```
Group: Operator (5 Levels)
```

Name1	Name2	Type	Estimate	LC
'(Intercept)'	'(Intercept)'	'std'	2.0457	0.

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.85462	0.52357	1.395

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

```
B = randomEffects(lme)
```

```
B =
```

```

0.5775
1.1757
-2.1715
2.3655
-1.9472

```

The estimated absolute deviation from the target quality characteristics for the third operator working the evening shift is

$$\begin{aligned}
 \hat{y}_{Evening, Operator3} &= \hat{\beta}_0 + \hat{\beta}_1 Shift\_Evening + \hat{b}_{03} \\
 &= 3.6525 - 0.53293 - 2.1715 \\
 &= 0.94807.
 \end{aligned}$$

You can also display this value as follows.

```
F = fitted(lme);
F(shift.Shift=='Evening' & shift.Operator=='3')
```

```
ans =
```

```
0.9481
```

Similarly, you can calculate the estimated absolute deviation from the target quality characteristics for the third operator working the morning shift as

$$\begin{aligned}\hat{y}_{Morning,Operator3} &= \hat{\beta}_0 + \hat{\beta}_2 Shift\_Morning + \hat{b}_{03} \\ &= 3.6525 - 0.91973 - 2.1715 \\ &= 0.56127.\end{aligned}$$

You can also display this value as follows.

```
F(shift.Shift=='Morning' & shift.Operator=='3')
```

```
ans =
```

```
0.5613
```

The operator tends to make a smaller magnitude of error during the morning shift.

## Split-Plot Experiment

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
```

```
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type) and the plots within blocks (tomato types within soil types) independently.

This model corresponds to

$$y_{imjk} = \beta_0 + \sum_{m=2}^4 \beta_{1m} I[F]_{im} + \sum_{j=2}^5 \beta_{2j} I[T]_{ij} + \sum_{j=2}^5 \sum_{m=2}^4 \beta_{3mj} I[F]_{im} I[T]_{ij} + b_{0k} S_k + b_{0jk} (S^* T)_{jk} + \varepsilon_{imjk},$$

where  $i = 1, 2, \dots, 60$ , index  $m$  corresponds to the fertilizer types,  $j$  corresponds to the tomato types, and  $k = 1, 2, 3$  corresponds to the blocks (soil).  $S_k$  represents the  $k$ th soil type, and  $(S^* T)_{jk}$  represents the  $j$ th tomato type nested in the  $k$ th soil type.  $I[F]_{im}$  is the dummy variable representing level  $m$  of the fertilizer. Similarly,  $I[T]_{ij}$  is the dummy variable representing level  $j$  of the tomato type.

The random effects and observation error have these prior distributions:  $b_{0k} \sim N(0, \sigma_S^2)$ ,  $b_{0jk} \sim N(0, \sigma_{S^*T}^2)$ , and  $\varepsilon_{imjk} \sim N(0, \sigma^2)$ .

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	60
Fixed effects coefficients	20
Random effects coefficients	18
Covariance parameters	3

Formula:

Yield ~ 1 + Tomato\*Fertilizer + (1 | Soil) + (1 | Soil:Tomato)

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
522.57	570.74	-238.29	476.57

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat
'(Intercept)'	77	8.5836	8.9706
'Tomato_Grape'	-16	11.966	-1.3371
'Tomato_Heirloom'	-6.6667	11.966	-0.55714
'Tomato_Plum'	32.333	11.966	2.7022
'Tomato_Vine'	-13	11.966	-1.0864
'Fertilizer_2'	34.667	8.572	4.0442
'Fertilizer_3'	33.667	8.572	3.9275
'Fertilizer_4'	47.667	8.572	5.5607
'Tomato_Grape:Fertilizer_2'	-2.6667	12.123	-0.21997
'Tomato_Heirloom:Fertilizer_2'	-8	12.123	-0.65992
'Tomato_Plum:Fertilizer_2'	-15	12.123	-1.2374
'Tomato_Vine:Fertilizer_2'	-16	12.123	-1.3198
'Tomato_Grape:Fertilizer_3'	16.667	12.123	1.3748
'Tomato_Heirloom:Fertilizer_3'	3.3333	12.123	0.27497
'Tomato_Plum:Fertilizer_3'	3.6667	12.123	0.30246
'Tomato_Vine:Fertilizer_3'	3	12.123	0.24747
'Tomato_Grape:Fertilizer_4'	13.333	12.123	1.0999
'Tomato_Heirloom:Fertilizer_4'	-19	12.123	-1.5673
'Tomato_Plum:Fertilizer_4'	-2.6667	12.123	-0.21997
'Tomato_Vine:Fertilizer_4'	8.6667	12.123	0.71492

Random effects covariance parameters (95% CIs):

Group: Soil (3 Levels)

Name1	Name2	Type	Estimate	Lo	Hi
'(Intercept)'	'(Intercept)'	'std'	2.5028	0.	0.

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate	Lo	Hi
'(Intercept)'	'(Intercept)'	'std'	10.225	6.	6.

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	10.499	8.5389	12.908

The  $p$ -values corresponding to the last 12 rows in the fixed-effects coefficients display (0.82701 to 0.47881) indicate that interaction coefficients between the tomato and fertilizer types are not significant. To test for the overall interaction between tomato and fertilizer, use the anova method after refitting the model using 'effects' contrasts.

The confidence interval for the standard deviations of the random-effects terms ( $\sigma^2_{\epsilon}$ ), where the intercept is grouped by soil, is very large. This term does not appear significant.

Refit the model after removing the interaction term `Tomato:Fertilizer` and the random-effects term (`1 | Soil`).

```
lme = fitlme(ds, 'Yield ~ Fertilizer + Tomato + (1|Soil:Tomato)')
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	60
Fixed effects coefficients	8
Random effects coefficients	15
Covariance parameters	2

Formula:

```
Yield ~ 1 + Tomato + Fertilizer + (1 | Soil:Tomato)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
511.06	532	-245.53	491.06

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pVal
'(Intercept)'	77.733	7.3293	10.606	52	1.31
'Tomato_Grape'	-9.1667	9.6045	-0.95441	52	0
'Tomato_Heirloom'	-12.583	9.6045	-1.3102	52	0.0
'Tomato_Plum'	28.833	9.6045	3.0021	52	0.0
'Tomato_Vine'	-14.083	9.6045	-1.4663	52	0
'Fertilizer_2'	26.333	4.5004	5.8514	52	3.30
'Fertilizer_3'	39	4.5004	8.6659	52	1.14
'Fertilizer_4'	47.733	4.5004	10.607	52	1.3

Random effects covariance parameters (95% CIs):

Group: Soil:Tomato (15 Levels)

Name1	Name2	Type	Estimate	Lo
'(Intercept)'	'(Intercept)'	'std'	10.02	6.

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	12.325	10.024	15.153

You can compare the two models using the `compare` method with the simulated likelihood ratio test since both a fixed-effect and a random-effect term are tested.

## Longitudinal Study with a Covariate

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs (A, B, C, D), and their weight loss is recorded over six 2-week time periods. This is simulated data.



Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);
ds.Subject = nominal(ds.Subject);
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

`fitlme` uses program A as a reference and creates the necessary dummy variables  $I[\cdot]$ . Since the model already has an intercept, `fitlme` only creates dummy variables for programs B, C, and D. This is also known as the 'reference' method of coding dummy variables. This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 Week_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i + \beta_5 I[PD]_i \\ + \beta_6 (Week_i * I[PB]_i) + \beta_7 (Week_i * I[PC]_i) + \beta_8 (Week_i * I[PD]_i) \\ + b_{0m} + b_{1m} Week_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{1m}$  and  $b_{0m}$  are random effects.  $IW$  stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing program B. The random effects and observation error have these prior distributions:  $b_{0m} \sim N(0, \sigma^2_0)$ ,  $b_{1m} \sim N(0, \sigma^2_1)$ , and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
lme = fitlme(ds, 'y ~ InitialWeight + Program*Week + (Week|Subject)')
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

```
Number of observations          120
```

```

Fixed effects coefficients          9
Random effects coefficients        40
Covariance parameters              4

```

Formula:

```
y ~ 1 + InitialWeight + Program*Week + (1 + Week | Subject)
```

Model fit statistics:

```

AIC          BIC          LogLikelihood    Deviance
-22.981      13.257      24.49          -48.981

```

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	0.66105	0.25892	2.5531	111
'InitialWeight'	0.0031879	0.0013814	2.3078	111
'Program_B'	0.36079	0.13139	2.746	111
'Program_C'	-0.033263	0.13117	-0.25358	111
'Program_D'	0.11317	0.13132	0.86175	111
'Week'	0.1732	0.067454	2.5677	111
'Program_B:Week'	0.038771	0.095394	0.40644	111
'Program_C:Week'	0.030543	0.095394	0.32018	111
'Program_D:Week'	0.033114	0.095394	0.34713	111

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.18407
'Week'	'(Intercept)'	'corr'	0.66841
'Week'	'Week'	'std'	0.15033

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The  $p$ -values 0.022863 and 0.011567 indicate significant effects of subject initial weights and time in the amount of weight lost. The weight loss of subjects who are in program B is significantly different

relative to the weight loss of subjects who are in program A. The lower and upper limits of the covariance parameters for the random effects do not include 0, thus they are significant. You can also test the significance of the random effects using the `compare` method.

## Definitions

### Formula

In general, a formula for model specification is a string of the form `'y ~ terms'`. For the linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` and `random` contain the fixed-effects and the random-effects terms. Suppose a dataset array `ds` contains the following:

- A response variable, `y`
- Predictor variables, `Xj`, which can be continuous or grouping variables
- Grouping variables, `g1`, `g2`, ..., `gR`,

where the grouping variables in `Xj` and `gr` can be categorical, logical, character arrays, or cell arrays of strings.

Then, in a formula of the form, `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`, the term `fixed` corresponds to a specification of the fixed-effects design matrix  $X$ , `random1` is a specification of the random-effects design matrix  $Z_1$  corresponding to grouping variable `g1`, and similarly `randomR` is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable `gR`. You can express the `fixed` and `random` terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X$ , $X^2$ , ..., $X^k$

Wilkinson Notation	Factors in Standard Notation
$X1 + X2$	X1, X2
$X1 * X2$	X1, X2, $X1.*X2$ (elementwise multiplication of X1 and X2)
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include X2
$X1 * X2 + X3$	X1, X2, X3, $X1 * X2$
$X1 + X2 + X3 + X1 : X2$	X1, X2, X3, $X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	X1, X2, X3, $X1 * X2$ , $X1 * X3$ , $X2 * X3$
$X1 * (X2 + X3)$	X1, X2, X3, $X1 * X2$ , $X1 * X3$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using -1. Here are some examples for linear mixed-effects model specification.

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, X1 and X2. This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	Fixed effects for X1 and X2. The implicit intercept term is suppressed by including -1.
'y ~ 1 + (1   g1)'	Intercept plus random effect for each level of the grouping variable g1.
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1)'	Random intercept and slope, with possible correlation between them.

Formula	Description
'y ~ X1 + (1   g1) + (-1 + X1   g1)'	Independent random intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for g1 and g2, plus an independent interaction effect.

### Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.

$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q^*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ . For example, if

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$

## Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## Alternatives

If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X,y,Z,G)`.

## References

[1] Pinheiro, J. C., and D. M. Bates. “Unconstrained Parametrizations for Variance-Covariance Matrices”. *Statistics and Computing*, Vol. 6, 1996, pp. 289–296.

## See Also

`LinearMixedModel` | `fitlmematrix`

# fitlmematrix

---

**Purpose** Fit linear mixed-effects model

**Syntax**

```
lme = fitlmematrix(X,y,Z,[])  
lme = fitlmematrix(X,y,Z,G)  
lme = fitlmematrix(__,Name,Value)
```

**Description** `lme = fitlmematrix(X,y,Z,[])` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix or matrices in `Z`.

`[]` implies that there is one group. That is, the grouping variable `G` is `ones(n,1)`, where  $n$  is the number of observations. Using `fitlmematrix(X,Y,Z,[])` without a specified covariance pattern most likely results in a nonidentifiable model. This syntax is recommended only if you build the grouping information into the random effects design `Z` and specify a covariance pattern for the random effects using the 'CovariancePattern' name-value pair argument.

`lme = fitlmematrix(X,y,Z,G)` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix `Z` or matrices in `Z`, and the grouping variable or variables in `G`.

`lme = fitlmematrix(__,Name,Value)` also creates a linear mixed-effects model with additional options specified by one or more `Name,Value` pair arguments, using any of the previous input arguments.

For example, you can specify the names of the response, predictor, and grouping variables. You can also specify the covariance pattern, fitting method, or the optimization algorithm.

## Input Arguments

### **X - Fixed-effects design matrix**

*n*-by-*p* matrix

Fixed-effects design matrix, specified as an *n*-by-*p* matrix, where  $n$  is the number of observations, and  $p$  is the number of fixed-effects



predictor variables. Each row of  $X$  corresponds to one observation, and each column of  $X$  corresponds to one variable.

**Data Types**

single | double

**y - Response values**

$n$ -by-1 vector

Response values, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations.

**Data Types**

single | double

**Z - Random-effects design**

$n$ -by- $q$  matrix | cell array of  $R$   $n$ -by- $q(r)$  matrices,  $r = 1, 2, \dots, R$

Random-effects design, specified as either of the following.

- If there is one random-effects term in the model, then  $Z$  must be an  $n$ -by- $q$  matrix, where  $n$  is the number of observations and  $q$  is the number of variables in the random-effects term.
- If there are  $R$  random-effects terms, then  $Z$  must be a cell array of length  $R$ . Each cell of  $Z$  contains an  $n$ -by- $q(r)$  design matrix  $Z\{r\}$ ,  $r = 1, 2, \dots, R$ , corresponding to each random-effects term. Here,  $q(r)$  is the number of random effects term in the  $r$ th random effects design matrix,  $Z\{r\}$ .

**Data Types**

single | double | cell

**G - Grouping variable or variables**

$n$ -by-1 vector | cell array of  $R$   $n$ -by-1 vectors

Grouping variable or variables, specified as either of the following.

- If there is one random-effects term, then  $G$  must be an  $n$ -by-1 vector corresponding to a single grouping variable with  $M$  levels or groups.

G can be a categorical vector, numeric vector, character array, or cell array of strings.

- If there are multiple random-effects terms, then G must be a cell array of length  $R$ . Each cell of G contains a grouping variable  $G\{r\}$ ,  $r = 1, 2, \dots, R$ , with  $M(r)$  levels.

$G\{r\}$  can be a categorical vector, numeric vector, character array, or cell array of strings.

## Data Types

single | double | char | cell

## Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

### Example:

`'CovariancePattern', 'Diagonal', 'DummyVarCoding', 'full', 'Optimizer', 'fminunc` specifies a random-effects covariance pattern with zero off-diagonal elements, creates a dummy variable for each level of a categorical variable, and uses the `fminunc` optimization algorithm.

## 'FixedEffectPredictors' - Names of columns in fixed-effects design matrix

`{'x1', 'x2', ..., 'xP'}` (default) | cell array of length  $p$

Names of columns in the fixed-effects design matrix  $X$ , specified as the comma-separated pair consisting of `'FixedEffectPredictors'` and a cell array of length  $p$ .

For example, if you have a constant term and two predictors, say `TimeSpent` and `Gender`, where `Female` is the reference level for `Gender`, as the fixed effects, then you can specify the names of your fixed effects in the following way. `Gender_Male` represents the dummy variable you must create for category `Male`. You can choose different names for these variables.

**Example:**

```
'FixedEffectPredictors',{ 'Intercept', 'TimeSpent', 'Gender_Male' },
```

**Data Types**

cell

**'RandomEffectPredictors' - Names of columns in random-effects design matrix or cell array**

cell array of length  $q$  | cell array of length  $R$  with elements of length  $q(r)$ ,  $r = 1, 2, \dots, R$

Names of columns in the random-effects design matrix or cell array  $Z$ , specified as the comma-separated pair consisting of 'RandomEffectPredictors' and either of the following:

- A cell array of length  $q$  when  $Z$  is an  $n$ -by- $q$  design matrix. In this case, the default is {'z1', 'z2', ..., 'zQ'}.
- A cell array of length  $R$ , when  $Z$  is a cell array of length  $R$  with each element  $Z\{r\}$  of length  $q(r)$ ,  $r = 1, 2, \dots, R$ . In this case, the default is {'z11', 'z12', ..., 'z1Q(1)' }, ..., {'zr1', 'zr2', ..., 'zrQ(r)' }.

For example, suppose you have correlated random effects for intercept and a variable named Acceleration. Then, you can specify the random-effects predictor names as follows.

**Example:**

```
'RandomEffectPredictors',{ 'Intercept', 'Acceleration' }
```

If you have two random effects terms, one for the intercept and the variable Acceleration grouped by variable g1, and the second for the intercept, grouped by the variable g2, then you specify the random-effects predictor names as follows.

**Example:**

```
'RandomEffectPredictors',{ {'Intercept', 'Acceleration'}, {'Intercept'} }
```

**Data Types**

cell

**'ResponseVarName' - Name of response variable**

'y' (default) | string

Name of response variable, specified as the comma-separated pair consisting of 'ResponseVarName' and a string.

For example, if your response variable name is `score`, then you can specify it as follows.

**Example:** 'ResponseVarName', 'score'

## Data Types

char

## 'RandomEffectGroups' - Names of random effects grouping variables

'g' or {'g1', 'g2', ..., 'gR'} (default) | string | cell array of strings

Names of random effects grouping variables, specified as the comma-separated pair 'RandomEffectGroups' and either of the following:

- String — If there is only one random-effects term, that is, if  $G$  is a vector, then the value of 'RandomEffectGroups' is a string containing the name for the grouping variable  $G$ . The default is 'g'.
- Cell array of strings — If there are multiple random-effects terms, that is, if  $G$  is a cell array of length  $R$ , then the value of 'RandomEffectGroups' is a cell array of length  $R$ , where each cell contains the name for the grouping variable  $G\{r\}$ . The default is {'g1', 'g2', ..., 'gR'}.

For example, if you have two random-effects terms, `z1` and `z2`, grouped by the grouping variables `sex` and `subject`, then you can specify the names of your grouping variables as follows.

**Example:** 'RandomEffectGroups', {'sex', 'subject'}

## Data Types

char | cell

## 'CovariancePattern' - Pattern of covariance matrix

'FullCholesky' (default) | string | square symmetric logical matrix | cell array of strings or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of 'CovariancePattern' and a string, a square symmetric logical matrix, or a cell array of strings or logical matrices.

If there are  $R$  random-effects terms, then the value of 'CovariancePattern' must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

- 'FullCholesky' Default. Full covariance matrix using the Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.
- 'Full' Full covariance matrix, using the log-Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.
- 'Diagonal' Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$

where  $\sigma_b^2$  is the common variance of the random-effects terms.

'CompSymm'

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$$

where  $\sigma_{b1}^2$  is the common variance of the random-effects terms and  $\sigma_{b1,b2}$  is the common covariance

between any two random-effects term .

PAT

Square symmetric logical matrix. If 'CovariancePattern' is defined by the matrix PAT, and if  $PAT(a,b) = \text{false}$ , then the (a,b) element of the corresponding covariance matrix is constrained to be 0.

**Example:** 'CovariancePattern', 'Diagonal'

**Example:** 'CovariancePattern', {'Full', 'Diagonal'}

## 'FitMethod' - Method for estimating parameters

'ML' (default) | 'REML'

Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of 'FitMethod' and either of the following.

'ML'

Default. Maximum likelihood estimation

'REML'

Restricted maximum likelihood estimation

**Example:** 'FitMethod', 'REML'

## 'Weights' - Observation weights

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$ , where  $n$  is the number of observations.

### Data Types

single | double

## **'Exclude' - Indices for rows to exclude**

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

**Example:** 'Exclude',[13,67]

## **Data Types**

single | double | logical

## **'DummyVarCoding' - Coding to use for dummy variables**

'reference' (default) | 'effects' | 'full'

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

**Example:** 'DummyVarCoding','effects'

## **'Optimizer' - Optimization algorithm**

'quasinevton' (default) | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.



'quasinewton'	Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using <code>statset('LinearMixedModel')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>statset('LinearMixedModel')</code> .
'fminunc'	You must have Optimization Toolbox to specify this option. Change the options of the algorithm using <code>optimoptions('fminunc')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>optimoptions('fminunc')</code> with 'Algorithm' set to 'quasi-newton'.

**Example:** 'Optimizer','fminunc'

### 'OptimizerOptions' - Options for optimization algorithm

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of 'OptimizerOptions' and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If 'Optimizer' is 'fminunc', then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options 'fminunc' uses. If 'Optimizer' is 'fminunc' and you do not supply 'OptimizerOptions', then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with 'Algorithm' set to 'quasi-newton'.

- If 'Optimizer' is 'quasinewton', then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The 'quasinewton' optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

**'TolFun' - Relative tolerance on gradient of objective function**

1e-6 (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

**'TolX' - Absolute tolerance on step size**

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

**'MaxIter' - Maximum number of iterations allowed**

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

**'Display' - Level of display**

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

**'StartMethod' - Method to start iterative optimization**

'default' (default) | 'random'

Method to start iterative optimization, specified as the comma-separated pair consisting of 'StartMethod' and either of the following.

'default'	Default. An internally defined default value.
'random'	A random initial value.

**Example:** 'StartMethod', 'random'

### 'Verbose' - Indicator to display optimization process on screen

false (default) | true

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of 'Verbose' and either false or true. Default is false.

The setting for 'Verbose' overrides the field 'Display' in 'OptimizerOptions'.

**Example:** 'Verbose', true

### 'CheckHessian' - Indicator to check positive definiteness of Hessian

false (default) | true

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of 'CheckHessian' and either false or true. Default is false.

Specify 'CheckHessian' as true to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

**Example:** 'CheckHessian', true

## Output Arguments

### lme - Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

For properties and methods of this object, see LinearMixedModel.

## Examples      No Grouping Variable Specified

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, where miles per gallon (MPG) is the response, weight is the predictor variable, and the intercept varies by model year. First, define the design matrices. Then, fit the model using the specified design matrices.

```
y = MPG;  
X = [ones(size(Weight)), Weight];  
Z = ones(size(y));  
lme = fitlmematrix(X,y,Z,Model_Year)
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

```
Formula:
```

```
y ~ x1 + x2 + (z11 | g1)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-33
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

Random effects covariance parameters (95% CIs):

Group: g1 (3 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.542

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

Now, fit the same model by building the grouping into the Z matrix.

```
Z = double([Model_Year==70, Model_Year==76, Model_Year==82]);
lme = fitlmematrix(X,y,Z,[],'Covariancepattern','Isotropic')
```

lme =

Linear mixed-effects model fit by ML

Model information:

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

Formula:

$$y \sim x1 + x2 + (z11 + z12 + z13 | g1)$$

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-33
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

Random effects covariance parameters (95% CIs):

Group: g1 (1 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.5421

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

## Longitudinal Study with a Covariate

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs (A, B, C, D) and their weight loss is recorded over six 2-week time periods. This is simulated data.

Define `Subject` and `Program` as categorical variables. Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, week, and the interaction between the week and type of program as the fixed effects. The intercept and coefficient of week vary by subject.

This model corresponds to

$$\begin{aligned} y_{im} = & \beta_0 + \beta_1 IW_i + \beta_2 Week_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i + \beta_5 I[PD]_i \\ & + \beta_6 (Week_i * I[PB]_i) + \beta_7 (Week_i * I[PC]_i) + \beta_8 (Week_i * I[PD]_i) \\ & + b_{0m} + b_{1m} Week_{im} + \varepsilon_{im}, \end{aligned}$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[\cdot]$  is a dummy variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing program type B. The random effects and observation error have these prior distributions:  $b_{0m} \sim N(0, \sigma_0^2)$ ,  $b_{1m} \sim N(0, \sigma_1^2)$ , and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
Subject = nominal(Subject);
Program = nominal(Program);
D = dummyvar(Program); % Create dummy variables for Program
X = [ones(120,1), InitialWeight, D(:,2:4), Week, ...
     D(:,2).*Week, D(:,3).*Week, D(:,4).*Week];
Z = [ones(120,1), Week];
G = Subject;
```

Since the model has an intercept, you only need the dummy variables for programs B, C, and D. This is also known as the 'reference' method of coding dummy variables.

Fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,y,Z,G,'FixedEffectPredictors',...
{'Intercept','InitWeight','PrgB','PrgC','PrgD','Week','Week_PrgB','Week_PrgC',...
'RandomEffectPredictors',{'Intercept','Week'}},'RandomEffectGroups',G);

lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

Formula:

# fitlmematrix

Linear Mixed Formula with 10 predictors.

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pVal
'Intercept'	0.66105	0.25892	2.5531	111	0.0115
'InitWeight'	0.0031879	0.0013814	2.3078	111	0.0228
'PrgB'	0.36079	0.13139	2.746	111	0.0072
'PrgC'	-0.033263	0.13117	-0.25358	111	0.8000
'PrgD'	0.11317	0.13132	0.86175	111	0.3880
'Week'	0.1732	0.067454	2.5677	111	0.0115
'Week_PrgB'	0.038771	0.095394	0.40644	111	0.6870
'Week_PrgC'	0.030543	0.095394	0.32018	111	0.7470
'Week_PrgD'	0.033114	0.095394	0.34713	111	0.7310

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.18407	0.1220
'Week'	'Intercept'	'corr'	0.66841	0.2100
'Week'	'Week'	'std'	0.15033	0.1100

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The  $p$ -values 0.0228 and 0.0115 indicate significant effects of the initial weights of the subjects and the time factor in the amount of weight lost. The weight loss of subjects who are in program B is significantly different relative to the weight loss of subjects who are in program A. The lower and upper limits of the covariance parameters for the random effects do not include zero, thus they seem significant. You can also test the significance of the random-effects using the compare method.



## Random Intercept Model

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables for estimated influenza rates (in 9 different regions, estimated from Google searches, plus a nationwide estimate from the Centers for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, where the influenza rates are the responses, combine the nine columns corresponding to the regions into a tall array that has a single response variable, `FluRate`, and a nominal variable, `Region`, the nationwide estimate `WtdILI`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...
            'IndVarName','Region');
flu2.Date = nominal(flu2.Date);
```

Define the design matrices for a random-intercept linear mixed-effects model, where the intercept varies by `Date`. The corresponding model is

$$y_{im} = \beta_0 + \beta_1 WtdILI_i + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable `Date`.  $b_{0m}$  is the random effect for level  $m$  of the grouping variable `Date` and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b_{0m} \sim N(0, \sigma_{FR}^2)$  and the error term has the distribution,  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
y = flu2.FluRate;
X = [ones(468,1) flu2.WtdILI];
Z = [ones(468,1)];
G = flu2.Date;
```

Fit the linear mixed-effects model.

# fitlmematrix

```
lme = fitlmematrix(X,y,Z,G,'FixedEffectPredictors',{'Intercept','NationalRate'},  
'RandomEffectPredictors',{'Intercept'},'RandomEffectGroups',{'Date'})
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

```
Formula:
```

```
y ~ Intercept + NationalRate + (Intercept | Date)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
286.24	302.83	-139.12	278.24

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.16385	0.057525	2.8484	466	0.0046
'NationalRate'	0.7236	0.032219	22.459	466	3.0502e-45

```
Random effects covariance parameters (95% CIs):
```

```
Group: Date (52 Levels)
```

Name1	Name2	Type	Estimate	Lower	Upper
'Intercept'	'Intercept'	'std'	0.17146	0.13227	0.22226

```
Group: Error
```

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

The confidence limits of the standard deviation of the random-effects term  $\sigma_{b_{0m}}^2$ , do not include zero (0.13227, 0.22226), which indicates that

the random-effects term is significant. You can also test the significance of the random-effects using compare method.

The estimated value of an observation is the sum of the fixed-effects values and value of the random effect at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28

$$\begin{aligned}\hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 WtdILLI_{28} + \hat{b}_{10/30/2005} \\ &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\ &= 1.46749,\end{aligned}$$

where  $\hat{b}$  is the best linear unbiased predictor (BLUP) of the random effects for the intercept. You can compute this value as follows.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % compute the random effects statistics
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level);

y_hat =

    1.4674
```

You can simply display the fitted value using the fitted(lme) method.

```
F = fitted(lme);
F(28)

ans =

    1.4674
```

## Randomized Block Design

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviations of the quality characteristics from the target value. This is simulated data.

Define the design matrices for a linear mixed-effects model with a random intercept grouped by operator, and shift as the fixed effects. Use the 'effects' contrasts. 'effects' contrasts mean that the coefficients sum to 0. You need to create two contrast coded variables in the fixed-effects design matrix,  $X1$  and  $X2$ , where

$$Shift\_Evening = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases} \quad \text{and} \quad Shift\_Morning = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}$$

The model corresponds to

$$\text{Morning Shift: } QCDev_{im} = \beta_0 + \beta_2 Shift\_Morning_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 5,$$

$$\text{Evening Shift: } QCDev_{im} = \beta_0 + \beta_1 Shift\_Evening_i + b_{0m} + \varepsilon_{im},$$

$$\text{Night Shift: } QCDev_{im} = \beta_0 - \beta_1 Shift\_Evening_i - \beta_2 Shift\_Morning_i + b_{0m} + \varepsilon_{im},$$

where  $i$  represents the observations, and  $m$  represents the operators,  $i = 1, 2, \dots, 15$ , and  $m = 1, 2, \dots, 5$ . The random effects and the observation error have these distributions:  $b_{0m} \sim N(0, \sigma_{b_{0m}}^2)$  and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
S = shift.Shift;
X1 = (S=='Morning') - (S=='Night');
X2 = (S=='Evening') - (S=='Night');
X = [ones(15,1), X1, X2];
y = shift.QCDev;
```

```
Z = ones(15,1);
G = shift.Operator;
```

Fit a linear mixed-effects model using the specified design matrices and restricted maximum likelihood method.

```
lme = fitlmematrix(X,y,Z,G,'FitMethod','REML','FixedEffectPredictors',
{'Intercept','S_Morning','S_Evening'},'RandomEffectPredictors',{
'RandomEffectGroups',{'Operator'},'DummyVarCoding','effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

```
Formula:
```

```
y ~ Intercept + S_Morning + S_Evening + (Intercept | Operator)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	3.6525	0.94109	3.8812	12	0.00218
'S_Morning'	-0.91973	0.31206	-2.9473	12	0.0122
'S_Evening'	-0.53293	0.31206	-1.7078	12	0.113

```
Random effects covariance parameters (95% CIs):
```

```
Group: Operator (5 Levels)
```

Name1	Name2	Type	Estimate	Low
-------	-------	------	----------	-----

# fitlmematrix

---

```
      'Intercept'      'Intercept'      'std'      2.0457      0.9820
Group: Error
  Name      Estimate      Lower      Upper
  'Res Std'      0.85462      0.52357      1.395
```

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

```
B = randomEffects(lme)
```

```
B =
```

```
  0.5775
  1.1757
 -2.1715
  2.3655
 -1.9472
```

The estimated deviation from the target quality characteristics for the third operator working the evening shift is

$$\begin{aligned}\hat{Y}_{Evening, Operator3} &= \hat{\beta}_0 + \hat{\beta}_1 Shift\_Evening + \hat{b}_{03} \\ &= 3.6525 - 0.53293 - 2.1715 \\ &= 0.94807.\end{aligned}$$

You can also display this value as follows.

```
F = fitted(lme);
F(shift.Shift=='Evening' & shift.Operator=='3')
```

```
ans =
```

```
  0.9481
```

## Correlated and Uncorrelated Random-Effects Terms

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower and the cylinders, and uncorrelated random effect for intercept and acceleration grouped by the model year. This model corresponds to

$$MPG_{im} = \beta_0 + \beta_1 Acc_i + \beta_2 HP + b_{0m} + b_{1m} Acc_{im} + \varepsilon_{im}, \quad m = 1, 2, 3,$$

with the random-effects terms having these distributions:  $b_{0m} \sim N(0, \sigma_0^2)$ , and  $b_{1m} \sim N(0, \sigma_1^2)$ .  $m$  represents the model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];
Z = {ones(406,1), Acceleration};
G = {Model_Year, Model_Year};
Model_Year = nominal(Model_Year);
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X, MPG, Z, G, 'FixedEffectPredictors', ...
{'Intercept', 'Acceleration', 'Horsepower'}, 'RandomEffectPredictors', ...
{'Intercept'}, {'Acceleration'}}, 'RandomEffectGroups', {'Model_Year'}, 'M
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	392
Fixed effects coefficients	3

# fitmematrix

---

Random effects coefficients            26  
Covariance parameters                 3

Formula:

$y \sim \text{Intercept} + \text{Acceleration} + \text{Horsepower} + (\text{Intercept} \mid \text{Model\_Year})$

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
2194.5	2218.3	-1091.3	2182.5

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pVal
'Intercept'	49.839	2.0518	24.291	389	5.61
'Acceleration'	-0.58565	0.10846	-5.3995	389	1.16
'Horsepower'	-0.16534	0.0071227	-23.213	389	1.97

Random effects covariance parameters (95% CIs):

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'Intercept'	'Intercept'	'std'	8.0669e-07	NaN	NaN

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate
'Acceleration'	'Acceleration'	'std'	0.18783

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	3.7258	3.4698	4.0007

The standard deviation of the random effect for the intercept does not seem significant.

Refit the model with potentially correlated random effects for intercept and acceleration. In this case, the random-effects terms has this prior distribution



$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix} \right),$$

where  $m$  represents the model year.

First, prepare the random-effects design matrix and grouping variable.

```
Z = [ones(406,1) Acceleration];
G = Model_Year;
```

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'})
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	4

Formula:

```
y ~ Intercept + Acceleration + Horsepower + (Intercept + Acceleration | Model_Year)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
2193.5	2221.3	-1089.7	2179.5

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'Intercept'	50.133	2.2652	22.132	389
'Acceleration'	-0.58327	0.13394	-4.3545	389
'Horsepower'	-0.16954	0.0072609	-23.35	389

Random effects covariance parameters (95% CIs):

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate
'Intercept'	'Intercept'	'std'	3.3475
'Acceleration'	'Intercept'	'corr'	-0.87971
'Acceleration'	'Acceleration'	'std'	0.33789

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	3.6874	3.4298	3.9644

The confidence intervals for the standard deviations and the correlation between the random effects for intercept and acceleration do not include 0s, hence they seem significant. You can compare these two models using the compare method.

## Specify the Covariance Pattern

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

weight contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Define Subject and Program as categorical variables.

```
Subject = nominal(Subject);
Program = nominal(Program);
```

Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, and week as the fixed effects.

```
D = dummyvar(Program);
X = [ones(120,1), InitialWeight, D(:,2:4), Week];
Z = [ones(120,1) Week];
G = Subject;
```

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 Week_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i + \beta_5 I[PD]_i \\ + b_{0m} + b_{1m} Week2_{im} + b_{2m} Week4_{im} + b_{3m} Week6_{im} + b_{4m} Week8_{im} \\ + b_{5m} Week10_{im} + b_{6m} Week12_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .

$\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{1m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[.]$  is a dummy variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing program type B. The random effects and observation error have these prior distributions:  $b_{0m} \sim N(0, \sigma^2_0)$ ,  $b_{1m} \sim N(0, \sigma^2_1)$ , and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

Fit the model using `fitlmematrix` with the defined design matrices and grouping variables. Assume the repeated observations collected on a subject have common variance along diagonals.

```
lme = fitlmematrix(X,y,Z,G,'FixedEffectPredictors',...
{'Intercept','InitWeight','PrgB','PrgC','PrgD','Week'},...
'RandomEffectPredictors',{'Intercept','Week'},...
'RandomEffectGroups',{'Subject'},'CovariancePattern','Isotropic')
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	120
Fixed effects coefficients	6

# fitlmematrix

```
Random effects coefficients      40
Covariance parameters           2
```

Formula:

```
y ~ Intercept + InitWeight + PrgB + PrgC + PrgD + Week + (Intercept +
```

Model fit statistics:

```
AIC      BIC      LogLikelihood  Deviance
-24.783  -2.483    20.391        -40.783
```

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pVal
'Intercept'	0.4208	0.28169	1.4938	114	0.034
'InitWeight'	0.0045552	0.0015338	2.9699	114	0.003
'PrgB'	0.36993	0.12119	3.0525	114	0.002
'PrgC'	-0.034009	0.1209	-0.28129	114	0.777
'PrgD'	0.121	0.12111	0.99911	114	0.321
'Week'	0.19881	0.037134	5.3538	114	4.51e-08

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.16561	0.1289

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10272	0.088014	0.11987

## Alternative Functionality

You can also fit a linear mixed-effects model using `fitlme(ds, formula)`, where `ds` is a dataset array containing the response `y`, the predictor variables `X`, and the grouping variables, and `formula` is of the form `'y ~ fixed + (random1|g1) + ... + (randomR|gR)'`.

## Definitions

### Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.

$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q^*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ . For example, if

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$

### Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

# fitlmematrix

---

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## See Also

[fitlmeLinearMixedModel](#) | [compare](#) |

**Purpose**

Fit probability distribution object to data

**Syntax**

```
pd = fitdist(x,distname)
pd = fitdist(x,distname,Name,Value)

[pdca,gn,gl] = fitdist(x,distname,'By',groupvar)
[pdca,gn,gl] = fitdist(x,distname,'By',groupvar,Name,Value)
```

**Description**

`pd = fitdist(x,distname)` creates a probability distribution object by fitting the distribution specified by `distname` to the data in column vector `x`.

`pd = fitdist(x,distname,Name,Value)` creates the probability distribution object with additional options specified by one or more name-value pair arguments. For example, you can indicate censored data or specify control parameters for the iterative fitting algorithm.

`[pdca,gn,gl] = fitdist(x,distname,'By',groupvar)` creates probability distribution objects by fitting the distribution specified by `distname` to the data in `x` based on the grouping variable `groupvar`. It returns a cell array of fitted probability distribution objects, `pdca`, a cell array of group labels, `gn`, and a cell array of grouping variable levels, `gl`.

`[pdca,gn,gl] = fitdist(x,distname,'By',groupvar,Name,Value)` returns the above output arguments using additional options specified by one or more name-value pair arguments. For example, you can indicate censored data or specify control parameters for the iterative fitting algorithm.

**Input Arguments****x - Input data**

column vector

Input data, specified as a column vector. `fitdist` ignores NaN values in `x`. Additionally, any NaN values in the censoring vector or frequency vector causes `fitdist` to ignore the corresponding values in `x`.

## Data Types

single | double

## distname - Distribution name

string

Distribution name, specified as one of the following strings. The distribution specified by `distname` determines the class type of the returned probability distribution object.

Distribution Name	Description	Distribution Class
'Beta'	Beta distribution	prob.BetaDistribution
'Binomial'	Binomial distribution	prob.BinomialDistribution
'BirnbbaumSaunders'	Birnbbaum-Saunders distribution	prob.BirnbbaumSaundersDistribution
'Burr'	Burr distribution	prob.BurrDistribution
'Exponential'	Exponential distribution	prob.ExponentialDistribution
'ExtremeValue'	Extreme Value distribution	prob.ExtremeValueDistribution
'Gamma'	Gamma distribution	prob.GammaDistribution
'GeneralizedExtremeValue'	Generalized Extreme Value distribution	prob.GeneralizedExtremeValueDistribution
'GeneralizedPareto'	Generalized Pareto distribution	prob.GeneralizedParetoDistribution
'InverseGaussian'	Inverse Gaussian distribution	prob.InverseGaussianDistribution
'Kernel'	Kernel distribution	prob.KernelDistribution
'Logistic'	Logistic distribution	prob.LogisticDistribution
'Loglogistic'	Loglogistic distribution	prob.LoglogisticDistribution



Distribution Name	Description	Distribution Class
'Lognormal'	Lognormal distribution	prob.LognormalDistribution
'Multinomial'	Multinomial distribution	prob.MultinomialDistribution
'Nakagami'	Nakagami distribution	prob.NakagamiDistribution
'NegativeBinomial'	Negative Binomial distribution	prob.NegativeBinomialDistribution
'Normal'	Normal distribution	prob.NormalDistribution
'Poisson'	Poisson distribution	prob.PoissonDistribution
'Rayleigh'	Rayleigh distribution	prob.RayleighDistribution
'Rician'	Rician distribution	prob.RicianDistribution
'tLocationScale'	$t$ Location-Scale distribution	prob.tLocationScaleDistribution
'Weibull'	Weibull distribution	prob.WeibullDistribution

### groupvar - Grouping variable

categorical array | logical or numeric vector | cell array of strings

Grouping variable, specified as a categorical array, logical or numeric vector, or cell array of strings. Each unique value in a grouping variable defines a group.

For example, if `Gender` is a cell array of strings with values 'Male' and 'Female', you can use `Gender` as a grouping variable to fit a distribution to your data by gender.

More than one grouping variable can be used by specifying a cell array of grouping variable names. Observations are placed in the same group if they have common values of all specified grouping variables.

For example, if `Smoker` is a logical vector with values 0 for nonsmokers and 1 for smokers, then specifying the cell array `{Gender,Smoker}`

divides observations into four groups: Male Smoker, Male Nonsmoker, Female Smoker, and Female Nonsmoker.

**Example:** {Gender,Smoker}

## Data Types

single | double | logical | cell | char

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `fitdist(x,'Kernel','Kernel','triangle')` fits a kernel distribution object to the data in `x` using a triangular kernel function.

## 'Censoring' - Logical flag for censored data

0 (default) | vector of logical values

Logical flag for censored data, specified as the comma-separated pair consisting of `'Censoring'` and a vector of logical values that is the same size as input vector `x`. The value is 1 when the corresponding element in `x` is a right-censored observation and 0 when the corresponding elements is an exact observation. The default is a vector of 0s, indicating that all observations are exact.

`fitdist` ignores any NaN values in this censoring vector. Additionally, any NaN values in `x` or the frequency vector causes `fitdist` to ignore the corresponding values in the censoring vector.

## Data Types

logical

## 'Frequency' - Observation frequency

1 (default) | vector of nonnegative integer values

Observation frequency, specified as the comma-separated pair consisting of `'Frequency'` and a vector of nonnegative integer values

that is the same size as input vector `x`. Each element of the frequency vector specifies the frequencies for the corresponding elements in `x`. The default is a vector of 1s, indicating that each value in `x` only appears once.

`fitdist` ignores any NaN values in this frequency vector are ignored by the fitting calculations. Additionally, any NaN values in `x` or the censoring vector causes `fitdist` to ignore the corresponding values in the frequency vector.

**Data Types**

logical

**'Options' - Control parameters**

structure

Control parameters for the iterative fitting algorithm, specified as the comma-separated pair consisting of 'Options' and a structure you create using `statset`.

**Data Types**

struct

**'NTrials' - Number of trials**

positive integer value

Number of trials for the binomial distribution, specified as the comma-separated pair consisting of 'NTrials' and a positive integer value. You must specify `distname` as 'Binomial' to use this option.

**Data Types**

single | double

**'Theta' - Threshold parameter**

0 (default) | scalar value

Threshold parameter for the generalized Pareto distribution, specified as the comma-separated pair consisting of 'Theta' and a scalar value. You must specify `distname` as 'GeneralizedPareto' to use this option.

## Data Types

single | double

### 'Kernel' - Kernel smoother type

'normal' (default) | 'box' | 'triangle' | 'epanechnikov'

Kernel smoother type, specified as the comma-separated pair consisting of 'Kernel' and one of the following:

- 'normal'
- 'box'
- 'triangle'
- 'epanechnikov'

You must specify `distname` as 'Kernel' to use this option.

### 'Support' - Kernel density support

'unbounded' (default) | 'positive' | two-element vector

Kernel density support, specified as the comma-separated pair consisting of 'Support' and a string or two-element vector. The string must be one of the following.

'unbounded'	Density can extend over the whole real line.
'positive'	Density is restricted to positive values.

Alternatively, you can specify a two-element vector giving finite lower and upper limits for the support of the density.

You must specify `distname` as 'Kernel' to use this option.

## Data Types

single | double

### 'Width' - Bandwidth of kernel smoothing window

scalar value

Bandwidth of the kernel smoothing window, specified as the comma-separated pair consisting of 'Width' and a scalar value. The default value used by `fitdist` is optimal for estimating normal densities, but you might want to choose a smaller value to reveal features such as multiple modes. You must specify `distname` as 'Kernel' to use this option.

**Data Types**

single | double

**Output Arguments****pd - Probability distribution**

probability distribution object

Probability distribution, returned as a probability distribution object. The distribution specified by `distname` determines the class type of the returned probability distribution object.

**pdca - Probability distribution objects**

cell array

Probability distribution objects of the type specified by `distname`, returned as a cell array.

**gn - Group labels**

cell array of strings

Group labels, returned as a cell array of strings.

**gl - Grouping variable levels**

cell array of strings

Grouping variable levels, returned as a cell array of strings containing one column for each grouping variable.

**Examples****Fit a Normal Distribution to Data**

Load the sample data. Create a vector containing the patients' weight data.

# fitdist

---

```
load hospital;  
x = hospital.Weight;
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x,'Normal')
```

```
pd =
```

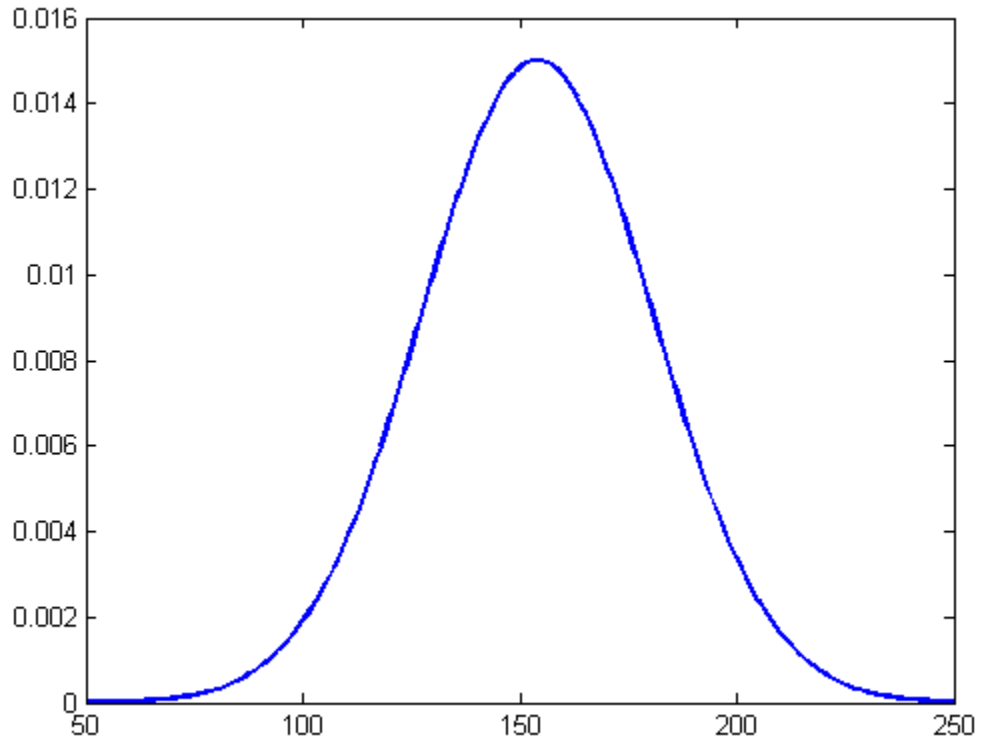
```
NormalDistribution
```

```
Normal distribution
```

```
mu =      154    [148.728, 159.272]  
sigma = 26.5714  [23.3299, 30.8674]
```

Plot the pdf of the distribution.

```
x_values = 50:1:250;  
pdf = pdf(pd,x_values);  
plot(x_values,pdf,'LineWidth',2)
```



### Fit a Kernel Distribution to Data

Load the sample data. Create a vector containing the patients' weight data.

```
load hospital;  
x = hospital.Weight;
```

Create a kernel distribution object by fitting it to the data. Use the Epanechnikov kernel function.

# fitdist

---

```
pd = fitdist(x,'Kernel','Kernel','epanechnikov')
```

```
pd =
```

```
KernelDistribution
```

```
Kernel = epanechnikov
```

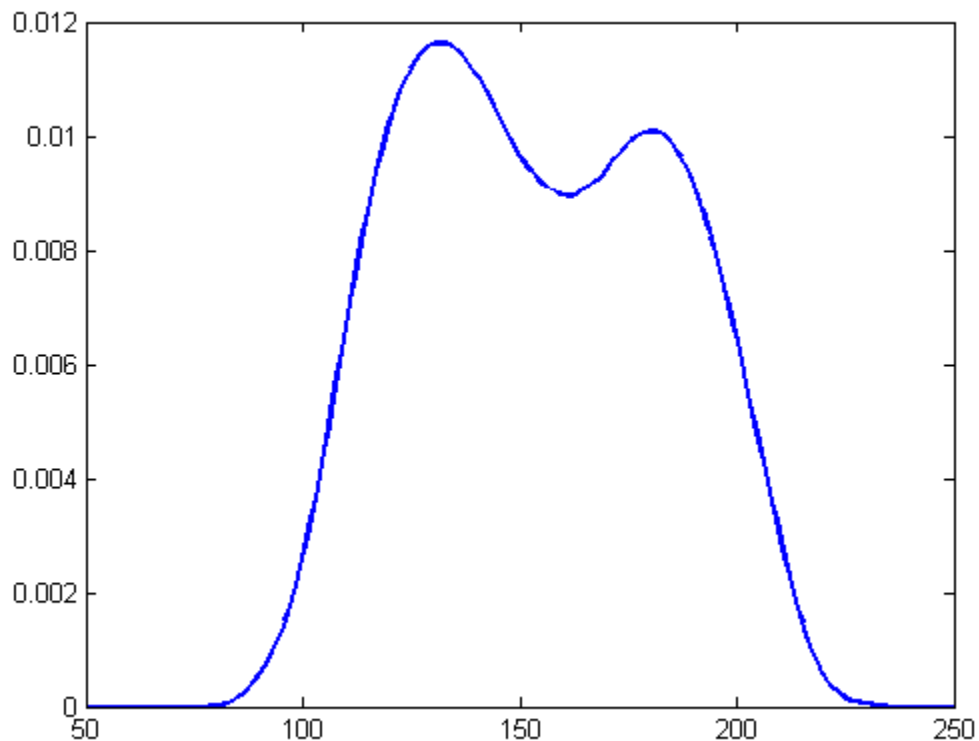
```
Bandwidth = 14.3792
```

```
Support = unbounded
```

Plot the pdf of the distribution.

```
x_values = 50:1:250;  
pdf = pdf(pd,x_values);  
plot(x_values,pdf)
```





### Fit Normal Distributions to Grouped Data

Load the sample data. Create a vector containing the patients' weight data.

```
load hospital;  
x = hospital.Weight;
```

Create normal distribution objects by fitting them to the data, grouped by patient gender.

# fitdist

---

```
gender = hospital.Sex;  
[pdca,gn,gl] = fitdist(x, 'Normal', 'By', gender)
```

```
pdca =
```

```
Column 1
```

```
[1x1 prob.NormalDistribution]
```

```
Column 2
```

```
[1x1 prob.NormalDistribution]
```

```
gn =
```

```
'Female'
```

```
'Male'
```

```
gl =
```

```
'Female'
```

```
'Male'
```

The cell array `pdca` contains two probability distribution objects, one for each gender group. The cell array `gn` contains two strings of the group labels. The cell array `gl` contains two strings of the group levels.

View each distribution in the cell array `pdca` to compare the mean, `mu`, and the standard deviation, `sigma`, grouped by patient gender.

```
female = pdca{1} % distribution for females
```

```
female =
```

```
NormalDistribution
```

```
Normal distribution
    mu = 130.472    [128.183, 132.76]
    sigma = 8.30339 [6.96947, 10.2736]

male = pdca{2} % distribution for males

male =

NormalDistribution

Normal distribution
    mu = 180.532    [177.833, 183.231]
    sigma = 9.19322 [7.63933, 11.5466]
```

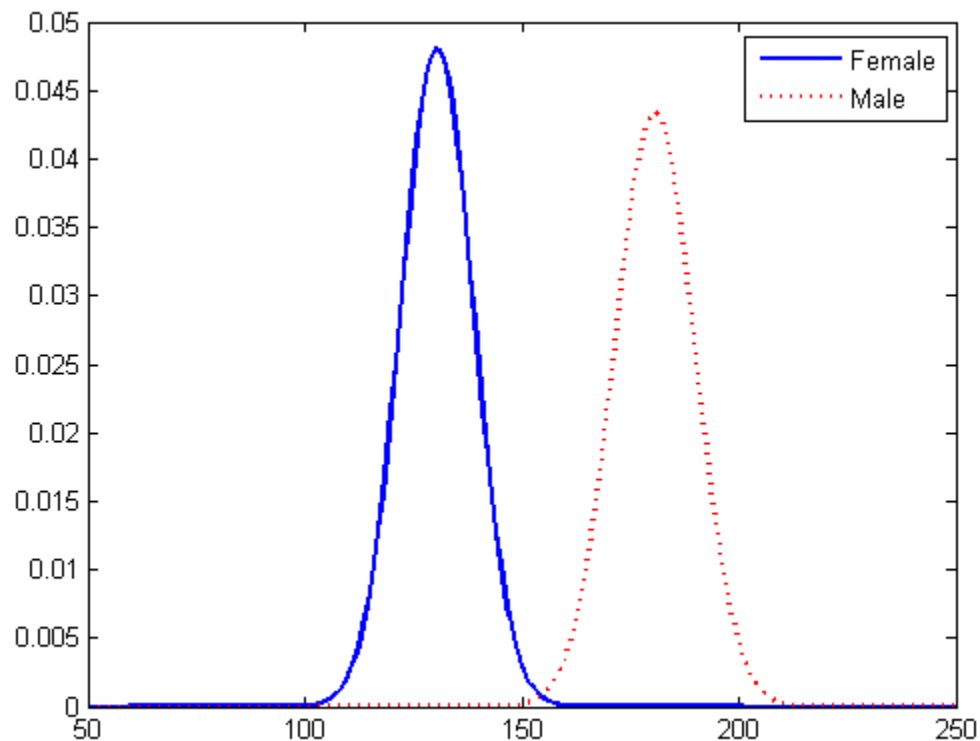
Compute the pdf of each distribution.

```
x_values = 50:1:250;
femalepdf = pdf(female,x_values);
malepdf = pdf(male,x_values);
```

Plot the pdfs for a visual comparison of weight distribution by gender.

```
figure;
plot(x_values,femalepdf,'LineWidth',2)
hold on;
plot(x_values,malepdf,'Color','r','LineStyle',':','LineWidth',2)
legend(gn,'Location','NorthEast');
hold off;
```

# fitdist



## Fit Kernel Distributions to Grouped Data

Load the sample data. Create a vector containing the patients' weight data.

```
load hospital;  
x = hospital.Weight;
```

Create kernel distribution objects by fitting them to the data, grouped by patient gender. Use a triangular kernel function.

```
gender = hospital.Sex;  
[pdca,gn,gl] = fitdist(x,'Kernel','By',gender,'Kernel','triangle');
```

View each distribution in the cell array pdca to see the kernel distributions for each gender.

```
female = pdca{1} % distribution for females
```

```
female =
```

```
KernelDistribution
```

```
Kernel = triangle  
Bandwidth = 4.25894  
Support = unbounded
```

```
male = pdca{2} % distribution for males
```

```
male =
```

```
KernelDistribution
```

```
Kernel = triangle  
Bandwidth = 5.08961  
Support = unbounded
```

Compute the pdf of each distribution.

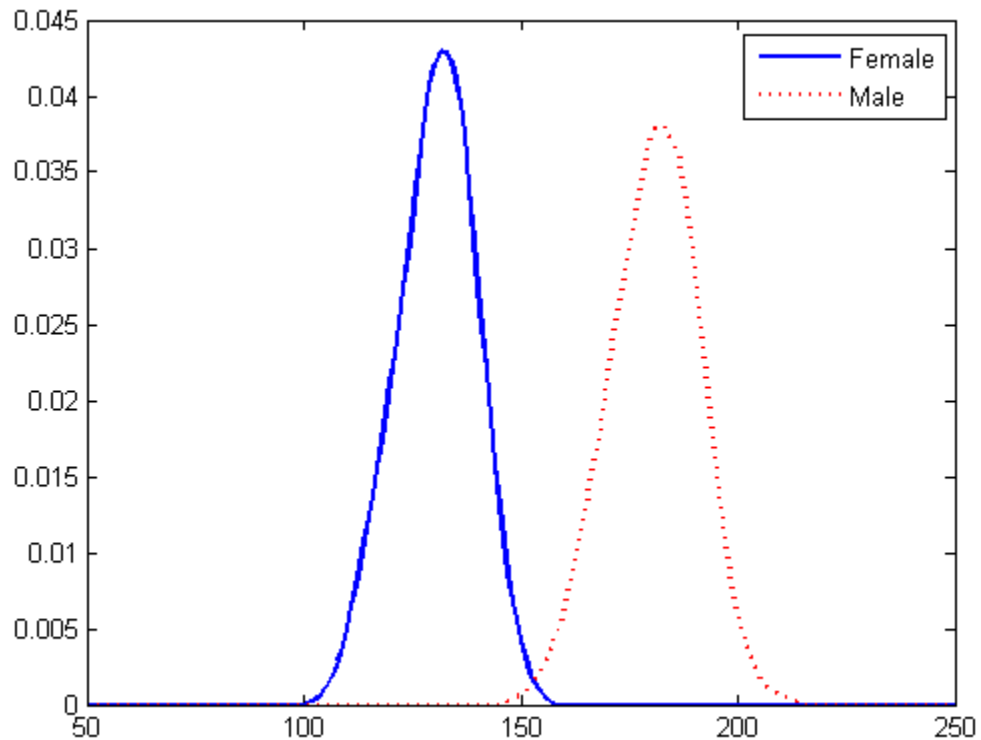
```
x_values = 50:1:250;  
femalepdf = pdf(female,x_values);  
malepdf = pdf(male,x_values);
```

Plot the pdfs for a visual comparison of weight distribution by gender.

```
figure;  
plot(x_values,femalepdf,'LineWidth',2)  
hold on;  
plot(x_values,malepdf,'Color','r','LineStyle',':','LineWidth',2)
```

# fitdist

```
legend(gn, 'Location', 'NorthEast');  
hold off;
```



## Algorithms

The `fitdist` function fits most distributions using maximum likelihood estimation. Two exceptions are the normal and lognormal distributions with uncensored data.

- For the uncensored normal distribution, the estimated value of the sigma parameter is the square root of the unbiased estimate of the variance.

- For the uncensored lognormal distribution, the estimated value of the sigma parameter is the square root of the unbiased estimate of the variance of the log of the data.

## Alternative Functionality

### App

The Distribution Fitting app opens a graphical user interface for you to import data from the workspace and interactively fit a probability distribution to that data. You can then save the distribution to the workspace as a probability distribution object. Open the Distribution Fitting app using `dfittool`, or click Distribution Fitting on the Apps tab.

## References

- [1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.
- [2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.
- [3] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

## See Also

`makedist` | `dfittool`

# fitensemble

---

## Purpose

Fitted ensemble for classification or regression

## Syntax

```
ens = fitensemble(X,Y,method,nlearn,learners)
ens = fitensemble(X,Y,method,nlearn,learners,Name,Value)
```

## Description

`ens = fitensemble(X,Y,method,nlearn,learners)` creates an ensemble model that predicts responses to data. The ensemble consists of models listed in `learners`.

`ens = fitensemble(X,Y,method,nlearn,learners,Name,Value)` creates an ensemble model with additional options specified by one or more `Name, Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Tips

- If `X` has categorical predictors with many levels, use 'GentleBoost' or 'LogitBoost' for binary classification. Using other algorithms in this case would likely cause long training times and can exhaust memory.

## Input Arguments

### X

Matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

### Y

For classification, `Y` is a categorical variable, character array, or cell array of strings. Each row of `Y` represents the classification of the corresponding row of `X`.

For regression, `Y` is a numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

### method

Case-insensitive string consisting of one of the following.

- For classification with two classes:



- 'AdaBoostM1'
- 'LogitBoost'
- 'GentleBoost'
- 'RobustBoost' (requires an Optimization Toolbox license)
- 'LPBoost' (requires an Optimization Toolbox license)
- 'TotalBoost' (requires an Optimization Toolbox license)
- 'RUSBoost'
- 'Subspace'
- 'Bag'
- For classification with three or more classes:
  - 'AdaBoostM2'
  - 'LPBoost' (requires an Optimization Toolbox license)
  - 'TotalBoost' (requires an Optimization Toolbox license)
  - 'RUSBoost'
  - 'Subspace'
  - 'Bag'
- For regression:
  - 'LSBoost'
  - 'Bag'

'Bag' applies to all methods. So when you use 'Bag', indicate whether you want a classifier or regressor with the `type` name-value pair set to 'classification' or 'regression'.

### **nlearn**

Number of ensemble learning cycles, a positive integer (or the string 'AllPredictorCombinations', see the next paragraph). At every training cycle, fitensemble loops over all learner templates in

learners and trains one weak learner for every template. The total number of trained learners in `ens` is `nlearn*numel(learners)`.

If you set `method` to `'Subspace'`, you can set `nlearn` to `'AllPredictorCombinations'`. With this setting, `fitensemble` constructs learners for all possible combinations of predictors taken `NPredToSample` at a time. This gives a total of `nchoosek(size(X,2),NPredToSample)` learners in the ensemble. You can use only one learner template for this setting.

`nlearn` for ensembles can vary from a few dozen to a few thousand. Usually, an ensemble with a good predictive power needs from a few hundred to a few thousand weak learners. You do not have to train an ensemble for that many cycles at once. You can start by growing a few dozen learners, inspect the ensemble performance and, if necessary, train more weak learners using the `resume` method of the ensemble.

## learners

One of the following:

- A string with the name of a weak learner:
  - `'Discriminant'` (applies only to `'Subspace'`)
  - `'KNN'` (applies only to `'Subspace'`)
  - `'tree'` (applies to all methods except `'Subspace'`)
- A single weak learner template you create with `ClassificationTree.template`, `RegressionTree.template`, `ClassificationKNN.template`, or `ClassificationDiscriminant.template`.
- A cell array of weak learner templates. Usually you should supply only one weak learner template.

Ensemble performance depends on the parameters of the weak learners, and you can get poor performance using weak learners with default parameters. Specify the parameters for the weak learners in the template. Specify parameters for the ensemble in the `fitensemble` name-value pairs.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### All Ensembles:

#### 'CategoricalPredictors'

List of categorical predictors. Pass `CategoricalPredictors` as one of:

- A numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .
- A logical vector of length  $p$ , where a true entry means that the corresponding column of  $X$  is a categorical variable.
- 'all', meaning all predictors are categorical.
- A cell array of strings, where each element in the array is the name of a predictor variable. The names must match entries in the `PredictorNames` property.
- A character matrix, where each row of the matrix is the name of a predictor variable. The names must match entries in the `PredictorNames` property. Pad the names with extra blanks so each row of the character matrix has the same length.

You can set `CategoricalPredictors` for these learners:

- 'Tree'
- 'KNN', when all predictors are categorical

**Default:** []

#### 'crossval'

If 'on', grows a cross-validated learner with 10 folds. You can use 'kfold', 'holdout', 'leaveout', or 'cvpartition' parameters to

override this cross-validation setting. You can only use one of these four parameters ('kfold', 'holdout', 'leaveout', or 'cvpartition') at a time when creating a cross-validated learner.

**Default:** 'off'

## **'cvpartition'**

Partition created with cvpartition to use in a cross-validated learner. You can only use one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## **'fresample'**

Fraction of the training set to be selected by resampling for every weak learner. A numeric scalar from 0 to 1. This parameter has no effect unless you grow an ensemble by bagging or set 'resample' to 'on'. The default setting is the one used most often for an ensemble grown by resampling.

**Default:** 1

## **'holdout'**

Holdout validation tests the specified fraction of the data, and uses the remaining data for training. Specify a numeric scalar from 0 to 1. You can only use one of these four options at a time for creating a cross-validated learner: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

## **'kfold'**

Number of folds to use in a cross-validated learner, a positive integer. You can only use one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

**Default:** 10

## **'leaveout'**

Use leave-one-out cross validation by setting to 'on'. You can only use one of these four options at a time: 'kfold', 'holdout', 'leaveout', or 'cvpartition'.

**'NPredToSample'**

Number of predictors in each random subspace learner, a positive integer from 1 to `size(X,2)`.

**Default:** 1

**'nprint'**

Printout frequency, a positive integer scalar. Set to 'off' for no printout. Use this parameter to track how many weak learners have been trained so far. This is useful when you train ensembles with many learners on large datasets. If you use one of the cross-validation options, this parameter defines the printout frequency per number of cross-validation folds.

**Default:** 'off'

**'PredictorNames'**

Cell array of names for the predictor variables, in the order in which they appear in X.

**Default:** {'x1', 'x2', ...}

**'replace'**

'on' or 'off'. If 'on', sample with replacement. If 'off', sample without replacement. This parameter has no effect unless you grow an ensemble by bagging or set `resample` to 'on'. If you set `resample` to 'on' and `replace` to 'off', `fitensemble` samples training observations assuming uniform weights, and boosts by reweighting observations.

**Default:** 'on'

## **'resample'**

'on' or 'off'. If 'on', grow an ensemble by resampling, with the resampling fraction given by `fresample`, and sampling with or without replacement given by `replace`.

- Boosting — When 'off', the boosting algorithm reweights observations at every learning iteration. When 'on', the algorithm samples training observations using updated weights as the multinomial sampling probabilities.
- Bagging — You can use only the default value of this parameter ('on').

**Default:** 'off' for boosting, 'on' for bagging

## **'ResponseName'**

Name of the response variable  $Y$ , a string.

**Default:** 'Y'

## **'type'**

String, either 'classification' or 'regression'. Specify type when the method is 'bag'.

## **'weights'**

Vector of observation weights. The length of weights is the number of rows in  $X$ .

**Default:** `ones(size(X,1),1)`

## **Classification Ensembles:**

### **'classnames'**

Array of class names. Specify a data type the same as exists in  $Y$ .

**Default:** Class names that exist in  $Y$

### **'cost'**

Square matrix  $C$ , where  $C(i, j)$  is the cost of classifying a point into class  $j$  if its true class is  $i$ . Alternatively, `cost` can be a structure  $S$  having two fields:  $S.ClassNames$  containing the group names as a categorical variable, character array, or cell array of strings; and  $S.ClassificationCosts$  containing the cost matrix  $C$ .

**Default:**  $C(i, j) = 1$  if  $i \neq j$ , and  $C(i, j) = 0$  if  $i = j$

### **'prior'**

Prior probabilities for each class. Specify as one of:

- A string:
  - `'empirical'` determines class probabilities from class frequencies in  $Y$ . If you pass observation weights, they are used to compute the class probabilities.
  - `'uniform'` sets all class probabilities equal.
- A vector (one scalar value for each class)
- A structure  $S$  with two fields:
  - $S.ClassNames$  containing the class names as a categorical variable, character array, or cell array of strings
  - $S.ClassProbs$  containing a vector of corresponding probabilities

If you set values for both `weights` and `prior`, the weights are renormalized to add up to the value of the prior probability in the respective class.

**Default:** `'empirical'`

**AdaBoostM1, AdaBoostM2, LogitBoost, GentleBoost, RUSBoost, and LSBoost:**

## **'LearnRate'**

Learning rate for shrinkage, a numeric scalar from 0 to 1. If you set the learning rate to less than 1, the ensemble requires more learning iterations but often achieves a better accuracy. 0.1 is a popular choice for an ensemble grown with shrinkage.

**Default:** 1

## **RUSBoost:**

### **'RatioToSmallest'**

Either a numeric scalar or vector with  $K$  elements when there are  $K$  classes. Every element of this vector is the sampling proportion for this class with respect to the class with fewest observations in  $Y$ . If you pass a scalar, `fitensemble` uses this sampling proportion for all classes. For example, suppose you have class A with 100 observations and class B with 10 observations. If you pass `[2 1]` for `'RatioToSmallest'`, every learner in the ensemble is trained on 20 observations of class A and 10 observations of class B. If you pass 2 or `[2 2]`, every learner is trained on 20 observations of class A and 20 observations of class B. If you pass `'ClassNames'`, `fitensemble` matches elements in the array of class names to elements in this vector.

**Default:** `ones(K,1)`

## **LPBoost and TotalBoost:**

### **'MarginPrecision'**

Margin precision, a numeric scalar between 0 and 1. `MarginPrecision` affects the number of boosting iterations required for conversion. Use a small value to grow an ensemble with many learners, and use a large value to grow an ensemble with few learners.

**Default:** 0.01



**RobustBoost:****'RobustErrorGoal'**

Target classification error for `RobustBoost`, a numeric scalar from 0 to 1. Usually there is an optimal range for this parameter for your training data. If you set the error goal too low or too high, `RobustBoost` can produce a model with poor classification accuracy.

**Default:** 0.1

**'RobustMarginSigma'**

Spread of the distribution of classification margins over the training set for `RobustBoost`, a numeric positive scalar. You should consult literature on `RobustBoost` before setting this parameter

**Default:** 0.1

**'RobustMaxMargin'**

Maximal classification margin for `RobustBoost` in the training set, a nonnegative numeric scalar. `RobustBoost` minimizes the number of observations in the training set with classification margins below `RobustMaxMargin`.

**Default:** 0

**Output Arguments****ens**

Ensemble object for predicting characteristics. The class of `ens` depends on settings. In the following table, cross-validation names are `crossval`, `kfold`, `holdout`, `leaveout`, and `cvpartition`.

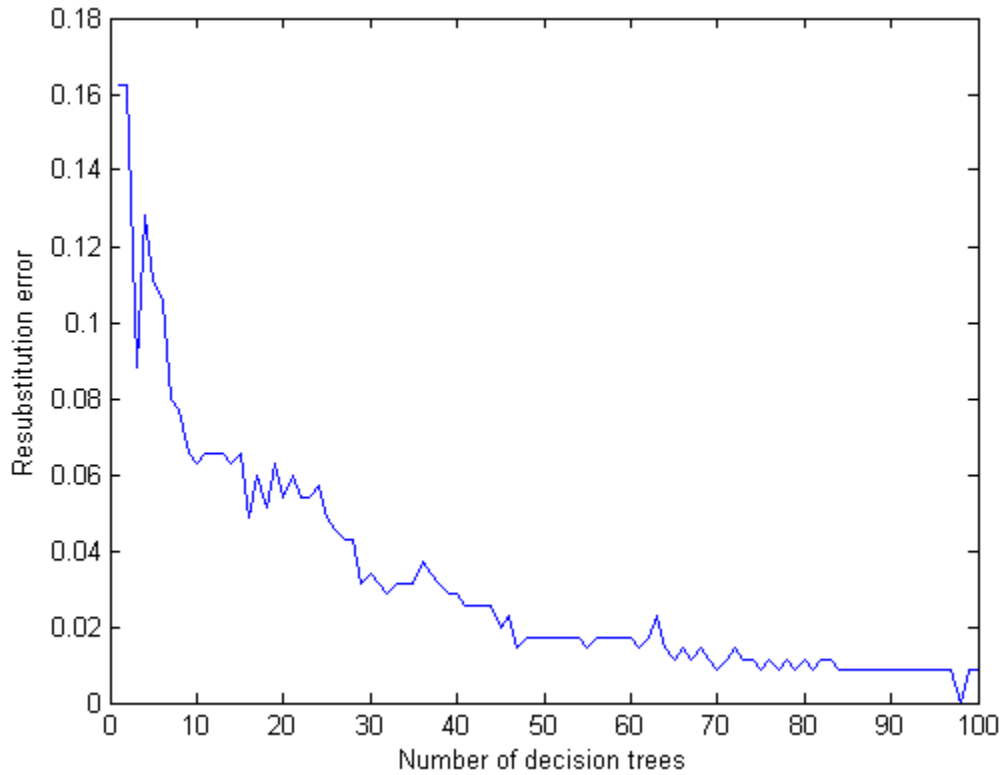
# fitensemble

Settings	Class
resample name-value pair is 'off', and you don't set a cross-validation name-value pair.	ClassificationEnsemble
resample name-value pair is 'off', and you don't set a cross-validation name-value pair.	RegressionEnsemble
resample name-value pair is 'on', type is 'classification', and you don't set a cross-validation name-value pair.	ClassificationBaggedEnsemble
resample name-value pair is 'on', type is 'regression', and you don't set a cross-validation name-value pair.	RegressionBaggedEnsemble
method is a classification method, and you set a cross-validation name-value pair.	ClassificationPartitionedEnsemble
method is a regression method, and you set a cross-validation name-value pair.	RegressionPartitionedEnsemble

## Examples

Train a boosting ensemble, and inspect the resubstitution loss:

```
load ionosphere;
ada = fitensemble(X,Y,'AdaBoostM1',100,'tree');
plot(resubLoss(ada,'mode','cumulative'));
xlabel('Number of decision trees');
ylabel('Resubstitution error');
```



Train a regression ensemble to predict car mileage based on the number of cylinders, engine displacement, horsepower, and weight. Predict the mileage for a four-cylinder car with a 200 cubic inch displacement, 150 horsepower, and weighing 3000 lbs.

# fitensemb

---

```
load carsmall
X = [Cylinders,Displacement,Horsepower,Weight];
xnames = {'Cylinders','Displacement','Horsepower','Weight'};
t = RegressionTree.template('Surrogate','on');
rens = fitensemble(X,MPG,'LSBoost',100,t,'PredictorNames',xnames)

rens =
classreg.learning.regr.ReggressionEnsemble:
    PredictorNames: {'Cylinders' 'Displacement' 'Horsepower' 'Weight'}
    CategoricalPredictors: []
    ResponseName: 'Y'
    ResponseTransform: 'none'
    NObservations: 94
    NTrained: 100
    Method: 'LSBoost'
    LearnerNames: {'Tree'}
    ReasonForTermination: [1x77 char]
    FitInfo: [100x1 double]
    FitInfoDescription: [2x83 char]
    Regularization: []

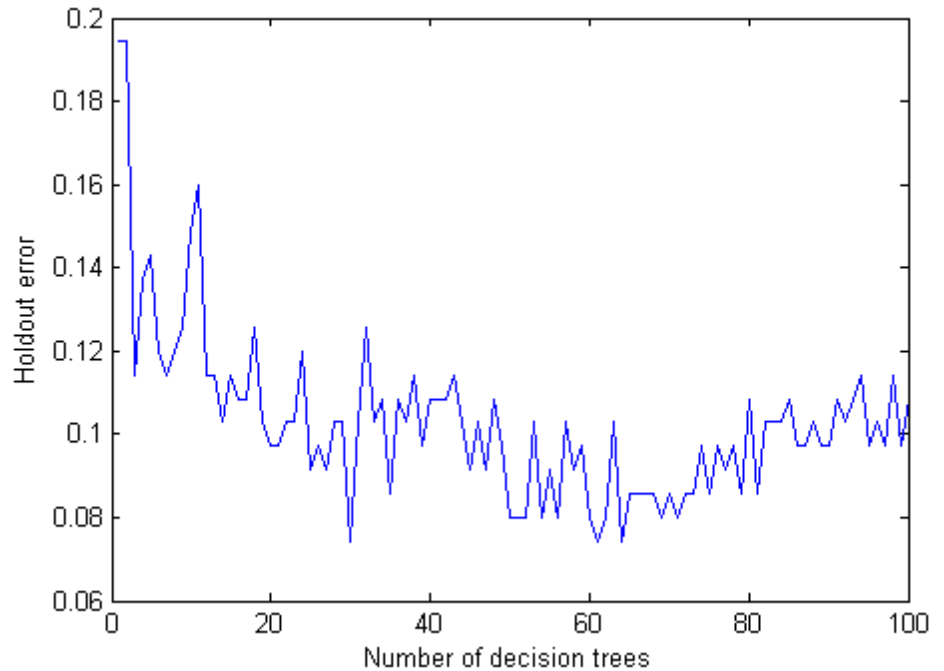
predict(rens,[4 200 150 3000])

ans =
    20.4982
```

---

Train and estimate the generalization error on a holdout sample:

```
load ionosphere;
ada = fitensemble(X,Y,'AdaBoostM1',100,'Tree',...
    'holdout',0.5);
plot(kfoldLoss(ada,'mode','cumulative'));
xlabel('Number of decision trees');
ylabel('Holdout error');
```



## Algorithms

For details of boosting and bagging algorithms, see “Ensemble Algorithms” on page 16-146.

## See Also

`ClassificationEnsemble` | `RegressionEnsemble` |  
`ClassificationBaggedEnsemble` | `RegressionBaggedEnsemble`  
| `ClassificationPartitionedEnsemble`  
| `RegressionPartitionedEnsemble` |  
`ClassificationDiscriminant.template` |  
`ClassificationKNN.template` | `ClassificationTree.template`  
| `RegressionTree.template`

## How To

- “Supervised Learning (Machine Learning) Workflow and Algorithms” on page 16-2

- “Ensemble Methods” on page 16-59

**Purpose**

Fit nonlinear regression model

**Syntax**

```
mdl = fitnlm(ds,modelfun,beta0)
mdl = fitnlm(X,y,modelfun,beta0)
mdl = fitnlm( ___,modelfun,beta0,Name,Value)
```

**Description**

`mdl = fitnlm(ds,modelfun,beta0)` fits the model specified by `modelfun` to variables in the dataset array `ds`, and returns the nonlinear model `mdl`.

`fitnlm` estimates model coefficients using an iterative procedure starting from the initial values in `beta0`.

`mdl = fitnlm(X,y,modelfun,beta0)` fits a nonlinear regression model using the column vector `y` as a response variable and the columns of the matrix `X` as predictor variables.

`mdl = fitnlm( ___,modelfun,beta0,Name,Value)` fits a nonlinear regression model with additional options specified by one or more `Name,Value` pair arguments.

**Input Arguments****ds - Input data**

dataset array

Input data, specified as a dataset array. If you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables and response variable must be numeric.

You specify the response and predictor names in your model string. If you do not provide a model string, you can set a different column as the response variable by using the `ResponseVar` name-value pair argument. You can select a subset of the columns as predictors by using the `PredictorVars` name-value pair argument.

**Data Types**

single | double | logical

## **X - Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

### **Data Types**

single | double | logical

## **y - Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### **Data Types**

single | double

## **modelfun - Functional form of the model**

function handle | string of the form 'y ~  
 $f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ '

Functional form of the model, specified as either of the following.

- Function handle `@modelfun` or `@(b,x)modelfun`, where
  - $b$  is a coefficient vector with the same number of elements as `beta0`.
  - $x$  is a matrix with the same number of columns as  $X$  or the number of predictor variable columns of `ds`.

`modelfun(b,x)` returns a column vector that contains the same number of rows as  $x$ . Each row of the vector is the result of evaluating `modelfun` on the corresponding row of  $x$ . In other words, `modelfun` is a vectorized function, one that operates on all data rows and returns all evaluations in one function call.



- String of the form  $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ , where  $f$  represents a scalar function of the scalar coefficient variables  $b_1, \dots, b_j$  and the scalar data variables  $x_1, \dots, x_k$ .

### **beta0 - Coefficients**

numeric vector

Coefficients for the nonlinear model, specified as a numeric vector. NonLinearModel starts its search for optimal coefficients from beta0.

### **Data Types**

single | double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

**Example:** 'ErrorModel', 'combined', 'Exclude', 2, 'Options', opt specifies the error model as the combined model, excludes the second observation from the fit, and uses the options defined in the structure opt to control the iterative fitting procedure.

### **'CoefficientNames' - Names of the model coefficients**

{ 'b1', 'b2', ..., 'bk' } (default) | cell array of strings

Names of the model coefficients, specified as a cell array of strings.

### **Data Types**

char

### **'ErrorModel' - Form of the error variance model**

'constant' (default) | 'proportional' | 'combined'

Form of the error variance model, specified as one of the following. Each model defines the error using a standard mean-zero and unit-variance variable  $e$  in combination with independent components: the function value  $f$ , and one or two parameters  $a$  and  $b$

'constant' (default)

$$y = f + ae$$

'proportional'

$$y = f + bfe$$

'combined'

$$y = f + (a + b|f|)e$$

The only allowed error model when using `Weights` is 'constant'.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than 'constant'.

---

**Example:** 'ErrorModel', 'proportional'

### 'ErrorParameters' - Initial estimates of the error model parameters

numeric array

Initial estimates of the error model parameters for the chosen `ErrorModel`, specified as a numeric array.

Error Model	Parameters	Default Values
'constant'	$a$	1
'proportional'	$b$	1
'combined'	$a, b$	[1,1]

You can only use the 'constant' error model when using `Weights`.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than 'constant'.

---

For example, if 'ErrorModel' has the value 'combined', you can specify the starting value 1 for  $a$  and the starting value 2 for  $b$  as follows.

**Example:** 'ErrorParameters',[1,2]

### Data Types

single | double

### 'Exclude' - Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

### Data Types

single | double | logical

### 'Options' - Options for controlling the iterative fitting procedure

[] (default) | structure

Options for controlling the iterative fitting procedure, specified as a structure created by `statset`. The relevant fields are the nonempty fields in the structure returned by the call `statset('fitnlm')`.

Option	Meaning	Default
DerivStep	Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics Toolbox function using the options structure.	$\text{eps}^{(1/3)}$
Display	Amount of information displayed by the fitting algorithm. <ul style="list-style-type: none"> <li>• 'off' — Displays no information.</li> <li>• 'final' — Displays the final output.</li> <li>• 'iter' — Displays iterative output to the Command Window.</li> </ul>	'off'
FunValCheck	String indicating to check for invalid values, such as NaN or Inf, from the model function.	'on'
MaxIter	Maximum number of iterations allowed. Positive integer.	200
RobustWgtFun	Weight function for robust fitting. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. If you use a function handle, give a Tune constant. See “Robust Options” on page 21-1143	[]
Tune	Tuning constant used in robust fitting to normalize the residuals before applying the weight function. A positive scalar. Required if the weight function is specified as a function handle.	See “Robust Options” on page 21-1143 for the default, which depends on RobustWgtFun.

Option	Meaning	Default
ToIFun	Termination tolerance for the objective function value. Positive scalar.	1e-8
ToIX	Termination tolerance for the parameters. Positive scalar.	1e-8

### Data Types

struct

#### 'PredictorVars' - Predictor variables

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array `ds`, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the `ds` dataset array, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `ds` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** 'PredictorVars',[2,3]

**Example:** 'PredictorVars',logical([0 1 1 0 0 0])

### Data Types

single | double | logical | cell

#### 'ResponseVar' - Response variable

last column in dataset array `ds` (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a string of the variable

name in the dataset array `ds`, or a logical or numeric index vector indicating which column is the response variable.

If you supply a model string, it specifies the response variable. Otherwise, when fitting a dataset array, `'ResponseVar'` indicates which dataset variable `fitnlm` should use as the response.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

**Example:** `'ResponseVar','yield'`

**Example:** `'ResponseVar',[4]`

**Example:** `'ResponseVar',logical([0 0 0 1 0 0])`

### Data Types

`single` | `double` | `logical` | `char`

### 'VarNames' - Names of variables in fit

`{'x1','x2',...,'xn','y'}` (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings including the names for the columns of `X` first, and the name for the response variable `y` last.

`'VarNames'` is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

### Example:

`'VarNames',{'Horsepower','Acceleration','Model_Year','MPG'}`

### Data Types

`cell`

### 'Weights' - Observation weights

`ones(n,1)` (default) | vector of nonnegative scalar values | function handle

Observation weights, specified as a vector of nonnegative scalar values or function handle.

- If you specify a vector, then it must have  $n$  elements, where  $n$  is the number of rows in `ds` or `y`.
- If you specify a function handle, then the function must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights, `W`, `NonLinearModel` estimates the error variance at observation `i` by  $\text{MSE} * (1/W(i))$ , where `MSE` is the mean squared error.

### Data Types

single | double | function\_handle

## Output Arguments

### `mdl` - Nonlinear model

`NonLinearModel` object

Nonlinear model representing a least-squares fit of the response to the data, returned as a `NonLinearModel` object.

If the `Options` structure contains a nonempty `RobustWgtFun` field, the model is not a least-squares fit, but uses the `RobustWgtFun` robust fitting function.

For properties and methods of the nonlinear model object, `mdl`, see the `NonLinearModel` class page.

## Examples

### Nonlinear Model from a Dataset Array

Create a nonlinear model for auto mileage based on the `carbig` data.

Load the data and create a nonlinear model.

```
load carbig
ds = dataset(Horsepower,Weight,MPG);
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = fitnlm(ds,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```

```
MPG ~ b1 + b2*Horsepower^b3 + b4*Weight^b5
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

```
Number of observations: 392, Error degrees of freedom: 387
```

```
Root Mean Squared Error: 3.96
```

```
R-Squared: 0.745, Adjusted R-Squared 0.743
```

```
F-statistic vs. constant model: 283, p-value = 1.79e-113
```

## Nonlinear Model from Matrix Data

Create a nonlinear model for auto mileage based on the carbig data.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = fitnlm(X,y,modelfun,beta0)
```

```
mdl =
```

```
Nonlinear regression model:
```



$$y \sim b_1 + b_2 * x_1^{b_3} + b_4 * x_2^{b_5}$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

Number of observations: 392, Error degrees of freedom: 387  
 Root Mean Squared Error: 3.96  
 R-Squared: 0.745, Adjusted R-Squared 0.743  
 F-statistic vs. constant model: 283, p-value = 1.79e-113

### Adjust Fitting Options in the Nonlinear Model

Create a nonlinear model for auto mileage based on the `carbig` data. Strive for more accuracy by lowering the `TolFun` option, and observe the iterations by setting the `Display` option.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
```

Create options to lower `TolFun` and to report iterative display, and create a model using the options.

```
opts = statset('Display','iter','TolFun',1e-10);
mdl = fitnlm(X,y,modelfun,beta0,'Options',opts);
```

Iteration	SSE	Norm of Gradient	Norm of Step
-----------	-----	---------------------	-----------------

```
-----  
      0      1.82248e+06  
      1      678600      788810      1691.07  
      2      616716      6.12739e+06      45.4738  
  
%%%      Many iterations deleted      %%%  
  
      122      6068.48      1.56393      0.629325  
      123      6068.48      1.13809      0.432543  
      124      6068.48      0.295962      0.297511  
  
Iterations terminated: relative change in SSE less than OPTIONS.TolFun
```

## Specify Nonlinear Regression Using Model String Syntax

Specify a nonlinear regression model for estimation using a function handle or model string syntax.

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Use a function handle to specify the Hougen-Watson model for the rate data.

```
mdl = fitnlm(X,y,@hougen,beta0)
```

```
mdl =
```

```
Nonlinear regression model:  
y ~ hougen(b,X)
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654

b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
 Root Mean Squared Error: 0.193  
 R-Squared: 0.999, Adjusted R-Squared 0.998  
 F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

Alternatively, you can use a string expression to specify the  
 Hougen-Watson model for the rate data.

```
myfun = 'y~(b1*x2-x3/b5)/(1+b2*x1+b3*x2+b4*x3)';
mdl2 = fitnlm(X,y,myfun,beta0)
```

mdl2 =

Nonlinear regression model:

$$y \sim (b1*x2 - x3/b5)/(1 + b2*x1 + b3*x2 + b4*x3)$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86701	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075157	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
 Root Mean Squared Error: 0.193  
 R-Squared: 0.999, Adjusted R-Squared 0.998  
 F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

## Estimate Nonlinear Regression Using Robust Fitting Options

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp\{-b_3 x\} + \varepsilon,$$

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x)(b(1)+b(2)*exp(-b(3)*x));
```

```
rng('default') % for reproducibility
b = [1;3;2];
x = exprnd(2,100,1);
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

Set robust fitting options.

```
opts = statset('nlinfit');
opts.RobustWgtFun = 'bisquare';
```

Fit the nonlinear model using the robust fitting options. Here, use a string expression to specify the model.

```
b0 = [2;2;2];
modelstr = 'y ~ b1 + b2*exp(-b3*x)';
```

```
mdl = fitnlm(x,y,modelstr,b0,'Options',opts)
```

```
mdl =
```

```
Nonlinear regression model (robust fit):
y ~ b1 + b2*exp( - b3*x)
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.0218	0.07202	14.188	2.1344e-25
b2	3.6619	0.25429	14.401	7.974e-26

```
b3      2.9732      0.38496      7.7232      1.0346e-11
```

```
Number of observations: 100, Error degrees of freedom: 97
Root Mean Squared Error: 0.501
R-Squared: 0.807, Adjusted R-Squared 0.803
F-statistic vs. constant model: 203, p-value = 2.34e-35
```

### Fit Nonlinear Regression Model Using Weights Function Handle

Load sample data.

```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Specify a function handle for observation weights. The function accepts the model fitted values as input, and returns a vector of weights.

```
a = 1; b = 1;
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
```

Fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
mdl = fitnlm(X,y,@hougen,beta0,'Weights',weights)
```

```
mdl =
```

```
Nonlinear regression model:
y ~ hougen(b,X)
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	0.83085	0.58224	1.427	0.19142
b2	0.04095	0.029663	1.3805	0.20477

b3	0.025063	0.019673	1.274	0.23842
b4	0.080053	0.057812	1.3847	0.20353
b5	1.8261	1.281	1.4256	0.19183

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 0.037  
R-Squared: 0.998, Adjusted R-Squared 0.998  
F-statistic vs. zero model: 1.14e+03, p-value = 3.49e-11

## Nonlinear Regression Model Using Nonconstant Error Model

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error variance model.

```
mdl = fitnlm(X,y,@hougen,beta0,'ErrorModel','combined')
```

```
mdl =
```

Nonlinear regression model:  
y ~ hougen(b,X)

Estimated Coefficients:

	Estimate	SE	tStat	pValue
b1	1.2526	0.86702	1.4447	0.18654
b2	0.062776	0.043561	1.4411	0.18753
b3	0.040048	0.030885	1.2967	0.23089
b4	0.11242	0.075158	1.4957	0.17309
b5	1.1914	0.83671	1.4239	0.1923

Number of observations: 13, Error degrees of freedom: 8  
 Root Mean Squared Error: 1.27  
 R-Squared: 0.999, Adjusted R-Squared 0.998  
 F-statistic vs. zero model: 3.91e+03, p-value = 2.54e-13

## Definitions

### Robust Options

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985
[]	No robust fitting	—

## Algorithms

fitnlm uses the same fitting algorithm as nlinfit.

## References

[1] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

[2] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.

[3] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.

## See Also

NonLinearModel | nlinfit

## Related Examples

- "Examine Quality and Adjust the Fitted Model" on page 11-8
- "Predict or Simulate Responses to New Data" on page 11-12
- "Nonlinear Regression Workflow" on page 11-16

## Concepts

- "Nonlinear Regression" on page 11-2



## Purpose

Fit linear mixed-effects model using design matrices

## Syntax

```
lme = LinearMixedModel.fitmatrix(X,y,Z,[])  
lme = LinearMixedModel.fitmatrix(X,y,Z,G)  
lme = LinearMixedModel.fitmatrix( __,Name,Value)
```

## Description

`lme = LinearMixedModel.fitmatrix(X,y,Z,[])` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix or matrices in `Z`.

`[]` implies that there is one group. That is, the grouping variable `G` is `ones(n,1)`, where `n` is the number of observations. Using `LinearMixedModel.fitmatrix(X,Y,Z,[])` without a specified covariance pattern most likely will result in a non-identifiable model. This syntax is recommended only if you build the grouping information into the random effects design `Z` and specify a covariance pattern for the random effects using 'CovariancePattern' name-value pair argument.

`lme = LinearMixedModel.fitmatrix(X,y,Z,G)` creates a linear mixed-effects model of the responses `y` using the fixed-effects design matrix `X` and random-effects design matrix `Z` or matrices in `Z`, and the grouping variable or variables in `G`.

`lme = LinearMixedModel.fitmatrix( __,Name,Value)` also creates a linear mixed-effects model with additional options specified by one or more `Name,Value` pair arguments, using any of the previous input arguments.

For example, you can specify the names of the response, predictor, and grouping variables. You can also specify the covariance pattern, fitting method, or the optimization algorithm.

## Tips

- Use the function `fitlmematrix` instead of `LinearMixedModel.fitmatrix`.

# LinearMixedModel.fitmatrix

---

- If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix`.

## Input Arguments

### **X - Fixed-effects design matrix**

*n*-by-*p* matrix

Fixed-effects design matrix, specified as an *n*-by-*p* matrix, where *n* is the number of observations, and *p* is the number of fixed-effects predictor variables. Each row of X corresponds to one observation, and each column of X corresponds to one variable.

### **Data Types**

single | double

### **y - Response values**

*n*-by-1 vector

Response values, specified as an *n*-by-1 vector, where *n* is the number of observations.

### **Data Types**

single | double

### **Z - Random-effects design**

*n*-by-*q* matrix | cell array of *R* *n*-by-*q*(*r*) matrices, *r* = 1, 2, ..., *R*

Random-effects design, specified as either of the following.

- If there is one random-effects term in the model, then Z must be an *n*-by-*q* matrix, where *n* is the number of observations and *q* is the number of variables in the random-effects term.
- If there are *R* random-effects terms, then Z must be a cell array of length *R*. Each cell of Z contains an *n*-by-*q*(*r*) design matrix Z{*r*}, *r* = 1, 2, ..., *R*, corresponding to each random-effects term. Here, *q*(*r*) is the number of random effects term in the *r*th random effects design matrix, Z{*r*}.

### **Data Types**

single | double | cell

## **G - Grouping variable or variables**

*n*-by-1 vector | cell array of *R* *n*-by-1 vectors

Grouping variable or variables, specified as either of the following.

- If there is one random-effects term, then **G** must be an *n*-by-1 vector corresponding to a single grouping variable with *M* levels or groups.

**G** can be a categorical vector, numeric vector, character array, or cell array of strings.

- If there are multiple random-effects terms, then **G** must be a cell array of length *R*. Each cell of **G** contains a grouping variable **G**{*r*}, *r* = 1, 2, ..., *R*, with *M*(*r*) levels.

**G**{*r*} can be a categorical vector, numeric vector, character array, or cell array of strings.

## **Data Types**

single | double | char | cell

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**,**Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**,**Value1**,...,**NameN**,**ValueN**.

## **'FixedEffectPredictors' - Names of columns in fixed-effects design matrix**

{'x1','x2',..., 'xP'} (default) | cell array of length *p*

Names of columns in the fixed-effects design matrix **X**, specified as the comma-separated pair consisting of **'FixedEffectPredictors'** and a cell array of length *p*.

For example, if you have a constant term and two predictors, say **TimeSpent** and **Gender**, where **Female** is the reference level for **Gender**, as the fixed effects, then you can specify the names of

# LinearMixedModel.fitmatrix

---

your fixed effects in the following way. `Gender_Male` represents the dummy variable you must create for category `Male`. You can choose different names for these variables.

**Example:**

```
'FixedEffectPredictors',{ 'Intercept', 'TimeSpent', 'Gender_Male' },
```

**Data Types**

cell

**'RandomEffectPredictors' - Names of columns in random-effects design matrix or cell array**

cell array of length  $q$  | cell array of length  $R$  with elements of length  $q(r)$ ,  $r = 1, 2, \dots, R$

Names of columns in the random-effects design matrix or cell array  $Z$ , specified as the comma-separated pair consisting of `'RandomEffectPredictors'` and either of the following:

- A cell array of length  $q$  when  $Z$  is an  $n$ -by- $q$  design matrix. In this case, the default is `{'z1', 'z2', ..., 'zQ'}`.
- A cell array of length  $R$ , when  $Z$  is a cell array of length  $R$  with each element  $Z\{r\}$  of length  $q(r)$ ,  $r = 1, 2, \dots, R$ . In this case, the default is `{'z11', 'z12', ..., 'z1Q(1)'}, ..., {'zr1', 'zr2', ..., 'zrQ(r)'}`.

For example, suppose you have correlated random effects for intercept and a variable named `Acceleration`. Then, you can specify the random-effects predictor names as follows.

**Example:**

```
'RandomEffectPredictors',{ 'Intercept', 'Acceleration' }
```

If you have two random effects terms, one for the intercept and the variable `Acceleration` grouped by variable `g1`, and the second for the intercept, grouped by the variable `g2`, then you specify the random-effects predictor names as follows.

**Example:**

```
'RandomEffectPredictors',{ {'Intercept', 'Acceleration'}, {'Intercept'}}
```

## Data Types

cell

### 'ResponseVarName' - Name of response variable

'y' (default) | string

Name of response variable, specified as the comma-separated pair consisting of 'ResponseVarName' and a string.

For example, if your response variable name is `score`, then you can specify it as follows.

**Example:** 'ResponseVarName', 'score'

## Data Types

char

### 'RandomEffectGroups' - Names of random effects grouping variables

'g' or {'g1', 'g2', ..., 'gR'} (default) | string | cell array of strings

Names of random effects grouping variables, specified as the comma-separated pair 'RandomEffectGroups' and either of the following:

- String — If there is only one random-effects term, that is, if  $G$  is a vector, then the value of 'RandomEffectGroups' is a string containing the name for the grouping variable  $G$ . The default is 'g'.
- Cell array of strings — If there are multiple random-effects terms, that is, if  $G$  is a cell array of length  $R$ , then the value of 'RandomEffectGroups' is a cell array of length  $R$ , where each cell contains the name for the grouping variable  $G\{r\}$ . The default is {'g1', 'g2', ..., 'gR'}.

For example, if you have two random-effects terms, `z1` and `z2`, grouped by the grouping variables `sex` and `subject`, then you can specify the names of your grouping variables as follows.

**Example:** 'RandomEffectGroups', {'sex', 'subject'}

# LinearMixedModel.fitmatrix

---

## Data Types

char | cell

### 'CovariancePattern' - Pattern of covariance matrix

'FullCholesky' (default) | string | square symmetric logical matrix | cell array of strings or logical matrices

Pattern of the covariance matrix of the random effects, specified as the comma-separated pair consisting of 'CovariancePattern' and a string, a square symmetric logical matrix, or a cell array of strings or logical matrices.

If there are  $R$  random-effects terms, then the value of 'CovariancePattern' must be a cell array of length  $R$ , where each element  $r$  of this cell array specifies the pattern of the covariance matrix of the random-effects vector associated with the  $r$ th random-effects term. The options for each element follow.

'FullCholesky'

Default. Full covariance matrix using the Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Full'

Full covariance matrix, using the log-Cholesky parameterization. `fitlme` estimates all elements of the covariance matrix.

'Diagonal'

Diagonal covariance matrix. That is, off-diagonal elements of the covariance matrix are constrained to be 0.

$$\begin{pmatrix} \sigma_{b1}^2 & 0 & 0 \\ 0 & \sigma_{b2}^2 & 0 \\ 0 & 0 & \sigma_{b3}^2 \end{pmatrix}$$

'Isotropic'

Diagonal covariance matrix with equal variances. That is, off-diagonal elements of the covariance matrix are constrained to be 0, and the diagonal elements are constrained to be equal. For example, if there are three random-effects terms with an isotropic covariance structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_b^2 & 0 & 0 \\ 0 & \sigma_b^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{pmatrix}$$

where  $\sigma_b^2$  is the common variance of the random-effects terms.

# LinearMixedModel.fitmatrix

---

'CompSymm'

Compound symmetry structure. That is, common variance along diagonals and equal correlation between all random effects. For example, if there are three random-effects terms with a covariance matrix having a compound symmetry structure, this covariance matrix looks like

$$\begin{pmatrix} \sigma_{b1}^2 & \sigma_{b1,b2} & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1}^2 & \sigma_{b1,b2} \\ \sigma_{b1,b2} & \sigma_{b1,b2} & \sigma_{b1}^2 \end{pmatrix}$$

where  $\sigma_{b1}^2$  is the common variance of the random-effects terms and  $\sigma_{b1,b2}$  is the common covariance between any two random-effects term .

PAT

Square symmetric logical matrix. If 'CovariancePattern' is defined by the matrix PAT, and if  $PAT(a,b) = \text{false}$ , then the (a,b) element of the corresponding covariance matrix is constrained to be 0.

**Example:** 'CovariancePattern', 'Diagonal'

**Example:** 'CovariancePattern', {'Full', 'Diagonal'}

**'FitMethod' - Method for estimating parameters**

'ML' (default) | 'REML'



Method for estimating parameters of the linear mixed-effects model, specified as the comma-separated pair consisting of 'FitMethod' and either of the following.

'ML'	Default. Maximum likelihood estimation
'REML'	Restricted maximum likelihood estimation

**Example:** 'FitMethod', 'REML'

## 'Weights' - Observation weights

vector of scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$ , where  $n$  is the number of observations.

## Data Types

single | double

## 'Exclude' - Indices for rows to exclude

use all rows without NaNs (default) | vector of integer or logical values

Indices for rows to exclude from the linear mixed-effects model in the data, specified as the comma-separated pair consisting of 'Exclude' and a vector of integer or logical values.

For example, you can exclude the 13th and 67th rows from the fit as follows.

**Example:** 'Exclude', [13,67]

## Data Types

single | double | logical

## 'DummyVarCoding' - Coding to use for dummy variables

'reference' (default) | 'effects' | 'full'

# LinearMixedModel.fitmatrix

---

Coding to use for dummy variables created from the categorical variables, specified as the comma-separated pair consisting of 'DummyVarCoding' and one of the following.

'reference'	Default. Coefficient for first category set to 0.
'effects'	Coefficients sum to 0.
'full'	One dummy variable for each category.

**Example:** 'DummyVarCoding','effects'

## 'Optimizer' - Optimization algorithm

'quasinevton' (default) | 'fminunc'

Optimization algorithm, specified as the comma-separated pair consisting of 'Optimizer' and either of the following.

'quasinevton'	Default. Uses a trust region based quasi-Newton optimizer. Change the options of the algorithm using <code>statset('LinearMixedModel')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>statset('LinearMixedModel')</code> .
'fminunc'	You must have Optimization Toolbox to specify this option. Change the options of the algorithm using <code>optimoptions('fminunc')</code> . If you don't specify the options, then <code>LinearMixedModel</code> uses the default options of <code>optimoptions('fminunc')</code> .

with 'Algorithm' set to  
'quasi-newton'.

**Example:** 'Optimizer','fminunc'

## 'OptimizerOptions' - Options for optimization algorithm

structure returned by `statset` | object returned by `optimoptions`

Options for the optimization algorithm, specified as the comma-separated pair consisting of 'OptimizerOptions' and a structure returned by `statset('LinearMixedModel')` or an object returned by `optimoptions('fminunc')`.

- If 'Optimizer' is 'fminunc', then use `optimoptions('fminunc')` to change the options of the optimization algorithm. See `optimoptions` for the options 'fminunc' uses. If 'Optimizer' is 'fminunc' and you do not supply 'OptimizerOptions', then the default for `LinearMixedModel` is the default options created by `optimoptions('fminunc')` with 'Algorithm' set to 'quasi-newton'.
- If 'Optimizer' is 'quasnewton', then use `statset('LinearMixedModel')` to change the optimization parameters. If you don't change the optimization parameters, then `LinearMixedModel` uses the default options created by `statset('LinearMixedModel')`:

The 'quasnewton' optimizer uses the following fields in the structure created by `statset('LinearMixedModel')`.

## 'TolFun' - Relative tolerance on gradient of objective function

1e-6 (default) | positive scalar value

Relative tolerance on the gradient of the objective function, specified as a positive scalar value.

## 'TolX' - Absolute tolerance on step size

1e-12 (default) | positive scalar value

Absolute tolerance on the step size, specified as a positive scalar value.

### 'MaxIter' - Maximum number of iterations allowed

10000 (default) | positive scalar value

Maximum number of iterations allowed, specified as a positive scalar value.

### 'Display' - Level of display

'off' (default) | 'iter' | 'final'

Level of display, specified as one of 'off', 'iter', or 'final'.

### 'StartMethod' - Method to start iterative optimization

'default' (default) | 'random'

Method to start iterative optimization, specified as the comma-separated pair consisting of 'StartMethod' and either of the following.

'default'	Default. An internally defined default value.
'random'	A random initial value.

**Example:** 'StartMethod', 'random'

### 'Verbose' - Indicator to display optimization process on screen

false (default) | true

Indicator to display the optimization process on screen, specified as the comma-separated pair consisting of 'Verbose' and either false or true. Default is false.

The setting for 'Verbose' overrides the field 'Display' in 'OptimizerOptions'.

**Example:** 'Verbose', true

## 'CheckHessian' - Indicator to check positive definiteness of Hessian

false (default) | true

Indicator to check the positive definiteness of the Hessian of the objective function with respect to unconstrained parameters at convergence, specified as the comma-separated pair consisting of 'CheckHessian' and either false or true. Default is false.

Specify 'CheckHessian' as true to verify optimality of the solution or to determine if the model is overparameterized in the number of covariance parameters.

**Example:** 'CheckHessian',true

## Output Arguments

### lme - Linear mixed-effects model

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

For properties and methods of this object, see LinearMixedModel.

## Examples

### No Grouping Variable Specified

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, where MPG is the response, weight is the predictor variable, and the intercept varies by model year. First, define the design matrices. Then, fit the model using the specified design matrices.

```
y = MPG;  
X = [ones(size(Weight)), Weight];  
Z = ones(size(y));  
lme = LinearMixedModel.fitmatrix(X,y,Z,Model_Year)
```

```
lme =
```

# LinearMixedModel.fitmatrix

---

Linear mixed-effects model fit by ML

Model information:

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

Formula:

$y \sim x1 + x2 + (z11 \mid g1)$

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-33
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

Random effects covariance parameters (95% CIs):

Group: g1 (3 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.5421

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

Now fit the same model by building the grouping into the Z matrix.

```
Z = double([Model_Year==70, Model_Year==76, Model_Year==82]);  
lme = LinearMixedModel.fitmatrix(X,y,Z,[],...  
'Covariancepattern','Isotropic')
```

lme =

Linear mixed-effects model fit by ML

Model information:

Number of observations	94
Fixed effects coefficients	2
Random effects coefficients	3
Covariance parameters	2

Formula:

$y \sim x1 + x2 + (z11 + z12 + z13 | g1)$

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
486.09	496.26	-239.04	478.09

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'x1'	43.575	2.3038	18.915	92	1.8371e-30
'x2'	-0.0067097	0.0004242	-15.817	92	5.5373e-28

Random effects covariance parameters (95% CIs):

Group: g1 (1 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'z11'	'z11'	'std'	3.301	1.4448	7.542

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	2.8997	2.5075	3.3532

## Longitudinal Study with Covariate

Navigate to a folder containing sample data.

`cd(matlabroot)`

# LinearMixedModel.fitmatrix

---

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs (A, B, C, D) and their weight loss is recorded over six two-week time periods. This is simulated data.

Define `Subject` and `Program` as categorical variables. Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, week, and the interaction between the week and type of program as the fixed effects. The intercept and coefficient of week vary by subject.

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 Week_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i + \beta_5 I[PD]_i \\ + \beta_6 (Week_i * I[PB]_i) + \beta_7 (Week_i * I[PC]_i) + \beta_8 (Week_i * I[PD]_i) \\ + b_{0m} + b_{1m} Week_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .  $\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{0m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[.]$  is a dummy variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:  $b_{0m} \sim N(0, \sigma^2_0)$ ,  $b_{1m} \sim N(0, \sigma^2_1)$ , and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
Subject = nominal(Subject);  
Program = nominal(Program);  
D = dummyvar(Program); % Create dummy variables for Program  
X = [ones(120,1), InitialWeight, D(:,2:4), Week, ...  
     D(:,2).*Week, D(:,3).*Week, D(:,4).*Week];  
Z = [ones(120,1), Week];  
G = Subject;
```



Since the model has an intercept, you only need the dummy variables for programs B, C, and D. This is also known as the 'reference' method of coding dummy variables.

Fit the model using `LinearMixedModel.fitmatrix` with the defined design matrices and grouping variables.

```
lme = LinearMixedModel.fitmatrix(X,y,Z,G,'FixedEffectPredictors',...  
{'Intercept','InitWeight','PrgB','PrgC','PrgD','Week','Week_PrgB','Wee'  
'RandomEffectPredictors',{{'Intercept','Week'}},'RandomEffectGroups',
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	120
Fixed effects coefficients	9
Random effects coefficients	40
Covariance parameters	4

Formula:

Linear Mixed Formula with 10 predictors.

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-22.981	13.257	24.49	-48.981

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'Intercept'	0.66105	0.25892	2.5531	111
'InitWeight'	0.0031879	0.0013814	2.3078	111
'PrgB'	0.36079	0.13139	2.746	111
'PrgC'	-0.033263	0.13117	-0.25358	111
'PrgD'	0.11317	0.13132	0.86175	111
'Week'	0.1732	0.067454	2.5677	111

# LinearMixedModel.fitmatrix

---

'Week_PrgB'	0.038771	0.095394	0.40644	111	0.
'Week_PrgC'	0.030543	0.095394	0.32018	111	0.
'Week_PrgD'	0.033114	0.095394	0.34713	111	0.

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.18407	0.122
'Week'	'Intercept'	'corr'	0.66841	0.210
'Week'	'Week'	'std'	0.15033	0.110

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10261	0.087882	0.11981

The  $p$ -values 0.0228 and 0.0115 indicate significant effects of the initial weights of the subjects and the time factor in the amount of weight lost. The weight loss of subjects that are in program B is significantly different relative to the weight loss of subjects that are in program A. The lower and upper limits of the covariance parameters for the random effects do not include zero, thus they seem significant. You can also test the significance of the random-effects using the `compare` method.

## Random-Intercept Model

Load the sample data.

```
load flu
```

`flu` dataset array has a `Date` variable, and 10 variables for estimated influenza rates (in 9 different regions, estimated from Google searches, plus a nationwide estimate from the CDC).

To fit a linear-mixed effects model, where the influenza rates are the responses, combine the nine columns corresponding to the regions into a tall array that has a single response variable, `FluRate`, and a nominal variable, `Region`, the nationwide estimate `WtdILI`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...  
            'IndVarName','Region');  
flu2.Date = nominal(flu2.Date);
```

Define the design matrices for a random-intercept linear mixed-effects model, where the intercept varies by Date. The corresponding model is

$$y_{im} = \beta_0 + \beta_1 WtdILL_i + b_{0m} + \varepsilon_{im}, \quad i = 1, 2, \dots, 468, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable Date.  $b_{0m}$  is the random effect for level  $m$  of the grouping variable Date and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b_{0m} \sim N(0, \sigma_{FR}^2)$  and the error term has the distribution,  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
y = flu2.FluRate;  
X = [ones(468,1) flu2.WtdILL];  
Z = [ones(468,1)];  
G = flu2.Date;
```

Fit the linear mixed-effects model.

```
lme = LinearMixedModel.fitmatrix(X,y,Z,G,'FixedEffectPredictors',{'Intercept'},  
'RandomEffectPredictors',{'Intercept'},'RandomEffectGroups',{'Date'});
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	468
Fixed effects coefficients	2
Random effects coefficients	52
Covariance parameters	2

Formula:

```
y ~ Intercept + NationalRate + (Intercept | Date)
```

# LinearMixedModel.fitmatrix

---

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
286.24	302.83	-139.12	278.24

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	0.16385	0.057525	2.8484	466	0.004
'NationalRate'	0.7236	0.032219	22.459	466	3.0502

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.17146	0.13227

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.30201	0.28217	0.32324

The confidence limits of the standard deviation of the random-effects term  $\sigma^2_{b_{0m}}$ , do not include zero (0.13227, 0.22226), which indicates that the random-effects term is significant. You can also test the significance of the random-effects using compare method.

The estimated value of an observation is the sum of the fixed-effects values and value of the random effect at the grouping variable level corresponding to that observation. For example, the estimated flu rate for observation 28

$$\begin{aligned}\hat{y}_{28} &= \hat{\beta}_0 + \hat{\beta}_1 WtdILLI_{28} + \hat{b}_{10/30/2005} \\ &= 0.1639 + 0.7236 * (1.343) + 0.3318 \\ &= 1.46749,\end{aligned}$$

where  $\hat{b}$  is the BLUP of the random effects for the intercept. You can compute this value in the following way.

```
beta = fixedEffects(lme);
```

```
[~,~,STATS] = randomEffects(lme); % compute the random effects statistics
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(2)*flu2.WtdILI(28) + STATS.Estimate(STATS.Level);

y_hat =

    1.4674
```

You can simply display the fitted value using the `fitted(lme)` method.

```
F = fitted(lme);
F(28)

ans =

    1.4674
```

## Randomized-Block Design

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The data set shows the deviations from the target quality characteristic measured from the products each of five operators manufacture over three different shifts, morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The performance measure is the deviations of the quality characteristics from the target value. This is simulated data.

Define the design matrices for a linear mixed-effects model with a random intercept grouped by operator, and shift as the fixed effects. Use the 'effects' contrasts. 'effects' contrasts mean that the

# LinearMixedModel.fitmatrix

---

coefficients sum to zero. You need to create two contrast coded variables in the fixed-effects design matrix,  $X1$  and  $X2$ , where

$$\text{Shift\_Evening} = \begin{cases} 0, & \text{if Morning} \\ 1, & \text{if Evening} \\ -1, & \text{if Night} \end{cases} \quad \text{and} \quad \text{Shift\_Morning} = \begin{cases} 1, & \text{if Morning} \\ 0, & \text{if Evening} \\ -1, & \text{if Night} \end{cases}.$$

The model corresponds to

$$\text{Morning Shift: } QCDev_{im} = \beta_0 + \beta_2 \text{Shift\_Morning}_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 5,$$

$$\text{Evening Shift: } QCDev_{im} = \beta_0 + \beta_1 \text{Shift\_Evening}_i + b_{0m} + \varepsilon_{im},$$

$$\text{Night Shift: } QCDev_{im} = \beta_0 - \beta_1 \text{Shift\_Evening}_i - \beta_2 \text{Shift\_Morning}_i + b_{0m} + \varepsilon_{im},$$

where  $i$  represents the observations, and  $m$  represents the operators,  $i = 1, 2, \dots, 15$ , and  $m = 1, 2, \dots, 5$ . The random effects and the observation error have the following distributions:  $b_{0m} \sim N(0, \sigma^2_{b_{0m}})$  and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
S = shift.Shift;
X1 = (S=='Morning') - (S=='Night');
X2 = (S=='Evening') - (S=='Night');
X = [ones(15,1), X1, X2];
y = shift.QCDev;
Z = ones(15,1);
G = shift.Operator;
```

Fit a linear mixed-effects model using the specified design matrices and restricted maximum likelihood method.

```
lme = LinearMixedModel.fitmatrix(X,y,Z,G,'FitMethod','REML','FixedEffectPredictors',{'Intercept','S_Morning','S_Evening'},'RandomEffectPredictors',{'Intercept'},'RandomEffectGroups',{'Operator'},'DummyVarCoding','effects')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

# LinearMixedModel.fitmatrix

Model information:

Number of observations	15
Fixed effects coefficients	3
Random effects coefficients	5
Covariance parameters	2

Formula:

$y \sim \text{Intercept} + S\_Morning + S\_Evening + (\text{Intercept} \mid \text{Operator})$

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
58.913	61.337	-24.456	48.913

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	pValue
'Intercept'	3.6525	0.94109	3.8812	12	0.00218
'S_Morning'	-0.91973	0.31206	-2.9473	12	0.0122
'S_Evening'	-0.53293	0.31206	-1.7078	12	0.113

Random effects covariance parameters (95% CIs):

Group: Operator (5 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'Intercept'	'Intercept'	'std'	2.0457	0.9	0.9

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.85462	0.52357	1.395

Compute the best linear unbiased predictor (BLUP) estimates of random effects.

`B = randomEffects(lme)`

B =

0.5775  
1.1757

# LinearMixedModel.fitmatrix

---

```
-2.1715  
2.3655  
-1.9472
```

The estimated deviation from the target quality characteristics for the third operator working in the evening shift is

$$\begin{aligned}\hat{y}_{\text{Evening,Operator3}} &= \hat{\beta}_0 + \hat{\beta}_1 \text{Shift\_Evening} + \hat{b}_{03} \\ &= 3.6525 - 0.53293 - 2.1715 \\ &= 0.94807.\end{aligned}$$

You can also display this value in the following way.

```
F = fitted(lme);  
F(shift.Shift=='Evening' & shift.Operator=='3')  
  
ans =  
  
0.9481
```

## Correlated and Uncorrelated Random-Effects Terms

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower and the cylinders, and uncorrelated random-effect for intercept and acceleration grouped by the model year. This model corresponds to

$$\text{MPG}_{im} = \beta_0 + \beta_1 \text{Acc}_i + \beta_2 \text{HP} + b_{0m} + b_{1m} \text{Acc}_{im} + \varepsilon_{im}, \quad m = 1, 2, 3,$$

with the random-effects terms having the following prior distributions:  $b_{0m} \sim N(0, \sigma_0^2)$ , and  $b_{1m} \sim N(0, \sigma_1^2)$ .  $m$  represents the model year.

First, prepare the design matrices for fitting the linear mixed-effects model.



# LinearMixedModel.fitmatrix

```
X = [ones(406,1) Acceleration Horsepower];
Z = {ones(406,1),Acceleration};
G = {Model_Year,Model_Year};
Model_Year = nominal(Model_Year);
```

Now, fit the model using `LinearMixedModel.fitmatrix` with the defined design matrices and grouping variables.

```
lme = LinearMixedModel.fitmatrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept'},{'Acceleration'},'RandomEffectGroups',{'Model_Year'},'M
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

```
Model information:
```

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	3

```
Formula:
```

```
y ~ Intercept + Acceleration + Horsepower + (Intercept | Model_Year)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
2194.5	2218.3	-1091.3	2182.5

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	p
'Intercept'	49.839	2.0518	24.291	389	5
'Acceleration'	-0.58565	0.10846	-5.3995	389	-
'Horsepower'	-0.16534	0.0071227	-23.213	389	-

```
Random effects covariance parameters (95% CIs):
```

# LinearMixedModel.fitmatrix

```
Group: Model_Year (13 Levels)
  Name1      Name2      Type      Estimate      Lower      Upper
  'Intercept' 'Intercept' 'std'      8.0669e-07      NaN
Group: Model_Year (13 Levels)
  Name1      Name2      Type      Estimate
  'Acceleration' 'Acceleration' 'std'      0.18783
Group: Error
  Name      Estimate      Lower      Upper
  'Res Std' 3.7258      3.4698      4.0007
```

The standard deviation of the random effect for the intercept does not seem significant.

Refit the model with potentially correlated random effects for intercept and acceleration. In this case the random-effects terms have the following prior distribution

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( 0, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix} \right),$$

where  $m$  represents the model year.

First prepare the random-effects design matrix and grouping variable.

```
Z = [ones(406,1) Acceleration];
G = Model_Year;
```

```
lme = LinearMixedModel.fitmatrix(X,MPG,Z,G,'FixedEffectPredictors',...
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...
{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'})
```

```
lme =
```

```
Linear mixed-effects model fit by ML
```

Model information:

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	4

Formula:

y ~ Intercept + Acceleration + Horsepower + (Intercept + Acceleration

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
2193.5	2221.3	-1089.7	2179.5

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF	p
'Intercept'	50.133	2.2652	22.132	389	7
'Acceleration'	-0.58327	0.13394	-4.3545	389	1
'Horsepower'	-0.16954	0.0072609	-23.35	389	0

Random effects covariance parameters (95% CIs):

Group: Model\_Year (13 Levels)

Name1	Name2	Type	Estimate
'Intercept'	'Intercept'	'std'	3.3479
'Acceleration'	'Intercept'	'corr'	-0.8797
'Acceleration'	'Acceleration'	'std'	0.3378

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	3.6874	3.4298	3.9644

The confidence intervals for the standard deviations and the correlation between the random effects for intercept and acceleration do not include zeros, hence they seem significant. You can compare these two models using the compare method.

## Specify the Covariance Pattern

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned 4 exercise programs, and their weight loss is recorded over six two-week time periods. This is simulated data.

Define `Subject` and `Program` as categorical variables.

```
Subject = nominal(Subject);
Program = nominal(Program);
```

Create the design matrices for a linear mixed-effects model, with the initial weight, type of program, and week are the fixed effects.

```
D = dummyvar(Program);
X = [ones(120,1), InitialWeight, D(:,2:4), Week];
Z = [ones(120,1) Week];
G = Subject;
```

This model corresponds to

$$y_{im} = \beta_0 + \beta_1 IW_i + \beta_2 Week_i + \beta_3 I[PB]_i + \beta_4 I[PC]_i + \beta_5 I[PD]_i \\ + b_{0m} + b_{1m} Week2_{im} + b_{2m} Week4_{im} + b_{3m} Week6_{im} + b_{4m} Week8_{im} \\ + b_{5m} Week10_{im} + b_{6m} Week12_{im} + \varepsilon_{im},$$

where  $i = 1, 2, \dots, 120$ , and  $m = 1, 2, \dots, 20$ .

$\beta_j$  are the fixed-effects coefficients,  $j = 0, 1, \dots, 8$ , and  $b_{1m}$  and  $b_{1m}$  are random effects.  $IW$  stands for initial weight and  $I[.]$  is a dummy

variable representing a type of program. For example,  $I[PB]_i$  is the dummy variable representing program type B. The random effects and observation error have the following prior distributions:  $b_{0m} \sim N(0, \sigma_0^2)$ ,  $b_{1m} \sim N(0, \sigma_1^2)$ , and  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

Fit the model using `LinearMixedModel.fitmatrix` with the defined design matrices and grouping variables. Assume the repeated observations collected on a subject have common variance along diagonals.

```
lme = LinearMixedModel.fitmatrix(X,y,Z,G,'FixedEffectPredictors',...  
{'Intercept','InitWeight','PrgB','PrgC','PrgD','Week'},...  
'RandomEffectPredictors',{{'Intercept','Week'}},...  
'RandomEffectGroups',{'Subject'},'CovariancePattern','Isotropic')
```

```
lme =
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	120
Fixed effects coefficients	6
Random effects coefficients	40
Covariance parameters	2

Formula:

```
y ~ Intercept + InitWeight + PrgB + PrgC + PrgD + Week + (Intercept
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
-24.783	-2.483	20.391	-40.783

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'Intercept'	0.4208	0.28169	1.4938	114
'InitWeight'	0.0045552	0.0015338	2.9699	114

# LinearMixedModel.fitmatrix

'PrgB'	0.36993	0.12119	3.0525	114	0.0
'PrgC'	-0.034009	0.1209	-0.28129	114	0.0
'PrgD'	0.121	0.12111	0.99911	114	0.0
'Week'	0.19881	0.037134	5.3538	114	4.51

Random effects covariance parameters (95% CIs):

Group: Subject (20 Levels)

Name1	Name2	Type	Estimate	Lower
'Intercept'	'Intercept'	'std'	0.16561	0.1289

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.10272	0.088014	0.11987

## Definitions

### Cholesky Parameterization

One of the assumptions of linear mixed-effects models is that the random effects have the following prior distribution.

$$b \sim N(0, \sigma^2 D(\theta)),$$

where  $D$  is a  $q$ -by- $q$  symmetric and positive semidefinite matrix, parameterized by a variance component vector  $\theta$ ,  $q$  is the number of variables in the random-effects term, and  $\sigma^2$  is the observation error variance. Since the covariance matrix of the random effects,  $D$ , is symmetric, it has  $q(q+1)/2$  free parameters. Suppose  $L$  is the lower triangular Cholesky factor of  $D(\theta)$  such that

$$D(\theta) = L(\theta)L(\theta)^T,$$

then the  $q*(q+1)/2$ -by-1 unconstrained parameter vector  $\theta$  is formed from elements in the lower triangular part of  $L$ . For example, if

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix},$$

then

$$\theta = \begin{bmatrix} L_{11} \\ L_{21} \\ L_{31} \\ L_{22} \\ L_{32} \\ L_{33} \end{bmatrix}.$$

## Log-Cholesky Parameterization

When the diagonal elements of  $L$  in Cholesky parameterization are constrained to be positive, then the solution for  $L$  is unique. Log-Cholesky parameterization is the same as Cholesky parameterization except that the logarithm of the diagonal elements of  $L$  are used to guarantee unique parameterization.

For example, for the 3-by-3 example in Cholesky parameterization, enforcing  $L_{ii} \geq 0$ ,

$$\theta = \begin{bmatrix} \log(L_{11}) \\ L_{21} \\ L_{31} \\ \log(L_{22}) \\ L_{32} \\ \log(L_{33}) \end{bmatrix}.$$

## Alternatives

You can also fit a linear mixed-effects model using `fitlme(ds, formula)`, where `ds` is a dataset array containing the response  $y$ , the predictor variables  $X$ , and the grouping variables, and `formula` is of the form ' $y \sim \text{fixed} + (\text{random}_1 | g_1) + \dots + (\text{random}_R | g_R)$ '.

If your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X, y, Z, G)`.

# LinearMixedModel.fitmatrix

---

## See Also

`fitlmeLinearMixedModel` |



**Purpose** Fitted responses from a linear mixed-effects model

**Syntax**  
`yfit = fitted(lme)`  
`yfit = fitted(lme,Name,Value)`

**Description** `yfit = fitted(lme)` returns the fitted conditional response from the linear mixed-effects model `lme`.

`yfit = fitted(lme,Name,Value)` returns the fitted response from the linear mixed-effects model `lme` with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify if you want to compute the fitted marginal response.

## Input Arguments

### **lme - Linear mixed-effects model**

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'Conditional' - Indicator for conditional response**

`True` (default) | `False`

Indicator for conditional response, specified as the comma-separated pair consisting of 'Conditional' and either of the following.

# LinearMixedModel.fitted

---

True	Contribution from both fixed effects and random effects (conditional)
False	Contribution from only fixed effects (marginal)

**Example:** 'Conditional','False'

## Output Arguments

### **yfit - Fitted response values**

*n*-by-1 vector

Fitted response values, returned as an *n*-by-1 vector, where *n* is the number of observations.

## Examples

### **Compute Fitted Conditional and Marginal Responses**

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google searches, plus a nationwide estimate from the Center for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and region as the predictor variable, combine the nine columns corresponding to the regions into a tall array. The new dataset array, `flu2`, must have the response variable, `FluRate`, the nominal variable, `Region`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10,'NewDataVarName','FluRate',...  
            'IndVarName','Region');  
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with fixed effects for region and a random intercept that varies by Date.

Region is a categorical variable. You can specify the contrasts for categorical variables using the `DummyVarCoding` name-value pair argument when fitting the model. When you do not specify the contrasts, `fitlme` uses the 'reference' contrast by default. Because the model has an intercept, `fitlme` takes the first region, NE, as the reference and creates eight dummy variables representing the other eight regions. For example,  $I[MidAtl]$  is the dummy variable representing the region MidAtl. For details, see “Dummy Indicator Variables” on page 2-55.

The corresponding model is

$$y_{im} = \beta_0 + \beta_1 I[MidAtl]_i + \beta_2 I[ENCentral]_i + \beta_3 I[WNCentral]_i + \beta_4 I[SAtl]_i + \beta_5 I[ESCentral]_i + \beta_6 I[WSCentral]_i + \beta_7 I[Mtn]_i + \beta_8 I[Pac]_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable Date,  $\beta_j$ ,  $j = 0, 1, \dots, 8$ , are the fixed-effects coefficients, with  $\beta_0$  being the coefficient for region NE.  $b_{0m}$  is the random effect for level  $m$  of the grouping variable Date, and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b_{0m} \sim N(0, \sigma_b^2)$  and the error term has the distribution,  $\varepsilon_{im} \sim N(0, \sigma^2)$ .

```
lme = fitlme(flu2, 'FluRate ~ 1 + Region + (1|Date)')
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	468
Fixed effects coefficients	9
Random effects coefficients	52
Covariance parameters	2

Formula:

```
FluRate ~ 1 + Region + (1|Date)
```

Model fit statistics:

# LinearMixedModel.fitted

AIC	BIC	LogLikelihood	Deviance
318.71	364.35	-148.36	296.71

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	1.2233	0.096678	12.654	459
'Region_MidAtl'	0.010192	0.052221	0.19518	459
'Region_ENCentral'	0.051923	0.052221	0.9943	459
'Region_WNCentral'	0.23687	0.052221	4.5359	459
'Region_SAtl'	0.075481	0.052221	1.4454	459
'Region_ESCentral'	0.33917	0.052221	6.495	459
'Region_WSCentral'	0.069	0.052221	1.3213	459
'Region_Mtn'	0.046673	0.052221	0.89377	459
'Region_Pac'	-0.16013	0.052221	-3.0665	459

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate
'(Intercept)'	'(Intercept)'	'std'	0.6443

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.26627	0.24878	0.285

The  $p$ -values 7.3324e-06 and 2.1623e-10 respectively show that the fixed effects of the flu rates in regions `WNCentral` and `ESCentral` are significantly different relative to the flu rates in region `NE`.

The confidence limits for the standard deviation of the random-effects term,  $\sigma^2_b$ , do not include 0 (0.5297, 0.78368), which indicates that the random-effects term is significant. You can also test the significance of the random-effects terms using the `compare` method.

The conditional fitted response from the model at a given observation includes contributions from fixed and random effects. For example, the estimated best linear unbiased predictor (BLUP) of the flu rate for region `WNCentral` in week 10/9/2005 is

$$\begin{aligned}\hat{y}_{WNCentral,10/9/2005} &= \hat{\beta}_0 + \hat{\beta}_3 I[WNCentral] + \hat{b}_{10/9/2005} \\ &= 1.2233 + 0.23687 - 0.1718 \\ &= 1.28837.\end{aligned}$$

This is the fitted conditional response, since it includes contributions to the estimate from both the fixed and random effects. You can compute this value as follows.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(4) + STATS.Estimate(STATS.Level=='10/9/2005')

y_hat =

    1.2884
```

In the previous calculation, `beta(1)` corresponds to the estimate for  $\beta_0$  and `beta(4)` corresponds to the estimate for  $\beta_3$ . You can simply display the fitted value using the `fitted` method.

```
F = fitted(lme);
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')

ans =

    1.2884
```

The estimated marginal response for region `WNCentral` in week `10/9/2005` is

$$\begin{aligned}\hat{y}_{WNCentral,10/9/2005}^{(\text{marginal})} &= \hat{\beta}_0 + \hat{\beta}_3 I[WNCentral] \\ &= 1.2233 + 0.23687 \\ &= 1.46017.\end{aligned}$$

Compute the fitted marginal response.

```
F = fitted(lme,'Conditional',false);
```

# LinearMixedModel.fitted

---

```
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')
```

```
ans =
```

```
1.4602
```

## Plot Residuals vs. Fitted Values

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);
ds.Subject = nominal(ds.Subject);
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

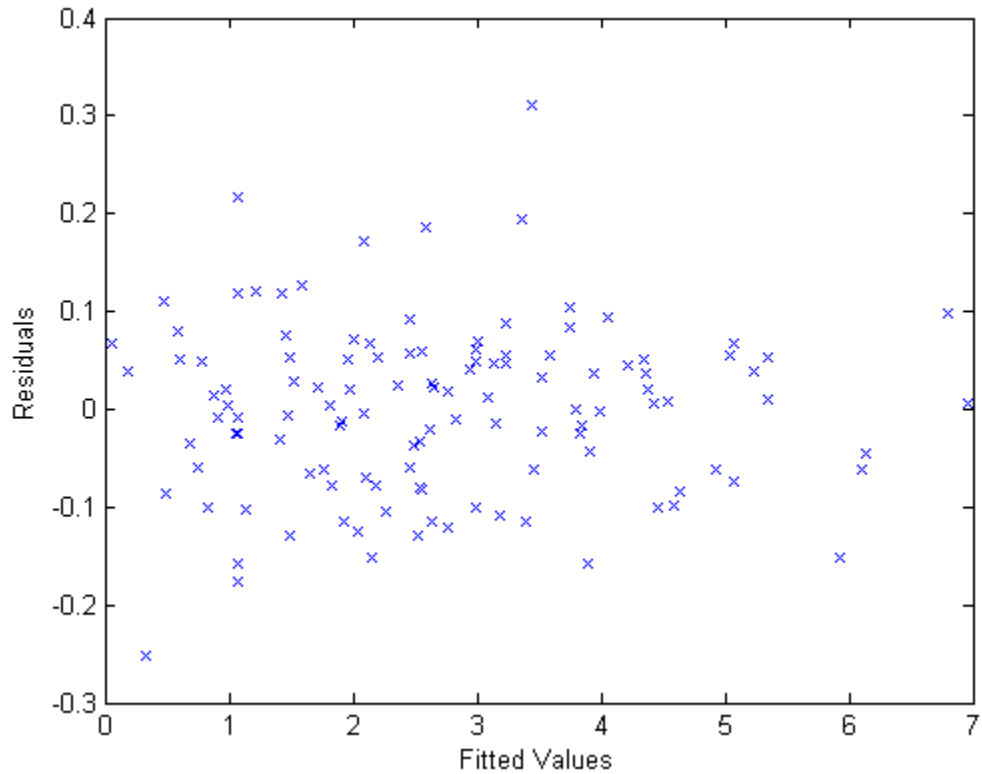
```
lme = fitlme(ds,'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Compute the fitted values and raw residuals.

```
F = fitted(lme);
R = residuals(lme);
```

Plot the residuals versus the fitted values.

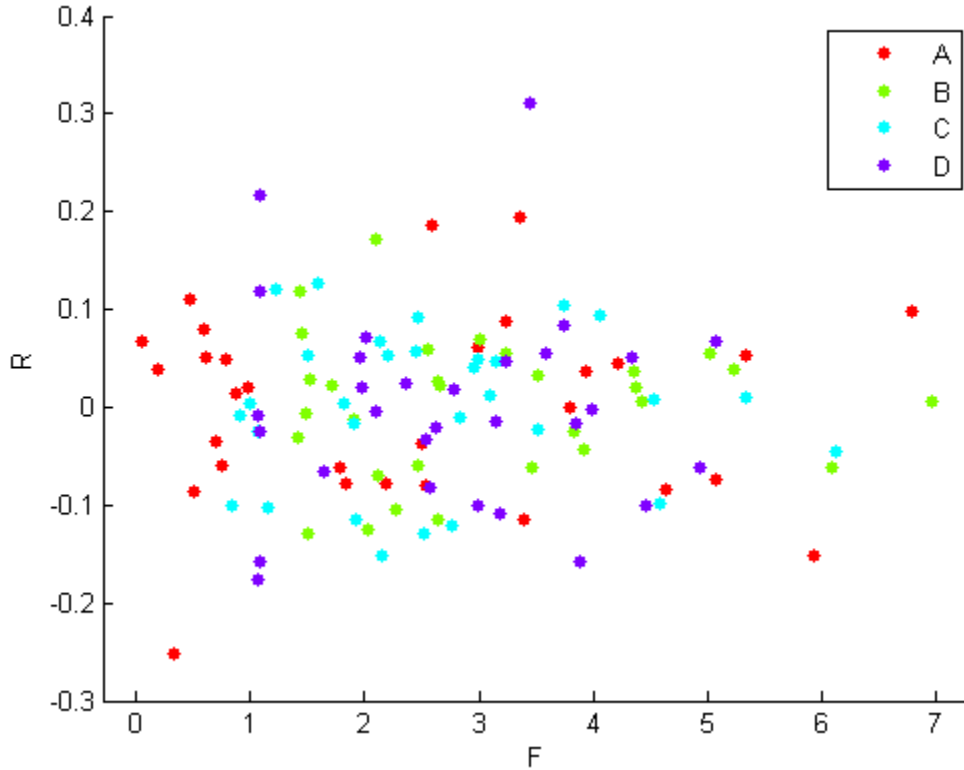
```
plot(F,R,'bx')  
xlabel('Fitted Values')  
ylabel('Residuals')
```



Now, plot the residuals versus the fitted values, grouped by program.

```
figure();  
gscatter(F,R,Program)
```

# LinearMixedModel.fitted



## Definitions

### Fitted Conditional and Marginal Response

A conditional response includes contributions from both fixed and random effects, whereas a marginal response includes contribution from only fixed effects.

Suppose the linear mixed-effects model, `lme`, has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also, suppose the  $p$ -by-1 estimated fixed-effects vector is  $\hat{\beta}$ , and the  $q$ -by-1



estimated best linear unbiased predictor (BLUP) vector of random effects is  $\hat{b}$ . The fitted conditional response is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b},$$

and the fitted marginal response is

$$\hat{y}_{Mar} = X\hat{\beta}.$$

## See Also

`LinearMixedModel` | `residuals` | `response` |

# categorical.flipdim

---

**Purpose** Flip categorical array along specified dimension

**Syntax** `B = flipdim(A,dim)`

**Description** `B = flipdim(A,dim)` returns the categorical array `A` with dimension `dim` flipped.

**See Also** `fliplr` | `flipud` | `permute` | `rot90`

**Purpose** Flip categorical matrix in left/right direction

**Syntax** `B = fliplr(A)`

**Description** `B = fliplr(A)` returns the 2-D categorical matrix `A` with rows preserved and columns flipped in the left/right direction.

**See Also** `flipdim` | `flipud` | `permute` | `rot90`

# categorical.flipud

---

**Purpose** Flip categorical matrix in up/down direction

**Syntax** `B = flipud(A)`

**Description** `B = flipud(A)` returns the 2-D categorical matrix `A` with rows preserved and columns flipped in the up/down direction.

**See Also** `flipdim` | `fliplr` | `permute` | `rot90`

## Purpose

Estimates of fixed effects and related statistics

## Syntax

```
beta = fixedEffects(lme)
[beta,betanames] = fixedEffects(lme)
[beta,betanames,stats] = fixedEffects(lme)
[beta,betanames,stats] = fixedEffects(lme,Name,Value)
```

## Description

`beta = fixedEffects(lme)` returns the estimated fixed-effects coefficients, `beta`, of the linear mixed-effects model `lme`.

`[beta,betanames] = fixedEffects(lme)` also returns the names of estimated fixed-effects coefficients in `betanames`. Each name corresponds to a fixed-effects coefficient in `beta`.

`[beta,betanames,stats] = fixedEffects(lme)` also returns the estimated fixed-effects coefficients of the linear mixed-effects model `lme` and related statistics in `stats`.

`[beta,betanames,stats] = fixedEffects(lme,Name,Value)` also returns the estimated fixed-effects coefficients of the linear mixed-effects model `lme` and related statistics with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **lme - Linear mixed-effects model**

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can

# LinearMixedModel.fixedEffects

---

specify several name and value pair arguments in any order as  
Name1, Value1, . . . , NameN, ValueN.

## 'Alpha' - Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

**Example:** 'Alpha', 0.01

## Data Types

single | double

## 'DFMethod' - Method for computing approximate degrees of freedom

'Residual' (default) | 'Satterthwaite' | 'None'

Method for computing approximate degrees of freedom for the  $t$ -statistic that tests the fixed-effects coefficients against 0, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'Residual'	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'Satterthwaite'	Satterthwaite approximation.
'None'	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

**Example:** 'DFMethod', 'Satterthwaite'

## Output Arguments

### **beta** - Fixed-effects coefficients estimates

vector

Fixed-effects coefficients estimates of the fitted linear mixed-effects model `lme`, returned as a vector.

### **betanames** - Names of fixed-effects coefficients

dataset array

Names of fixed-effects coefficients in `beta`, returned as a dataset array.

### **stats** - Fixed-effects estimates and related statistics

dataset array

Fixed-effects estimates and related statistics, returned as a dataset array that has one row for each of the fixed effects and one column for each of the following statistics.

Name	Name of the fixed effect coefficient
Estimate	Estimated coefficient value
SE	Standard error of the estimate
tStat	<i>t</i> -statistic for a test that the coefficient is zero
DF	Estimated degrees of freedom for the <i>t</i> -statistic
pValue	<i>p</i> -value for the <i>t</i> -statistic

# LinearMixedModel.fixedEffects

---

Lower	Lower limit of a 95% confidence interval for the fixed-effect coefficient
Upper	Upper limit of a 95% confidence interval for the fixed-effect coefficient

## Examples

### Display Fixed-Effects Coefficient Estimates and Names

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

The data set `weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

```
ds = dataset(InitWeight,Program,Subject,Week,y);
ds.Subject = nominal(ds.Subject);
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between week and program are the fixed effects. The intercept and week vary by subject.

```
lme = fitlme(ds,'y ~ InitWeight + Program*Week + (Week|Subject)');
```

Display the fixed-effects coefficient estimates and corresponding fixed-effects names.



```
[beta,betanames] = fixedEffects(lme);
```

```
beta =
```

```
    0.6610  
    0.0032  
    0.3608  
   -0.0333  
    0.1132  
    0.1732  
    0.0388  
    0.0305  
    0.0331
```

```
betanames =
```

```
    Name  
    '(Intercept)'  
    'InitWeight'  
    'Program_B'  
    'Program_C'  
    'Program_D'  
    'Week'  
    'Program_B:Week'  
    'Program_C:Week'  
    'Program_D:Week'
```

## Compute Coefficient Estimates and Related Statistics

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and horsepower, and potentially correlated random effects for intercept and acceleration grouped by model year. First, store the data in a dataset array.

# LinearMixedModel.fixedEffects

---

```
ds = dataset(Acceleration,Horsepower,Model_Year,MPG);
```

Fit the model.

```
lme = fitlme(ds, 'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Year)');
```

Compute the fixed-effects coefficients estimates and related statistics.

```
[~,~,stats] = fixedEffects(lme)
```

```
stats =
```

```
Fixed effect coefficients: DFMethod = 'Residual', Alpha = 0.05
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	50.133	2.2652	22.132	389	7.77e-45
'Acceleration'	-0.58327	0.13394	-4.3545	389	1.70e-05
'Horsepower'	-0.16954	0.0072609	-23.35	389	5.10e-47

The small  $p$ -values (under pValue) indicate that all fixed-effects coefficients are significant.

## Compute Confidence Intervals with Specified Options

Navigate to a folder containing sample data.

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

Load the sample data.

```
load shift
```

The data shows the deviations from the target quality characteristic measured from the products that five operators manufacture during three shifts: morning, evening, and night. This is a randomized block design, where the operators are the blocks. The experiment is designed to study the impact of the time of shift on the performance. The

performance measure is the deviation of the quality characteristics from the target value. This is simulated data.

Shift and Operator are nominal variables.

```
shift.Shift = nominal(shift.Shift);  
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by operator to assess if performance significantly differs according to the time of the shift.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

Compute the 99% confidence intervals for fixed-effects coefficients, using the residual method to compute the degrees of freedom. This is the default method.

```
[~,~,stats] = fixedEffects(lme, 'alpha', 0.01)
```

```
stats =
```

```
Fixed effect coefficients: DFMethod = 'Residual', Alpha = 0.01
```

Name	Estimate	SE	tStat	DF	pValue
'(Intercept)'	3.1196	0.88681	3.5178	12	0.0025
'Shift_Morning'	-0.3868	0.48344	-0.80009	12	0.4361
'Shift_Night'	1.9856	0.48344	4.1072	12	0.0007

Compute the 99% confidence intervals for fixed-effects coefficients, using the Satterthwaite approximation to compute the degrees of freedom.

```
[~,~,stats] = fixedEffects(lme, 'DFMethod', 'Satterthwaite', 'alpha', 0.01)
```

```
stats =
```

# LinearMixedModel.fixedEffects

---

Fixed effect coefficients: DFMethod = 'Satterthwaite', Alpha = 0.01

Name	Estimate	SE	tStat	DF	pV
'(Intercept)'	3.1196	0.88681	3.5178	6.123	0.
'Shift_Morning'	-0.3868	0.48344	-0.80009	10	0.
'Shift_Night'	1.9856	0.48344	4.1072	10	0.

The Satterthwaite approximation usually produces smaller DF values than the residual method. That is why it produces larger  $p$ -values (pValue) and larger confidence intervals (see Lower and Upper).

## See Also

[LinearMixedModel](#) | [fitlmecoefCI](#) | [coefTest](#) | [randomEffects](#) |

**Purpose**  $F$  probability density function

**Syntax**  $Y = \text{fpdf}(X, V1, V2)$

**Description**  $Y = \text{fpdf}(X, V1, V2)$  computes the  $F$  pdf at each of the values in  $X$  using the corresponding numerator degrees of freedom  $V1$  and denominator degrees of freedom  $V2$ .  $X$ ,  $V1$ , and  $V2$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.  $V1$  and  $V2$  parameters must contain real positive values, and the values in  $X$  must lie on the interval  $[0 \infty)$ .

The probability density function for the  $F$  distribution is

$$y = f(x | v_1, v_2) = \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right]}{\Gamma\left(\frac{v_1}{2}\right)\Gamma\left(\frac{v_2}{2}\right)} \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} \frac{x^{\frac{v_1-2}{2}}}{\left[1 + \left(\frac{v_1}{v_2}\right)x\right]^{\frac{v_1+v_2}{2}}}$$

**Examples**  $y = \text{fpdf}(1:6, 2, 2)$

$y =$   
0.2500 0.1111 0.0625 0.0400 0.0278 0.0204

$z = \text{fpdf}(3, 5:10, 5:10)$

$z =$   
0.0689 0.0659 0.0620 0.0577 0.0532 0.0487

**See Also** pdf | fcdf | finv | fstat | frnd

**How To** • “F Distribution” on page B-41

# fracfact

---

**Purpose** Fractional factorial design

**Syntax**  
`X = fracfact(gen)`  
`[X,conf] = fracfact(gen)`  
`[X,conf] = fracfact(gen,Name,Value)`

**Description** `X = fracfact(gen)` creates the two-level fractional factorial design defined by the generator string `gen`.

`[X,conf] = fracfact(gen)` returns a cell array of strings containing the confounding pattern for the design.

`[X,conf] = fracfact(gen,Name,Value)` creates a fractional factorial designs with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **gen**

Either a cell array of strings where each cell contains one “word,” or a string consisting of “words” separated by spaces. “Words” consist of case-sensitive letters or groups of letters, where 'a' represents string 1, 'b' represents string 2, ..., 'A' represents string 27, ..., 'Z' represents string 52.

Each word defines how the corresponding factor’s levels are defined as products of generators from a  $2^K$  full-factorial design.  $K$  is the number of letters of the alphabet in `gen`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'FactorNames'**

Cell array specifying the name for each factor.

**Default:** {'X1', 'X2', ...}

### 'MaxInt'

Positive integer setting the maximum level of interaction to include in the confounding output.

**Default:** 2

## Output Arguments

### X

The two-level fractional factorial design. X is a matrix of size N-by-P, where

- $N = 2^K$ , where K is the number of letters of the alphabet in gen.
- P is the number of words in gen.

Because X is a two-level design, the components of X are  $-1$ . For the meaning of X, see “Fractional Factorial Designs” on page 18-5.

### conf

Cell array of strings containing the confounding pattern for the design.

## Examples

Generate a fractional factorial design for four variables, where the fourth variable is the product of the first three:

```
x = fracfact('a b c abc')
```

```
x =
    -1    -1    -1    -1
    -1    -1     1     1
    -1     1    -1     1
    -1     1     1    -1
     1    -1    -1     1
     1    -1     1    -1
     1     1    -1    -1
     1     1     1     1
```

# fracfact

---

Find generators for a six-factor design that uses four factors and achieves resolution IV using fracfactgen. Use the result to specify the design:

```
generators = fracfactgen('a b c d e f',4, ... % 4 factors  
4) % resolution 4
```

```
generators =  
'a'  
'b'  
'c'  
'd'  
'bcd'  
'acd'
```

```
x = fracfact(generators)
```

```
x =  
-1 -1 -1 -1 -1 -1  
-1 -1 -1 1 1 1  
-1 -1 1 -1 1 1  
-1 -1 1 1 -1 -1  
-1 1 -1 -1 1 -1  
-1 1 -1 1 -1 1  
-1 1 1 1 1 -1  
1 -1 -1 -1 -1 1  
1 -1 -1 1 1 -1  
1 -1 1 -1 1 -1  
1 -1 1 1 -1 1  
1 1 -1 -1 1 1  
1 1 -1 1 -1 -1  
1 1 1 -1 -1 -1  
1 1 1 1 1 1
```



**References**

[1] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

**See Also**

ff2n | fracfactgen | fullfact | hadamard

**How To**

• “Fractional Factorial Designs” on page 18-5

# fracfactgen

---

**Purpose** Fractional factorial design generators

**Syntax**

```
generators = fracfactgen(terms)
generators = fracfactgen(terms,k)
generators = fracfactgen(terms,k,R)
generators = fracfactgen(terms,k,R,basic)
```

**Description** `generators = fracfactgen(terms)` uses the Franklin-Bailey algorithm to find generators for the smallest two-level fractional-factorial design for estimating linear model terms specified by `terms`. `terms` is a string consisting of words formed from the 52 case-sensitive letters a-Z, separated by spaces. Use 'a' - 'z' for the first 26 factors, and, if necessary, 'A' - 'Z' for the remaining factors. For example, `terms = 'a b c ab ac'`. Single-character words indicate main effects to be estimated; multiple-character words indicate interactions. Alternatively, `terms` is an  $m$ -by- $n$  matrix of 0s and 1s where  $m$  is the number of model terms to be estimated and  $n$  is the number of factors. For example, if `terms` contains rows `[0 1 0 0]` and `[1 0 0 1]`, then the factor `b` and the interaction between factors `a` and `d` are included in the model. `generators` is a cell array of strings with one generator per cell. Pass `generators` to `fracfact` to produce the fractional-factorial design and corresponding confounding pattern.

`generators = fracfactgen(terms,k)` returns generators for a two-level fractional-factorial design with  $2^k$ -runs, if possible. If `k` is [], `fracfactgen` finds the smallest design.

`generators = fracfactgen(terms,k,R)` finds a design with resolution `R`, if possible. The default resolution is 3.

A design of *resolution R* is one in which no  $n$ -factor interaction is confounded with any other effect containing less than  $R - n$  factors. Thus a resolution III design does not confound main effects with one another but may confound them with two-way interactions, while a resolution IV design does not confound either main effects or two-way interactions but may confound two-way interactions with each other.

If `fracfactgen` is unable to find a design at the requested resolution, it tries to find a lower-resolution design sufficient to calibrate the model.

If it is successful, it returns the generators for the lower-resolution design along with a warning. If it fails, it returns an error.

`generators = fracfactgen(terms,k,R,basic)` also accepts a vector `basic` specifying the indices of factors that are to be treated as basic. These factors receive full-factorial treatments in the design. The default includes factors that are part of the highest-order interaction in `terms`.

## Examples

Suppose you wish to determine the effects of four two-level factors, for which there may be two-way interactions. A full-factorial design would require  $2^4 = 16$  runs. The `fracfactgen` function finds generators for a resolution IV (separating main effects) fractional-factorial design that requires only  $2^3 = 8$  runs:

```
generators = fracfactgen('a b c d',3,4)
generators =
  'a'
  'b'
  'c'
  'abc'
```

The more economical design and the corresponding confounding pattern are returned by `fracfact`:

```
[dfF,confounding] = fracfact(generators)
dfF =
  -1  -1  -1  -1
  -1  -1   1   1
  -1   1  -1   1
  -1   1   1  -1
   1  -1  -1   1
   1  -1   1  -1
   1   1  -1  -1
   1   1   1   1
confounding =
  'Term'      'Generator'    'Confounding'
  'X1'        'a'            'X1'
  'X2'        'b'            'X2'
```

# fracfactgen

---

'X3'	'c'	'X3'
'X4'	'abc'	'X4'
'X1*X2'	'ab'	'X1*X2 + X3*X4'
'X1*X3'	'ac'	'X1*X3 + X2*X4'
'X1*X4'	'bc'	'X1*X4 + X2*X3'
'X2*X3'	'bc'	'X1*X4 + X2*X3'
'X2*X4'	'ac'	'X1*X3 + X2*X4'
'X3*X4'	'ab'	'X1*X2 + X3*X4'

The confounding pattern shows, for example, that the two-way interaction between X1 and X2 is confounded by the two-way interaction between X3 and X4.

## References

[1] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

## See Also

fracfact | hadamard

## How To

- “Fractional Factorial Designs” on page 18-5

**Purpose**

Friedman's test

**Syntax**

```
p = friedman(X, reps)
p = friedman(X, reps, displayopt)
[p, table] = friedman(...)
[p, table, stats] = friedman(...)
```

**Description**

`p = friedman(X, reps)` performs the nonparametric Friedman's test to compare column effects in a two-way layout. Friedman's test is similar to classical balanced two-way ANOVA, but it tests only for column effects after adjusting for possible row effects. It does not test for row effects or interaction effects. Friedman's test is appropriate when columns represent treatments that are under study, and rows represent nuisance effects (blocks) that need to be taken into account but are not of any interest.

The different columns of `X` represent changes in a factor A. The different rows represent changes in a blocking factor B. If there is more than one observation for each combination of factors, input `reps` indicates the number of replicates in each "cell," which must be constant.

The matrix below illustrates the format for a set-up where column factor A has three levels, row factor B has two levels, and there are two replicates (`reps=2`). The subscripts indicate row, column, and replicate, respectively.

$$\begin{bmatrix} x_{111} & x_{121} & x_{131} \\ x_{112} & x_{122} & x_{132} \\ x_{211} & x_{221} & x_{231} \\ x_{212} & x_{222} & x_{232} \end{bmatrix}$$

Friedman's test assumes a model of the form

$$x_{ijk} = \mu + \alpha_i + \beta_j + \varepsilon_{ijk}$$

where  $\mu$  is an overall location parameter,  $\alpha_i$  represents the column effect,  $\beta_j$  represents the row effect, and  $\varepsilon_{ijk}$  represents the error. This test ranks the data within each level of B, and tests for a difference across levels of A. The  $p$  that `friedman` returns is the  $p$  value for the null hypothesis that  $\alpha_i = 0$ . If the  $p$  value is near zero, this casts doubt on the null hypothesis. A sufficiently small  $p$  value suggests that at least one column-sample median is significantly different than the others; i.e., there is a main effect due to factor A. The choice of a critical  $p$  value to determine whether a result is “statistically significant” is left to the researcher. It is common to declare a result significant if the  $p$  value is less than 0.05 or 0.01.

`friedman` also displays a figure showing an ANOVA table, which divides the variability of the ranks into two or three parts:

- The variability due to the differences among the column effects
- The variability due to the interaction between rows and columns (if `reps` is greater than its default value of 1)
- The remaining variability not explained by any systematic source

The ANOVA table has six columns:

- The first shows the source of the variability.
- The second shows the Sum of Squares (SS) due to each source.
- The third shows the degrees of freedom (df) associated with each source.
- The fourth shows the Mean Squares (MS), which is the ratio SS/df.
- The fifth shows Friedman’s chi-square statistic.
- The sixth shows the  $p$  value for the chi-square statistic.

`p = friedman(X, reps, displayopt)` enables the ANOVA table display when `displayopt` is 'on' (default) and suppresses the display when `displayopt` is 'off'.

`[p,table] = friedman(...)` returns the ANOVA table (including column and row labels) in cell array `table`. (You can copy a text version of the ANOVA table to the clipboard by selecting Copy Text from the **Edit** menu.

`[p,table,stats] = friedman(...)` returns a `stats` structure that you can use to perform a follow-up multiple comparison test. The `friedman` test evaluates the hypothesis that the column effects are all the same against the alternative that they are not all the same. Sometimes it is preferable to perform a test to determine which pairs of column effects are significantly different, and which are not. You can use the `multcompare` function to perform such tests by supplying the `stats` structure as input.

### Assumptions

Friedman's test makes the following assumptions about the data in `X`:

- All data come from populations having the same continuous distribution, apart from possibly different locations due to column and row effects.
- All observations are mutually independent.

The classical two-way ANOVA replaces the first assumption with the stronger assumption that data come from normal distributions.

## Examples

### Test For Column Effects Using Friedman's Test

This example shows how to test for column effects in a two-way layout using Friedman's test.

Load the sample data.

```
load popcorn
popcorn
```

```
popcorn =
    5.5000    4.5000    3.5000
```

# friedman

---

5.5000	4.5000	4.0000
6.0000	4.0000	3.0000
6.5000	5.0000	4.0000
7.0000	5.5000	5.0000
7.0000	5.0000	4.5000

This data comes from a study of popcorn brands and popper type (Hogg 1987). The columns of the matrix `popcorn` are brands (Gourmet, National, and Generic). The rows are popper type (Oil and Air). The study popped a batch of each brand three times with each popper. The values are the yield in cups of popped popcorn.

Use Friedman's test to determine whether the popcorn brand affects the yield of popcorn.

```
p = friedman(popcorn,3)
```

```
p =
```

```
0.0010
```



Friedman's ANOVA Table					
Source	SS	df	MS	Chi-sq	Prob>Chi-sq
Columns	99.75	2	49.875	13.76	0.001
Interaction	0.0833	2	0.0417		
Error	16.1667	12	1.3472		
Total	116	17			

Test for column effects after row effects are removed

The small value of  $p = 0.001$  indicates the popcorn brand affects the yield of popcorn.

## References

- [1] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.
- [2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

# friedman

---

## **See Also**

`anova2` | `multcompare` | `kruskalwallis`

**Purpose**  $F$  random numbers

**Syntax**  
`R = frnd(V1,V2)`  
`R = frnd(V1,V2,m,n,...)`  
`R = frnd(V1,V2,[m,n,...])`

**Description** `R = frnd(V1,V2)` generates random numbers from the  $F$  distribution with numerator degrees of freedom  $V1$  and denominator degrees of freedom  $V2$ .  $V1$  and  $V2$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for  $V1$  or  $V2$  is expanded to a constant array with the same dimensions as the other input.

`R = frnd(V1,V2,m,n,...)` or `R = frnd(V1,V2,[m,n,...])` generates an  $m$ -by- $n$ -by-... array containing random numbers from the  $F$  distribution with parameters  $V1$  and  $V2$ .  $V1$  and  $V2$  can each be scalars or arrays of the same size as  $R$ .

**Examples**

```
n1 = frnd(1:6,1:6)
n1 =
    0.0022    0.3121    3.0528    0.3189    0.2715    0.9539

n2 = frnd(2,2,[2 3])
n2 =
    0.3186    0.9727    3.0268
    0.2052  148.5816    0.2191

n3 = frnd([1 2 3;4 5 6],1,2,3)
n3 =
    0.6233    0.2322   31.5458
    2.5848    0.2121    4.4955
```

**See Also** `random` | `fpdf` | `fcdf` | `finv` | `fstat`

**How To**

- “F Distribution” on page B-41

# fstat

---

**Purpose**  $F$  mean and variance

**Syntax** `[M,V] = fstat(V1,V2)`

**Description** `[M,V] = fstat(V1,V2)` returns the mean of and variance for the  $F$  distribution with numerator degrees of freedom  $V1$  and denominator degrees of freedom  $V2$ .  $V1$  and  $V2$  can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $V1$  or  $V2$  is expanded to a constant arrays with the same dimensions as the other input.

The mean of the  $F$  distribution for values of  $v_2$  greater than 2 is

$$\frac{v_2}{v_2 - 2}$$

The variance of the  $F$  distribution for values of  $v_2$  greater than 4 is

$$\frac{2v_2^2(v_1 + v_2 - 2)}{v_1(v_2 - 2)^2(v_2 - 4)}$$

The mean of the  $F$  distribution is undefined if  $v_2$  is less than 3. The variance is undefined for  $v_2$  less than 5.

**Examples** `fstat` returns NaN when the mean and variance are undefined.

```
[m,v] = fstat(1:5,1:5)
m =
    NaN    NaN    3.0000    2.0000    1.6667
v =
    NaN    NaN    NaN    NaN    8.8889
```

**See Also** `fpdf` | `fcdf` | `finv` | `frnd`

**How To** • “F Distribution” on page B-41

**Purpose**

Interactive contour plot

**Syntax**

```
fsurfht(fun,xlims,ylim)  
fsurfht(fun,xlims,ylim,p1,p2,p3,p4,p5)
```

**Description**

`fsurfht(fun,xlims,ylim)` is an interactive contour plot of the function specified by the text variable `fun`. The  $x$ -axis limits are specified by `xlims` in the form `[xmin xmax]`, and the  $y$ -axis limits are specified by `ylim` in the form `[ymin ymax]`.

`fsurfht(fun,xlims,ylim,p1,p2,p3,p4,p5)` allows for five optional parameters that you can supply to the function `fun`.

The intersection of the vertical and horizontal reference lines on the plot defines the current  $x$  value and  $y$  value. You can drag these reference lines and watch the calculated  $z$ -values (at the top of the plot) update simultaneously. Alternatively, you can type the  $x$  value and  $y$  value into editable text fields on the  $x$ -axis and  $y$ -axis.

**Examples**

Plot the Gaussian likelihood function for the `gas.mat` data.

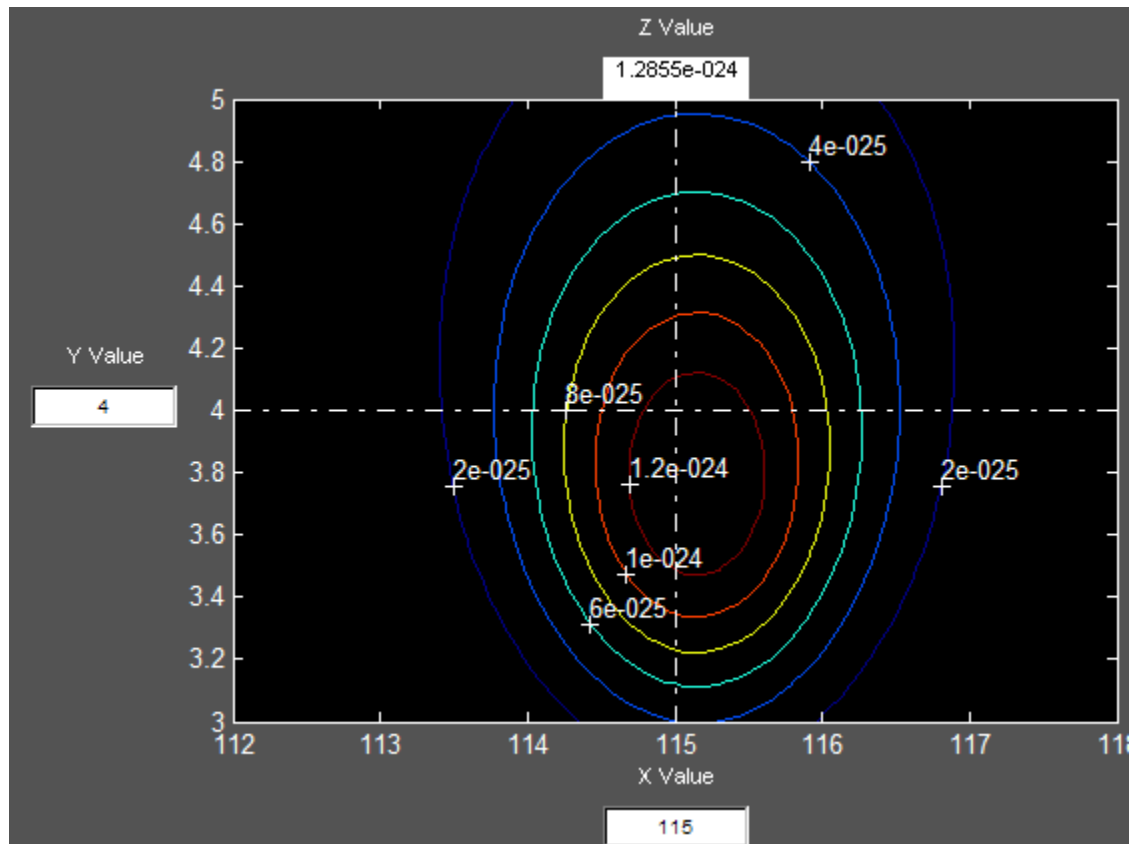
```
load gas
```

Create a function containing the following commands, and name it `gauslike.m`.

```
function z = gauslike(mu,sigma,p1)  
n = length(p1);  
z = ones(size(mu));  
for i = 1:n  
z = z .* (normpdf(p1(i),mu,sigma));  
end
```

The `gauslike` function calls `normpdf`, treating the data sample as fixed and the parameters  $\mu$  and  $\sigma$  as variables. Assume that the gas prices are normally distributed, and plot the likelihood surface of the sample.

```
fsurfht('gauslike',[112 118],[3 5],price1)
```



The sample mean is the  $x$  value at the maximum, but the sample standard deviation is *not* the  $y$  value at the maximum.

```

mumax = mean(price1)
mumax =
    115.1500
sigmax = std(price1)*sqrt(19/20)
sigmax =
    3.7719

```



# fullfact

---

**Purpose** Full factorial design

**Syntax** `dFF = fullfact(levels)`

**Description** `dFF = fullfact(levels)` gives factor settings `dFF` for a full factorial design with  $n$  factors, where the number of levels for each factor is given by the vector `levels` of length  $n$ . `dFF` is  $m$ -by- $n$ , where  $m$  is the number of treatments in the full-factorial design. Each row of `dFF` corresponds to a single treatment. Each column contains the settings for a single factor, with integer values from one to the number of levels.

**Examples** The following generates an eight-run full-factorial design with two levels in the first factor and four levels in the second factor:

```
dFF = fullfact([2 4])
dFF =
     1     1
     2     1
     1     2
     2     2
     1     3
     2     3
     1     4
     2     4
```

**See Also** `ff2n`



**Purpose**

Gage repeatability and reproducibility study

**Syntax**

```
gagerr(y, {part, operator})  
gagerr(y, GROUP)  
gagerr(y, part)  
gagerr(..., param1, val1, param2, val2, ...)  
[TABLE, stats] = gagerr(...)
```

**Description**

`gagerr(y, {part, operator})` performs a gage repeatability and reproducibility study on measurements in `y` collected by `operator` on `part`. `y` is a column vector containing the measurements on different parts. `part` and `operator` are categorical variables, numeric vectors, character matrices, or cell arrays of strings. The number of elements in `part` and `operator` should be the same as in `y`.

`gagerr` prints a table in the command window in which the decomposition of variance, standard deviation, study var (5.15 x standard deviation) are listed with respective percentages for different sources. Summary statistics are printed below the table giving the number of distinct categories (NDC) and the percentage of Gage R&R of total variations (PRR).

`gagerr` also plots a bar graph showing the percentage of different components of variations. Gage R&R, repeatability, reproducibility, and part-to-part variations are plotted as four vertical bars. Variance and study var are plotted as two groups.

To determine the capability of a measurement system using NDC, use the following guidelines:

- If  $NDC > 5$ , the measurement system is capable.
- If  $NDC < 2$ , the measurement system is not capable.
- Otherwise, the measurement system may be acceptable.

To determine the capability of a measurement system using PRR, use the following guidelines:

- If  $PRR < 10\%$ , the measurement system is capable.

- If  $PRR > 30\%$ , the measurement system is not capable.
- Otherwise, the measurement system may be acceptable.

`gagerr(y, GROUP)` performs a gage R&R study on measurements in `y` with part and operator represented in `GROUP`. `GROUP` is a numeric matrix whose first and second columns specify different parts and operators, respectively. The number of rows in `GROUP` should be the same as the number of elements in `y`.

`gagerr(y, part)` performs a gage R&R study on measurements in `y` without operator information. The assumption is that all variability is contributed by `part`.

`gagerr(..., param1, val1, param2, val2, ...)` performs a gage R&R study using one or more of the following parameter name/value pairs:

- `'spec'` — A two-element vector that defines the lower and upper limit of the process, respectively. In this case, summary statistics printed in the command window include Precision-to-Tolerance Ratio (PTR). Also, the bar graph includes an additional group, the percentage of tolerance.

To determine the capability of a measurement system using PTR, use the following guidelines:

- If  $PTR < 0.1$ , the measurement system is capable.
- If  $PTR > 0.3$ , the measurement system is not capable.
- Otherwise, the measurement system may be acceptable.
- `'printtable'` — A string with a value `'on'` or `'off'` that indicates whether the tabular output should be printed in the command window or not. The default value is `'on'`.
- `'printgraph'` — A string with a value `'on'` or `'off'` that indicates whether the bar graph should be plotted or not. The default value is `'on'`.
- `'randomoperator'` — A logical value, `true` or `false`, that indicates whether the effect of operator is random or not. The default value is `true`.

- 'model' — The model to use, specified by one of:
  - 'linear' — Main effects only (default)
  - 'interaction' — Main effects plus two-factor interactions
  - 'nested' — Nest operator in part

The default value is 'linear'.

[TABLE, stats] = gagerr(...) returns a 6-by-5 matrix TABLE and a structure stats. The columns of TABLE, from left to right, represent variance, percentage of variance, standard deviations, study var, and percentage of study var. The rows of TABLE, from top to bottom, represent different sources of variations: gage R&R, repeatability, reproducibility, operator, operator and part interactions, and part. stats is a structure containing summary statistics for the performance of the measurement system. The fields of stats are:

- ndc — Number of distinct categories
- prr — Percentage of gage R&R of total variations
- ptr — Precision-to-tolerance ratio. The value is NaN if the parameter 'spec' is not given.

## Examples

### Gage R&R Study

Simulate a measurement system by randomly generating the operators, parts, and the measurements, y, operators do on the parts.

```
rng(1234, 'twister')           % for reproducibility
y = randn(100,1);             % measurements
part = ceil(3*rand(100,1));    % parts
operator = ceil(4*rand(100,1)); % operators
```

Conduct a gage R&R study for this system using a mixed ANOVA model without interactions.

```
gagerr(y, {part, operator}, 'randomoperator', true)
```

Columns 1 through 4

'Source'	'Variance'	'% Variance'	'sigma'
'Gage R&R'	[ 0.9715]	[ 99.2653]	[0.9857]
' Repeatability'	[ 0.9535]	[ 97.4201]	[0.9765]
' Reproducibility'	[ 0.0181]	[ 1.8452]	[0.1344]
' Operator'	[ 0.0181]	[ 1.8452]	[0.1344]
'Part'	[ 0.0072]	[ 0.7347]	[0.0848]
'Total'	[ 0.9787]	[ 100]	[0.9893]

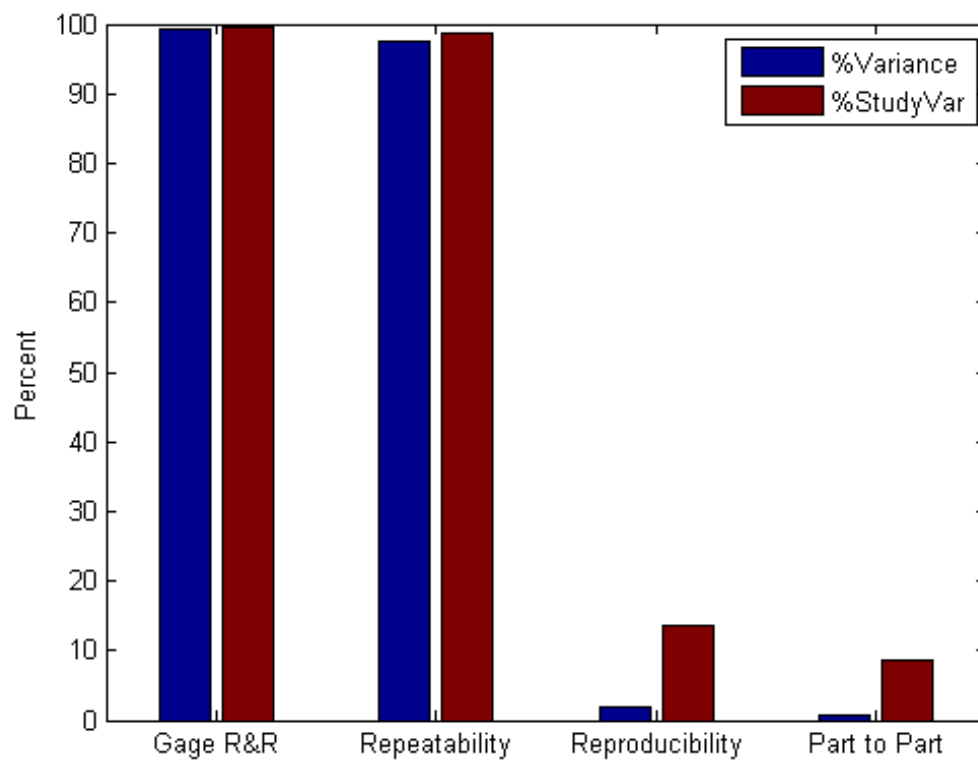
Columns 5 through 6

'5.15*sigma'	'% 5.15*sigma'
[ 5.0762]	[ 99.6320]
[ 5.0288]	[ 98.7016]
[ 0.6921]	[ 13.5838]
[ 0.6921]	[ 13.5838]
[ 0.4367]	[ 8.5716]
[ 5.0949]	''

Number of distinct categories (NDC):0

% of Gage R&R of total variations (PRR): 99.63

Note: The last column of the above table does not have to sum to 100%

**How To**

- “Grouping Variables” on page 2-51

# gamcdf

---

**Purpose** Gamma cumulative distribution function

**Syntax**  
`gamcdf(x,a,b)`  
`[p,plo,pup] = gamcdf(x,a,b,pcov,alpha)`  
`[p,plo,pup] = gamcdf( __ , 'upper' )`

**Description** `gamcdf(x,a,b)` returns the gamma cdf at each of the values in `x` using the corresponding shape parameters in `a` and scale parameters in `b`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `a` and `b` must be positive.

`[p,plo,pup] = gamcdf(x,a,b,pcov,alpha)` produces confidence bounds for `p` when the input parameters `a` and `b` are estimates. `pcov` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1-alpha)% confidence bounds. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p,plo,pup] = gamcdf( __ , 'upper' )` returns the complement of the gamma cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use the 'upper' argument with any of the previous syntaxes.

The gamma cdf is

$$p = F(x | a, b) = \frac{1}{b^a \Gamma(a)} \int_0^x t^{a-1} e^{-\frac{t}{b}} dt$$

The result, `p`, is the probability that a single observation from a gamma distribution with parameters `a` and `b` will fall in the interval `[0 x]`.

`gammainc` is the gamma distribution with `b` fixed at 1.

**Examples****Compute Gamma Distribution CDF**

The mean of the gamma distribution is the product of the parameters,  $ab$ . In this example, the mean approaches the median as it increases (i.e., the distribution becomes more symmetric).

```
a = 1:6;  
b = 5:10;  
prob = gamcdf(a.*b,a,b)
```

```
prob =  
    0.6321    0.5940    0.5768    0.5665    0.5595    0.5543
```

**See Also**

[cdf](#) | [gampdf](#) | [gaminv](#) | [gamstat](#) | [gamfit](#) | [gamlike](#) | [gamrnd](#) | [gamma](#)

**How To**

- “Gamma Distribution” on page B-43

# gamfit

---

## Purpose

Gamma parameter estimates

## Syntax

```
phat = gamfit(data)
[phat,pci] = gamfit(data)
[phat,pci] = gamfit(data,alpha)
[...] = gamfit(data,alpha,censoring,freq,options)
```

## Description

`phat = gamfit(data)` returns the maximum likelihood estimates (MLEs) for the parameters of the gamma distribution given the data in vector `data`.

`[phat,pci] = gamfit(data)` returns MLEs and 95% percent confidence intervals. The first row of `pci` is the lower bound of the confidence intervals; the last row is the upper bound.

`[phat,pci] = gamfit(data,alpha)` returns  $100(1 - \alpha)\%$  confidence intervals. For example, `alpha = 0.01` yields 99% confidence intervals.

`[...] = gamfit(data,alpha,censoring)` accepts a Boolean vector of the same size as `data` that is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = gamfit(data,alpha,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but may contain any nonnegative values.

`[...] = gamfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The gamma fit function accepts an `options` structure which can be created using the function `statset`. Enter `statset('gamfit')` to see the names and default values of the parameters that `gamfit` accepts in the `options` structure.

## Examples

Fit a gamma distribution to random data generated from a specified gamma distribution:



```
a = 2; b = 4;
data = gamrnd(a,b,100,1);

[p,ci] = gamfit(data)
p =
    2.1990    3.7426
ci =
    1.6840    2.8298
    2.7141    4.6554
```

**References**

[1] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 88.

**See Also**

mle | gamlike | gampdf | gamcdf | gaminv | gamstat | gamrnd

**How To**

- “Gamma Distribution” on page B-43

# prob.GammaDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Gamma probability distribution object

**Description** prob.GammaDistribution is an object consisting of parameters, a model description, and sample data for a gamma probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Gamma')` creates a gamma probability distribution object using the default parameter values.

`pd = makedist('Gamma','a',a,'b',b)` creates a gamma probability distribution object using the specified parameter values.

## Input Arguments

### **a** - Shape parameter

1 (default) | positive scalar value

Shape parameter for the gamma distribution, specified as a positive scalar value.

### Data Types

single | double

### **b** - Scale parameter

1 (default) | nonnegative scalar value

Scale parameter for the gamma distribution, specified as a nonnegative scalar value.

### Data Types

single | double

## Properties

**a**

Shape parameter for the gamma distribution, stored as a positive scalar value.

**Data Types**

single | double

**b**

Scale parameter for the gamma distribution, stored as a nonnegative scalar value.

**Data Types**

single | double

**DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

**Data Types**

char

**InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

**Data Types**

single | double

**IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

**Data Types**

logical

# prob.GammaDistribution

---

## **NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## **Data Types**

single | double

## **ParameterCovariance**

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## **Data Types**

single | double

## **ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## **Data Types**

char

## **ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## **Data Types**

logical

## **ParameterNames**

Names of distribution parameters, stored as a cell array of strings.  
This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This  
property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a  
vector containing the lower and upper truncation boundaries.  
This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object

# prob.GammaDistribution

---

random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Gamma Distribution

The gamma distribution is a two-parameter family of distributions used to model sums of exponentially distributed random variables. The chi-square and the exponential distributions, which are children of the gamma distribution, are one-parameter distributions that fix one of the two gamma parameters.

The gamma distribution uses the following parameters.

Parameter	Description	Support
a	Shape parameter	$a > 0$
b	Scale parameter	$b \geq 0$

The probability density function (pdf) is

$$f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}} ; \quad x > 0,$$

where  $\Gamma(\cdot)$  is the Gamma function.

## Examples

### Create a Gamma Distribution Object Using Default Parameters

Create a gamma distribution object using the default parameter values.

```
pd = makedist('Gamma')
```

```
pd =
```

```
GammaDistribution
```

```
Gamma distribution
```

```
  a = 1
```

```
  b = 1
```

### Create a Gamma Distribution Object Using Specified Parameters

Create a gamma distribution object by specifying the parameter values.

```
pd = makedist('Gamma', 'a',2,'b',4)
```

```
pd =
```

```
GammaDistribution
```

```
Gamma distribution
```

```
  a = 2
```

```
  b = 4
```

Compute the mean of the distribution.

# prob.GammaDistribution

---

`m = mean(pd)`

`m =`

`8`

## See Also

`makedist` | `fitdist` | `dfittool`

## Concepts

- “Gamma Distribution” on page B-43
- Class Attributes
- Property Attributes



**Purpose** Gamma inverse cumulative distribution function

**Syntax** `X = gaminv(P,A,B)`  
`[X,XLO,XUP] = gamcdf(P,A,B,pcov,alpha)`

**Description** `X = gaminv(P,A,B)` computes the inverse of the gamma cdf with shape parameters in `A` and scale parameters in `B` for the corresponding probabilities in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `A` and `B` must all be positive, and the values in `P` must lie on the interval `[0 1]`.

The gamma inverse function in terms of the gamma cdf is

$$x = F^{-1}(p | a, b) = \{x : F(x | a, b) = p\}$$

where

$$p = F(x | a, b) = \frac{1}{b^a \Gamma(a)} \int_0^x t^{a-1} e^{-\frac{t}{b}} dt$$

`[X,XLO,XUP] = gamcdf(P,A,B,pcov,alpha)` produces confidence bounds for `P` when the input parameters `A` and `B` are estimates. `pcov` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1-alpha)% confidence bounds. `PLO` and `PUP` are arrays of the same size as `P` containing the lower and upper confidence bounds.

**Algorithms** There is no known analytical solution to the integral equation above. `gaminv` uses an iterative approach (Newton's method) to converge on the solution.

**Examples** This example shows the relationship between the gamma cdf and its inverse function.

# gaminv

---

```
a = 1:5;  
b = 6:10;  
x = gaminv(gamcdf(1:5,a,b),a,b)  
x =  
    1.0000    2.0000    3.0000    4.0000    5.0000
```

## See Also

[icdf](#) | [gamcdf](#) | [gampdf](#) | [gamstat](#) | [gamfit](#) | [gamlike](#) | [gamrnd](#)

## How To

- “Gamma Distribution” on page B-43

**Purpose**

Gamma negative log-likelihood

**Syntax**

```
nlogL = gamlike(params,data)
[nlogL,AVAR] = gamlike(params,data)
```

**Description**

`nlogL = gamlike(params,data)` returns the negative of the gamma log-likelihood of the parameters, `params`, given `data`. `params(1)=A`, shape parameters, and `params(2)=B`, scale parameters.

`[nlogL,AVAR] = gamlike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates when the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`[...] = gamlike(params,data,censoring)` accepts a Boolean vector of the same size as `data` that is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = gamfit(params,data,censoring,freq)` accepts a frequency vector of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but may contain any non-negative values.

`gamlike` is a utility function for maximum likelihood estimation of the gamma distribution. Since `gamlike` returns the negative gamma log-likelihood function, minimizing `gamlike` using `fminsearch` is the same as maximizing the likelihood.

**Examples**

Compute the negative log-likelihood of parameter estimates computed by the `gamfit` function:

```
a = 2; b = 3;
r = gamrnd(a,b,100,1);
```

```
[nlogL,AVAR] = gamlike(gamfit(r),r)
nlogL =
    267.5648
AVAR =
```

# gamlike

---

0.0788 -0.1104  
-0.1104 0.1955

## See Also

[gamfit](#) | [gampdf](#) | [gamcdf](#) | [gaminv](#) | [gamstat](#) | [gamrnd](#)

## How To

- “Gamma Distribution” on page B-43

<b>Purpose</b>	Gamma probability density function
<b>Syntax</b>	<code>Y = gampdf(X,A,B)</code>
<b>Description</b>	<p><code>Y = gampdf(X,A,B)</code> computes the gamma pdf at each of the values in <code>X</code> using the corresponding shape parameters in <code>A</code> and scale parameters in <code>B</code>. <code>X</code>, <code>A</code>, and <code>B</code> can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in <code>A</code> and <code>B</code> must all be positive, and the values in <code>X</code> must lie on the interval <math>[0 \infty)</math>.</p> <p>The gamma pdf is</p>

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}}$$

The gamma probability density function is useful in reliability models of lifetimes. The gamma distribution is more flexible than the exponential distribution in that the probability of a product surviving an additional period may depend on its current age. The exponential and  $\chi^2$  functions are special cases of the gamma function.

<b>Examples</b>	The exponential distribution is a special case of the gamma distribution.
-----------------	---

```
mu = 1:5;

y = gampdf(1,1,mu)
y =
    0.3679    0.3033    0.2388    0.1947    0.1637

y1 = exppdf(1,mu)
y1 =
    0.3679    0.3033    0.2388    0.1947    0.1637
```

<b>See Also</b>	<code>pdf</code>   <code>gamcdf</code>   <code>gaminv</code>   <code>gamstat</code>   <code>gamfit</code>   <code>gamlike</code>   <code>gamrnd</code>
-----------------	--

## How To

- “Gamma Distribution” on page B-43

<b>Purpose</b>	Gamma random numbers
<b>Syntax</b>	<pre>R = gamrnd(A,B) R = gamrnd(A,B,m,n,...) R = gamrnd(A,B,[m,n,...])</pre>
<b>Description</b>	<p><code>R = gamrnd(A,B)</code> generates random numbers from the gamma distribution with shape parameters in <code>A</code> and scale parameters in <code>B</code>. <code>A</code> and <code>B</code> can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for <code>A</code> or <code>B</code> is expanded to a constant array with the same dimensions as the other input.</p> <p><code>R = gamrnd(A,B,m,n,...)</code> or <code>R = gamrnd(A,B,[m,n,...])</code> generates an <code>m-by-n-by-...</code> array containing random numbers from the gamma distribution with parameters <code>A</code> and <code>B</code>. <code>A</code> and <code>B</code> can each be scalars or arrays of the same size as <code>R</code>.</p>
<b>Examples</b>	<pre>n1 = gamrnd(1:5,6:10) n1 =     9.1132  12.8431  24.8025  38.5960 106.4164  n2 = gamrnd(5,10,[1 5]) n2 =     30.9486  33.5667  33.6837  55.2014  46.8265  n3 = gamrnd(2:6,3,1,5) n3 =     12.8715  11.3068  3.0982  15.6012  21.6739</pre>
<b>See Also</b>	<code>randg</code>   <code>random</code>   <code>gampdf</code>   <code>gamcdf</code>   <code>gaminv</code>   <code>gamstat</code>   <code>gamfit</code>   <code>gamlike</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Gamma Distribution” on page B-43</li></ul>

# gamstat

---

**Purpose** Gamma mean and variance

**Syntax** `[M,V] = gamstat(A,B)`

**Description** `[M,V] = gamstat(A,B)` returns the mean of and variance for the gamma distribution with shape parameters in *A* and scale parameters in *B*. *A* and *B* can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of *M* and *V*. A scalar input for *A* or *B* is expanded to a constant array with the same dimensions as the other input.

The mean of the gamma distribution with parameters *a* and *b* is  $ab$ . The variance is  $ab^2$ .

**Examples** `[m,v] = gamstat(1:5,1:5)`

```
m =  
    1    4    9   16   25  
v =  
    1    8   27   64  125
```

```
[m,v] = gamstat(1:5,1./(1:5))
```

```
m =  
    1    1    1    1    1  
v =  
  1.0000  0.5000  0.3333  0.2500  0.2000
```

**See Also** `gampdf` | `gamcdf` | `gaminv` | `gamfit` | `gamlike` | `gamrnd`

**How To** • “Gamma Distribution” on page B-43



**Purpose** Greater than or equal relation for handles

**Syntax** `h1 >= h2`

**Description** `h1 >= h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `>=` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = ge(h1, h2)` stores the result in a logical array of the same dimensions.

**See Also** `grandstream | eq | gt | le | lt | ne`

# prob.GeneralizedExtremeValueDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Generalized extreme value probability distribution object

**Description** `prob.GeneralizedExtremeValueDistribution` is an object consisting of parameters, a model description, and sample data for a generalized extreme value probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('GeneralizedExtremeValue')` creates a generalized extreme value probability distribution object using the default parameter values.

`pd = makedist('GeneralizedExtremeValue','k',k,'sigma',sigma,'mu',mu)` creates a generalized extreme value probability distribution object using the specified parameter values.

## Input Arguments

### **k** - Shape parameter

0 (default) | scalar value

Shape parameter for the generalized extreme value distribution, specified as a scalar value.

### **Data Types**

single | double

### **sigma** - Scale parameter

1 (default) | nonnegative scalar value

Scale parameter for the generalized extreme value distribution, specified as a nonnegative scalar value.

### **Data Types**

single | double

# prob.GeneralizedExtremeValueDistribution

---

## **mu - Location parameter**

0 (default) | scalar value

Location parameter for the generalized extreme value distribution, specified as a scalar value.

## **Data Types**

single | double

## **Properties**

### **k**

Shape parameter of the generalized extreme value distribution, stored as a scalar value.

## **Data Types**

single | double

### **sigma**

Scale parameter of the generalized extreme value distribution, stored as a nonnegative scalar value.

## **Data Types**

single | double

### **mu**

Location parameter of the generalized extreme value distribution, stored as a scalar value.

## **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## **Data Types**

char

### **InputData**

# prob.GeneralizedExtremeValueDistribution

---

Data used for distribution fitting, stored as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

# prob.GeneralizedExtremeValueDistribution

---

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the ParameterNames array is not fixed. If 1, the corresponding parameter in the ParameterNames array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

# prob.GeneralizedExtremeValueDistribution

---

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object

# prob.GeneralizedExtremeValueDistribution

std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Generalized Extreme Value Distribution

The generalized extreme value distribution is often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. It combines three simpler distributions into a single form, allowing a continuous range of possible shapes that include all three of the simpler distributions.

The three distribution types correspond to the limiting distribution of block maxima from different classes of underlying distributions:

- Type 1 — Distributions whose tails decrease exponentially, such as the normal distribution
- Type 2 — Distributions whose tails decrease as a polynomial, such as Student's  $t$  distribution
- Type 3 — Distributions whose tails are finite, such as the beta distribution

The generalized extreme value distribution uses the following parameters.

Parameter	Description	Support
k	Shape parameter	$-\infty \leq k \leq \infty$
sigma	Scale parameter	$\sigma \geq 0$
mu	Location parameter	$-\infty \leq \mu \leq \infty$

# prob.GeneralizedExtremeValueDistribution

---

The probability density function (pdf) for a Type 1 distribution, where shape parameter  $k = 0$ , is

$$f(x | 0, \mu, \sigma) = \left(\frac{1}{\sigma}\right) \exp\left(-\exp\left(-\frac{(x-\mu)}{\sigma}\right) - \frac{(x-\mu)}{\sigma}\right) ; \quad -\infty < x < \infty.$$

When  $k \neq 0$ , the pdf is

$$f(x | k, \mu, \sigma) = \left(\frac{1}{\sigma}\right) \exp\left[-\left(1 + k \frac{(x-\mu)}{\sigma}\right)^{-\frac{1}{k}}\right] \left(1 + k \frac{(x-\mu)}{\sigma}\right)^{-1-\frac{1}{k}}$$

for

$$1 + k \frac{(x-\mu)}{\sigma} > 0.$$

For the Type 2 case,  $k > 0$  and  $x \geq \mu - \frac{\sigma}{k}$ . For the Type 3 case,  $k < 0$

and  $x < \mu - \frac{\sigma}{k}$ .

## Examples

### Create a Generalized Extreme Value Distribution Object Using Default Parameters

Create a generalized extreme value distribution object using the default parameter values.

```
pd = makedist('GeneralizedExtremeValue')
```

```
pd =
```

```
GeneralizedExtremeValueDistribution
```

```
Generalized Extreme Value distribution  
k = 0
```



# prob.GeneralizedExtremeValueDistribution

---

```
sigma = 1  
mu = 0
```

## Create a Generalized Extreme Value Distribution Object Using Specified Parameters

Create a generalized extreme value distribution object by specifying values for the parameters.

```
pd = makedist('GeneralizedExtremeValue','k',0,'sigma',2,'mu',1)
```

```
pd =
```

```
GeneralizedExtremeValueDistribution
```

```
Generalized Extreme Value distribution
```

```
k = 0  
sigma = 2  
mu = 1
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
2.1544
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Generalized Extreme Value Distribution” on page B-48
- Class Attributes
- Property Attributes

# GeneralizedLinearModel

---

**Purpose** Generalized linear regression model class

**Description** An object comprising training data, model description, diagnostic information, and fitted coefficients for a generalized linear regression. Predict model responses with the `predict` or `feval` methods.

**Construction** `mdl = fitglm(ds)` or `mdl = fitglm(X,y)` creates a generalized linear model of a dataset array `ds`, or of the responses `y` to a data matrix `X`. For details, see `fitglm`.

`mdl = stepwiseglm(ds)` or `mdl = stepwiseglm(X,y)` creates a generalized linear model of a dataset array `ds`, or of the responses `y` to a data matrix `X`, with unimportant predictors excluded. For details, see `stepwiseglm`.

## Input Arguments

### **ds** - Input data

dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **Data Types**

single | double | logical

### **X** - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

### Data Types

single | double | logical

### **y** - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### Data Types

single | double

## Properties

### CoefficientCovariance

Covariance matrix of coefficient estimates.

### CoefficientNames

Cell array of strings containing a label for each coefficient.

### Coefficients

Table of coefficient values in a dataset array. **Coefficients** has one row for each coefficient and these columns:

- Estimate — Estimated coefficient value
- SE — Standard error of the estimate
- tStat —  $t$  statistic for a test that the coefficient is zero
- pValue —  $p$ -value for the  $t$  statistic

# GeneralizedLinearModel

---

To obtain any of these columns as a vector, index into the property using dot notation. For example, in `mdl` the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use `coefTest` to perform other tests on the coefficients.

## Deviance

Deviance of the fit. It is useful for comparing two models when one is a special case of the other. The difference between the deviance of the two models has a chi-square distribution with degrees of freedom equal to the difference in the number of estimated parameters between the two models. For more information on deviance, see “Deviance” on page 21-1266.

## DFE

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients.

## Diagnostics

Dataset array with diagnostics helpful in finding outliers and influential observations. The structure contains the following fields:

Field	Meaning	Utility
Leverage	Diagonal elements of <code>HatMatrix</code>	Leverage indicates to what extent the predicted value for an observation is determined by the observed value for that observation. A value close to 1 indicates that the prediction is largely determined by that observation, with little contribution from the other observations. A value close to 0 indicates the fit is largely determined by the other observations. For a model with $p$ coefficients and $n$ observations, the average value of Leverage is $p/n$ . An observation

Field	Meaning	Utility
		with Leverage larger than $2*p/n$ can be an outlier.
CooksDistance	Cook's measure of scaled change in fitted values	CooksDistance is a measure of scaled change in fitted values. An observation with CooksDistance larger than three times the mean Cook's distance can be an outlier.
HatMatrix	Projection matrix to compute fitted from observed responses	HatMatrix is an $n$ -by- $n$ matrix such that $\text{Fitted} = \text{HatMatrix} * Y$ , where $Y$ is the response vector and $\text{Fitted}$ is the vector of fitted response values.

All of these quantities are computed on the scale of the linear predictor. So, for example, in the equation that defines the hat matrix,

```
Yfit = glm.Fitted.LinearPredictor
Y = glm.Fitted.LinearPredictor + glm.Residuals.LinearPredictor
```

## Dispersion

Scale factor of the variance of the response. Dispersion multiplies the variance function for the distribution.

For example, the variance function for the binomial distribution is  $p(1-p)/n$ , where  $p$  is the probability parameter and  $n$  is the sample size parameter. If Dispersion is near 1, the variance of the data appears to agree with the theoretical variance of the binomial distribution. If Dispersion is larger than 1, the data are "overdispersed" relative to the binomial distribution.

## DispersionEstimated

Logical value indicating whether fitglm used the Dispersion property to compute standard errors for the coefficients in Coefficients.SE. If DispersionEstimated is false, fitglm used the theoretical value of the variance.

# GeneralizedLinearModel

---

- DispersionEstimated can be false only for 'binomial' or 'poisson' distributions.
- Set DispersionEstimated by setting the DispersionFlag name-value pair in fitglm.

## Distribution

Structure with the following fields relating to the generalized distribution:

Field	Description
Name	Name of the distribution, one of 'normal', 'binomial', 'poisson', 'gamma', or 'inverse gamma'.
DevianceFunction	Function that computes the components of the deviance as a function of the fitted parameter values and the response values.
VarianceFunction	Function that computes the theoretical variance for the distribution as a function of the fitted parameter values. When DispersionEstimated is true, Dispersion multiplies the variance function in the computation of the coefficient standard errors.

## Fitted

Table of predicted (fitted) values based on the training data, a dataset array with one row for each observation and the following columns.

Field	Description
Response	Predicted values on the scale of the response.
LinearPredictor	Predicted values on the scale of the linear predictor. These are the same as the link function applied to the Response fitted values.
Probability	Fitted probabilities (this column is included only with the binomial distribution).

To obtain any of the columns as a vector, index into the property using dot notation. For example, in the model `mdl`, the vector `f` of fitted values on the response scale is

```
f = mdl.Fitted.Response
```

Use `predict` to compute predictions for other predictor values, or to compute confidence bounds on `Fitted`.

## Formula

Object containing information about the model.

## Link

Structure with fields relating to the link function. The link is a function  $f$  that links the distribution parameter  $\mu$  to the fitted linear combination  $Xb$  of the predictors:

$$f(\mu) = Xb.$$

The structure has the following fields.

Field	Description
Name	Name of the link function, or '' if you specified the link as a function handle rather than a string.
LinkFunction	The function that defines $f$ , a function handle.
DevianceFunction	Derivative of $f$ , a function handle.
VarianceFunction	Inverse of $f$ , a function handle.

## LogLikelihood

Log likelihood of the model distribution at the response values, with mean fitted from the model, and other parameters estimated as part of the model fit.

## ModelCriterion

# GeneralizedLinearModel

---

AIC and other information criteria for comparing models. A structure with fields:

- AIC — Akaike information criterion
- AICc — Akaike information criterion corrected for sample size
- BIC — Bayesian information criterion
- CAIC — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the AIC value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

## **NumCoefficients**

Number of coefficients in the model, a positive integer. `NumCoefficients` includes coefficients that are set to zero when the model terms are rank deficient.

## **NumEstimatedCoefficients**

Number of estimated coefficients in the model, a positive integer. `NumEstimatedCoefficients` does not include coefficients that are set to zero when the model terms are rank deficient. `NumEstimatedCoefficients` is the degrees of freedom for regression.

## **NumObservations**

Number of observations the fitting function used in fitting. This is the number of observations supplied in the original dataset or matrix, minus any excluded rows (set with the `Excluded` name-value pair) or rows with missing values.

## **NumPredictors**

Number of variables `fitlm` used as predictors for fitting.

## **NumVariables**



Number of variables in the data. `NumVariables` is the number of variables in the original dataset when the fit is based on a dataset, or the total number of columns in the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

## ObservationInfo

Dataset with the same number of rows as the input data (ds or X).

Field	Description
Weights	Observation weights. Default is all 1.
Excluded	Logical value, 1 indicates an observation that you excluded from the fit with the <code>Exclude</code> name-value pair.
Missing	Logical value, 1 indicates a missing value in the input. Missing values are not used in the fit.
Subset	Logical value, 1 indicates the observation is not excluded or missing, so is used in the fit.

## ObservationNames

Cell array of strings containing the names of the observations used in the fit.

- If the fit is based on a dataset containing observation names, `ObservationNames` uses those names.
- Otherwise, `ObservationNames` is an empty cell array

## Offset

Vector with the same length as the number of rows in the data, passed from `fitglm` or `stepwiseglm` in the `Offset` name-value pair. The fitting function used `Offset` as a predictor variable, but

# GeneralizedLinearModel

---

with the coefficient set to exactly 1. In other words, the formula for fitting was

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the Offset predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor A. By using the log link function and by specifying  $\log(A)$  as an offset, you can force the model to satisfy this theoretical constraint.

## PredictorNames

Cell array of strings, the names of the predictors used in fitting the model.

## Residuals

Dataset array containing a table of residuals, with one row for each observation and these variables.

Field	Description
Raw	Observed minus fitted values.
LinearPredictor	Residuals on the linear predictor scale, equal to the adjusted response value minus the fitted linear combination of the predictors.
Pearson	Raw residuals divided by the estimated standard deviation of the response.
Anscombe	Residuals defined on transformed data with the transformation chosen to remove skewness.
Deviance	Residuals based on the contribution of each observation to the deviance.

To obtain any of these columns as a vector, index into the property using dot notation. For example, in a model `mdl`, the ordinary raw residual vector `r` is:

```
r = mdl.Residuals.Raw
```

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exceptions:

- `raw` contains the difference between the observed and predicted values.
- `standardized` is the residual, standardized in the usual way.
- `studentized` matches the standardized values because this residual is not used in the estimate of the residual standard deviation.

## **ResponseName**

String giving naming the response variable.

## **Rsquared**

Proportion of total sum of squares explained by the model. The ordinary R-squared value relates to the SSR and SST properties:

$$\text{Rsquared} = \text{SSR}/\text{SST} = 1 - \text{SSE}/\text{SST}.$$

`Rsquared` is a structure with two fields:

- `Ordinary` — Ordinary (unadjusted) R-squared
- `Adjusted` — R-squared adjusted for the number of coefficients

# GeneralizedLinearModel

---

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

```
r2 = mdl.Rsquared.Adjusted
```

## SSE

Sum of squared errors (residuals).

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

## SSR

Regression sum of squares, the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

## SST

Total sum of squares, the sum of squared deviations of  $y$  from  $\text{mean}(y)$ .

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

## Steps

Structure that is empty unless `stepwiselm` constructed the model.

Field	Description
Start	Formula representing the starting model
Lower	Formula representing the lower bound model, these terms that must remain in the model

Field	Description
Upper	Formula representing the upper bound model, model cannot contain more terms than Upper
Criterion	Criterion used for the stepwise algorithm, such as 'sse'
PEnter	Value of the parameter, such as 0.05
PRemove	Value of the parameter, such as 0.10
History	Dataset representing the steps taken in the fit

The History dataset has one row for each step including the initial fit, and the following variables (columns).

Field	Description
Action	Action taken during this step, one of: <ul style="list-style-type: none"><li>• 'Start' — First step</li><li>• 'Add' — A term is added</li><li>• 'Remove' — A term is removed</li></ul>
TermName	<ul style="list-style-type: none"><li>• 'Start' step: The starting model specification</li><li>• 'Add' or 'Remove' steps: The term moved in that step</li></ul>
Terms	Terms matrix (see <code>modelspec</code> of <code>fitlm</code> )
DF	Regression degrees of freedom after this step
delDF	Change in regression degrees of freedom from previous step (negative for steps that remove a term)
Deviance	Deviance (residual sum of squares) at that step

# GeneralizedLinearModel

---

Field	Description
FStat	$F$ statistic that led to this step
PValue	$p$ -value of the $F$ statistic

## VariableInfo

Dataset array containing metadata about Variables. There is one row for each term in the model, and the following columns.

Field	Description
Class	String giving variable class, such as 'double'
Range	Cell array giving variable range: <ul style="list-style-type: none"><li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li><li>• Categorical variable — Cell array of distinct variable values</li></ul>
InModel	Logical vector, where true indicates the variable is in the model
IsCategorical	Logical vector, where true indicates a categorical variable

## VariableNames

Cell array of strings containing names of the variables in the fit.

- If the fit is based on a dataset, this property provides the names of the variables in that dataset.
- If the fit is based on a predictor matrix and response vector, VariableNames is the values in the VarNames name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

## Variables

Dataset array containing the data, both observations and responses, that the fitting function used to construct the fit. If the fit is based on a dataset array, `Variables` is a copy of that dataset. Otherwise, `Variables` is a dataset created from the input data matrix `X` and response vector `y`.

## Methods

<code>addTerms</code>	Add terms to generalized linear model
<code>coefCI</code>	Confidence intervals of coefficient estimates of generalized linear model
<code>coefTest</code>	Linear hypothesis test on generalized linear regression model coefficients
<code>devianceTest</code>	Analysis of deviance
<code>disp</code>	Display generalized linear regression model
<code>feval</code>	Evaluate generalized linear regression model prediction
<code>fit</code>	Create generalized linear regression model
<code>plotDiagnostics</code>	Plot diagnostics of generalized linear regression model
<code>plotResiduals</code>	Plot residuals of generalized linear regression model
<code>plotSlice</code>	Plot of slices through fitted generalized linear regression surface

# GeneralizedLinearModel

---

predict	Predict response of generalized linear regression model
random	Simulate responses for generalized linear regression model
removeTerms	Remove terms from generalized linear model
step	Improve generalized linear regression model by adding or removing terms
stepwise	Create generalized linear regression model by stepwise regression

## Definitions Canonical Link Function

The default link function for a generalized linear model is the *canonical link function*.

## Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$



## Hat Matrix

The *hat matrix*  $H$  is defined in terms of the data matrix  $X$  and a diagonal weight matrix  $W$ :

$$H = X(X^T W X)^{-1} X^T W^T.$$

$W$  has diagonal elements  $w_i$ :

$$w_i = \frac{g'(\mu_i)}{\sqrt{V(\mu_i)}},$$

where

- $g$  is the link function mapping  $y_i$  to  $x_i b$ .
- $g'$  is the derivative of the link function  $g$ .
- $V$  is the variance function.
- $\mu_i$  is the  $i$ th mean.

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

## Leverage

The *leverage* of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

# Generalized Linear Model

---

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = w_i \frac{e_i^2}{p\hat{\phi}} \frac{h_{ii}}{(1-h_{ii})^2},$$

where

- $\hat{\phi}$  is the dispersion parameter (estimated or theoretical).
- $e_i$  is the linear predictor residual,  $g(y_i) - x_i\hat{\beta}$ , where
  - $g$  is the link function.
  - $y_i$  is the observed response.
  - $x_i$  is the observation.
  - $\hat{\beta}$  is the estimated coefficient vector.
- $p$  is the number of coefficients in the regression model.
- $h_{ii}$  is the  $i$ th diagonal element of the Hat Matrix  $H$ .

## Deviance

Deviance of a model  $M_1$  is twice the difference between the loglikelihood of that model and the saturated model,  $M_S$ . The saturated model is the model with the maximum number of parameters that can be estimated. For example, if there are  $n$  observations  $y_i$ ,  $i = 1, 2, \dots, n$ , with potentially different values for  $X_i^T\beta$ , then you can define a saturated model with  $n$  parameters. Let  $L(b, y)$  denote the maximum value of the likelihood function for a model. Then the deviance of model  $M_1$  is

$$-2(\log L(b_1, y) - \log L(b_S, y)),$$

where  $b_1$  are the estimated parameters for model  $M_1$  and  $b_S$  are the estimated parameters for the saturated model. The deviance has a chi-square distribution with  $n - p$  degrees of freedom, where  $n$  is the

number of parameters in the saturated model and  $p$  is the number of parameters in model  $M_1$ .

If  $M_1$  and  $M_2$  are two different generalized linear models, then the fit of the models can be assessed by comparing the deviances  $D_1$  and  $D_2$  of these models. The difference of the deviances is

$$\begin{aligned} D &= D_2 - D_1 = -2(\log L(b_2, y) - \log L(b_S, y)) + 2(\log L(b_1, y) - \log L(b_S, y)) \\ &= -2(\log L(b_2, y) - \log L(b_1, y)). \end{aligned}$$

Asymptotically, this difference has a chi-square distribution with degrees of freedom  $v$  equal to the number of parameters that are estimated in one model but fixed (typically at 0) in the other. That is, it is equal to the difference in the number of parameters estimated in  $M_1$  and  $M_2$ . You can get the  $p$ -value for this test using `1 - chi2cdf(D, V)`, where  $D = D_2 - D_1$ .

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Fit a Generalized Linear Model

Fit a logistic regression model of probability of smoking as a function of age, weight, and sex, using a two-way interactions model.

Load the hospital dataset array.

```
load hospital
ds = hospital; % just to use the ds name
```

Specify the model using a formula that allows up to two-way interactions.

```
modelspec = 'Smoker ~ Age*Weight*Sex - Age:Weight:Sex';
```

Create the generalized linear model.

```
mdl = fitglm(ds, modelspec, 'Distribution', 'binomial')
```

# GeneralizedLinearModel

---

```
mdl =
```

```
Generalized Linear regression model:
```

```
logit(Smoker) ~ 1 + Sex*Age + Sex*Weight + Age*Weight
```

```
Distribution = Binomial
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	-6.0492	19.749	-0.3063	0.75938
Sex_Male	-2.2859	12.424	-0.18399	0.85402
Age	0.11691	0.50977	0.22934	0.81861
Weight	0.031109	0.15208	0.20455	0.83792
Sex_Male:Age	0.020734	0.20681	0.10025	0.92014
Sex_Male:Weight	0.01216	0.053168	0.22871	0.8191
Age:Weight	-0.00071959	0.0038964	-0.18468	0.85348

```
100 observations, 93 error degrees of freedom
```

```
Dispersion: 1
```

```
Chi^2-statistic vs. constant model: 5.07, p-value = 0.535
```

The large  $p$ -value indicates the model might not differ statistically from a constant.

## Create a Generalized Linear Model Stepwise

Create response data using just three of 20 predictors, and create a generalized linear model stepwise to see if it uses just the correct predictors.

Create data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:,[5 10 15])*[.4;.2;.3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = stepwiseglm(X,y,...  
    'constant','upper','linear','Distribution','poisson')
```

1. Adding x5, Deviance = 134.439, Chi2Stat = 52.24814, PValue = 4.891229e-13
2. Adding x15, Deviance = 106.285, Chi2Stat = 28.15393, PValue = 1.1204e-07
3. Adding x10, Deviance = 95.0207, Chi2Stat = 11.2644, PValue = 0.000790094

```
mdl =
```

Generalized Linear regression model:

```
log(y) ~ 1 + x5 + x10 + x15  
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0115	0.064275	15.737	8.4217e-56
x5	0.39508	0.066665	5.9263	3.0977e-09
x10	0.18863	0.05534	3.4085	0.0006532
x15	0.29295	0.053269	5.4995	3.8089e-08

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 91.7, p-value = 9.61e-20

## See Also

[fitglm](#) | [stepwiseglmLinearModel](#) | [NonLinearModel](#) |

## Related Examples

- “Generalized Linear Model Workflow” on page 10-42

## Concepts

- “Generalized Linear Models” on page 10-12

# prob.GeneralizedParetoDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Generalized Pareto probability distribution object

**Description** prob.GeneralizedParetoDistribution is an object consisting of parameters, a model description, and sample data for a generalized Pareto probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('GeneralizedParetoDistribution')` creates a generalized Pareto probability distribution object using default parameter values.

`pd = makedist('GeneralizedParetoDistribution','k',k,'sigma',sigma,'theta',theta)` creates a generalized Pareto probability distribution object using the specified parameter values.

## Input Arguments

### **k - Shape parameter**

1 (default) | scalar value

Shape parameter for the generalized Pareto distribution, specified as a scalar value.

### **Data Types**

single | double

### **sigma - Scale parameter**

1 (default) | nonnegative scalar value

Scale parameter for the generalized Pareto distribution, specified as a nonnegative scalar value.

### **Data Types**

single | double

## **theta - Location parameter**

1 (default) | scalar value

Location parameter for the generalized Pareto distribution, specified as a scalar value.

## **Data Types**

single | double

## **Properties**

### **k**

Shape parameter for the generalized Pareto distribution, stored as a scalar value.

## **Data Types**

single | double

### **sigma**

Scale parameter for the generalized Pareto distribution, stored as a nonnegative scalar value.

## **Data Types**

single | double

### **theta**

Location parameter for the generalized Pareto distribution, stored as a scalar value.

## **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## **Data Types**

char

### **InputData**

# prob.GeneralizedParetoDistribution

---

Data used for distribution fitting, stored as a structure containing the following:

- `data`: Data vector used for distribution fitting.
- `cens`: Censoring vector, or empty if none.
- `freq`: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.



**Data Types**

single | double

**ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

**Data Types**

char

**ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the ParameterNames array is not fixed. If 1, the corresponding parameter in the ParameterNames array is fixed. This property is read-only.

**Data Types**

logical

**ParameterNames**

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

**Data Types**

char

**ParameterValues**

Values of distribution parameters, stored as a vector. This property is read-only.

**Data Types**

single | double

**Truncation**

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

# prob.GeneralizedParetoDistribution

---

## Data Types

single | double

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object

std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Generalized Pareto Distribution

The generalized Pareto distribution is used to model the tails of another distribution. It allows a continuous range of possible shapes that include both the exponential and Pareto distributions as special cases. It has three basic forms, each corresponding to a limiting distribution of exceedence data from a different class of underlying distributions.

- Distributions whose tails decrease exponentially, such as the normal, lead to a generalized Pareto shape parameter of zero.
- Distributions whose tails decrease polynomially, such as the Student's  $t$ , lead to a positive shape parameter.
- Distributions whose tails are finite, such as the beta, lead to a negative shape parameter.

The generalized Pareto distribution uses the following parameters.

Parameter	Description	Support
k	Shape parameter	$-\infty < k < \infty$
sigma	Scale parameter	$\sigma \geq 0$
theta	Location parameter	$-\infty < \theta < \infty$

The probability density function (pdf) of the generalized Pareto distribution with shape parameter  $k \neq 0$  is

# prob.GeneralizedParetoDistribution

---

$$f(x | k, \sigma, \theta) = \left(\frac{1}{\sigma}\right) \left(1 + k \frac{(x - \theta)}{\sigma}\right)^{-1 - \frac{1}{k}}$$

for  $x > \theta$ , when  $k > 0$ , or for  $\theta < x < -\frac{\sigma}{k}$ , when  $k < 0$ .

For  $k = 0$ , the pdf is

$$y = f(x | 0, \sigma, \theta) = \left(\frac{1}{\sigma}\right) \exp\left(-\frac{(x - \theta)}{\sigma}\right)$$

for  $x > \theta$ .

If  $k = 0$  and  $\theta = 0$ , the generalized Pareto distribution is equivalent

to the exponential distribution. If  $k > 0$  and  $\theta = \frac{\sigma}{k}$ , the generalized Pareto distribution is equivalent to the Pareto distribution.

## Examples

### Create a Generalized Pareto Distribution Object Using Default Parameters

Create a generalized Pareto distribution object using the default parameter values.

```
pd = makedist('GeneralizedPareto')
```

```
pd =
```

```
GeneralizedParetoDistribution
```

```
Generalized Pareto distribution
```

```
    k = 1  
    sigma = 1  
    theta = 1
```

## Create a Generalized Pareto Distribution Object Using Specified Parameters

Create a generalized Pareto distribution object by specifying parameter values.

```
pd = makedist('GeneralizedPareto','k',0,'sigma',2,'theta',1)
```

```
pd =
```

```
GeneralizedParetoDistribution
```

```
Generalized Pareto distribution
```

```
    k = 0
```

```
    sigma = 2
```

```
    theta = 1
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
    2.1544
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Generalized Pareto Distribution” on page B-53
- Class Attributes
- Property Attributes

# geocdf

---

**Purpose** Geometric cumulative distribution function

**Syntax**  
`y = geocdf(x,P)`  
`y = geocdf(x,P, 'upper')`

**Description** `y = geocdf(x,P)` computes the geometric cdf at each of the values in `x` using the corresponding probabilities in `P`. `x` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `P` must lie on the interval `[0 1]`.

`y = geocdf(x,P, 'upper')` returns the complement of the geometric cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The geometric cdf is

$$y = F(x | p) = \sum_{i=0}^{\text{floor}(x)} pq^i$$

where  $q = 1 - p$ .

The result, `y`, is the probability of observing up to `x` trials before a success, when the probability of success in any given trial is `p`.

## Examples **Compute Geometric Distribution CDF**

Suppose you toss a fair coin repeatedly. If the coin lands face up (heads), that is a success. What is the probability of observing three or fewer tails before getting a heads?

```
p = geocdf(3,0.5)
```

```
p =  
0.9375
```

**See Also** `cdf` | `geopdf` | `geoinv` | `geostat` | `geornd` | `mle`

**How To**

- “Geometric Distribution” on page B-57

**Purpose** Geometric inverse cumulative distribution function

**Syntax** `X = geoinv(Y,P)`

**Description** `X = geoinv(Y,P)` returns the smallest positive integer  $X$  such that the geometric cdf evaluated at  $X$  is equal to or exceeds  $Y$ . You can think of  $Y$  as the probability of observing  $X$  successes in a row in independent trials where  $P$  is the probability of success in each trial.

$Y$  and  $P$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for  $P$  or  $Y$  is expanded to a constant array with the same dimensions as the other input. The values in  $P$  and  $Y$  must lie on the interval  $[0\ 1]$ .

**Examples** The probability of correctly guessing the result of 10 coin tosses in a row is less than 0.001 (unless the coin is not fair).

```
psychic = geoinv(0.999,0.5)
psychic =
     9
```

The example below shows the inverse method for generating random numbers from the geometric distribution.

```
rndgeo = geoinv(rand(2,5),0.5)
rndgeo =
     0     1     3     1     0
     0     1     0     2     0
```

**See Also** `icdf` | `geocdf` | `geopdf` | `geostat` | `geornd`

**How To** • “Geometric Distribution” on page B-57



**Purpose** Geometric mean

**Syntax** `m = geomean(x)`  
`geomean(X,dim)`

**Description** `m = geomean(x)` calculates the geometric mean of a sample. For vectors, `geomean(x)` is the geometric mean of the elements in `x`. For matrices, `geomean(X)` is a row vector containing the geometric means of each column. For N-dimensional arrays, `geomean` operates along the first nonsingleton dimension of `X`.

`geomean(X,dim)` takes the geometric mean along the dimension `dim` of `X`.

The geometric mean is

$$m = \left[ \prod_{i=1}^n x_i \right]^{\frac{1}{n}}$$

**Examples** The arithmetic mean is greater than or equal to the geometric mean.

```
x = exprnd(1,10,6);
```

```
geometric = geomean(x)
```

```
geometric =  
    0.7466    0.6061    0.6038    0.2569    0.7539    0.3478
```

```
average = mean(x)
```

```
average =  
    1.3509    1.1583    0.9741    0.5319    1.0088    0.8122
```

**See Also** `mean` | `median` | `harmmean` | `trimmean`

**How To** • “Geometric Distribution” on page B-57

**Purpose** Geometric probability density function

**Syntax** `Y = geopdf(X,P)`

**Description** `Y = geopdf(X,P)` computes the geometric pdf at each of the values in `X` using the corresponding probabilities in `P`. `X` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `P` must lie on the interval `[0 1]`.

The geometric pdf is

$$y = f(x | p) = pq^x I_{(0,1,\dots)}(x)$$

where  $q = 1 - p$ .

**Examples** Suppose you toss a fair coin repeatedly. If the coin lands face up (heads), that is a success. What is the probability of observing exactly three tails before getting a heads?

```
p = geopdf(3,0.5)
```

```
p =  
0.0625
```

**See Also** `pdf` | `geocdf` | `geoinv` | `geostat` | `geornd`

**How To** • “Geometric Distribution” on page B-57

**Purpose** Geometric random numbers

**Syntax**

```
R = geornd(P)
R = geornd(P,m,n,...)
R = geornd(P,[m,n,...])
```

**Description** `R = geornd(P)` generates geometric random numbers with probability parameter `P`. `P` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `P`. The geometric distribution is useful when you want to model the number of successive failures preceding a success, where the probability of success in any given trial is the constant `P`. The parameters in `P` must lie in the interval `[0 1]`.

`R = geornd(P,m,n,...)` or `R = geornd(P,[m,n,...])` generates an `m`-by-`n`-by-... array containing random numbers from the geometric distribution with probability parameter `P`. `P` can be a scalar or an array of the same size as `R`.

**Examples**

```
r1 = geornd(1 ./ 2.^(1:6))
r1 =
     2    10     2     5     2    60
```

```
r2 = geornd(0.01,[1 5])
r2 =
    65    18   334   291    63
```

```
r3 = geornd(0.5,1,6)
r3 =
     0     7     1     3     1     0
```

**See Also** `random` | `geopdf` | `geocdf` | `geoinv` | `geostat`

**How To**

- “Geometric Distribution” on page B-57

# geostat

---

**Purpose** Geometric mean and variance

**Syntax** `[M,V] = geostat(P)`

**Description** `[M,V] = geostat(P)` returns the mean of and variance for the geometric distribution with corresponding probabilities in P.

The mean of the geometric distribution with parameter  $p$  is  $q/p$ , where  $q = 1-p$ . The variance is  $q/p^2$ .

**Examples**

```
[m,v] = geostat(1./(1:6))
m =
    0    1.0000    2.0000    3.0000    4.0000    5.0000
v =
    0    2.0000    6.0000   12.0000   20.0000   30.0000
```

**See Also** `geopdf` | `geocdf` | `geoinv` | `geornd`

**How To**

- “Geometric Distribution” on page B-57

**Superclasses** ClusterCriterion

**Purpose** Gap criterion clustering evaluation object

**Description** clustering.evaluation.GapEvaluation is an object consisting of sample data, clustering data, and gap criterion values used to evaluate the optimal number of clusters. Create a gap criterion clustering evaluation object using evalclusters.

**Construction** `eva = evalclusters(x,clust,'Gap')` creates a gap criterion clustering evaluation object.

`eva = evalclusters(x,clust,'Gap',Name,Value)` creates a gap criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

### **x - Input data**

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

### **Data Types**

single | double

### **clust - Clustering algorithm**

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

# clustering.evaluation.GapEvaluation

---

'kmeans'	Cluster the data in $X$ using the kmeans clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in $X$ using the clusterdata agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in $X$ using the gmdistribution Gaussian mixture distribution algorithm, with 'SharedCov' set to true and 'Replicates' set to 5.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using the function\_handle (@) operator. The function must be of the form

```
C = clustfun(DATA,K)
```

where DATA is the data to be clustered, and K is the number of clusters. The output of clustfun must be one of the following:

- A vector of integers representing the cluster index for each observation in DATA. There must be K unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify clust as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'KList',[1:5],'Distance','cityblock'` specifies to test 1, 2, 3, 4, and 5 clusters using the sum of absolute differences distance measure.

## B - Number of reference data sets

100 (default) | positive integer value

Number of reference data sets generated from the reference distribution `ReferenceDistribution`, specified as the comma-separated pair consisting of `'B'` and a positive integer value.

**Example:** `'B',150`

## Distance - Distance metric

`'sqEuclidean'` (default) | `'Euclidean'` | `'cityblock'` | function | ...

Distance metric used for computing the criterion values, specified as the comma-separated pair consisting of `'Distance'` and one of the following.

<code>'sqEuclidean'</code>	Squared Euclidean distance
<code>'Euclidean'</code>	Euclidean distance
<code>'cityblock'</code>	Sum of absolute differences
<code>'cosine'</code>	One minus the cosine of the included angle between points (treated as vectors)
<code>'correlation'</code>	One minus the sample correlation between points (treated as sequences of values)

# clustering.evaluation.GapEvaluation

---

'Hamming'	Percentage of coordinates that differ
'Jaccard'	Percentage of nonzero coordinates that differ

For detailed information about each distance metric, see `pdist`.

You can also specify a function for the distance metric by using the `function_handle` (`@`) operator. The distance function must be of the form

```
d2 = distfun(XI,XJ),
```

where `XI` is a 1-by- $n$  vector corresponding to a single row of the input matrix `X`, and `XJ` is an  $m_2$ -by- $n$  matrix corresponding to multiple rows of `X`. `distfun` must return an  $m_2$ -by-1 vector of distances `d2`, whose  $k$ th element is the distance between `XI` and `XJ(k, :)`.

If `Criterion` is 'silhouette', you can also specify `Distance` as the output vector output created by the function `pdist`.

When `Clust` a string representing a built-in clustering algorithm, `evalclusters` uses the distance metric specified for `Distance` to cluster the data, except for the following:

- If `Clust` is 'linkage', and `Distance` is either 'sqEuclidean' or 'Euclidean', then the clustering algorithm uses Euclidean distance and Ward linkage.
- If `Clust` is 'linkage' and `Distance` is any other metric, then the clustering algorithm uses the specified distance metric and average linkage.

In all other cases, the distance metric specified for `Distance` must match the distance metric used in the clustering algorithm to obtain meaningful results.

**Example:** 'Distance', 'Euclidean'

**KList - List of number of clusters to evaluate**



vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of 'KList' and a vector of positive integer values. You must specify KList when `clust` is a clustering algorithm name string or a function handle. When `criterion` is 'gap', `clust` must be a string or a function handle, and you must specify KList.

**Example:** 'KList',[1:6]

## **ReferenceDistribution - Reference data generation method**

'PCA' (default) | 'uniform'

Reference data generation method, specified as the comma-separated pair consisting of 'ReferenceDistributions' and one of the following.

'PCA'	Generate reference data from a uniform distribution over a box aligned with the principal components of the data matrix <code>x</code> .
'uniform'	Generate reference data uniformly over the range of each feature in the data matrix <code>x</code> .

**Example:** 'ReferenceDistribution','uniform'

## **SearchMethod - Method for selecting optimal number of clusters**

'firstmaxse' | 'globalmaxse'

Method for selecting the optimal number of clusters, specified as the comma-separated pair consisting of 'SearchMethod' and one of the following.

# clustering.evaluation.GapEvaluation

---

'firstmaxse'

Evaluate each proposed number of clusters in `KList` and select the smallest number of clusters satisfying

$$\text{Gap}(K) \geq \text{Gap}(K + 1) - \text{SE}(K + 1),$$

where  $K$  is the number of clusters,  $\text{Gap}(K)$  is the gap value for the clustering solution with  $K$  clusters, and  $\text{SE}(K + 1)$  is the standard error of the clustering solution with  $K + 1$  clusters. This is the default if the sorted `KList` contains consecutive integer values.

'globalmaxse'

Evaluate each proposed number of clusters in `KList` and select the smallest number of clusters satisfying

$$\text{Gap}(K) \geq \text{GAPMAX} - \text{SE}(\text{GAPMAX}),$$

where  $K$  is the number of clusters,  $\text{Gap}(K)$  is the gap value for the clustering solution with  $K$  clusters,  $\text{GAPMAX}$  is the largest gap value, and  $\text{SE}(\text{GAPMAX})$  is the standard error corresponding to the largest gap value. This is the default if `KList` contains only one value, or contains any nonconsecutive integer values.

**Example:** 'SearchMethod', 'globalmaxse'

## Properties

**B**

Number of data sets generated from the reference distribution, stored as a positive integer value.

## ClusteringFunction

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name string or function handle. If the clustering solutions are provided in the input, ClusteringFunction is empty.

## **CriterionName**

Name of the criterion used for clustering evaluation, stored as a valid criterion name string.

## **CriterionValues**

Criterion values corresponding to each proposed number of clusters in InspectedK, stored as a vector of numerical values.

## **Distance**

Distance measure used for clustering data, stored as a valid distance measure name string.

## **ExpectedLogW**

Expectation of the natural logarithm of  $W$  based on the generated reference data, stored as a vector of scalar values.  $W$  is the within-cluster dispersion computed using the distance measurement Distance.

## **InspectedK**

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

## **LogW**

Natural logarithm of  $W$  based on the input data, stored as a vector of scalar values.  $W$  is the within-cluster dispersion computed using the distance measurement Distance.

## **Missing**

Logical flag for excluded data, stored as a column vector of logical values. If Missing equals true, then the corresponding value in the data matrix  $x$  is not used in the clustering solution.

# clustering.evaluation.GapEvaluation

---

## **NumObservations**

Number of observations in the data matrix  $X$ , minus the number of missing (NaN) values in  $X$ , stored as a positive integer value.

## **OptimalK**

Optimal number of clusters, stored as a positive integer value.

## **OptimalY**

Optimal clustering solution corresponding to **OptimalK**, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, **OptimalY** is empty.

## **ReferenceDistribution**

Reference data generation method, stored as a valid reference distribution name string.

## **SE**

Standard error of the natural logarithm of  $W$  with respect to the reference data for each number of clusters in **InspectedK**, stored as a vector of scalar values.  $W$  is the within-cluster dispersion computed using the distance measurement **Distance**.

## **SearchMethod**

Method for determining the optimal number of clusters, stored as a valid search method name string.

## **StdLogW**

Standard deviation of the natural logarithm of  $W$  with respect to the reference data for each number of clusters in **InspectedK**.  $W$  is the within-cluster dispersion computed using the distance measurement **Distance**.

## **X**

Data used for clustering, stored as a matrix of numerical values.

## Methods

### Inherited Methods

compact	Compact clustering evaluation object
plot	Plot clustering evaluation object criterion values

## Definitions

### Gap Value

A common graphical approach to cluster evaluation involves plotting an error measurement versus several proposed numbers of clusters, and locating the “elbow” of this plot. The “elbow” occurs at the most dramatic decrease in error measurement. The gap criterion formalizes this approach by estimating the “elbow” location as the number of clusters with the largest gap value. Therefore, under the gap criterion, the optimal number of clusters occurs at the solution with the largest local or global gap value within a tolerance range.

The gap value is defined as

$$\text{Gap}_n(k) = E_n^* \{ \log(W_k) \} - \log(W_k),$$

where  $n$  is the sample size,  $k$  is the number of clusters being evaluated, and  $W_k$  is the pooled within-cluster dispersion measurement

$$W_k = \sum_{r=1}^k \frac{1}{2n_r} D_r,$$

where  $n_r$  is the number of data points in cluster  $r$ , and  $D_r$  is the sum of the pairwise distances for all points in cluster  $r$ .

The expected value  $E_n^* \{ \log(W_k) \}$  is determined by Monte Carlo sampling from a reference distribution, and  $\log(W_k)$  is computed from the sample data.

The gap value is defined even for clustering solutions that contain only one cluster, and can be used with any distance metric. However, the

# clustering.evaluation.GapEvaluation

---

gap criterion is more computationally expensive than other cluster evaluation criteria, because the clustering algorithm must be applied to the reference data for each proposed clustering solution.

## Examples

### Evaluate the Clustering Solution Using Gap Criterion

Evaluate the optimal number of clusters using the gap clustering evaluation criterion.

Load the sample data.

```
load fisheriris;
```

The data contains sepal and petal measurements from three species of iris flowers.

Evaluate the number of clusters based on the gap criterion values. Cluster the data using kmeans.

```
rng('default'); % For reproducibility
eva = evalclusters(meas,'kmeans','gap','KList',[1:6])
```

```
eva =
```

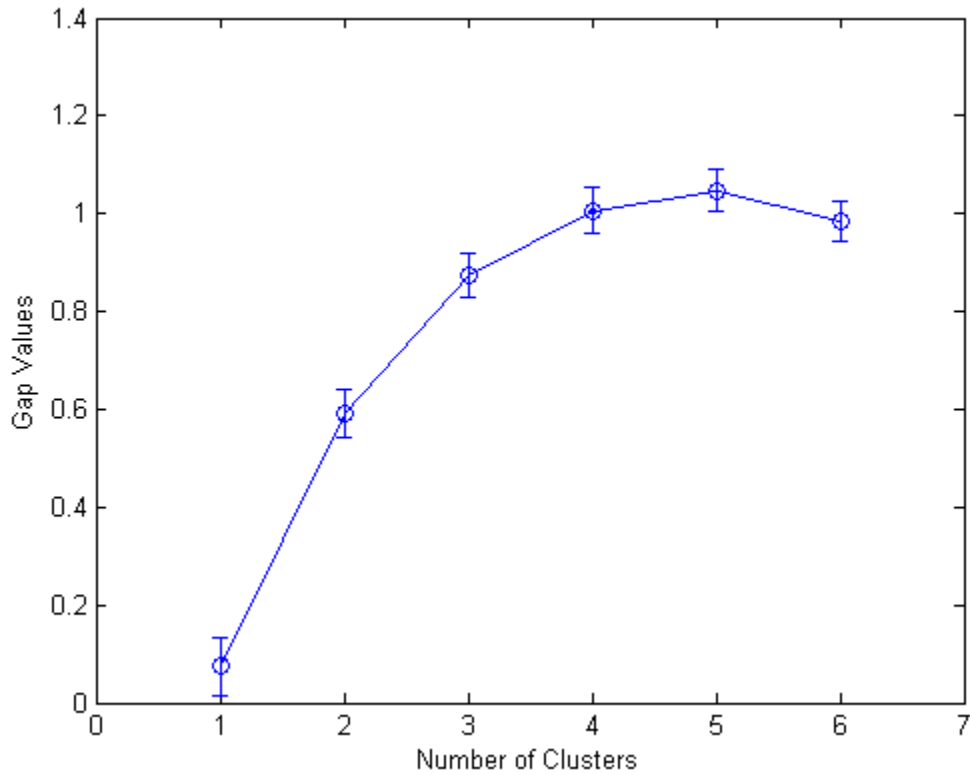
```
GapEvaluation with properties:
```

```
NumObservations: 150
InspectecedK: [1 2 3 4 5 6]
CriterionValues: [1x6 double]
OptimalK: 4
```

The OptimalK value indicates that, based on the gap criterion, the optimal number of clusters is three.

Plot the gap criterion values for each number of clusters tested.

```
figure;
plot(eva);
```



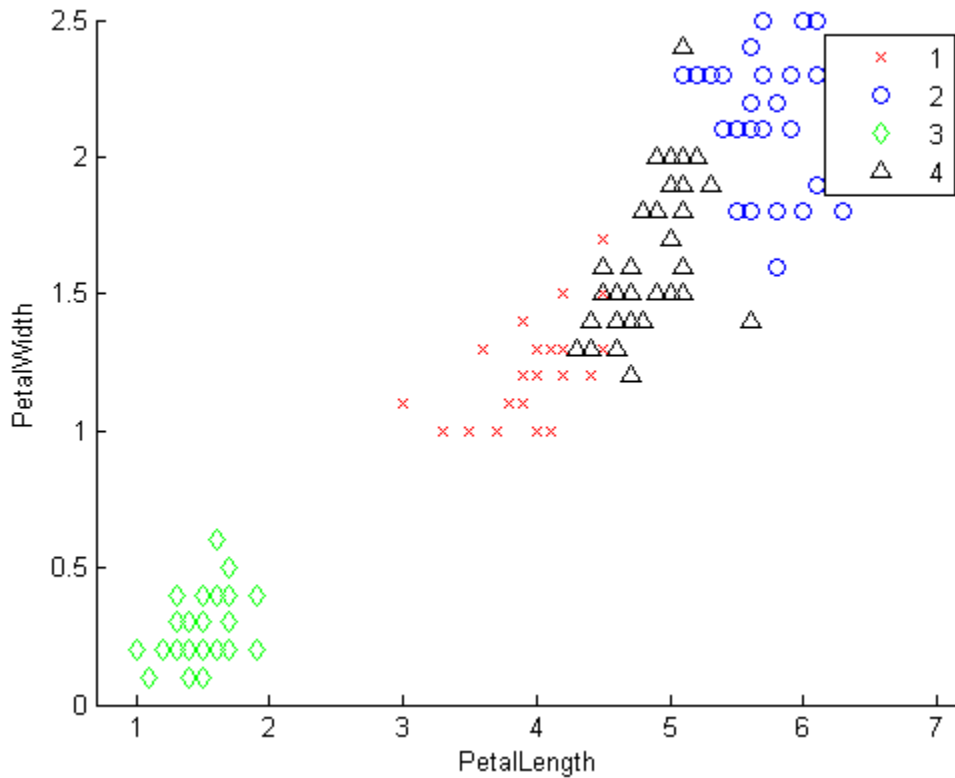
Based on the plot, the maximum value of the gap criterion occurs at five clusters. However, the value at four clusters is within one standard error of the maximum, so the suggested optimal number of clusters is four.

Create a grouped scatter plot to examine the relationship between petal length and width. Group the data by suggested clusters.

```
figure;  
PetalLength = meas(:,3);
```

# clustering.evaluation.GapEvaluation

```
PetalWidth = meas(:,4);  
ClusterGroup = eva.OptimalY;  
figure;  
gscatter(PetalLength,PetalWidth,ClusterGroup,'rbgk','xod^');
```



The plot shows cluster 1 in the lower-left corner, completely separated from the other three clusters. Cluster 1 contains flowers with the smallest petal widths and lengths. Cluster 4 is in the upper-right corner, and contains flowers with the largest petal widths and lengths.



Clusters 2 and 3 are near the center of the plot, and contain flowers with measurements between the two extremes.

## References

[1] Tibshirani, R., G. Walther, and T. Hastie. “Estimating the number of clusters in a data set via the gap statistic.” *Journal of the Royal Statistical Society: Series B*. Vol. 63, Part 2, 2001, pp. 411–423.

## See Also

[evalclusters](#) |  
[evalclustersclustering.evaluation.CalinskiHarabaszEvaluation](#)  
| [clustering.evaluation.DaviesBouldinEvaluation](#) |  
[clustering.evaluation.SilhouetteEvaluation](#) |

## Concepts

- Class Attributes
- Property Attributes

# dataset.get

---

**Purpose** Access dataset array properties

**Syntax**

```
get(A)
s = get(A)
p = get(A,PropertyName)
p = get(A,{PropertyName1,PropertyName2,...})
```

**Description**

`get(A)` displays a list of property/value pairs for the dataset array `A`.

`s = get(A)` returns the values in a scalar structure `s` with field names given by the properties.

`p = get(A,PropertyName)` returns the value of the property specified by the string `PropertyName`.

`p = get(A,{PropertyName1,PropertyName2,...})` allows multiple property names to be specified and returns their values in a cell array.

**Examples** Create a dataset array from Fisher's iris data and access the information:

```
load fisheriris
NumObs = size(meas,1);
NameObs = strcat({'Obs'},num2str((1:NumObs)','%-d'));
iris = dataset({nominal(species),'species'},...
              {meas,'SL','SW','PL','PW'},...
              'ObsNames',NameObs);

get(iris)
Description: ''
Units: {}
DimNames: {'Observations' 'Variables'}
UserData: []
ObsNames: {150x1 cell}
VarNames: {'species' 'SL' 'SW' 'PL' 'PW'}

ON = get(iris,'ObsNames');
ON(1:3)
```

```
ans =  
  'Obs1'  
  'Obs2'  
  'Obs3'
```

**See Also**

set | summary

# categorical.getlabels

---

**Purpose** Access categorical array labels

**Syntax** `labels = getlabels(A)`

**Description** `labels = getlabels(A)` returns the labels of the levels in the categorical array `A` as a cell array of strings `labels`. For ordinal `A`, the labels are returned in the order of the levels.

## **Examples** **Example 1**

Display levels in a nominal and an ordinal array:

```
standings = nominal({'Leafs','Canadiens','Bruins'});
getlabels(standings)
ans =
    'Bruins'    'Canadiens'    'Leafs'
```

```
standings = ordinal(1:3,{'Leafs','Canadiens','Bruins'});
getlabels(standings)
ans =
    'Leafs'    'Canadiens'    'Bruins'
```

## **Example 2**

Display age groups containing data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)','%d'),{'s'});
AgeGroup = ordinal(hospital.Age,labels,[],edges);
AgeGroup = droplevels(AgeGroup);
getlabels(AgeGroup)
ans =
    '20s'    '30s'    '40s'    '50s'
```

**See Also** `getlevels` | `setlabels`

**Purpose** Get categorical array levels

**Syntax** `S = getlevels(A)`

**Description** `S = getlevels(A)` returns the levels for the categorical array `A`. `S` is a vector with the same type as `A`.

**See Also** `getlabels`

# gevcdf

---

<b>Purpose</b>	Generalized extreme value cumulative distribution function
<b>Syntax</b>	<pre>p = gevcdf(x,k,sigma,mu) p = gevcdf(x,k,sigma,mu, 'upper')</pre>
<b>Description</b>	<p><code>p = gevcdf(x,k,sigma,mu)</code> returns the cdf of the generalized extreme value (GEV) distribution with shape parameter <code>k</code>, scale parameter <code>sigma</code>, and location parameter, <code>mu</code>, evaluated at the values in <code>x</code>. The size of <code>p</code> is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.</p> <p><code>p = gevcdf(x,k,sigma,mu, 'upper')</code> returns the complement of the cdf of the GEV distribution, using an algorithm that more accurately computes the extreme upper tail probabilities.</p> <p>Default values for <code>k</code>, <code>sigma</code>, and <code>mu</code> are 0, 1, and 0, respectively.</p> <p>When <math>k &lt; 0</math>, the GEV is the type III extreme value distribution. When <math>k &gt; 0</math>, the GEV distribution is the type II, or Frechet, extreme value distribution. If <math>w</math> has a Weibull distribution as computed by the <code>wblcdf</code> function, then <math>-w</math> has a type III extreme value distribution and <math>1/w</math> has a type II extreme value distribution. In the limit as <math>k</math> approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the <code>evcdf</code> function.</p> <p>The mean of the GEV distribution is not finite when <math>k \geq 1</math>, and the variance is not finite when <math>k \geq 1/2</math>. The GEV distribution has positive density only for values of <math>X</math> such that <math>k*(X-mu)/sigma &gt; -1</math>.</p>
<b>References</b>	<p>[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. <i>Modelling Extremal Events for Insurance and Finance</i>. New York: Springer, 1997.</p> <p>[2] Kotz, S., and S. Nadarajah. <i>Extreme Value Distributions: Theory and Applications</i>. London: Imperial College Press, 2000.</p>
<b>See Also</b>	<code>cdf</code>   <code>gevpdf</code>   <code>gevinv</code>   <code>gevstat</code>   <code>gevfit</code>   <code>gevlike</code>   <code>gevrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Generalized Extreme Value Distribution” on page B-48</li></ul>

**Purpose** Generalized extreme value parameter estimates

**Syntax**

```
parmhat = gevfit(X)
[parmhat,parmci] = gevfit(X)
[parmhat,parmci] = gevfit(X,alpha)
[...] = gevfit(X,alpha,options)
```

**Description**

`parmhat = gevfit(X)` returns maximum likelihood estimates of the parameters for the generalized extreme value (GEV) distribution given the data in X. `parmhat(1)` is the shape parameter, `k`, `parmhat(2)` is the scale parameter, `sigma`, and `parmhat(3)` is the location parameter, `mu`.

`[parmhat,parmci] = gevfit(X)` returns 95% confidence intervals for the parameter estimates.

`[parmhat,parmci] = gevfit(X,alpha)` returns  $100(1-\alpha)\%$  confidence intervals for the parameter estimates.

`[...] = gevfit(X,alpha,options)` specifies control parameters for the iterative algorithm used to compute ML estimates. This argument can be created by a call to `statset`. See `statset('gevfit')` for parameter names and default values. Pass in `[]` for `alpha` to use the default values.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblfit` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evfit` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution is defined for  $k*(X-\mu)/\sigma > -1$ .

**References**

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

## See Also

mle | gevlike | gevpdf | gevCDF | gevinv | gevstat | gevrnd

## How To

- “Generalized Extreme Value Distribution” on page B-48



---

<b>Purpose</b>	Generalized extreme value inverse cumulative distribution function
<b>Syntax</b>	<code>X = gevinv(P,k,sigma,mu)</code>
<b>Description</b>	<p><code>X = gevinv(P,k,sigma,mu)</code> returns the inverse cdf of the generalized extreme value (GEV) distribution with shape parameter <code>k</code>, scale parameter <code>sigma</code>, and location parameter <code>mu</code>, evaluated at the values in <code>P</code>. The size of <code>X</code> is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.</p> <p>Default values for <code>k</code>, <code>sigma</code>, and <code>mu</code> are 0, 1, and 0, respectively.</p> <p>When <math>k &lt; 0</math>, the GEV is the type III extreme value distribution. When <math>k &gt; 0</math>, the GEV distribution is the type II, or Frechet, extreme value distribution. If <math>w</math> has a Weibull distribution as computed by the <code>wblinv</code> function, then <math>-w</math> has a type III extreme value distribution and <math>1/w</math> has a type II extreme value distribution. In the limit as <math>k</math> approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the <code>evinv</code> function.</p> <p>The mean of the GEV distribution is not finite when <math>k \geq 1</math>, and the variance is not finite when <math>k \geq 1/2</math>. The GEV distribution has positive density only for values of <math>X</math> such that <math>k*(X-mu)/sigma &gt; -1</math>.</p>
<b>References</b>	<p>[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. <i>Modelling Extremal Events for Insurance and Finance</i>. New York: Springer, 1997.</p> <p>[2] Kotz, S., and S. Nadarajah. <i>Extreme Value Distributions: Theory and Applications</i>. London: Imperial College Press, 2000.</p>
<b>See Also</b>	<code>icdf</code>   <code>gevcdf</code>   <code>gevpdf</code>   <code>gevstat</code>   <code>gevfit</code>   <code>gevlike</code>   <code>gevrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Generalized Extreme Value Distribution” on page B-48</li></ul>

# gevlike

---

<b>Purpose</b>	Generalized extreme value negative log-likelihood
<b>Syntax</b>	<pre>nlogL = gevlike(params,data) [nlogL,ACOV] = gevlike(params,data)</pre>
<b>Description</b>	<p><code>nlogL = gevlike(params,data)</code> returns the negative of the log-likelihood <code>nlogL</code> for the generalized extreme value (GEV) distribution, evaluated at parameters <code>params</code>. <code>params(1)</code> is the shape parameter, <code>k</code>, <code>params(2)</code> is the scale parameter, <code>sigma</code>, and <code>params(3)</code> is the location parameter, <code>mu</code>.</p> <p><code>[nlogL,ACOV] = gevlike(params,data)</code> returns the inverse of Fisher's information matrix, <code>ACOV</code>. If the input parameter values in <code>params</code> are the maximum likelihood estimates, the diagonal elements of <code>ACOV</code> are their asymptotic variances. <code>ACOV</code> is based on the observed Fisher's information, not the expected information.</p> <p>When <math>k &lt; 0</math>, the GEV is the type III extreme value distribution. When <math>k &gt; 0</math>, the GEV distribution is the type II, or Frechet, extreme value distribution. If <math>w</math> has a Weibull distribution as computed by the <code>wbllike</code> function, then <math>-w</math> has a type III extreme value distribution and <math>1/w</math> has a type II extreme value distribution. In the limit as <math>k</math> approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the <code>evlike</code> function.</p> <p>The mean of the GEV distribution is not finite when <math>k \geq 1</math>, and the variance is not finite when <math>k \geq 1/2</math>. The GEV distribution has positive density only for values of <math>X</math> such that <math>k*(X-\mu)/\sigma &gt; -1</math>.</p>
<b>References</b>	<p>[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. <i>Modelling Extremal Events for Insurance and Finance</i>. New York: Springer, 1997.</p> <p>[2] Kotz, S., and S. Nadarajah. <i>Extreme Value Distributions: Theory and Applications</i>. London: Imperial College Press, 2000.</p>
<b>See Also</b>	<code>gevfit</code>   <code>gevpdf</code>   <code>gevcdf</code>   <code>gevinv</code>   <code>gevstat</code>   <code>gevrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Generalized Extreme Value Distribution” on page B-48</li></ul>

<b>Purpose</b>	Generalized extreme value probability density function
<b>Syntax</b>	$Y = \text{gevpdf}(X, k, \text{sigma}, \mu)$
<b>Description</b>	<p><math>Y = \text{gevpdf}(X, k, \text{sigma}, \mu)</math> returns the pdf of the generalized extreme value (GEV) distribution with shape parameter <math>k</math>, scale parameter <math>\text{sigma}</math>, and location parameter, <math>\mu</math>, evaluated at the values in <math>X</math>. The size of <math>Y</math> is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.</p> <p>Default values for <math>k</math>, <math>\text{sigma}</math>, and <math>\mu</math> are 0, 1, and 0, respectively.</p> <p>When <math>k &lt; 0</math>, the GEV is the type III extreme value distribution. When <math>k &gt; 0</math>, the GEV distribution is the type II, or Frechet, extreme value distribution. If <math>w</math> has a Weibull distribution as computed by the <code>wblpdf</code> function, then <math>-w</math> has a type III extreme value distribution and <math>1/w</math> has a type II extreme value distribution. In the limit as <math>k</math> approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the <code>evcdf</code> function.</p> <p>The mean of the GEV distribution is not finite when <math>k \geq 1</math>, and the variance is not finite when <math>k \geq 1/2</math>. The GEV distribution has positive density only for values of <math>X</math> such that <math>k*(X-\mu)/\text{sigma} &gt; -1</math>.</p>
<b>References</b>	<p>[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. <i>Modelling Extremal Events for Insurance and Finance</i>. New York: Springer, 1997.</p> <p>[2] Kotz, S., and S. Nadarajah. <i>Extreme Value Distributions: Theory and Applications</i>. London: Imperial College Press, 2000.</p>
<b>See Also</b>	<code>pdf</code>   <code>gevcdf</code>   <code>gevinv</code>   <code>gevstat</code>   <code>gevfit</code>   <code>gevlike</code>   <code>gevrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Generalized Extreme Value Distribution” on page B-48</li></ul>

# gevrnd

---

**Purpose** Generalized extreme value random numbers

**Syntax**  
R = gevrnd(k, sigma, mu)  
R = gevrnd(k, sigma, mu, m, n, ...)  
R = gevrnd(k, sigma, mu, [m, n, ...])

**Description** R = gevrnd(k, sigma, mu) returns an array of random numbers chosen from the generalized extreme value (GEV) distribution with shape parameter k, scale parameter sigma, and location parameter, mu. The size of R is the common size of the input arguments if all are arrays. If any parameter is a scalar, the size of R is the size of the other parameters.

R = gevrnd(k, sigma, mu, m, n, ...) or R = gevrnd(k, sigma, mu, [m, n, ...]) generates an m-by-n-by-... array containing random numbers from the GEV distribution with parameters k, sigma, and mu. The k, sigma, mu parameters can each be scalars or arrays of the same size as R.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If w has a Weibull distribution as computed by the wblrnd function, then -w has a type III extreme value distribution and 1/w has a type II extreme value distribution. In the limit as k approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the evrnd function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of X such that  $k*(X-mu)/sigma > -1$ .

**References**  
[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.  
[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

**See Also** random | gevpdf | gevCDF | gevinv | gevstat | gevfit | gevlike

**How To**

- “Generalized Extreme Value Distribution” on page B-48

# gevstat

---

**Purpose** Generalized extreme value mean and variance

**Syntax** `[M,V] = gevstat(k,sigma,mu)`

**Description** `[M,V] = gevstat(k,sigma,mu)` returns the mean of and variance for the generalized extreme value (GEV) distribution with shape parameter `k`, scale parameter `sigma`, and location parameter, `mu`. The sizes of `M` and `V` are the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `k`, `sigma`, and `mu` are 0, 1, and 0, respectively.

When  $k < 0$ , the GEV is the type III extreme value distribution. When  $k > 0$ , the GEV distribution is the type II, or Frechet, extreme value distribution. If  $w$  has a Weibull distribution as computed by the `wblstat` function, then  $-w$  has a type III extreme value distribution and  $1/w$  has a type II extreme value distribution. In the limit as  $k$  approaches 0, the GEV is the mirror image of the type I extreme value distribution as computed by the `evstat` function.

The mean of the GEV distribution is not finite when  $k \geq 1$ , and the variance is not finite when  $k \geq 1/2$ . The GEV distribution has positive density only for values of  $X$  such that  $k*(X-\mu)/\text{sigma} > -1$ .

**References** [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

**See Also** `gevpdf` | `gevcdf` | `gevinv` | `gevfit` | `gevlike` | `gevrnd`

**How To** • “Generalized Extreme Value Distribution” on page B-48

**Purpose**

Interactively add line to plot

**Syntax**

```
gline(h)
gline
hline = gline(...)
```

**Description**

`gline(h)` allows you to draw a line segment in the figure with handle `h` by clicking the pointer at the two endpoints. A rubber-band line tracks the pointer movement.

`gline` with no input arguments defaults to `h = gcf` and draws in the current figure.

`hline = gline(...)` returns the handle `hline` to the line.

**Examples**

Use `gline` to connect two points in a plot:

```
x = 1:10;

y = x + randn(1,10);
scatter(x,y,25,'b','*')

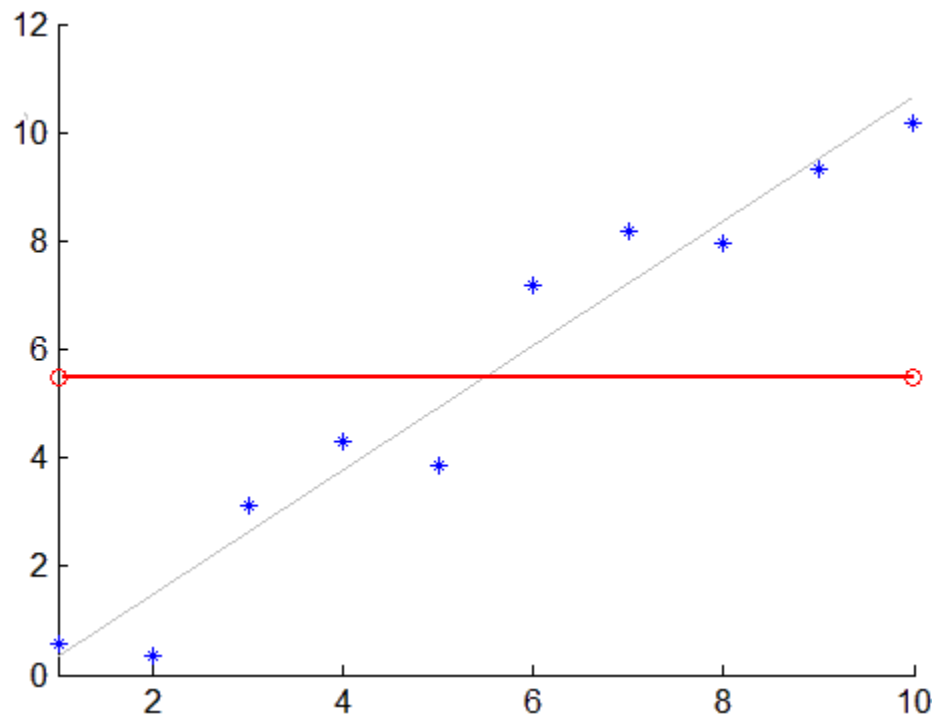
lsline

mu = mean(y);
hold on
plot([1 10],[mu mu],'ro')

hline = gline; % Connect circles
set(hline,'Color','r')
```

# gline

---



## See Also

`refline` | `refcurve` | `lsline`



**Purpose**

Generalized linear model regression

**Syntax**

```
b = glmfit(X,y,distr)
b = glmfit(X,y,distr,param1,val1,param2,val2,...)
[b,dev] = glmfit(...)
[b,dev,stats] = glmfit(...)
```

**Description**

`b = glmfit(X,y,distr)` returns a  $(p + 1)$ -by-1 vector `b` of coefficient estimates for a generalized linear regression of the responses in `y` on the predictors in `X`, using the distribution `distr`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `distr` can be any of the following strings: 'binomial', 'gamma', 'inverse gaussian', 'normal' (the default), and 'poisson'.

In most cases, `y` is an  $n$ -by-1 vector of observed responses. For the binomial distribution, `y` can be a binary vector indicating success or failure at each observation, or a two column matrix with the first column indicating the number of successes for each observation and the second column indicating the number of trials for each observation.

This syntax uses the canonical link (see below) to relate the distribution to the predictors.

---

**Note** By default, `glmfit` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`. You can change the default behavior of `glmfit` using the 'constant' parameter, below.

---

`glmfit` treats NaNs in either `X` or `y` as missing values, and ignores them.

`b = glmfit(X,y,distr,param1,val1,param2,val2,...)` additionally allows you to specify optional parameter name/value pairs to control the model fit. Acceptable parameters are as follows.

Parameter	Value	Description
'link'	'identity', default for the distribution 'normal'	$\mu = Xb$
	'log', default for the distribution 'poisson'	$\log(\mu) = Xb$
	'logit', default for the distribution 'binomial'	$\log(\mu/(1 - \mu)) = Xb$
	'probit'	$\text{norminv}(\mu) = Xb$
	'comploglog'	$\log(-\log(1 - \mu)) = Xb$
	'reciprocal', default for the distribution 'gamma'	$1/\mu = Xb$
	'loglog'	$\log(-\log(\mu)) = Xb$
	p (a number), default for the distribution 'inverse gaussian' (with $p = -2$ )	$\mu^p = Xb$
'link'	cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI).	Custom-defined link function. You must provide <ul style="list-style-type: none"> <li>• FL(mu)</li> <li>• FD = dFL(mu) / dmu</li> <li>• FI = FL^(-1)</li> </ul>

Parameter	Value	Description
'estdisp'	'on'	Estimates a dispersion parameter for the binomial or Poisson distribution.
	'off' (Default for binomial or Poisson distribution)	Uses the theoretical value of 1.0 for those distributions.
'offset'	Vector	Used as an additional predictor variable, but with a coefficient value fixed at 1.0.
'weights'	Vector of prior weights, such as the inverses of the relative variance of each observation	
'constant'	'on' (default)	Includes a constant term in the model. The coefficient of the constant term is the first element of b.
	'off'	Omit the constant term.

`[b,dev] = glmfit(...)` returns `dev`, the deviance of the fit at the solution vector. The deviance is a generalization of the residual sum of squares. It is possible to perform an analysis of deviance to compare several models, each a subset of the other, and to test whether the model with more terms is significantly better than the model with fewer terms.

`[b,dev,stats] = glmfit(...)` returns `dev` and `stats`.

`stats` is a structure with the following fields:

- `beta` — Coefficient estimates `b`
- `dfe` — Degrees of freedom for error
- `s` — Theoretical or estimated dispersion parameter

- `sfit` — Estimated dispersion parameter
- `se` — Vector of standard errors of the coefficient estimates `b`
- `coeffcorr` — Correlation matrix for `b`
- `covb` — Estimated covariance matrix for `B`
- `t` —  $t$  statistics for `b`
- `p` —  $p$ -values for `b`
- `resid` — Vector of residuals
- `residp` — Vector of Pearson residuals
- `residd` — Vector of deviance residuals
- `resida` — Vector of Anscombe residuals

If you estimate a dispersion parameter for the binomial or Poisson distribution, then `stats.s` is set equal to `stats.sfit`. Also, the elements of `stats.se` differ by the factor `stats.s` from their theoretical values.

## Examples

### Fit Generalized Linear Model with Probit Link

Enter sample data.

```
x = [2100 2300 2500 2700 2900 3100 ...  
     3300 3500 3700 3900 4100 4300]';  
n = [48 42 31 34 31 21 23 23 21 16 17 21]';  
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

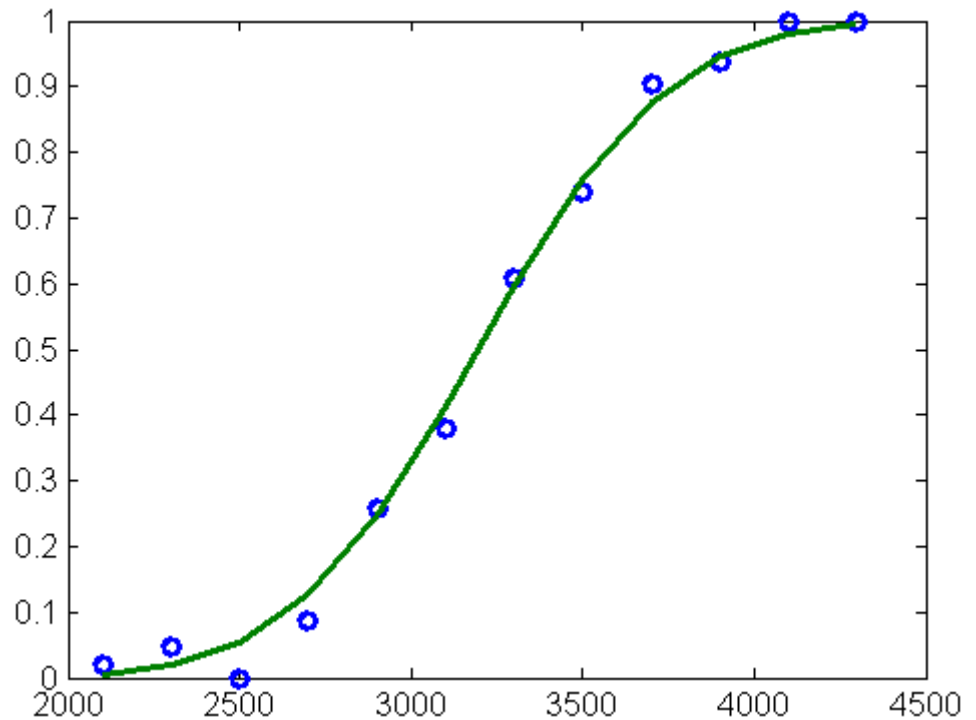
Each `y` value is the number of successes in corresponding number of trials `inn`, and `x` contains the predictor variable values.

Fit a probit regression model for `y` on `x`.

```
b = glmfit(x,[y n], 'binomial', 'link', 'probit');
```

Compute the estimated number of successes and plot the percent observed and estimated percent success versus the x values.

```
yfit = glmval(b,x,'probit','size',n);  
plot(x, y./n, 'o', x, yfit./n, '-','LineWidth',2)
```



### Use Custom-defined Link Function

Load the sample data.

```
load fisheriris
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists

of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Define the response and predictor variables.

```
X = meas(51:end,:);  
y = strcmp('versicolor',species(51:end));
```

Define three function handles, created using @, that define the link, the derivative of the link, and the inverse link for a logit link function, and store them in a cell array.

```
link = @(mu) log(mu ./ (1-mu));  
derlink = @(mu) 1 ./ (mu .* (1-mu));  
invlink = @(resp) 1 ./ (1 + exp(-resp));  
F = {link, derlink, invlink};
```

Fit a logistic regression using `glmfit` with the link function you defined.

```
b = glmfit(X,y,'binomial','link',F)
```

```
b =  
  
    42.6378  
     2.4652  
     6.6809  
    -9.4294  
   -18.2861
```

Now, fit a generalized linear model using the `logit` link function and compare the results.

```
b = glmfit(X,y,'binomial','link','logit')
```

```
b =  
  
    42.6378  
     2.4652  
     6.6809
```

---

-9.4294  
-18.2861

**References**

- [1] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.

**See Also**

glmval | regress | regstats | GeneralizedLinearModel | fitglm  
| stepwiseglm

# glmval

---

**Purpose** Generalized linear model values

**Syntax**  
`yhat = glmval(b,X,link)`  
`[yhat,dylo,dyhi] = glmval(b,X,link,stats)`  
`[...] = glmval(...,param1,val1,param2,val2,...)`

**Description** `yhat = glmval(b,X,link)` computes predicted values for the generalized linear model with link function `link` and predictors `X`. Distinct predictor variables should appear in different columns of `X`. `b` is a vector of coefficient estimates as returned by the `glmfit` function. `link` can be any of the strings or the custom-defined link functions used as values for the 'link' name-value pair argument in the `glmfit` function.

---

**Note** By default, `glmval` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`. You can change the default behavior of `glmval` using the 'constant' parameter, below.

---

`[yhat,dylo,dyhi] = glmval(b,X,link,stats)` also computes 95% confidence bounds for the predicted values. When the `stats` structure output of the `glmfit` function is specified, `dylo` and `dyhi` are also returned. `dylo` and `dyhi` define a lower confidence bound of `yhat-dylo`, and an upper confidence bound of `yhat+dyhi`. Confidence bounds are nonsimultaneous, and apply to the fitted curve, not to a new observation.

`[...] = glmval(...,param1,val1,param2,val2,...)` specifies optional parameter name/value pairs to control the predicted values. Acceptable parameters are:



Parameter	Value
'confidence' — the confidence level for the confidence bounds	A scalar between 0 and 1
'size' — the size parameter (N) for a binomial model	A scalar, or a vector with one value for each row of X
'offset' — used as an additional predictor variable, but with a coefficient value fixed at 1.0	A vector
'constant'	<ul style="list-style-type: none"> <li>'on' — Includes a constant term in the model. The coefficient of the constant term is the first element of b.</li> <li>'off' — Omit the constant term</li> </ul>
'simultaneous' — Compute simultaneous confidence intervals (true), or compute non-simultaneous confidence intervals (default false)	true or false

## Examples

### Fit Generalized Linear Model with Probit Link

Enter sample data.

```
x = [2100 2300 2500 2700 2900 3100 ...
      3300 3500 3700 3900 4100 4300]';
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

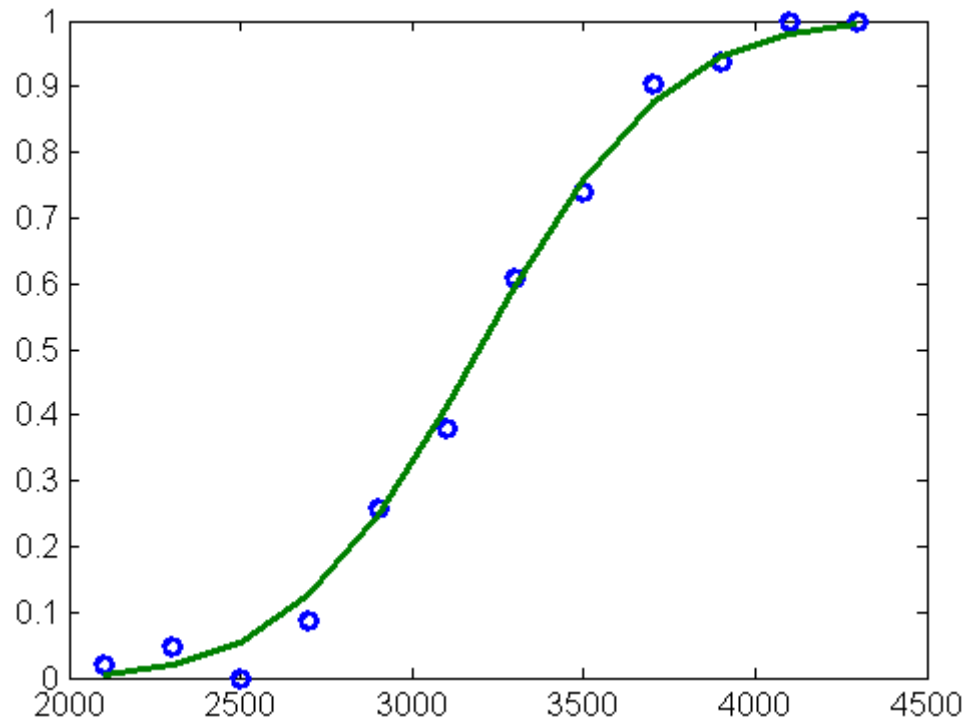
Each y value is the number of successes in corresponding number of trials inn, and x contains the predictor variable values.

Fit a generalized linear model for y on x using a probit link function.

```
b = glmfit(x,[y n],'binomial','link','probit');
```

Compute the estimated number of successes and plot the observed and estimated percent success versus the x values.

```
yfit = glmval(b,x,'probit','size',n);  
plot(x, y./n,'o',x,yfit./n,'-','LineWidth',2)
```



## Use Custom-defined Link Function

Enter sample data.

```
x = [2100 2300 2500 2700 2900 3100 ...  
     3300 3500 3700 3900 4100 4300]';  
n = [48 42 31 34 31 21 23 23 21 16 17 21]';
```

```
y = [1 2 0 3 8 8 14 17 19 15 17 21]';
```

Each  $y$  value is the number of successes in corresponding number of trials in  $n$ , and  $x$  contains the predictor variable values.

Now define three function handles, created using `@`, that define the link, the derivative of the link, and the inverse link for a probit link function, and store them in a cell array.

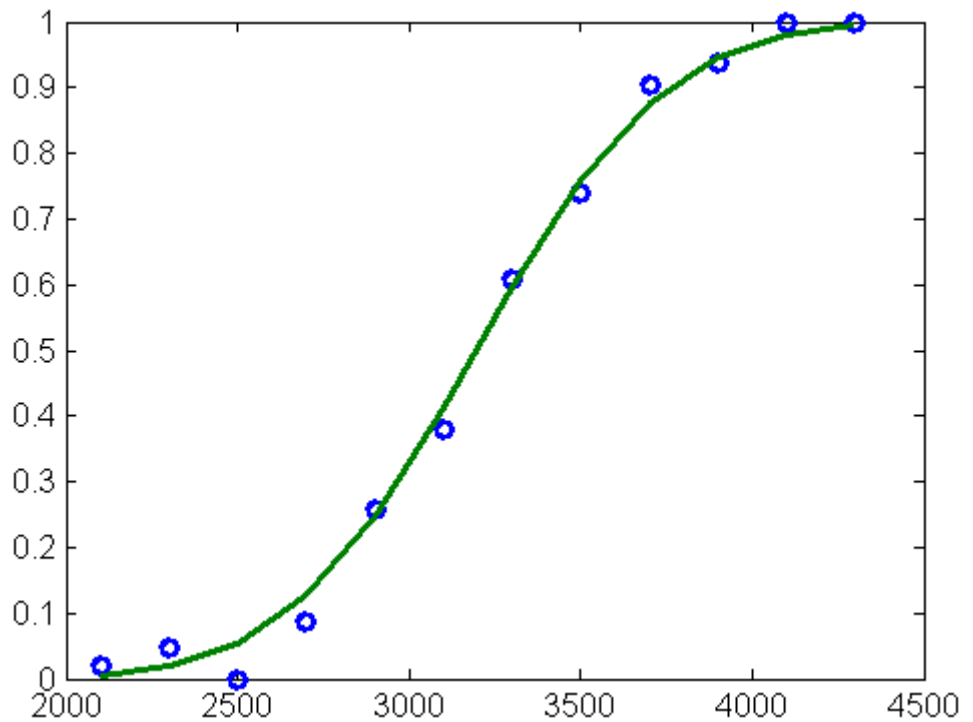
```
link = @(mu) norminv(mu);  
derlink = @(mu) 1 ./ normpdf(norminv(mu));  
invlink = @(resp) normcdf(resp);  
F = {link, derlink, invlink};
```

Fit a generalized linear model for  $y$  on  $x$  using the link function you defined.

```
b = glmfit(x,[y n],'binomial','link',F);
```

Compute the estimated number of successes and plot the observed and estimated percent success versus the  $x$  values.

```
yfit = glmval(b,x,F,'size',n);  
plot(x, y./n,'o',x,yfit./n,'-','LineWidth',2)
```



## References

- [1] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.

## See Also

`glmfit` | `GeneralizedLinearModel` | `fitglm` | `stepwiseglm`

## Purpose

Glyph plot

## Syntax

```
glyphplot(X)
glyphplot(X, 'glyph', 'face')
glyphplot(X, 'glyph', 'face', 'features', f)
glyphplot(X, ..., 'grid', [rows, cols])
glyphplot(X, ..., 'grid', [rows, cols], 'page', p)
glyphplot(X, ..., 'centers', C)
glyphplot(X, ..., 'centers', C, 'radius', r)
glyphplot(X, ..., 'obslabels', labels)
glyphplot(X, ..., 'standardize', method)
glyphplot(X, ..., prop1, val1, ...)
h = glyphplot(X, ...)
```

## Description

`glyphplot(X)` creates a star plot from the multivariate data in the  $n$ -by- $p$  matrix  $X$ . Rows of  $X$  correspond to observations, columns to variables. A star plot represents each observation as a “star” whose  $i$ th spoke is proportional in length to the  $i$ th coordinate of that observation. `glyphplot` standardizes  $X$  by shifting and scaling each column separately onto the interval  $[0, 1]$  before making the plot, and centers the glyphs on a rectangular grid that is as close to square as possible. `glyphplot` treats NaNs in  $X$  as missing values, and does not plot the corresponding rows of  $X$ . `glyphplot(X, 'glyph', 'star')` is a synonym for `glyphplot(X)`.

`glyphplot(X, 'glyph', 'face')` creates a face plot from  $X$ . A face plot represents each observation as a “face,” whose  $i$ th facial feature is drawn with a characteristic proportional to the  $i$ th coordinate of that observation. The features are described in “Face Features” on page 21-1327 Face Features.

`glyphplot(X, 'glyph', 'face', 'features', f)` creates a face plot where the  $i$ th element of the index vector  $f$  defines which facial feature will represent the  $i$ th column of  $X$ .  $f$  must contain integers from 0 to 17, where 0 indicate that the corresponding column of  $X$  should not be plotted. See “Face Features” on page 21-1327 for more information.

# glyphplot

---

`glyphplot(X, ..., 'grid', [rows, cols])` organizes the glyphs into a rows-by-cols grid.

`glyphplot(X, ..., 'grid', [rows, cols], 'page', p)` organizes the glyph into one or more pages of a rows-by-cols grid, and displays the page `p`. If `p` is a vector, `glyphplot` displays multiple pages in succession. If `p` is 'all', `glyphplot` displays all pages. If `p` is 'scroll', `glyphplot` displays a single plot with a scrollbar.

`glyphplot(X, ..., 'centers', C)` creates a plot with each glyph centered at the locations in the  $n$ -by-2 matrix `C`.

`glyphplot(X, ..., 'centers', C, 'radius', r)` creates a plot with glyphs positioned using `C`, and scale the glyphs so the largest has radius `r`.

`glyphplot(X, ..., 'obslabels', labels)` labels each glyph with the text in the character array or cell array of strings `labels`. By default, the glyphs are labelled 1:N. Use '' for blank labels.

`glyphplot(X, ..., 'standardize', method)` standardizes `X` before making the plot. Choices for `method` are

- 'column' — Maps each column of `X` separately onto the interval  $[0, 1]$ . This is the default.
- 'matrix' — Maps the entire matrix `X` onto the interval  $[0, 1]$ .
- 'PCA' — Transforms `X` to its principal component scores, in order of decreasing eigenvalue, and maps each one onto the interval  $[0, 1]$ .
- 'off' — No standardization. Negative values in `X` may make a star plot uninterpretable.

`glyphplot(X, ..., prop1, val1, ...)` sets properties to the specified property values for all line graphics objects created by `glyphplot`.

`h = glyphplot(X, ...)` returns a matrix of handles to the graphics objects created by `glyphplot`. For a star plot, `h(:, 1)` and `h(:, 2)` contain handles to the line objects for each star's perimeter and spokes, respectively. For a face plot, `h(:, 1)` and `h(:, 2)` contain object handles

to the lines making up each face and to the pupils, respectively. `h(:,3)` contains handles to the text objects for the labels, if present.

### Face Features

The following table describes the correspondence between the columns of the vector `f`, the value of the 'Features' input parameter, and the facial features of the glyph plot. If `X` has fewer than 17 columns, unused features are displayed at their default value.

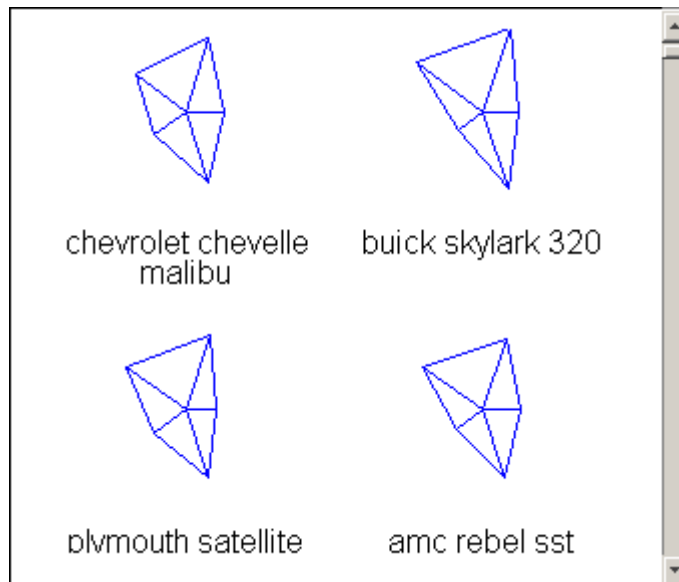
Column	Facial Feature
1	Size of face
2	Forehead/jaw relative arc length
3	Shape of forehead
4	Shape of jaw
5	Width between eyes
6	Vertical position of eyes
7	Height of eyes
8	Width of eyes (this also affects eyebrow width)
9	Angle of eyes (this also affects eyebrow angle)
10	Vertical position of eyebrows
11	Width of eyebrows (relative to eyes)
12	Angle of eyebrows (relative to eyes)
13	Direction of pupils
14	Length of nose
15	Vertical position of mouth
16	Shape of mouth
17	Mouth arc length

# glyphplot

## Examples

```
load carsmall
X = [Acceleration Displacement Horsepower MPG Weight];

glyphplot(X, 'standardize', 'column', ...
          'obslabels', Model, ...
          'grid', [2 2], ...
          'page', 'scroll');
```



```
glyphplot(X, 'glyph', 'face', ...
          'obslabels', Model, ...
          'grid', [2 3], ...
          'page', 9);
```





pontiac ventura sj



amc pacer d/i



volkswagen rabbit



datsun b-210



toyota corolla



ford pinto

## See Also

[andrewsplot](#) | [parallelcoords](#)

# gmdistribution

---

**Purpose** Gaussian mixture models

**Description** An object of the `gmdistribution` class defines a Gaussian mixture distribution, which is a multivariate distribution that consists of a mixture of one or more multivariate Gaussian distribution components. The number of components for a given `gmdistribution` object is fixed. Each multivariate Gaussian component is defined by its mean and covariance, and the mixture is defined by a vector of mixing proportions.

**Construction** To create a Gaussian mixture distribution by specifying the distribution parameters, use the `gmdistribution` constructor. To fit a Gaussian mixture distribution model to data, use `gmdistribution.fit`.

<code>fit</code>	Gaussian mixture parameter estimates
<code>gmdistribution</code>	Construct Gaussian mixture distribution

**Properties** All objects of the class have the properties listed in the following table.

<code>CovType</code>	Type of covariance matrices
<code>DistName</code>	Type of distribution
<code>mu</code>	Input matrix of means $\mu$
<code>NComponents</code>	Number $k$ of mixture components
<code>NDimensions</code>	Dimension $d$ of multivariate Gaussian distributions
<code>PComponents</code>	Input vector of mixing proportions
<code>SharedCov</code>	<code>true</code> if all covariance matrices are restricted to be the same
<code>Sigma</code>	Input array of covariances

Objects constructed with `fit` have the additional properties listed in the following table.

AIC	Akaike Information Criterion
BIC	Bayes Information Criterion
Converged	Determine if algorithm converged
Iters	Number of iterations
NlogL	Negative of log-likelihood
RegV	Value of 'Regularize' parameter

## Methods

<code>cdf</code>	Cumulative distribution function for Gaussian mixture distribution
<code>cluster</code>	Construct clusters from Gaussian mixture distribution
<code>disp</code>	Display Gaussian mixture distribution object
<code>display</code>	Display Gaussian mixture distribution object
<code>fit</code>	Gaussian mixture parameter estimates
<code>mahal</code>	Mahalanobis distance to component means
<code>pdf</code>	Probability density function for Gaussian mixture distribution
<code>posterior</code>	Posterior probabilities of components
<code>random</code>	Random numbers from Gaussian mixture distribution

subsasgn	Subscripted reference for Gaussian mixture distribution object
suboref	Subscripted reference for Gaussian mixture distribution object

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

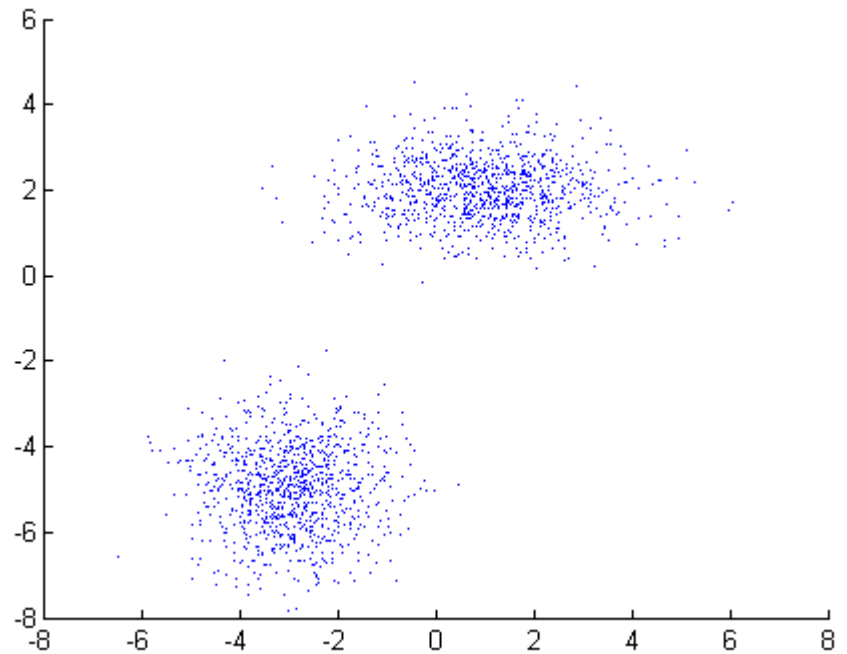
### Fit a Gaussian Mixture Model

Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function. Fit the resulting data.

Generate the data using 1000 points from each distribution.

```
MU1 = [1 2];  
SIGMA1 = [2 0; 0 .5];  
MU2 = [-3 -5];  
SIGMA2 = [1 0; 0 1];  
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];
```

```
scatter(X(:,1),X(:,2),10,'.')  
hold on
```



Fit a two-component Gaussian mixture model.

```
options = statset('Display','final');  
obj = gmdistribution.fit(X,2,'Options',options);
```

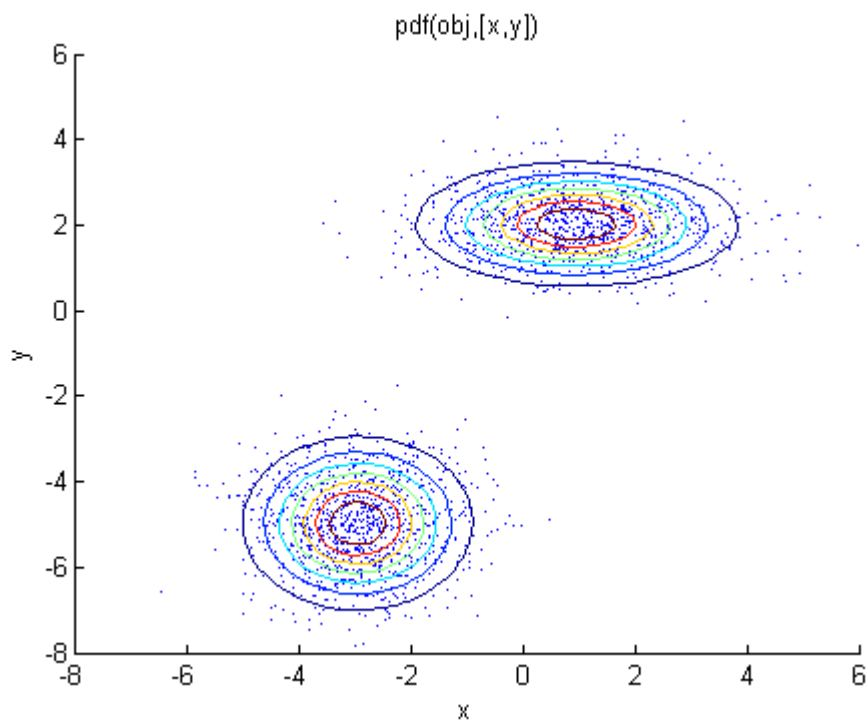
```
18 iterations, log-likelihood = -7058.35
```

Plot the fit.

```
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

# gmdistribution

---



## References

- [1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

**See Also** `gmdistribution.fit`

## Related Examples

- “Normal Distribution” on page B-110

## Concepts

**Purpose** Construct Gaussian mixture distribution

**Syntax** `obj = gmdistribution(mu,sigma,p)`

**Description** `obj = gmdistribution(mu,sigma,p)` constructs an object `obj` of the `gmdistribution` class defining a Gaussian mixture distribution.

`mu` is a  $k$ -by- $d$  matrix specifying the  $d$ -dimensional mean of each of the  $k$  components.

`sigma` specifies the covariance of each component. The size of `sigma` is:

- $d$ -by- $d$ -by- $k$  if there are no restrictions on the form of the covariance. In this case, `sigma(:, :, I)` is the covariance of component  $I$ .
- 1-by- $d$ -by- $k$  if the covariance matrices are restricted to be diagonal, but not restricted to be same across components. In this case, `sigma(:, :, I)` contains the diagonal elements of the covariance of component  $I$ .
- $d$ -by- $d$  matrix if the covariance matrices are restricted to be the same across components, but not restricted to be diagonal. In this case, `sigma` is the pooled estimate of covariance.
- 1-by- $d$  if the covariance matrices are restricted to be diagonal and the same across components. In this case, `sigma` contains the diagonal elements of the pooled estimate of covariance.

`p` is an optional 1-by- $k$  vector specifying the mixing proportions of each component. If `p` does not sum to 1, `gmdistribution` normalizes it. The default is equal proportions.

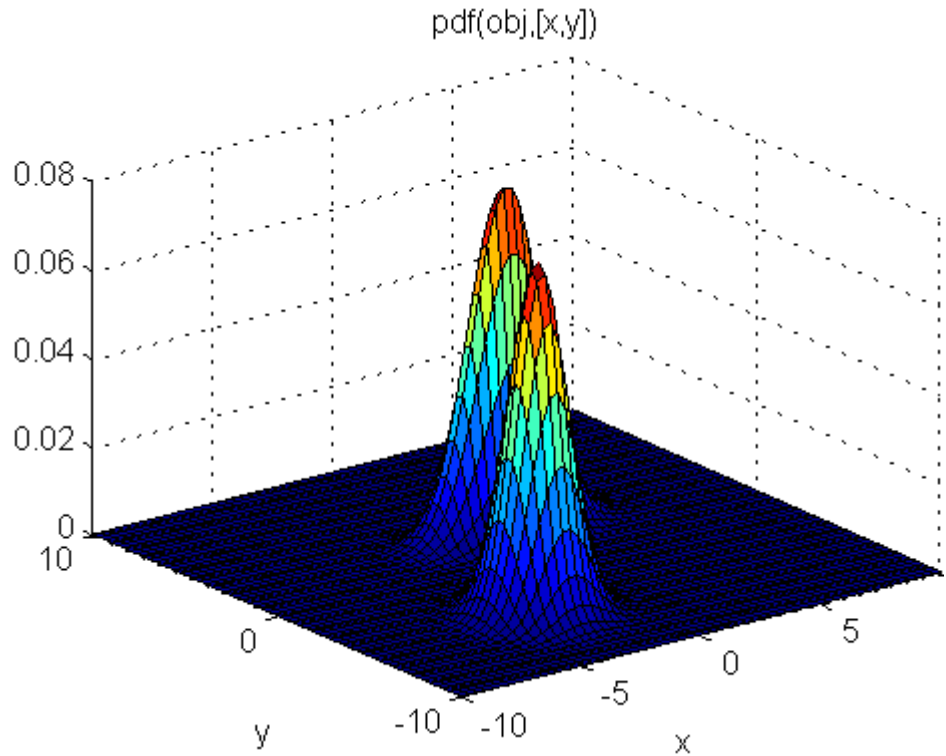
**Examples** Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions:

```
mu = [1 2; -3 -5];  
sigma = cat(3, [2 0; 0 .5], [1 0; 0 1]);  
p = ones(1,2)/2;  
obj = gmdistribution(mu,sigma,p);
```

# gmdistribution

---

```
ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
```



## References

[1] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.

## See Also

[fit](#) | [pdf](#) | [cdf](#) | [random](#) | [cluster](#) | [posterior](#) | [mahal](#)



**Purpose**

Add case names to plot

**Syntax**

```
gname(cases)
gname
h = gname(cases,line_handle)
```

**Description**

`gname(cases)` displays a figure window and waits for you to press a mouse button or a keyboard key. The input argument `cases` is a character array or a cell array of strings, in which each row of the character array or each element of the cell array contains the case name of a point. Moving the mouse over the graph displays a pair of cross-hairs. If you position the cross-hairs near a point with the mouse and click once, the graph displays the label corresponding to that point. Alternatively, you can click and drag the mouse to create a rectangle around several points. When you release the mouse button, the graph displays the labels for all points in the rectangle. Right-click a point to remove its label. When you are done labelling points, press the **Enter** or **Escape** key to stop labeling.

`gname` with no arguments labels each case with its case number.

`cases` typically contains unique case names for each point, and is a cell array of strings or a character matrix with each row representing a name. `cases` can also be any grouping variable, which `gname` converts to labels.

`h = gname(cases,line_handle)` returns a vector of handles to the text objects on the plot. Use the scalar `line_handle` to identify the correct line if there is more than one line object on the plot.

You can use `gname` to label plots created by the `plot`, `scatter`, `gscatter`, `plotmatrix`, and `gplotmatrix` functions.

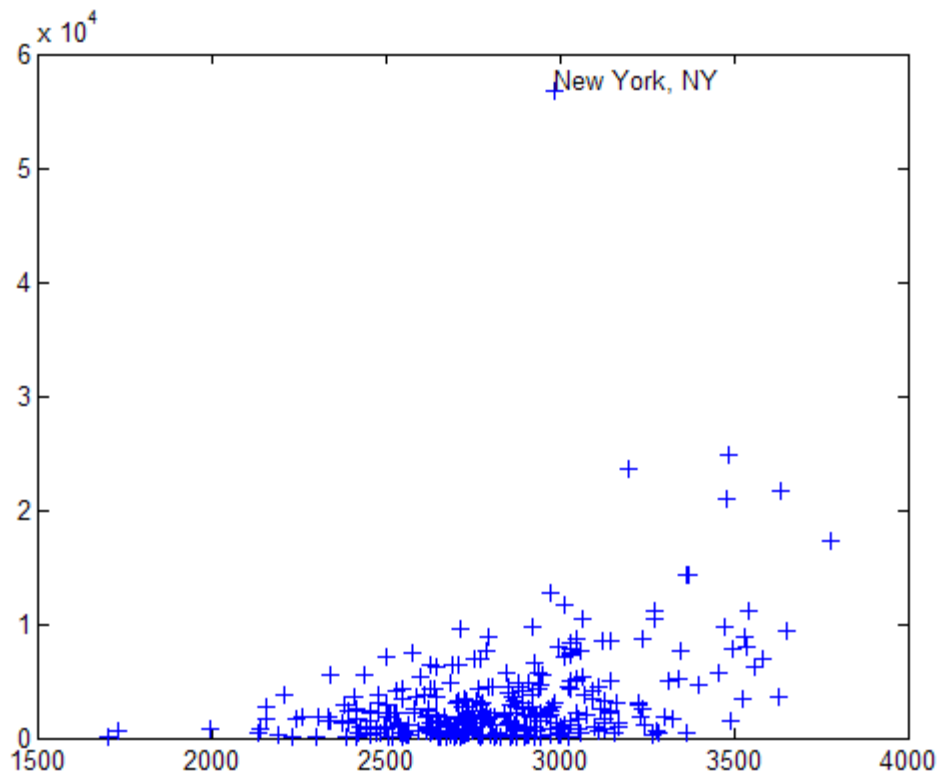
**Examples**

This example uses the city ratings data sets to find out which cities are the best and worst for education and the arts.

```
load cities
education = ratings(:,6);
arts = ratings(:,7);
```

# gname

```
plot(education, arts, '+' )  
gname(names)
```



Click the point at the top of the graph to display its label, “New York.”

## See Also

`gtext` | `gscatter` | `gplotmatrix`

<b>Purpose</b>	Generalized Pareto cumulative distribution function
<b>Syntax</b>	<pre>p = gpcdf(x,x,sigma,theta) p = gpcdf(x,k,sigma,theta,'upper')</pre>
<b>Description</b>	<p><code>p = gpcdf(x,x,sigma,theta)</code> returns the cdf of the generalized Pareto (GP) distribution with the tail index (shape) parameter <code>k</code>, scale parameter <code>sigma</code>, and threshold (location) parameter, <code>theta</code>, evaluated at the values in <code>x</code>. The size of <code>p</code> is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.</p> <p><code>p = gpcdf(x,k,sigma,theta,'upper')</code> returns the complement of the cdf of the generalized Pareto (GP) distribution, using an algorithm that more accurately computes the extreme upper tail probabilities.</p> <p>Default values for <code>k</code>, <code>sigma</code>, and <code>theta</code> are 0, 1, and 0, respectively.</p> <p>When <code>k = 0</code> and <code>theta = 0</code>, the GP is equivalent to the exponential distribution. When <code>k &gt; 0</code> and <code>theta = sigma/k</code>, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when <math>k \geq 1</math>, and the variance is not finite when <math>k \geq 1/2</math>. When <math>k \geq 0</math>, the GP has positive density for</p> <p><code>x &gt; theta</code>, or, when</p> $k < 0, 0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}.$
<b>References</b>	<p>[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. <i>Modelling Extremal Events for Insurance and Finance</i>. New York: Springer, 1997.</p> <p>[2] Kotz, S., and S. Nadarajah. <i>Extreme Value Distributions: Theory and Applications</i>. London: Imperial College Press, 2000.</p>
<b>See Also</b>	<code>cdf</code>   <code>gppdf</code>   <code>gpinv</code>   <code>gpstat</code>   <code>gpfit</code>   <code>gplike</code>   <code>gprnd</code>
<b>How To</b>	<ul style="list-style-type: none"> <li>“Generalized Pareto Distribution” on page B-53</li> </ul>

**Purpose** Generalized Pareto parameter estimates

**Syntax**

```
parmhat = gpfif(X)  
[parmhat,parmci] = gpfif(X)  
[parmhat,parmci] = gpfif(X,alpha)  
[...] = gpfif(X,alpha,options)
```

**Description** `parmhat = gpfif(X)` returns maximum likelihood estimates of the parameters for the two-parameter generalized Pareto (GP) distribution given the data in `X`. `parmhat(1)` is the tail index (shape) parameter, `K` and `parmhat(2)` is the scale parameter, `sigma`. `gpfif` does not fit a threshold (location) parameter.

`[parmhat,parmci] = gpfif(X)` returns 95% confidence intervals for the parameter estimates.

`[parmhat,parmci] = gpfif(X,alpha)` returns  $100(1-\text{alpha})\%$  confidence intervals for the parameter estimates.

`[...] = gpfif(X,alpha,options)` specifies control parameters for the iterative algorithm used to compute ML estimates. This argument can be created by a call to `statset`. See `statset('gpfif')` for parameter names and default values.

Other functions for the generalized Pareto, such as `gpcdf` allow a threshold parameter, `theta`. However, `gpfif` does not estimate `theta`. It is assumed to be known, and subtracted from `X` before calling `gpfif`.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma/K`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when  $K \geq 1$ , and the variance is not finite when  $K \geq 1/2$ . When  $K \geq 0$ , the GP has positive density for

`X > theta`, or, when  $K < 0$ , for

$$0 \leq \frac{x-\theta}{\sigma} \leq -\frac{1}{k}$$

**References**

[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

**See Also**

mle | gplike | gppdf | gpcdf | gpinv | gpstat | gprnd

**How To**

- “Generalized Pareto Distribution” on page B-53

# gpinv

---

**Purpose** Generalized Pareto inverse cumulative distribution function

**Syntax** `X = gpinv(P,K,sigma,theta)`

**Description** `X = gpinv(P,K,sigma,theta)` returns the inverse cdf for a generalized Pareto (GP) distribution with tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter `theta`, evaluated at the values in `P`. The size of `X` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma/K`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when  $K \geq 1$ , and the variance is not finite when  $K \geq 1/2$ . When  $K \geq 0$ , the GP has positive density for

`X > theta`, or, when

$$K < 0, 0 \leq \frac{x-\theta}{\sigma} \leq -\frac{1}{k}.$$

**References** [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

**See Also** `icdf` | `gpcdf` | `gppdf` | `gpstat` | `gpfit` | `gplike` | `gprnd`

**How To** • “Generalized Pareto Distribution” on page B-53

<b>Purpose</b>	Generalized Pareto negative log-likelihood
<b>Syntax</b>	<code>nlogL = gplike(params,data)</code> <code>[nlogL,ACOV] = gplike(params,data)</code>
<b>Description</b>	<p><code>nlogL = gplike(params,data)</code> returns the negative of the log-likelihood <code>nlogL</code> for the two-parameter generalized Pareto (GP) distribution, evaluated at parameters <code>params</code>. <code>params(1)</code> is the tail index (shape) parameter, <code>K</code>, <code>params(2)</code> is the scale parameter, <code>sigma</code>, and <code>params(3)</code> is the threshold (location) parameter, <code>mu</code>.</p> <p><code>[nlogL,ACOV] = gplike(params,data)</code> returns the inverse of Fisher's information matrix, <code>ACOV</code>. If the input parameter values in <code>params</code> are the maximum likelihood estimates, the diagonal elements of <code>ACOV</code> are their asymptotic variances. <code>ACOV</code> is based on the observed Fisher's information, not the expected information.</p> <p>When <code>K = 0</code> and <code>theta = 0</code>, the GP is equivalent to the exponential distribution. When <code>K &gt; 0</code> and <code>theta = sigma/K</code>, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when <math>K \geq 1</math>, and the variance is not finite when <math>K \geq 1/2</math>. When <math>K \geq 0</math>, the GP has positive density for</p> <p><math>X &gt; \text{theta}</math>, or, when</p> $K < 0, 0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}.$
<b>References</b>	<p>[1] Embrechts, P., C. Klüppelberg, and T. Mikosch. <i>Modelling Extremal Events for Insurance and Finance</i>. New York: Springer, 1997.</p> <p>[2] Kotz, S., and S. Nadarajah. <i>Extreme Value Distributions: Theory and Applications</i>. London: Imperial College Press, 2000.</p>
<b>See Also</b>	<code>gpfit</code>   <code>gppdf</code>   <code>gpcdf</code>   <code>gpinv</code>   <code>gpstat</code>   <code>gprnd</code>
<b>How To</b>	<ul style="list-style-type: none"> <li>• “Generalized Pareto Distribution” on page B-53</li> </ul>

**Purpose** Generalized Pareto probability density function

**Syntax** `P = gppdf(X,K,sigma,theta)`

**Description** `P = gppdf(X,K,sigma,theta)` returns the pdf of the generalized Pareto (GP) distribution with the tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter, `theta`, evaluated at the values in `X`. The size of `P` is the common size of the input arguments. A scalar input functions as a constant matrix of the same size as the other inputs.

Default values for `K`, `sigma`, and `theta` are 0, 1, and 0, respectively.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma/K`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when  $K \geq 1$ , and the variance is not finite when  $K \geq 1/2$ . When  $K \geq 0$ , the GP has positive density for

`X > theta`, or, when

$$K < 0, 0 \leq \frac{x-\theta}{\sigma} \leq -\frac{1}{k}.$$

**References** [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

**See Also** `pdf` | `gpcdf` | `gpinv` | `gpstat` | `gpfit` | `gplike` | `gprnd`

**How To** • “Generalized Pareto Distribution” on page B-53



**Purpose**

Matrix of scatter plots by group

**Syntax**

```
gplotmatrix(x,y,group)
gplotmatrix(x,y,group,clr,sym,siz)
gplotmatrix(x,y,group,clr,sym,siz,doleg)
gplotmatrix(x,y,group,clr,sym,siz,doleg,dispopt)
gplotmatrix(x,y,group,clr,sym,siz,doleg,dispopt,xnam,ynam)
[h,ax,bigax] = gplotmatrix(...)
```

**Description**

`gplotmatrix(x,y,group)` creates a matrix of scatter plots. Each individual set of axes in the resulting figure contains a scatter plot of a column of `x` against a column of `y`. All plots are grouped by the grouping variable `group`.

`x` and `y` are matrices with the same number of rows. If `x` has  $p$  columns and `y` has  $q$  columns, the figure contains a  $p$ -by- $q$  matrix of scatter plots. If you omit `y` or specify it as the empty matrix, `[]`, `gplotmatrix` creates a square matrix of scatter plots of columns of `x` against each other.

`group` is a grouping variable that can be a categorical variable, vector, string array, or cell array of strings. `group` must have the same number of rows as `x` and `y`. Points with the same value of `group` are placed in the same group, and appear on the graph with the same marker and color. Alternatively, `group` can be a cell array containing several grouping variables (such as `{g1 g2 g3}`); in that case, observations are in the same group if they have common values of all grouping variables.

`gplotmatrix(x,y,group,clr,sym,siz)` specifies the color, marker type, and size for each group. `clr` is a string array of colors recognized by the `plot` function. The default for `clr` is `'bgrcmyk'`. `sym` is a string array of symbols recognized by the `plot` command, with the default value `'.'`. `siz` is a vector of sizes, with the default determined by the `DefaultLineMarkerSize` property. If you do not specify enough values for all groups, `gplotmatrix` cycles through the specified values as needed.

`gplotmatrix(x,y,group,clr,sym,siz,doleg)` controls whether a legend is displayed on the graph (`doleg` is `'on'`, the default) or not (`doleg` is `'off'`).

`gplotmatrix(x,y,group,clr,sym,siz,doleg,dispopt)` controls what appears along the diagonal of a plot matrix of *y* versus *x*. Allowable values are 'none', to leave the diagonals blank, 'hist', to plot histograms, or 'variable', to write the variable names. `gplotmatrix` displays histograms along the diagonal only when there is only one variable (i.e., `gplotmatrix(x,[],[],[],[],[],[], 'hist')`).

`gplotmatrix(x,y,group,clr,sym,siz,doleg,dispopt,xnam,yname)` specifies the names of the columns in the *x* and *y* arrays. These names are used to label the *x*- and *y*-axes. *xname* and *yname* must be character arrays or cell arrays of strings, with one name for each column of *x* and *y*, respectively.

`[h,ax,bigax] = gplotmatrix(...)` returns three arrays of handles. *h* is an array of handles to the lines on the graphs. The array's third dimension corresponds to groups in *G*. *ax* is a matrix of handles to the axes of the individual plots. If *dispopt* is 'hist', *ax* contains one extra row of handles to invisible axes in which the histograms are plotted. *bigax* is a handle to big (invisible) axes framing the entire plot matrix. *bigax* is fixed to point to the current axes, so a subsequent `title`, `xlabel`, or `ylabel` command will produce labels that are centered with respect to the entire plot matrix.

## Create Matrix of Scatter Plots With Grouped Data

This example shows how to create a matrix of scatter plots using grouped sample data.

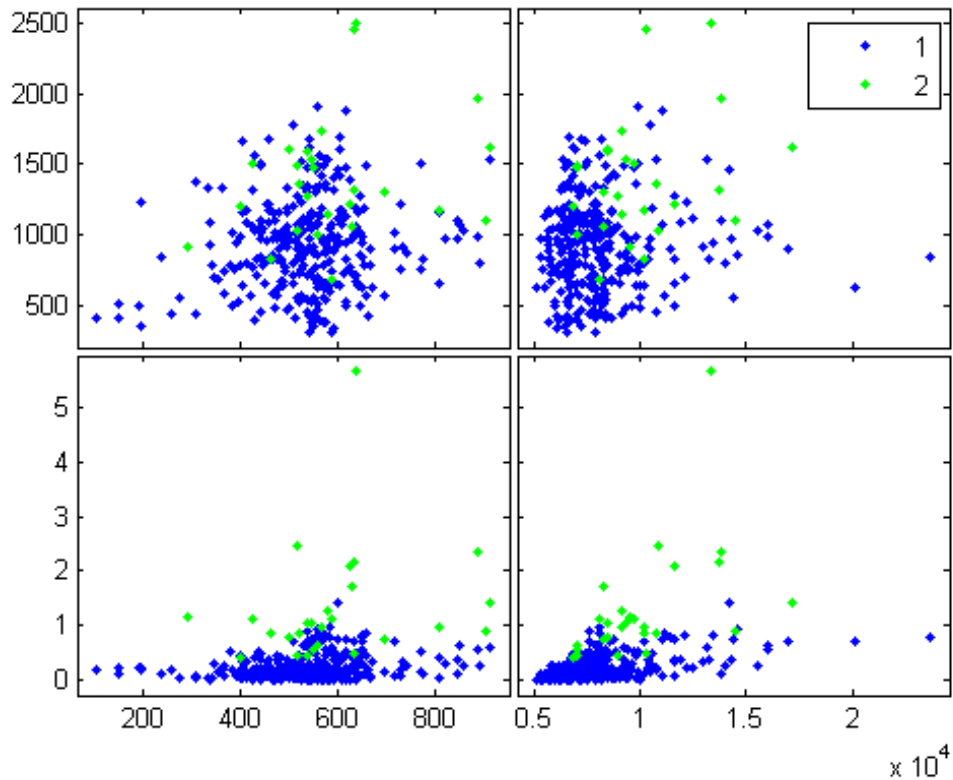
Load the sample data.

```
load discrim;
```

The ratings array contains rating values for 329 U.S. cities in the nine different categories listed in the `categories` array. The group array contains a city size code that is equal to 2 for the 26 largest cities, and 1 otherwise.

Create a matrix of scatter plots to compare the first two categories, climate and housing, with categories 4 (crime) and 7 (arts). Specify group as the grouping variable to visually distinguish the data for large and small cities.

```
figure;  
gplotmatrix(ratings(:,1:2),ratings(:,[4 7]),group);
```

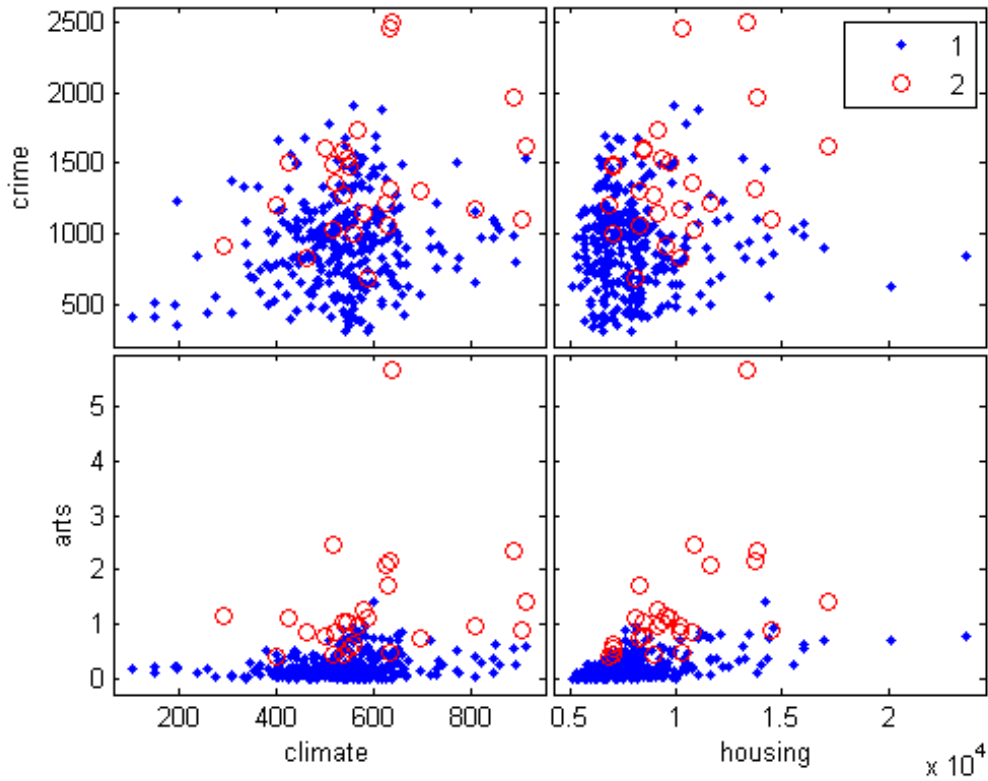


The figure displays a matrix of scatter plots for the specified comparisons, with each city size group represented by a different color.

For better clarity, you can adjust the appearance of the graphs by specifying colors and plotting symbols, and labeling the axes with the rating categories.

# gplotmatrix

```
figure;  
gplotmatrix(ratings(:,1:2),ratings(:,[4 7]),group,...  
    'br','o',[],'on','',categories(1:2,:),...  
    categories([4 7],:));
```



## See Also

[grpstats](#) | [gscatter](#) | [plotmatrix](#)

## How To

- “Grouping Variables” on page 2-51

**Purpose**

Generalized Pareto random numbers

**Syntax**

```
R = gprnd(K,sigma,theta)
R = gprnd(K,sigma,theta,m,n,...)
R = gprnd(K,sigma,theta,[m,n,...])
```

**Description**

`R = gprnd(K,sigma,theta)` returns an array of random numbers chosen from the generalized Pareto (GP) distribution with tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter, `theta`. The size of `R` is the common size of the input arguments if all are arrays. If any parameter is a scalar, the size of `R` is the size of the other parameters.

`R = gprnd(K,sigma,theta,m,n,...)` or `R = gprnd(K,sigma,theta,[m,n,...])` generates an `m`-by-`n`-by-... array. The `K`, `sigma`, `theta` parameters can each be scalars or arrays of the same size as `R`.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma/K`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when  $K \geq 1$ , and the variance is not finite when  $K \geq 1/2$ . When  $K \geq 0$ , the GP has positive density for

$X > \text{theta}$ , or, when

$$0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}$$

**References**

- [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.
- [2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

**See Also**

random | gppdf | gpcdf | gpinv | gpstat | gpfit | gplike

**Purpose** Generalized Pareto mean and variance

**Syntax** `[M,V] = gpstat(K,sigma,theta)`

**Description** `[M,V] = gpstat(K,sigma,theta)` returns the mean of and variance for the generalized Pareto (GP) distribution with the tail index (shape) parameter `K`, scale parameter `sigma`, and threshold (location) parameter, `theta`.

The default value for `theta` is 0.

When `K = 0` and `theta = 0`, the GP is equivalent to the exponential distribution. When `K > 0` and `theta = sigma/K`, the GP is equivalent to the Pareto distribution. The mean of the GP is not finite when  $K \geq 1$ , and the variance is not finite when  $K \geq 1/2$ . When  $K \geq 0$ , the GP has positive density for  $X > \theta$ , or when

$$K < 0, 0 \leq \frac{x - \theta}{\sigma} \leq -\frac{1}{k}.$$

**References** [1] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

[2] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.

**See Also** `gppdf` | `gpcdf` | `gpinv` | `gpfit` | `gplike` | `gprnd`

## Purpose

Train additional trees and add to ensemble

## Syntax

```
B = growTrees(B,ntrees)
B = growTrees(B,ntrees,'param1',val1,'param2',val2,...)
```

## Description

`B = growTrees(B,ntrees)` grows `ntrees` new trees and appends them to those trees already stored in the ensemble `B`.

`B = growTrees(B,ntrees,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

`'nprint'` Specifies that a diagnostic message showing training progress should display after every value training cycles (grown trees). Default is no diagnostic messages.

`'options'` A struct that specifies options that govern computation when growing the ensemble of decision trees. One option requests that the computation of decision trees on multiple bootstrap replicates uses multiple processors, if the Parallel Computing Toolbox is available. Two options specify the random number streams to use in selecting bootstrap replicates. You can create this argument with a call to `statset`. You can retrieve values of the individual fields with a call to `statget`. Applicable `statset` parameters are:

- `'UseParallel'` — If `true` and if a `parpool` of the Parallel Computing Toolbox is open, compute decision trees drawn on separate bootstrap replicates in parallel. If the Parallel Computing Toolbox is not installed, or a `parpool` is not open, computation occurs in serial mode. Default is `false`, or serial computation.

# TreeBagger.growTrees

---

- **UseSubstreams** — Set to `true` to compute in parallel in a reproducible fashion. Default is `false`. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`.
- **Streams** — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `growTrees` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open `Parallel` pool
  - `UseParallel` is `true`
  - `UseSubstreams` is `false`In that case, use a cell array the same size as the `Parallel` pool.

## See Also

`TreeBagger` | `TreeBagger` | `ClassificationTree.fit` | `RegressionTree.fit` | `statset` | `statget`



**Purpose** Create index vector from grouping variable

**Syntax** `[G,GN]=grp2idx(S)`  
`[G,GN,GL] = grp2idx(S)`

**Description** `[G,GN]=grp2idx(S)` creates an index vector `G` from the grouping variable `S`. `S` can be a categorical, numeric, or logical vector; a cell vector of strings; or a character matrix with each row representing a group label. The result `G` is a vector taking integer values from 1 up to the number `K` of distinct groups. `GN` is a cell array of strings representing group labels. `GN(G)` reproduces `S` (aside from any differences in type).

The order of `GN` depends on the grouping variable:

- For numeric and logical grouping variables, the order is the sorted order of `S`.
- For categorical grouping variables, the order is the order of `getlabels(S)`.
- For string grouping variables, the order is the order of first appearance in `S`.

`[G,GN,GL] = grp2idx(S)` returns a column vector `GL` representing the group levels. The set of groups and their order in `GL` and `GN` are the same, except that `GL` has the same type as `S`. If `S` is a character matrix, `GL(G,:)` reproduces `S`, otherwise `GL(G)` reproduces `S`.

`grp2idx` treats NaNs (numeric or logical), empty strings (char or cell array of strings), or `<undefined>` values (categorical) in `S` as missing values and returns NaNs in the corresponding rows of `G`. `GN` and `GL` don't include entries for missing values.

**Examples** Load the data in `hospital.mat` and create a categorical grouping variable:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)', '%d'), {'s'});
AgeGroup = ordinal(hospital.Age, labels, [], edges);
```

```
ages = hospital.Age(1:5)
ages =
    38
    43
    38
    40
    49

group = AgeGroup(1:5)
group =
    30s
    40s
    30s
    40s
    40s

indices = grp2idx(group)
indices =
    4
    5
    4
    5
    5
```

## See Also

[gscatter](#) | [grpstats](#) | [crosstab](#) | [getlabels](#)

## How To

- “Grouping Variables” on page 2-51

**Purpose**

Summary statistics organized by group

**Syntax**

```
statarray = grpstats(ds,groupvar)
statarray = grpstats(ds,groupvar,whichstats)
statarray = grpstats(ds,groupvar,whichstats,Name,Value)

means = grpstats(X,group)
[stats1,...,statsN] = grpstats(X,group,whichstats)
[stats1,...,statsN] = grpstats(X,group,whichstats,'Alpha',
    alpha)

grpstats(X,group,alpha)
```

**Description**

`statarray = grpstats(ds,groupvar)` returns a dataset array with the means for the data groups in the dataset array `ds` determined by the values of the grouping variable or variables specified in `groupvar`.

- If there is a single grouping variable, then there is a row in `statarray` for each value of the grouping variable. `grpstats` sorts the groups by order of appearance (if the grouping variable is a character array), in ascending numeric order (if the grouping variable is numeric), or in order of the levels (if the grouping variable is categorical).
- If `groupvar` is a cell array of strings containing multiple grouping variable names, or a vector of column numbers, then there is a row in `statarray` for each observed unique combination of values of the grouping variables. `grpstats` sorts the groups by the values of the first grouping variable, then the second grouping variable, and so on.
- If any variables in `ds` (other than those specified in `groupvar`) are not numeric or logical arrays, then you must specify the names or column numbers of the numeric and logical variables for which you want to calculate means using the name-value pair argument, `DataVars`.

`statarray = grpstats(ds,groupvar,whichstats)` returns the group values for the summary statistics types specified in `whichstats`.

`statarray = grpstats(ds,groupvar,whichstats,Name,Value)`  
uses additional options specified by one or more `Name,Value` pair arguments.

`means = grpstats(X,group)` returns a column vector or matrix with the means of the groups of the data in the matrix or vector `X` determined by the values of the grouping variable or variables, `group`. The rows of `means` correspond to the grouping variable values.

- If there is a single grouping variable, then there is a row in `means` for each value of the grouping variable. `grpstats` sorts the groups by order of appearance (if the grouping variable is a character array), in ascending numeric order (if the grouping variable is numeric), or in order of the levels (if the grouping variable is categorical).
- If `group` is a cell array of grouping variables, then there is a row in `means` for each observed unique combination of values of the grouping variables. `grpstats` sorts the groups by the values of the first grouping variable, then the second grouping variable, and so on.
- If `X` is a matrix, then `means` is a matrix with the same number of columns as `X`. Each column of `means` has the group means for the corresponding column of `X`.

`[stats1,...,statsN] = grpstats(X,group,whichstats)` returns column vectors or arrays with group values for the summary statistic types specified in `whichstats`.

`[stats1,...,statsN] = grpstats(X,group,whichstats,'Alpha',alpha)` specifies the significance level for confidence and prediction intervals.

`grpstats(X,group,alpha)` plots the means of the groups of data in the vector or matrix `X` determined by the values of the grouping variable, `group`. The grouping variable values are on the horizontal plot axis. Each group mean has  $100 \times (1 - \alpha)\%$  confidence intervals.

- If `X` is a matrix, then `grpstats` plots the means and confidence intervals for each column of `X`.
- If `group` is a cell array of grouping variables, then `grpstats` plots the means and confidence intervals for the groups of data in `X` determined by the unique combinations of values of the grouping variables. For example, if there are two grouping variables, each with two values, there are four possible combinations of grouping variable values. The plot includes only the combinations of values that exist in the input grouping variables (not all possible combinations).

## Input Arguments

### **ds** - Input data

dataset array

Input data, specified as a `dataset` array. The dataset array must include at least one variable that is a grouping variable.

Summary statistics can only be calculated for dataset array variables that have a numeric or logical data type. If any variables in `ds` (other than the grouping variables) are not numeric or logical arrays, then use the name-value pair argument `DataVars` to specify the names or column numbers of the numeric and logical variables for which to calculate summary statistics.

### **groupvar** - Identifiers for the grouping variables

cell array of strings | vector of positive integers | logical vector | []

Identifiers for the grouping variables in the input dataset array, `ds`, specified as one of the following:

String or cell array of strings	Names of the grouping variables
Positive integer or vector of positive integers	Variable numbers of the grouping variables

Vector of logical values with number of elements equal to the number of variables in <code>ds</code>	Logical indicator with value <code>true</code> for grouping variables and <code>false</code> otherwise
<code>[]</code>	No groups (returns summary statistics for all data)

Any variable that is identified by `groupvar` as a grouping variable must have a valid grouping variable data type: categorical array, logical or numeric vector, or cell array of strings.

For example, consider an input dataset array, `ds`, with six variables. The fourth variable is named `Gender`. To be a valid grouping variable, the data type of `Gender` might be a cell array of strings or a nominal array, with the unique values `Male` and `Female`. To specify the variable `Gender` as the grouping variable, you can use any of these syntaxes:

- `statarray = grpstats(ds, 'Gender')`
- `statarray = grpstats(ds, 4)`
- `statarray = grpstats(ds, logical([0 0 0 1 0 0]))`

## Data Types

`double` | `logical` | `cell` | `char`

## whichstats - Types of summary statistics

`string` | `function handle`

Types of summary statistics to compute, specified as a string or function handle, or a cell array of strings and function handles. Use a cell array to specify multiple types of summary statistics.

Possible string values are:

<code>'mean'</code>	Mean
<code>'sem'</code>	Standard error of the mean
<code>'numel'</code>	Count, or number, of non-NaN elements
<code>'gname'</code>	Group name

'std'	Standard deviation
'var'	Variance
'min'	Minimum
'max'	Maximum
'range'	Range
'meanci'	95% confidence interval for the mean
'predci'	95% prediction interval for a new observation

**Example:** `[stat1,stat2] = grpstats(X,group,{'mean','sem'})`

You can specify different significance levels for the 'meanci' and 'predci' options using the name-value pair argument, Alpha.

To specify other types of summary statistics, you can use function handles. You can use the handle to any function that accepts a column or matrix of data, and returns the same size output each time `grpstats` calls it (even if the output for some groups is empty).

If the function accepts a column of data, then the function can return either a scalar value, or an *nvals*-by-1 column vector for descriptive statistics of length *nvals* (for example, confidence intervals have length two). If the function accepts a matrix, it must either return a 1-by-*ncols* row vector, or an *nvals*-by-*ncols* matrix, where *ncols* is the number of columns in the input data matrix.

**Example:** `[stat1,stat2,stat3] = grpstats(X,group,{'mean','std',@skewness})`

### Data Types

char | function\_handle

### alpha - Significance level

scalar value in the range (0,1)

Significance level, specified as a scalar value in the range (0,1).

- When you specify 'meanci' or 'predci' in `whichstats`, you can use `alpha` to specify the significance level for the confidence or prediction

intervals. If you specify `alpha`, then `grpstats` returns  $100 \times (1 - \alpha)\%$  confidence or prediction intervals. If you do not specify `alpha`, then `grpstats` returns 95% intervals (`alpha = 0.05`).

- Use `alpha` with the `grpstats(X,group,alpha)` syntax to plot group means and corresponding  $100 \times (1 - \alpha)\%$  confidence intervals.

### Data Types

double

### X - Input data

vector | matrix

Input data, specified as a vector or a matrix. If `X` is a matrix, then `grpstats` returns summary statistics for each column of `X`.

### Data Types

double | single

### group - Grouping variable

categorical array | logical or numeric vector | cell array of strings | []

Grouping variable, specified as a categorical array, logical or numeric vector, or cell array of strings. Each unique value in a grouping variable defines a group. `grpstats` groups data for summary statistics using the grouping variable values.

There must be a grouping variable value for each row of the input data `X`. Observations (rows) with the same value of the grouping variable are in the same group. Use [] to compute summary statistics for all data, without using groups.

For example, if `Gender` is a cell array of strings with values 'Male' and 'Female', you can use `Gender` as a grouping variable to summarize your data by gender.

You can also use more than one grouping variable to group data for summary statistics. In this case, specify a cell array of grouping variables.



For example, if `Smoker` is a logical vector with values 0 for nonsmokers and 1 for smokers, then specifying the cell array `{Gender,Smoker}` divides observations into four groups: Male Smoker, Male Nonsmoker, Female Smoker, and Female Nonsmoker. `grpstats` returns summary statistics only for the combinations of values that exist in the input grouping variables (not all possible combinations).

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'DataVars',[1,3,4],'Alpha',0.01` specifies that summary statistics be calculated for the 1st, 3rd, and 4th variables in a dataset array, with 99% confidence intervals.

### 'Alpha' - Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level for confidence and prediction intervals, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

When you include `'meanci'` or `'predci'` in `whichstats`, you can use `Alpha` to specify the significance level for confidence or prediction intervals. If you specify the value  $\alpha$ , then `grpstats` returns  $100 \times (1 - \alpha)\%$  confidence or prediction intervals.

If you do not specify a value for `Alpha`, then `grpstats` returns 95% intervals ( $\alpha = 0.05$ ).

**Example:** `'Alpha',0.1`

### Data Types

double

### 'DataVars' - Variable names or columns

cell array of strings | vector of positive integers | logical vector

Variable names or columns indicating which variables in the input dataset array you want to compute summary statistics for, specified as the comma-separated pair consisting of 'DataVars' and a cell array of strings, vector of positive integers, or a logical vector. Use a string to specify a variable name, a positive integer to specify a variable column number, or logical values to indicate which variables to include (true if you want to compute summary statistics, false otherwise).

You must specify DataVars if there are any variables in `ds` (other than the grouping variables specified in `groupvar`) that are not numeric or logical arrays. Summary statistics can only be calculated for dataset array variables that have a numeric or logical data type.

**Example:** 'DataVars',{ 'Height', 'Weight' }

### Data Types

double | cell | char

### 'VarNames' - Variable names for output dataset array

cell array of strings

Variable names for the output dataset array, `statarray`, specified as the comma-separated pair consisting of 'VarNames' and a cell array of strings. By default, `grpstats` constructs output variable names by appending a prefix to the variable names from the input dataset array, `ds`. This prefix corresponds to the summary statistic name.

**Example:** 'VarNames',{ 'MaleMean', 'FemaleMean' }

### Data Types

cell

## Output Arguments

### `statarray` - Group summary statistics

dataset array

Group summary statistics, returned as a dataset array. `statarray` contains summary statistic values for the groups of data in the dataset array `ds` determined by the levels of the grouping variables specified by `groupvar`. There is a row in `statarray` for each observed value

or combination of values in the variables specified by `groupvar`. The dataset array, `statarray`, contains:

- All grouping variables specified by `groupvar`.
- The variable `GroupCount`, containing the number of observations in each group.
- Group summary statistic values for all variables in `ds` (other than those specified by `groupvar`), or for only the variables specified using `DataVars`.

The total number of variables in `statarray` is  $ngroupvars + 1 + ndatavars \times nstats$ , where *ngroupvars* is the number of variables in `groupvar`, *ndatavars* is the number of variables for which summary statistics are computed, and *nstats* is the number of summary statistic types specified in `whichstats`.

`grpstats` assigns default names to the variables in `statarray`, unless you specify variable names using the name-value pair argument `VarNames`.

### **means - Group means**

column vector | array

Group means for the groups of data in the vector or matrix `X` determined by the levels of `group`, returned as an *ngroups*-by-*ncols* array. Here, *ngroups* is the number of unique values in the grouping variable, and *ncols* is the number of columns in `X`. If `X` is a vector, then `means` is a column vector.

### **stats1,...,statsN - Group summary statistics**

column vectors | arrays

Group summary statistics for the groups of data in the vector or matrix `X` determined by the levels of `group`, returned as *ngroups*-by-*ncols* arrays. Here, *ngroups* is the number of unique values in the grouping variable, and *ncols* is the number of columns in `X`. You must specify an output argument for each type of summary statistic specified in `whichstats`.

If a summary statistic type in which `stats` returns a value of length *nvals* (for example, a confidence interval is a descriptive statistic of length two), then the corresponding output argument is an *ngroups-by-ncols-by-nvals* array.

## Examples

### Dataset Array Summary Statistics Organized by Group

Load the sample data.

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

The dataset array, `hospital`, has 100 observations and 7 variables.

Create a dataset array with only the variables `Sex`, `Age`, `Weight`, and `Smoker`.

```
ds = hospital(:, {'Sex', 'Age', 'Weight', 'Smoker'});
```

`Sex` is a nominal array, with levels `Male` and `Female`. The variables `Age` and `Weight` have numeric values, and `Smoker` has logical values.

Compute the mean for the numeric and logical arrays, `Age`, `Weight`, and `Smoker`, grouped by the levels in `Sex`.

```
statarray = grpstats(ds, 'Sex')
```

```
statarray =
```

	Sex	GroupCount	mean_Age	mean_Weight	mean_Smoker
Female	Female	53	37.717	130.47	0.24528
Male	Male	47	38.915	180.53	0.44681

`statarray` is a dataset array with two rows, corresponding to the levels in `Sex`. `GroupCount` is the number of observations in each group. The means of `Age`, `Weight`, and `Smoker`, grouped by `Sex`, are given in `mean_Age`, `mean_Weight`, and `mean_Smoker`.

Compute the mean for `Age` and `Weight`, grouped by the values in `Smoker`.

```
statarray = grpstats(ds, 'Smoker', 'mean', 'DataVars', {'Age', 'Weight'})
```

```
statarray =
```

	Smoker	GroupCount	mean_Age	mean_Weight
0	false	66	37.97	149.91
1	true	34	38.882	161.94

In this case, not all variables in `ds` (excluding the grouping variable, `Smoker`) are numeric or logical arrays; the variable `Sex` is a nominal array. When not all variables in the input dataset array are numeric or logical arrays, you must specify the variables for which you want to calculate summary statistics using `DataVars`.

Compute the minimum and maximum weight, grouped by the combinations of values in `Sex` and `Smoker`.

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

```
statarray =
```

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

There are two unique values in `Smoker` and two levels in `Sex`, for a total of four possible combinations of values: Female Nonsmoker (`Female_0`), Female Smoker (`Female_1`), Male Nonsmoker (`Male_0`), and Male Smoker (`Male_1`).

### Summary Statistics for a Dataset Array Without Grouping

Load the sample data.

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

The dataset array, `hospital`, has 100 observations and 7 variables.

Create a dataset array with only the variables `Age`, `Weight`, and `Smoker`.

```
ds = hospital(:, {'Age', 'Weight', 'Smoker'});
```

The variables `Age` and `Weight` have numeric values, and `Smoker` has logical values.

Compute the mean, minimum, and maximum for the numeric and logical arrays, `Age`, `Weight`, and `Smoker`, with no grouping.

```
statarray = grpstats(ds, [], {'mean', 'min', 'max'})
```

```
statarray =
```

	GroupCount	mean_Age	min_Age	max_Age	mean_Weight	
All	100	38.28	25	50	154	
	min_Weight	max_Weight	mean_Smoker	min_Smoker	max_S	
All	111	202	0.34	false	true	

The observation name `All` indicates that all observations in `ds` were used to compute the summary statistics.

## Group Means for a Matrix Using One or More Grouping Variables

Load the sample data.

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

All variables are measured for 100 cars. `Origin` is the country of origin for each car (France, Germany, Italy, Japan, Sweden, or USA). `Cylinders` has three unique values, 4, 6, and 8, indicating the number of cylinders in each car.

Calculate the mean acceleration, grouped by country of origin.

```
means = grpstats(Acceleration,Origin)
```

```
means =  
    18.0500  
    16.3778  
    15.5000  
    15.8867  
    16.6000  
    14.4377
```

`means` is a 6-by-1 vector of mean accelerations, where each value corresponds to a country of origin.

Calculate the mean acceleration, grouped by both country of origin and number of cylinders.

```
means = grpstats(Acceleration,{Origin,Cylinders})
```

```
means =  
    18.0500  
    16.3375  
    16.7000  
    15.5000  
    15.9143  
    15.5000  
    16.6000  
    17.0818  
    16.5267  
    11.6406
```

There are 18 possible combinations of grouping variable values because `Origin` has 6 unique values and `Cylinders` has 3 unique values. Only 10 of the possible combinations appear in the data, so `means` is a 10-by-1 vector of group means corresponding to the observed combinations of values.

# grpstats

---

Return the group names along with the mean acceleration for each group.

```
[means,grps] = grpstats(Acceleration,{Origin,Cylinders},...
                        {'mean','gname'})
```

```
means =
```

```
17.0818
16.5267
11.6406
18.0500
15.9143
15.5000
16.3375
16.7000
16.6000
15.5000
```

```
grps =
```

```
'USA'      '4'
'USA'      '6'
'USA'      '8'
'France'   '4'
'Japan'    '4'
'Japan'    '6'
'Germany'  '4'
'Germany'  '6'
'Sweden'   '4'
'Italy'    '4'
```

The output `grps` shows the 10 observed combinations of grouping variable values. For example, the mean acceleration of 4-cylinder cars made in France is 18.05.



## Multiple Summary Statistics for a Matrix Organized by Group

Load the sample data.

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

The variable `Acceleration` was measured for 100 cars. The variable `Origin` is the country of origin for each car (France, Germany, Italy, Japan, Sweden, or USA).

Return the minimum, median, and maximum acceleration, grouped by country of origin.

```
[grpMin,grpMed,grpMax,grp] = grpstats(Acceleration,Origin,...  
                                     {'min','median','max','gname'})
```

```
grpMin =
```

```
8.0000  
15.3000  
13.9000  
12.2000  
15.7000  
15.5000
```

```
grpMed =
```

```
14.7000  
17.5000  
15.7000  
15.3000  
16.6000  
15.5000
```

```
grpMax =
```

```
22.2000
21.9000
18.2000
24.6000
17.5000
15.5000
```

```
grp =
    'USA'
    'France'
    'Japan'
    'Germany'
    'Sweden'
    'Italy'
```

The sample car with the lowest acceleration is made in the USA, and the sample car with the highest acceleration is made in Germany.

## Plot Prediction Intervals for a New Observation in Each Group

Load the sample data.

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

The variable `Weight` was measured for 100 cars. The variable `Model_Year` has three unique values, 70, 76, and 82, which correspond to model years 1970, 1976, and 1982.

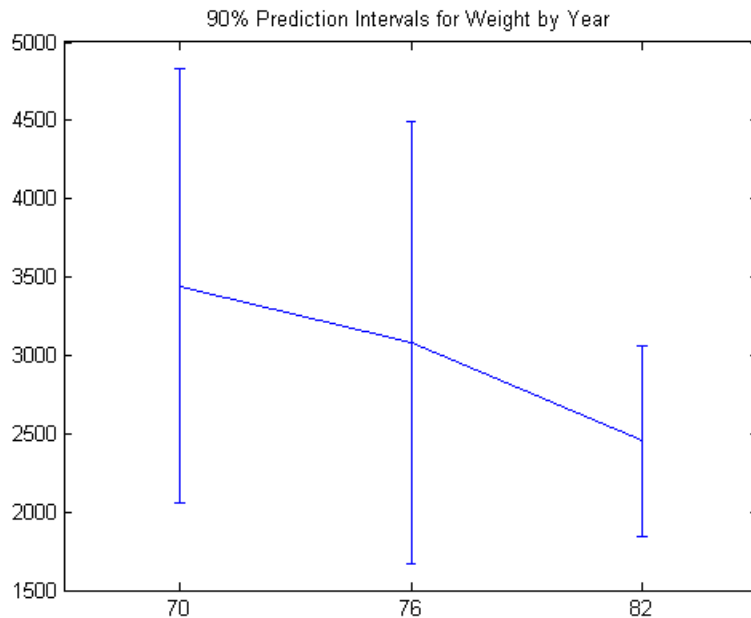
Calculate the mean weight and 90% prediction intervals for each model year.

```
[means,pred,grp] = grpstats(Weight,Model_Year,...
    {'mean','predci','gname'},'Alpha',0.1);
```

Plot error bars showing the mean weight and 90% prediction intervals, grouped by model year. Label the horizontal axis with the group names.

```
ngrps = length(grp); % Number of groups

figure()
errorbar((1:ngrps)', means, pred(:,2) - means)
set(gca, 'xtick', 1:ngrps, 'xticklabel', grp)
title('90% Prediction Intervals for Weight by Year')
```



### Plot Group Means and Confidence Intervals

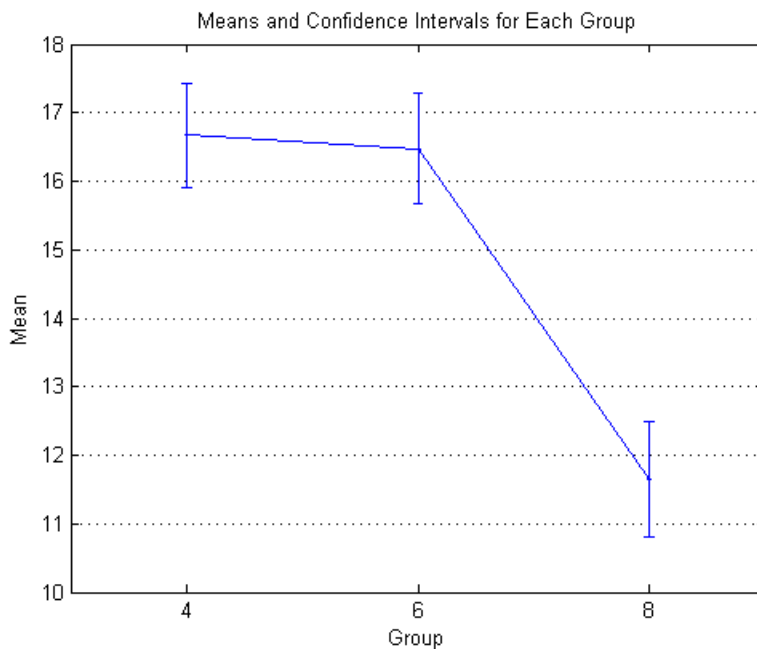
Load the sample data.

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

The variables `Acceleration` and `Weight` are the acceleration and weight values measured for 100 cars. The variable `Cylinders` is the number of cylinders in each car. The variable `Model_Year` has three unique values, 70, 76, and 82, which correspond to model years 1970, 1976, and 1982.

Plot mean acceleration, grouped by `Cylinders`, with 95% confidence intervals.

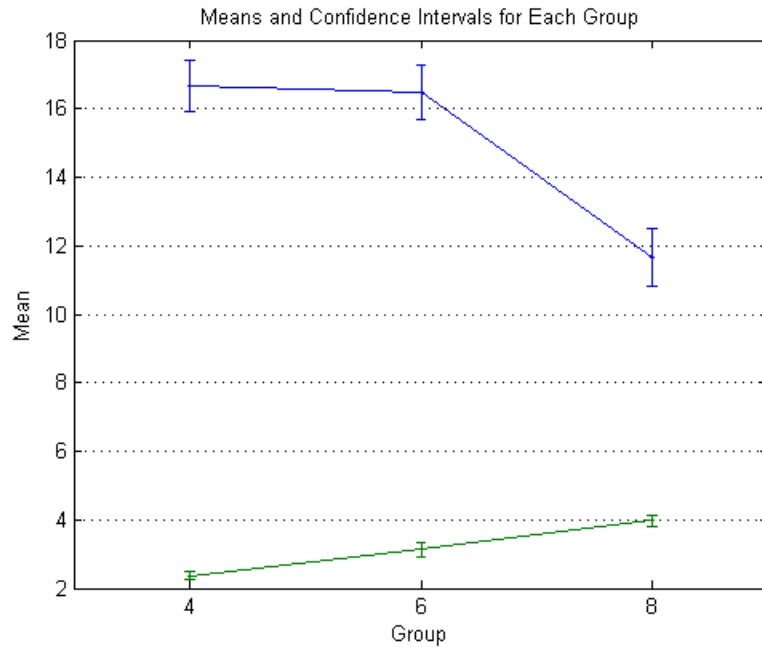
```
grpstats(Acceleration,Cylinders,0.05)
```



The mean acceleration for cars with 8 cylinders is significantly lower than for cars with 4 or 6 cylinders.

Plot mean acceleration and weight, grouped by Cylinders, and 95% confidence intervals. Scale the Weight values by 1000 so the means of Weight and Acceleration are the same order of magnitude.

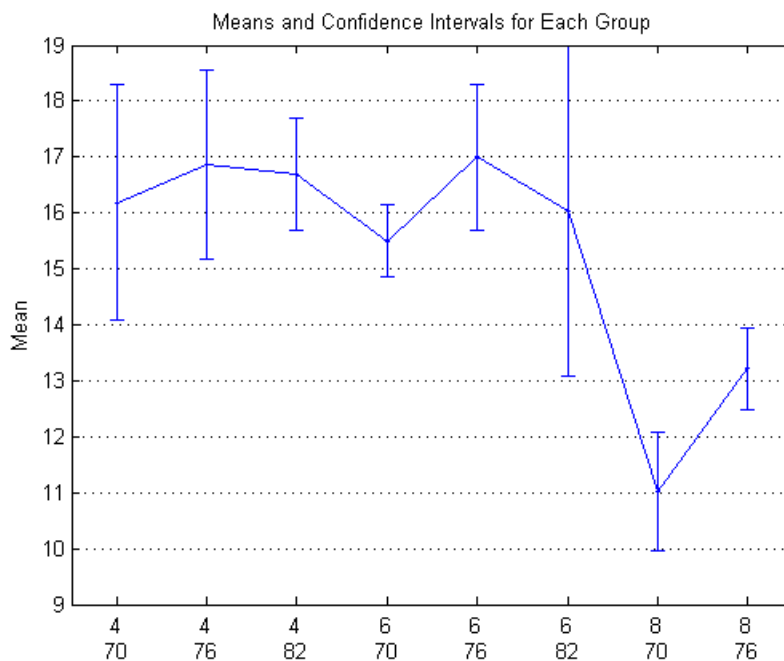
```
grpstats([Acceleration,Weight/1000],Cylinders,0.05)
```



The average weight of cars increases with the number of cylinders, and the average acceleration decreases with the number of cylinders.

Plot mean acceleration, grouped by both Cylinders and Model\_Year. Specify 95% confidence intervals.

```
grpstats(Acceleration,{Cylinders,Model_Year},0.05)
```



There are nine possible combinations of grouping variable values because there are three unique values in `Cylinders` and three unique values in `Model_Year`. The plot does not show 8-cylinder cars with model year 1982 because the data did not include this combination.

The mean acceleration of 8-cylinder cars made in 1976 is significantly larger than the mean acceleration of 8-cylinder cars made in 1970.

## Algorithms

- `grpstats` treats NaNs as missing values, and removes them from the input data before calculating summary statistics.
- `grpstats` ignores empty group names.

**See Also** `dataset` |

**Related Examples**

- “Summary Statistics Grouped by Category” on page 2-37
- “Test Differences Between Category Means” on page 2-28
- “Plot Data Grouped by Category” on page 2-24
- “Calculations on Dataset Arrays” on page 2-111

**Concepts**

- “Dataset Arrays” on page 2-135
- “Grouping Variables” on page 2-51
- “Categorical Arrays” on page 2-41

# gscatter

---

## Purpose

Scatter plot by group

## Syntax

```
gscatter(x,y,group)
gscatter(x,y,group,clr,sym,siz)
gscatter(x,y,group,clr,sym,siz,doleg)
gscatter(x,y,group,clr,sym,siz,doleg,xnam,ynam)
h = gscatter(...)
```

## Description

`gscatter(x,y,group)` creates a scatter plot of `x` and `y`, grouped by `group`. `x` and `y` are vectors of the same size. `group` is a grouping variable in the form of a categorical variable, vector, string array, or cell array of strings. Alternatively, `group` can be a cell array containing several grouping variables (such as `{g1 g2 g3}`), in which case observations are in the same group if they have common values of all grouping variables. Points in the same group and appear on the graph with the same marker and color.

`gscatter(x,y,group,clr,sym,siz)` specifies the color, marker type, and size for each group. `clr` is a string array of colors recognized by the `plot` function. The default for `clr` is `'bgrcmk'`. `sym` is a string array of symbols recognized by the `plot` command, with the default value `'.'`. `siz` is a vector of sizes, with the default determined by the `'DefaultLineMarkerSize'` property. If you do not specify enough values for all groups, `gscatter` cycles through the specified values as needed.

`gscatter(x,y,group,clr,sym,siz,doleg)` controls whether a legend is displayed on the graph (`doleg` is `'on'`, the default) or not (`doleg` is `'off'`).

`gscatter(x,y,group,clr,sym,siz,doleg,xnam,ynam)` specifies the name to use for the `x`-axis and `y`-axis labels. If the `x` and `y` inputs are simple variable names and `xnam` and `ynam` are omitted, `gscatter` labels the axes with the variable names.

`h = gscatter(...)` returns an array of handles to the lines on the graph.



**Examples****Scatter Plot of Climate and Housing Ratings**

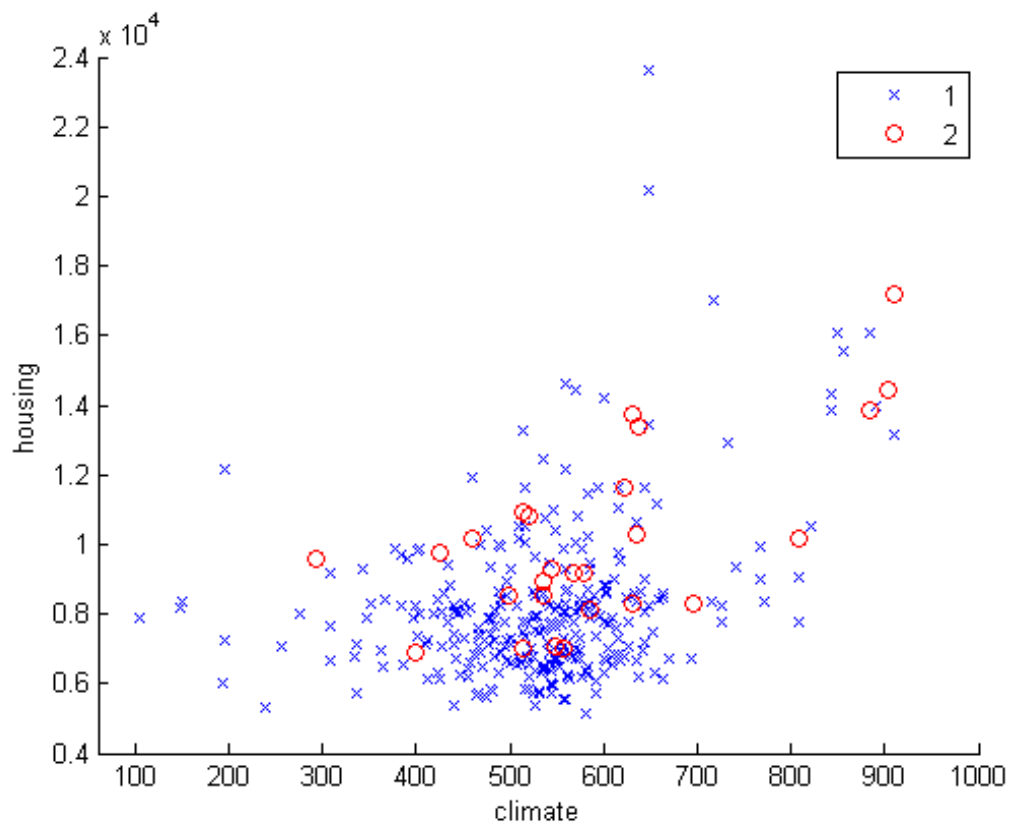
Load the sample data.

```
load discrim
```

discrim data set has ratings of cities according to nine factors such as climate, housing, education, and health in the matrix ratings .

Look at the relationship between the ratings for climate (first column) and housing (second column) grouped by city size in the matrix group. Also choose different colors and plotting symbols for each group.

```
gscatter(ratings(:,1),ratings(:,2),group,'br','xo')  
xlabel('climate');  
ylabel('housing');
```



**See Also** [gplotmatrix](#) | [grpstats](#) | [scatter](#)

**How To** • “Grouping Variables” on page 2-51

**Purpose** Greater than relation for handles

**Syntax** `h1 > h2`

**Description** `h1 > h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `>` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = gt(h1, h2)` stores the result in a logical array of the same dimensions.

**See Also** `grandstream | eq | ge | le | lt | ne`

# haltonset

---

**Superclasses** grandset

**Purpose** Halton quasi-random point sets

**Description** haltonset is a quasi-random point set class that produces points from the Halton sequence.

**Construction** haltonset Construct Halton quasi-random point set

## Methods

### Inherited Methods

Methods in the following table are inherited from grandset.

disp	Display grandset object
end	Last index in indexing expression for point set
length	Length of point set
ndims	Number of dimensions in matrix
net	Generate quasi-random point set
scramble	Scramble quasi-random point set
size	Number of dimensions in matrix
suboref	Subscripted reference for grandset

## Properties

### Inherited Properties

Properties in the following table are inherited from grandset.

Dimensions	Number of dimensions
Leap	Interval between points

ScrambleMethod	Settings that control scrambling
Skip	Number of initial points to omit from sequence
Type	Name of sequence on which point set P is based

## Copy Semantics

Handle. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## References

[1] Kocis, L., and W. J. Whiten, "Computational Investigations of Low-Discrepancy Sequences," *ACM Transactions on Mathematical Software*, Vol. 23, No. 2, pp. 266-294, 1997.

## See Also

`sobolset`

## How To

- “Quasi-Random Point Sets” on page 6-16

# haltonset

---

**Purpose** Construct Halton quasi-random point set

**Syntax**  
`p = haltonset(d)`  
`p = haltonset(d,prop1,val1,prop2,val2,...)`

**Description**  
`p = haltonset(d)` constructs a  $d$ -dimensional point set `p` of the `haltonset` class, with default property settings.  
`p = haltonset(d,prop1,val1,prop2,val2,...)` specifies property name/value pairs used to construct `p`.

The object `p` returned by `haltonset` encapsulates properties of a specified quasi-random sequence. The point set is finite, with a length determined by the `Skip` and `Leap` properties and by limits on the size of point set indices (maximum value of  $2^{53}$ ). Values of the point set are not generated and stored in memory until you access `p` using `net` or parenthesis indexing.

**Examples**  
Generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:

```
p = haltonset(3,'Skip',1e3,'Leap',1e2)
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
        Leap : 100
        ScrambleMethod : none
```

Use `scramble` to apply reverse-radix scrambling:

```
p = scramble(p,'RR2')
p =
    Halton point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
        Leap : 100
        ScrambleMethod : RR2
```

Use `net` to generate the first four points:

```
X0 = net(p,4)
X0 =
    0.0928    0.6950    0.0029
    0.6958    0.2958    0.8269
    0.3013    0.6497    0.4141
    0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

## References

[1] Kocis, L., and W. J. Whiten. “Computational Investigations of Low-Discrepancy Sequences.” *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.

## See Also

`net` | `scramble` | `sobolset`

# harmmean

---

**Purpose** Harmonic mean

**Syntax** `m = harmmean(X)`  
`harmmean(X,dim)`

**Description** `m = harmmean(X)` calculates the harmonic mean of a sample. For vectors, `harmmean(x)` is the harmonic mean of the elements in `x`. For matrices, `harmmean(X)` is a row vector containing the harmonic means of each column. For  $N$ -dimensional arrays, `harmmean` operates along the first nonsingleton dimension of `X`.

`harmmean(X,dim)` takes the harmonic mean along dimension `dim` of `X`.

The harmonic mean is

$$m = \frac{n}{\sum_{i=1}^n \frac{1}{x_i}}$$

**Examples** The arithmetic mean is greater than or equal to the harmonic mean.

```
x = exprnd(1,10,6);
```

```
harmonic = harmmean(x)
harmonic =
    0.3382    0.3200    0.3710    0.0540    0.4936    0.0907
```

```
average = mean(x)
average =
    1.3509    1.1583    0.9741    0.5319    1.0088    0.8122
```

**See Also** `mean` | `median` | `geomean` | `trimmean`



**Purpose**

Bivariate histogram

**Syntax**

```

hist3(X)
hist3(X,nbins)
hist3(X,ctr)
hist3(X,'Edges',edges)
N = hist3(X,...)
[N,C] = hist3(X,...)
hist3(...,param1,val1,param2,val2,...)

```

**Description**

`hist3(X)` bins the elements of the  $m$ -by-2 matrix  $X$  into a 10-by-10 grid of equally spaced containers, and plots a histogram. Each column of  $X$  corresponds to one dimension in the bin grid.

`hist3(X,nbins)` plots a histogram using an `nbins(1)`-by-`nbins(2)` grid of bins. `hist3(X,'Nbins',nbins)` is equivalent to `hist3(X,nbins)`.

`hist3(X,ctr)`, where `ctr` is a two-element cell array of numeric vectors with monotonically non-decreasing values, uses a 2-D grid of bins centered on `ctr{1}` in the first dimension and on `ctr{2}` in the second. `hist3` assigns rows of  $X$  falling outside the range of that grid to the bins along the outer edges of the grid, and ignores rows of  $X$  containing NaNs. `hist3(X,'Ctrs',ctr)` is equivalent to `hist3(X,ctr)`.

`hist3(X,'Edges',edges)`, where `edges` is a two-element cell array of numeric vectors with monotonically non-decreasing values, uses a 2-D grid of bins with edges at `edges{1}` in the first dimension and at `edges{2}` in the second. The  $(i,j)$ th bin includes the value  $X(k,:)$  if

$$\begin{aligned} \text{edges}\{1\}(i) &\leq X(k,1) < \text{edges}\{1\}(i+1) \\ \text{edges}\{2\}(j) &\leq X(k,2) < \text{edges}\{2\}(j+1) \end{aligned}$$

Rows of  $X$  that fall on the upper edges of the grid, `edges{1}(end)` or `edges{2}(end)`, are counted in the  $(I,j)$ th or  $(i,J)$ th bins, where  $I$  and  $J$  are the lengths of `edges{1}` and `edges{2}`. `hist3` does not count rows of  $X$  falling outside the range of the grid. Use `-Inf` and `Inf` in `edges` to include all non-NaN values.

# hist3

---

`N = hist3(X, ...)` returns a matrix containing the number of elements of `X` that fall in each bin of the grid, and does not plot the histogram.

`[N,C] = hist3(X, ...)` returns the positions of the bin centers in a 1-by-2 cell array of numeric vectors, and does not plot the histogram.

`hist3(ax,X, ...)` plots onto an axes with handle `ax` instead of the current axes. See the axes reference page for more information about handles to plots.

`hist3(...,param1,val1,param2,val2,...)` allows you to specify graphics parameter name/value pairs to fine-tune the plot.

## Examples

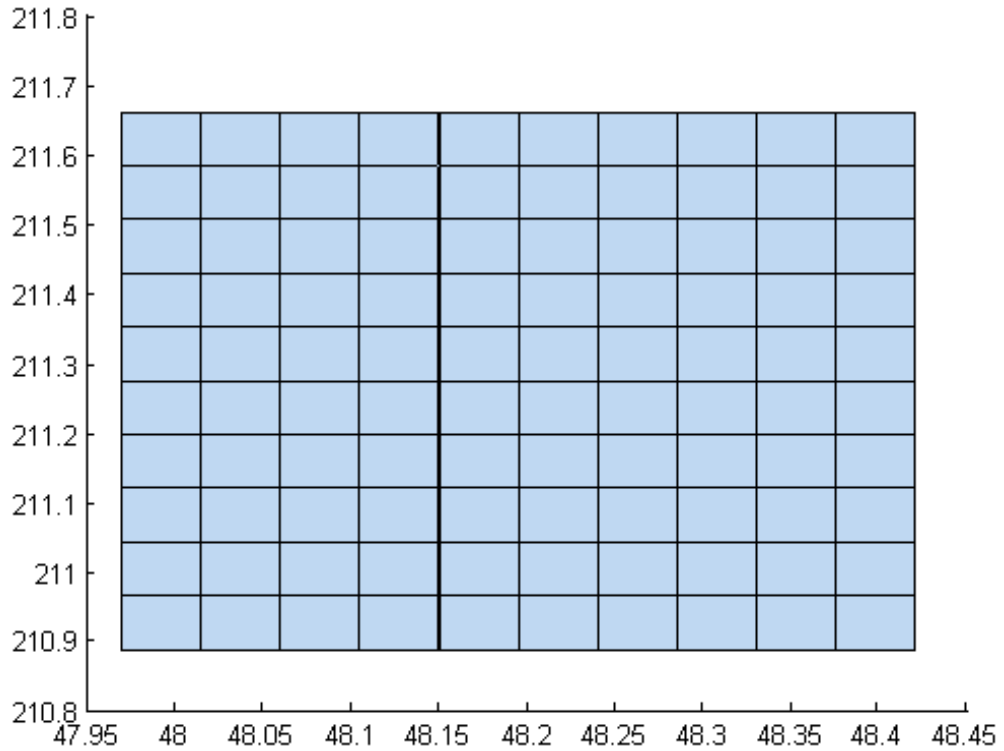
### Plot Density Histogram with Intensity Map

Load the sample data.

```
load seamount
```

Correct grid for negative y-values and draw histogram in 2D.

```
hold on  
dat = [-y,x];  
hist3(dat)
```



Extract histogram data.

```
n = hist3(dat); % default is to 10x10 bins
n1 = n';
n1(size(n,1) + 1, size(n,2) + 1) = 0;
```

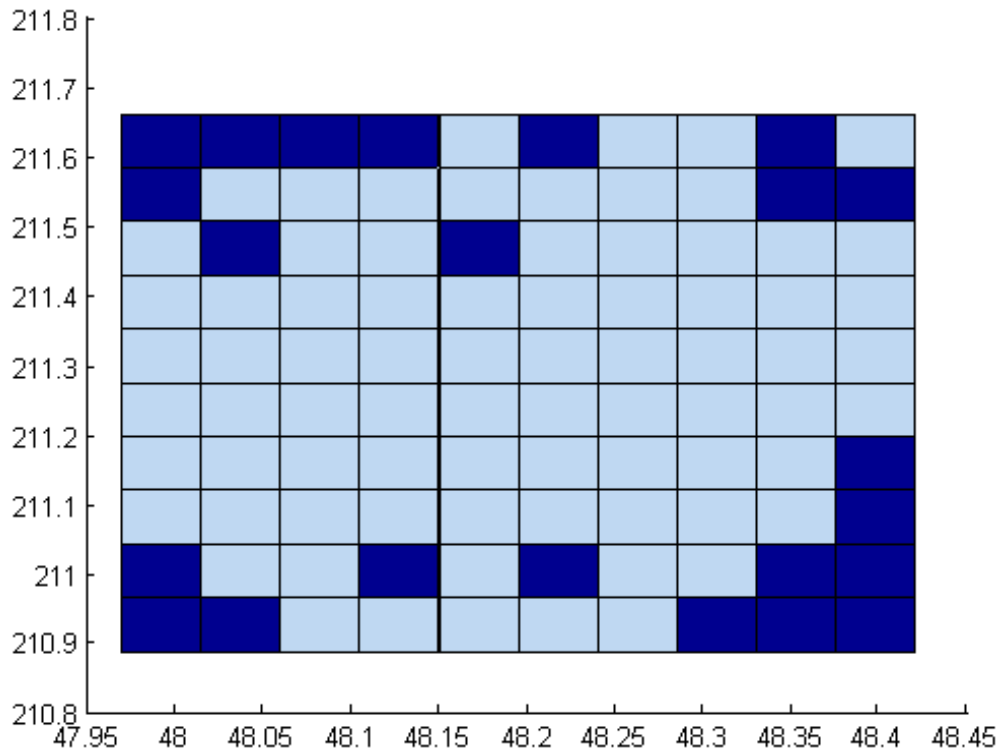
Generate grid for 2-D projected view of intensities.

```
xb = linspace(min(dat(:,1)),max(dat(:,1)),size(n,1)+1);
yb = linspace(min(dat(:,2)),max(dat(:,2)),size(n,1)+1);
```

# hist3

Make a pseudocolor plot.

```
h = pcolor(xb,yb,n1);
```

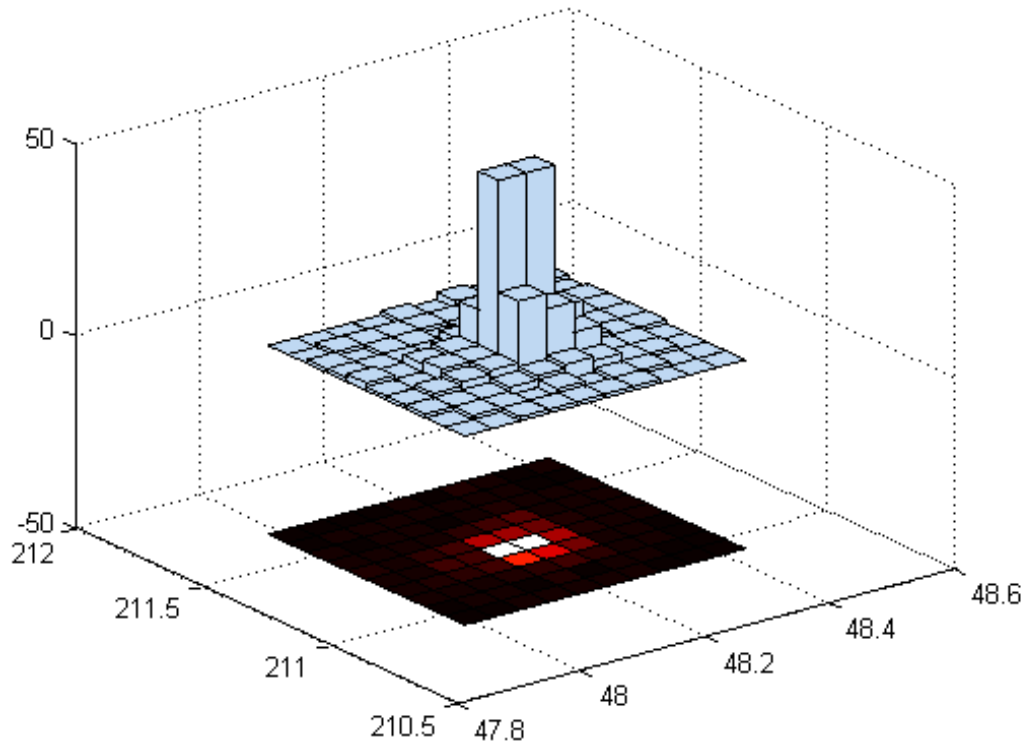


Set the z-level and colormap of the displayed grid, and display the default 3-D perspective view.

```
set(h, 'zdata', ones(size(n1)) * -max(max(n)))  
colormap(hot) % heat map
```

```
title('Seamount:Data Point Density Histogram and Intensity Map');  
grid on  
view(3);
```

Seamount:Data Point Density Histogram and Intensity Map



### Histogram with Semi-Transparent Bars

Load the sample data.

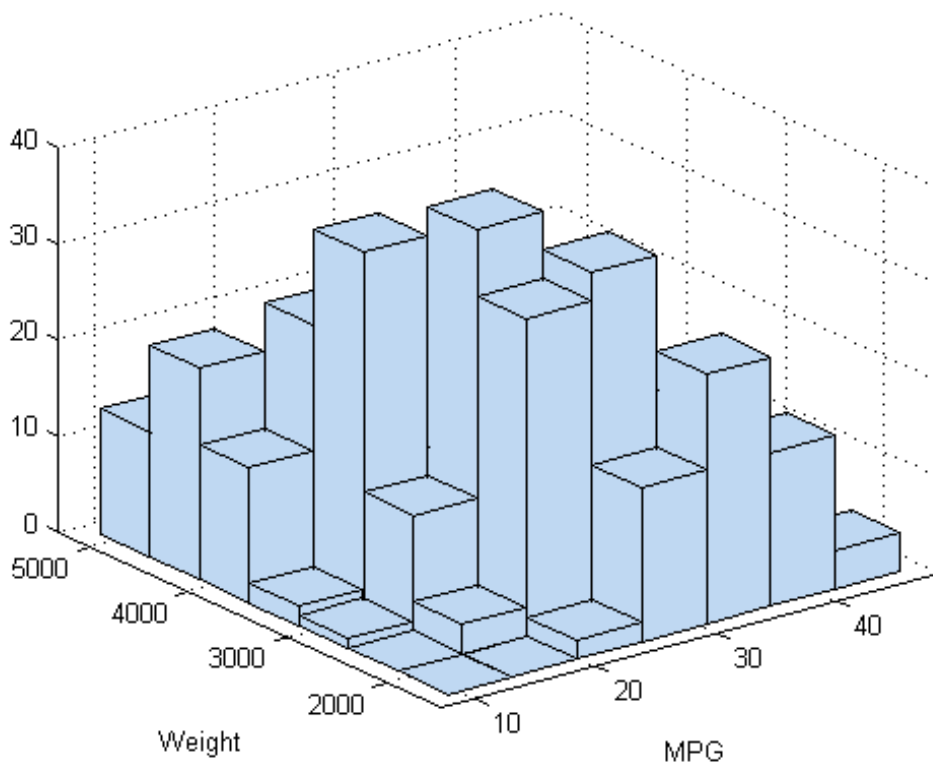
```
load carbig
```

# hist3

---

Use the data to make histogram on a 7-by-7 grid of bins.

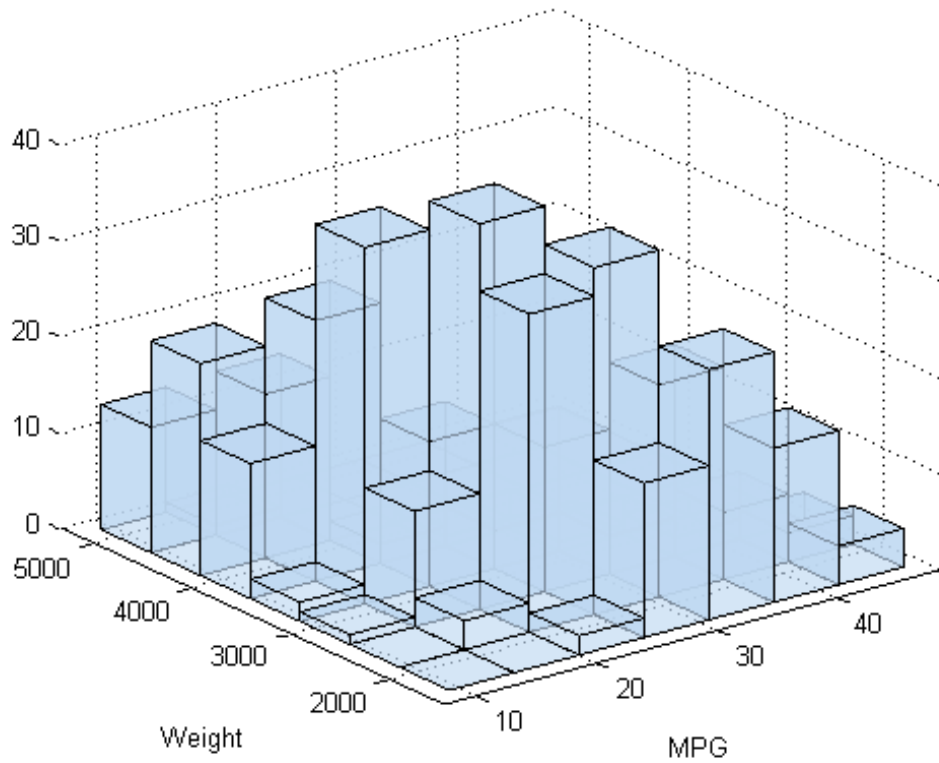
```
X = [MPG,Weight];  
hist3(X,[7 7]);  
xlabel('MPG'); ylabel('Weight');
```



Make a histogram with semi-transparent bars.

```
hist3(X,[7 7],'FaceAlpha',.65);  
xlabel('MPG'); ylabel('Weight');
```

```
set(gcf, 'renderer', 'opengl');
```



Specify bin centers, different in each direction; get back counts, but don't make the plot.

```
cnt = hist3(X, {0:10:50 2000:500:5000});
```

## Histogram Bars Colored According to Height

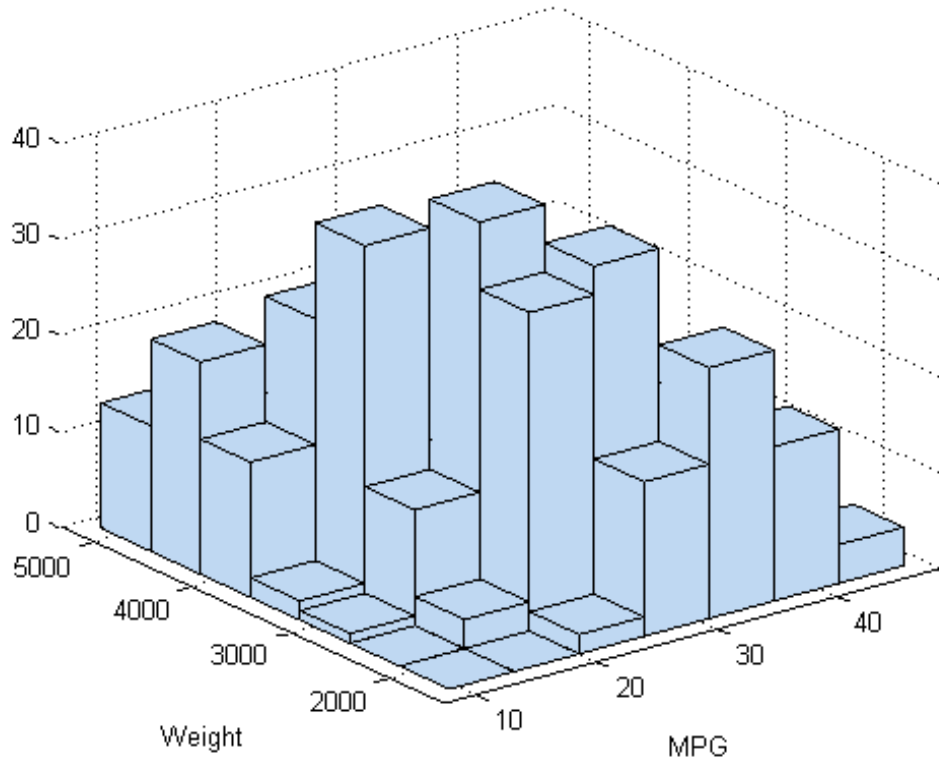
Load the sample data.

```
load carbig
```

Make a histogram on a 7-by-7 grid of bins.

```
X = [MPG,Weight];  
hist3(X,[7 7]);  
xlabel('MPG'); ylabel('Weight');  
set(gcf,'renderer','opengl');
```



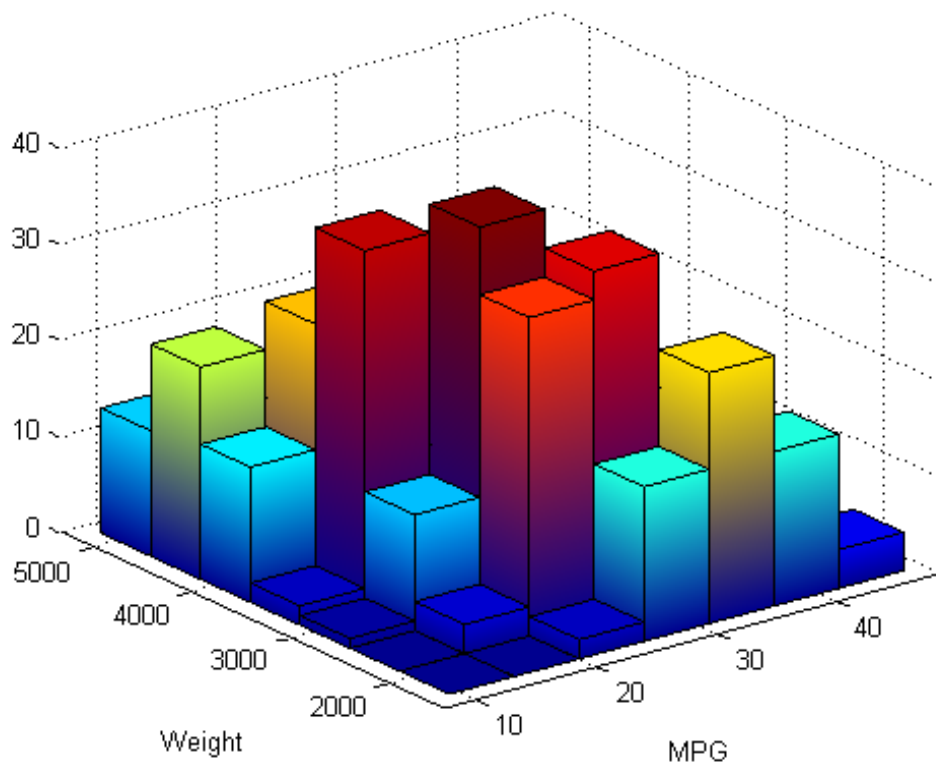


Color the bars based on the frequency of the observations, i.e. according to the height of the bars.

```
set(get(gca,'child'),'FaceColor','interp','CDataMode','auto');
```

# hist3

---



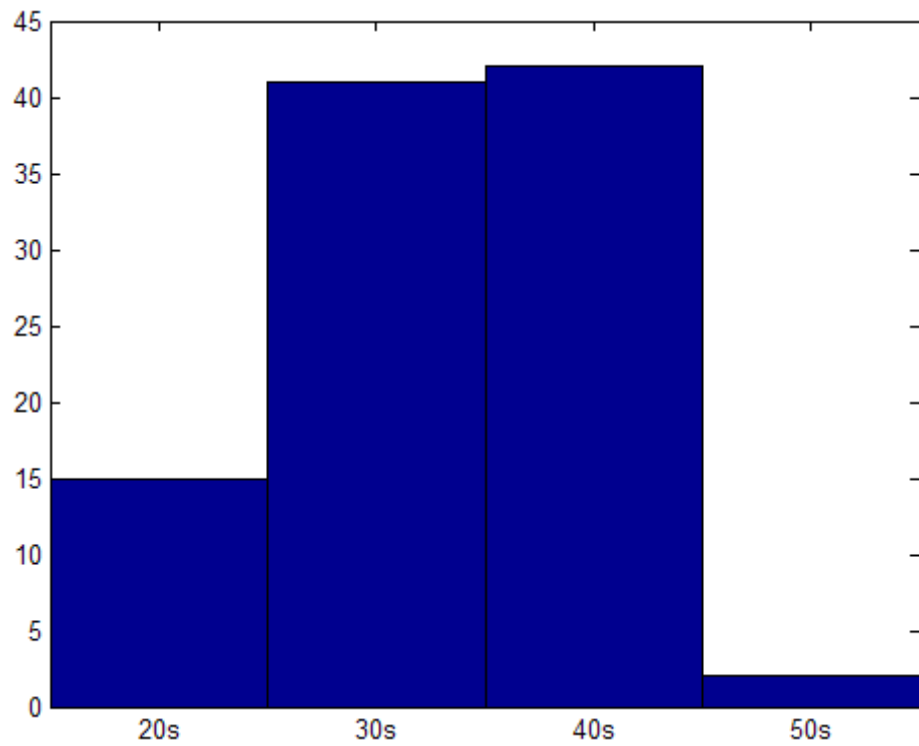
## See Also

[accumarray](#) | [bar](#) | [bar3](#) | [hist](#) | [histc](#)

<b>Purpose</b>	Plot histogram of categorical data
<b>Syntax</b>	<pre>hist(Y) hist(Y,X) hist(ax,...) N = hist(...) [N,X] = hist(...)</pre>
<b>Description</b>	<p><code>hist(Y)</code> plots a histogram bar plot of the counts for each level of the categorical vector <code>Y</code>. If <code>Y</code> is an <math>m</math>-by-<math>n</math> categorical matrix, <code>hist</code> computes counts for each column of <code>Y</code>, and plots a group of <math>n</math> bars for each categorical level.</p> <p><code>hist(Y,X)</code> plots bars only for the levels specified in <code>X</code>. <code>X</code> is a categorical vector or a cell array of level names as strings.</p> <p><code>hist(ax,...)</code> plots into the axes with handle <code>ax</code> instead of <code>gca</code>.</p> <p><code>N = hist(...)</code> returns the counts for each categorical level. If <code>Y</code> is a matrix, <code>hist</code> works down the columns of <code>Y</code> and returns a matrix of counts with one column for each column of <code>Y</code> and one row for each categorical level.</p> <p><code>[N,X] = hist(...)</code> returns the categorical levels corresponding to each count in <code>N</code>, or corresponding to each column of <code>N</code> if <code>Y</code> is a matrix.</p>
<b>Examples</b>	<p>Create a histogram of age groups from the <code>hospital.mat</code> dataset:</p> <pre>load hospital edges = 0:10:100; labels = strcat(num2str((0:10:90)', '%d'), {'s'}); AgeGroup = ordinal(hospital.Age, labels, [], edges); AgeGroup = droplevels(AgeGroup); hist(AgeGroup)</pre>

## categorical.hist

---



### See Also

`hist` | `categorical.levelcounts` | `categorical.getlevels`

---

<b>Purpose</b>	Histogram with a distribution fit
<b>Syntax</b>	<pre>histfit(data) histfit(data,nbins) histfit(data,nbins,dist)  h = histfit( __ )</pre>
<b>Description</b>	<p><code>histfit(data)</code> plots a histogram of values in <code>data</code> using the number of bins equal to the square root of the number of elements in <code>data</code> and fits a normal density function.</p> <p><code>histfit(data,nbins)</code> plots a histogram using <code>nbins</code> bins and fits a normal density function.</p> <p><code>histfit(data,nbins,dist)</code> plots a histogram with <code>nbins</code> bins and fits a density function from the distribution specified by <code>dist</code>.</p> <p><code>h = histfit( __ )</code> returns a vector of handles <code>h</code>, where <code>h(1)</code> is the handle to the histogram and <code>h(2)</code> is the handle to the density curve. It can include any of the input arguments in previous syntaxes.</p>
<b>Input Arguments</b>	<p><b>data - Input data</b> vector Input data, specified as a vector.</p> <p><b>Example:</b> <code>data = [1.5 2.5 4.6 1.2 3.4]</code></p> <p><b>Example:</b> <code>data = [1.5 2.5 4.6 1.2 3.4]'</code></p> <p><b>Data Types</b> double   single</p> <p><b>nbins - Number of bins</b> positive integer   []</p>

# histfit

---

Number of bins for the histogram, specified as a positive integer. Default value is the square root of the number of elements in `data`, rounded up. Use `[]` for the default number of bins when fitting a distribution.

**Example:** `y = histfit(x,8)`

**Example:** `y = histfit(x,10,'gamma')`

**Example:** `y = histfit(x,[],'weibull')`

## Data Types

`double` | `single`

## dist - Distribution to fit

`'normal'` (default) | string

Distribution to fit to the histogram, specified as a string. The following table shows the supported distributions.

dist	Description
<code>'beta'</code>	Beta
<code>'birnbaumsaunders'</code>	Birnbaum-Saunders
<code>'burr'</code>	Burr Type XII
<code>'exponential'</code>	Exponential
<code>'extreme value'</code> or <code>'ev'</code>	Extreme value
<code>'gamma'</code>	Gamma
<code>'generalized extreme value'</code> or <code>'gev'</code>	Generalized extreme value
<code>'generalized pareto'</code> or <code>'gp'</code>	Generalized Pareto (threshold 0)
<code>'inversegaussian'</code>	Inverse Gaussian
<code>'logistic'</code>	Logistic
<code>'loglogistic'</code>	Loglogistic
<code>'lognormal'</code>	Lognormal

<b>dist</b>	<b>Description</b>
'nakagami'	Nakagami
'negative binomial' or 'nbin'	Negative binomial
'normal'	Normal
'poisson'	Poisson
'rayleigh'	Rayleigh
'rician'	Rician
'tlocationscale'	t location-scale
'weibull' or 'wbl'	Weibull
'kernel'	Nonparametric kernel-smoothing distribution. The density is evaluated at 100 equally spaced points that cover the range of the data in data. It works best with continuously distributed samples.

**Data Types**

char

**Output Arguments****h - Handles for the plot**

plot handle

Handles for the plot, returned as a vector, where  $h(1)$  is the handle to the histogram, and  $h(2)$  is the handle to the density curve.

**Examples****Histogram with a Normal Distribution Fit**

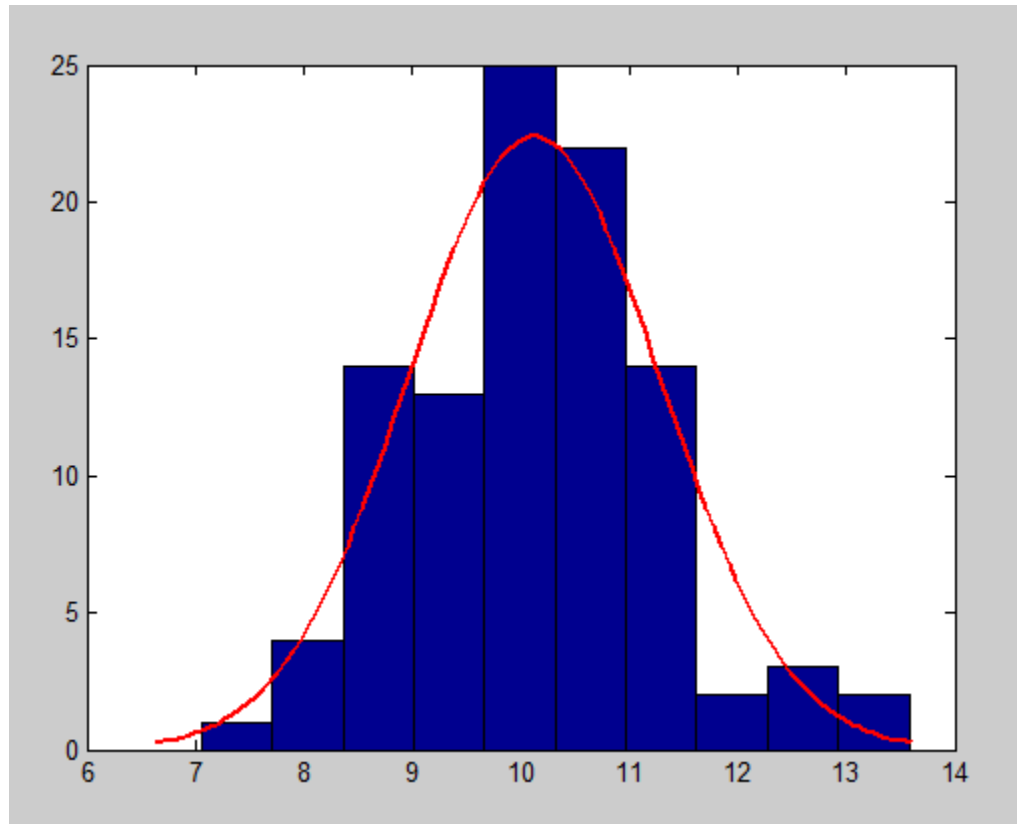
Generate a sample of size 100 from a normal distribution with mean 10 and variance 1.

```
rng('default'); % for reproducibility
r = normrnd(10,1,100,1);
```

# histfit

Construct a histogram with a normal distribution fit.

```
histfit(r)
```



## Histogram for a Given Number of Bins

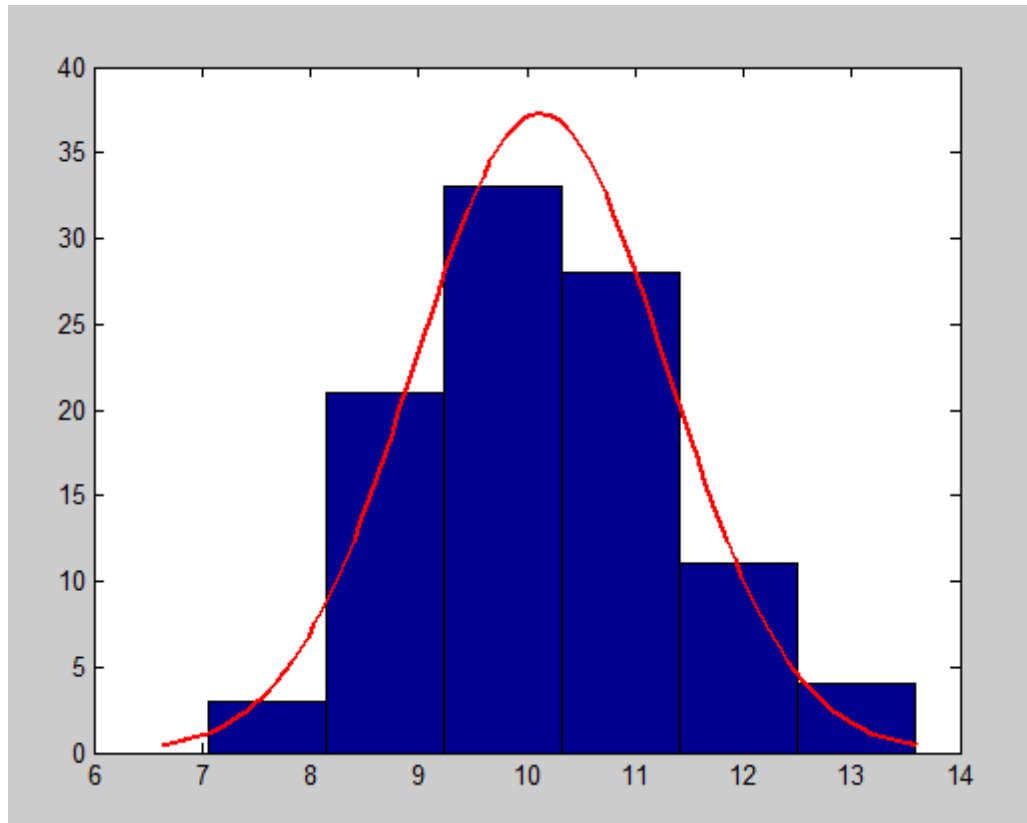
Generate a sample of size 100 from a normal distribution with mean 10 and variance 1.

```
rng('default'); % for reproducibility  
r = normrnd(10,1,100,1);
```



Construct a histogram using six bins with a normal distribution fit.

```
histfit(r,6)
```



### Histogram with a Specified Distribution Fit

Generate a sample of size 100 from a beta distribution with parameters (3,10).

```
rng('default') % for reproducibility
```

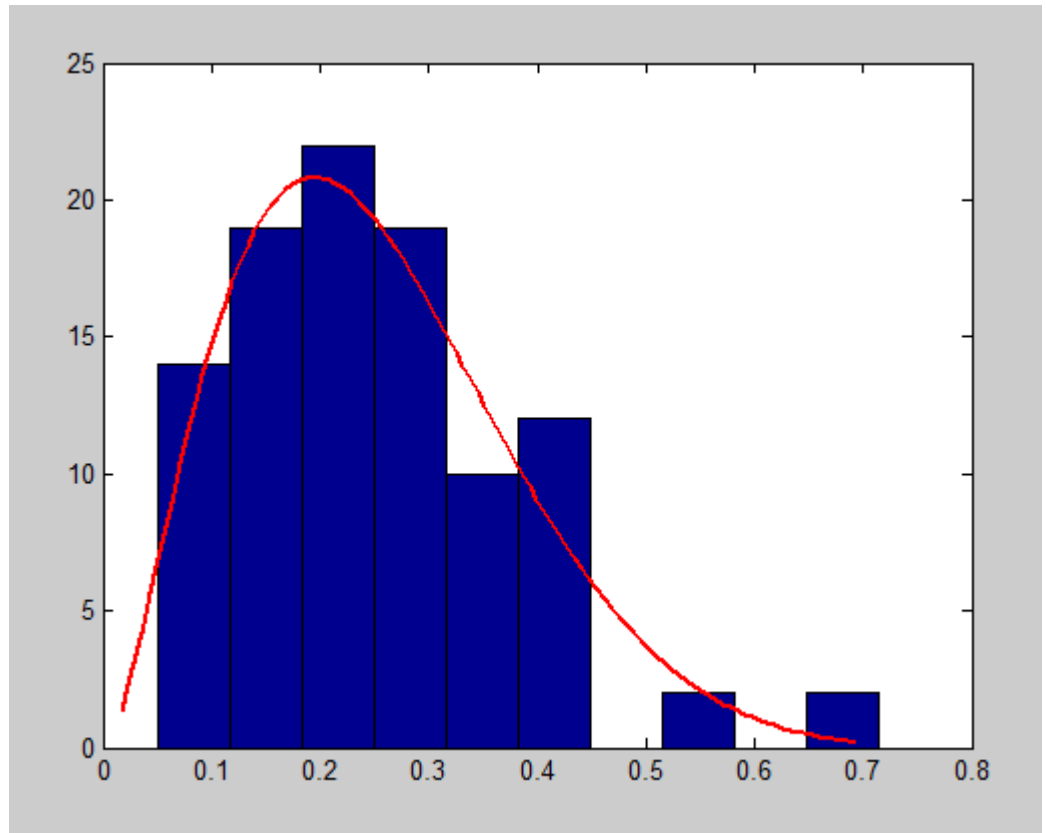
# histfit

---

```
b = betarnd(3,10,100,1);
```

Construct a histogram using 10 bins with a beta distribution fit.

```
histfit(b,10,'beta')
```



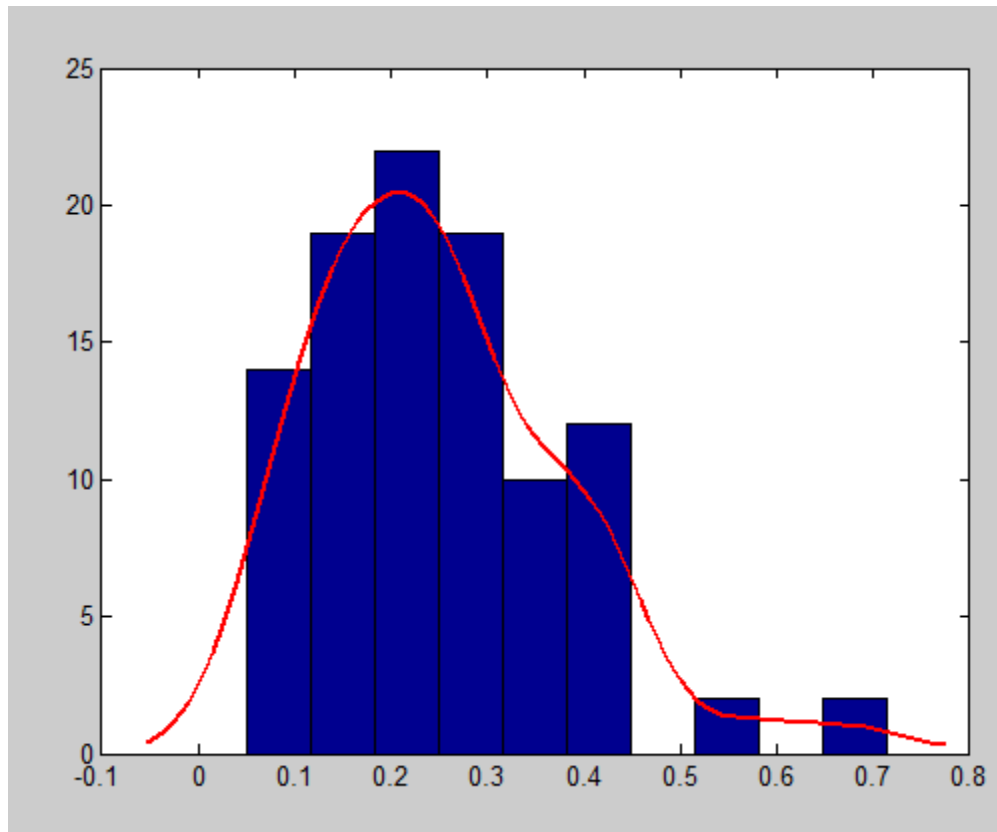
## Histogram with a Kernel Smoothing Function Fit

Generate a sample of size 100 from a beta distribution with parameters (3,10).

```
rng('default') % for reproducibility  
b = betarnd(3,10,[100,1]);
```

Construct a histogram using 10 bins with a smoothing function fit.

```
histfit(b,10,'kernel')
```



### Handle for a Histogram with a Distribution Fit

Generate a sample of size 100 from a normal distribution with mean 10 and variance 1.

# histfit

---

```
rng('default'); % for reproducibility  
r = normrnd(10,1,100,1);
```

Construct a histogram with a normal distribution fit.

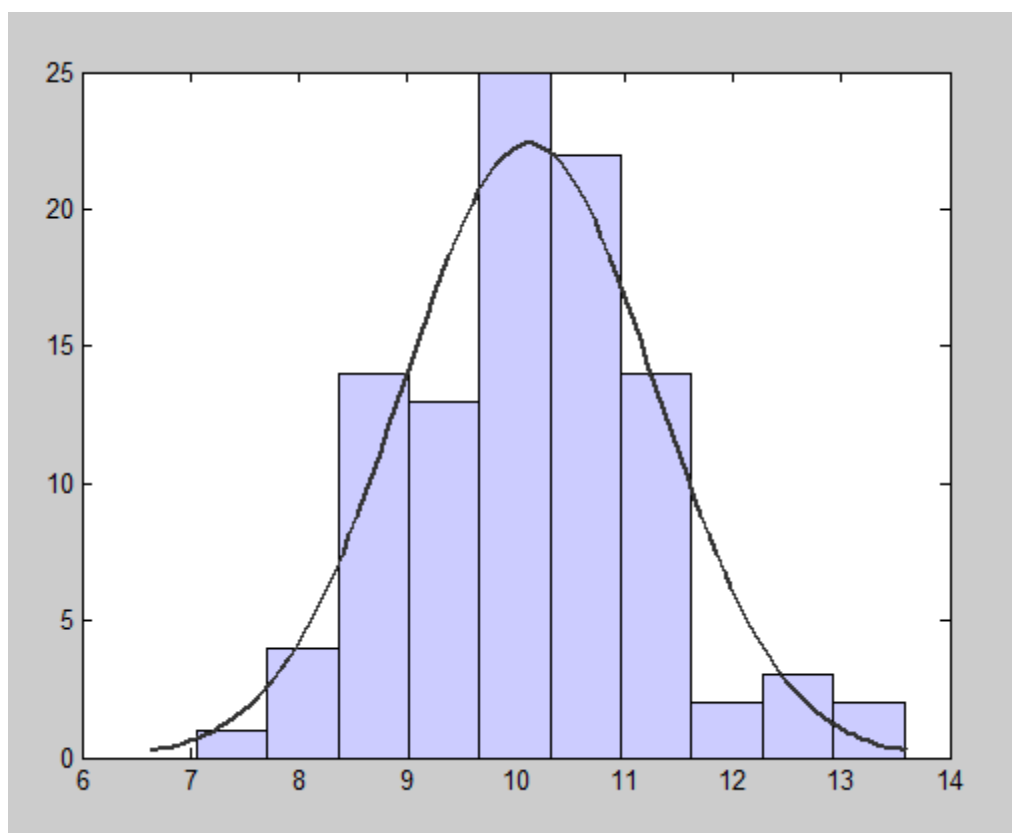
```
h = histfit(r,10,'normal')
```

Change the bar colors of the histogram.

```
set(h(1),'FaceColor',[.8 .8 1])
```

Change the color of the density curve.

```
set(h(2),'Color',[.2 .2 .2])
```



**See Also** `hist` | `normfit` | `dfittool`

# hmmdecode

---

**Purpose** Hidden Markov model posterior state probabilities

**Syntax**

```
PSTATES = hmmdecode(seq,TRANS,EMIS)
[PSTATES,logpseq] = hmmdecode(...)
[PSTATES,logpseq,FORWARD,BACKWARD,S] = hmmdecode(...)
hmmdecode(...,'Symbols',SYMBOLS)
```

**Description** PSTATES = hmmdecode(seq,TRANS,EMIS) calculates the posterior state probabilities, PSTATES, of the sequence seq, from a hidden Markov model. The posterior state probabilities are the conditional probabilities of being at state  $k$  at step  $i$ , given the observed sequence of symbols, sym. You specify the model by a transition probability matrix, TRANS, and an emissions probability matrix, EMIS. TRANS( $i, j$ ) is the probability of transition from state  $i$  to state  $j$ . EMIS( $k, seq$ ) is the probability that symbol seq is emitted from state  $k$ .

PSTATES is an array with the same length as seq and one row for each state in the model. The ( $i, j$ )th element of PSTATES gives the probability that the model is in state  $i$  at the  $j$ th step, given the sequence seq.

---

**Note** The function hmmdecode begins with the model in state 1 at step 0, prior to the first emission. hmmdecode computes the probabilities in PSTATES based on the fact that the model begins in state 1.

---

[PSTATES,logpseq] = hmmdecode(...) returns logpseq, the logarithm of the probability of sequence seq, given transition matrix TRANS and emission matrix EMIS.

[PSTATES,logpseq,FORWARD,BACKWARD,S] = hmmdecode(...) returns the forward and backward probabilities of the sequence scaled by S.

hmmdecode(...,'Symbols',SYMBOLS) specifies the symbols that are emitted. SYMBOLS can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through N, where N is the number of possible emissions.

## References

[1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

## Examples

```
trans = [0.95,0.05;  
         0.10,0.90];  
emis = [1/6 1/6 1/6 1/6 1/6 1/6;  
        1/10 1/10 1/10 1/10 1/10 1/2];  
  
[seq,states] = hmmgenerate(100,trans,emis);  
pStates = hmmdecode(seq,trans,emis);  
[seq,states] = hmmgenerate(100,trans,emis,...  
    'Symbols',{'one','two','three','four','five','six'})  
pStates = hmmdecode(seq,trans,emis,...  
    'Symbols',{'one','two','three','four','five','six'});
```

## See Also

[hmmgenerate](#) | [hmmestimate](#) | [hmmviterbi](#) | [hmmtrain](#)

# hmmestimate

---

**Purpose** Hidden Markov model parameter estimates from emissions and states

**Syntax**

```
[TRANS,EMIS] = hmmestimate(seq,states)
hmmestimate(...,'Symbols',SYMBOLS)
hmmestimate(...,'Statenames',STATENAMES)
hmmestimate(...,'Pseudoemissions',PSEUDOIE)
hmmestimate(...,'Pseudotransitions',PSEUDOTR)
```

**Description**

[TRANS,EMIS] = hmmestimate(seq,states) calculates the maximum likelihood estimate of the transition, TRANS, and emission, EMIS, probabilities of a hidden Markov model for sequence, seq, with known states, states.

hmmestimate(...,'Symbols',SYMBOLS) specifies the symbols that are emitted. SYMBOLS can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through N, where N is the number of possible emissions.

hmmestimate(...,'Statenames',STATENAMES) specifies the names of the states. STATENAMES can be a numeric array or a cell array of the names of the states. The default state names are 1 through M, where M is the number of states.

hmmestimate(...,'Pseudoemissions',PSEUDOIE) specifies pseudocount emission values in the matrix PSEUDO. Use this argument to avoid zero probability estimates for emissions with very low probability that might not be represented in the sample sequence. PSEUDOIE should be a matrix of size  $m$ -by- $n$ , where  $m$  is the number of states in the hidden Markov model and  $n$  is the number of possible emissions. If the  $i \rightarrow k$  emission does not occur in seq, you can set PSEUDOIE(i,k) to be a positive number representing an estimate of the expected number of such emissions in the sequence seq.

hmmestimate(...,'Pseudotransitions',PSEUDOTR) specifies pseudocount transition values. You can use this argument to avoid zero probability estimates for transitions with very low probability that might not be represented in the sample sequence. PSEUDOTR should be a matrix of size  $m$ -by- $m$ , where  $m$  is the number of states in the hidden



Markov model. If the  $i \rightarrow j$  transition does not occur in `states`, you can set `PSEUDOTR(i, j)` to be a positive number representing an estimate of the expected number of such transitions in the sequence `states`.

### Pseudotransitions and Pseudoemissions

If the probability of a specific transition or emission is very low, the transition might never occur in the sequence `states`, or the emission might never occur in the sequence `seq`. In either case, the algorithm returns a probability of 0 for the given transition or emission in `TRANS` or `EMIS`. You can compensate for the absence of transition with the 'Pseudotransitions' and 'Pseudoemissions' arguments. The simplest way to do this is to set the corresponding entry of `PSEUDO` or `PSEUDOTR` to 1. For example, if the transition  $i \rightarrow j$  does not occur in `states`, set `PSEUOTR(i, j) = 1`. This forces `TRANS(i, j)` to be positive.

If you have an estimate for the expected number of transitions  $i \rightarrow j$  in a sequence of the same length as `states`, and the actual number of transitions  $i \rightarrow j$  that occur in `seq` is substantially less than what you expect, you can set `PSEUOTR(i, j)` to the expected number. This increases the value of `TRANS(i, j)`. For transitions that do occur in `states` with the frequency you expect, set the corresponding entry of `PSEUDOTR` to 0, which does not increase the corresponding entry of `TRANS`.

If you do not know the sequence of states, use `hmmtrain` to estimate the model parameters.

## References

[1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

## Examples

```
trans = [0.95,0.05; 0.10,0.90];
emis = [1/6 1/6 1/6 1/6 1/6 1/6;
        1/10 1/10 1/10 1/10 1/10 1/2];

[seq,states] = hmmgenerate(1000,trans,emis);
[estimateTR,estimateE] = hmmestimate(seq,states);
```

# hmmestimate

---

## **See Also**

`hmmgenerate` | `hmmdecode` | `hmmviterbi` | `hmmtrain`

**Purpose** Hidden Markov model states and emissions

**Syntax** `[seq,states] = hmmgenerate(len,TRANS,EMIS)`  
`hmmgenerate(...,'Symbols',SYMBOLS)`  
`hmmgenerate(...,'Statenames',STATENAMES)`

**Description** `[seq,states] = hmmgenerate(len,TRANS,EMIS)` takes a known Markov model, specified by transition probability matrix TRANS and emission probability matrix EMIS, and uses it to generate

- A random sequence `seq` of emission symbols
- A random sequence `states` of states

The length of both `seq` and `states` is `len`.  $\text{TRANS}(i,j)$  is the probability of transition from state  $i$  to state  $j$ .  $\text{EMIS}(k,l)$  is the probability that symbol  $l$  is emitted from state  $k$ .

---

**Note** The function `hmmgenerate` begins with the model in state 1 at step 0, prior to the first emission. The model then makes a transition to state  $i_1$ , with probability  $T_{1i_1}$ , and generates an emission  $a_{k_1}$  with probability  $E_{i_1k_1}$ . `hmmgenerate` returns  $i_1$  as the first entry of `states`, and  $a_{k_1}$  as the first entry of `seq`.

---

`hmmgenerate(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. SYMBOLS can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through N, where N is the number of possible emissions.

`hmmgenerate(...,'Statenames',STATENAMES)` specifies the names of the states. STATENAMES can be a numeric array or a cell array of the names of the states. The default state names are 1 through M, where M is the number of states.

Since the model always begins at state 1, whose transition probabilities are in the first row of TRANS, in the following example, the first entry of

# hmmgenerate

---

the output states is be 1 with probability 0.95 and 2 with probability 0.05.

## Examples

```
trans = [0.95,0.05;
         0.10,0.90];
emis = [1/6 1/6 1/6 1/6 1/6 1/6;
        1/10 1/10 1/10 1/10 1/10 1/2];

[seq,states] = hmmgenerate(100,trans,emis)
[seq,states] = hmmgenerate(100,trans,emis,...
    'Symbols',{'one','two','three','four','five','six'},...
    'Statenames',{'fair';'loaded'})
```

## See Also

[hmmviterbi](#) | [hmmdecode](#) | [hmmestimate](#) | [hmmtrain](#)

**Purpose**

Hidden Markov model parameter estimates from emissions

**Syntax**

```
[ESTTR,ESTEMIT] = hmmtrain(seq,TRGUESS,EMITGUESS)
hmmtrain(...,'Algorithm',algorithm)
hmmtrain(...,'Symbols',SYMBOLS)
hmmtrain(...,'Tolerance',tol)
hmmtrain(...,'Maxiterations',maxiter)
hmmtrain(...,'Verbose',true)
hmmtrain(...,'Pseudoemissions',PSEUDOE)
hmmtrain(...,'Pseudotransitions',PSEUDOTR)
```

**Description**

[ESTTR,ESTEMIT] = `hmmtrain(seq,TRGUESS,EMITGUESS)` estimates the transition and emission probabilities for a hidden Markov model using the Baum-Welch algorithm. `seq` can be a row vector containing a single sequence, a matrix with one row per sequence, or a cell array with each cell containing a sequence. `TRGUESS` and `EMITGUESS` are initial estimates of the transition and emission probability matrices. `TRGUESS(i,j)` is the estimated probability of transition from state `i` to state `j`. `EMITGUESS(i,k)` is the estimated probability that symbol `k` is emitted from state `i`.

`hmmtrain(...,'Algorithm',algorithm)` specifies the training algorithm. *algorithm* can be either 'BaumWelch' or 'Viterbi'. The default algorithm is 'BaumWelch'.

`hmmtrain(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through `N`, where `N` is the number of possible emissions.

`hmmtrain(...,'Tolerance',tol)` specifies the tolerance used for testing convergence of the iterative estimation process. The default tolerance is `1e-4`.

`hmmtrain(...,'Maxiterations',maxiter)` specifies the maximum number of iterations for the estimation process. The default maximum is 100.

`hmmtrain(..., 'Verbose', true)` returns the status of the algorithm at each iteration.

`hmmtrain(..., 'Pseudoemissions', PSEUDOE)` specifies pseudocount emission values for the Viterbi training algorithm. Use this argument to avoid zero probability estimates for emissions with very low probability that might not be represented in the sample sequence. `PSEUDOE` should be a matrix of size  $m$ -by- $n$ , where  $m$  is the number of states in the hidden Markov model and  $n$  is the number of possible emissions. If the  $i \rightarrow k$  emission does not occur in `seq`, you can set `PSEUDOE(i, k)` to be a positive number representing an estimate of the expected number of such emissions in the sequence `seq`.

`hmmtrain(..., 'Pseudotransitions', PSEUDOTR)` specifies pseudocount transition values for the Viterbi training algorithm. Use this argument to avoid zero probability estimates for transitions with very low probability that might not be represented in the sample sequence. `PSEUDOTR` should be a matrix of size  $m$ -by- $m$ , where  $m$  is the number of states in the hidden Markov model. If the  $i \rightarrow j$  transition does not occur in `states`, you can set `PSEUDOTR(i, j)` to be a positive number representing an estimate of the expected number of such transitions in the sequence `states`.

If you know the states corresponding to the sequences, use `hmmestimate` to estimate the model parameters.

## Tolerance

The input argument `'tolerance'` controls how many steps the `hmmtrain` algorithm executes before the function returns an answer. The algorithm terminates when all of the following three quantities are less than the value that you specify for `tolerance`:

- The log likelihood that the input sequence `seq` is generated by the currently estimated values of the transition and emission matrices
- The change in the norm of the transition matrix, normalized by the size of the matrix
- The change in the norm of the emission matrix, normalized by the size of the matrix

The default value of 'tolerance' is .0001. Increasing the tolerance decreases the number of steps the hmmtrain algorithm executes before it terminates.

**maxiterations**

The maximum number of iterations, 'maxiterations', controls the maximum number of steps the algorithm executes before it terminates. If the algorithm executes maxiter iterations before reaching the specified tolerance, the algorithm terminates and the function returns a warning. If this occurs, you can increase the value of 'maxiterations' to make the algorithm reach the desired tolerance before terminating.

**References**

[1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

**Examples**

```
trans = [0.95,0.05;  
         0.10,0.90];  
emis = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6;  
        1/10, 1/10, 1/10, 1/10, 1/10, 1/2];  
  
seq1 = hmmgenerate(100,trans,emis);  
seq2 = hmmgenerate(200,trans,emis);  
seqs = {seq1,seq2};  
[estTR,estE] = hmmtrain(seqs,trans,emis);
```

**See Also**

[hmmgenerate](#) | [hmmdecode](#) | [hmmestimate](#) | [hmmviterbi](#)

# hmmviterbi

---

**Purpose** Hidden Markov model most probable state path

**Syntax**  
`STATES = hmmviterbi(seq,TRANS,EMIS)`  
`hmmviterbi(...,'Symbols',SYMBOLS)`  
`hmmviterbi(...,'Statenames',STATENAMES)`

**Description** `STATES = hmmviterbi(seq,TRANS,EMIS)` given a sequence, `seq`, calculates the most likely path through the hidden Markov model specified by transition probability matrix, `TRANS`, and emission probability matrix `EMIS`. `TRANS(i,j)` is the probability of transition from state `i` to state `j`. `EMIS(i,k)` is the probability that symbol `k` is emitted from state `i`.

---

**Note** The function `hmmviterbi` begins with the model in state 1 at step 0, prior to the first emission. `hmmviterbi` computes the most likely path based on the fact that the model begins in state 1.

---

`hmmviterbi(...,'Symbols',SYMBOLS)` specifies the symbols that are emitted. `SYMBOLS` can be a numeric array or a cell array of the names of the symbols. The default symbols are integers 1 through `N`, where `N` is the number of possible emissions.

`hmmviterbi(...,'Statenames',STATENAMES)` specifies the names of the states. `STATENAMES` can be a numeric array or a cell array of the names of the states. The default state names are 1 through `M`, where `M` is the number of states.

## Examples

```
trans = [0.95,0.05;  
         0.10,0.90];  
emis = [1/6 1/6 1/6 1/6 1/6 1/6;  
        1/10 1/10 1/10 1/10 1/10 1/2];  
  
[seq,states] = hmmgenerate(100,trans,emis);  
estimatedStates = hmmviterbi(seq,trans,emis);
```



```
[seq,states] = ...  
    hmmgenerate(100,trans,emis,...  
                'Statenames',{'fair';'loaded'});  
estimatesStates = ...  
    hmmviterbi(seq,trans,emis,...  
               'Statenames',{'fair';'loaded'});
```

## References

[1] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

## See Also

[hmmgenerate](#) | [hmmdecode](#) | [hmmestimate](#) | [hmmtrain](#)

# categorical.horzcat

---

**Purpose** Horizontal concatenation for categorical arrays

**Syntax** `C = horzcat(dim,A,B,...)`  
`C = horzcat(A,B)`

**Description** `C = horzcat(dim,A,B,...)` horizontally concatenates the categorical arrays `A,B,...`. For matrices, all inputs must have the same number of rows. For n-D arrays, all inputs must have the same sizes except in the second dimension. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs, as determined by their labels. `C = horzcat(A,B)` is called for the syntax `[A B]`.

**See Also** `cat` | `vertcat`

**Purpose** Horizontal concatenation for dataset arrays

**Syntax** `ds = horzcat(ds1, ds2, ...)`

**Description** `ds = horzcat(ds1, ds2, ...)` horizontally concatenates the dataset arrays `ds1`, `ds2`, ... . You may concatenate dataset arrays that have duplicate variable names, however, the variables must contain identical data, and `horzcat` includes only one copy of the variable in the output dataset.

Observation names for all dataset arrays that have them must be identical except for order. `horzcat` concatenates by matching observation names when present, or by position for datasets that do not have observation names.

**See Also** `cat` | `vertcat`

# hougen

---

**Purpose** Hougen-Watson model

**Syntax** `yhat = hougen(beta,x)`

**Description** `yhat = hougen(beta,x)` returns the predicted values of the reaction rate, `yhat`, as a function of the vector of parameters, `beta`, and the matrix of data, `X`. `beta` must have 5 elements and `X` must have three columns.

`hougen` is a utility function for `rsmdemo`.

The model form is:

$$\hat{y} = \frac{\beta_1 x_2 - x_3 / \beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

**References** [1] Bates, D. M., and D. G. Watts. *Nonlinear Regression Analysis and Its Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1988.

**See Also** `rsmdemo`

**Purpose** Hypergeometric cumulative distribution function

**Syntax** `hygecdf(x,M,K,N)`  
`hygecdf(x,M,K,N,'upper')`

**Description** `hygecdf(x,M,K,N)` computes the hypergeometric cdf at each of the values in `x` using the corresponding size of the population, `M`, number of items with the desired characteristic in the population, `K`, and number of samples drawn, `N`. Vector or matrix inputs for `x`, `M`, `K`, and `N` must all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

`hygecdf(x,M,K,N,'upper')` returns the complement of the hypergeometric cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The hypergeometric cdf is

$$p = F(x | M, K, N) = \sum_{i=0}^x \frac{\binom{K}{i} \binom{M-K}{N-i}}{\binom{M}{N}}$$

The result,  $p$ , is the probability of drawing up to  $x$  of a possible  $K$  items in  $N$  drawings without replacement from a group of  $M$  objects.

## Examples **Compute Hypergeometric Distribution CDF**

Suppose you have a lot of 100 floppy disks and you know that 20 of them are defective. What is the probability of drawing zero to two defective floppies if you select 10 at random?

`p = hygecdf(2,100,20,10)`

`p =`  
 0.6812

# hygecdf

---

## See Also

[cdf](#) | [hygepdf](#) | [hygeinv](#) | [hygestat](#) | [hygernd](#)

---

<b>Purpose</b>	Hypergeometric inverse cumulative distribution function
<b>Syntax</b>	hygeinv(P,M,K,N)
<b>Description</b>	hygeinv(P,M,K,N) returns the smallest integer $X$ such that the hypergeometric cdf evaluated at $X$ equals or exceeds $P$ . You can think of $P$ as the probability of observing $X$ defective items in $N$ drawings without replacement from a group of $M$ items where $K$ are defective.
<b>Examples</b>	<p>Suppose you are the Quality Assurance manager for a floppy disk manufacturer. The production line turns out floppy disks in batches of 1,000. You want to sample 50 disks from each batch to see if they have defects. You want to accept 99% of the batches if there are no more than 10 defective disks in the batch. What is the maximum number of defective disks should you allow in your sample of 50?</p> <pre>x = hygeinv(0.99,1000,10,50) x =     3</pre> <p>What is the median number of defective floppy disks in samples of 50 disks from batches with 10 defective disks?</p> <pre>x = hygeinv(0.50,1000,10,50) x =     0</pre>
<b>See Also</b>	icdf   hygecdf   hygepdf   hygestat   hygernd

# hygepdf

---

**Purpose** Hypergeometric probability density function

**Syntax** `Y = hygepdf(X,M,K,N)`

**Description** `Y = hygepdf(X,M,K,N)` computes the hypergeometric pdf at each of the values in `X` using the corresponding size of the population, `M`, number of items with the desired characteristic in the population, `K`, and number of samples drawn, `N`. `X`, `M`, `K`, and `N` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

The parameters in `M`, `K`, and `N` must all be positive integers, with  $N \leq M$ . The values in `X` must be less than or equal to all the parameter values.

The hypergeometric pdf is

$$y = f(x | M, K, N) = \frac{\binom{K}{x} \binom{M-K}{N-x}}{\binom{M}{N}}$$

The result, `y`, is the probability of drawing exactly `x` of a possible `K` items in `n` drawings without replacement from a group of `M` objects.

**Examples** Suppose you have a lot of 100 floppy disks and you know that 20 of them are defective. What is the probability of drawing 0 through 5 defective floppy disks if you select 10 at random?

```
p = hygepdf(0:5,100,20,10)
p =
    0.0951    0.2679    0.3182    0.2092    0.0841    0.0215
```

**See Also** `pdf` | `hygecdf` | `hygeinv` | `hygestat` | `hygernd`



**Purpose** Hypergeometric random numbers

**Syntax**  
R = hygernd(M,K,N)  
R = hygernd(M,K,N,m,n,...)  
R = hygernd(M,K,N,[m,n,...])

**Description** R = hygernd(M,K,N) generates random numbers from the hypergeometric distribution with corresponding size of the population, M, number of items with the desired characteristic in the population, K, and number of samples drawn, N. M, K, and N can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of R. A scalar input for M, K, or N is expanded to a constant array with the same dimensions as the other inputs.

R = hygernd(M,K,N,m,n,...) or R = hygernd(M,K,N,[m,n,...]) generates an m-by-n-by-... array. The M, K, N parameters can each be scalars or arrays of the same size as R.

**Examples**  
numbers = hygernd(1000,40,50)  
numbers =  
1

**See Also** random | hygepdf | hygecdf | hygeinv | hygestat

# hygestat

---

**Purpose** Hypergeometric mean and variance

**Syntax** `[MN,V] = hygestat(M,K,N)`

**Description** `[MN,V] = hygestat(M,K,N)` returns the mean of and variance for the hypergeometric distribution with corresponding size of the population, M, number of items with the desired characteristic in the population, K, and number of samples drawn, N. Vector or matrix inputs for M, K, and N must have the same size, which is also the size of MN and V. A scalar input for M, K, or N is expanded to a constant matrix with the same dimensions as the other inputs.

The mean of the hypergeometric distribution with parameters M, K, and N is  $NK/M$ , and the variance is  $NK(M-K)(M-N) / [M^2(M-1)]$ .

**Examples** The hypergeometric distribution approaches the binomial distribution, where  $p = K/M$ , as M goes to infinity.

```
[m,v] = hygestat(10.^(1:4),10.^(0:3),9)
m =
    0.9000    0.9000    0.9000    0.9000
v =
    0.0900    0.7445    0.8035    0.8094
```

```
[m,v] = binostat(9,0.1)
m =
    0.9000
v =
    0.8100
```

**See Also** `hygepdf` | `hygecdf` | `hygeinv` | `hygernd`

**Purpose**

Inverse cumulative distribution functions

**Syntax**

```
Y = icdf(name,X,A)
Y = icdf(name,X,A,B)
Y = icdf(name,X,A,B,C)
```

**Description**

`Y = icdf(name,X,A)` computes the inverse cumulative distribution function for the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`. The inverse cumulative distribution function is evaluated at the values in `X` and its values are returned in `Y`.

If `X` and `A` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`. If `A` is a scalar, it is expanded to a constant matrix the same size as `X`.

`Y` is the common size of `X` and `A` after any necessary scalar expansion.

`Y = icdf(name,X,A,B)` computes the inverse cumulative distribution function for two-parameter families of distributions, where parameter values are given in `A` and `B`.

If `X`, `A`, and `B` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A` and `B`. If either `A` or `B` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, and `B` after any necessary scalar expansion.

`Y = icdf(name,X,A,B,C)` computes the inverse cumulative distribution function for three-parameter families of distributions, where parameter values are given in `A`, `B`, and `C`.

If `X`, `A`, `B`, and `C` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`, `B`, and `C`. If any of `A`, `B` or `C` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, `B` and `C` after any necessary scalar expansion.

Acceptable strings for `name` are:

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'beta' or 'Beta'	“Beta Distribution” on page B-5	a	b	—
'bino' or 'Binomial'	“Binomial Distribution” on page B-9	n: number of trials	p: probability of success for each trial	—
'birnbaumsaunders'	“Birnbbaum-Saunders Distribution” on page B-13	$\beta$	$\gamma$	—
'burr' or 'Burr'	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale parameter	c: shape parameter	k: shape parameter
'chi2' or 'Chisquare'	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—
'exp' or 'Exponential'	“Exponential Distribution” on page B-32	$\mu$ : mean	—	—
'ev' or 'Extreme Value'	“Extreme Value Distribution” on page B-35	$\mu$ : location parameter	$\sigma$ : scale parameter	—
'f' or 'F'	“F Distribution” on page B-41	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	—
'gam' or 'Gamma'	“Gamma Distribution” on page B-43	a: shape parameter	b: scale parameter	—
'gev' or 'Generalized Extreme Value'	“Generalized Extreme Value Distribution” on page B-48	k: shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'gp' or 'Generalized Pareto'	“Generalized Pareto Distribution” on page B-53	k: tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter
'geo' or 'Geometric'	“Geometric Distribution” on page B-57	p: probability parameter	—	—
'hyge' or 'Hypergeometric'	“Hypergeometric Distribution” on page B-60	M: size of the population	K: number of items with the desired characteristic in the population	n: number of samples drawn
'inversegaussian'	“Inverse Gaussian Distribution” on page B-63	$\mu$	$\lambda$	—
'logistic'	“Logistic Distribution” on page B-76	$\mu$	$\sigma$	—
'loglogistic'	“Loglogistic Distribution” on page B-78	$\mu$	$\sigma$	—
'logn' or 'Lognormal'	“Lognormal Distribution” on page B-80	$\mu$	$\sigma$	—
'nakagami'	“Nakagami Distribution” on page B-97	$\mu$	$\omega$	—
'nbin' or 'Negative Binomial'	“Negative Binomial Distribution” on page B-98	r: number of successes	p: probability of success in a single trial	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'ncf' or 'Noncentral F'	“Noncentral F Distribution” on page B-106	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	$\delta$ : noncentrality parameter
'nct' or 'Noncentral t'	“Noncentral t Distribution” on page B-108	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—
'ncx2' or 'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-104	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—
'norm' or 'Normal'	“Normal Distribution” on page B-110	$\mu$ : mean	$\sigma$ : standard deviation	—
'poiss' or 'Poisson'	“Poisson Distribution” on page B-118	$\lambda$ : mean	—	—
'ray1' or 'Rayleigh'	“Rayleigh Distribution” on page B-121	$b$ : scale parameter	—	—
'rician'	“Rician Distribution” on page B-123	$s$ : noncentrality parameter	$\sigma$ : scale parameter	—
't' or 'T'	“Student’s t Distribution” on page B-124	$\nu$ : degrees of freedom	—	—
'tlocationscale'	“t Location-Scale Distribution” on page B-126	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter
'unif' or 'Uniform'	“Uniform Distribution (Continuous)” on page B-133	$a$ : lower endpoint (minimum)	$b$ : upper endpoint (maximum)	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'unid' or 'Discrete Uniform'	“Uniform Distribution (Discrete)” on page B-139	N: maximum observable value	—	—
'wbl' or 'Weibull'	“Weibull Distribution” on page B-141	a: scale parameter	b: shape parameter	—

## Examples

Compute the icdf of the normal distribution with mean 0 and standard deviation 1 at inputs 0.1, 0.3, ..., 0.9:

```
x1 = icdf('Normal',0.1:0.2:0.9,0,1)
x1 =
    -1.2816   -0.5244    0    0.5244    1.2816
```

The order of the parameters is the same as for `norminv`.

Compute the icdfs of Poisson distributions with rate parameters 0, 1, ..., 4 at inputs 0.1, 0.3, ..., 0.9, respectively:

```
x2 = icdf('Poisson',0.1:0.2:0.9,0:4)
x2 =
    NaN    0    2    4    7
```

The order of the parameters is the same as for `poissinv`.

## See Also

`cdf` | `mle` | `pdf` | `random`

# piecewisedistribution.icdf

---

**Purpose** Inverse cumulative distribution function for piecewise distribution

**Syntax** `X = icdf(obj,P)`

**Description** `X = icdf(obj,P)` returns an array `X` of values of the inverse cumulative distribution function for the piecewise distribution object `obj`, evaluated at the values in the array `P`.

**Examples** Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432

icdf(obj,p)
ans =
   -1.7766
    1.8432
```

**See Also** `paretotails` | `cdf`



<b>Purpose</b>	Return inverse cumulative distribution function (ICDF) for ProbDistUnivKernel object	
<b>Syntax</b>	$Y = \text{icdf}(PD, P)$	
<b>Description</b>	$Y = \text{icdf}(PD, P)$ returns $Y$ , an array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivKernel object $PD$ , evaluated at values in $P$ .	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivKernel.
	$P$	A numeric array of values from 0 to 1 where you want to evaluate the ICDF.
<b>Output Arguments</b>	$Y$	An array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivKernel object $PD$ .
<b>See Also</b>	icdf	

# ProbDistUnivParam.icdf

---

<b>Purpose</b>	Return inverse cumulative distribution function (ICDF) for ProbDistUnivParam object	
<b>Syntax</b>	$Y = \text{icdf}(PD, P)$	
<b>Description</b>	$Y = \text{icdf}(PD, P)$ returns $Y$ , an array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivParam object $PD$ , evaluated at values in $P$ .	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivParam.
	$P$	A numeric array of values from 0 to 1 where you want to evaluate the ICDF.
<b>Output Arguments</b>	$Y$	An array containing the inverse cumulative distribution function (ICDF) for the ProbDistUnivParam object $PD$ .
<b>See Also</b>	icdf	

<b>Purpose</b>	Inverse cumulative distribution function of probability distribution object
<b>Syntax</b>	<code>y = icdf(pd,prob)</code>
<b>Description</b>	<code>y = icdf(pd,prob)</code> returns the inverse cumulative distribution function (icdf) values of the probability distribution <code>pd</code> at the probabilities in <code>prob</code> .
<b>Input Arguments</b>	<p><b>pd - Probability distribution</b> probability distribution object</p> <p>Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code>. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using <code>fitdist</code> or the Distribution Fitting app.</p> <p><b>prob - Probabilities</b> array of scalar values in the range [0,1]</p> <p>Probabilities at which to compute the icdf, specified as an array of scalar values in the range [0,1]. For example, specifying <code>[.25 .5 .75]</code> returns a vector containing three icdf values corresponding to these probabilities.</p> <p><b>Data Types</b> single   double</p>
<b>Output Arguments</b>	<p><b>y - Inverse cumulative distribution function</b> array</p> <p>Inverse cumulative distribution function (icdf) values of the specified probability distribution, evaluated at the probabilities in <code>prob</code>, returned as an array. <code>y</code> has the same dimensions as <code>x</code>.</p>
<b>Examples</b>	<p><b>Compute Standard Normal Critical Values</b></p> <p>Create a standard normal distribution object.</p>

# prob.TruncatableDistribution.icdf

---

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Determine the critical values at the 5% significance level for a test statistic with a standard normal distribution, by computing the upper and lower 2.5% values.

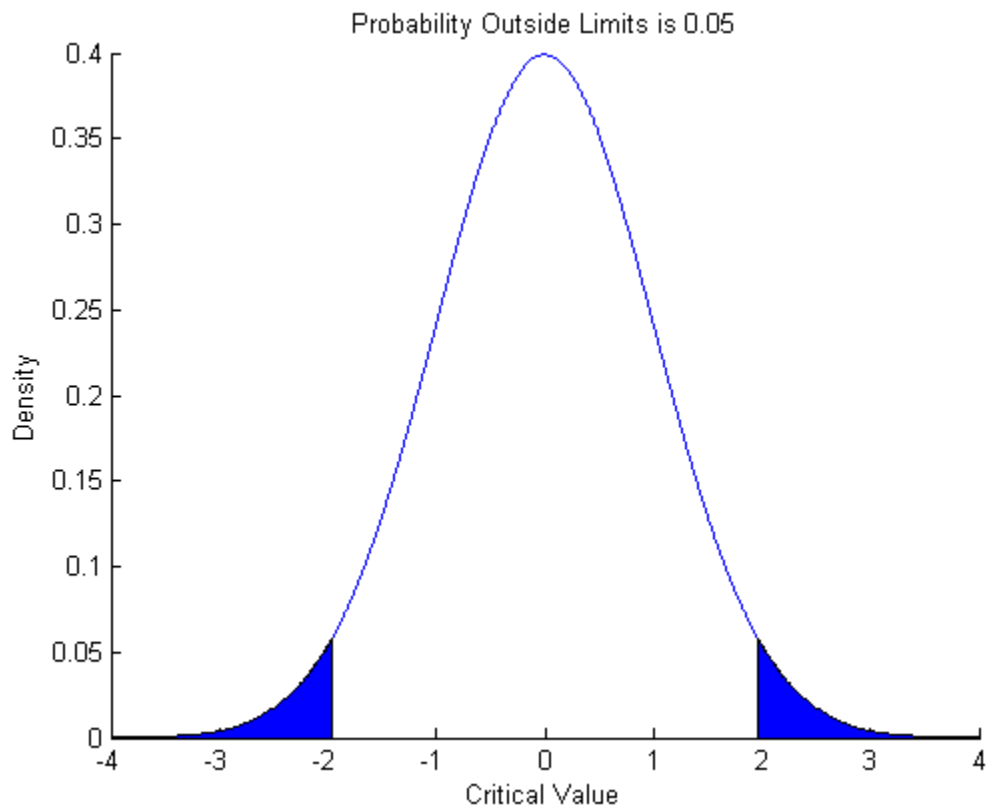
```
y = icdf(pd, [.025, .975])
```

```
y =
```

```
-1.9600    1.9600
```

Plot the cdf and shade the critical regions.

```
p = normspec(y,0,1,'outside')
```



## See Also

`makedist` | `fitdist` | `cdf` | `pdf` | `normspec` | `dfittool`

# inconsistent

---

**Purpose** Inconsistency coefficient

**Syntax**  $Y = \text{inconsistent}(Z)$   
 $Y = \text{inconsistent}(Z, d)$

**Description**  $Y = \text{inconsistent}(Z)$  computes the inconsistency coefficient for each link of the hierarchical cluster tree  $Z$ , where  $Z$  is an  $(m-1)$ -by-3 matrix generated by the linkage function. The inconsistency coefficient characterizes each link in a cluster tree by comparing its height with the average height of other links at the same level of the hierarchy. The higher the value of this coefficient, the less similar the objects connected by the link.

$Y = \text{inconsistent}(Z, d)$  computes the inconsistency coefficient for each link in the hierarchical cluster tree  $Z$  to depth  $d$ , where  $d$  is an integer denoting the number of levels of the cluster tree that are included in the calculation. By default,  $d=2$ .

The output,  $Y$ , is an  $(m-1)$ -by-4 matrix formatted as follows.

Column	Description
1	Mean of the heights of all the links included in the calculation.
2	Standard deviation of the heights of all the links included in the calculation.
3	Number of links included in the calculation.
4	Inconsistency coefficient.

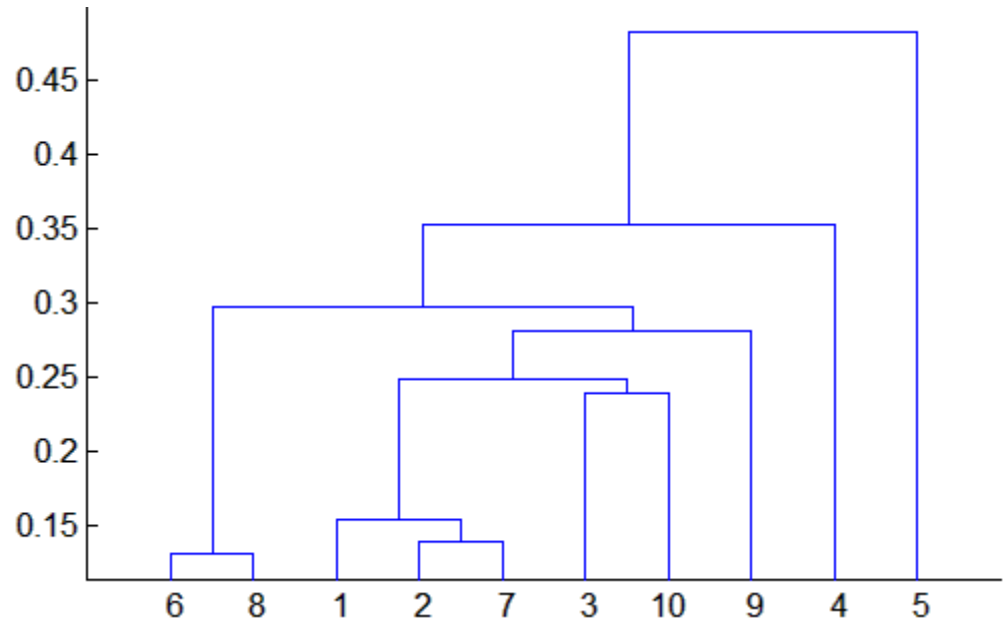
For each link,  $k$ , the inconsistency coefficient is calculated as:

$$Y(k, 4) = (z(k, 3) - Y(k, 1)) / Y(k, 2)$$

For leaf nodes, nodes that have no further nodes under them, the inconsistency coefficient is set to 0.

## Examples

```
X = gallery('uniformdata',[10 2],12);
Y = pdist(X);
Z = linkage(Y,'single');
dendrogram(Z)
```



```
W = inconsistent(Z,3)
```

```
W =
```

0.1313	0	1.0000	0
0.1386	0	1.0000	0
0.1463	0.0109	2.0000	0.7071
0.2391	0	1.0000	0
0.1951	0.0568	4.0000	0.9425
0.2308	0.0543	4.0000	0.9320
0.2395	0.0748	4.0000	0.7636
0.2654	0.0945	4.0000	0.9203
0.3769	0.0950	3.0000	1.1040

# inconsistent

---

## References

[1] Jain, A., and R. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ: Prentice-Hall, 1988.

[2] Zahn, C. T. "Graph-theoretical methods for detecting and describing Gestalt clusters." *IEEE Transactions on Computers*. Vol. C-20, Issue 1, 1971, pp. 68–86.

## See Also

`cluster` | `cophenet` | `clusterdata` | `dendrogram` | `linkage` | `pdist` | `squareform`



<b>Purpose</b>	Read-only structure containing information about input data to ProbDist object
<b>Description</b>	<p>InputData is a read-only property of the ProbDist class. InputData is a structure containing information about input data to a ProbDist object. It includes the following fields:</p> <ul style="list-style-type: none"><li>• data</li><li>• cens</li><li>• freq</li></ul>
<b>Values</b>	<p>Possible values for the three fields in the structure are any data supplied to the <code>fitdist</code> function:</p> <ul style="list-style-type: none"><li>• <code>data</code> — Data passed to the <code>fitdist</code> function when creating the ProbDist object. This field is empty if the ProbDist object was created without fitting to data, that is by using the <code>ProbDistUnivParam.ProbDistUnivParam</code> constructor.</li><li>• <code>cens</code> — The vector supplied with the 'censoring' parameter when creating the ProbDist object using the <code>fitdist</code> function. This field is empty if the ProbDist object was created without fitting to data, that is by using the <code>ProbDistUnivParam.ProbDistUnivParam</code> constructor.</li><li>• <code>freq</code> — The vector supplied with the 'frequency' parameter when creating the ProbDist object using the <code>fitdist</code> function. This field is empty if the ProbDist object was created without fitting to data, that is by using the <code>ProbDistUnivParam.ProbDistUnivParam</code> constructor.</li></ul> <p>Use this information to view and compare the data supplied to create distributions.</p>

# categorical.int8

---

**Purpose** Convert categorical array to signed 8-bit integer array

**Syntax** `B = int8(A)`

**Description** `B = int8(A)` converts the categorical array `A` to a signed 8-bit integer array. Each element of `B` contains the internal categorical level code for the corresponding element of `A`.

Undefined elements of `A` are assigned the value 0 in `B`. If `A` contains more than `intmax('int8')` levels, the internal codes will saturate to `intmax('int8')` when cast to `int8`.

**See Also** `double` | `uint8`

**How To**

- “Integers”

<b>Purpose</b>	Convert categorical array to signed 16-bit integer array
<b>Syntax</b>	<code>B = int16(A)</code>
<b>Description</b>	<p><code>B = int16(A)</code> converts the categorical array <code>A</code> to a signed 16-bit integer array. Each element of <code>B</code> contains the internal categorical level code for the corresponding element of <code>A</code>.</p> <p>Undefined elements of <code>A</code> are assigned the value 0 in <code>B</code>.</p>
<b>See Also</b>	<code>double</code>   <code>uint16</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Integers”</li></ul>

# categorical.int32

---

**Purpose** Convert categorical array to signed 32-bit integer array

**Syntax** `B = int32(A)`

**Description** `B = int32(A)` converts the categorical array `A` to a signed 32-bit integer array. Each element of `B` contains the internal categorical level code for the corresponding element of `A`.

Undefined elements of `A` are assigned the value 0 in `B`.

**See Also** `double` | `uint32`

**How To**

- “Integers”

<b>Purpose</b>	Convert categorical array to signed 64-bit integer array
<b>Syntax</b>	<code>B = int64(A)</code>
<b>Description</b>	<p><code>B = int64(A)</code> converts the categorical array <code>A</code> to a signed 64-bit integer array. Each element of <code>B</code> contains the internal categorical level code for the corresponding element of <code>A</code>.</p> <p>Undefined elements of <code>A</code> are assigned the value 0 in <code>B</code>.</p>
<b>See Also</b>	<code>double</code>   <code>uint64</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Integers”</li></ul>

# interactionplot

---

**Purpose** Interaction plot for grouped data

**Syntax**  
`interactionplot(Y,GROUP)`  
`interactionplot(Y,GROUP,'varnames',VARNAMES)`  
`[h,AX,bigax] = interactionplot(...)`

**Description** `interactionplot(Y,GROUP)` displays the two-factor interaction plot for the group means of matrix `Y` with groups defined by entries in the cell array `GROUP`. `Y` is a numeric matrix or vector. If `Y` is a matrix, the rows represent different observations and the columns represent replications of each observation. If `Y` is a vector, the rows give the means of each entry in the cell array `GROUP`. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or a single-column cell array of strings. `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of rows as `Y`. The number of grouping variables must be greater than 1.

The interaction plot is a matrix plot, with the number of rows and columns both equal to the number of grouping variables. The grouping variable names are printed on the diagonal of the plot matrix. The plot at off-diagonal position  $(i,j)$  is the interaction of the two variables whose names are given at row diagonal  $(i,i)$  and column diagonal  $(j,j)$ , respectively.

`interactionplot(Y,GROUP,'varnames',VARNAMES)` displays the interaction plot with user-specified grouping variable names `VARNAMES`. `VARNAMES` is a character matrix or a cell array of strings, one per grouping variable. Default names are 'X1', 'X2', ... .

`[h,AX,bigax] = interactionplot(...)` returns a handle `h` to the figure window, a matrix `AX` of handles to the subplot axes, and a handle `bigax` to the big (invisible) axes framing the subplots.

## Examples **Display Interaction Plots**

Randomly generate data for a response variable `y` .

```
rng('default'); % for reproducibility  
y = randn(1000,1);
```

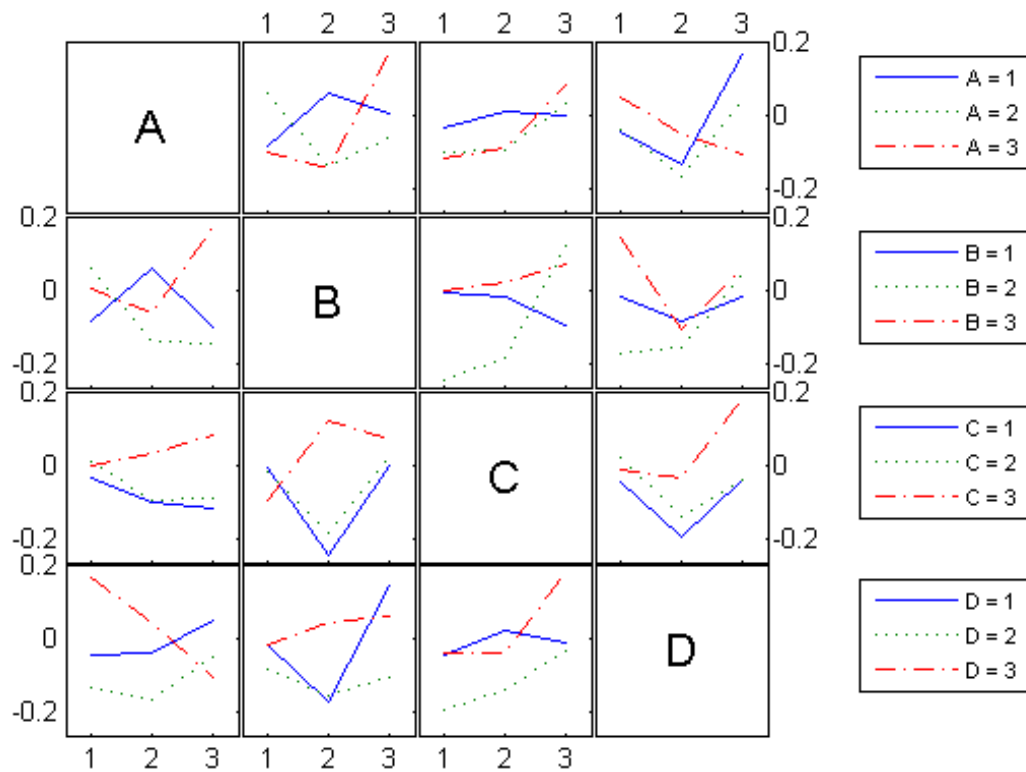
Randomly generate data for four three-level factors.

```
group = ceil(3*rand(1000,4));
```

Display the interaction plots for the factors and name the factors 'A', 'B', 'C', 'D'.

```
interactionplot(y,group,'varnames',{'A','B','C','D'})
```

# interactionplot



**See Also** [maineffectsplot](#) | [multivarichart](#)



**Purpose** Set intersection for categorical arrays

---

**Note** In a future release, the behavior of `categorical.intersect` will change to be consistent with the MATLAB function `intersect`. This behavior change is optional in R2012a. For a demonstration of using the 'R2012a' flag to preview the future behavior, or the 'legacy' flag to preserve the current behavior in your existing code, see the documentation for `intersect`.

---

**Syntax**

```
C = intersect(A,B)
[C,IA,IB] = intersect(A,B)
[...] = intersect(A,B,'rows')
[...] = intersect(...,'R2012a')
[...] = intersect(...,'legacy')
[...] = intersect(A,B,setOrder)
[...] = intersect(A,B,'rows',setOrder)
```

**Description** `C = intersect(A,B)` for categorical vectors A and B, returns a categorical vector C containing the values common to the two vectors with no repetitions. The result C is sorted. The set of categorical levels for C is the sorted union of the sets of levels of the inputs.

`[C,IA,IB] = intersect(A,B)` also returns index vectors IA and IB such that `C = A(IA)` and `C = B(IB)`. If there are repeated common values in A or B, then the index of the last occurrence of each repeated value is returned.

`[...] = intersect(A,B,'rows')` for categorical matrices A and B with the same number of columns, returns the rows common to the two matrices. The rows of the matrix C are sorted. The set of categorical levels for C is the sorted union of the sets of levels of the inputs. The optional outputs IA and IB are index vectors such that `C = A(IA,:)` and `C = B(IB,:)`.

## categorical.intersect

---

`[...] = intersect(..., 'R2012a')` adopts the future behavior of `intersect`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

`[...] = intersect(..., 'legacy')` preserves the current behavior of `intersect`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

`[...] = intersect(A,B,setOrder)` and `[...] = intersect(A,B, 'rows', setOrder)` returns the observations of `C` in a specific order. `setOrder='sorted'` returns the values or rows of `C` in sorted order. `setOrder='stable'` returns the values or rows of `C` in the same order as `A`. If `A` and `B` are row vectors, then `C` is also a row vector. Otherwise, `C` is a column vector. If `A` and `B` are column vectors, then `C` is also a column vector. If there are repeated common values in `A` or `B`, then the index of the first occurrence of each repeated value is returned.

### See Also

`ismember` | `setdiff` | `setxor` | `union` | `unique`

## Purpose

Set intersection for dataset array observations

## Syntax

```
C = intersect(A,B)
C = intersect(A,B,vars)
C = intersect(A,B,vars,setOrder)
[C,iA,iB] = intersect( ___ )
```

## Description

`C = intersect(A,B)` for dataset arrays `A` and `B` returns the common set of observations from the two arrays, with repetitions removed. The observations in the dataset array `C` are in sorted order.

`C = intersect(A,B,vars)` returns the set of common observations from the two arrays, considering only the variables specified in `vars`, with repetitions removed. The observations in the dataset array `C` are sorted by those variables.

The values for variables not specified in `vars` for each observation in `C` are taken from the corresponding observations in `A`. If there are multiple observations in `A` that correspond to an observation in `C`, then those values are taken from the first occurrence.

`C = intersect(A,B,vars,setOrder)` returns the observations in `C` in the order specified by `setOrder`.

`[C,iA,iB] = intersect( ___ )` also returns index vectors `iA` and `iB` such that `C = A(iA,:)` and `C = B(iB,:)`. If there are repeated observations in `A` or `B`, then `intersect` returns the index of the first occurrence. You can use any of the previous input arguments.

## Input Arguments

### **A,B**

Input dataset arrays.

### **vars**

Cell array of strings containing variable names or a vector of integers containing variable column numbers, indicating the variables in `A` and `B` that `intersect` considers.

Specify `vars` as `[]` to use its default value of all variables.

# dataset.intersect

---

## setOrder

Flag indicating the sorting order for the observations in C. The possible values of setOrder are:

'sorted'	Observations in C are in sorted order (default).
'stable'	Observations in C are in the same order that they appear in A.

## Output Arguments

### C

Dataset array with the common set of observations in A and B, with repetitions removed. C is in sorted order (by default), or the order specified by setOrder.

### iA

Index vector, indicating the observations in A that are common to B. The vector iA contains the index to the first occurrence of any repeated observations in A.

### iB

Index vector, indicating the observations in B that are common to A. The vector iB contains the index to the first occurrence of any repeated observations in B.

## Examples

### Intersection of Two Dataset Arrays

Navigate to the folder containing sample data, and load sample data.

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

A = dataset('XLSFile','hospitalSmall.xlsx');
B = dataset('XLSFile','hospitalSmall.xlsx','Sheet',2);
```

Return the intersection and index vectors.

```
[C,iA,iB] = intersect(A,B);
```

```
C =
```

id	name	sex	age	wgt	smoke
'TRW-072'	'WHITE'	'm'	39	202	1

There is one observation in common between A and B.

Find the observation in the original dataset arrays.

```
A(iA,:)
```

```
ans =
```

id	name	sex	age	wgt	smoke
'TRW-072'	'WHITE'	'm'	39	202	1

```
B(iB,:)
```

```
ans =
```

id	name	sex	age	wgt	smoke
'TRW-072'	'WHITE'	'm'	39	202	1

## See Also

[dataset](#) | [ismember](#) | [setdiff](#) | [setxor](#) | [sortrows](#) | [union](#) | [unique](#) |

## Related Examples

- “Merge Dataset Arrays” on page 2-101

## Concepts

- “Dataset Arrays” on page 2-135

# prob.InverseGaussianDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Inverse Gaussian probability distribution object

**Description** `prob.InverseGaussianDistribution` is an object consisting of parameters, a model description, and sample data for an inverse Gaussian probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('InverseGaussian')` creates an inverse Gaussian probability distribution object using the default parameter values.

`pd = makedist('InverseGaussian','mu',mu,'lambda',lambda)` creates an inverse Gaussian probability distribution object using the specified parameter values.

## Input Arguments

### **mu - Scale parameter**

1 (default) | positive scalar value

Scale parameter for the inverse Gaussian distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **lambda - Shape parameter**

1 (default) | positive scalar value

Shape parameter for the inverse Gaussian distribution, specified as a positive scalar value.

### **Data Types**

single | double

## Properties

### **mu**

Scale parameter for the inverse Gaussian distribution, stored as a positive scalar value.

#### **Data Types**

single | double

### **lambda**

Shape parameter for the inverse Gaussian distribution, stored as a positive scalar value.

#### **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

#### **Data Types**

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

#### **Data Types**

single | double

### **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

# prob.InverseGaussianDistribution

---

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.



## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object

# prob.InverseGaussianDistribution

---

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions **Inverse Gaussian Distribution**

Also known as the Wald distribution, the inverse Gaussian is used to model nonnegative positively skewed data. Inverse Gaussian distributions have many similarities to standard Gaussian (normal) distributions, which lead to applications in inferential statistics.

The inverse Gaussian distribution uses the following parameters.

Parameter	Description	Support
mu	Scale parameter	$\mu > 0$
lambda	Shape parameter	$\lambda > 0$

The probability density function (pdf) is

$$f(x | \mu, \lambda) = \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left\{-\frac{\lambda}{2\mu^2 x}(x - \mu)^2\right\} ; \quad x > 0.$$

## Examples

### Create an Inverse Gaussian Distribution Object Using Default Parameters

Create an inverse Gaussian distribution object using the default parameter values.

```
pd = makedist('InverseGaussian')
```

```
pd =
```

```
InverseGaussianDistribution
```

```
Inverse Gaussian distribution
```

```
mu = 1
```

```
lambda = 1
```

### Create an Inverse Gaussian Distribution Object Using Specified Parameters

Create an inverse Gaussian distribution object by specifying parameter values.

```
pd = makedist('InverseGaussian','mu',2,'lambda',4)
```

```
pd =
```

# prob.InverseGaussianDistribution

---

```
InverseGaussianDistribution
```

```
Inverse Gaussian distribution
```

```
mu = 2
```

```
lambda = 4
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

```
1.4142
```

## See Also

```
makedist | fitdist | dfittool
```

## Concepts

- “Inverse Gaussian Distribution” on page B-63
- Class Attributes
- Property Attributes

**Purpose** Inverse prediction

**Syntax**  
`X0 = invpred(X,Y,Y0)`  
`[X0,DXLO,DXUP] = invpred(X,Y,Y0)`  
`[X0,DXLO,DXUP] = invpred(X,Y,Y0,name1,val1,name2,val2,...)`

**Description** `X0 = invpred(X,Y,Y0)` accepts vectors `X` and `Y` of the same length, fits a simple regression, and returns the estimated value `X0` for which the height of the line is equal to `Y0`. The output, `X0`, has the same size as `Y0`, and `Y0` can be an array of any size.

`[X0,DXLO,DXUP] = invpred(X,Y,Y0)` also computes 95% inverse prediction intervals. `DXLO` and `DXUP` define intervals with lower bound `X0 - DXLO` and upper bound `X0+DXUP`. Both `DXLO` and `DXUP` have the same size as `Y0`.

The intervals are not simultaneous and are not necessarily finite. Some intervals may extend from a finite value to `-Inf` or `+Inf`, and some may extend over the entire real line.

`[X0,DXLO,DXUP] = invpred(X,Y,Y0,name1,val1,name2,val2,...)` specifies optional argument name/value pairs chosen from the following list. Argument names are case insensitive and partial matches are allowed.

Name	Value
'alpha'	A value between 0 and 1 specifying a confidence level of $100 \cdot (1 - \text{alpha})\%$ . Default is <code>alpha=0.05</code> for 95% confidence.
'predopt'	Either <code>'observation'</code> , the default value to compute the intervals for <code>X0</code> at which a new observation could equal <code>Y0</code> , or <code>'curve'</code> to compute intervals for the <code>X0</code> value at which the curve is equal to <code>Y0</code> .

# invpred

---

## Examples

```
x = 4*rand(25,1);  
y = 10 + 5*x + randn(size(x));  
scatter(x,y)  
x0 = invpred(x,y,20)
```

## See Also

[polyfit](#) | [polyval](#) | [polyconf](#) | [polytool](#)

<b>Purpose</b>	Inverse permute dimensions of categorical array
<b>Syntax</b>	<code>A = ipermute(B,order)</code>
<b>Description</b>	<code>A = ipermute(B,order)</code> is the inverse of <code>permute</code> . <code>ipermute</code> rearranges the dimensions of the categorical array <code>B</code> so that <code>permute(A,order)</code> will produce <code>B</code> . The array produced has the same values of <code>A</code> but the order of the subscripts needed to access any particular element are rearranged as specified by <code>order</code> . The elements of <code>order</code> must be a rearrangement of the numbers from 1 to <code>n</code> .
<b>See Also</b>	<code>permute</code>

# iqr

---

**Purpose** Interquartile range

**Syntax** `y = iqr(X)`  
`iqr(X,dim)`

**Description** `y = iqr(X)` returns the interquartile range of the values in `X`. For vector input, `y` is the difference between the 75th and the 25th percentiles of the sample in `X`. For matrix input, `y` is a row vector containing the interquartile range of each column of `X`. For N-dimensional arrays, `iqr` operates along the first nonsingleton dimension of `X`.

`iqr(X,dim)` calculates the interquartile range along the dimension `dim` of `X`.

**Tips** The IQR is a robust estimate of the spread of the data, since changes in the upper and lower 25% of the data do not affect it. If there are outliers in the data, then the IQR is more representative than the standard deviation as an estimate of the spread of the body of the data. The IQR is less efficient than the standard deviation as an estimate of the spread when the data is all from the normal distribution.

Multiply the IQR by 0.7413 to estimate  $\sigma$  (the second parameter of the normal distribution.)

**Examples** This Monte Carlo simulation shows the relative efficiency of the IQR to the sample standard deviation for normal data.

```
x = normrnd(0,1,100,100);
s = std(x);
s_IQR = 0.7413*iqr(x);
efficiency = (norm(s-1)./norm(s_IQR-1)).^2
efficiency =
    0.3297
```

**See Also** `std` | `mad` | `range`



<b>Purpose</b>	Return interquartile range (IQR) for ProbDistUnivKernel object		
<b>Syntax</b>	$Y = \text{iqr}(PD)$		
<b>Description</b>	$Y = \text{iqr}(PD)$ returns $Y$ , the interquartile range for the ProbDistUnivKernel object $PD$ . The interquartile range is the distance between the 75th and 25th percentiles.		
<b>Input Arguments</b>	<table><tr><td><math>PD</math></td><td>An object of the class ProbDistUnivKernel.</td></tr></table>	$PD$	An object of the class ProbDistUnivKernel.
$PD$	An object of the class ProbDistUnivKernel.		
<b>Output Arguments</b>	<table><tr><td><math>Y</math></td><td>The value of the interquartile range for the ProbDistUnivKernel object <math>PD</math>.</td></tr></table>	$Y$	The value of the interquartile range for the ProbDistUnivKernel object $PD$ .
$Y$	The value of the interquartile range for the ProbDistUnivKernel object $PD$ .		
<b>See Also</b>	<a href="#">iqr</a>   <a href="#">ProbDistUnivKernel.icdf</a>		

# ProbDistUnivParam.iqr

---

<b>Purpose</b>	Return interquartile range (IQR) for ProbDistUnivParam object	
<b>Syntax</b>	$Y = \text{iqr}(PD)$	
<b>Description</b>	$Y = \text{iqr}(PD)$ returns $Y$ , the interquartile range for the ProbDistUnivParam object $PD$ . The interquartile range is the distance between the 75th and 25th percentiles.	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivParam.
<b>Output Arguments</b>	$Y$	The value of the interquartile range for the ProbDistUnivParam object $PD$ .
<b>See Also</b>	<a href="#">iqr</a>   <a href="#">ProbDistUnivParam.icdf</a>	

<b>Purpose</b>	Interquartile range of probability distribution object
<b>Syntax</b>	<code>r = iqr(pd)</code>
<b>Description</b>	<code>r = iqr(pd)</code> returns the interquartile range <code>r</code> of the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<p><b>pd - Probability distribution</b> probability distribution object</p> <p>Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code>. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using <code>fitdist</code> or the Distribution Fitting app.</p>
<b>Output Arguments</b>	<p><b>r - Interquartile range</b> scalar value</p> <p>Interquartile range of the probability distribution, returned as a scalar value. The value of <code>r</code> is the difference between the values of the 75th and 25th percentile of the probability distribution.</p>
<b>Examples</b>	<p><b>Interquartile Range of a Fitted Distribution</b></p> <p>Load the sample data. Create a vector containing the first column of students' exam grade data.</p> <pre>load examgrades; x = grades(:,1);</pre> <p>Create a normal distribution object by fitting it to the data.</p> <pre>pd = fitdist(x,'Normal')</pre> <pre>pd =  NormalDistribution</pre>

# prob.TruncatableDistribution.iqr

---

```
Normal distribution
  mu = 75.0083 [73.4321, 76.5846]
  sigma = 8.7202 [7.7391, 9.98843]
```

Compute the interquartile range of the fitted distribution.

```
r = iqr(pd)
```

```
r =
```

```
11.7634
```

The returned result indicates that the difference between the 75th and 25th percentile of the students' grades is 11.7634.

Use `icdf` to determine the 75th and 25th percentiles of the students' grades.

```
y = icdf(pd, [0.25, 0.75])
```

```
y =
```

```
69.1266 80.8900
```

Calculate the difference between the 75th and 25th percentiles. This yields the same result as `iqr`.

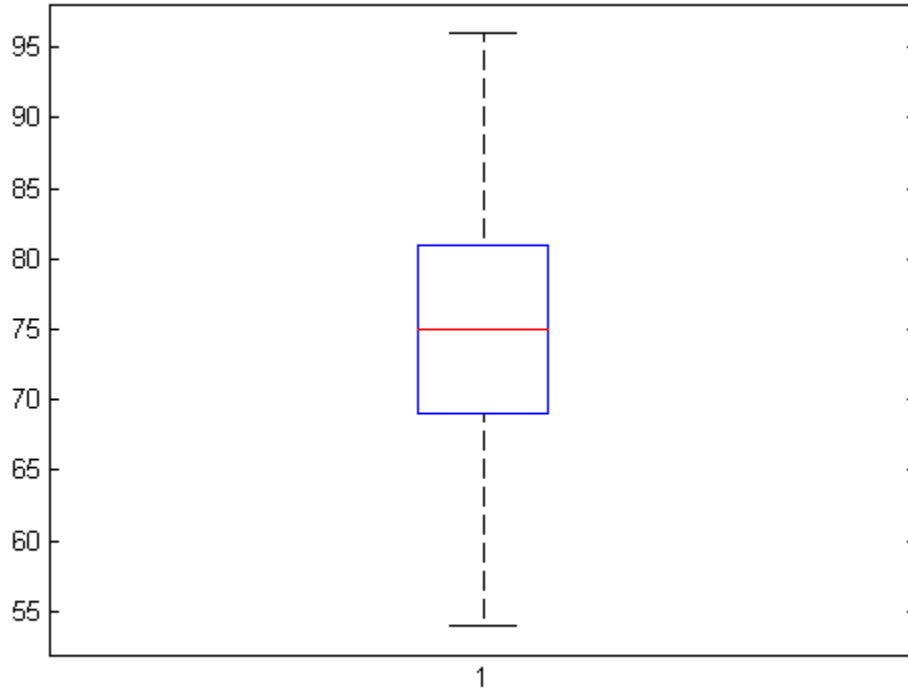
```
y(2) - y(1)
```

```
ans =
```

```
11.7634
```

Use `boxplot` to visualize the interquartile range.

```
boxplot(x)
```



The top line of the box shows the 75th percentile, and the bottom line shows the 25th percentile. The center line shows the median, which is the 50th percentile.

## See Also

`boxplot` | `makedist` | `fitdist` | `dfittool`

# classregtree.isbranch

---

**Purpose** Test node for branch

**Syntax** `ib = isbranch(t)`  
`ib = isbranch(t,nodes)`

**Description** `ib = isbranch(t)` returns an  $n$ -element logical vector `ib` that is true for each branch node and false for each leaf node.

`ib = isbranch(t,nodes)` takes a vector `nodes` of node numbers and returns a vector of logical values for the specified nodes.

**Examples** Create a classification tree for Fisher's iris data:

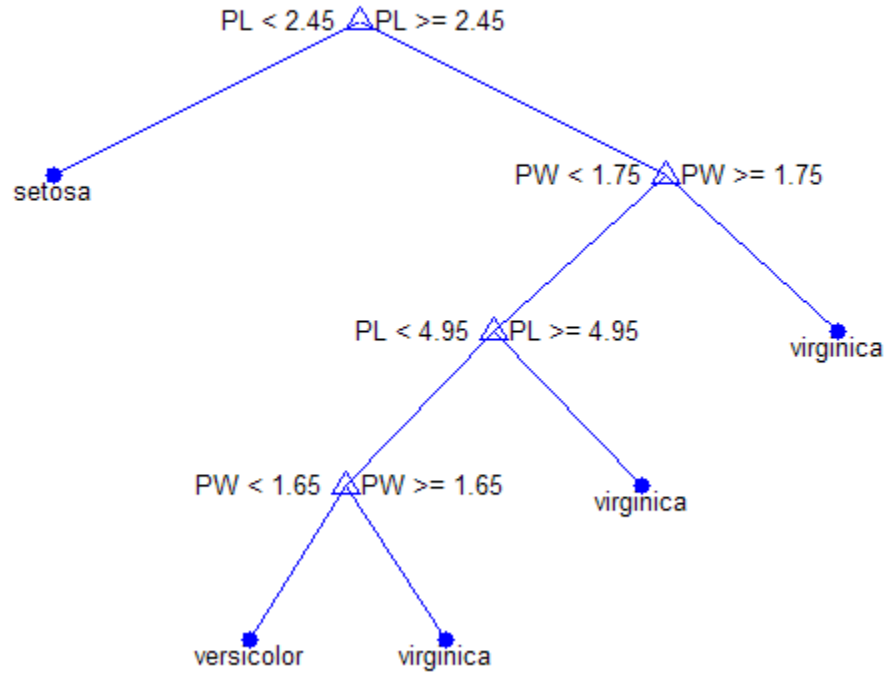
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})

t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

Click to display:  Magnification:  Pruning level:



```
ib = isbranch(t)
```

```
ib =
    1
    0
    1
    1
    0
    1
    0
    0
```

# classregtree.isbranch

---

0

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `cutvar` | `numnodes`



**Purpose** True for empty categorical array

**Syntax** TF = isEmpty(A)

**Description** TF = isEmpty(A) returns true (1) if A is an empty categorical array and false (0) otherwise. An empty array has no elements, that is numel(A)==0.

**See Also** numel | size

# dataset.isempty

---

**Purpose** True for empty dataset array

**Syntax** `tf = isempty(A)`

**Description** `tf = isempty(A)` returns true (1) if A is an empty dataset and false (0) otherwise. An empty array has no elements, that is `prod(size(A))==0`.

**See Also** `size`

**Purpose** True if categorical arrays are equal

**Syntax** TF = isequal(A,B)  
TF = isequal(A,B,C,...)

**Description** TF = isequal(A,B) is true (1) if the categorical arrays A and B are the same class, have the same size and the same sets of levels, and contain the same values, and false (0) otherwise.

TF = isequal(A,B,C,...) is true (1) if all the input arguments are equal.

Elements with undefined levels are not considered equal to each other.

**See Also** getlabels

# categorical.islevel

---

**Purpose** Test for levels

**Syntax** `I = islevel(levels,A)`

**Description** `I = islevel(levels,A)` returns a logical array `I` the same size as the string, cell array of strings, or 2-D character matrix `levels`. `I` is true (1) where the corresponding element of `levels` is the label of a level in the categorical array `A`, even if the level contains no elements. `I` is false (0) otherwise.

**Examples** Display age levels in the data in `hospital.mat`, before and after dropping occupied levels:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)', '%d'), {'s'});
disp(labels')
'0s' '10s' '20s' '30s' '40s' '50s' '60s' '70s' '80s' '90s'

AgeGroup = ordinal(hospital.Age, labels, [], edges);
I = islevel(labels, AgeGroup);
disp(I')
1 1 1 1 1 1 1 1 1 1

AgeGroup = droplevels(AgeGroup);
I = islevel(labels, AgeGroup);
disp(I')
0 0 1 1 1 1 0 0 0 0
```

**See Also** `ismember` | `isundefined`

**Purpose** True for elements of categorical array in set

---

**Note** In a future release, the behavior of `categorical.ismember` will change to be consistent with the MATLAB function `ismember`. This behavior change is optional in R2012a. For a demonstration of using the 'R2012a' flag to preview the future behavior, or the 'legacy' flag to preserve the current behavior in your existing code, see the documentation for `ismember`.

---

**Syntax**

```
TF = ismember(A,B)
[TF,LOCB] = ismember(A,B)
[...] = ismember(A,B,'rows')
[...] = intersect(...,'R2012a')
[...] = intersect(...,'legacy')
```

**Description** `TF = ismember(A,B)` for categorical arrays `A` and `B`, returns a logical array the same size as `A`, containing true (1) where the elements of `A` are in `B`, and false (0) otherwise. `B` can also be a cell array of strings or two-dimensional character array containing level labels.

`[TF,LOCB] = ismember(A,B)` also returns an index array `LOCB` containing the highest absolute index in `B` for each element in `A` which is a member of `B`, and 0 if there is no such index.

`[...] = ismember(A,B,'rows')`, for categorical matrices `A` and `B` with the same number of columns, returns a logical vector containing true (1) where the rows of `A` are also rows of `B`, and false (0) otherwise. The optional output `LOCB` is an index vector containing the highest absolute index in `B` for each row in `A` which is a member of `B` and 0 if there is no such index.

`[...] = intersect(...,'R2012a')` adopts the future behavior of `ismember`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or 'rows'.

# categorical.ismember

---

[...] = intersect(...,'legacy') preserves the current behavior of ismember. You can specify the flag as the final argument with any previous syntax that accepts A, B, or 'rows'.

## See Also

intersect | islevel | setdiff | setxor | union | unique

## Purpose

Dataset array elements that are members of set

## Syntax

```
LiA = ismember(A,B)  
LiA = ismember(A,B,vars)  
[LiA,LocB] = ismember( ___ )
```

## Description

`LiA = ismember(A,B)` for dataset arrays `A` and `B` returns a vector of logical values the same length as `A`. The output vector, `LiA`, has value 1 (true) in the elements that correspond to observations in `A` that are also present in `B`, and 0 (false) otherwise.

`LiA = ismember(A,B,vars)` returns a vector of logical values the same length as `A`. The output vector, `LiA`, has value 1 (true) in the elements that correspond to observations in `A` that are also present in `B` for the variables specified in `vars` only, and 0 (false) otherwise.

`[LiA,LocB] = ismember( ___ )` also returns a vector the same length as `A` containing the index to the first observation in `B` that corresponds to each observation in `A`, or 0 if there is no such observation. You can use any of the previous input arguments.

## Input Arguments

### A

Query dataset array, containing the observations to be found in `B`.

### B

Set dataset array. When an observation in `A` is found in `B`, for all variables or only those variables specified in `vars`, the corresponding element of `LiA` is 1.

### vars

Cell array of strings containing variable names or a vector of integers containing variable column numbers, indicating which variables to match observations on in `A` and `B`.

# dataset.ismember

---

## Output Arguments

### LiA

Vector of logical values the same length as A. LiA has value 1 (true) when the corresponding observation in A is present in B. Otherwise, LiA has value 0 (false).

If you specify `vars`, LiA has value 1 when the corresponding observation in A is present in B for the variables in `vars` only.

### LocB

Vector the same length as A containing the index to the first observation in B that corresponds to each observation in A, for all variables or only those variables specified in `vars`.

## Examples

### Find Observations That Are Members of a Dataset Array

Load sample data.

```
load('hospital')
B = hospital(1:50,1:5);
```

This set dataset array, B, has 50 observations on 5 variables.

Specify a query dataset array.

```
rng('default')
rIx = randsample(100,10);
A = hospital(rIx,1:5)
```

A =

	LastName	Sex	Age	Weight	Smoker
YLN-495	'COLEMAN'	Male	39	188	false
LQW-768	'TAYLOR'	Female	31	132	false
DGC-290	'BUTLER'	Male	38	184	true
DAU-529	'REED'	Male	50	186	true
REV-997	'ALEXANDER'	Male	25	171	true
QEQ-082	'COX'	Female	28	111	false
AGR-528	'SIMMONS'	Male	45	181	false



PUE-347	'YOUNG'	Female	25	114	false
HVR-372	'RUSSELL'	Male	44	188	true
XUE-826	'JACKSON'	Male	25	174	false

Check which observations in A are present in B.

```
LiA = ismember(A,B)
```

```
LiA =
```

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

Display the observations in A that are present in B.

```
A(LiA,:)
```

```
ans =
```

	LastName	Sex	Age	Weight	Smoker
LQW-768	'TAYLOR'	Female	31	132	false
PUE-347	'YOUNG'	Female	25	114	false
XUE-826	'JACKSON'	Male	25	174	false

Find the location of the observations in B.

```
[~,LocB] = ismember(A,B)
```

```
LocB =
```

# dataset.ismember

---

```
0
10
0
0
0
0
0
0
28
0
13
```

Display the observations in B that match observations in A.

```
B(LocB(LocB>0), :)
```

```
ans =
```

	LastName	Sex	Age	Weight	Smoker
LQW-768	'TAYLOR'	Female	31	132	false
PUE-347	'YOUNG'	Female	25	114	false
XUE-826	'JACKSON'	Male	25	174	false

## See Also

`dataset` | `intersect` | `setdiff` | `setxor` | `sortrows` | `union` | `unique` |

## Concepts

- “Dataset Arrays” on page 2-135

**Purpose** Find dataset array elements with missing values

**Syntax**

```
I = ismissing(ds)  
I = ismissing(ds,Name,Value)
```

**Description** `I = ismissing(ds)` returns a logical array that indicates which elements in the dataset array, `ds`, contain a missing value. By default, `ismissing` recognizes NaN as a missing value in numeric variables, '' as a missing value in string variables, and <undefined> as a missing value in categorical arrays.

- `ds2 = ds(~any(I,2),:)` creates a new dataset array containing only the complete observations in `ds`.
- `ds2 = ds(:,~any(I,1))` creates a new dataset array containing only the variables from `ds` with no missing values.

`I = ismissing(ds,Name,Value)` returns missing value indices with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**ds**  
dataset array

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### 'NumericTreatAsMissing'

# dataset.ismissing

---

Vector of numeric values to treat as missing value indicators in floating point `ds` variables. `ismissing` always treats a NaN value as a missing value.

## 'StringTreatAsMissing'

String or cell array of strings to treat as missing value indicators in string `ds` variables. `ismissing` always treats the empty string '' as a missing value.

## Output Arguments

**I**

Logical array indicating which elements in `ds` contain a missing value. `I` is the same size as `ds`, with value 1 for elements that contain a missing value.

## See Also

`dataset` | `isempty` | `isnanisundefined` | `replaceWithMissing` |

## Related Examples

- “Clean Messy and Missing Data” on page 2-117

## Concepts

- “Dataset Arrays” on page 2-135

<b>Purpose</b>	True if categorical array is scalar
<b>Syntax</b>	TF = isscalar(A)
<b>Description</b>	TF = isscalar(A) returns true (1) if the categorical array A is a 1-by-1 matrix, and false (0) otherwise.
<b>See Also</b>	isempty   isvector   size

# categorical.isundefined

---

**Purpose** Test for undefined elements

**Syntax** `I = isundefined(A)`

**Description** `I = isundefined(A)` returns a logical array `I` the same size as the categorical array `A`. `I` is true (1) where the corresponding element of `A` is not assigned to any level. `I` is false (0) where the corresponding element of `A` is assigned to a level.

**Examples** Create and display undefined levels in an ordinal array:

```
A = ordinal([1 2 3 2 1],{'lo','med','hi'})
```

```
A =  
    lo      med      hi      med      lo
```

```
A = droplevels(A,{'med','hi'})
```

```
Warning: OLDLEVELS contains categorical levels that  
were present in A, caused some array elements to  
have undefined levels.
```

```
A =  
    lo <undefined> <undefined> <undefined> lo
```

```
I = isundefined(A)
```

```
I =  
    0     1     1     1     0
```

**See Also** `islevel` | `ismember`

**Purpose** Test handle validity

**Syntax** `tf = isvalid(h)`

**Description** `tf = isvalid(h)` performs an element-wise check for validity on the handle elements of `h`. The result is a logical array of the same dimensions as `h`, where each element is the element-wise validity result.

A handle is invalid if it has been deleted or if it is an element of a handle array and has not yet been initialized.

**See Also** `delete` | `grandstream`

# categorical.isvector

---

**Purpose** True if categorical array is vector

**Syntax** TF = isvector(A)

**Description** TF = isvector(A) returns true (1) if the categorical array A is a 1-by-n or n-by-1 vector, where n >= 0, and false (0) otherwise.

**See Also** isempty | isscalar | size



## gmdistribution.Iters property

---

**Purpose** Number of iterations

**Description** The number of iterations of the algorithm.

---

**Note** This property applies only to gmdistribution objects constructed with `fit`.

---

# iwishrnd

---

**Purpose** Inverse Wishart random numbers

**Syntax**  
`W = wishrnd(Tau,df)`  
`W = wishrnd(Tau,df,DI)`  
`[W,DI] = wishrnd(Tau,df)`

**Description** `W = wishrnd(Tau,df)` generates a random matrix `W` from the inverse Wishart distribution with parameters `Tau` and `df`. The inverse of `W` has the Wishart distribution with covariance matrix `Sigma = inv(Tau)` and with `df` degrees of freedom. `Tau` is a symmetric and positive definite matrix.

`W = wishrnd(Tau,df,DI)` expects `DI` to be the transpose of the inverse of the Cholesky factor of `Tau`, so that  $DI' * DI = inv(Tau)$ , where `inv` is the MATLAB inverse function. `DI` is lower-triangular and the same size as `Tau`. If you call `wishrnd` multiple times using the same value of `Tau`, it is more efficient to supply `DI` instead of computing it each time.

`[W,DI] = wishrnd(Tau,df)` returns `DI` so you can use it as an input in future calls to `wishrnd`.

Note that different sources use different parametrizations for the inverse Wishart distribution. This function defines the parameter `tau` so that the mean of the output matrix is  $Tau / (df - d - 1)$  where `d` is the dimension of `Tau`.

**See Also** `wishrnd`

**How To** • “Inverse Wishart Distribution” on page B-64

**Purpose**

Jackknife sampling

**Syntax**

```
jackstat = jackknife(jackfun,X)
jackstat = jackknife(jackfun,X,Y,...)
jackstat = jackknife(jackfun,...,'Options',option)
```

**Description**

`jackstat = jackknife(jackfun,X)` draws jackknife data samples from the  $n$ -by- $p$  data array `X`, computes statistics on each sample using the function `jackfun`, and returns the results in the matrix `jackstat`. `jackknife` regards each row of `X` as one data sample, so there are  $n$  data samples. Each of the  $n$  rows of `jackstat` contains the results of applying `jackfun` to one jackknife sample. `jackfun` is a function handle specified with `@`. Row  $i$  of `jackstat` contains the results for the sample consisting of `X` with the  $i$ th row omitted:

```
s = x;
s(i,:) = [];
jackstat(i,:) = jackfun(s);
```

If `jackfun` returns a matrix or array, then this output is converted to a row vector for storage in `jackstat`. If `X` is a row vector, it is converted to a column vector.

`jackstat = jackknife(jackfun,X,Y,...)` accepts additional arguments to be supplied as inputs to `jackfun`. They may be scalars, column vectors, or matrices. `jackknife` creates each jackknife sample by sampling with replacement from the rows of the non-scalar data arguments (these must have the same number of rows). Scalar data are passed to `jackfun` unchanged. Non-scalar arguments must have the same number of rows, and each jackknife sample omits the same row from these arguments.

`jackstat = jackknife(jackfun,...,'Options',option)` provides an option to perform jackknife iterations in parallel, if the Parallel Computing Toolbox is available. Set `'Options'` as a structure you create with `statset`. `jackknife` uses the following field in the structure:

'UseParallel' If true and if a parpool of the Parallel Computing Toolbox is open, use multiple processors to compute jackknife iterations. If the Parallel Computing Toolbox is not installed, or a parpool is not open, computation occurs in serial mode. Default is false, or serial computation.

## Examples

Estimate the bias of the MLE variance estimator of random samples taken from the vector  $y$  using `jackknife`. The bias has a known formula in this problem, so you can compare the `jackknife` value to this formula.

```
sigma = 5;
y = normrnd(0,sigma,100,1);
m = jackknife(@var, y, 1);
n = length(y);
bias = -sigma^2 / n % known bias formula
jbias = (n - 1)*(mean(m)-var(y,1)) % jackknife bias estimate

bias =
    -0.2500

jbias =
    -0.3378
```

## See Also

`bootstrp` | `random` | `randsample` | `hist` | `ksdensity`

## Tutorials

- “Jackknife” on page 3-19

**Purpose**

Jarque-Bera test

**Syntax**

```
h = jbtest(x)
h = jbtest(x,alpha)
h = jbtest(x,alpha,mctol)
[h,p] = jbtest( ___ )
[h,p,jbstat,critval] = jbtest( ___ )
```

**Description**

`h = jbtest(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a normal distribution with an unknown mean and variance, using the Jarque-Bera test. The alternative hypothesis is that it does not come from such a distribution. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = jbtest(x,alpha)` returns a test decision for the null hypothesis at the significance level specified by `alpha`.

`h = jbtest(x,alpha,mctol)` returns a test decision based on a  $p$ -value computed using a Monte Carlo simulation with a maximum Monte Carlo standard error less than or equal to `mctol`.

`[h,p] = jbtest( ___ )` also returns the  $p$ -value `p` of the hypothesis test, using any of the input arguments from the previous syntaxes.

`[h,p,jbstat,critval] = jbtest( ___ )` also returns the test statistic `jbstat` and the critical value `critval` for the test.

**Input Arguments****`x` - Sample data**

vector

Sample data for the hypothesis test, specified as a vector. `jbtest` treats NaN values in `x` as missing values and ignores them.

## Data Types

single | double

## alpha - Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as a scalar value in the range (0,1). If alpha is in the range [0.001,0.50], and if the sample size is less than or equal to 2000, `jbtest` looks up the critical value for the test in a table of precomputed values. To conduct the test at a significance level outside of these specifications, use `mctol`.

**Example:** 0.01

## Data Types

single | double

## mctol - Maximum Monte Carlo standard error

nonnegative scalar value

Maximum Monte Carlo standard error for the  $p$ -value,  $p$ , specified as a nonnegative scalar value. If you specify a value for `mctol`, `jbtest` computes a Monte Carlo approximation for  $p$  directly, rather than interpolating into a table of precomputed values. `jbtest` chooses the number of Monte Carlo replications large enough to make the Monte Carlo standard error for  $p$  less than `mctol`.

If you specify a value for `mctol`, you must also specify a value for alpha. You can specify alpha as `[]` to use the default value of 0.05.

**Example:** 0.0001

## Data Types

single | double

## Output Arguments

## h - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the  $\alpha$  significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the  $\alpha$  significance level.

**p - p-value**

scalar value in the range (0,1)

$p$ -value of the test, returned as a scalar value in the range (0,1).  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

`jbstest` warns when  $p$  is not found within the tabulated range of [0.001,0.50], and returns either the smallest or largest tabulated value. In this case, you can use `mctol` to compute a more accurate  $p$ -value.

**jbstat - Test statistic**

nonnegative scalar value

Test statistic for the Jarque-Bera test, returned as a nonnegative scalar value.

**critval - Critical value**

nonnegative scalar value

Critical value for the Jarque-Bera test at the  $\alpha$  significance level, returned as a nonnegative scalar value. If `alpha` is in the range [0.001,0.50], and if the sample size is less than or equal to 2000, `jbstest` looks up the critical value for the test in a table of precomputed values. If you use `mctol`, `jbstest` determines the critical value of the test using a Monte Carlo simulation. The null hypothesis is rejected when `jbstat > critval`.

## Examples

### Test for a Normal Distribution

Load the data set.

```
load carbig;
```

Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars.

```
h = jbtest(MPG)
```

```
h =
```

```
1
```

The returned value of  $h = 1$  indicates that `jbtest` rejects the null hypothesis at the default 5% significance level.

## Test the Hypothesis at a Different Significance Level

Load the data set.

```
load carbig;
```

Test the null hypothesis that car mileage in miles per gallon (MPG) follows a normal distribution across different makes of cars at the 1% significance level.

```
[h,p] = jbtest(MPG,0.01)
```

```
h =
```

```
1
```

```
p =
```

```
0.0022
```

The returned value of  $h = 1$ , and the returned  $p$ -value less than  $\alpha = 0.01$  indicate that `jbtest` rejects the null hypothesis.

## Test for a Normal Distribution Using Monte Carlo Simulation

Load the data set.



```
load carbig;
```

Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars. Use a Monte Carlo simulation to obtain an exact  $p$ -value.

```
[h,p,jbstat,critval] = jbstest(MPG,[],0.0001)
```

```
h =
```

```
    1
```

```
p =
```

```
    0.0022
```

```
jbstat =
```

```
    18.2275
```

```
critval =
```

```
    5.8461
```

The returned value of  $h = 1$  indicates that `jbstest` rejects the null hypothesis at the default 5% significance level. Additionally, the test statistic, `jbstat`, is larger than the critical value, `critval`, which indicates rejection of the null hypothesis.

## Definitions

### Jarque-Bera Test

The Jarque-Bera test is a two-sided goodness-of-fit test suitable when a fully specified null distribution is unknown and its parameters must be estimated.

The test is specifically designed for alternatives in the Pearson system of distributions. The test statistic is

$$JB = \frac{n}{6} \left( s^2 + \frac{(k-3)^2}{4} \right),$$

where  $n$  is the sample size,  $s$  is the sample skewness, and  $k$  is the sample kurtosis. For large sample sizes, the test statistic has a chi-square distribution with two degrees of freedom.

### **Monte Carlo Standard Error**

The Monte Carlo standard error is the error due to simulating the  $p$ -value.

The Monte Carlo standard error is calculated as

$$SE = \sqrt{\frac{(\hat{p})(1-\hat{p})}{\text{mcreps}}},$$

where  $\hat{p}$  is the estimated  $p$ -value of the hypothesis test, and  $\text{mcreps}$  is the number of Monte Carlo replications performed. `jbtest` chooses the number of Monte Carlo replications, `mcreps`, large enough to make the Monte Carlo standard error for  $\hat{p}$  less than the value specified for `mctol`.

### **Algorithm**

Jarque-Bera tests often use the chi-square distribution to estimate critical values for large samples, deferring to the Lilliefors test (see `lillietest`) for small samples. `jbtest`, by contrast, uses a table of critical values computed using Monte Carlo simulation for sample sizes less than 2000 and significance levels from 0.001 to 0.50. Critical values for a test are computed by interpolating into the table, using the analytic chi-square approximation only when extrapolating for larger sample sizes.

## References

[1] Jarque, C. M., and A. K. Bera. “A Test for Normality of Observations and Regression Residuals.” *International Statistical Review*. Vol. 55, No. 2, 1987, pp. 163–172.

[2] Deb, P., and M. Sefton. “The Distribution of a Lagrange Multiplier Test of Normality.” *Economics Letters*. Vol. 51, 1996, pp. 123–130. This paper proposed a Monte Carlo simulation for determining the distribution of the test statistic. The results of this function are based on an independent Monte Carlo simulation, not the results in this paper.

## See Also

`adtest` | `kstest` | `lillietest`

## Concepts

- “Generating Data Using the Pearson System” on page 6-28

# johnsrnd

---

**Purpose** Johnson system random numbers

**Syntax**

```
r = johnsrnd(quantiles,m,n)
r = johnsrnd(quantiles)
[r,type] = johnsrnd(...)
[r,type,coefs] = johnsrnd(...)
```

**Description** `r = johnsrnd(quantiles,m,n)` returns an  $m$ -by- $n$  matrix of random numbers drawn from the distribution in the Johnson system that satisfies the quantile specification given by `quantiles`. `quantiles` is a four-element vector of quantiles for the desired distribution that correspond to the standard normal quantiles  $[-1.5 -0.5 0.5 1.5]$ . In other words, you specify a distribution from which to draw random values by designating quantiles that correspond to the cumulative probabilities  $[0.067 0.309 0.691 0.933]$ . `quantiles` may also be a 2-by-4 matrix whose first row contains four standard normal quantiles, and whose second row contains the corresponding quantiles of the desired distribution. The standard normal quantiles must be spaced evenly.

---

**Note** Because `r` is a random sample, its sample quantiles typically differ somewhat from the specified distribution quantiles.

---

`r = johnsrnd(quantiles)` returns a scalar value.

`r = johnsrnd(quantiles,m,n,...)` or `r = johnsrnd(quantiles,[m,n,...])` returns an  $m$ -by- $n$ -by-... array.

`[r,type] = johnsrnd(...)` returns the type of the specified distribution within the Johnson system. `type` is 'SN', 'SL', 'SB', or 'SU'. Set `m` and `n` to zero to identify the distribution type without generating any random values.

The four distribution types in the Johnson system correspond to the following transformations of a normal random variate:

- 'SN' — Identity transformation (normal distribution)

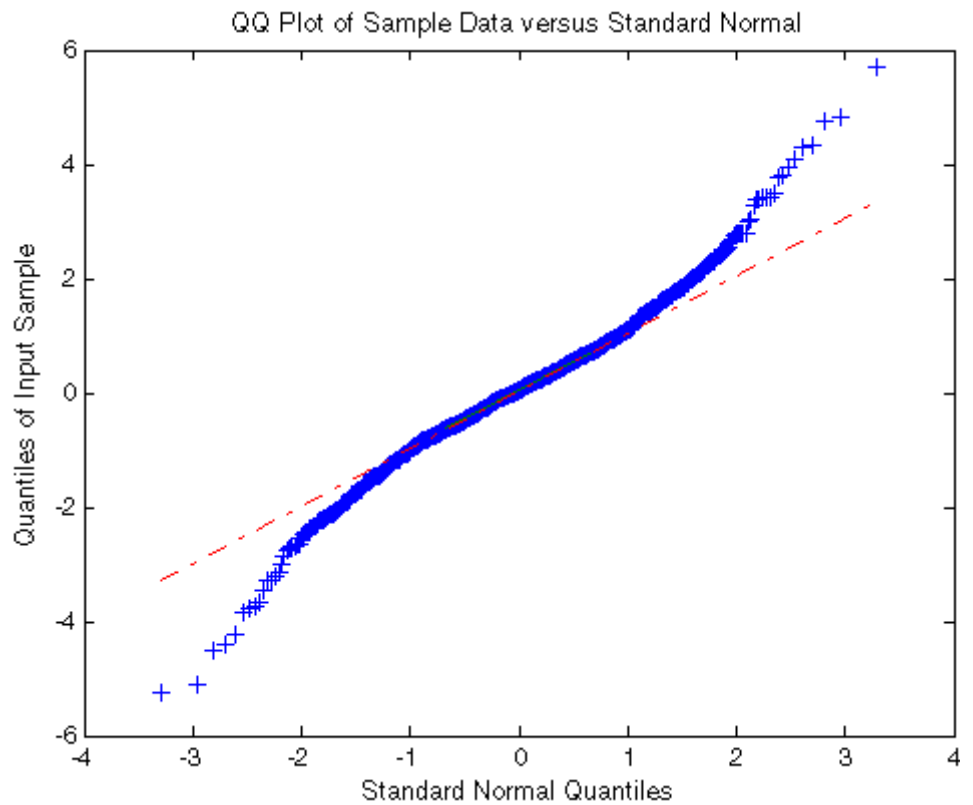
- 'SL' — Exponential transformation (lognormal distribution)
- 'SB' — Logistic transformation (bounded)
- 'SU' — Hyperbolic sine transformation (unbounded)

`[r,type,coefs] = johnsrnd(...)` returns coefficients `coefs` of the transformation that defines the distribution. `coefs` is `[gamma, eta, epsilon, lambda]`. If `z` is a standard normal random variable and `h` is one of the transformations defined above, `r = lambda*h((z-gamma)/eta)+epsilon` is a random variate from the distribution type corresponding to `h`.

## Examples

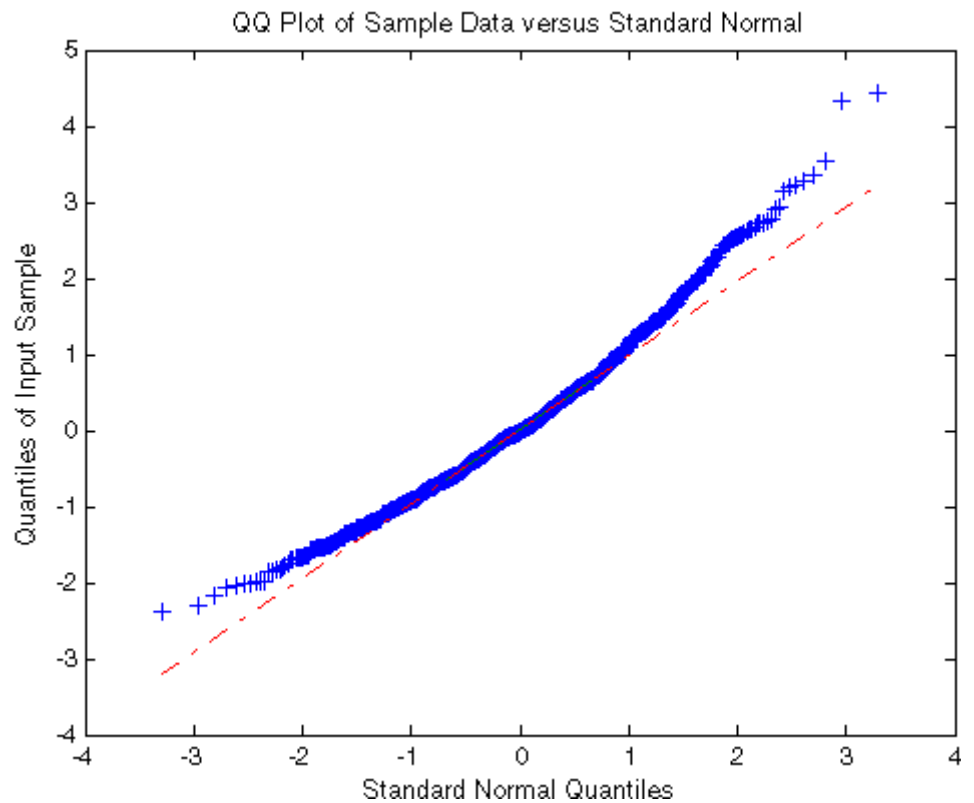
Generate random values with longer tails than a standard normal:

```
r = johnsrnd([-1.7 -.5 .5 1.7],1000,1);  
qqplot(r);
```



Generate random values skewed to the right:

```
r = johnsrnd([-1.3 -.5 .5 1.7],1000,1);  
qqplot(r);
```



Generate random values that match some sample data well in the right-hand tail:

```
load carbig;
qnorm = [.5 1 1.5 2];
q = quantile(Acceleration, normcdf(qnorm));
r = johnsrnd([qnorm;q],1000,1);
[q;q;quantile(r,normcdf(qnorm))]
ans =
    16.7000    18.2086    19.5376    21.7263
    16.8190    18.2474    19.4492    22.4156
```

Determine the distribution type and the coefficients:

```
[r,type,coefs] = johnsrnd([qnorm;q],0)
r =
    []
type =
    SU
coefs =
    1.0920    0.5829    18.4382    1.4494
```

## See Also

random | pearsrnd

## How To

- “Johnson System” on page B-66



**Purpose**

Merge observations

**Syntax**

```
C = join(A,B)
C = join(A,B,keys)
C = join(A,B,param1,va11,param2,va12,...)
[C,IB] = join(...)
C = join(A,B,'Type',TYPE,...)
C = join(A,B,'Type',TYPE,'MergeKeys',true,...)
[C,IA,IB] = join(A,B,'Type',TYPE,...)
```

**Description**

`C = join(A,B)` creates a dataset array `C` by merging observations from the two dataset arrays `A` and `B`. `join` performs the merge by first finding *key variables*, that is, pairs of dataset variables, one in `A` and one in `B`, that share the same name. Each observation in `B` must contain a unique combination of values in the key variables, and must contain all combinations of values that are present in the keys from `A`. `join` then uses these key variables to define a many-to-one correspondence between observations in `A` and those in `B`. `join` uses this correspondence to replicate the observations in `B` and combine them with the observations in `A` to create `C`.

`C = join(A,B,keys)` performs the merge using the variables specified by `keys` as the key variables in both `A` and `B`. `keys` is a positive integer, a vector of positive integers, a variable name, a cell array of variable names, or a logical vector.

`C` contains one observation for each observation in `A`. Variables in `C` include all of the variables from `A`, as well as one variable corresponding to each variable in `B` (except for the keys from `B`). If `A` and `B` contain variables with identical names, `join` adds the suffix `'_left'` and `'_right'` to the corresponding variables in `C`.

`C = join(A,B,param1,va11,param2,va12,...)` specifies optional parameter name/value pairs to control how the dataset variables in `A` and `B` are used in the merge. Parameters are:

- `'Keys'` — Specifies the variables to use as keys in both `A` and `B`.
- `'LeftKeys'` — Specifies the variables to use as keys in `A`.

## dataset.join

---

- `'RightKeys'` — Specifies the variables to use as keys in B.

You may provide either the `'Keys'` parameter, or both the `'LeftKeys'` and `'RightKeys'` parameters. The value for these parameters is a positive integer, a vector of positive integers, a variable name, a cell array containing variable names, or a logical vector. `'LeftKeys'` or `'RightKeys'` must both specify the same number of key variables, and `join` pairs the left and right keys in the order specified.

- `'LeftVars'` — Specifies which variables from A to include in C. By default, `join` includes all variables from A.
- `'RightVars'` — Specifies which variables from B to include in C. By default, `join` includes all variables from B except the key variables.

You can use `'LeftVars'` or `'RightVars'` to include or exclude key variables as well as data variables. The value for these parameters is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

`[C,IB] = join(...)` returns an index vector `IB`, where `join` constructs C by horizontally concatenating `A(:,LeftVars)` and `B(IB,RightVars)`. `join` can also perform more complicated inner and outer join operations that allow a many-to-many correspondence between A and B, and allow unmatched observations in either A or B.

`C = join(A,B,'Type',TYPE,...)` performs the join operation specified by `TYPE`. `TYPE` is one of `'inner'`, `'leftouter'`, `'rightouter'`, `'fullouter'`, or `'outer'` (which is a synonym for `'fullouter'`). For an inner join, C only contains observations corresponding to a combination of key values that occurred in both A and B. For a left (or right) outer join, C also contains observations corresponding to keys in A (or B) that did not match any in B (or A). Variables in C taken from A (or B) contain null values in those observations. A full outer join is equivalent to a left and right outer join. C contains variables corresponding to the key variables from both A and B, and `join` sorts the observations in C by the key values.

For inner and outer joins, C contains variables corresponding to the key variables from both A and B by default, as well as all the remaining variables. join sorts the observations in the result C by the key values.

`C = join(A,B, 'Type', TYPE, 'MergeKeys', true, ...)` includes a single variable in C for each key variable pair from A and B, rather than including two separate variables. For outer joins, join creates the single variable by merging the key values from A and B, taking values from A where a corresponding observation exists in A, and from B otherwise. Setting the 'MergeKeys' parameter to true overrides inclusion or exclusion of any key variables specified via the 'LeftVars' or 'RightVars' parameter. Setting the 'MergeKeys' parameter to false is equivalent to not passing in the 'MergeKeys' parameter.

`[C, IA, IB] = join(A,B, 'Type', TYPE, ...)` returns index vectors IA and IB indicating the correspondence between observations in C and those in A and B. For an inner join, join constructs C by horizontally concatenating `A(IA, LeftVars)` and `B(IB, RightVars)`. For an outer join, IA or IB may also contain zeros, indicating the observations in C that do not correspond to observations in A or B, respectively.

## Examples

Create a dataset array from Fisher's iris data:

```
load fisheriris
NumObs = size(meas,1);
NameObs = strcat({'Obs'},num2str((1:NumObs)', '%-d'));
iris = dataset({nominal(species), 'species'}, ...
               {meas, 'SL', 'SW', 'PL', 'PW'}, ...
               'ObsNames', NameObs);
```

Create a separate dataset array with the diploid chromosome counts for each species of iris:

```
snames = nominal({'setosa'; 'versicolor'; 'virginica'});
CC = dataset({snames, 'species'}, {[38;108;70], 'cc'})
CC =
    species      cc
```

# dataset.join

---

```
setosa      38
versicolor 108
virginica   70
```

---

Broadcast the data in CC to the rows of iris using the key variable species in each dataset:

```
iris2 = join(iris,CC);
iris2([1 2 51 52 101 102],:)
ans =
```

	species	SL	SW	PL	PW	cc
Obs1	setosa	5.1	3.5	1.4	0.2	38
Obs2	setosa	4.9	3	1.4	0.2	38
Obs51	versicolor	7	3.2	4.7	1.4	108
Obs52	versicolor	6.4	3.2	4.5	1.5	108
Obs101	virginica	6.3	3.3	6	2.5	70
Obs102	virginica	5.8	2.7	5.1	1.9	70

---

Create two datasets and join them using the 'MergeKeys' flag:

```
% Create two data sets that both contain the key variable
% 'Key1'. The two arrays contain observations with common
% values of Key1, but each array also contains observations
% with values of Key1 not present in the other.
a = dataset({'a' 'b' 'c' 'e' 'h'},[1 2 3 11 17]',...
    'VarNames',{'Key1' 'Var1'})
b = dataset({'a' 'b' 'd' 'e'},[4 5 6 7]',...
    'VarNames',{'Key1' 'Var2'})

% Combine a and b with an outer join, which matches up
% observations with common key values, but also retains
% observations whose key values don't have a match.
% Keep the key values as separate variables in the result.
couter = join(a,b,'key','Key1','Type','outer')
```

```
% Join a and b, merging the key values as a single variable  
% in the result.
```

```
coutermerge = join(a,b,'key','Key1','Type','outer',...  
  'MergeKeys',true)
```

```
% Join a and b, retaining only observations whose key  
% values match.
```

```
cinner = join(a,b,'key','Key1','Type','inner',...  
  'MergeKeys',true)
```

```
a =
```

Key1	Var1
'a'	1
'b'	2
'c'	3
'e'	11
'h'	17

```
b =
```

Key1	Var2
'a'	4
'b'	5
'd'	6
'e'	7

```
couter =
```

Key1_left	Var1	Key1_right	Var2
'a'	1	'a'	4
'b'	2	'b'	5
'c'	3	' '	NaN
' '	NaN	'd'	6
'e'	11	'e'	7

# dataset.join

---

```
'h'          17      ''          NaN
```

```
coutermerge =
```

Key1	Var1	Var2
'a'	1	4
'b'	2	5
'c'	3	NaN
'd'	NaN	6
'e'	11	7
'h'	17	NaN

```
cinner =
```

Key1	Var1	Var2
'a'	1	4
'b'	2	5
'e'	11	7

## See Also

```
sortrows
```

**Superclasses** NeighborSearcher

**Purpose** Nearest neighbors search using *kd*-tree

**Description** A KDTreeSearcher object represents *k*NN (*k*-nearest neighbor) search using a *kd*-tree. Search objects store information about the data used, the distance metric and parameters, and the maximal number of data points in each leaf node. You cannot create this object for sparse input data. The search performance for this object, compared with the ExhaustiveSearcher object, tends to be better for smaller dimensions (10 or fewer) and worse for larger dimensions. For more information on search objects, see “What Are Search Objects?” on page 16-17.

**Construction** `NS = KDTreeSearcher(X, 'Name', Value)` constructs a *kd*-tree based on *X* and saves the information in a KDTreeSearcher object where rows of *X* correspond to observations and columns correspond to variables. You can use this tree to find neighbors in *X* nearest to the query points. See the following section for optional name/value pairs.

`NS = createns(X, 'NSMethod', 'kdtree', 'Name', Value)` creates a *kd*-tree based on *X* using `createns` and saves the information in a KDTreeSearcher object where rows of *X* correspond to observations and columns correspond to variables. You can use this tree to find neighbors in *X* nearest to the query points. See the following section for optional name/value pairs.

### Name-Value Pair Arguments

KDTreeSearcher and createns accept one or more of the following optional name/value pairs as input:

#### ‘Distance’

A string specifying the default distance metric used when you call the `knnsearch` method.

- 'euclidean' — Euclidean distance (default).
- 'cityblock' — City block distance.

# KDTreeSearcher

---

- 'chebychev' — Chebychev distance (maximum coordinate difference).
- 'minkowski' — Minkowski distance.

For more information on these distance metrics, see “Distance Metrics” on page 16-9.

## **'p'**

A positive scalar indicating the exponent of the Minkowski distance. This parameter is only valid when `Distance` is 'minkowski'. Default is 2.

## **'BucketSize'**

A positive integer, indicating the maximum number of data points in each leaf node of the *kd*-tree. Default is 50.

## **Properties**

### **X**

A matrix used to create the object

### **Distance**

A string specifying a built-in distance metric that you provide when you create the object. This property is the default distance metric used when you call the `knsearch` method to find nearest neighbors for future query points.

### **DistParameter**

Specifies the additional parameter for the chosen distance metric. The value is:

- If 'Distance' is 'minkowski': A positive scalar indicating the exponent of the Minkowski distance.
- Otherwise: Empty.



## Methods

<code>knnsearch</code>	Find $k$ -nearest neighbors using KDTreeSearcher object
<code>rangesearch</code>	Find all neighbors within specified distance using object

## Examples

Create a KDTreeSearcher object using the constructor:

```
load fisheriris
x = meas(:,3:4);
kdtreeobj = KDTreeSearcher(x,'distance','minkowski')

kdtreeobj =

    KDTreeSearcher

Properties:
    BucketSize: 50
             X: [150x2 double]
    Distance: 'minkowski'
    DistParameter: 2
```

---

Create a KDTreeSearcher object using createns:

```
load fisheriris
x = meas(:,3:4);
kdtreeobj = createns(x,'NsMethod','kdtree',...
    'distance','minkowski')

kdtreeobj =

    KDTreeSearcher

Properties:
    BucketSize: 50
```

# KDTreeSearcher

---

```
X: [150x2 double]
Distance: 'minkowski'
DistParameter: 2
```

For more in-depth examples using the `knnsearch` method, see the method reference page or see “Classifying Query Data Using `knnsearch`” on page 16-18.

## References

[1] Friedman, J. H., Bentely, J., and Finkel, R. A. (1977). *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, ACM Transactions on Mathematical Software 3, 209.

## See Also

`createns` | `ExhaustiveSearcher` | `NeighborSearcher`

## How To

- “*k*-Nearest Neighbor Search and Radius Search” on page 16-12

# ProbDistKernel.Kernel property

---

**Purpose** Read-only string specifying name of kernel smoothing function for ProbDistKernel object

**Description** Kernel is a read-only property of the ProbDistKernel class. Kernel is a string specifying the name of the kernel smoothing function used to create a ProbDistKernel object.

**Values**

- 'normal'
- 'box'
- 'triangle'
- 'epanechnikov'

Use this information to view and compare the kernel smoothing function used to create distributions.

**See Also** [ksdensity](#)

# prob.KernelDistribution

---

**Superclasses** TruncatableDistribution

**Purpose** Kernel probability distribution object

**Description** `prob.KernelDistribution` is an object consisting of parameters, a model description, and sample data for a nonparametric kernel-smoothing distribution. Create a `prob.KernelDistribution` object using `fitdist` or `dfittool`.

**Construction** `pd = fitdist(x,'Kernel')` creates a probability distribution object by fitting a kernel-smoothing distribution to the data in `x`.

`pd = fitdist(x,'Kernel',Name,Value)` creates a probability distribution object with additional options specified by one or more name-value pair arguments. For example, you can change the kernel function or specify the kernel bandwidth.

## Input Arguments

### **x - Input data**

column vector

Input data to fit with a kernel-smoothing distribution, specified as a column vector of scalar values. `fitdist` ignores NaN values in `x`.

### **Data Types**

single | double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **Kernel - Kernel smoother type**

'normal' (default) | 'box' | 'triangle' | 'epanechnikov'

Kernel smoother type, specified as the comma-separated pair consisting of 'Kernel' and one of the following kernel smoothing function types:

- 'normal'
- 'box'
- 'triangle'
- 'epanechnikov'

### Support - Kernel density support

'unbounded' (default) | 'positive' | two-element vector

Kernel density support, specified as the comma-separated pair consisting of 'Support' and a string or two-element vector. The string must be one of the following.

'unbounded'	Density can extend over the whole real line.
'positive'	Density is restricted to positive values.

Alternatively, you can specify a two-element vector giving finite lower and upper limits for the support of the density.

### Data Types

single | double

### Width - Bandwidth of kernel smoothing window

scalar value

Bandwidth of the kernel smoothing window, specified as the comma-separated pair consisting of 'Width' and a scalar value. The default value used by `fitdist` is optimal for estimating normal densities, but you might want to choose a smaller value to reveal features such as multiple modes.

# prob.KernelDistribution

---

## Data Types

single | double

## Properties

### Kernel

Kernel function type, stored as a valid kernel function type name.

### BandWidth

Bandwidth of the kernel smoothing window, stored as a positive scalar value.

## Data Types

single | double

### DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

### InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

### IsTruncated

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

<code>mean</code>	Mean of probability distribution object
<code>negloglik</code>	Negative loglikelihood
<code>std</code>	Standard deviation of probability distribution object
<code>var</code>	Variance of probability distribution object

## Inherited Methods

<code>cdf</code>	Cumulative distribution function of probability distribution object
<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
<code>iqr</code>	Interquartile range of probability distribution object
<code>median</code>	Median of probability distribution object

# prob.KernelDistribution

---

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object

## Definitions Kernel Distribution

The kernel distribution is a nonparametric estimation of the probability density function (pdf) of a random variable.

The kernel distribution uses the following options.

Option	Description	Possible Values
Kernel	Kernel function type	normal, box, triangle, epanechnikov
BandWidth	Kernel smoothing parameter	BandWidth > 0

The kernel density estimator is

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right) ; \quad -\infty < x < \infty ,$$

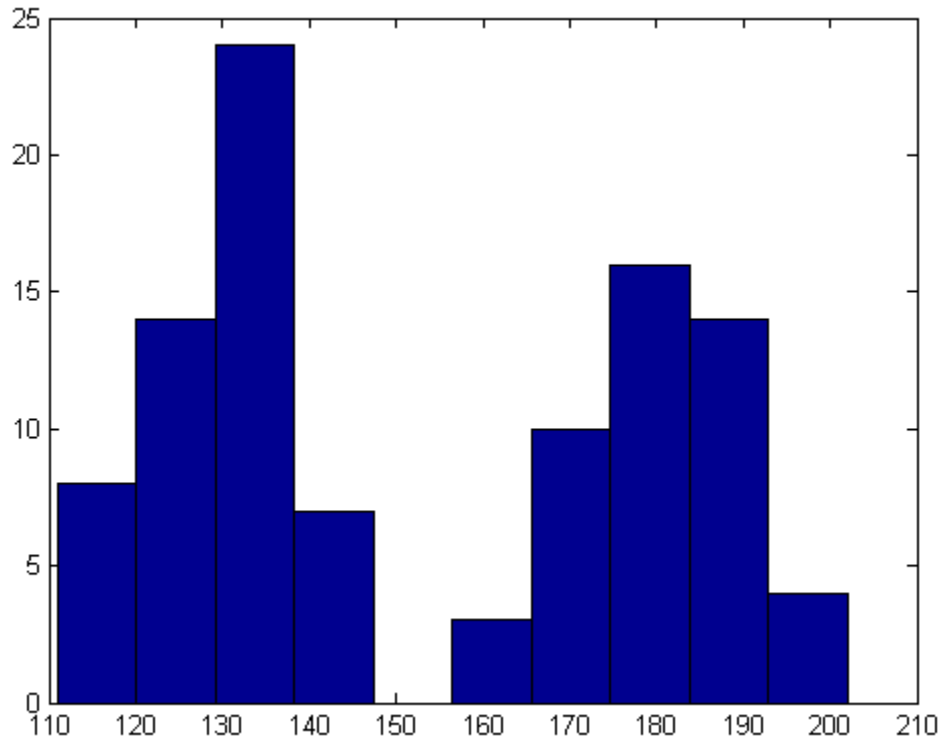
where  $n$  is the sample size,  $K(\square)$  is the kernel function, and  $h$  is the bandwidth.

## Examples Fit a Kernel Distribution Object to Data

Load the sample data. Visualize the patient weight data using a histogram.

```
load hospital;  
hist(hospital.Weight)
```





The histogram shows that the data has two modes, one for female patients and one for male patients.

Create a probability distribution object by fitting a kernel distribution to the patient weight data.

```
pd_kernel = fitdist(hospital.Weight, 'Kernel')
```

```
pd_kernel =
```

# prob.KernelDistribution

---

```
KernelDistribution

Kernel = normal
Bandwidth = 14.3792
Support = unbounded
```

For comparison, create another probability distribution object by fitting a normal distribution to the patient weight data.

```
pd_normal = fitdist(hospital.Weight, 'Normal')
```

```
pd_normal =
```

```
NormalDistribution

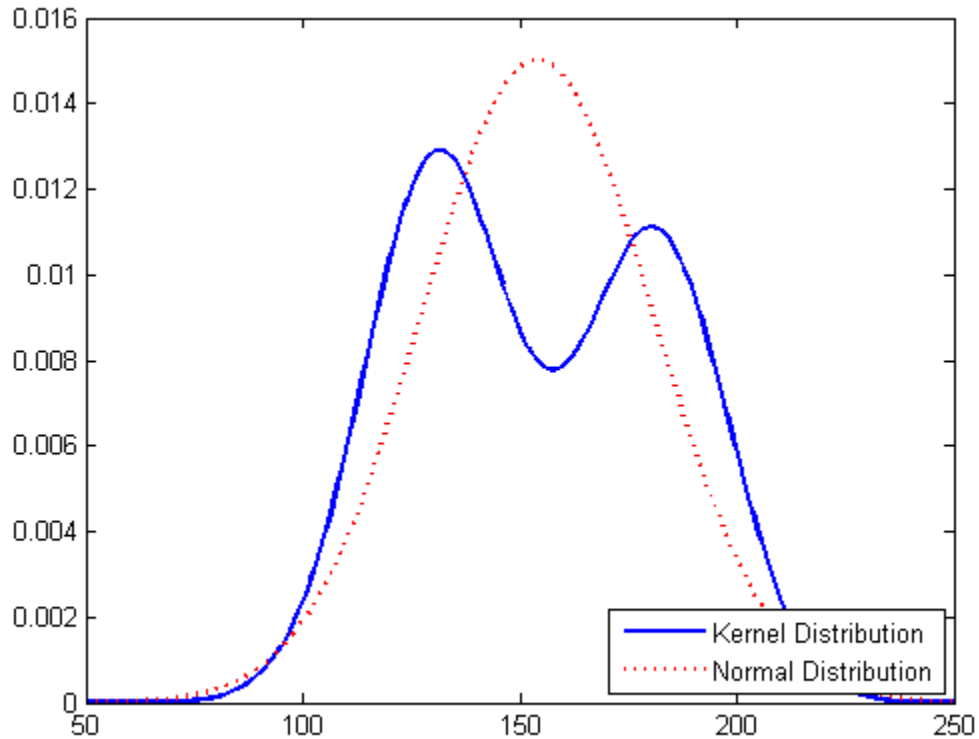
Normal distribution
mu = 154 [148.728, 159.272]
sigma = 26.5714 [23.3299, 30.8674]
```

Define the x values and compute the pdf of each distribution.

```
x = 50:1:250;
pdf_kernel = pdf(pd_kernel,x);
pdf_normal = pdf(pd_normal,x);
```

Plot the pdf of each distribution.

```
plot(x,pdf_kernel,'Color','b','LineWidth',2);
hold on;
plot(x,pdf_normal,'Color','r','LineStyle',':', 'LineWidth',2);
legend('Kernel Distribution','Normal Distribution','Location','SouthEast');
hold off;
```



Fitting a kernel distribution instead of a unimodal distribution such as the normal reveals the separate modes for the female and male patients.

## See Also

`fitdist` | `dfittool`

## Concepts

- “Kernel Distribution” on page B-67
- Class Attributes
- Property Attributes

# ClassificationPartitionedEnsemble.kfoldEdge

---

**Purpose** Classification edge for observations not used for training

**Syntax**  
E = kfoldEdge(obj)  
E = kfoldEdge(obj,Name,Value)

**Description** E = kfoldEdge(obj) returns classification edge (average classification margin) obtained by cross-validated classification ensemble obj. For every fold, this method computes classification edge for in-fold observations using an ensemble trained on out-of-fold observations.

E = kfoldEdge(obj,Name,Value) calculates edge with additional options specified by one or more Name,Value pair arguments. You can specify several name-value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

**Input Arguments**

**ens**  
Object of class ClassificationPartitionedEnsemble. Create ens with fitensemble along with one of the cross-validation options: 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'. Alternatively, create ens from a classification ensemble with crossval.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### **'folds'**

Indices of folds ranging from 1 to ens.KFold. Use only these folds for predictions.

**Default:** 1:ens.KFold

### **'mode'**

# ClassificationPartitionedEnsemble.kfoldEdge

---

String representing the meaning of the output `edge`:

- 'average' — `edge` is a scalar value, the average over all folds.
- 'individual' — `edge` is a vector of length `ens.KFold` with one element per fold.
- 'cumulative' — `edge` is a vector of length `min(ens.NTrainedPerFold)` in which element `J` is obtained by averaging values across all folds for weak learners `1:J` in each fold.

**Default:** 'average'

## Output Arguments

**E**

The average classification margin. `E` is a scalar or vector, depending on the setting of the `mode` name-value pair.

## Definitions

### Edge

The *edge* is the weighted mean value of the classification margin. The weights are the class probabilities in `obj.Prior`.

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `obj.X`.

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

# ClassificationPartitionedEnsemble.kfoldEdge

---

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## Examples

Compute the k-fold edge for an ensemble trained on the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
cvens = crossval(ens);
E = kfoldEdge(cvens)
```

```
E =
    3.2078
```

## See Also

[kfoldPredict](#) | [kfoldLoss](#) | [kfoldMargin](#) | [kfoldfun](#) | [crossval](#)

# ClassificationPartitionedModel.kfoldEdge

---

<b>Purpose</b>	Classification edge for observations not used for training
<b>Syntax</b>	<pre>E = kfoldEdge(obj) E = kfoldEdge(obj,Name,Value)</pre>
<b>Description</b>	<p><code>E = kfoldEdge(obj)</code> returns classification edge (average classification margin) obtained by cross-validated classification model <code>obj</code>. For every fold, this method computes classification edge for in-fold observations using an ensemble trained on out-of-fold observations.</p> <p><code>E = kfoldEdge(obj,Name,Value)</code> calculates edge with additional options specified by one or more <code>Name,Value</code> pair arguments. You can specify several name-value pair arguments in any order as <code>Name1,Value1, ...,NameN,ValueN</code>.</p>
<b>Input Arguments</b>	<p><b>obj</b> Object of class <code>ClassificationPartitionedModel</code>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1, ...,NameN,ValueN</code>.</p> <p><b>'folds'</b></p> <p>Indices of folds ranging from 1 to <code>obj.KFold</code>. Use only these folds for predictions.</p> <p><b>Default:</b> <code>1:obj.KFold</code></p> <p><b>'mode'</b></p> <p>String representing the meaning of the output edge:</p> <ul style="list-style-type: none"><li>'average' — edge is a scalar value, the average over all folds.</li></ul>

# ClassificationPartitionedModel.kfoldEdge

---

- 'individual' — edge is a vector of length `obj.KFold` with one element per fold.

**Default:** 'average'

## Output Arguments

**E**

The average classification margin. `E` is a scalar or vector, depending on the setting of the `mode` name-value pair.

## Definitions

### Edge

The *edge* is the weighted mean value of the classification *margin*. The weights are the class probabilities in `obj.Prior`. If you supply weights in the `weights` name-value pair, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average.

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `obj.X`. A high value of margin indicates a more reliable prediction than a low value.

### Score (discriminant analysis)

For discriminant analysis, the *score* of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 15-7.

### Score (tree)

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.



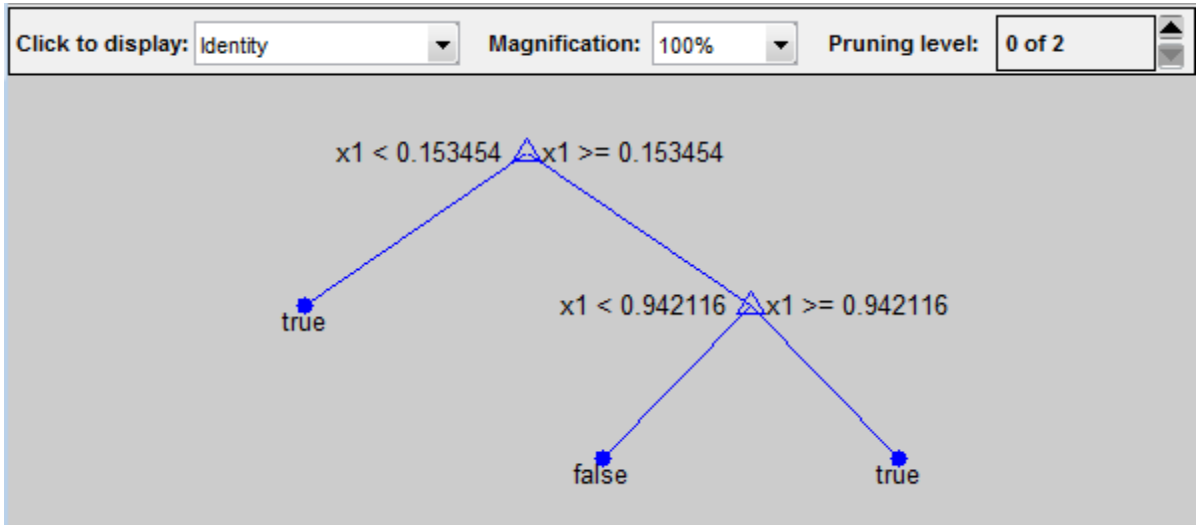
# ClassificationPartitionedModel.kfoldEdge

For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

1

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```

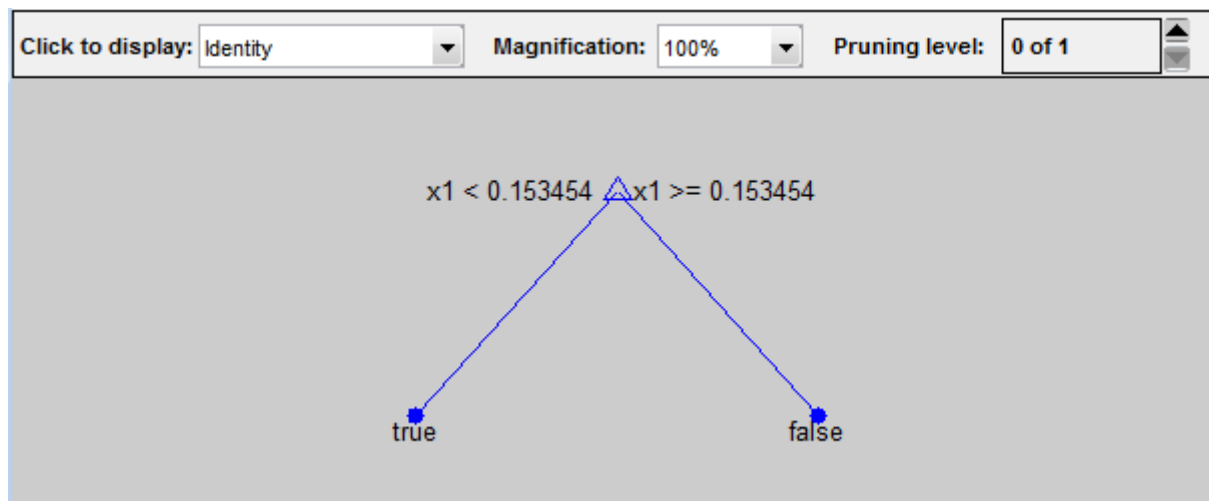


2

Prune the tree:

```
tree1 = prune(tree,'level',1);
view(tree1,'mode','graph')
```

# ClassificationPartitionedModel.kfoldEdge



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785
```

```
0.9059    0.0941    0.5469
0.9059    0.0941    0.9575
0.9059    0.0941    0.9649
```

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## Examples

Compute the k-fold edge for a model trained on the Fisher iris data:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
cvtree = crossval(tree);
E = kfoldEdge(cvtree)
```

```
E =
    0.8711
```

## See Also

```
kfoldPredict | kfoldMargin | kfoldLoss | kfoldfun
| crossval | ClassificationPartitionedEnsemble |
ClassificationPartitionedModel
```

# ClassificationPartitionedModel.kfoldfun

---

**Purpose** Cross validate function

**Syntax** `vals = kfoldfun(obj,fun)`

**Description** `vals = kfoldfun(obj,fun)` cross validates the function `fun` by applying `fun` to the data stored in the cross-validated model `obj`. You must pass `fun` as a function handle.

## Input Arguments

### **obj**

Object of class `ClassificationPartitionedModel` or `ClassificationPartitionedEnsemble`.

### **fun**

A function handle for a cross-validation function. `fun` has the syntax

```
testvals = fun(CMP,Xtrain,Ytrain,Wtrain,Xtest,Ytest,Wtest)
```

- `CMP` is a compact model stored in one element of the `obj.Trained` property.
- `Xtrain` is the training matrix of predictor values.
- `Ytrain` is the training array of response values.
- `Wtrain` are the training weights for observations.
- `Xtest` and `Ytest` are the test data, with associated weights `Wtest`.
- The returned value `testvals` must have the same size across all folds.

## Output Arguments

### **vals**

The arrays of `testvals` output, concatenated vertically over all folds. For example, if `testvals` from every fold is a numeric vector of length `N`, `kfoldfun` returns a `KFold-by-N` numeric matrix with one row per fold.

## Examples

Cross validate a classification tree, and obtain the classification error (see `kfoldLoss`):

```
load fisheriris
t = ClassificationTree.fit(meas,species);
rng(0,'twister') % for reproducibility
cv = crossval(t);
L = kfoldLoss(cv)
```

```
L =
    0.0467
```

Examine the result when the error of misclassifying a flower as 'versicolor' is 10, and any other error is 1:

- 1 Write a function file that gives a cost of 1 for misclassification, but 10 for misclassifying a flower as `versicolor`.

```
function averageCost = noversicolor(CMP,Xtrain,Ytrain,Wtrain,Xtest,Ytest,Wtest)

Ypredict = predict(CMP,Xtest);
misclassified = not(strcmp(Ypredict,Ytest)); % different result
classifiedAsVersicolor = strcmp(Ypredict,'versicolor'); % index of bad decisions
cost = sum(misclassified) + ...
    9*sum(misclassified & classifiedAsVersicolor); % total differences
averageCost = cost/numel(Ytest); % average error
```

- 2 Save the file as `noversicolor.m` on your MATLAB path.
- 3 Compute the mean misclassification error with the `noversicolor` cost:

```
mean(kfoldfun(cv,@noversicolor))

ans =
    0.1667
```

# ClassificationPartitionedModel.kfoldfun

---

## See Also

`ClassificationPartitionedModel` | `kfoldPredict` | `kfoldEdge` |  
`kfoldMargin` | `kfoldLoss` | `crossval`

# RegressionPartitionedModel.kfoldfun

---

## Purpose

Cross validate function

## Syntax

```
vals = kfoldfun(obj,fun)
```

## Description

`vals = kfoldfun(obj,fun)` cross validates the function `fun` by applying `fun` to the data stored in the cross-validated model `obj`. You must pass `fun` as a function handle.

## Input Arguments

### **obj**

Object of class `RegressionPartitionedModel` or `RegressionPartitionedEnsemble`. Create `obj` with `RegressionTree.fit` or `fitensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `obj` from a regression tree or regression ensemble with `crossval`.

### **fun**

A function handle for a cross-validation function. `fun` has the syntax

```
testvals = fun(CMP,Xtrain,Ytrain,Wtrain,Xtest,Ytest,Wtest)
```

- `CMP` is a compact model stored in one element of the `obj.Trained` property.
- `Xtrain` is the training matrix of predictor values.
- `Ytrain` is the training array of response values.
- `Wtrain` are the training weights for observations.
- `Xtest` and `Ytest` are the test data, with associated weights `Wtest`.
- The returned value `testvals` must have the same size across all folds.

# RegressionPartitionedModel.kfoldfun

---

## Output Arguments

### vals

The arrays of testvals output, concatenated vertically over all folds. For example, if testvals from every fold is a numeric vector of length N, kfoldfun returns a KFold-by-N numeric matrix with one row per fold.

## Examples

Cross validate a regression tree, and obtain the mean squared error (see kfoldLoss):

```
load imports-85
t = RegressionTree.fit(X(:,[4 5]),X(:,16),...
    'predictorname',{'length' 'width'},...
    'response', 'price');
cv = crossval(t);
L = kfoldLoss(cv)
```

```
L =
    1.5489e+007
```

Examine the result of simple averaging of responses instead of using predictions:

```
f = @(cmp,Xtrain,Ytrain,Wtrain,Xtest,Ytest,Wtest)...
    mean((Ytest-mean(Ytrain)).^2)
mean(kfoldfun(cv,f))

ans =
    6.3497e+007
```

## See Also

RegressionPartitionedEnsemble | kfoldPredict | kfoldLoss | RegressionPartitionedModel | crossval



# ClassificationPartitionedEnsemble.kfoldLoss

---

## Purpose

Classification loss for observations not used for training

## Syntax

```
L = kfoldLoss(ens)
L = kfoldLoss(ens,Name,Value)
```

## Description

`L = kfoldLoss(ens)` returns loss obtained by cross-validated classification model `ens`. For every fold, this method computes classification loss for in-fold observations using a model trained on out-of-fold observations.

`L = kfoldLoss(ens,Name,Value)` calculates loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### **ens**

Object of class `ClassificationPartitionedEnsemble`. Create `ens` with `fitensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `ens` from a classification ensemble with `crossval`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### **'folds'**

Indices of folds ranging from 1 to `ens.KFold`. Use only these folds for predictions.

**Default:** `1:ens.KFold`

### **'lossfun'**

# ClassificationPartitionedEnsemble.kfoldLoss

---

Function handle or string representing a loss function. Built-in loss functions:

- 'binodeviance' — See “Loss Functions” on page 21-1538
- 'classiferror' — Fraction of misclassified data
- 'exponential' — See “Loss Functions” on page 21-1538

You can write your own loss function in the syntax described in “Loss Functions” on page 21-1538.

**Default:** 'classiferror'

## **'mode'**

A string for determining the output of `kfoldLoss`:

- 'average' — `L` is a scalar, the loss averaged over all folds.
- 'individual' — `L` is a vector of length `ens.KFold`, where each entry is the loss for a fold.
- 'cumulative' — `L` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

**Default:** 'average'

## **Output Arguments**

**L**

Loss, by default the fraction of misclassified data. `L` can be a vector, and can mean different things, depending on the name-value pair settings.

## **Definitions**

### **Loss Functions**

The built-in loss functions are:

# ClassificationPartitionedEnsemble.kfoldLoss

---

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'classiferror' — Fraction of misclassified data, weighted by  $w$ .
- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

To write your own loss function, create a function file of the form

```
function loss = lossfun(C,S,W,COST)
```

- $N$  is the number of rows of `ens.X`.
- $K$  is the number of classes in `ens`, represented in `ens.ClassNames`.
- $C$  is an  $N$ -by- $K$  logical matrix, with one `true` per row for the true class. The index for each class is its position in `tree.ClassNames`.
- $S$  is an  $N$ -by- $K$  numeric matrix.  $S$  is a matrix of posterior probabilities for classes with one row per observation, similar to the `score` output from `predict`.
- $W$  is a numeric vector with  $N$  elements, the observation weights.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. The default 'classiferror' gives a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

# ClassificationPartitionedEnsemble.kfoldLoss

---

## Examples

Find the average cross-validated classification error for an ensemble model of the ionosphere data:

```
load ionosphere
ens = fitensemble(X,Y,'AdaBoostM1',100,'Tree');
cvens = crossval(ens);
L = kfoldLoss(cvens)
```

```
L =
    0.0826
```

## See Also

[kfoldPredict](#) | [kfoldEdge](#) | [kfoldMargin](#) | [kfoldfun](#) | [crossval](#)

# ClassificationPartitionedModel.kfoldLoss

---

## Purpose

Classification loss for observations not used for training

## Syntax

```
L = kfoldLoss(obj)
L = kfoldLoss(obj,Name,Value)
```

## Description

`L = kfoldLoss(obj)` returns loss obtained by cross-validated classification model `obj`. For every fold, this method computes classification loss for in-fold observations using a model trained on out-of-fold observations.

`L = kfoldLoss(obj,Name,Value)` calculates loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### **obj**

Object of class `ClassificationPartitionedModel`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### **'folds'**

Indices of folds ranging from 1 to `obj.KFold`. Use only these folds for predictions.

**Default:** `1:obj.KFold`

### **'lossfun'**

Function handle or string representing a loss function. Built-in loss functions:

# ClassificationPartitionedModel.kfoldLoss

---

- 'binodeviance' — See “Loss Functions” on page 21-1543.
- 'classiferror' — Fraction of misclassified observations. See “Loss Functions” on page 21-1543.
- 'exponential' — See “Loss Functions” on page 21-1543.
- 'mincost' — Smallest misclassification cost as given by the `obj.Cost` matrix. See “Loss Functions” on page 21-1543.

You can write your own loss function in the syntax described in “Loss Functions” on page 21-1543.

**Default:** 'mincost'

## 'mode'

A string for determining the output of `kfoldLoss`:

- 'average' —  $L$  is a scalar, the loss averaged over all folds.
- 'individual' —  $L$  is a vector of length `obj.KFold`, where each entry is the loss for a fold.

**Default:** 'average'

## Output Arguments

**L**

Loss, by default the fraction of misclassified data.  $L$  can be a vector, and can mean different things, depending on the name-value pair settings.

## Definitions

### Classification Error

The default classification error is the fraction of the data  $X$  that `obj` misclassifies, where  $Y$  are the true classifications.

Weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when `obj` misclassifies the  $i$ th row of  $X$ , divided by the sum of the weights.

## Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

- 'classiferror' — Predict the label with the largest posterior probability. The loss is then the fraction of misclassified observations.
- 'mincost' — Predict the label with the smallest expected misclassification cost, with expectation taken over the posterior probability, and cost as given by the Cost property of the classifier (a matrix). The loss is then the true misclassification cost averaged over the observations.

To write your own loss function, create a function file in this form:

```
function loss = lossfun(C,S,W,COST)
```

- N is the number of rows of X.
- K is the number of classes in the classifier, represented in the ClassNames property.
- C is an N-by-K logical matrix, with one true per row for the true class. The index for each class is its position in the ClassNames property.
- S is an N-by-K numeric matrix. S is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from predict.

# ClassificationPartitionedModel.kfoldLoss

---

- $W$  is a numeric vector with  $N$  elements, the observation weights. If you pass  $W$ , the elements are normalized to sum to the prior probabilities in the respective classes.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, you can use  $COST = \text{ones}(K) - \text{eye}(K)$ , which means a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## Examples

Find the average cross-validated classification error for a model of the ionosphere data:

```
load ionosphere
tree = ClassificationTree.fit(X,Y);
cvtree = crossval(tree);
L = kfoldLoss(cvtree)
```

```
L =
    0.1197
```

## See Also

[ClassificationPartitionedModel](#) | [kfoldPredict](#) | [kfoldEdge](#) | [kfoldMargin](#) | [kfoldfun](#) | [crossval](#)

## How To

- “Examine the Quality of a KNN Classifier” on page 16-27



# RegressionPartitionedEnsemble.kfoldLoss

---

## Purpose

Cross-validation loss of partitioned regression ensemble

## Syntax

```
L = kfoldLoss(cvens)
L = kfoldLoss(cvens,Name,Value)
```

## Description

`L = kfoldLoss(cvens)` returns the cross-validation loss of `cvens`.

`L = kfoldLoss(cvens,Name,Value)` returns cross-validation loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### `cvens`

Object of class `RegressionPartitionedEnsemble`. Create `obj` with `fitensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `obj` from a regression ensemble with `crossval`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### **'folds'**

Indices of folds ranging from 1 to `cvens.KFold`. Use only these folds for predictions.

**Default:** `1:cvens.KFold`

### **'lossfun'**

# RegressionPartitionedEnsemble.kfoldLoss

---

Function handle for loss function, or the string 'mse', meaning mean squared error. If you pass a function handle `fun`, `loss` calls it as

```
fun(Y,Yfit,W)
```

where `Y`, `Yfit`, and `W` are numeric vectors of the same length.

- `Y` is the observed response.
- `Yfit` is the predicted response.
- `W` is the observation weights.

The returned value `fun(Y,Yfit,W)` should be a scalar.

**Default:** 'mse'

## 'mode'

String representing the meaning of the output `L`:

- 'ensemble' — `L` is a scalar value, the loss for the entire ensemble.
- 'individual' — `L` is a vector with one element per trained learner.
- 'cumulative' — `L` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**L**

The loss (mean squared error) between the observations in a fold when compared against predictions made with an ensemble trained on the out-of-fold data. `L` can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

Find the cross-validation loss for a regression ensemble of the carsmall data:

```
load carsmall
X = [Displacement Horsepower Weight];
rens = fitensemble(X,MPG,'LSboost',100,'Tree');
cvrens = crossval(rens);
L = kfoldLoss(cvrens)
```

```
L =
    25.6935
```

## See Also

[RegressionPartitionedEnsemble](#) | [loss](#) | [kfoldPredict](#)

# RegressionPartitionedModel.kfoldLoss

---

**Purpose** Cross-validation loss of partitioned regression model

**Syntax**  
L = kfoldLoss(cvmodel)  
L = kfoldLoss(cvmodel,Name,Value)

**Description** L = kfoldLoss(cvmodel) returns the cross-validation loss of cvmodel.  
L = kfoldLoss(cvmodel,Name,Value) returns cross-validation loss with additional options specified by one or more Name,Value pair arguments. You can specify several name-value pair arguments in any order as Name1,Value1, ,NameN,ValueN.

**Input Arguments**

**cvmodel**  
Object of class RegressionPartitionedModel. Create obj with RegressionTree.fit along with one of the cross-validation options: 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'. Alternatively, create obj from a regression tree with crossval.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, . . . ,NameN,ValueN.

### **'folds'**

Indices of folds ranging from 1 to obj.KFold. Use only these folds for predictions.

**Default:** 1:obj.KFold

### **'lossfun'**

# RegressionPartitionedModel.kfoldLoss

---

Function handle for loss function, or the string 'mse', meaning mean squared error. If you pass a function handle `fun`, `kfoldLoss` calls it as

```
fun(Y,Yfit,W)
```

where `Y`, `Yfit`, and `W` are numeric vectors of the same length.

- `Y` is the observed response.
- `Yfit` is the predicted response.
- `W` is the observation weights.

The returned value `fun(Y,Yfit,W)` should be a scalar.

**Default:** 'mse'

## 'mode'

One of the following strings:

- 'average' — `L` is the average loss over all folds.
- 'individual' — `L` is a vector of the individual losses of in-fold observations trained on out-of-fold data.

**Default:** 'average'

## Output Arguments

**L**

The loss (mean squared error) between the observations in a fold when compared against predictions made with a tree trained on the out-of-fold data. If `mode` is 'individual', `L` is a vector of the losses. If `mode` is 'average', `L` is the average loss.

## Examples

Construct a partitioned regression model, and examine the cross-validation losses for the folds:

```
load carsmall
```

# RegressionPartitionedModel.kfoldLoss

---

```
XX = [Cylinders Displacement Horsepower Weight];
YY = MPG;
cvmodel = RegressionTree.fit(XX,YY,'crossval','on');
L = kfoldLoss(cvmodel,'mode','individual')
```

```
L =
    44.9635
    11.8525
    18.2046
     9.2965
    29.4329
    54.8659
    24.6446
     8.2085
    19.7593
    16.7394
```

## Alternatives

You can avoid constructing a cross-validated tree model by calling `cvLoss` instead of `kfoldLoss`. The cross-validated tree can save time if you are going to examine it more than once.

## See Also

`loss` | `kfoldPredict`

# ClassificationPartitionedModel.kfoldMargin

---

<b>Purpose</b>	Classification margins for observations not used for training
<b>Syntax</b>	<code>M = kfoldMargin(obj)</code>
<b>Description</b>	<code>M = kfoldMargin(obj)</code> returns classification margins obtained by cross-validated classification model <code>obj</code> . For every fold, this method computes classification margins for in-fold observations using a model trained on out-of-fold observations.
<b>Input Arguments</b>	<b>obj</b> A partitioned classification model of type <code>ClassificationPartitionedModel</code> or <code>ClassificationPartitionedEnsemble</code> .
<b>Output Arguments</b>	<b>M</b> The classification margin.
<b>Definitions</b>	<b>Margin</b> The classification <i>margin</i> is the difference between the classification <i>score</i> for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix <code>Obj.X</code> . A high value of margin indicates a more reliable prediction than a low value. <b>Score (discriminant analysis)</b> For discriminant analysis, the <i>score</i> of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 15-7. <b>Score (ensemble)</b> For ensembles, a classification <i>score</i> represents the confidence of a classification into a class. The higher the score, the higher the confidence.

# ClassificationPartitionedModel.kfoldMargin

---

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## Score (tree)

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

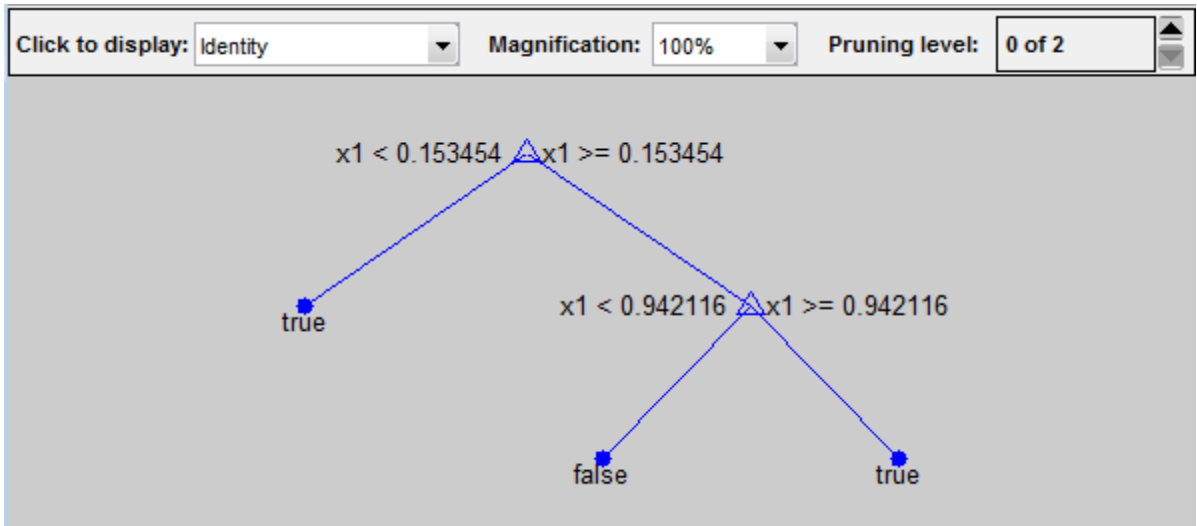
1

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```



# ClassificationPartitionedModel.kfoldMargin

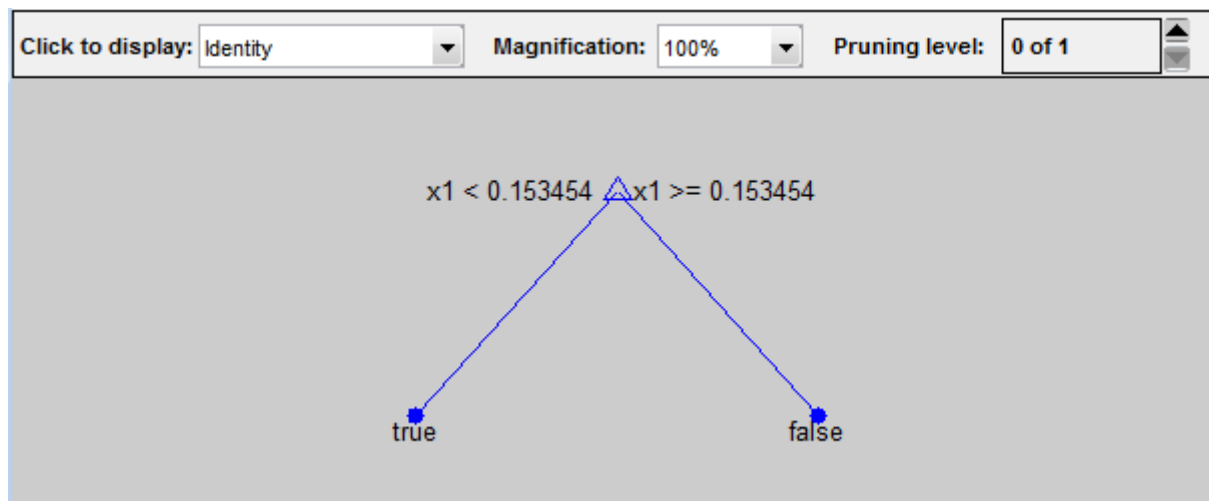


2

Prune the tree:

```
tree1 = prune(tree, 'level', 1);  
view(tree1, 'mode', 'graph')
```

# ClassificationPartitionedModel.kfoldMargin



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785
```

# ClassificationPartitionedModel.kfoldMargin

---

```
0.9059    0.0941    0.5469
0.9059    0.0941    0.9575
0.9059    0.0941    0.9649
```

Indeed, every value of X (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of X have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in X in the range (.95, 1) instead of the expected 5 observations.

## Examples

Find the k-fold margins for an ensemble that classifies the ionosphere data:

```
load ionosphere
ens = fitensemble(X,Y,'AdaBoostM1',100,'Tree');
cvens = crossval(ens);
M = kfoldMargin(cvens);
[min\(M\) mean\(M\) max\(M\)]

ans =
    -7.6394     7.3469    22.4833
```

## See Also

[ClassificationPartitionedModel](#) | [kfoldPredict](#) | [kfoldEdge](#) | [kfoldLoss](#) | [kfoldfun](#) | [crossval](#)

# ClassificationPartitionedModel.kfoldPredict

---

**Purpose** Predict response for observations not used for training

**Syntax**

```
label = kfoldPredict(obj)
[label,score] = kfoldPredict(obj)
[label,score,cost] = kfoldPredict(obj)
```

**Description** `label = kfoldPredict(obj)` returns class labels predicted by `obj`, a cross-validated classification. For every fold, `kfoldPredict` predicts class labels for in-fold observations using a model trained on out-of-fold observations.

`[label,score] = kfoldPredict(obj)` returns the predicted classification scores for in-fold observations using a model trained on out-of-fold observations.

`[label,score,cost] = kfoldPredict(obj)` returns misclassification costs.

## Input Arguments

**obj**  
Object of class `ClassificationPartitionedModel` or `ClassificationPartitionedEnsemble`.

## Output Arguments

**label**  
Vector of class labels of the same type as the response data used in training `obj`. Each entry of `label` corresponds to a predicted class label for the corresponding row of `X`.

**score**  
Numeric matrix of size N-by-K, where N is the number of observations (rows) in `obj.X`, and K is the number of classes (in `obj.ClassNames`). `score(i,j)` represents the confidence that row `i` of `obj.X` is of class `j`. For details, see “Definitions” on page 21-1557.

**cost**

Numeric matrix of misclassification costs of size N-by-K.  $\text{cost}(i, j)$  is the average misclassification cost of predicting that row  $i$  of  $\text{obj.X}$  is of class  $j$ .

## Definitions

### Cost (discriminant analysis)

The *average misclassification cost* is the mean misclassification cost for predictions made by the cross-validated classifiers trained on out-of-fold observations. The matrix of expected costs per observation is defined in “Cost” on page 15-8.

### Score (discriminant analysis)

For discriminant analysis, the *score* of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 15-7.

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

### Score (tree)

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

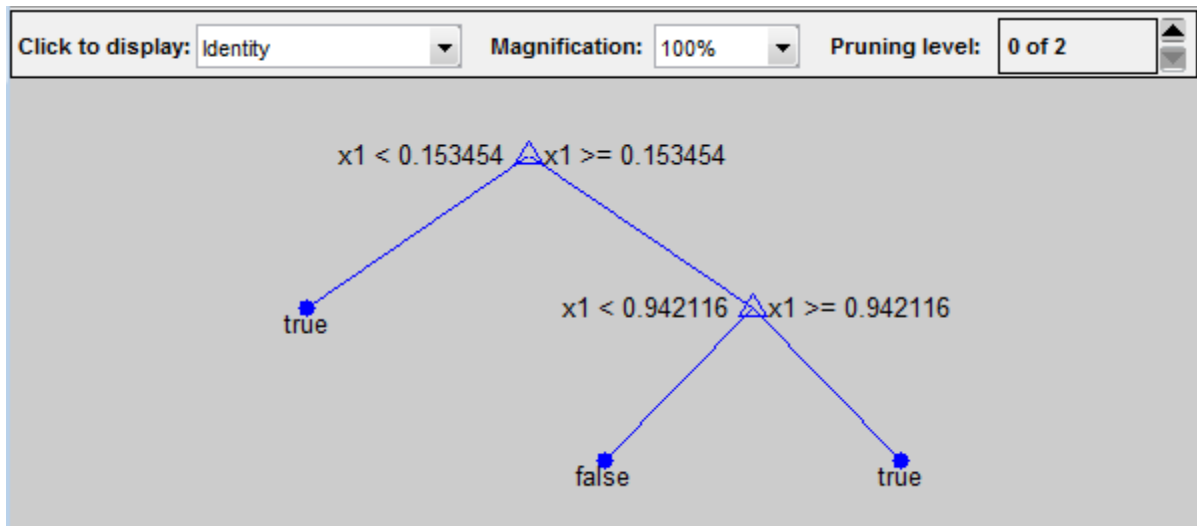
For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

# ClassificationPartitionedModel.kfoldPredict

1

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```

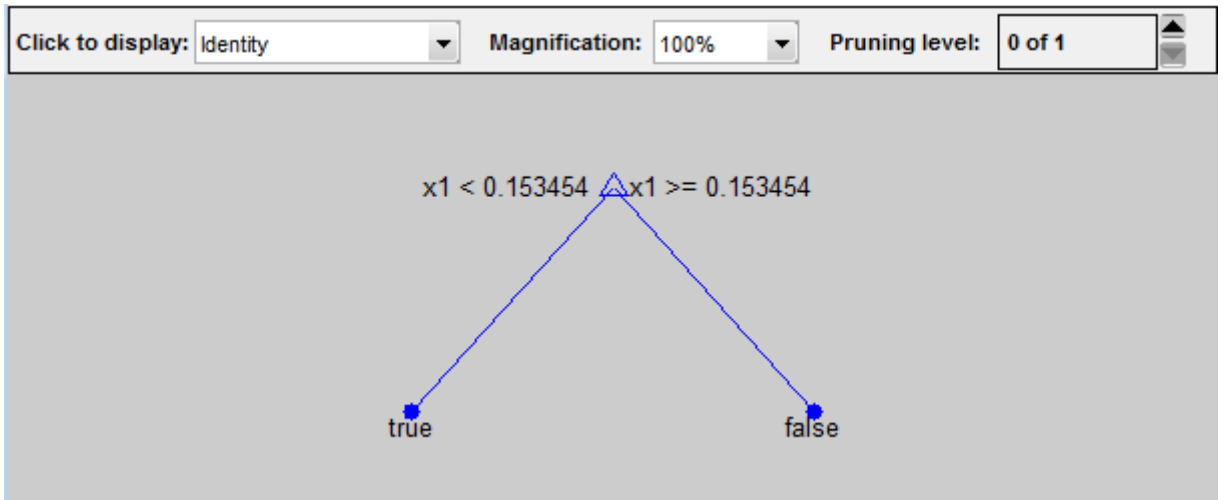


2

Prune the tree:

```
tree1 = prune(tree,'level',1);
view(tree1,'mode','graph')
```

# ClassificationPartitionedModel.kfoldPredict



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785
```

# ClassificationPartitionedModel.kfoldPredict

---

```
0.9059    0.0941    0.5469
0.9059    0.0941    0.9575
0.9059    0.0941    0.9649
```

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range (.95, 1) instead of the expected 5 observations.

## Examples

Find the cross-validation predictions for a model based on the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
cvens = crossval(ens);
[elabel escore] = kfoldPredict(cvens);
max(escore)
```

```
ans =
    9.3634    8.5624    9.3981
```

```
min(escore)
```

```
ans =
    0.0017    3.7518    0.8911
```

## See Also

[ClassificationPartitionedModel](#) | [kfoldEdge](#) | [kfoldMargin](#) | [kfoldLoss](#) | [kfoldfun](#) | [crossval](#)



# RegressionPartitionedModel.kfoldPredict

---

## Purpose

Predict response for observations not used for training.

## Syntax

```
yfit = kfoldPredict(obj)
```

## Description

`yfit = kfoldPredict(obj)` returns the predicted values for the responses of the training data based on `obj`, an object trained on out-of-fold observations.

## Input Arguments

### **obj**

Object of class `RegressionPartitionedModel`. Create `obj` with `RegressionTree.fit` or `fitensemble` along with one of the cross-validation options: `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. Alternatively, create `obj` from a regression tree or regression ensemble with `crossval`.

## Output Arguments

### **yfit**

A vector of predicted values for the response data based on a model trained on out-of-fold observations.

## Examples

Construct a partitioned regression model, and examine the cross-validation loss. The cross-validation loss is the mean squared error between `yfit` and the true response data:

```
load carsmall
XX = [Cylinders Displacement Horsepower Weight];
YY = MPG;
tree = RegressionTree.fit(XX,YY);
cvmodel = crossval(tree);
L = kfoldLoss(cvmodel)

L =
    26.5271

yfit = kfoldPredict(cvmodel);
mean( (yfit - tree.Y).^2 )
```

# RegressionPartitionedModel.kfoldPredict

---

```
ans =  
  26.5271
```

**See Also**      `kfoldLoss`

**Purpose**

*K*-means clustering

**Syntax**

```
IDX = kmeans(X,k)
[IDX,C] = kmeans(X,k)
[IDX,C,sumd] = kmeans(X,k)
[IDX,C,sumd,D] = kmeans(X,k)
[...] = kmeans(...,param1,val1,param2,val2,...)
```

**Description**

`IDX = kmeans(X,k)` partitions the points in the *n*-by-*p* data matrix *X* into *k* clusters. This iterative partitioning minimizes the sum, over all clusters, of the within-cluster sums of point-to-cluster-centroid distances. Rows of *X* correspond to points, columns correspond to variables. `kmeans` returns an *n*-by-1 vector `IDX` containing the cluster indices of each point. By default, `kmeans` uses squared Euclidean distances. When *X* is a vector, `kmeans` treats it as an *n*-by-1 data matrix, regardless of its orientation.

`[IDX,C] = kmeans(X,k)` returns the *k* cluster centroid locations in the *k*-by-*p* matrix `C`.

`[IDX,C,sumd] = kmeans(X,k)` returns the within-cluster sums of point-to-centroid distances in the 1-by-*k* vector `sumd`.

`[IDX,C,sumd,D] = kmeans(X,k)` returns distances from each point to every centroid in the *n*-by-*k* matrix `D`.

`[...] = kmeans(...,param1,val1,param2,val2,...)` enables you to specify optional parameter/value pairs to control the iterative algorithm used by `kmeans`. Valid parameter strings are listed in the following table.

# kmeans

---

Parameter	Value
'distance'	Distance measure, in p-dimensional space. kmeans minimizes with respect to this parameter. kmeans computes centroid clusters differently for the different supported distance measures.
	'sqEuclidean' Squared Euclidean distance (default). Each centroid is the mean of the points in that cluster.
	'cityblock' Sum of absolute differences, i.e., the L1 distance. Each centroid is the component-wise median of the points in that cluster.
	'cosine' One minus the cosine of the included angle between points (treated as vectors). Each centroid is the mean of the points in that cluster, after normalizing those points to unit Euclidean length.
	'correlation' One minus the sample correlation between points (treated as sequences of values). Each centroid is the component-wise mean of the points in that cluster, after centering and normalizing those points to zero mean and unit standard deviation.
	'Hamming' Percentage of bits that differ (only suitable for binary data). Each centroid is the component-wise median of points in that cluster.

Parameter	Value	
'emptyaction'	Action to take if a cluster loses all its member observations.	
	'error'	Treat an empty cluster as an error (default).
	'drop'	Remove any clusters that become empty. kmeans sets the corresponding return values in C and D to NaN.
	'singleton'	Create a new cluster consisting of the one point furthest from its centroid.
'onlinephase'	Flag indicating whether kmeans should perform an online update phase in addition to a batch update phase. The online phase can be time consuming for large data sets, but guarantees a solution that is a local minimum of the distance criterion, that is, a partition of the data where moving any single point to a different cluster increases the total sum of distances.	
	'on'	Perform online update (default).
	'off'	Do not perform online update.

# kmeans

Parameter	Value
'options'	Structure specifying options for the iterative algorithm used to minimize the fitting criteria. Create the options structure with <code>statset</code> . Applicable <code>statset</code> parameters are:
Display	Level of display output. Choices are <code>'off'</code> (default), <code>'iter'</code> , and <code>'final'</code> .
MaxIter	Maximum number of iterations allowed. The default is 100.
UseParallel	If true and if a <code>parpool</code> of the Parallel Computing Toolbox is open, compute in parallel. If the Parallel Computing Toolbox is not installed, or a <code>parpool</code> is not open, computation occurs in serial mode. Default is <code>default</code> , meaning serial computation.
UseSubstreams	Set to true to compute in parallel in a reproducible fashion. Default is false. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code> .
Streams	A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code> , <code>kmeans</code> uses the default stream or streams. If you choose to specify <code>Streams</code> , use a single object except in the case: <ul style="list-style-type: none"><li>• You have an open Parallel pool</li><li>• <code>UseParallel</code> is true</li></ul>

Parameter	Value	
	<ul style="list-style-type: none"> <li>UseSubstreams is false</li> </ul> <p>In that case, use a cell array the same size as the Parallel pool. If a Parallel pool is not open, then Streams must supply a single random number stream.</p>	
'replicates'	Number of times to repeat the clustering, each with a new set of initial cluster centroid positions. kmeans returns the solution with the lowest value for sumd. You can supply 'replicates' implicitly by supplying a 3D array as the value for the 'start' parameter.	
'start'	Method used to choose the initial cluster centroid positions, sometimes known as <i>seeds</i> .	
	'sample'	Select k observations from X at random (default).
	'uniform'	Select k points uniformly at random from the range of X. Not valid with Hamming distance.
	'cluster'	Perform a preliminary clustering phase on a random 10% subsample of X. This preliminary phase is itself initialized using 'sample'.
	Matrix	k-by-p matrix of centroid starting locations. In this case, you can pass in [ ] for k, and kmeans infers k from the first dimension of the matrix. You can also supply a 3-D array, implying a value for the

# kmeans

---

Parameter	Value
	'replicates' parameter from the array's third dimension.

## Algorithms

kmeans uses a two-phase iterative algorithm to minimize the sum of point-to-centroid distances, summed over all k clusters:

- 1** The first phase uses *batch updates*, where each iteration consists of reassigning points to their nearest cluster centroid, all at once, followed by recalculation of cluster centroids. This phase occasionally does not converge to solution that is a local minimum, that is, a partition of the data where moving any single point to a different cluster increases the total sum of distances. This is more likely for small data sets. The batch phase is fast, but potentially only approximates a solution as a starting point for the second phase.
- 2** The second phase uses *online updates*, where points are individually reassigned if doing so will reduce the sum of distances, and cluster centroids are recomputed after each reassignment. Each iteration during the second phase consists of one pass though all the points. The second phase will converge to a local minimum, although there may be other local minima with lower total sum of distances. The problem of finding the global minimum can only be solved in general by an exhaustive (or clever, or lucky) choice of starting points, but using several replicates with random starting points typically results in a solution that is a global minimum.

## References

- [1] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.
- [2] Spath, H. *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*. Translated by J. Goldschmidt. New York: Halsted Press, 1985.



**Examples**

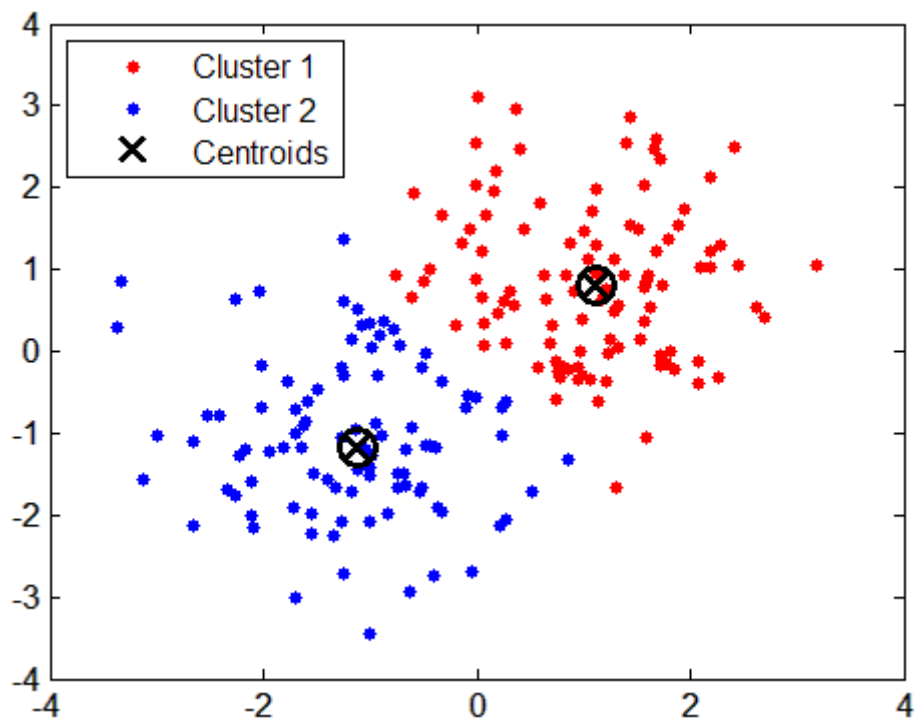
The following creates two clusters from separated random data:

```
X = [randn(100,2)+ones(100,2);...
      randn(100,2)-ones(100,2)];
opts = statset('Display','final');

[idx,ctrs] = kmeans(X,2,...
                   'Distance','city',...
                   'Replicates',5,...
                   'Options',opts);
5 iterations, total sum of distances = 284.671
4 iterations, total sum of distances = 284.671
4 iterations, total sum of distances = 284.671
3 iterations, total sum of distances = 284.671
3 iterations, total sum of distances = 284.671

plot(X(idx==1,1),X(idx==1,2),'r.','MarkerSize',12)
hold on
plot(X(idx==2,1),X(idx==2,2),'b.','MarkerSize',12)
plot(ctrs(:,1),ctrs(:,2),'kx',...
      'MarkerSize',12,'LineWidth',2)
plot(ctrs(:,1),ctrs(:,2),'ko',...
      'MarkerSize',12,'LineWidth',2)
legend('Cluster 1','Cluster 2','Centroids',...
      'Location','NW')
```

# kmeans



## See Also

[linkage](#) | [clusterdata](#) | [silhouette](#)

**Purpose** Find  $k$ -nearest neighbors using ExhaustiveSearcher object

**Syntax**

```
IDX = knnsearch(NS,Y)
[IDX,D] = knnsearch(NS,Y)
[IDX,D] = knnsearch(NS,Y, 'Name', Value)
```

**Description** `IDX = knnsearch(NS,Y)` finds the nearest neighbor (closest point) in `NS.X` for each point in `Y`. Rows of `Y` correspond to observations and columns correspond to features. `Y` must have the same number of columns as `NS.X`. `IDX` is a column vector with  $n_y$  rows, where  $n_y$  is the number of rows in `Y`. Each row in `IDX` contains the index of observation in `NS.X` which has the smallest distance to the corresponding observation in `Y`.

`[IDX,D] = knnsearch(NS,Y)` returns a column vector `D` containing the distances between each observation in `Y` and the corresponding closest observation in `NS.X`. That is, `D(i)` is the distance between `NS.X(IDX(i),:)` and `Y(i,:)`.

`[IDX,D] = knnsearch(NS,Y, 'Name', Value)` accepts one or more comma-separated argument name/value pairs. Specify *Name* inside single quotes.

## Input Arguments

### Name-Value Pair Arguments

#### 'K'

A positive integer,  $k$ , specifying the number of nearest neighbors in `NS.X` for each point in `Y`. Default is 1. `IDX` and `D` are  $n_y$ -by- $k$  matrices. `D` sorts the distances in each row in ascending order. Each row in `IDX` contains the indices of the  $k$  closest neighbors in `NS.X` corresponding to the  $k$  smallest distances in `D`.

#### 'IncludeTies'

A logical value indicating whether `knnsearch` includes all the neighbors whose distance values are equal to the  $K$ th smallest distance. If `IncludeTies` is `true`, `knnsearch` includes all these neighbors. In this case, `IDX` and `D` are  $n_y$ -by-1 cell arrays. Each

# ExhaustiveSearcher.knnsearch

---

row in `IDX` and `D` contains a vector with at least `K` numeric numbers. `D` sorts the distances in each vector in ascending order. Each row in `IDX` contains the indices of the closest neighbors corresponding to these smallest distances in `D`.

**Default:** `false`

## 'Distance'

- `'euclidean'` — Euclidean distance (default).
- `'seuclidean'` — Standardized Euclidean distance. Each coordinate difference between `X` and each query point is scaled by dividing by a scale value `S`. The default value of `S` is `NS.DistanceParameter` if `NS.Distance` is `'seuclidean'`, otherwise the default is the standard deviation computed from `X`, `S=nanstd(X)`. To specify another value for `S`, use the `'Scale'` argument.
- `'cityblock'` — City block distance.
- `'chebychev'` — Chebychev distance (maximum coordinate difference).
- `'minkowski'` — Minkowski distance.
- `'mahalanobis'` — Mahalanobis distance, which is computed using a positive definite covariance matrix `C`. The default value of `C` is `nancov(X)`. To change the value of `C`, use the `Cov` parameter.
- `'cosine'` — One minus the cosine of the included angle between observations (treated as vectors).
- `'correlation'` — One minus the sample linear correlation between observations (treated as sequences of values).
- `'spearman'` — One minus the sample Spearman's rank correlation between observations (treated as sequences of values).

- 'hamming' — Hamming distance, which is the percentage of coordinates that differ.
- 'jaccard' — One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
- custom distance function — A distance function specified using @ (for example, @distfun). A distance function must be of the form function D2 = distfun(ZI, ZJ), taking as arguments a 1-by- $n$  vector ZI containing a single row of from X or from the query points Y, an  $m2$ -by- $n$  matrix ZJ containing multiple rows of X or Y, and returning an  $m2$ -by-1 vector of distances D2, whose  $j$ th element is the distance between the observations ZI and ZJ( $j$ ,:).

Default is NS.Distance. For more information on these distance metrics, see “Distance Metrics” on page 16-9.

## **'p'**

A positive scalar,  $p$ , indicating the exponent of the Minkowski distance. This parameter is only valid if knnsearch uses the 'minkowski' distance metric. Default is NS.Distance if NS.Distance is 'minkowski' and 2 otherwise.

## **'Cov'**

A positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This parameter is only valid when knnsearch uses the 'mahalanobis' distance metric. Default is NS.Distance if NS.Distance is 'mahalanobis', or nancov(X) otherwise.

## **'Scale'**

A vector S with the length equal to the number of columns in X. Each coordinate of X and each query point is scaled by the corresponding element of S when computing the standardized Euclidean distance. This parameter is only valid when Distance is 'seuclidean'. Default is nanstd(X).

# ExhaustiveSearcher.knnsearch

---

## Examples

Create an ExhaustiveSearcher object specifying 'cosine' as the distance metric. Perform a  $k$ -nearest neighbors search on the object using the mahalanobis metric and compare the results:

```
load fisheriris
x = meas(:,3:4);
exhaustiveobj = ExhaustiveSearcher(x,'Distance','cosine')

exhaustiveobj =

    ExhaustiveSearcher

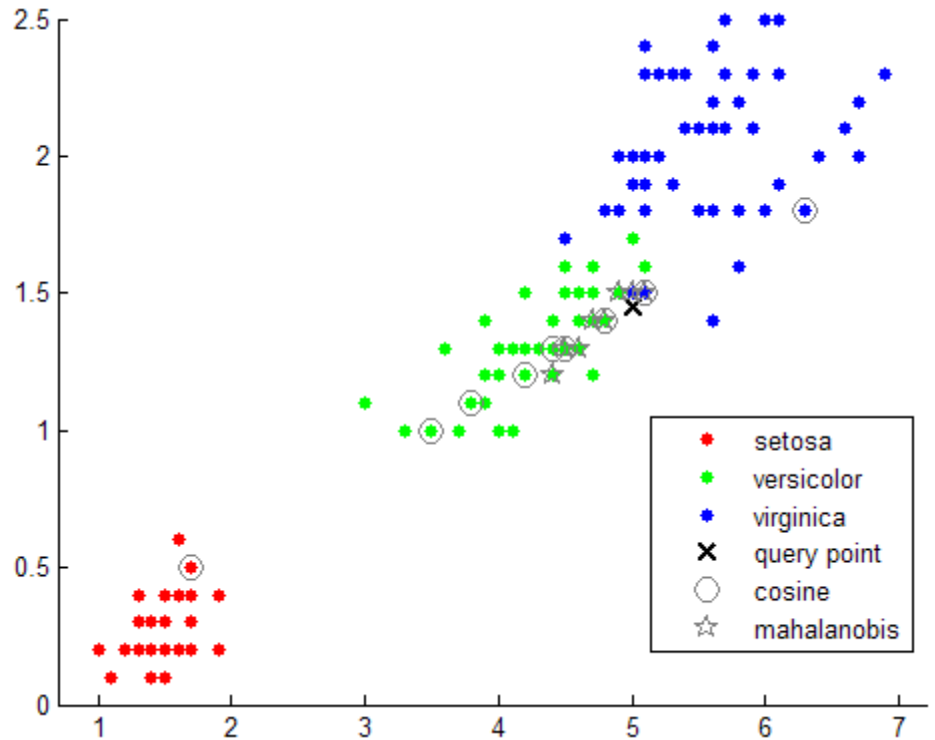
    Properties:
        X: [150x2 double]
        Distance: 'cosine'
        DistParameter: []

% Perform a knnsearch between x and a query point, using
% first cosine then mahalanobis distance metrics:
newpoint = [5 1.45];
[n,d]=knnsearch(exhaustiveobj,newpoint,'k',10);
[nmah,dmah] = knnsearch(exhaustiveobj,newpoint,'k',10,...
    'distance','mahalanobis');

% Visualize the results of the two different nearest
% neighbors searches:

% First plot the training data:
gscatter(x(:,1),x(:,2),species)
% Plot an X for the query point:
line(newpoint(1),newpoint(2),'marker','x','color','k',...
    'markersize',10,'linewidth',2,'linestyle','none')
% Use circles to denote the cosine nearest neighbors:
line(x(n,1),x(n,2),'color',[.5 .5 .5],'marker','o',...
    'linestyle','none','markersize',10)
% Use pentagrams to denote the mahalanobis nearest neighbors:
line(x(nmah,1),x(nmah,2),'color',[.5 .5 .5],'marker','p',...
```

```
'linestyle','none','markersize',10)  
legend('setosa','versicolor','virginica','query point',...  
       'cosine','mahalanobis')  
set(legend,'location','best')
```



## Algorithms

For information on a specific search algorithm, see “Distance Metrics” on page 16-9.

## See Also

[createns](#) | [ExhaustiveSearcher](#) | [KDTreeSearcher.knnsearch](#) | [knnsearch](#) | [rangesearch](#)

## How To

- “*k*-Nearest Neighbor Search and Radius Search” on page 16-12

# ExhaustiveSearcher.knnsearch

---

- “Distance Metrics” on page 16-9



## Purpose

Find  $k$ -nearest neighbors using KDTreeSearcher object

## Syntax

```
IDX = knnsearch(NS,Y)
[IDX,D] = knnsearch(NS,Y)
[IDX,D] = knnsearch(NS,Y,'Name',Value)
```

## Description

IDX = knnsearch(NS,Y) finds the nearest neighbor (closest point) in NS.X for each point in Y. Rows of Y correspond to observations and columns correspond to features. Y must have the same number of columns as NS.X. IDX is a column vector with  $n_y$  rows, where  $n_y$  is the number of rows in Y. Each row in IDX contains the index of observation in NS.X which has the smallest distance to the corresponding observation in Y.

[IDX,D] = knnsearch(NS,Y) returns a column vector D containing the distances between each observation in Y and the corresponding closest observation in NS.X. That is, D(i) is the distance between NS.X(IDX(i),:) and Y(i,:).

[IDX,D] = knnsearch(NS,Y,'Name',Value) accepts one or more comma-separated name/value pairs. Specify *Name* inside single quotes.

## Input Arguments

### Name-Value Pair Arguments

#### 'K'

A positive integer,  $k$ , specifying the number of nearest neighbors in NS.X for each point in Y. Default is 1. IDX and D are  $n_y$ -by- $k$  matrices. D sorts the distances in each row in ascending order. Each row in IDX contains the indices of the  $k$  closest neighbors in NS.X corresponding to the  $k$  smallest distances in D.

#### 'Distance'

Select one of the following distance algorithms.

- 'euclidean' — Euclidean distance (default).
- 'cityblock' — City block distance.

# KDTreeSearcher.knnsearch

---

- 'chebychev' — Chebychev distance (maximum coordinate difference).
- 'minkowski' — Minkowski distance.

Default is `NS.Distance`. For more information on these distance metrics, see “Distance Metrics” on page 16-9.

## 'IncludeTies'

A logical value indicating whether `knnsearch` includes all the neighbors whose distance values are equal to the  $k$ th smallest distance. If `IncludeTies` is `true`, `knnsearch` includes all these neighbors. In this case, `IDX` and `D` are  $n_y$ -by-1 cell arrays. Each row in `IDX` and `D` contains a vector with at least  $K$  numeric numbers. `D` sorts the distances in each vector in ascending order. Each row in `IDX` contains the indices of the closest neighbors corresponding to these smallest distances in `D`.

**Default:** `false`

## 'P'

A positive scalar,  $p$ , indicating the exponent of the Minkowski distance. This parameter is only valid when the `Distance` is 'minkowski'. Default is `NS.DistanceParameter` if `NS.Distance` is 'minkowski' and 2 otherwise.

## Examples

Create a `KDTreeSearcher` object specifying 'minkowski' as the distance metric with an exponent of 5. Perform a  $k$ -nearest neighbors search on the object using the chebychev metric and compare the results:

```
load fisheriris
x = meas(:,3:4);
kdtreeNS = KDTreeSearcher(x,'Distance','minkowski','P',5)

kdtreeNS =

    KDTreeSearcher
```

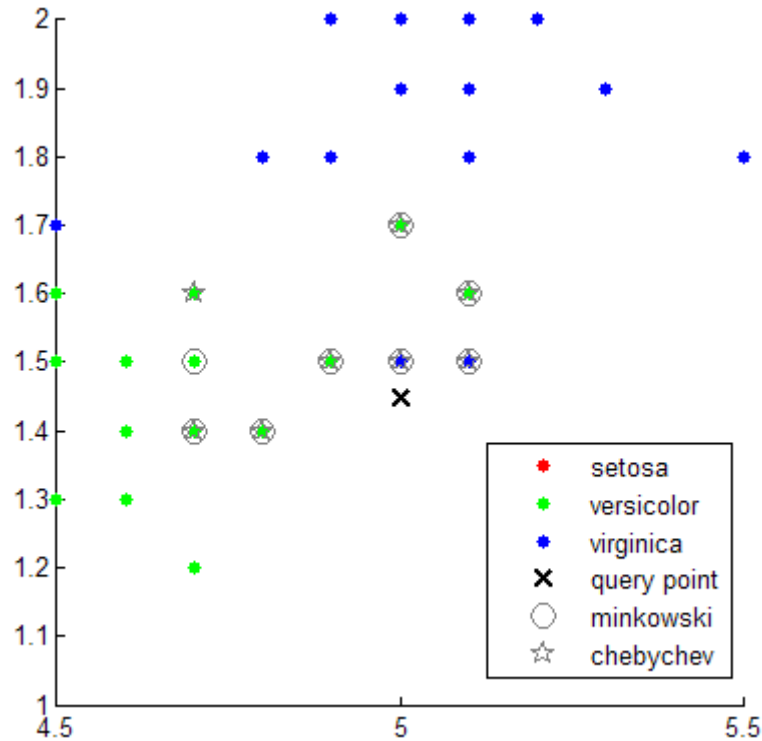
```
Properties:
    BucketSize: 50
           X: [150x2 double]
           Distance: 'minkowski'
    DistParameter: 5

% Perform a knnsearch between X and a query point, using
% first Minkowski then Chebychev distance metrics:
newpoint = [5 1.45];
[n,d]=knnsearch(kdtreeNS,newpoint,'k',10);
[ncb,dcb] = knnsearch(kdtreeNS,newpoint,'k',10,...
    'distance','chebychev');

% Visualize the results of the two different nearest
% neighbors searches:

% First plot the training data:
gscatter(x(:,1),x(:,2),species)
% Zoom in on the points of interest:
set(gca,'xlim',[4.5 5.5],'ylim',[1 2]); axis square
% Plot an X for the query point:
line(newpoint(1),newpoint(2),'marker','x','color','k',...
    'markersize',10,'linewidth',2,'linestyle','none')
% Use circles to denote the Minkowski nearest neighbors:
line(x(n,1),x(n,2),'color',[.5 .5 .5],'marker','o',...
    'linestyle','none','markersize',10)
% Use pentagrams to denote the Chebychev nearest neighbors:
line(x(ncb,1),x(ncb,2),'color',[.5 .5 .5],'marker','p',...
    'linestyle','none','markersize',10)
legend('setosa','versicolor','virginica','query point',...
    'minkowski','chebychev')
set(legend,'location','best')
```

# KDTreeSearcher.knnsearch



## Algorithms

For information on a specific search algorithm, see “Distance Metrics” on page 16-9.

## References

[1] Friedman, J. H., Bentely, J., and Finkel, R. A. (1977). *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, ACM Transactions on Mathematical Software 3, 209.

## See Also

[createns](#) | [ExhaustiveSearcher](#) | [ExhaustiveSearcher.knnsearch](#) | [knnsearch](#) | [rangearch](#)

## How To

- “*k*-Nearest Neighbor Search and Radius Search” on page 16-12

- “Distance Metrics” on page 16-9

# knnsearch

---

**Purpose** Find  $k$ -nearest neighbors using data

**Syntax**  
`IDX = knnsearch(X,Y)`  
`[IDX,D] = knnsearch(X,Y)`  
`[IDX,D] = knnsearch(X,Y, 'Name', Value)`

**Description** `IDX = knnsearch(X,Y)` finds the nearest neighbor in  $X$  for each point in  $Y$ .  $X$  is an  $m_x$ -by- $n$  matrix and  $Y$  is an  $m_y$ -by- $n$  matrix. Rows of  $X$  and  $Y$  correspond to observations and columns correspond to variables. `IDX` is a column vector with  $m_y$  rows. Each row in `IDX` contains the index of nearest neighbor in  $X$  for the corresponding row in  $Y$ .

`[IDX,D] = knnsearch(X,Y)` returns an  $m_y$ -by-1 vector `D` containing the distances between each observation in  $Y$  and the corresponding closest observation in  $X$ . That is, `D(i)` is the distance between  $X(\text{IDX}(i), :)$  and  $Y(i, :)$ .

`[IDX,D] = knnsearch(X,Y, 'Name', Value)` accepts one or more optional comma-separated name/value pairs. Specify *Name* inside single quotes.

`knnsearch` does not save a search object. To create a search object, use `createns`.

**Tips**

- For a fixed positive integer  $K$ , `knnsearch` finds the  $K$  points in  $X$  that are nearest each point in  $Y$ . In contrast, for a fixed positive real value  $r$ , `rangesearch` finds all the points in  $X$  that are within a distance  $r$  of each point in  $Y$ .

**Input Arguments** **Name-Value Pair Arguments**

**'K'**

Positive integer specifying the number of nearest neighbors in  $X$  for each point in  $Y$ . Default is 1. `IDX` and `D` are  $m_y$ -by- $K$  matrices. `D` sorts the distances in each row in ascending order. Each row in `IDX` contains the indices of the  $K$  closest neighbors in  $X$  corresponding to the  $K$  smallest distances in `D`.

**'IncludeTies'**

A logical value indicating whether `knnsearch` includes all the neighbors whose distance values are equal to the *K*th smallest distance. If `IncludeTies` is true, `knnsearch` includes all these neighbors. In this case, `IDX` and `D` are *m<sub>y</sub>*-by-1 cell arrays. Each row in `IDX` and `D` contains a vector with at least *K* numeric numbers. `D` sorts the distances in each vector in ascending order. Each row in `IDX` contains the indices of the closest neighbors corresponding to these smallest distances in `D`.

**Default:** false

**'NSMethod'**

Nearest neighbors search method. Value is either:

- `'kdtree'` — Creates and uses a Kd-tree to find nearest neighbors. This is the default value when the number of columns of *X* is less than 10, *X* is not sparse, and the distance measure is one of the following measures. `'kdtree'` is only valid when the distance measure is one of the following:
  - `'euclidean'`
  - `'cityblock'`
  - `'minkowski'`
  - `'chebychev'`
- `'exhaustive'` — Uses the exhaustive search algorithm by computing the distance values from all the points in *X* to each point in *Y* to find nearest neighbors.

**'Distance'**

A string or a function handle specifying the distance metric. The value can be one of the following:

- `'euclidean'` — Euclidean distance (default).

- 'seuclidean' — Standardized Euclidean distance. Each coordinate difference between rows in  $X$  and the query matrix is scaled by dividing by the corresponding element of the standard deviation computed from  $X$ ,  $S = \text{nanstd}(X)$ . To specify another value for  $S$ , use the `Scale` argument.
- 'cityblock' — City block distance.
- 'chebychev' — Chebychev distance (maximum coordinate difference).
- 'minkowski' — Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' argument.
- 'mahalanobis' — Mahalanobis distance, computed using a positive definite covariance matrix  $C$ . The default value of  $C$  is  $\text{nancov}(X)$ . To change the value of  $C$ , use the `Cov` parameter.
- 'cosine' — 1 minus the cosine of the included angle between observations (treated as vectors).
- 'correlation' — One minus the sample linear correlation between observations (treated as sequences of values).
- 'spearman' — One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
- 'hamming' — Hamming distance, which is the percentage of coordinates that differ.
- 'jaccard' — One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
- custom distance function — A distance function specified using `@` (for example, `@distfun`). A distance function must be of the form function  $D2 = \text{distfun}(ZI, ZJ)$ , taking as arguments a 1-by- $n$  vector  $ZI$  containing a single row of  $X$  or  $Y$ , an  $m2$ -by- $n$  matrix  $ZJ$  containing multiple rows of  $X$  or  $Y$ , and returning an  $m2$ -by-1 vector of distances  $D2$ , whose  $j$ th element is the distance between the observations  $ZI$  and  $ZJ(j,:)$ .

For more information on these distance metrics, see “Distance Metrics” on page 16-9.



**'p'**

A positive scalar,  $p$ , indicating the exponent of the Minkowski distance. This parameter is only valid if the Distance is 'minkowski'. Default is 2.

**'Cov'**

A positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This parameter is only valid when Distance is 'mahalanobis'. Default is `nancov(X)`.

**'Scale'**

A vector `S` containing nonnegative values, with length equal to the number of columns in `X`. Each coordinate of `X` and each query point is scaled by the corresponding element of `S` when computing the standardized Euclidean distance. This argument is only valid when Distance is 'seuclidean'. Default is `nanstd(X)`.

**'BucketSize'**

The maximum number of data points in the leaf node of the *kd*-tree. This argument is only meaningful when using the *kd*-tree search method. Default is 50.

**Examples**

Find the 10 nearest neighbors in `x` to each point in `y` using first the 'minkowski' distance metric with a  $p$  value of 5, and then using the 'chebychev' distance metric. Visually compare the results:

```
load fisheriris
x = meas(:,3:4);
y = [5 1.45;6 2;2.75 .75];

% Perform a knnsearch between x and the query points in y,
% using first Minkowski then Chebychev distance metrics.
[n,d]=knnsearch(x,y,'k',10,'distance','minkowski','p',5);
[ncb,dcb] = knnsearch(x,y,'k',10,...
    'distance','chebychev');
```

# knnsearch

---

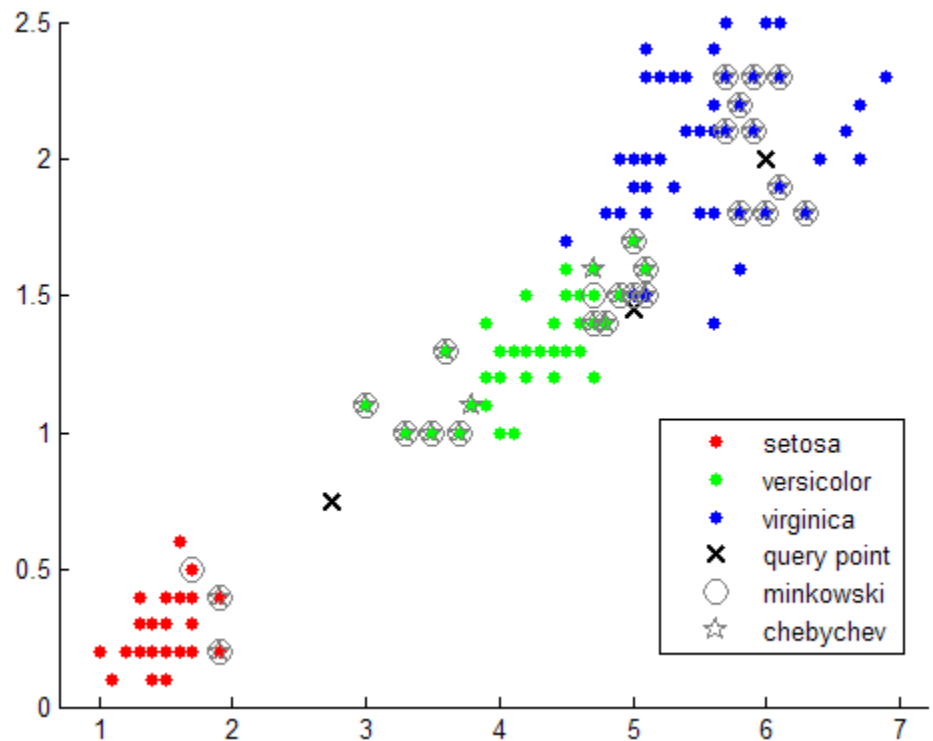
```
% Visualize the results of the two different nearest
% neighbors searches.

% First plot the training data.
gscatter(x(:,1),x(:,2),species)

% Plot an X for the query points.
line(y(:,1),y(:,2),'marker','x','color','k',...
      'markersize',10,'linewidth',2,'linestyle','none')

% Use circles to denote the Minkowski nearest neighbors.
line(x(n,1),x(n,2),'color',[.5 .5 .5],'marker','o',...
      'linestyle','none','markersize',10)

% Use pentagrams to denote the Chebychev nearest neighbors.
line(x(ncb,1),x(ncb,2),'color',[.5 .5 .5],'marker','p',...
      'linestyle','none','markersize',10)
legend('setosa','versicolor','virginica','query point',...
      'minkowski','chebychev')
set(legend,'location','best')
```



## Algorithms

For information on a specific search algorithm, see “Distance Metrics” on page 16-9.

## References

[1] Friedman, J. H., Bentley, J., and Finkel, R. A. (1977) An Algorithm for Finding Best Matches in Logarithmic Expected Time, ACM Transactions on Mathematical Software 3, 209.

## See Also

createns | KDTreeSearcher.knnsearch | ExhaustiveSearcher.knnsearch | rangearch

## How To

• “*k*-Nearest Neighbor Search and Radius Search” on page 16-12

# kruskalwallis

---

## Purpose

Kruskal-Wallis test

## Syntax

```
p = kruskalwallis(x)
p = kruskalwallis(x,group)
p = kruskalwallis(x,group,displayopt)
[p,tbl,stats] = kruskalwallis( ___ )
```

## Description

`p = kruskalwallis(x)` returns the  $p$ -value for the null hypothesis that the data in each column of the matrix `x` comes from the same distribution, using a Kruskal-Wallis test. The alternative hypothesis is that not all samples come from the same distribution. `kruskalwallis` also returns an ANOVA table and a box plot.

`p = kruskalwallis(x,group)` returns the  $p$ -value for a test of the null hypothesis that the data in each categorical group, as specified by the grouping variable `group` comes from the same distribution. The alternative hypothesis is that not all groups come from the same distribution.

`p = kruskalwallis(x,group,displayopt)` returns the  $p$ -value of the test and lets you display or suppress the ANOVA table and box plot.

`[p,tbl,stats] = kruskalwallis( ___ )` also returns the ANOVA table as the cell array `tbl` and the structure `stats` containing information about the test statistics.

## Input Arguments

### **x - Sample data**

vector | matrix

Sample data for the hypothesis test, specified as a vector or an  $m$ -by- $n$  matrix. If `x` is an  $m$ -by- $n$  matrix, each of the  $n$  columns represents an independent sample containing  $m$  mutually independent observations.

### **Data Types**

single | double

**group - Grouping variable**

categorical variable | vector | character array | cell array

Grouping variable, specified as a categorical variable, vector, character array, or cell array.

- If `x` is a vector, then each element in `group` identifies the group to which the corresponding element in `x` belongs, and `group` must be a vector of the same length as `x`. If a row of `group` contains an empty cell or empty string, that row and the corresponding observation in `x` are disregarded. NaN values in either `x` or `group` are similarly ignored.
- If `x` is a matrix, then each column in `x` represents a different group, and you can use `group` to specify labels for these columns. The number of elements in `group` and the number of columns in `x` must be equal.

The labels contained in `group` also annotate the box plot.

**Example:**

```
{ 'red', 'blue', 'green', 'blue', 'red', 'blue', 'green', 'green', 'red' }
```

**Data Types**

single | double | char

**displayopt - Display option**

'on' (default) | 'off'

Display option, specified as 'on' or 'off'. If `displayopt` is 'on', `kruskalwallis` displays the following figures:

- An ANOVA table containing the sums of squares, degrees of freedom, and other quantities calculated based on the ranks of the data in `x`.
- A box plot of the data in each column of the data matrix `x`. The box plots are based on the actual data values, rather than on the ranks.

If `displayopt` is 'off', `kruskalwallis` does not display these figures.

If you specify a value for `displayopt`, you must also specify a value for `group`. If you do not have a grouping variable, specify `group` as `[]`.

**Example:** 'off'

# kruskalwallis

---

## Output Arguments

### **p** - **p-value**

scalar value in the range [0,1]

*p*-value of the test, returned as a scalar value in the range [0,1]. *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of *p* cast doubt on the validity of the null hypothesis.

### **tbl** - **ANOVA table**

cell array

ANOVA table of test results, returned as a cell array. *tbl* includes the sums of squares, degrees of freedom, and other quantities calculated based on the ranks of the data in *x*, as well as column and row labels.

### **stats** - **Test data**

structure

Test data, returned as a structure. You can perform followup multiple comparison tests on pairs of sample medians by using *multcompare*, with *stats* as the input value.

## Examples

### **Test Data Samples for the Same Distribution**

Create two different normal probability distribution objects. The first distribution has *mu* = 0 and *sigma* = 1, and the second distribution has *mu* = 2 and *sigma* = 1.

```
pd1 = makedist('Normal');  
pd2 = makedist('Normal','mu',2,'sigma',1);
```

Create a matrix of sample data by generating random numbers from these two distributions.

```
rng('default'); % for reproducibility  
x = [random(pd1,20,2),random(pd2,20,1)];
```

The first two columns of  $x$  contain data generated from the first distribution, while the third column contains data generated from the second distribution.

Test the null hypothesis that the sample data from each column in  $x$  comes from the same distribution.

```
p = kruskalwallis(x)
```

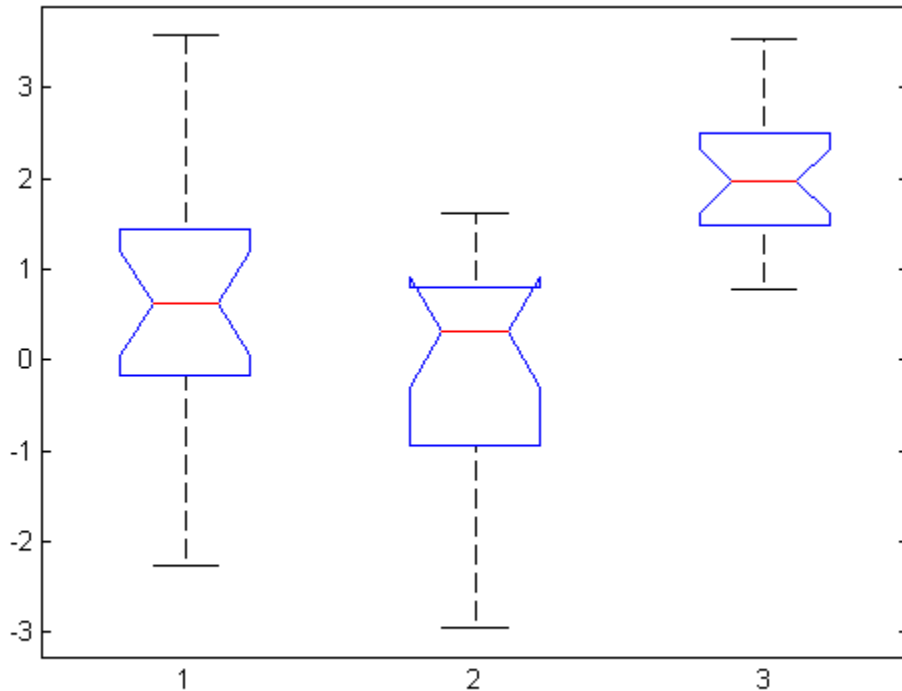
```
p =
```

```
3.6896e-06
```

Kruskal-Wallis ANOVA Table					
Source	SS	df	MS	Chi-sq	Prob>Chi-sq
Columns	7631.1	2	3815.55	25.02	3.68957e-06
Error	10363.9	57	181.82		
Total	17995	59			

# kruskalwallis

---



The returned value of `p` indicates that `kruskalwallis` rejects the null hypothesis that all three data samples come from the same distribution at a 1% significance level. The ANOVA table provides additional test results, and the box plot visually presents the summary statistics for each column in `x`.

## Conduct Followup Tests for Unequal Medians

Create two different normal probability distribution objects. The first distribution has `mu = 0` and `sigma = 1`. The second distribution has `mu = 2` and `sigma = 1`.



```
pd1 = makedist('Normal');  
pd2 = makedist('Normal','mu',2,'sigma',1);
```

Create a matrix of sample data by generating random numbers from these two distributions.

```
rng('default'); % for reproducibility  
x = [random(pd1,20,2),random(pd2,20,1)];
```

The first two columns of `x` contain data generated from the first distribution, while the third column contains data generated from the second distribution.

Test the null hypothesis that the sample data from each column in `x` comes from the same distribution. Suppress the output displays, and generate the structure `stats` to use in further testing.

```
[p,tbl,stats] = kruskalwallis(x,[],'off')
```

```
p =
```

```
3.6896e-06
```

```
tbl =
```

```
Columns 1 through 4
```

'Source'	'SS'	'df'	'MS'
'Columns'	[7.6311e+03]	[ 2]	[3.8155e+03]
'Error'	[1.0364e+04]	[57]	[ 181.8228]
'Total'	[ 17995]	[59]	[ ]

```
Columns 5 through 6
```

'Chi-sq'	'Prob>Chi-sq'
[25.0200]	[ 3.6896e-06]
[ ]	[ ]

# kruskalwallis

---

```
    []          []
```

```
stats =
```

```
    gnames: [3x1 char]
         n: [20 20 20]
    source: 'kruskalwallis'
meanranks: [26.7500 18.9500 45.8000]
    sumt: 0
```

The returned value of `p` indicates that the test rejects the null hypothesis at the 1% significance level. You can use the structure `stats` to perform additional followup testing. The cell array `tbl` contains the same data as the graphical ANOVA table, including column and row labels.

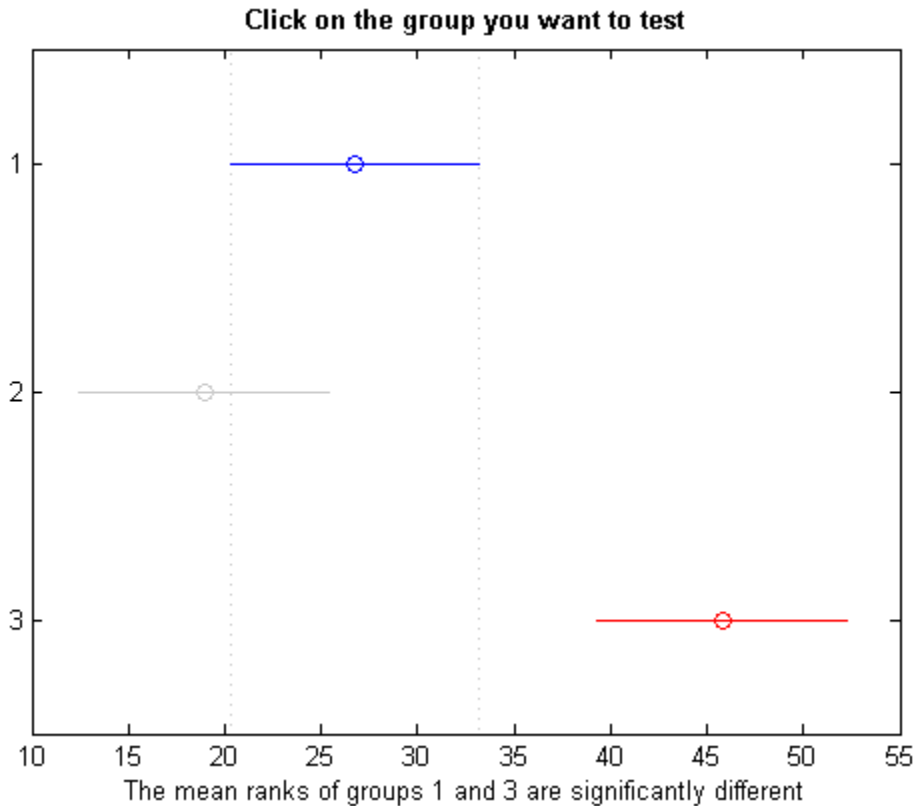
Conduct a followup test to identify which data sample comes from a different distribution.

```
c = multcompare(stats)
```

Note: Intervals can be used for testing but are not simultaneous confidence

```
c =
```

```
    1.0000    2.0000   -5.1435    7.8000   20.7435
    1.0000    3.0000  -31.9935  -19.0500   -6.1065
    2.0000    3.0000  -39.7935  -26.8500  -13.9065
```



The results indicate that there is a significant difference between groups 1 and 3, so the test rejects the null hypothesis that the data in these two groups comes from the same distribution. The same is true for groups 2 and 3. However, there is not a significant difference between groups 1 and 2, so the test does not reject the null hypothesis that these two groups come from the same distribution. Therefore, these results suggest that the data in groups 1 and 2 come from the same distribution, and the data in group 3 comes from a different distribution.

## Test for the Same Distribution Across Groups

Create a vector, `strength`, containing measurements of the strength of metal beams. Create a second vector, `alloy`, containing strings indicating the type of metal alloy from which the corresponding beam is made.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...  
           78 75 76 77 79 79 77 78 82 79];
```

```
alloy = {'st','st','st','st','st','st','st','st',...  
        'al1','al1','al1','al1','al1','al1',...  
        'al2','al2','al2','al2','al2','al2'};
```

Test the null hypothesis that the beam strength measurements have the same distribution across all three alloys.

```
p = kruskalwallis(strength,alloy,'off')
```

```
p =
```

```
0.0018
```

The returned value of `p` indicates that the test rejects the null hypothesis at the 1% significance level.

## Definitions

### Kruskal-Wallis Test

The Kruskal-Wallis test is a nonparametric version of classical one-way ANOVA, and an extension of the Wilcoxon rank sum test to more than two groups. It compares the medians of the groups of data in  $x$  to determine if the samples come from the same population (or, equivalently, from different populations with the same distribution).

The Kruskal-Wallis test uses ranks of the data, rather than numeric values, to compute the test statistics. It finds ranks by ordering the data from smallest to largest across all groups, and taking the numeric index of this ordering. The rank for a tied observation is equal to the average rank of all observations tied with it. The  $F$ -statistic used in

classical one-way ANOVA is replaced by a chi-square statistic, and the  $p$ -value measures the significance of the chi-square statistic.

The Kruskal-Wallis test assumes that all samples come from populations having the same continuous distribution, apart from possibly different locations due to group effects, and that all observations are mutually independent. By contrast, classical one-way ANOVA replaces the first assumption with the stronger assumption that the populations have normal distributions.

## See Also

`anova1` | `boxplot` | `friedman` | `multcompare` | `ranksum`

## Concepts

- “Grouping Variables” on page 2-51

# ksdensity

---

**Purpose** Kernel smoothing function estimate

**Syntax**

```
[f,xi] = ksdensity(x)
[f,xi] = ksdensity(x,pts)
[f,xi] = ksdensity(x,pts,Name,Value)
[f,xi,bw] = ksdensity( ___ )
```

```
ksdensity( ___ )
ksdensity(ax, ___ )
```

**Description** `[f,xi] = ksdensity(x)` returns a probability density estimate, `f`, for the sample in the vector `x`. The estimate is based on a normal kernel function, and is evaluated at 100 equally spaced points, `xi`, that cover the range of the data in `x`.

`ksdensity` works best with continuously distributed samples.

`[f,xi] = ksdensity(x,pts)` returns a probability density estimate, `f`, for the sample in the vector `x`, evaluated at the specified values in vector `pts`. Here, the `xi` and `pts` vectors contain identical values.

`[f,xi] = ksdensity(x,pts,Name,Value)` returns a probability density estimate, `f`, for the sample in the vector `x`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can define the function type `ksdensity` evaluates, such as probability density, cumulative probability, survivor function, and so on. Or you can specify the bandwidth of the smoothing window.

`[f,xi,bw] = ksdensity( ___ )` also returns the bandwidth of the kernel smoothing window, `bw`. The default bandwidth is the optimal for normal densities.

`ksdensity( ___ )` plots the kernel smoothing function estimate.

`ksdensity(ax, ___)` plots the results using axes with the handle, `ax`, instead of the current axes returned by `gca`.

## Input Arguments

### **x** - Sample data

column vector

Sample data, for which `ksdensity` returns `f` values, specified as a column vector.

**Example:** `[f,xi] = ksdensity(x)`

### Data Types

single | double

### **pts** - Points to evaluate `f`

vector

Points to evaluate `f` at, specified as a vector. `pts` can be a row or column vector. `f` has the same dimensions as `pts`.

**Example:** `pts = (0:1:25); ksdensity(x,pts);`

### Data Types

single | double

### **ax** - Axes handle

handle

Axes handle for the figure `ksdensity` plots to, specified as a handle.

For example, if `h` is a handle for a figure, then `ksdensity` can plot to that figure as follows.

**Example:** `ksdensity(h,x)`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

# ksdensity

---

## Example:

`'censoring',cens,'kernel','triangle','npoints',20,'function','cdf'` specifies that `ksdensity` estimates the cdf by evaluating at 20 equally spaced points that covers the range of data, using the triangle kernel smoothing function and accounting for the censored data information in vector `cens`.

## 'censoring' - Logical vector

vector of 0s (default) | vector of 0s and 1s

Logical vector indicating which entries are censored, specified as a vector of binary values. A value of 0 indicates there is no censoring, 1 indicates that observation is censored. Default is there is no censoring.

Example: `'censoring',censdata`

## Data Types

logical

## 'kernel' - Type of kernel smoother

`'normal'` (default) | `'box'` | `'triangle'` | `'epanechnikov'` | function handle | string

Type of kernel smoother, specified as the comma-separated pair consisting of `'kernel'` and one of the following.

- `'normal'` (default)
- `'box'`
- `'triangle'`
- `'epanechnikov'`
- You can also specify a custom kernel function, as a function handle or as a string, e.g., `@normpdf` or `'normpdf'`. This calls the function with one argument that is an array of distances between data values and locations where the density is evaluated. The function must return an array of the same size containing corresponding values of the kernel function.



When 'function' is 'pdf', this kernel function returns density values. Otherwise, it returns cumulative probability values.

Specifying a custom kernel when 'function' is 'icdf' returns an error.

If 'support' is 'positive', then `ksdensity` transforms `x` using a log function, estimates the density of the transformed values, and transforms back to the original scale. If 'support' is a vector [L U], then `ksdensity` uses the transformation  $\log((X-L)/(U-X))$ . The width parameter and `bw` outputs are on the scale of the transformed values.

**Example:** 'kernel', 'box'

### Data Types

char | function\_handle

### 'npoints' - Number of equally spaced points

100 (default) | scalar value

Number of equally spaced points in `xi`, specified as the comma-separated pair consisting of 'npoints' and a scalar value.

For instance, for a kernel smooth estimate of a specified function at 80 equally spaced points within the range of sample data, input:

**Example:** 'npoints', 80

### Data Types

single | double

### 'support' - Support for the density

'unbounded' (default) | 'positive' | two-element vector, [L U]

Support for the density, specified as the comma-separated pair consisting of 'support' and one of the following.

# ksdensity

---

'unbounded'	Default. Allow the density to extend over the whole real line.
'positive'	Restrict the density to positive values.
Two-element vector, [L U]	Give the finite lower and upper bounds for the support of the density.

**Example:** 'support','positive'

**Example:** 'support',[0 10]

## Data Types

single | double | char

## 'weights' - Weights for each x value

vector

Weights for each x value, specified as the comma-separated pair consisting of 'weights' and a vector of the same length as x.

For instance, if the weights for the data values are in vector xw, then you can specify the weights as follows.

**Example:** 'weights',xw

## Data Types

single | double

## 'bandwidth' - Bandwidth of the kernel smoothing window

optimal value for normal densities (default) | scalar value

The bandwidth of the kernel-smoothing window, which is a function of the number of points in x, specified as the comma-separated pair consisting of 'width' and a scalar. The default is optimal for estimating normal densities, but you might want to choose a larger or smaller value to smooth more or less.

**Example:** 'bandwidth',0.8

## Data Types

single | double

**'function' - Function to estimate**`'pdf' (default) | 'cdf' | 'icdf' | 'survivor' | 'cumhazard'`

Function to estimate, specified as the comma-separated pair consisting of 'function' and one of the following.

<code>'pdf'</code>	Default. Probability density function.
<code>'cdf'</code>	Cumulative distribution function.
<code>'icdf'</code>	Inverse cumulative distribution function. For 'icdf', $f = \text{ksdensity}(x, pi, 'function', 'icdf')$ computes
<code>'survivor'</code>	Survivor function.
<code>'cumhazard'</code>	Cumulative hazard function.

**Example:** `'function','icdf'`

**Data Types**

char

**Output Arguments****f - Estimated function values**

vector

Estimated function values, returned as a vector of the same dimension as `xi` or `pts`.

**xi - Evaluation points**

100 equally spaced points (default) | vector

Evaluation points at which `ksdensity` calculates `f`, returned as a vector. Default is 100 equally spaced points that cover the range of data in `x`.

**bw - Bandwidth of smoothing window**

scalar value

Bandwidth of smoothing window, returned as a scalar value.

**Examples****Estimate Density**

Generate a sample data set from a mixture of two normal distributions.

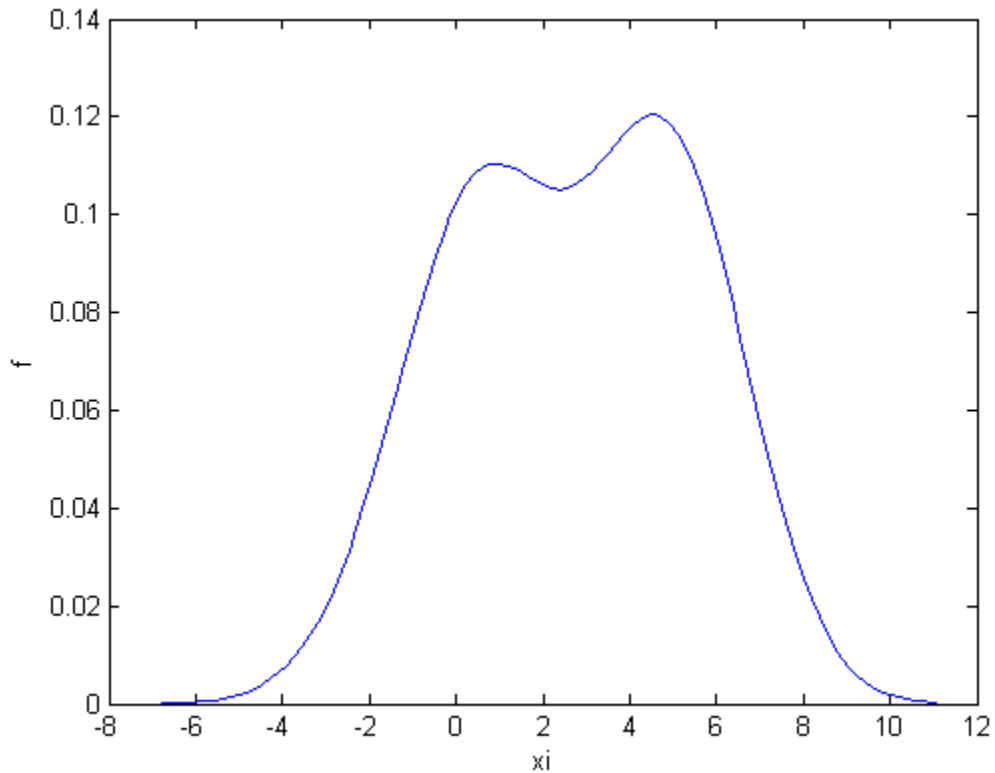
# ksdensity

---

```
rng('default') % for reproducibility  
x = [randn(30,1); 5+randn(30,1)];
```

Plot the estimated density.

```
[f,xi] = ksdensity(x);  
figure()  
plot(xi,f);
```



The density estimate shows the bimodality of the sample.

**Estimate Cumulative Distribution Function at Specified Values**

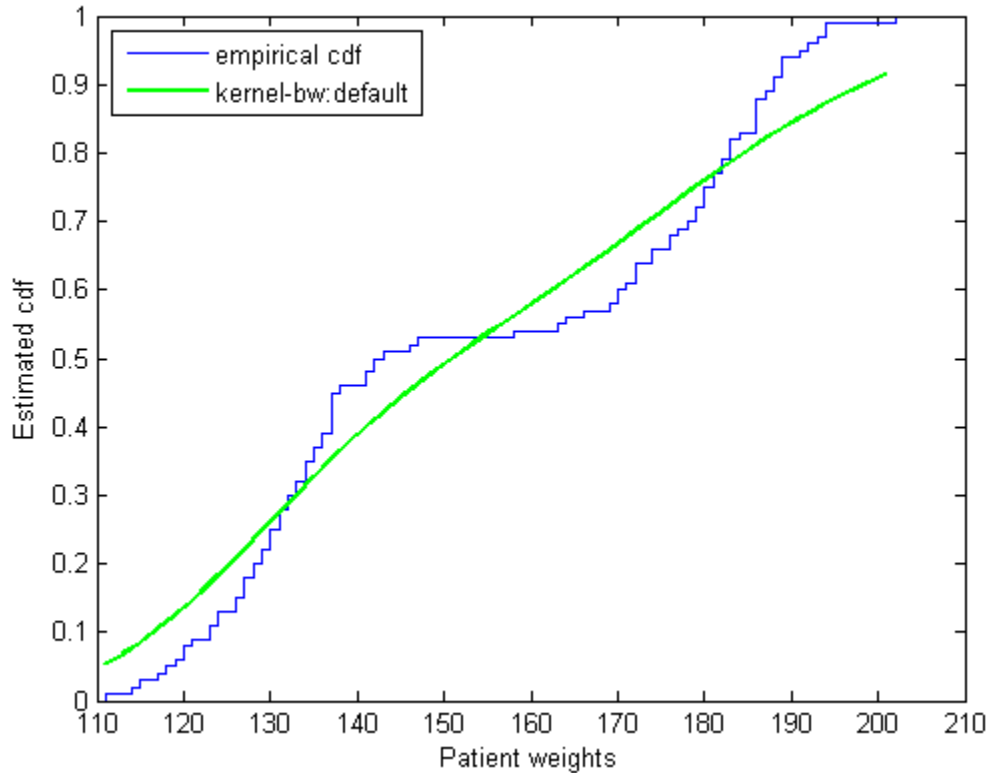
Load the sample data.

```
load hospital
```

Compute and plot the estimated cdf evaluated at a specified set of values.

```
pts = (min(hospital.Weight):2:max(hospital.Weight));  
figure()  
ecdf(hospital.Weight)  
hold on  
[f,xi,bw] = ksdensity(hospital.Weight,pts,'support','positive',...  
    'function','cdf');  
plot(xi,f,'-g','LineWidth',2)  
legend('empirical cdf','kernel-bw:default','Location','NorthWest')  
xlabel('Patient weights')  
ylabel('Estimated cdf')
```

# ksdensity



`ksdensity` seems to smooth the cumulative distribution function estimate too much. An estimate with a smaller bandwidth might produce a closer estimate to the empirical cumulative distribution function.

Return the bandwidth of the smoothing window.

`bw`

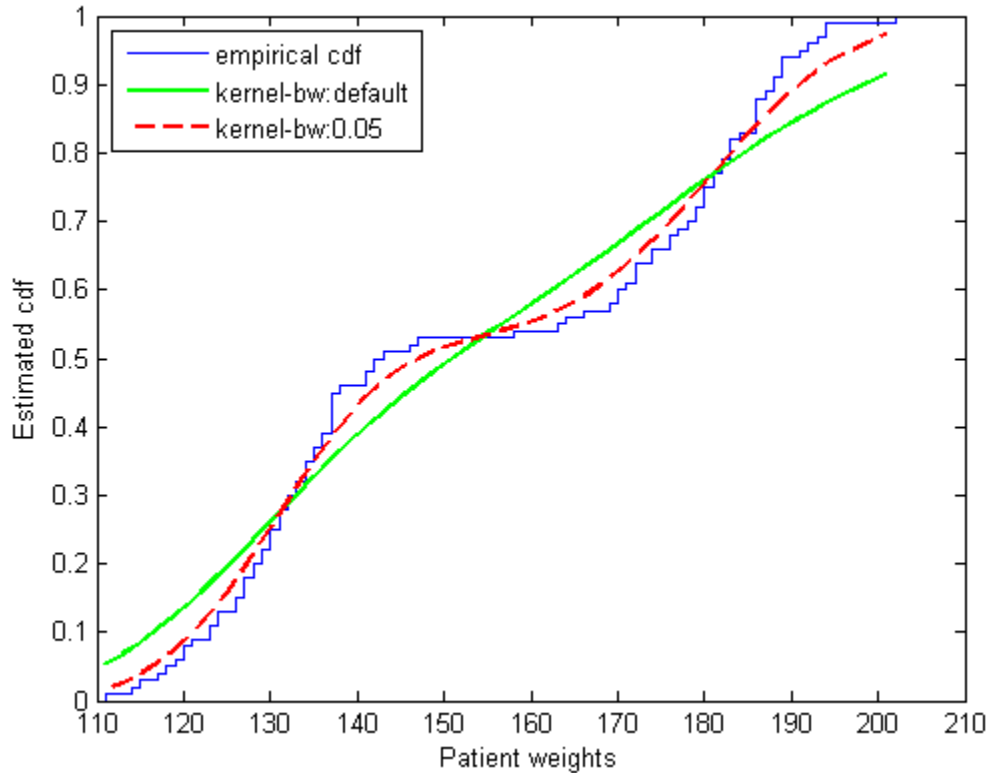
`bw =`

0.1070

Plot the cumulative distribution function estimate using a smaller bandwidth.

```
[f,xi] = ksdensity(hospital.Weight,pts,'support','positive',...  
    'function','cdf','bandwidth',0.05);  
plot(xi,f,'--r','LineWidth',2)  
legend('empirical cdf','kernel-bw:default','kernel-bw:0.05',...  
    'Location','NorthWest')  
hold off
```

# ksdensity



The `ksdensity` estimate with a smaller bandwidth matches the empirical cumulative distribution function better.

## Plot Estimated Cumulative Density Function for Given Number of Points

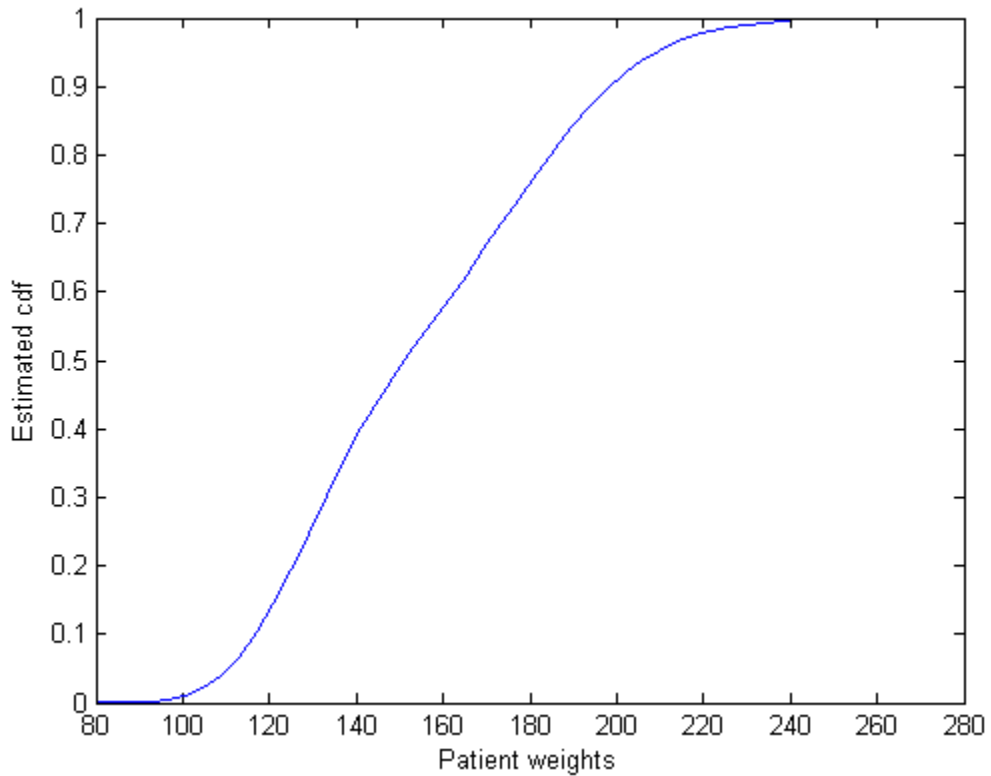
Load the sample data.

```
load hospital
```

Plot the estimated cdf evaluated at 50 equally spaced points.



```
figure()  
ksdensity(hospital.Weight,'support','positive','function','cdf',...  
  'npoints',50)  
xlabel('Patient weights')  
ylabel('Estimated cdf')
```



### Estimate Survivor and Cumulative Hazard for Censored Failure Data

Generate sample data from an exponential distribution with mean 3.

# ksdensity

---

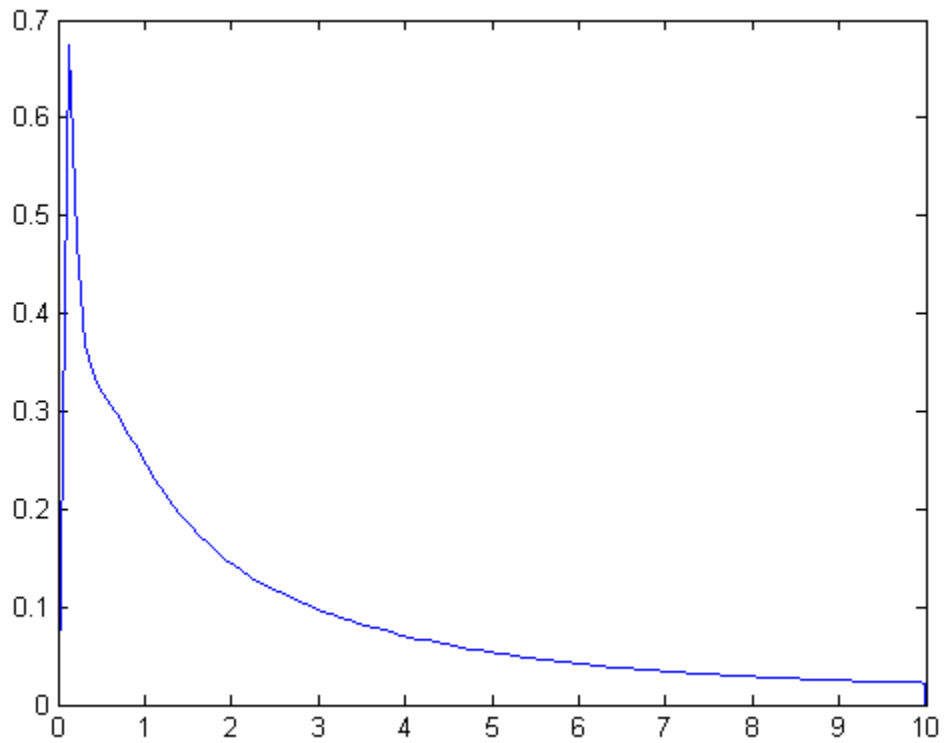
```
rng('default') % for reproducibility  
x = random('exp',3,100,1);
```

Create a logical vector that indicates censoring. Here, observations with lifetimes longer than 10 are censored.

```
T = 10;  
cens = (x>10);
```

Compute and plot the estimated density function.

```
figure()  
ksdensity(x,'support','positive','censoring',cens);
```

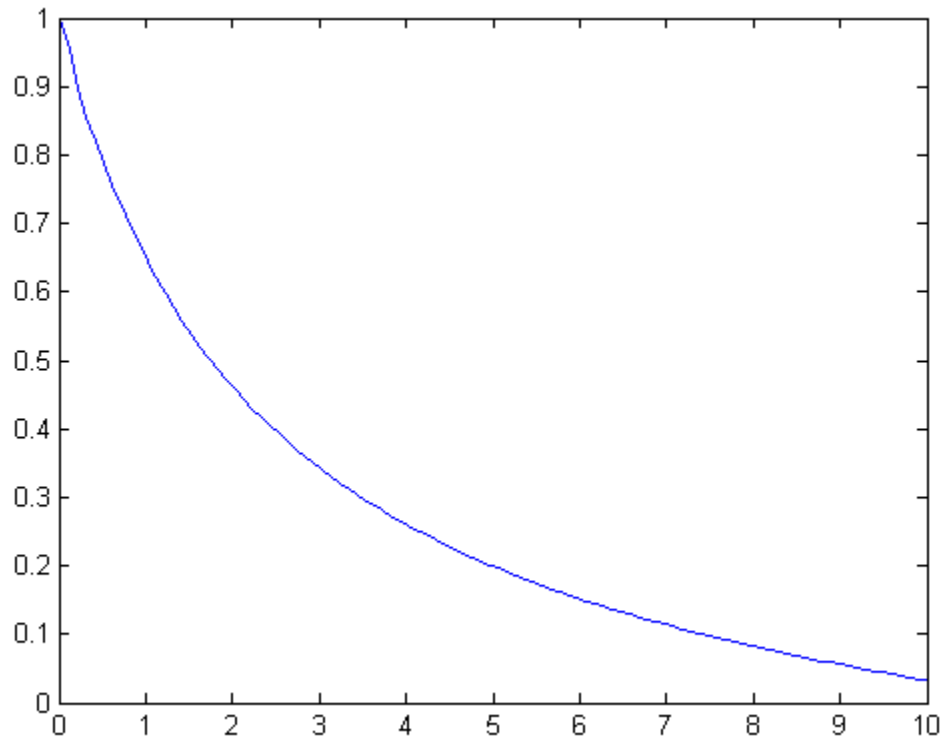


Compute and plot the survivor function.

```
figure()  
ksdensity(x,'support','positive','censoring',cens,...  
  'function','survivor');
```

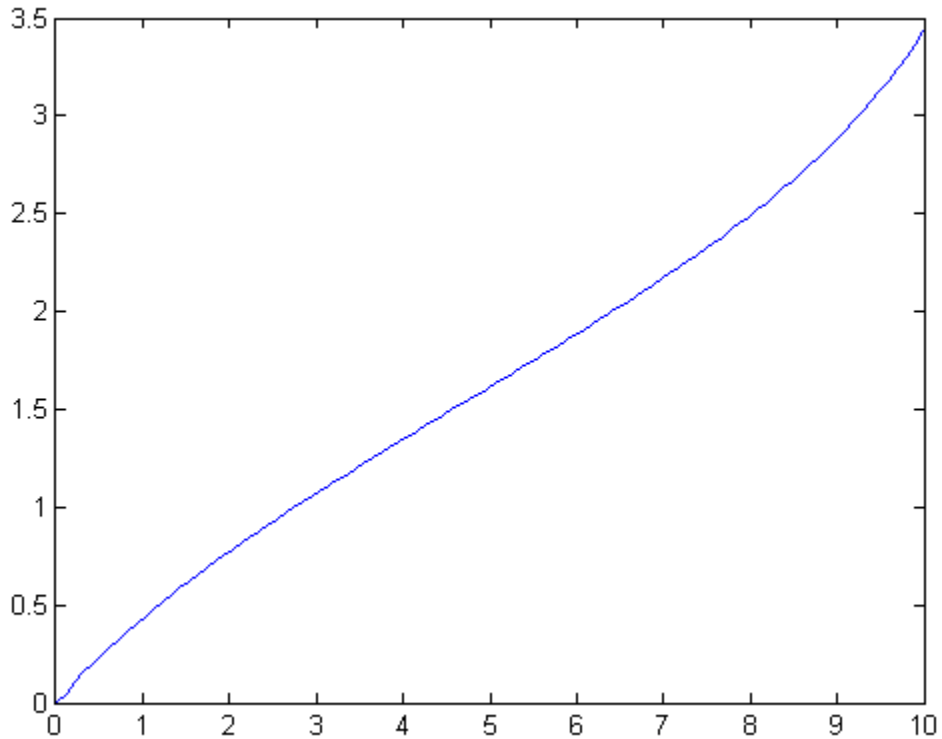
# ksdensity

---



Compute and plot the cumulative hazard function.

```
figure()  
ksdensity(x, 'support', 'positive', 'censoring', cens, ...  
  'function', 'cumhazard');
```



### Estimate Inverse Cumulative Distribution Function for Specified Probability Values

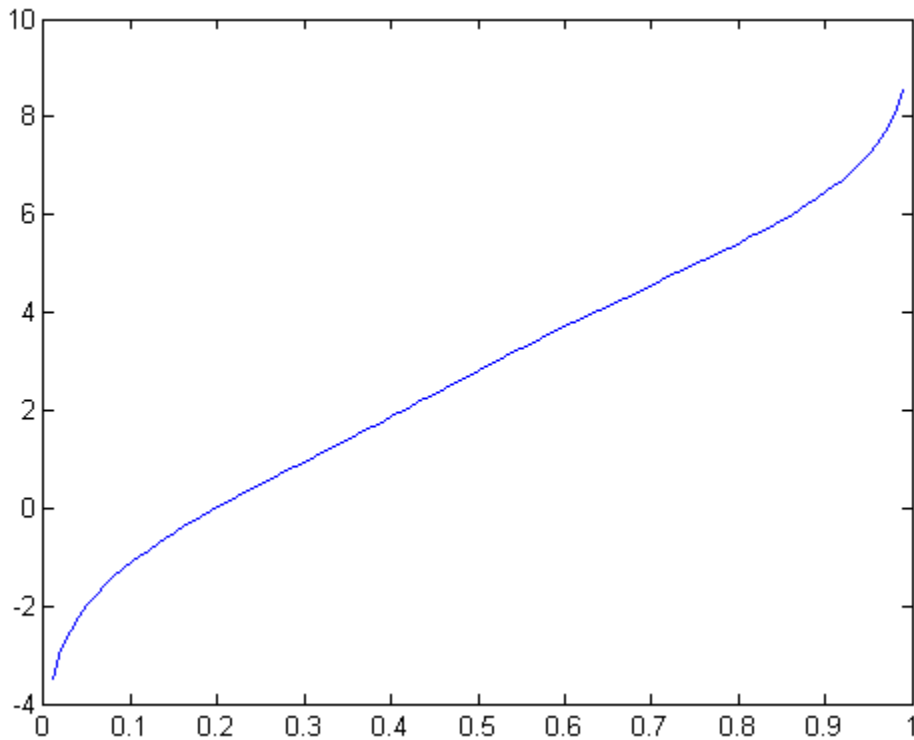
Generate a mixture of two normal distributions, and plot the estimated inverse cumulative distribution function at a specified set of probability values.

```
rng('default') % for reproducibility
x = [randn(30,1); 5+randn(30,1)];
pi = linspace(.01,.99,99);
```

# ksdensity

---

```
figure()  
ksdensity(x,pi,'function','icdf');
```



## Return Bandwidth of Smoothing Window

Generate a mixture of two normal distributions.

```
rng('default') % for reproducibility  
x = [randn(30,1); 5+randn(30,1)];
```

Return the bandwidth of the smoothing window for the probability density estimate.

```
[f,xi,bw] = ksdensity(x);  
bw
```

```
bw =
```

```
1.5141
```

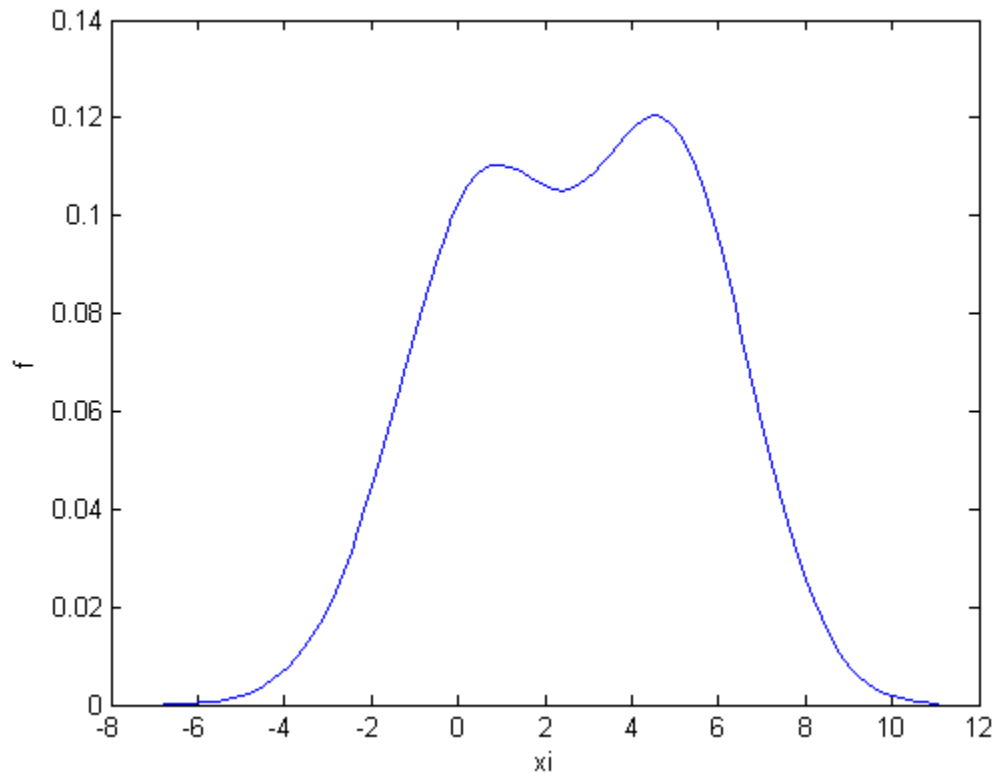
The default bandwidth is optimal for normal densities.

Plot the estimated density.

```
figure()  
plot(xi,f);  
xlabel('xi')  
ylabel('f')  
hold on
```

# ksdensity

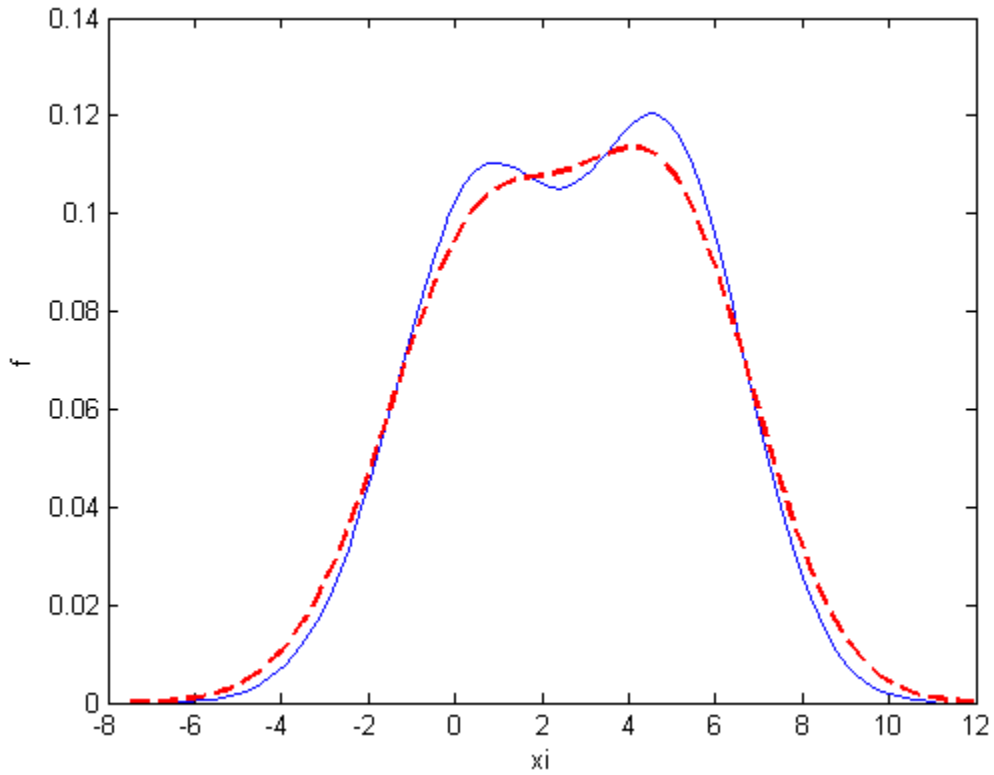
---



Plot the density using an increased bandwidth value.

```
[f,xi] = ksdensity(x,'width',1.8);  
plot(xi,f,'--r','LineWidth',1.5)
```



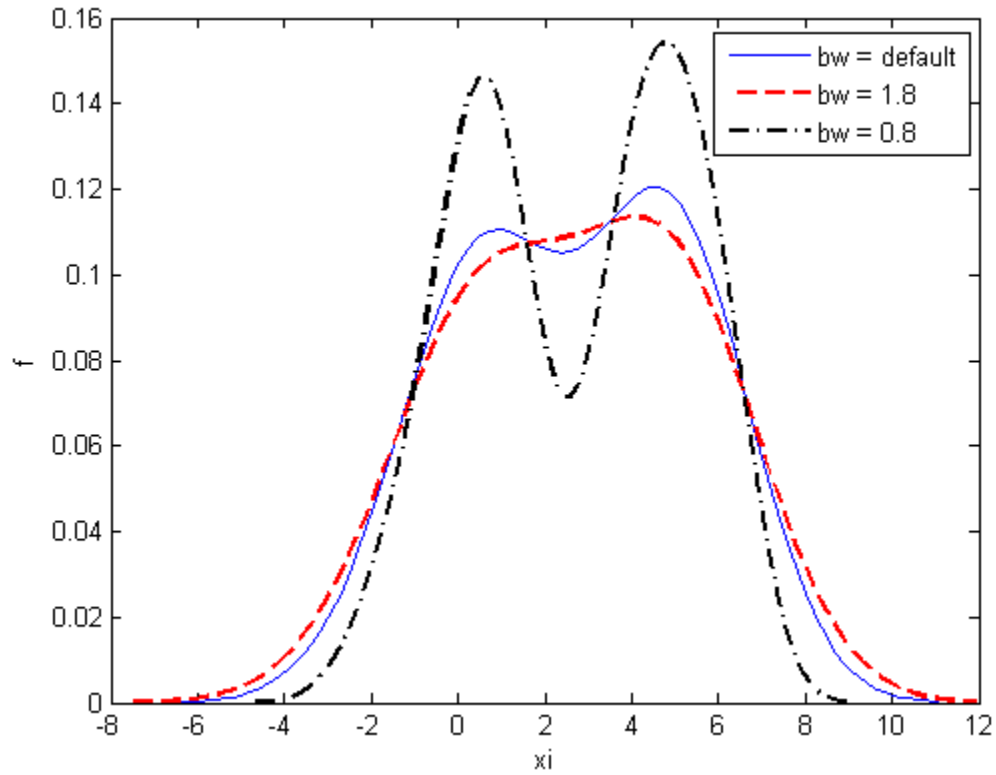


A higher bandwidth further smooths the density estimate, which might mask some characteristics of the distribution.

Now, plot the density using a decreased bandwidth value.

```
[f,xi] = ksdensity(x,'width',0.8);  
plot(xi,f,'-.k','LineWidth',1.5)  
legend('bw = default','bw = 1.8','bw = 0.8')  
hold off
```

# ksdensity



A smaller bandwidth smooths the density estimate less, which exaggerate some characteristics of the sample.

## References

[1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press Inc., 1997.

**See Also** `hist`

# kstest

---

**Purpose** One-sample Kolmogorov-Smirnov test

**Syntax**

```
h = kstest(x)
h = kstest(x,Name,Value)
[h,p] = kstest( ___ )
[h,p,ksstat,cv] = kstest( ___ )
```

**Description** `h = kstest(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a standard normal distribution, against the alternative that it does not come from such a distribution, using the one-sample Kolmogorov-Smirnov test. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, or 0 otherwise.

`h = kstest(x,Name,Value)` returns a test decision for the one-sample Kolmogorov-Smirnov test with additional options specified by one or more name-value pair arguments. For example, you can test for a distribution other than standard normal, change the significance level, or conduct a one-sided test.

`[h,p] = kstest( ___ )` also returns the  $p$ -value `p` of the hypothesis test, using any of the input arguments from the previous syntaxes.

`[h,p,ksstat,cv] = kstest( ___ )` also returns the value of the test statistic `ksstat` and the approximate critical value `cv` of the test.

## Input Arguments

### **x - Sample data**

vector

Sample data, specified as a vector.

### **Data Types**

single | double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding

value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**Example:** 'Tail', 'right', 'Alpha', 0.01 specifies a right-tailed hypothesis test at the 1% significance level.

### 'Alpha' - Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha', 0.01

### Data Types

single | double

### 'CDF' - cdf of hypothesized continuous distribution

matrix | probability distribution object

cdf of hypothesized continuous distribution, specified the comma-separated pair consisting of 'CDF' and either a two-column matrix or a continuous probability distribution object. When CDF is a matrix, column 1 contains a set of possible  $x$  values, and column 2 contains the corresponding hypothesized cumulative distribution function values  $G(x)$ . The calculation is most efficient if CDF is specified such that column 1 contains the values in the data vector  $x$ . If there are values in  $x$  not found in column 1 of CDF, `kstest` approximates  $G(x)$  by interpolation. All values in  $x$  must lie in the interval between the smallest and largest values in the first column of CDF. By default, `kstest` tests for a standard normal distribution.

The one-sample Kolmogorov-Smirnov test is only valid for continuous cumulative distribution functions, and requires CDF to be predetermined. The result is not accurate if CDF is estimated from the data. To test  $x$  against the normal, lognormal, extreme value, Weibull, or exponential distribution without specifying distribution parameters, use `lillietest` instead.

## Data Types

single | double

## 'Tail' - Type of alternative hypothesis

'unequal' (default) | 'larger' | 'smaller'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

- |           |  |
|-----------|--|
| 'unequal' | Test the alternative hypothesis that the cdf of the population from which $x$ is drawn is not equal to the cdf of the hypothesized distribution. |
| 'larger'  | Test the alternative hypothesis that the cdf of the population from which $x$ is drawn is greater than the cdf of the hypothesized distribution. |
| 'smaller' | Test the alternative hypothesis that the cdf of the population from which $x$ is drawn is less than the cdf of the hypothesized distribution.    |

If the values in the data vector  $x$  tend to be larger than expected from the hypothesized distribution, the empirical distribution function of  $x$  tends to be smaller, and vice versa.

**Example:** 'Tail', 'larger'

## Output Arguments

### **h** - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** - p-value

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

**ksstat - Test statistic**

nonnegative scalar value

Test statistic of the hypothesis test, returned as a nonnegative scalar value.

**cv - Critical value**

nonnegative scalar value

Critical value, returned as a nonnegative scalar value.

**Examples****Test for a Standard Normal Distribution**

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
test1 = grades(:,1);
```

Test the null hypothesis that the data comes from a normal distribution with a mean of 75 and a standard deviation of 10. Use these parameters to center and scale each element of the data vector since, by default, `kstest` tests for a standard normal distribution.

```
x = (test1-75)/10;  
h = kstest(x)
```

```
h =  
0
```

The returned value of  $h = 0$  indicates that `kstest` fails to reject the null hypothesis at the default 5% significance level.

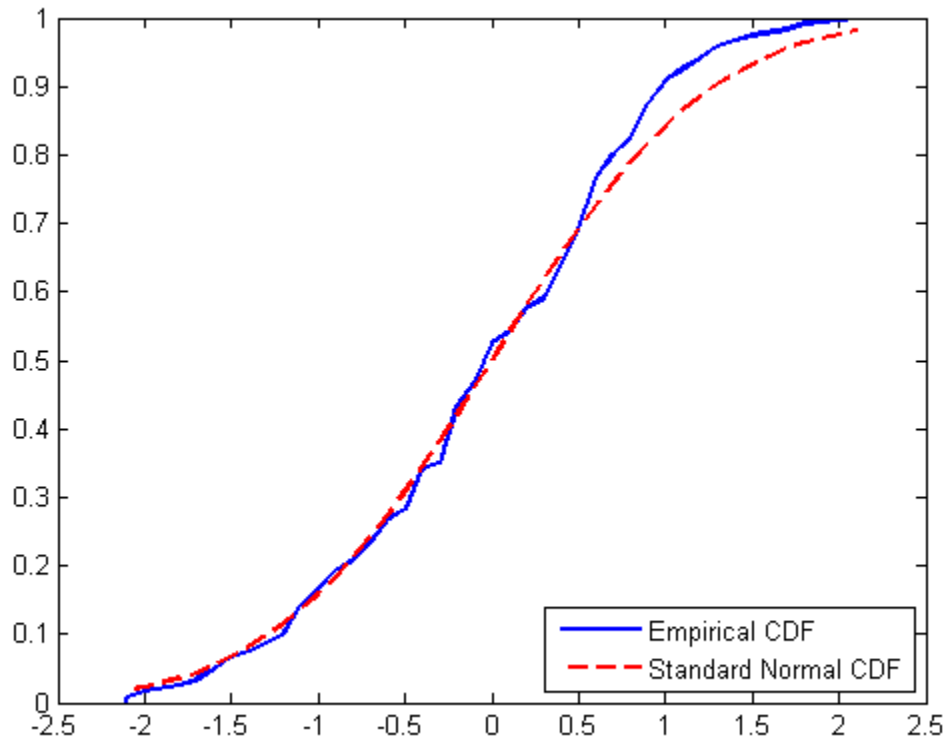
# kstest

---

Plot the empirical cumulative distribution function (cdf) and the standard normal cdf for a visual comparison.

```
[f,x_values] = ecdf(x);
F = plot(x_values,f);
set(F,'LineWidth',2);
hold on;
G = plot(x_values,normcdf(x_values,0,1),'r-');
set(G,'LineWidth',2);
legend([F G],...
       'Empirical CDF','Standard Normal CDF',...
       'Location','SE');
```





The plot shows the similarity between the empirical cdf of the centered and scaled data vector and the cdf of the standard normal distribution.

### **Specify the Hypothesized Distribution Using a Two-Column Matrix**

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Specify the hypothesized distribution as a two-column matrix. Column 1 contains the data vector  $x$ . Column 2 contains cdf values evaluated at each value in  $x$  for a hypothesized Student's  $t$  distribution with a location parameter of 75, a scale parameter of 10, and one degree of freedom.

```
test_cdf = [x,cdf('tlocation',x,75,10,1)];
```

Test if the data are from the hypothesized distribution.

```
h = kstest(x,'CDF',test_cdf)
```

```
h =  
1
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis at the default 5% significance level.

## **Specify the Hypothesized Distribution Using a Probability Distribution Object**

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Create a probability distribution object to test if the data comes from a Student's  $t$  distribution with a location parameter of 75, a scale parameter of 10, and one degree of freedom.

```
test_cdf = makedist('tlocation','mu',75,'sigma',10,'nu',1);
```

Test the null hypothesis that the data comes from the hypothesized distribution.

```
h = kstest(x,'CDF',test_cdf)
```

```
h =  
    1
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis at the default 5% significance level.

### Test the Hypothesis at Different Significance Levels

Load the sample data. Create a vector containing the first column of the students' exam grades.

```
load examgrades;  
test1 = grades(:,1);
```

Create a probability distribution object to test if the data comes from a Student's  $t$  distribution with a location parameter of 75, a scale parameter of 10, and one degree of freedom.

```
test_cdf = makedist('tlocationscale','mu',75,'sigma',10,'nu',1);
```

Test the null hypothesis that data comes from the hypothesized distribution at the 1% significance level.

```
[h,p] = kstest(x,'CDF',test_cdf,'Alpha',0.01)
```

```
h =  
    1  
  
p =  
    0.0021
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis at the 1% significance level.

### Conduct a One-Sided Hypothesis Test

Load the sample data. Create a vector containing the third column of the stock return data matrix.

# kstest

---

```
load stockreturns;  
x = stocks(:,3);
```

Test the null hypothesis that the data comes from a standard normal distribution, against the alternative hypothesis that the population cdf of the data is larger than the standard normal cdf.

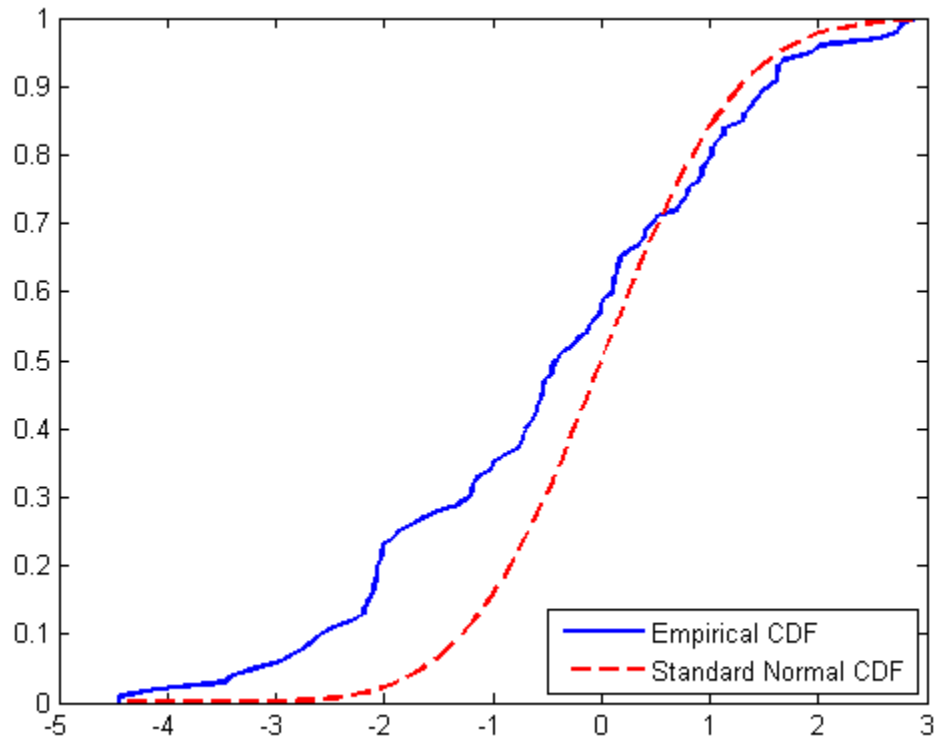
```
[h,p,k,c] = kstest(x, 'Tail', 'larger')
```

```
h =  
1  
p =  
5.0854e-05  
k =  
0.2197  
c =  
0.1207
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis in favor of the alternative hypothesis at the default 5% significance level.

Plot the empirical cdf and the standard normal cdf for a visual comparison.

```
[f,x_values] = ecdf(x);  
J = plot(x_values,f);  
hold on;  
K = plot(x_values,normcdf(x_values), 'r--');  
set(J, 'LineWidth', 2);  
set(K, 'LineWidth', 2);  
legend([J K], 'Empirical CDF', 'Standard Normal CDF', 'Location', 'SE');
```



The plot shows the difference between the empirical cdf of the data vector  $x$  and the cdf of the standard normal distribution.

## Definitions

### One-Sample Kolmogorov-Smirnov Test

The one-sample Kolmogorov-Smirnov test is a nonparametric test of the null hypothesis that the population cdf of the data is equal to the hypothesized cdf.

The two-sided test for “unequal” cdf functions tests the null hypothesis against the alternative that the population cdf of the data is not

equal to the hypothesized cdf. The test statistic is the maximum absolute difference between the empirical cdf calculated from  $x$  and the hypothesized cdf:

$$D^* = \max_x \left( \left| \hat{F}(x) - G(x) \right| \right),$$

where  $\hat{F}(x)$  is the empirical cdf and  $G(x)$  is the cdf of the hypothesized distribution.

The one-sided test for a “larger” cdf function tests the null hypothesis against the alternative that the population cdf of the data is greater than the hypothesized cdf. The test statistic is the maximum amount by which the empirical cdf calculated from  $x$  exceeds the hypothesized cdf:

$$D^* = \max_x \left( \hat{F}(x) - G(x) \right).$$

The one-sided test for a “smaller” cdf function tests the null hypothesis against the alternative that the population cdf of the data is less than the hypothesized cdf. The test statistic is the maximum amount by which the hypothesized cdf exceeds the empirical cdf calculated from  $x$ :

$$D^* = \max_x \left( G(x) - \hat{F}(x) \right).$$

`kstest` computes the critical value `cv` using an approximate formula or by interpolation in a table. The formula and table cover the range  $0.01 \leq \alpha \leq 0.2$  for two-sided tests and  $0.005 \leq \alpha \leq 0.1$  for one-sided tests. `cv` is returned as `NaN` if `alpha` is outside this range.

## Algorithms

`kstest` decides to reject the null hypothesis by comparing the  $p$ -value `p` with the significance level `Alpha`, not by comparing the test statistic `ksstat` with the critical value `cv`. Since `cv` is approximate, comparing `ksstat` with `cv` occasionally leads to a different conclusion than comparing `p` with `Alpha`.

## References

- [1] Massey, F. J. “The Kolmogorov-Smirnov Test for Goodness of Fit.” *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.
- [2] Miller, L. H. “Table of Percentage Points of Kolmogorov Statistics.” *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.
- [3] Marsaglia, G., W. Tsang, and J. Wang. “Evaluating Kolmogorov’s Distribution.” *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.

## See Also

`kstest2` | `lillietest` | `adtest`

# kstest2

---

**Purpose** Two-sample Kolmogorov-Smirnov test

**Syntax**

```
h = kstest2(x1,x2)
h = kstest2(x1,x2,Name,Value)
[h,p] = kstest2( ___ )
[h,p,ks2stat] = kstest2( ___ )
```

**Description** `h = kstest2(x1,x2)` returns a test decision for the null hypothesis that the data in vectors `x1` and `x2` are from the same continuous distribution, using the two-sample Kolmogorov-Smirnov test. The alternative hypothesis is that `x1` and `x2` are from different continuous distributions. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = kstest2(x1,x2,Name,Value)` returns a test decision for a two-sample Kolmogorov-Smirnov test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = kstest2( ___ )` also returns the asymptotic  $p$ -value `p`, using any of the input arguments from the previous syntaxes.

`[h,p,ks2stat] = kstest2( ___ )` also returns the test statistic `ks2stat`.

## Input Arguments

### **x1 - Sample data**

vector

Sample data from the first sample, specified as a vector. Data vectors `x1` and `x2` do not need to be the same size.

### **Data Types**

single | double

### **x2 - Sample data**

vector



Sample data from the second sample, specified as a vector. Data vectors `x1` and `x2` do not need to be the same size.

**Data Types**

single | double

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

**Example:**

**'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

**Data Types**

single | double

**'Tail' - Type of alternative hypothesis**

'unequal' (default) | 'larger' | 'smaller'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

- 'unequal' Test the alternative hypothesis that the empirical cdf of `x1` is unequal to the empirical cdf of `x2`.
- 'larger' Test the alternative hypothesis that the empirical cdf of `x1` is larger than the empirical cdf of `x2`.
- 'smaller' Test the alternative hypothesis that the empirical cdf of `x1` is smaller than the empirical cdf of `x2`.

If the data values in `x1` tend to be larger than those in `x2`, the empirical distribution function of `x1` tends to be smaller than that of `x2`, and vice versa.

**Example:** 'Tail', 'larger'

## Output Arguments

### **h - Hypothesis test result**

1 | 0

Hypothesis test result, returned as a logical value.

- If `h = 1`, this indicates the rejection of the null hypothesis at the Alpha significance level.
- If `h = 0`, this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p - Asymptotic p-value**

scalar value in the range (0,1)

Asymptotic *p*-value of the test, returned as a scalar value in the range (0,1). *p* is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. The asymptotic *p*-value becomes very accurate for large sample sizes, and is believed to be reasonably accurate for sample sizes `n1` and `n2`, such that  $(n1*n2)/(n1 + n2) \geq 4$ .

### **ks2stat - Test statistic**

nonnegative scalar value

Test statistic, returned as a nonnegative scalar value.

## Examples

### **Test Two Samples for the Same Distribution**

Generate sample data from two different Weibull distributions.

```
rng(1);          % For reproducibility
x1 = wblrnd(1,1,1,50);
x2 = wblrnd(1.2,2,1,50);
```

Test the null hypothesis that data in vectors `x1` and `x2` comes from populations with the same distribution.

```
h = kstest2(x1,x2)
```

```
h =  
1
```

The returned value of `h = 1` indicates that `kstest` rejects the null hypothesis at the default 5% significance level.

### **Test the Hypothesis at Different Significance Levels**

Generate sample data from two different Weibull distributions.

```
rng(1);      % For reproducibility  
x1 = wblrnd(1,1,1,50);  
x2 = wblrnd(1.2,2,1,50);
```

Test the null hypothesis that data vectors `x1` and `x2` are from populations with the same distribution at the 1% significance level.

```
[h,p] = kstest2(x1,x2,'Alpha',0.01)
```

```
h =  
0  
p =  
0.0317
```

The returned value of `h = 0` indicates that `kstest` does not reject the null hypothesis at the 1% significance level.

### **One-Sided Hypothesis Test**

Generate sample data from two different Weibull distributions.

```
rng(1);      % For reproducibility  
x1 = wblrnd(1,1,1,50);  
x2 = wblrnd(1.2,2,1,50);
```

Test the null hypothesis that data in vectors  $x_1$  and  $x_2$  comes from populations with the same distribution, against the alternative hypothesis that the cdf of the distribution of  $x_1$  is larger than the cdf of the distribution of  $x_2$ .

```
[h,p,k] = kstest2(x1,x2,'Tail','larger')
```

```
h =  
  1  
p =  
  0.0158  
k =  
  0.2800
```

The returned value of  $h = 1$  indicates that `kstest` rejects the null hypothesis, in favor of the alternative hypothesis that the cdf of the distribution of  $x_1$  is larger than the cdf of the distribution of  $x_2$ , at the default 5% significance level. The returned value of  $k$  is the test statistic for the two-sample Kolmogorov-Smirnov test.

## Definitions

### Two-Sample Kolmogorov-Smirnov Test

The two-sample Kolmogorov-Smirnov test is a nonparametric hypothesis test that evaluates the difference between the cdfs of the distributions of the two sample data vectors over the range of  $x$  in each data set.

The two-sided test uses the maximum absolute difference between the cdfs of the distributions of the two data vectors. The test statistic is

$$D^* = \max_x (|F_1(x) - F_2(x)|),$$

where  $\hat{F}_1(x)$  is the proportion of  $x_1$  values less than or equal to  $x$  and  $\hat{F}_2(x)$  is the proportion of  $x_2$  values less than or equal to  $x$ .

The one-sided test uses the actual value of the difference between the cdfs of the distributions of the two data vectors rather than the absolute value. The test statistic is

$$D^* = \max_x (F_1(x) - F_2(x)).$$

## Algorithms

In `kstest2`, the decision to reject the null hypothesis is based on comparing the  $p$ -value `p` with the significance level `Alpha`, not by comparing the test statistic `ks2stat` with a critical value.

## References

- [1] Massey, F. J. “The Kolmogorov-Smirnov Test for Goodness of Fit.” *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.
- [2] Miller, L. H. “Table of Percentage Points of Kolmogorov Statistics.” *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.
- [3] Marsaglia, G., W. Tsang, and J. Wang. “Evaluating Kolmogorov’s Distribution.” *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.

## See Also

`kstest` | `lillietest` | `adtest`

# kurtosis

---

**Purpose** Kurtosis

**Syntax**  
`k = kurtosis(X)`  
`k = kurtosis(X,flag)`  
`k = kurtosis(X,flag,dim)`

**Description** `k = kurtosis(X)` returns the sample kurtosis of  $X$ . For vectors, `kurtosis(x)` is the kurtosis of the elements in the vector  $x$ . For matrices `kurtosis(X)` returns the sample kurtosis for each column of  $X$ . For  $N$ -dimensional arrays, `kurtosis` operates along the first nonsingleton dimension of  $X$ .

`k = kurtosis(X,flag)` specifies whether to correct for bias (`flag` is 0) or not (`flag` is 1, the default). When  $X$  represents a sample from a population, the kurtosis of  $X$  is biased, that is, it will tend to differ from the population kurtosis by a systematic amount that depends on the size of the sample. You can set `flag` to 0 to correct for this systematic bias.

`k = kurtosis(X,flag,dim)` takes the kurtosis along dimension `dim` of  $X$ .

`kurtosis` treats NaNs as missing values and removes them.

**Algorithms** Kurtosis is a measure of how outlier-prone a distribution is. The kurtosis of the normal distribution is 3. Distributions that are more outlier-prone than the normal distribution have kurtosis greater than 3; distributions that are less outlier-prone have kurtosis less than 3.

The kurtosis of a distribution is defined as

$$k = \frac{E(x - \mu)^4}{\sigma^4}$$

where  $\mu$  is the mean of  $x$ ,  $\sigma$  is the standard deviation of  $x$ , and  $E(t)$  represents the expected value of the quantity  $t$ . `kurtosis` computes a sample version of this population value.

**Note** Some definitions of kurtosis subtract 3 from the computed value, so that the normal distribution has kurtosis of 0. The kurtosis function does not use this convention.

When you set `flag` to 1, the following equation applies:

$$k_1 = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^4}{\left( \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^2}$$

When you set `flag` to 0, the following equation applies:

$$k_0 = \frac{n-1}{(n-2)(n-3)} ((n+1)k_1 - 3(n-1)) + 3$$

This bias-corrected formula requires that `X` contain at least four elements.

## Examples

```
X = randn([5 4])
X =
  1.1650  1.6961  -1.4462  -0.3600
  0.6268  0.0591  -0.7012  -0.1356
  0.0751  1.7971  1.2460  -1.3493
  0.3516  0.2641  -0.6390  -1.2704
 -0.6965  0.8717  0.5774  0.9846

k = kurtosis(X)
k =
  2.1658  1.2967  1.6378  1.9589
```

## See Also

[mean](#) | [moment](#) | [skewness](#) | [std](#) | [var](#)

# categorical.labels property

---

**Purpose** Text labels for levels

**Description** Text labels for levels. Access labels with `getLabels`.



---

<b>Purpose</b>	Regularized least-squares regression using lasso or elastic net algorithms
<b>Syntax</b>	<pre>B = lasso(X,Y) [B,FitInfo] = lasso(X,Y) [B,FitInfo] = lasso(X,Y,Name,Value)</pre>
<b>Description</b>	<p><code>B = lasso(X,Y)</code> returns fitted least-squares regression coefficients for a set of regularization coefficients <code>Lambda</code>.</p> <p><code>[B,FitInfo] = lasso(X,Y)</code> returns a structure containing information about the fits.</p> <p><code>[B,FitInfo] = lasso(X,Y,Name,Value)</code> fits regularized regressions with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Input Arguments</b>	<p><b>X</b></p> <p>Numeric matrix with <math>n</math> rows and <math>p</math> columns. Each row represents one observation, and each column represents one predictor (variable).</p> <p><b>Y</b></p> <p>Numeric vector of length <math>n</math>, where <math>n</math> is the number of rows of <math>X</math>. <math>Y(i)</math> is the response to row <math>i</math> of <math>X</math>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>'Alpha'</b></p> <p>Scalar value from 0 to 1 (excluding 0) representing the weight of lasso (<math>L^1</math>) versus ridge (<math>L^2</math>) optimization. <code>Alpha = 1</code> represents lasso regression, <code>Alpha</code> close to 0 approaches ridge regression, and other</p>

values represent elastic net optimization. See “Definitions” on page 21-1645.

**Default:** 1

## **‘CV’**

Method `lasso` uses to estimate mean squared error:

- `K`, a positive integer — `lasso` uses `K`-fold cross validation.
- `cvp`, a `cvpartition` object — `lasso` uses the cross-validation method expressed in `cvp`. You cannot use a 'leaveout' partition with `lasso`.
- 'resubstitution' — `lasso` uses `X` and `Y` to fit the model and to estimate the mean squared error, without cross validation.

**Default:** 'resubstitution'

## **‘DFmax’**

Maximum number of nonzero coefficients in the model. `lasso` returns results only for `Lambda` values that satisfy this criterion.

**Default:** Inf

## **‘Lambda’**

Vector of nonnegative `Lambda` values. See “Definitions” on page 21-1645.

- If you do not supply `Lambda`, `lasso` calculates the largest value of `Lambda` that gives a nonnull model. In this case, `LambdaRatio` gives the ratio of the smallest to the largest value of the sequence, and `NumLambda` gives the length of the vector.
- If you supply `Lambda`, `lasso` ignores `LambdaRatio` and `NumLambda`.

**Default:** Geometric sequence of `NumLambda` values, the largest just sufficient to produce  $B = 0$

## **‘LambdaRatio’**

Positive scalar, the ratio of the smallest to the largest `Lambda` value when you do not set `Lambda`.

If you set `LambdaRatio = 0`, `lasso` generates a default sequence of `Lambda` values, and replaces the smallest one with 0.

**Default:** 1e-4

### **'MCReps'**

Positive integer, the number of Monte Carlo repetitions for cross validation.

- If CV is 'resubstitution' or a `cvpartition` of type 'resubstitution', `MCReps` must be 1.
- If CV is a `cvpartition` of type 'holdout', `MCReps` must be greater than 1.

**Default:** 1

### **'NumLambda'**

Positive integer, the number of `Lambda` values `lasso` uses when you do not set `Lambda`. `lasso` can return fewer than `NumLambda` fits if the residual error of the fits drops below a threshold fraction of the variance of `Y`.

**Default:** 100

### **'Options'**

Structure that specifies whether to cross validate in parallel, and specifies the random stream or streams. Create the `Options` structure with `statset`. Option fields:

- `UseParallel` — Set to `true` to compute in parallel. Default is `false`.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. To compute reproducibly, set `Streams` to a type allowing substreams: 'mlfg6331\_64' or 'mrg32k3a'. Default is `false`.

- **Streams** — A `RandStream` object or cell array consisting of one such object. If you do not specify `Streams`, `lasso` uses the default stream.

### **'PredictorNames'**

Cell array of strings representing names of the predictor variables, in the order in which they appear in `X`.

**Default:** {}

### **'RelTol'**

Convergence threshold for the coordinate descent algorithm (see Friedman, Tibshirani, and Hastie [3]). The algorithm terminates when successive estimates of the coefficient vector differ in the  $L^2$  norm by a relative amount less than `RelTol`.

**Default:** 1e-4

### **'Standardize'**

Boolean value specifying whether `lasso` scales `X` before fitting the models.

**Default:** true

### **'Weights'**

Observation weights, a nonnegative vector of length `n`, where `n` is the number of rows of `X`. `lasso` scales `Weights` to sum to 1.

**Default:** `1/n * ones(n,1)`

## **Output Arguments**

### **B**

Fitted coefficients, a `p`-by-`L` matrix, where `p` is the number of predictors (columns) in `X`, and `L` is the number of `Lambda` values.

### **FitInfo**

Structure containing information about the model fits.

Field in FitInfo	Description
Intercept	Intercept term $\beta_0$ for each linear model, a 1-by-L vector
Lambda	Lambda parameters in ascending order, a 1-by-L vector
Alpha	Value of Alpha parameter, a scalar
DF	Number of nonzero coefficients in B for each value of Lambda, a 1-by-L vector
MSE	Mean squared error (MSE), a 1-by-L vector

If you set the CV name-value pair to cross validate, the FitInfo structure contains additional fields.

Field in FitInfo	Description
SE	The standard error of MSE for each Lambda, as calculated during cross validation, a 1-by-L vector
LambdaMinMSE	The Lambda value with minimum MSE, a scalar
Lambda1SE	The largest Lambda such that MSE is within one standard error of the minimum, a scalar
IndexMinMSE	The index of Lambda with value LambdaMinMSE, a scalar
Index1SE	The index of Lambda with value Lambda1SE, a scalar

## Definitions

### Lasso

For a given value of  $\lambda$ , a nonnegative parameter, lasso solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda \sum_{j=1}^p |\beta_j| \right),$$

where

- $N$  is the number of observations.
- $y_i$  is the response at observation  $i$ .
- $x_i$  is data, a vector of  $p$  values at observation  $i$ .
- $\lambda$  is a nonnegative regularization parameter corresponding to one value of **Lambda**.
- The parameters  $\beta_0$  and  $\beta$  are scalar and  $p$ -vector respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

## Elastic Net

For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{2N} \sum_{i=1}^N (y_i - \beta_0 - x_i^T \beta)^2 + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right).$$

Elastic net is the same as lasso when  $\alpha = 1$ . As  $\alpha$  shrinks toward 0, elastic net approaches **ridge** regression. For other values of  $\alpha$ , the penalty term  $P_\alpha(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ .

## Examples

### Remove Redundant Predictors

Construct a data set with redundant predictors, and identify those predictors using cross-validated lasso.

Create a matrix  $X$  of 100 five-dimensional normal variables and a response vector  $Y$  from just two components of  $X$ , with small added noise.

```
X = randn(100,5);  
r = [0;2;0;-3;0]; % only two nonzero coefficients  
Y = X*r + randn(100,1)*.1; % small added noise
```

Construct the default lasso fit.

```
B = lasso(X,Y);
```

Find the coefficient vector for the 25th value in  $B$ .

```
B(:,25)
```

```
ans =
```

```
    0  
  1.6093  
    0  
 -2.5865  
    0
```

lasso identifies and removes the redundant predictors.

### Plot a Regularized Fit with Cross Validation

Visually examine the cross-validated error of various levels of regularization.

Load the acetylene data and prepare the data with interactions for fitting.

```
load acetylene  
Xs = [x1 x2 x3];
```

# lasso

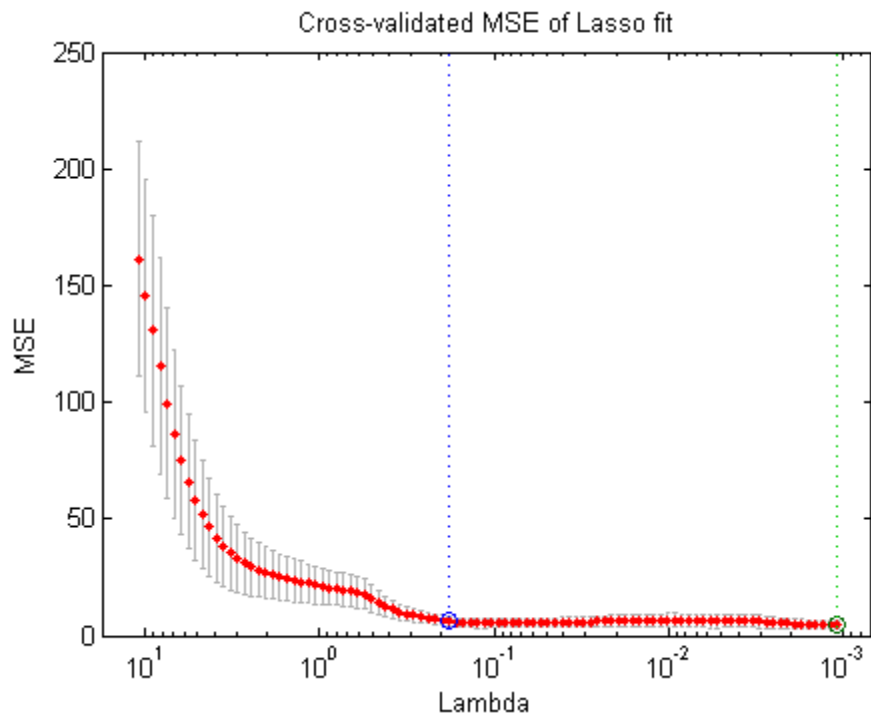
```
X = x2fx(Xs,'interaction');  
X(:,1) = []; % No constant term
```

Construct the lasso fit using ten-fold cross validation. Include the FitInfo output so you can plot the result.

```
[B FitInfo] = lasso(X,y,'CV',10);
```

Plot the cross-validated fits.

```
lassoPlot(B,FitInfo,'PlotType','CV');
```





**References**

- [1] Tibshirani, R. *Regression shrinkage and selection via the lasso*. Journal of the Royal Statistical Society, Series B, Vol 58, No. 1, pp. 267–288, 1996.
- [2] Zou, H. and T. Hastie. *Regularization and variable selection via the elastic net*. Journal of the Royal Statistical Society, Series B, Vol. 67, No. 2, pp. 301–320, 2005.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. *Regularization paths for generalized linear models via coordinate descent*. Journal of Statistical Software, Vol 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>
- [4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition. Springer, New York, 2008.

**See Also**

lassoPlot | ridge

**How To**

- “Lasso and Elastic Net” on page 9-132

# lassoglm

---

**Purpose** Lasso or elastic net regularization for generalized linear model regression

**Syntax**

```
B = lassoglm(X,Y)
[B,FitInfo] = lassoglm(X,Y)
[B,FitInfo] = lassoglm(X,Y,distr)
[B,FitInfo] = lassoglm(X,Y,distr,Name,Value)
```

**Description** `B = lassoglm(X,Y)` returns penalized maximum-likelihood fitted coefficients for a generalized linear model of the response `Y` to the data matrix `X`. `Y` are assumed to have a Gaussian probability distribution.

`[B,FitInfo] = lassoglm(X,Y)` returns a structure containing information about the fits.

`[B,FitInfo] = lassoglm(X,Y,distr)` fits the model using the probability distribution type for `Y` as specified in `distr`.

`[B,FitInfo] = lassoglm(X,Y,distr,Name,Value)` fits regularized generalized linear regressions with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **X**

Numeric matrix with `n` rows and `p` columns. Each row represents one observation, and each column represents one predictor (variable).

### **Y**

When `distr` is not `'binomial'`, `Y` is a numeric vector of length `n`, where `n` is the number of rows of `X`. `Y(i)` is the response to row `i` of `X`.

When `distr` is `'binomial'`, `Y` is either a:

- Numeric vector of length `n`, where each entry represents success (1) or failure (0)
- Logical vector of length `n`, where each entry represents success or failure

- Two column numeric matrix, where the first column contains the number of successes for each observation, and the second column contains the total number of trials

**distr**

Distributional family for the nonsystematic variation in the responses, a string. Choices:

- 'normal'
- 'binomial'
- 'poisson'
- 'gamma'
- 'inverse gaussian'

By default, `lassoglm` uses the canonical link function corresponding to `distr`. Specify another link function using the 'link' name-value pair.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**'Alpha'**

Scalar value from 0 to 1 (excluding 0) representing the weight of lasso ( $L^1$ ) versus ridge ( $L^2$ ) optimization. `Alpha = 1` represents lasso regression, and other values represent elastic net optimization. `Alpha` close to 0 approaches ridge regression. See “Definitions” on page 21-1645.

**Default:** 1

**'CV'**

Method `lassoglm` uses to estimate deviance:

- `K`, a positive integer — `lassoglm` uses `K`-fold cross validation.
- `cvp`, a `cvpartition` object — `lassoglm` uses the cross-validation method expressed in `cvp`. You cannot use a 'leaveout' partition with `lassoglm`.
- 'resubstitution' — `lassoglm` uses `X` and `Y` to fit the model and to estimate the deviance, without cross validation.

**Default:** 'resubstitution'

## 'DFmax'

Maximum number of nonzero coefficients in the model. `lassoglm` returns results for `Lambda` values that satisfy this criterion.

**Default:** Inf

## 'Lambda'

Vector of nonnegative `Lambda` values. See "Lasso" on page 21-1658.

- If you do not supply `Lambda`, `lassoglm` estimates the largest value of `Lambda` that gives a nonnull model. In this case, `LambdaRatio` gives the ratio of the smallest to the largest value of the sequence, and `NumLambda` gives the length of the vector.
- If you supply `Lambda`, `lassoglm` ignores `LambdaRatio` and `NumLambda`.

**Default:** Geometric sequence of `NumLambda` values, the largest just sufficient to produce  $B = 0$

## 'LambdaRatio'

Positive scalar, the ratio of the smallest to the largest `Lambda` value when you do not explicitly set `Lambda`.

If you set `LambdaRatio = 0`, `lassoglm` generates a default sequence of `Lambda` values, and replaces the smallest one with 0.

**Default:** `1e-4`

### 'Link'

Specify the mapping between the mean  $\mu$  of the response and the linear predictor  $Xb$ .

Value	Description
'comploglog'	$\log(-\log((1-\mu))) = Xb$
'identity', default for the distribution 'normal'	$\mu = Xb$
'log', default for the distribution 'poisson'	$\log(\mu) = Xb$
'logit', default for the distribution 'binomial'	$\log(\mu/(1-\mu)) = Xb$
'loglog'	$\log(-\log(\mu)) = Xb$
'probit'	$\Phi^{-1}(\mu) = Xb$ , where $\Phi$ is the normal (Gaussian) CDF function
'reciprocal', default for the distribution 'gamma'	$\mu^{-1} = Xb$

Value	Description
$p$ (a number), default for the distribution 'inverse gaussian' (with $p = -2$ )	$\mu^p = Xb$
Cell array of the form {FL FD FI}, containing three function handles, created using @, that define the link (FL), the derivative of the link (FD), and the inverse link (FI). Equivalently, can be a structure of function handles with field Link containing FL, field Derivative containing FD, and field Inverse containing FI.	User-specified link function (see “Custom Link Function” on page 10-16)

### **'MCReps'**

Positive integer, the number of Monte Carlo repetitions for cross validation.

- If CV is 'resubstitution' or a cvpartition of type 'resubstitution', MCReps must be 1.
- If CV is a cvpartition of type 'holdout', MCReps must be greater than 1.

**Default:** 1

### **'NumLambda'**

Positive integer, the number of Lambda values lassoglm uses when you do not set Lambda. lassoglm can return fewer than NumLambda fits if the

deviance of the fits drops below a threshold fraction of the null deviance (deviance of the fit without any predictors  $X$ ).

**Default:** 100

### **'Offset'**

Numeric vector with the same number of rows as  $X$ . `lassoglm` uses `Offset` as an additional predictor variable, but keeps its coefficient value fixed at 1.0.

### **'Options'**

Structure that specifies whether to cross validate in parallel, and specifies the random stream or streams. Create the `Options` structure with `statset`. Option fields:

- `UseParallel` — Set to `true` to compute in parallel. Default is `false`.
- `UseSubstreams` — Set to `true` to compute in parallel in a reproducible fashion. To compute reproducibly, set `Streams` to a type allowing substreams: `'mlfg6331_64'` or `'mrg32k3a'`. Default is `false`.
- `Streams` — `RandStream` object or cell array consisting of one such object. If you do not specify `Streams`, `lassoglm` uses the default stream.

### **'PredictorNames'**

Cell array of strings representing names of the predictor variables, in the order in which they appear in  $X$ .

**Default:** {}

### **'RelTol'**

Convergence threshold for the coordinate descent algorithm (see Friedman, Tibshirani, and Hastie [3]). The algorithm terminates when successive estimates of the coefficient vector differ in the  $L^2$  norm by a relative amount less than `RelTol`.

# lassoglm

---

**Default:** 1e-4

## 'Standardize'

Boolean value specifying whether `lassoglm` scales X before fitting the models.

**Default:** true

## 'Weights'

Observation weights, a nonnegative vector of length n, where n is the number of rows of X. At least two values must be positive.

**Default:**  $1/n * \text{ones}(n,1)$

## Output Arguments

### B

Fitted coefficients, a p-by-L matrix, where p is the number of predictors (columns) in X, and L is the number of Lambda values.

### FitInfo

Structure containing information about the model fits.

Field in FitInfo	Description
Alpha	Value of Alpha parameter, a scalar.
Deviance	Deviance of the fitted model for each value of Lambda, a 1-by-L vector. If cross validation was performed, the values for Deviance represent the estimated expected deviance of the model applied to new data, as calculated by cross validation. Otherwise, Deviance is the deviance of the fitted model applied to the data used to perform the fit.
DF	Number of nonzero coefficients in B for each Lambda value, a 1-by-L vector.



Field in FitInfo	Description
Intercept	Intercept term $\beta_0$ for each linear model, a 1-by-L vector.
Lambda	Lambda parameters in ascending order, a 1-by-L vector.

If you set the CV name-value pair to cross validate, the `FitInfo` structure contains additional fields.

Field in FitInfo	Description
IndexMinDeviance	Index of Lambda with value <code>LambdaMinDeviance</code> , a scalar.
Index1SE	Index of Lambda with value <code>Lambda1SE</code> , a scalar.
LambdaMinDeviance	Lambda value with minimum expected deviance, as calculated by cross validation, a scalar.
Lambda1SE	Largest Lambda such that Deviance is within one standard error of the minimum, a scalar.
SE	Standard error of Deviance for each Lambda, as calculated during cross validation, a 1-by-L vector.

## Definitions

### Link Function

A link function  $f(\mu)$  maps a distribution with mean  $\mu$  to a linear model with data  $X$  and coefficient vector  $b$  using the formula

$$f(\mu) = Xb.$$

Find the formulas for the link functions in the `Link` name-value pair description. Here, “typical” means a link function that is typically used for the listed distribution.

# lassoglm

Distributional Family	Link Function (typical, {default})
'normal'	{'identity'}
'binomial'	'comploglog', 'loglog', 'probit', {'logit'}
'poisson'	{'log'}
'gamma'	{'reciprocal'}
'inverse gaussian'	{-2}

## Lasso

For a nonnegative value of  $\lambda$ , lasso solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda \sum_{j=1}^p |\beta_j| \right),$$

where

- Deviance is the deviance of the model fit to the responses using intercept  $\beta_0$  and predictor coefficients  $\beta$ . The formula for Deviance depends on the `distr` parameter you supply to `lassoglm`. Minimizing the  $\lambda$ -penalized deviance is equivalent to maximizing the  $\lambda$ -penalized log likelihood.
- $N$  is the number of observations.
- $\lambda$  is a nonnegative regularization parameter corresponding to one value of `Lambda`.
- Parameters  $\beta_0$  and  $\beta$  are scalar and  $p$ -vector respectively.

As  $\lambda$  increases, the number of nonzero components of  $\beta$  decreases.

The lasso problem involves the  $L^1$  norm of  $\beta$ , as contrasted with the elastic net algorithm.

## Elastic Net

For an  $\alpha$  strictly between 0 and 1, and a nonnegative  $\lambda$ , elastic net solves the problem

$$\min_{\beta_0, \beta} \left( \frac{1}{N} \text{Deviance}(\beta_0, \beta) + \lambda P_\alpha(\beta) \right),$$

where

$$P_\alpha(\beta) = \frac{(1-\alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 = \sum_{j=1}^p \left( \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right).$$

Elastic net is the same as lasso when  $\alpha = 1$ . For other values of  $\alpha$ , the penalty term  $P_\alpha(\beta)$  interpolates between the  $L^1$  norm of  $\beta$  and the squared  $L^2$  norm of  $\beta$ . As  $\alpha$  shrinks toward 0, elastic net approaches ridge regression.

## Examples

### Lasso Regularization of a Generalized Linear Model

Construct data from a Poisson model, and identify the important predictors using `lassoglm`.

Create data with 20 predictors, and Poisson responses using just three of the predictors, plus a constant.

```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15]) * [.4; .2; .3] + 1);
y = poissrnd(mu);
```

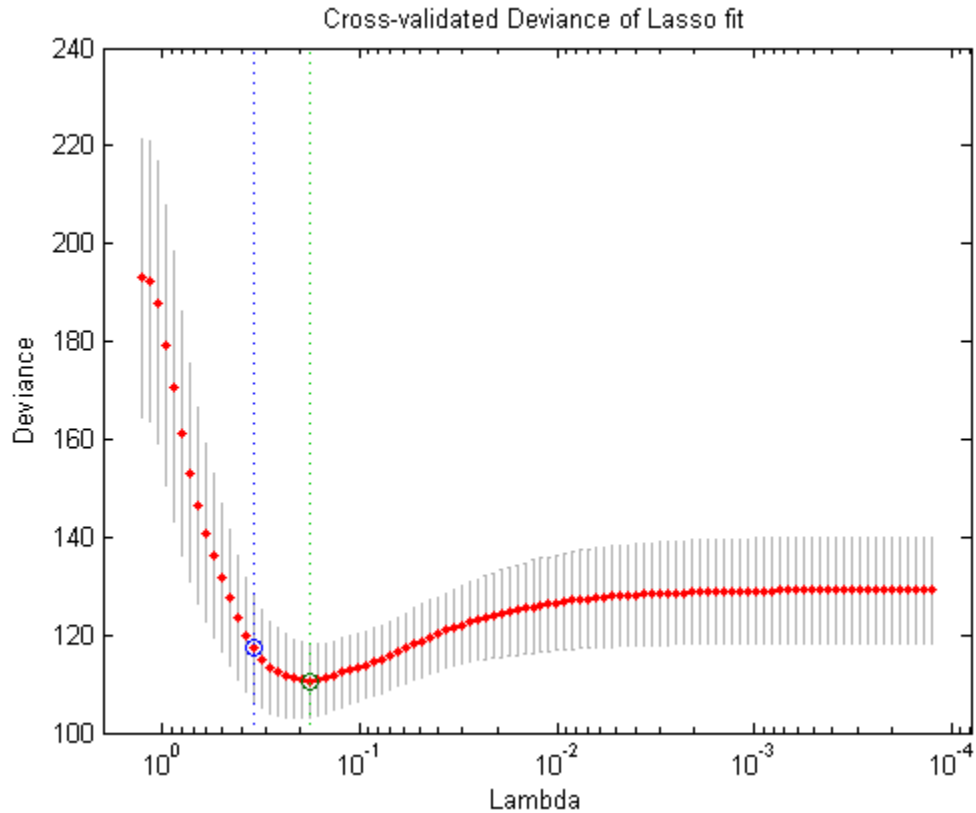
Construct a cross-validated lasso regularization of a Poisson regression model of the data.

```
[B FitInfo] = lassoglm(X,y,'poisson','CV',10);
```

Examine the cross-validation plot to see the effect of the Lambda regularization parameter.

# lassoglm

```
lassoPlot(B,FitInfo,'plotttype','CV');
```



The green circle and dashed line locate the Lambda with minimal cross-validation error. The blue circle and dashed line locate the point with minimal cross-validation error plus one standard deviation.

Find the nonzero model coefficients corresponding to the two identified points.

```
minpts = find(B(:,FitInfo.IndexMinDeviance))
```

```
min1pts =  
    3  
    5  
    6  
   10  
   11  
   15  
   16  
  
min1pts = find(B(:,FitInfo.Index1SE))  
  
min1pts =  
    5  
   10  
   15
```

The coefficients from the minimal plus one standard error point are exactly those coefficients used to create the data.

## References

- [1] Tibshirani, R. *Regression Shrinkage and Selection via the Lasso*. Journal of the Royal Statistical Society, Series B, Vol. 58, No. 1, pp. 267–288, 1996.
- [2] Zou, H. and T. Hastie. *Regularization and Variable Selection via the Elastic Net*. Journal of the Royal Statistical Society, Series B, Vol. 67, No. 2, pp. 301–320, 2005.
- [3] Friedman, J., R. Tibshirani, and T. Hastie. *Regularization Paths for Generalized Linear Models via Coordinate Descent*. Journal of Statistical Software, Vol. 33, No. 1, 2010. <http://www.jstatsoft.org/v33/i01>
- [4] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition. Springer, New York, 2008.

[5] Dobson, A. J. *An Introduction to Generalized Linear Models*, 2nd edition. Chapman & Hall/CRC Press, New York, 2002.

[6] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*, 2nd edition. Chapman & Hall/CRC Press, New York, 1989.

[7] Collett, D. *Modelling Binary Data*, 2nd edition. Chapman & Hall/CRC Press, New York, 2003.

## See Also

`glmfit` | `lasso` | `lassoPlot` | `ridge`

## Related Examples

- “Regularize Poisson Regression” on page 10-48
- “Regularize Logistic Regression” on page 10-52
- “Regularize Wide Data in Parallel” on page 10-59

## Concepts

- “Lasso Regularization of Generalized Linear Models” on page 10-48

<b>Purpose</b>	Trace plot of lasso fit
<b>Syntax</b>	<pre>ax = lassoPlot(B) ax = lassoPlot(B,FitInfo) ax = lassoPlot(B,FitInfo,Name,Value) [ax,figh] = lassoPlot(B,...)</pre>
<b>Description</b>	<p><code>ax = lassoPlot(B)</code> creates a trace plot of the values in <code>B</code> against the <math>L^1</math> norm of <code>B</code>. <code>ax</code> is a handle to the plot axis.</p> <p><code>ax = lassoPlot(B,FitInfo)</code> creates a plot with type depending on the data type of <code>FitInfo</code> and the value, if any, of the <code>plotType</code> name-value pair.</p> <p><code>ax = lassoPlot(B,FitInfo,Name,Value)</code> creates a plot with additional options specified by one or more <code>Name,Value</code> pair arguments.</p> <p><code>[ax,figh] = lassoPlot(B,...)</code> returns a handle to the figure window.</p>
<b>Input Arguments</b>	<p><b>B</b></p> <p>Coefficients of a sequence of regression fits, as returned from the <code>lasso</code> or <code>lassoglm</code> functions. <code>B</code> is a <code>p</code>-by-<code>NLambda</code> matrix, where <code>p</code> is the number of predictors, and each column of <code>B</code> is a set of coefficients <code>lasso</code> calculates using one <code>Lambda</code> penalty value.</p> <p><b>FitInfo</b></p> <p>Information controlling the plot:</p> <ul style="list-style-type: none"><li>• <code>FitInfo</code> is a structure, especially as returned from <code>lasso</code> or <code>lassoglm</code> — <code>lassoPlot</code> creates a plot based on the <code>PlotType</code> name-value pair.</li><li>• <code>FitInfo</code> is a vector — <code>lassoPlot</code> forms the <math>x</math>-axis of the plot from the values in <code>FitInfo</code>. The length of <code>FitInfo</code> must equal the number of columns of <code>B</code>.</li></ul>

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### 'Parent'

Axis in which to draw the plot.

**Default:** New plot

### 'PlotType'

Choose the plot type when you give a `FitInfo` vector or structure:

FitInfo Type	PlotType	Plot
Vector or Structure	'L1'	<code>plotLasso</code> creates the <i>x</i> -axis from the $L^1$ norm of the coefficients in <code>B</code> . The <i>x</i> -axis at the top of the plot contains the degrees of freedom ( <code>df</code> ), meaning the number of nonzero coefficients of <code>B</code> .
Structure	'Lambda'	<code>plotLasso</code> creates the <i>x</i> -axis from the <code>Lambda</code> field of <code>FitInfo</code> . The <i>x</i> -axis at the top of the plot contains the degrees of freedom ( <code>df</code> ), meaning the number of nonzero coefficients of <code>B</code> .
Cross-Validated Structure	'CV'	<ul style="list-style-type: none"><li>• For each <code>Lambda</code>, plots an estimate of the mean squared prediction error on new data for the model fitted by <code>lasso</code> with that value of <code>Lambda</code>.</li><li>• Plots error bars for the estimates.</li><li>• Plots the value of <code>Lambda</code> with minimum cross-validated MSE.</li></ul>



FitInfo Type	PlotType	Plot
		<ul style="list-style-type: none"> <li>Plots the greatest Lambda that is within one standard error of minimum MSE (so makes the sparsest model within that region).</li> </ul>

**Default:** 'L1'

### 'PredictorNames'

Cell array of strings to label each coefficient of **B**. If the length of **PredictorNames** is less than the number of rows of **B**, the remaining labels are padded with default values.

`lassoPlot` uses the predictor names in **FitInfo** only if:

- You created **FitInfo** with a call to `lasso` that included a **PredictorNames** name-value pair.
- You call `lassoPlot` *without* a **PredictorNames** name-value pair.
- You include **FitInfo** in your `lassoPlot` call.

**Default:** {'B1', 'B2', ...}

### 'XScale'

- 'linear' for linear x-axis
- 'log' for logarithmic scaled x-axis

**Default:** 'linear', except 'log' for the 'CV' plot type

## Output Arguments

### **ax**

Handle to the axis of the plot (see “Setting Axis Parameters”).

### **figh**

Handle to the figure window (see “Graphics Windows — the Figure”).

# lassoPlot

---

## Examples

### Lasso Plot with Default Plot Type

Load the sample data

```
load acetylene
```

Prepare the design matrix for lasso fit with interactions.

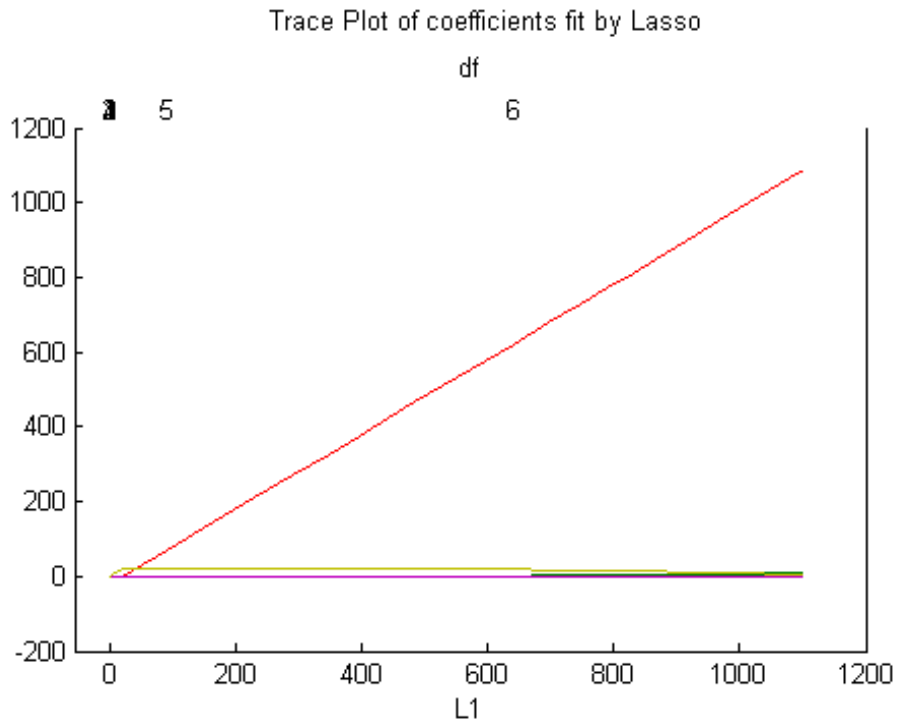
```
X = [x1 x2 x3];  
D = x2fx(X, 'interaction');  
D(:,1) = []; % No constant term
```

Fit a regularized model of the data using `lasso`.

```
B = lasso(D,y);
```

Plot the fits with the default plot type.

```
lassoPlot(B);
```



### Lasso Plot with Lambda Plot Type

Load the sample data.

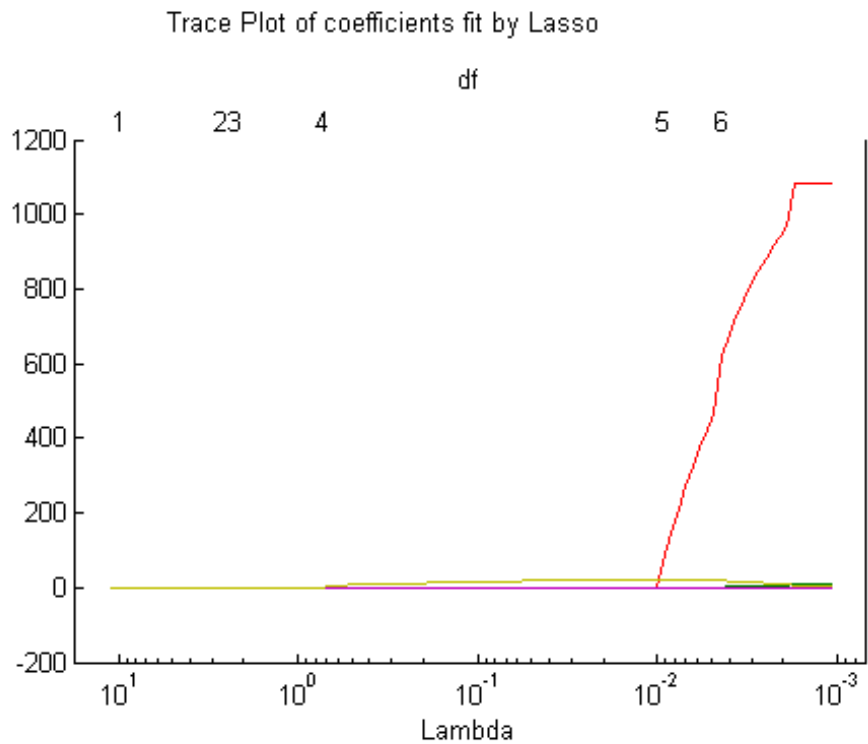
```
load acetylene
```

Prepare the data for lasso fit with interactions.

```
X = [x1 x2 x3];  
D = x2fx(X, 'interaction');
```

# lassoPlot

```
D(:,1) = []; % No constant term  
Fit a regularized model of the data with lasso .  
[B FitInfo] = lasso(D,y);  
Plot the fits with the Lambda plot type and logarithmic scaling.  
lassoPlot(B,FitInfo,'PlotType','Lambda','XScale','log');
```



## Lasso Plot with Cross-Validated Fits

Load the sample data.

```
load acetylene
```

Prepare the design matrix for a lasso fit with interactions.

```
X = [x1 x2 x3];  
D = x2fx(X, 'interaction');  
D(:,1) = []; % No constant term
```

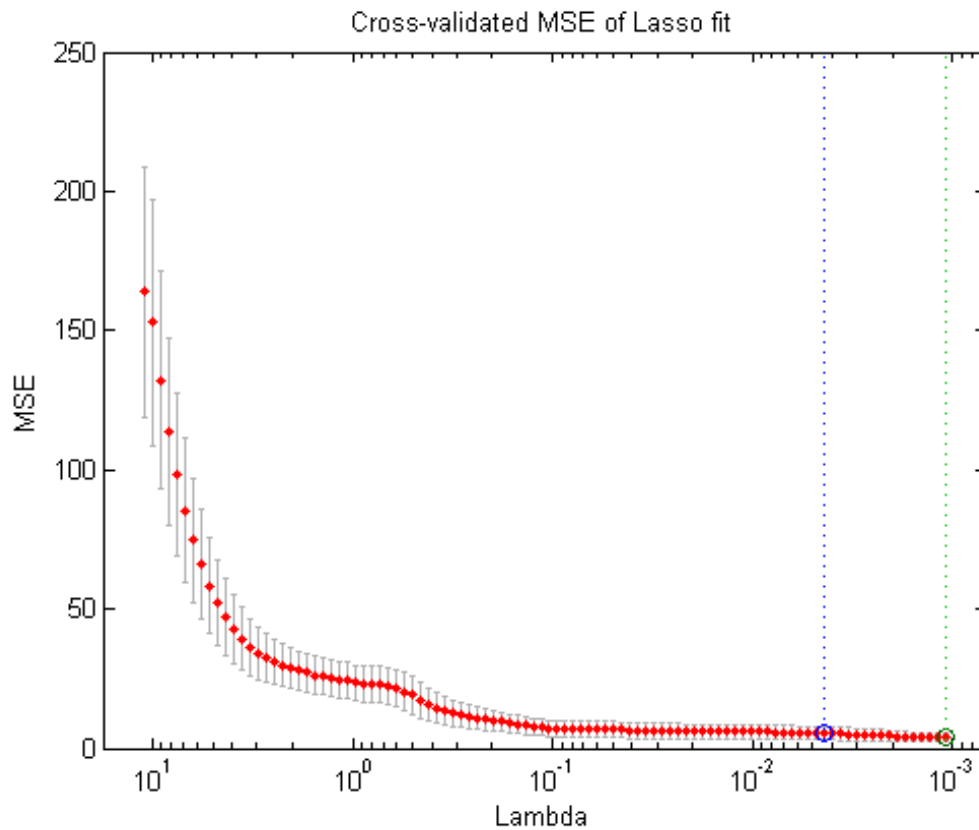
Fit a regularized model of the data with lasso and cross validation.

```
[B FitInfo] = lasso(D,y, 'CV', 10);
```

Plot the cross-validated fits.

```
lassoPlot(B,FitInfo, 'PlotType', 'CV');
```

# lassoPlot



## See Also

`lasso` | `lassoglm`

## How To

- “Lasso and Elastic Net” on page 9-132

<b>Purpose</b>	Less than or equal relation for handles
<b>Syntax</b>	<code>h1 &lt;= h2</code>
<b>Description</b>	<p>Handles are equal if they are handles for the same object. All comparisons use a number associated with each handle object. Nothing can be assumed about the result of a handle comparison except that the repeated comparison of two handles in the same MATLAB session will yield the same result. The order of handle values is purely arbitrary and has no connection to the state of the handle objects being compared.</p> <p><code>h1 &lt;= h2</code> performs element-wise comparisons between handle arrays <code>h1</code> and <code>h2</code>. <code>h1</code> and <code>h2</code> must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise <code>&lt;=</code> result.</p> <p>If one of <code>h1</code> or <code>h2</code> is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.</p> <p><code>tf = le(h1, h2)</code> stores the result in a logical array of the same dimensions.</p>
<b>See Also</b>	<code>grandstream</code>   <code>eq</code>   <code>ge</code>   <code>gt</code>   <code>lt</code>   <code>ne</code>

# grandset.Lean property

---

**Purpose** Interval between points

**Description** Number of points to leap over and omit for each point taken from the sequence. The Leap property of a point set contains a positive integer which specifies the number of points in the sequence to leap over and omit for every point taken. The default Leap value is 0, which corresponds to taking every point from the sequence.

Leaping is a technique used to improve the quality of a point set. However, you must choose the Leap values with care; many Leap values create sequences that fail to touch on large sub-hyper-rectangles of the unit hypercube, and so fail to be a uniform quasi-random point set.

## Choosing Leap Values for Halton Sets

A known rule for choosing Leap values for Halton sets is to set it to  $(P-1)$  where  $P$  is a prime number that has not been used to generate one of the dimensions, i.e. for a  $k$ -dimensional point set  $P$  would be the  $(k+1)$ th or greater prime.

**Examples** Experiment with different leap values:

```
% No leaping produces the standard Halton sequence.
P = haltonset(5);
P(1:5,:)

% Set a leap of 1. The point set now includes every other
% point from the sequence.
P.Lean = 1;
P(1:5,)
```

**See Also** net | grandset | Skip | subsref | haltonset



**Purpose** Length of dataset array

**Syntax** `n = length(A)`

**Description** `n = length(A)` returns the number of observations in the dataset A. `length` is equivalent to `size(A,1)`.

**See Also** `size`

# grandset.length

---

**Purpose** Length of point set

**Syntax** length(p)

**Description** length(p) returns the number of points in the point set p. It is equivalent to size(p, 1).

**See Also** grandset | size

**Purpose** Length of categorical array

**Syntax** `n = length(A)`

**Description** `n = length(A)` returns the size of the longest dimension of the categorical array `A` when `A` is not empty. If `A` is a vector, this is the same as its length. `length` is equivalent to `max(size(x))` for nonempty arrays, and 0 for empty arrays.

**See Also** `isempty` | `isscalar` | `size`

# categorical.levelcounts

---

**Purpose** Element counts by level

**Syntax** `C = levelcounts(A)`  
`C = levelcounts(A,dim)`

**Description** `C = levelcounts(A)` for a categorical vector `A` counts the number of elements in `A` equal to each of the possible levels in `A`. The output is a vector `C` containing those counts, and has as many elements as `A` has levels. For matrix `A`, `C` is a matrix of column counts. For  $N$ -dimensional arrays, `levelcounts` operates along the first nonsingleton dimension. `C = levelcounts(A,dim)` operates along the dimension `dim`.

**Examples** Count the number of patients in each age group in the data in `hospital.mat`:

```
load hospital
edges = 0:10:100;
labels = strcat(num2str((0:10:90)', '%d'), {'s'});
disp(labels')
'0s' '10s' '20s' '30s' '40s' '50s' '60s' '70s' '80s' '90s'

AgeGroup = ordinal(hospital.Age, labels, [], edges);
I = islevel(labels, AgeGroup);
disp(I')
0 1 1 1 1 1 1 1 1 1
c = levelcounts(AgeGroup);
disp(c')
0 0 15 41 42 2 0 0 0 0

AgeGroup = droplevels(AgeGroup);
I = islevel(labels, AgeGroup);
disp(I')
0 0 1 1 1 1 0 0 0 0
c = levelcounts(AgeGroup);
disp(c')
15 41 42 2
```

**See Also**     `islevel` | `ismember`

# leverage

---

## Purpose

Leverage

## Syntax

```
h = leverage(data)
h = leverage(data,model)
```

## Description

`h = leverage(data)` finds the leverage of each row (point) in the matrix data for a linear additive regression model.

`h = leverage(data,model)` finds the leverage on a regression, using a specified model type, where *model* can be one of these strings:

- 'linear' - includes constant and linear terms
- 'interaction' - includes constant, linear, and cross product terms
- 'quadratic' - includes interactions and squared terms
- 'purequadratic' - includes constant, linear, and squared terms

Leverage is a measure of the influence of a given observation on a regression due to its location in the space of the inputs.

## Algorithms

```
[Q,R] = qr(x2fx(data,'model'));
leverage = (sum(Q' .* Q'))'
```

## Examples

One rule of thumb is to compare the leverage to  $2p/n$  where  $n$  is the number of observations and  $p$  is the number of parameters in the model. For the Hald data set this value is 0.7692.

```
load hald
h = max(leverage(ingredients,'linear'))
h =
    0.7004
```

Since  $0.7004 < 0.7692$ , there are no high leverage points using this rule.

## References

[1] Goodall, C. R. "Computation Using the QR Decomposition." *Handbook in Statistics*. Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.

## How To

- regstats

# lhsdesign

---

**Purpose** Latin hypercube sample

**Syntax**

```
X = lhsdesign(n,p)
X = lhsdesign(...,'smooth','off')
X = lhsdesign(...,'criterion',criterion)
X = lhsdesign(...,'iterations',k)
```

**Description** `X = lhsdesign(n,p)` returns an  $n$ -by- $p$  matrix,  $X$ , containing a latin hypercube sample of  $n$  values on each of  $p$  variables. For each column of  $X$ , the  $n$  values are randomly distributed with one from each interval  $(0, 1/n)$ ,  $(1/n, 2/n)$ , ...,  $(1-1/n, 1)$ , and they are randomly permuted.

`X = lhsdesign(...,'smooth','off')` produces points at the midpoints of the above intervals:  $0.5/n$ ,  $1.5/n$ , ...,  $1-0.5/n$ . The default is 'on'.

`X = lhsdesign(...,'criterion',criterion)` iteratively generates latin hypercube samples to find the best one according to the criterion *criterion*, which can be one of the following strings.

Criterion	Description
'none'	No iteration.
'maximin'	Maximize minimum distance between points. This is the default.
'correlation'	Reduce correlation.

`X = lhsdesign(...,'iterations',k)` iterates up to  $k$  times in an attempt to improve the design according to the specified criterion. The default is  $k = 5$ .

**See Also** haltonset | sobolset | lhsnorm | unifrnd



---

<b>Purpose</b>	Latin hypercube sample from normal distribution
<b>Syntax</b>	<pre>X = lhsnorm(mu,sigma,n) X = lhsnorm(mu,sigma,n,flag) [X,Z] = lhsnorm(...)</pre>
<b>Description</b>	<p><code>X = lhsnorm(mu,sigma,n)</code> returns an <math>n</math>-by-<math>p</math> matrix, <math>X</math>, containing a latin hypercube sample of size <math>n</math> from a <math>p</math>-dimensional multivariate normal distribution with mean vector, <math>\mu</math>, and covariance matrix, <math>\sigma</math>.</p> <p><math>X</math> is similar to a random sample from the multivariate normal distribution, but the marginal distribution of each column is adjusted so that its sample marginal distribution is close to its theoretical normal distribution.</p> <p><code>X = lhsnorm(mu,sigma,n,flag)</code> controls the amount of smoothing in the sample. If <code>flag</code> is 'off', each column has points equally spaced on the probability scale. In other words, each column is a permutation of the values <math>G(0.5/n)</math>, <math>G(1.5/n)</math>, ..., <math>G(1-0.5/n)</math> where <math>G</math> is the inverse normal cumulative distribution for that column's marginal distribution. If <code>flag</code> is 'on' (the default), each column has points uniformly distributed on the probability scale. For example, in place of <math>0.5/n</math> you use a value having a uniform distribution on the interval <math>(0/n, 1/n)</math>.</p> <p><code>[X,Z] = lhsnorm(...)</code> also returns <math>Z</math>, the original multivariate normal sample before the marginals are adjusted to obtain <math>X</math>.</p>
<b>References</b>	[1] Stein, M. "Large sample properties of simulations using latin hypercube sampling." <i>Technometrics</i> . Vol. 29, No. 2, 1987, pp. 143–151. Correction, Vol. 32, p. 367.
<b>See Also</b>	lhsdesign   mvnrnd

# lillietest

---

**Purpose** Lilliefors test

**Syntax**

```
h = lillietest(x)
h = lillietest(x,Name,Value)
[h,p] = lillietest( __ )
[h,p,kstat,critval] = lillietest( __ )
```

**Description** `h = lillietest(x)` returns a test decision for the null hypothesis that the data in vector `x` comes from a distribution in the normal family, against the alternative that it does not come from such a distribution, using a Lilliefors test. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = lillietest(x,Name,Value)` returns a test decision with additional options specified by one or more name-value pair arguments. For example, you can test the data against a different distribution family, change the significance level, or calculate the  $p$ -value using a Monte Carlo approximation.

`[h,p] = lillietest( __ )` also returns the  $p$ -value `p`, using any of the input arguments from the previous syntaxes.

`[h,p,kstat,critval] = lillietest( __ )` also returns the test statistic `kstat` and the critical value `critval` for the test.

## Input Arguments

### **x - Sample data**

vector

Sample data, specified as a vector.

### **Data Types**

single | double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding

value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**Example:** 'Distr','exp','Alpha',0.01 tests the null hypothesis that the population distribution belongs to the exponential distribution family at the 1% significance level.

### **'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

- If MCTol is not used, Alpha must be in the range [0.001,0.50].
- If MCTol is used, Alpha must be in the range (0,1).

**Example:** 'Alpha',0.01

### **Data Types**

single | double

### **'Distr' - Distribution family**

'norm' (default) | 'exp' | 'ev'

Distribution family for the hypothesis test, specified as the comma-separated pair consisting of 'Distr' and one of the following.

'norm'	Normal distribution
'exp'	Exponential distribution
'ev'	Extreme value distribution

**Example:** 'Distr','exp'

### **'MCTol' - Maximum Monte Carlo standard error**

scalar value in the range (0,1)

Maximum Monte Carlo standard error for  $p$ , the  $p$ -value of the test, specified as the comma-separated pair consisting of 'MCTol' and a scalar value in the range (0,1).

**Example:** 'MCTol',0.001

## Data Types

single | double

## Output Arguments

### h - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### p - p-value

scalar value in the range (0,1)

$p$ -value of the test, returned as a scalar value in the range (0,1).  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

- If MCTol is not used,  $p$  is computed using inverse interpolation into the table of critical values, and is returned as a scalar value in the range [0.001,0.50]. lillietest warns when  $p$  is not found within the tabulated range and returns either the smallest or largest tabulated value.
- If MCTol is used, lillietest conducts a Monte Carlo simulation to compute a more accurate  $p$ -value, and  $p$  is returned as a scalar value in the range (0,1).

### kstat - Test statistic

nonnegative scalar value

Test statistic, returned as a nonnegative scalar value.

### **critval - Critical value**

nonnegative scalar value

Critical value for the hypothesis test, returned as a nonnegative scalar value.

## **Examples**

### **Test for a Normal Distribution**

Load the sample data. Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars.

```
load carbig.mat;
[h,p,k,c] = lillietest(MPG)
```

Warning: P is less than the smallest tabulated value, returning 0.001.

```
h =
    1
p =
 1.0000e-003
k =
    0.0789
c =
    0.0451
```

The test statistic  $k$  is greater than the critical value  $c$ , so `lillietest` returns a result of  $h = 1$  to indicate rejection of the null hypothesis at the default 5% significance level. The warning indicates that the returned  $p$ -value is the smallest value in the table of precomputed values. To find a more accurate  $p$ -value, use `MCTol` to run a Monte Carlo approximation.

### **Test the Hypothesis at Different Significance Levels**

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Test the null hypothesis that the sample data comes from a normal distribution at the 1% significance level.

```
[h,p] = lillietest(x,'Alpha',0.01)
```

```
h =  
0  
p =  
0.0348
```

The returned value of  $h = 0$  indicates that `lillietest` does not reject the null hypothesis at the 1% significance level.

## Test for an Exponential Distribution

Load the sample data. Test the null hypothesis that car mileage, in miles per gallon (MPG), follows an exponential distribution across different makes of cars.

```
load carbig.mat;  
h = lillietest(MPG,'Distr','exp')
```

```
h =  
1
```

The returned value of  $h = 1$  indicates that `lillietest` rejects the null hypothesis at the default 5% significance level.

## Determine the $p$ -value Using Monte Carlo Approximation

Load the sample data. Test the null hypothesis that car mileage, in miles per gallon (MPG), follows a normal distribution across different makes of cars. Determine the  $p$ -value using a Monte Carlo approximation with a maximum Monte Carlo standard error of  $1e-4$ .

```
load carbig.mat;  
[h,p] = lillietest(MPG,'MCTol',1e-4)
```

```

h =
  1
p =
  0

```

The returned value of `h = 1` indicates that `lillietest` rejects the null hypothesis that the data comes from a normal distribution at the 5% significance level.

## Definitions

### Lilliefors Test

The Lilliefors test is a two-sided goodness-of-fit test suitable when the parameters of the null distribution are unknown and must be estimated. This is in contrast to the one-sample Kolmogorov-Smirnov test, which requires the null distribution to be completely specified.

The Lilliefors test statistic is:

$$D^* = \max_x |F(x) - G(x)|,$$

where  $\hat{F}(x)$  is the empirical cdf of the sample data and  $G(x)$  is the cdf of the hypothesized distribution with estimated parameters equal to the sample parameters.

`lillietest` can be used to test whether the data vector  $\mathbf{x}$  has a lognormal or Weibull distribution by applying a transformation to the data vector and running the appropriate Lilliefors test:

- To test  $\mathbf{x}$  for a lognormal distribution, test if  $\exp(\mathbf{x})$  has a normal distribution.
- To test  $\mathbf{x}$  for a Weibull distribution, test if  $\log(\mathbf{x})$  has an extreme value distribution.

The Lilliefors test cannot be used when the null hypothesis is not a location-scale family of distributions.

## **Monte Carlo Standard Error**

The Monte Carlo standard error is the error due to simulating the  $p$ -value.

The Monte Carlo standard error is calculated as:

$$SE = \sqrt{\frac{(\hat{p})(1-\hat{p})}{\text{mcreps}}},$$

where  $\hat{p}$  is the estimated  $p$ -value of the hypothesis test, and `mcreps` is the number of Monte Carlo replications performed.

The number of Monte Carlo replications, `mcreps`, is determined such that the Monte Carlo standard error for  $\hat{p}$  less than the value specified for `MCTol`.

## **Algorithms**

To compute the critical value for the hypothesis test, `lillietest` interpolates into a table of critical values pre-computed using Monte Carlo simulation for sample sizes less than 1000 and significance levels between 0.001 and 0.50. The table used by `lillietest` is larger and more accurate than the table originally introduced by Lilliefors. If a more accurate  $p$ -value is desired, or if the desired significance level is less than 0.001 or greater than 0.50, the `MCTol` input argument can be used to run a Monte Carlo simulation to calculate the  $p$ -value more exactly.

When the computed value of the test statistic is greater than the critical value, `lillietest` rejects the null hypothesis at significance level `Alpha`.

`lillietest` treats NaN values in `x` as missing values and ignores them.



## References

- [1] Conover, W. J. *Practical Nonparametric Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [2] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for the exponential distribution with mean unknown.” *Journal of the American Statistical Association*. Vol. 64, 1969, pp. 387–389.
- [3] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for normality with mean and variance unknown.” *Journal of the American Statistical Association*. Vol. 62, 1967, pp. 399–402.

## See Also

`jbtest` | `kstest` | `kstest2` | `cdfplot` | `adtest`

# LinearModel

---

**Purpose** Linear regression model class

**Description** An object comprising training data, model description, diagnostic information, and fitted coefficients for a linear regression. Predict model responses with the `predict` or `feval` methods.

**Construction** `mdl = fitlm(ds)` or `mdl = fitlm(X,y)` create a linear model of a dataset array `ds`, or of the responses `y` to a data matrix `X`. For details, see `fitlm`.

`mdl = stepwiselm(ds)` or `mdl = stepwiselm(X,y)` create a linear model of a dataset array `ds`, or of the responses `y` to a data matrix `X`, with unimportant predictors excluded. For details, see `stepwiselm`.

## Input Arguments

### **ds** - Input data

dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **Data Types**

single | double | logical

### **X** - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

### Data Types

single | double | logical

### **y** - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### Data Types

single | double

## Properties

### CoefficientCovariance

Covariance matrix of coefficient estimates.

### CoefficientNames

Cell array of strings containing a label for each coefficient.

### Coefficients

Table of coefficient values in a dataset array. **Coefficients** has one row for each coefficient and these columns:

- Estimate — Estimated coefficient value
- SE — Standard error of the estimate
- tStat —  $t$  statistic for a test that the coefficient is zero
- pValue —  $p$ -value for the  $t$  statistic

# LinearModel

---

To obtain any of these columns as a vector, index into the property using dot notation. For example, in `mdl` the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use `coefTest` to perform other tests on the coefficients.

## DFE

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients.

## Diagnostics

Dataset array with the same number of rows as the input data (`ds` or `X`). `Diagnostics` contains diagnostics helpful in finding outliers and influential observations. Many diagnostics describe the effect on the fit of deleting single observations. `Diagnostics` contains the following fields.

Field	Meaning	Utility
Leverage	Diagonal elements of <code>HatMatrix</code>	<code>Leverage</code> indicates to what extent the predicted value for an observation is determined by the observed value for that observation. A value close to 1 indicates that the prediction is largely determined by that observation, with little contribution from the other observations. A value close to 0 indicates the fit is largely determined by the other observations. For a model with <code>P</code> coefficients and <code>N</code> observations, the average value of <code>Leverage</code> is <code>P/N</code> . An observation with <code>Leverage</code> larger than <code>2*P/N</code> can be regarded as having high leverage.
<code>CooksDistance</code>	Cook's measure of scaled change in fitted values	<code>CooksDistance</code> is a measure of scaled change in fitted values. An observation with <code>CooksDistance</code> larger than three times the mean Cook's distance can be an outlier.

Field	Meaning	Utility
Dffits	Delete-1 scaled differences in fitted values vs. observation number	Dffits is the scaled change in the fitted values for each observation that would result from excluding that observation from the fit. Values with an absolute value larger than $2 \cdot \sqrt{P/N}$ may be considered influential.
S2_i	Delete-1 variance vs. observation number	S2_i is a set of residual variance estimates obtained by deleting each observation in turn. These can be compared with the value of the MSE property.
CovRatio	Delete-1 ratio of determinant of covariance vs. observation number	CovRatio is the ratio of the determinant of the coefficient covariance matrix with each observation deleted in turn to the determinant of the covariance matrix for the full model. Values larger than $1 + 3 \cdot P/N$ or smaller than $1 - 3 \cdot P/N$ indicate influential points.
Dfbetas	Delete-1 scaled differences in covariance estimates vs. observation number	Dfbetas is an N-by-P matrix of the scaled change in the coefficient estimates that would result from excluding each observation in turn. Values larger than $3 / \sqrt{N}$ in absolute value indicate that the observation has a large influence on the corresponding coefficient.
HatMatrix	Projection matrix to compute fitted from observed responses	HatMatrix is an N-by-N matrix such that $\text{Fitted} = \text{HatMatrix} \cdot Y$ , where Y is the response vector and Fitted is the vector of fitted response values.

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exception: Delete-1 diagnostics refer to the statistic with and without that observation (row) included in the fit. These diagnostics help identify important observations.

# LinearModel

---

## **Fitted**

Predicted response to the input data by using the model. Use `predict` to compute predictions for other predictor values, or to compute confidence bounds on `Fitted`.

## **Formula**

Object containing information about the model.

## **LogLikelihood**

Log likelihood of the model distribution at the response values, with mean fitted from the model, and other parameters estimated as part of the model fit.

## **ModelCriterion**

AIC and other information criteria for comparing models. A structure with fields:

- `AIC` — Akaike information criterion
- `AICc` — Akaike information criterion corrected for sample size
- `BIC` — Bayesian information criterion
- `CAIC` — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the AIC value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

## **MSE**

Mean squared error (residuals),  $SSE/DFE$ .

## **NumCoefficients**

Number of coefficients in the model, a positive integer. `NumCoefficients` includes coefficients that are set to zero when the model terms are rank deficient.

## NumEstimatedCoefficients

Number of estimated coefficients in the model, a positive integer. `NumEstimatedCoefficients` does not include coefficients that are set to zero when the model terms are rank deficient. `NumEstimatedCoefficients` is the degrees of freedom for regression.

## NumObservations

Number of observations the fitting function used in fitting. This is the number of observations supplied in the original dataset or matrix, minus any excluded rows (set with the `Excluded` name-value pair) or rows with missing values.

## NumPredictors

Number of variables `fitlm` used as predictors for fitting.

## NumVariables

Number of variables in the data. `NumVariables` is the number of variables in the original dataset when the fit is based on a dataset, or the total number of columns in the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

## ObservationInfo

Dataset with the same number of rows as the input data (`ds` or `X`).

Field	Description
Weights	Observation weights. Default is all 1.
Excluded	Logical value, 1 indicates an observation that you excluded from the fit with the <code>Exclude</code> name-value pair.

# LinearModel

---

Field	Description
Missing	Logical value, 1 indicates a missing value in the input. Missing values are not used in the fit.
Subset	Logical value, 1 indicates the observation is not excluded or missing, so is used in the fit.

## ObservationNames

Cell array of strings containing the names of the observations used in the fit.

- If the fit is based on a dataset containing observation names, `ObservationNames` uses those names.
- Otherwise, `ObservationNames` is an empty cell array

## PredictorNames

Cell array of strings, the names of the predictors used in fitting the model.

## Residuals

Dataset array containing a table of residuals, with one row for each observation and these variables.

Field	Description
Raw	Observed minus fitted values.
Pearson	Raw residuals divided by RMSE.



Field	Description
Standardized	Raw residuals divided by their estimated standard deviation.
Studentized	Residual divided by an independent estimate of the residual standard deviation. The residual for observation $i$ is divided by an estimate of the error standard deviation based on all observations except for observation $i$ .

To obtain any of these columns as a vector, index into the property using dot notation. For example, in a model `mdl`, the ordinary raw residual vector `r` is:

```
r = mdl.Residuals.Raw
```

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exceptions:

- `raw` contains the difference between the observed and predicted values.
- `standardized` is the residual, standardized in the usual way.
- `studentized` matches the standardized values because this residual is not used in the estimate of the residual standard deviation.

## **ResponseName**

String giving naming the response variable.

## **RMSE**

Root mean squared error (residuals),  $\sqrt{\text{MSE}}$ .

## **Robust**

# LinearModel

---

Structure that is empty unless `fitlm` constructed the model using robust regression.

Field	Description
WgtFun	Robust weighting function, such as 'bisquare' (see <code>robustfit</code> )
Tune	Value specified for tuning parameter (can be [ ])
Weights	Vector of weights used in final iteration of robust fit

## **Rsquared**

Proportion of total sum of squares explained by the model. The ordinary R-squared value relates to the SSR and SST properties:

$$\text{Rsquared} = \text{SSR}/\text{SST} = 1 - \text{SSE}/\text{SST}.$$

Rsquared is a structure with two fields:

- Ordinary — Ordinary (unadjusted) R-squared
- Adjusted — R-squared adjusted for the number of coefficients

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

$$r2 = \text{mdl.Rsquared.Adjusted}$$

## **SSE**

Sum of squared errors (residuals).

The Pythagorean theorem implies

$$\text{SST} = \text{SSE} + \text{SSR}.$$

## SSR

Regression sum of squares, the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

## SST

Total sum of squares, the sum of squared deviations of  $y$  from  $\text{mean}(y)$ .

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

## Steps

Structure that is empty unless `stepwiselm` constructed the model.

Field	Description
Start	Formula representing the starting model
Lower	Formula representing the lower bound model, these terms that must remain in the model
Upper	Formula representing the upper bound model, model cannot contain more terms than Upper
Criterion	Criterion used for the stepwise algorithm, such as 'sse'
PEnter	Value of the parameter, such as 0.05
PRemove	Value of the parameter, such as 0.10
History	Dataset representing the steps taken in the fit

The `History` dataset has one row for each step including the initial fit, and the following variables (columns).

# LinearModel

---

Field	Description
Action	Action taken during this step, one of: <ul style="list-style-type: none"><li>'Start' — First step</li><li>'Add' — A term is added</li><li>'Remove' — A term is removed</li></ul>
TermName	<ul style="list-style-type: none"><li>'Start' step: The starting model specification</li><li>'Add' or 'Remove' steps: The term moved in that step</li></ul>
Terms	Terms matrix (see <code>modelspec</code> of <code>fitlm</code> )
DF	Regression degrees of freedom after this step
delDF	Change in regression degrees of freedom from previous step (negative for steps that remove a term)
Deviance	Deviance (residual sum of squares) at that step
FStat	$F$ statistic that led to this step
PValue	$p$ -value of the $F$ statistic

## VariableInfo

Dataset array containing metadata about `Variables`. There is one row for each term in the model, and the following columns.

Field	Description
Class	String giving variable class, such as 'double'
Range	Cell array giving variable range: <ul style="list-style-type: none"><li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li><li>• Categorical variable — Cell array of distinct variable values</li></ul>
InModel	Logical vector, where true indicates the variable is in the model
IsCategorical	Logical vector, where true indicates a categorical variable

## VariableNames

Cell array of strings containing names of the variables in the fit.

- If the fit is based on a dataset, this property provides the names of the variables in that dataset.
- If the fit is based on a predictor matrix and response vector, `VariableNames` is the values in the `VarNames` name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

## Variables

Dataset array containing the data, both observations and responses, that the fitting function used to construct the fit. If the fit is based on a dataset array, `Variables` is a copy of that dataset. Otherwise, `Variables` is a dataset created from the input data matrix  $X$  and response vector  $y$ .

# LinearModel

---

## Methods

<code>addTerms</code>	Add terms to linear regression model
<code>anova</code>	Analysis of variance for linear model
<code>coefCI</code>	Confidence intervals of coefficient estimates of linear model
<code>coefTest</code>	Linear hypothesis test on linear regression model coefficients
<code>disp</code>	Display linear regression model
<code>dwtest</code>	Durbin-Watson test of linear model
<code>feval</code>	Evaluate linear regression model prediction
<code>fit</code>	Create linear regression model
<code>plot</code>	Scatter plot or added variable plot of linear model
<code>plotAdded</code>	Added variable plot or leverage plot for linear model
<code>plotAdjustedResponse</code>	Adjusted response plot for linear regression model
<code>plotDiagnostics</code>	Plot diagnostics of linear regression model
<code>plotEffects</code>	Plot main effects of each predictor in linear regression model
<code>plotInteraction</code>	Plot interaction effects of two predictors in linear regression model
<code>plotResiduals</code>	Plot residuals of linear regression model

plotSlice	Plot of slices through fitted linear regression surface
predict	Predict response of linear regression model
random	Simulate responses for linear regression model
removeTerms	Remove terms from linear model
step	Improve linear regression model by adding or removing terms
stepwise	Create linear regression model by stepwise regression

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Definitions

### Hat Matrix

The *hat matrix*  $H$  is defined in terms of the data matrix  $X$ :

$$H = X(X^T X)^{-1} X^T.$$

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

### Leverage

The *leverage* of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the

number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

Cook's distance is the scaled change in fitted values. Each element in `CooksDistance` is the normalized change in the vector of coefficients due to the deletion of an observation. The Cook's distance,  $D_i$ , of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $\text{MSE}$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p \text{MSE}} \left( \frac{h_{ii}}{(1 - h_{ii})^2} \right),$$

where  $r_i$  is the  $i$ th residual, and  $h_{ii}$  is the  $i$ th leverage value.

`CooksDistance` is an  $n$ -by-1 column vector in the `Diagnostics` dataset array of the `LinearModel` object.



## Examples

### Linear Regression Model of Matrix Data

Fit a linear model of the Hald data.

Load the data.

```
load hald
X = ingredients; % predictor variables
y = heat; % response
```

Fit a default linear model to the data.

```
mdl = fitlm(X,y)
```

```
mdl =
```

Linear regression model:

$$y \sim 1 + x_1 + x_2 + x_3 + x_4$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	62.405	70.071	0.8906	0.39913
x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

Number of observations: 13, Error degrees of freedom: 8

Root Mean Squared Error: 2.45

R-squared: 0.982, Adjusted R-Squared 0.974

F-statistic vs. constant model: 111, p-value = 4.76e-07

### Linear Regression with Categorical Predictor and Nonlinear Model

Fit a model of a dataset array that contains a categorical predictor.  
Use a nonlinear response formula.

Load the carsmall data.

# LinearModel

---

```
load carsmall
```

Construct a dataset containing continuous predictor variable `Weight`, nominal predictor variable `Year`, and response variable `MPG`.

```
ds = dataset(MPG,Weight);  
ds.Year = nominal(Model_Year);
```

Create a fitted model of `MPG` as a function of `Year`, `Weight`, and `Weight2`. (You don't have to include `Weight` explicitly in your formula because it is a lower-order term of `Weight2`. For details, see "Formula" on page 21-938.)

```
mdl = fitlm(ds,'MPG ~ Year + Weight^2')
```

```
mdl =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Year + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

```
Number of observations: 94, Error degrees of freedom: 89
```

```
Root Mean Squared Error: 2.78
```

```
R-squared: 0.885, Adjusted R-Squared 0.88
```

```
F-statistic vs. constant model: 172, p-value = 5.52e-41
```

`fitlm` creates two dummy (indicator) variables for the nominal variate, `Year`. The dummy variable `Year_76` takes the value 1 if model year is 1976 and takes the value 0 if it is not. The dummy variable `Year_82` takes the value 1 if model year is 1982 and takes the value 0 if it is not. And the year 1970 is the reference year (for more information on

dummy variables see “Dummy Indicator Variables” on page 2-55). The corresponding model is

$$\hat{MPG} = 54.206 - 0.0164Weight + 2.0887(Year\_76) + 8.1864(Year\_82) + 1.557e - 06Weight^2$$

## Robust Linear Regression Model

Fit a linear regression model of the Hald data using robust fitting.

Load the data.

```
load hald
X = ingredients; % predictor variables
y = heat; % response
```

Fit a robust linear model to the data.

```
mdl = fitlm(X,y,'linear','RobustOpts','on')
```

```
mdl =
```

```
Linear regression model (robust fit):
```

```
y ~ 1 + x1 + x2 + x3 + x4
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	60.09	75.818	0.79256	0.4509
x1	1.5753	0.80585	1.9548	0.086346
x2	0.5322	0.78315	0.67957	0.51596
x3	0.13346	0.8166	0.16343	0.87424
x4	-0.12052	0.7672	-0.15709	0.87906

```
Number of observations: 13, Error degrees of freedom: 8
```

```
Root Mean Squared Error: 2.65
```

```
R-squared: 0.979, Adjusted R-Squared 0.969
```

```
F-statistic vs. constant model: 94.6, p-value = 9.03e-07
```

# LinearModel

---

## Algorithms

The main fitting algorithm is QR decomposition. For robust fitting, the algorithm is `robustfit`.

## Alternatives

To remove redundant predictors in linear regression using lasso or elastic net, use the `lasso` function.

To regularize a regression with correlated terms using ridge regression, use the `ridge` or `lasso` functions.

To regularize a regression with correlated terms using partial least squares, use the `plsregress` function.

## See Also

`fitlm` | `stepwiselm`

## Tutorials

- “Linear Regression Workflow” on page 9-44
- “Robust Regression versus Standard Least-Squares Fit” on page 9-125

## How To

- “Linear Regression” on page 9-12

## Purpose

Linear mixed-effects model class

## Description

A `LinearMixedModel` object represents a model of a response variable with fixed and random effects. It comprises data, a model description, fitted coefficients, covariance parameters, design matrices, residuals, residual plots, and other diagnostic information for a linear mixed-effects model. You can predict model responses with the `predict` function and generate random data at new design points using the `random` function.

## Construction

You can fit a linear mixed-effects model using `fitlme(ds, formula)` if your data is in a `dataset` array. Alternatively, if your model is not easily described using a formula, you can create matrices to define the fixed and random effects, and fit the model using `fitlmematrix(X, y, Z, G)`.

## Input Arguments

### **ds - Input data**

`dataset` array

Input data, which includes the response variable, predictor variables, and grouping variables, specified as a `dataset` array. The predictor variables can be continuous or grouping variables (see “Grouping Variables” on page 2-51). You must specify the model for the variables using `formula`.

### **Data Types**

`single` | `double` | `char` | `cell`

### **formula - Formula for model specification**

string of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`

Formula for model specification, specified as a string of the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`. For a full description, see “Formula” on page 21-1722.

**Example:** `'y ~ treatment +(1|block)'`

# LinearMixedModel

---

## **X - Fixed-effects design matrix**

*n*-by-*p* matrix

Fixed-effects design matrix, specified as an *n*-by-*p* matrix, where *n* is the number of observations, and *p* is the number of fixed-effects predictor variables. Each row of X corresponds to one observation, and each column of X corresponds to one variable.

### **Data Types**

single | double

## **y - Response values**

*n*-by-1 vector

Response values, specified as an *n*-by-1 vector, where *n* is the number of observations.

### **Data Types**

single | double

## **Z - Random-effects design**

*n*-by-*q* matrix | cell array of *R* *n*-by-*q*(*r*) matrices, *r* = 1, 2, ..., *R*

Random-effects design, specified as either of the following.

- If there is one random-effects term in the model, then Z must be an *n*-by-*q* matrix, where *n* is the number of observations and *q* is the number of variables in the random-effects term.
- If there are *R* random-effects terms, then Z must be a cell array of length *R*. Each cell of Z contains an *n*-by-*q*(*r*) design matrix Z{*r*}, *r* = 1, 2, ..., *R*, corresponding to each random-effects term. Here, *q*(*r*) is the number of random effects term in the *r*th random effects design matrix, Z{*r*}.

### **Data Types**

single | double | cell

## **G - Grouping variable or variables**

*n*-by-1 vector | cell array of *R* *n*-by-1 vectors

Grouping variable or variables, specified as either of the following.

- If there is one random-effects term, then  $G$  must be an  $n$ -by-1 vector corresponding to a single grouping variable with  $M$  levels or groups.

$G$  can be a categorical vector, numeric vector, character array, or cell array of strings.

- If there are multiple random-effects terms, then  $G$  must be a cell array of length  $R$ . Each cell of  $G$  contains a grouping variable  $G\{r\}$ ,  $r = 1, 2, \dots, R$ , with  $M(r)$  levels.

$G\{r\}$  can be a categorical vector, numeric vector, character array, or cell array of strings.

## Data Types

single | double | char | cell

## Properties

### Coefficients

Fixed-effects coefficient estimates and related statistics, stored as a dataset array containing the following fields.

Name	Name of the term.
Estimate	Estimated value of the coefficient.
SE	Standard error of the coefficient.
tStat	$t$ -statistics for testing the null hypothesis that the coefficient is equal to zero.

# LinearMixedModel

---

DF	Degrees of freedom for the <i>t</i> -test. Method to compute DF is specified by the 'DFMethod' name-value pair argument. <code>Coefficients</code> always uses the 'Residual' method for 'DFMethod'.
pValue	<i>p</i> -value for the <i>t</i> -test.
Lower	Lower limit of the confidence interval for coefficient. <code>Coefficients</code> always uses the 95% confidence level, i.e. 'alpha' is 0.05.
Upper	Upper limit of confidence interval for coefficient. <code>Coefficients</code> always uses the 95% confidence level, i.e. 'alpha' is 0.05.

You can change 'DFMethod' and 'alpha' while computing confidence intervals for or testing hypotheses involving fixed- and random-effects, using the `coefCI` and `coefTest` methods.

## **CoefficientCovariance**

Covariance of the estimated fixed-effects coefficients of the linear mixed-effects model, stored as a *p*-by-*p* matrix, where *p* is the number of fixed-effects coefficients.

You can display the covariance parameters associated with the random effects using the `covarianceParameters` method.

## **Data Types**

double

## **CoefficientNames**



1-by- $p$  cell array of strings containing the names of the fixed-effects coefficients of a linear mixed-effects model.

## Data Types

cell

## DFE

Residual degrees of freedom, stored as a positive integer value.  $DFE = n - p$ , where  $n$  is the number of observations, and  $p$  is the number of fixed-effects coefficients.

This corresponds to the 'Residual' method of calculating degrees of freedom in the `fixedEffects` and `randomEffects` methods.

## Data Types

double

## FitMethod

Method used to fit the linear mixed-effects model, stored as either of the following strings.

- ML, if the fitting method is maximum likelihood
- REML, if the fitting method is restricted maximum likelihood

## Data Types

char

## Formula

Specification of the fixed-effects terms, random-effects terms, and grouping variables that define the linear mixed-effects model, stored as an object.

For more information on how to specify the model to fit using a formula, see “Formula” on page 21-1722.

## LogLikelihood

# LinearMixedModel

---

Maximized log likelihood or maximized restricted log likelihood of the fitted linear mixed-effects model depending on the fitting method you choose, stored as a scalar value.

## Data Types

double

## ModelCriterion

Model criterion that can be used to compare fitted linear mixed-effects models, stored as a dataset with the following columns.

AIC	Akaike Information Criterion
BIC	Bayesian Information Criterion
Loglikelihood	Log likelihood value of the model
Deviance	-2 times the log likelihood of the model

If  $n$  is the number of observations used in fitting the model, and  $p$  is the number of fixed-effects coefficients, then for calculating AIC and BIC,

- The total number of parameters is  $nc + p + 1$ , where  $nc$  is the total number of parameters in the random-effects covariance excluding the residual variance
- The effective number of observations is
  - $n$ , when the fitting method is maximum likelihood (ML)
  - $n - p$ , when the fitting method is restricted maximum likelihood (REML)

## MSE

ML or REML estimate, based on the fitting method used for estimating  $\sigma^2$ , stored as a positive scalar value.  $\sigma^2$  is the residual variance or variance of the observation error term of the linear mixed-effects model.

**Data Types**

double

**NumCoefficients**

Number of fixed-effects coefficients in the fitted linear mixed-effects model, stored as a positive integer value.

**Data Types**

double

**NumEstimatedCoefficients**

Number of estimated fixed-effects coefficients in the fitted linear mixed-effects model, stored as a positive integer value.

**Data Types**

double

**NumObservations**

Number of observations used in the fit, stored as a positive integer value. This is the number of rows in the dataset array, or the design matrices minus the excluded rows or rows with NaN values.

**Data Types**

double

**NumPredictors**

Number of variables used as predictors in the linear mixed-effects model, stored as a positive integer value.

**Data Types**

double

**NumVariables**

# LinearMixedModel

---

Total number of variables including the response and predictors, stored as a positive integer value.

- If the sample data is in a dataset array `ds`, `NumVariables` is the total number of variables in `ds` including the response variable.
- If the fit is based on matrix input, `NumVariables` is the total number of columns in the predictor matrix or matrices, and response vector.

`NumVariables` includes variables, if there are any, that are not used as predictors or as the response.

## Data Types

double

## ObservationInfo

Information about the observations used in the fit, stored as a dataset array.

`ObservationInfo` has one row for each observation and the following four columns.

Weights	The value of the weighted variable for that observation. Default value is 1.
Excluded	true, if the observation was excluded from the fit using the 'Exclude' name-value pair argument, false, otherwise. 1 stands for true and 0 stands for false.

Missing

true, if the observation was excluded from the fit because any response or predictor value is missing, false, otherwise. Missing values include NaN for numeric variables, empty cells for cell arrays, blank rows for character arrays, and the <undefined> value for categorical arrays.

Subset

true, if the observation was used in the fit, false, if it was not used because it is missing or excluded.

## ObservationNames

Names of observations used in the fit, stored as a cell array of strings.

- If the data is in a dataset array, `ds`, containing observation names, `ObservationNames` has those names.
- If the data is provided in matrices or a dataset array without observation names, then `ObservationNames` is an empty cell array.

## Data Types

cell

## PredictorNames

Names of the variables that you use as predictors in the fit, stored as a cell array of strings that has the same length as `NumPredictors`.

## Data Types

cell

# LinearMixedModel

---

## ResponseName

Name of the variable used as the response variable in the fit, stored as a character string.

## Data Types

char

## Rsquared

Proportion of variability in the response explained by the fitted model, stored as a structure. It is the multiple correlation coefficient or R-squared. Rsquared has two fields.

Ordinary

R-squared value, stored as a scalar value in a structure.  
Rsquared.Ordinary = 1 - SSE./SST

Adjusted

R-squared value adjusted for the number of fixed-effects coefficients, stored as a scalar value in a structure.  
Rsquared.Adjusted = 1 - (SSE./SST)\*(DFT./DFE),  
where DFE = n - p, DFT = n - 1, and n is the total number of observations, p is the number of fixed-effects coefficients.

## Data Types

struct

## SSE

Error sum of squares, that is, sum of the squared conditional residuals, stored as a positive scalar value.

$SSE = \sum (y - F)^2$ , where  $y$  is the response vector, and  $F$  is the fitted conditional response of the linear mixed-effects model. The conditional model has contributions from both fixed and random effects.

## Data Types

double

## SSR

Regression sum of squares, that is, the sum of squares explained by the linear mixed-effects regression, stored as a positive scalar value. It is the sum of squared deviations of the conditional fitted values from their mean.

$SSR = \sum (F - \text{mean}(F))^2$ , where  $F$  is the fitted conditional response of the linear mixed-effects model. The conditional model has contributions from both fixed and random effects.

## Data Types

double

## SST

Total sum of squares, that is, the sum of the squared deviations of the observed response values from their mean, stored as a positive scalar value.

$SST = \sum (y - \text{mean}(y))^2 = SSR + SSE$ , where  $y$  is the response vector.

## Data Types

double

## Variables

Variables, stored as a dataset array.

- If the fit is based on a dataset array `ds`, then `Variables` is identical to `ds`.

# LinearMixedModel

---

- If the fit is based on matrix input, then `Variables` is a dataset array containing all the variables in the predictor matrix or matrices, and response variable.

## VariableInfo

Information about the variables used in the fit, stored as a dataset array.

`VariableInfo` has one row for each variable and contains the following four columns.

<code>Class</code>	Class of the variable ('double', 'cell', 'nominal', and so on).
<code>Range</code>	Value range of the variable. <ul style="list-style-type: none"><li>• For a numerical variable, it is a two-element vector of the form <code>[min,max]</code>.</li><li>• For a cell or categorical variable, it is a cell or categorical array containing all unique values of the variable.</li></ul>
<code>InModel</code>	<code>true</code> , if the variable is a predictor in the fitted model. <code>false</code> , if the variable is not in the fitted model.
<code>IsCategorical</code>	<code>true</code> , if the variable has a type that is treated as a categorical predictor, such as cell, logical, or categorical, or if it is specified as categorical by the 'Categorical' name-value pair argument of the fit



method.false, if it is a continuous predictor.

## VariableNames

Names of the variables used in the fit, stored as a cell array of strings.

- If sample data is in a dataset array `ds`, `VariableNames` contains the names of the variables in `ds`.
- If sample data is in matrix format, then `VariableInfo` includes variable names you supply while fitting the model. If you do not supply the variable names, then `VariableInfo` contains the default names.

## Data Types

cell

## Methods

<code>anova</code>	Analysis of variance for linear mixed-effects model
<code>coefCI</code>	Confidence intervals for coefficients of linear mixed-effects model
<code>coefTest</code>	Hypothesis test on fixed and random effects of linear mixed-effects model
<code>compare</code>	Compare linear mixed-effects models
<code>covarianceParameters</code>	Extract covariance parameters of linear mixed-effects model
<code>designMatrix</code>	Fixed- and random-effects design matrices

# LinearMixedModel

---

<code>disp</code>	Display linear mixed-effects model
<code>fit</code>	Fit linear mixed-effects model using dataset arrays
<code>fitmatrix</code>	Fit linear mixed-effects model using design matrices
<code>fitted</code>	Fitted responses from a linear mixed-effects model
<code>fixedEffects</code>	Estimates of fixed effects and related statistics
<code>plotResiduals</code>	Plot residuals of linear mixed-effects model
<code>predict</code>	Predict response of linear mixed-effects model
<code>random</code>	Generate random responses from fitted linear mixed-effects model
<code>randomEffects</code>	Estimates of random effects and related statistics
<code>residuals</code>	Residuals of fitted linear mixed-effects model
<code>response</code>	Response vector of the linear mixed-effects model

## Definitions

### Formula

In general, a formula for model specification is a string of the form `'y ~ terms'`. For the linear mixed-effects models, this formula is in the form `'y ~ fixed + (random1|grouping1) + ... + (randomR|groupingR)'`, where `fixed` and `random` contain the fixed-effects and the random-effects terms. Suppose a dataset array `ds` contains the following:

- A response variable,  $y$
- Predictor variables,  $X_j$ , which can be continuous or grouping variables
- Grouping variables,  $g_1, g_2, \dots, g_R$ ,

where the grouping variables in  $X_j$  and  $g_r$  can be categorical, logical, character arrays, or cell arrays of strings.

Then, in a formula of the form, ' $y \sim \text{fixed} + (\text{random}_1 | g_1) + \dots + (\text{random}_R | g_R)$ ', the term `fixed` corresponds to a specification of the fixed-effects design matrix  $X$ , `random1` is a specification of the random-effects design matrix  $Z_1$  corresponding to grouping variable  $g_1$ , and similarly `randomR` is a specification of the random-effects design matrix  $Z_R$  corresponding to grouping variable  $g_R$ . You can express the fixed and random terms using Wilkinson notation.

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$X^k$ , where $k$ is a positive integer	$X, X^2, \dots, X^k$
$X1 + X2$	$X1, X2$
$X1 * X2$	$X1, X2, X1.*X2$ (elementwise multiplication of $X1$ and $X2$ )
$X1 : X2$	$X1.*X2$ only
$- X2$	Do not include $X2$
$X1 * X2 + X3$	$X1, X2, X3, X1 * X2$
$X1 + X2 + X3 + X1 : X2$	$X1, X2, X3, X1 * X2$
$X1 * X2 * X3 - X1 : X2 : X3$	$X1, X2, X3, X1 * X2, X1 * X3, X2 * X3$
$X1 * (X2 + X3)$	$X1, X2, X3, X1 * X2, X1 * X3$

# LinearMixedModel

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`. Here are some examples for linear mixed-effects model specification.

Formula	Description
'y ~ X1 + X2'	Fixed effects for the intercept, X1 and X2. This is equivalent to 'y ~ 1 + X1 + X2'.
'y ~ -1 + X1 + X2'	Fixed effects for X1 and X2. The implicit intercept term is suppressed by including <code>-1</code> .
'y ~ 1 + (1   g1)'	Intercept plus random effect for each level of the grouping variable <code>g1</code> .
'y ~ X1 + (1   g1)'	Random intercept model with a fixed slope.
'y ~ X1 + (X1   g1)'	Random intercept and slope, with possible correlation between them.
'y ~ X1 + (1   g1) + (-1 + X1   g1)'	Independent random intercept and slope.
'y ~ 1 + (1   g1) + (1   g2) + (1   g1:g2)'	Random intercept model with independent main effects for <code>g1</code> and <code>g2</code> , plus an independent interaction effect.

## Copy Semantics

Value. To learn how value classes affect copy operations, see [Copying Objects in the MATLAB documentation](#).

## Examples

### Random Intercept Model with Categorical Predictor

Load the sample data.

```
load flu
```

The `flu` dataset array has a `Date` variable, and 10 variables containing estimated influenza rates (in 9 different regions, estimated from Google searches, plus a nationwide estimate from the Center for Disease Control and Prevention, CDC).

To fit a linear-mixed effects model, your data must be in a properly formatted dataset array. To fit a linear mixed-effects model with the influenza rates as the responses and region as the predictor variable, combine the nine columns corresponding to the regions into a tall array. The new dataset array, `flu2`, must have the response variable, `FluRate`, the nominal variable, `Region`, that shows which region each estimate is from, and the grouping variable `Date`.

```
flu2 = stack(flu,2:10, 'NewDataVarName', 'FluRate', ...
            'IndVarName', 'Region');
flu2.Date = nominal(flu2.Date);
```

Fit a linear mixed-effects model with fixed effects for region and a random intercept that varies by `Date`.

Because region is a nominal variable, `fitlme` takes the first region, NE, as the reference and creates eight dummy variables representing the other eight regions. For example,  $I[MidAtl]$  is the dummy variable representing the region `MidAtl`. For details, see “Dummy Indicator Variables” on page 2-55.

The corresponding model is

$$y_{im} = \beta_0 + \beta_1 I[MidAtl]_i + \beta_2 I[ENCentral]_i + \beta_3 I[WNCentral]_i + \beta_4 I[SAtl]_i \\ + \beta_5 I[ESCentral]_i + \beta_6 I[WSCentral]_i + \beta_7 I[Mtn]_i + \beta_8 I[Pac]_i + b_{0m} + \varepsilon_{im}, \quad m = 1, 2, \dots, 52,$$

where  $y_{im}$  is the observation  $i$  for level  $m$  of grouping variable `Date`,  $\beta_j$ ,  $j = 0, 1, \dots, 8$ , are the fixed-effects coefficients,  $b_{0m}$  is the random effect for level  $m$  of the grouping variable `Date`, and  $\varepsilon_{im}$  is the observation error for observation  $i$ . The random effect has the prior distribution,  $b \sim N(0, \sigma_b^2)$  and the error term has the distribution,  $\varepsilon \sim N(0, \sigma_\varepsilon^2)$ .

# LinearMixedModel

---

```
lme = fitlme(flu2, 'FluRate ~ 1 + Region + (1|Date)')
```

Linear mixed-effects model fit by ML

Model information:

Number of observations	468
Fixed effects coefficients	9
Random effects coefficients	52
Covariance parameters	2

Formula:

```
FluRate ~ 1 + Region + (1|Date)
```

Model fit statistics:

AIC	BIC	LogLikelihood	Deviance
318.71	364.35	-148.36	296.71

Fixed effects coefficients (95% CIs):

Name	Estimate	SE	tStat	DF
'(Intercept)'	1.2233	0.096678	12.654	459
'Region_MidAtl'	0.010192	0.052221	0.19518	459
'Region_ENCentral'	0.051923	0.052221	0.9943	459
'Region_WNCentral'	0.23687	0.052221	4.5359	459
'Region_SAtl'	0.075481	0.052221	1.4454	459
'Region_ESCentral'	0.33917	0.052221	6.495	459
'Region_WSCentral'	0.069	0.052221	1.3213	459
'Region_Mtn'	0.046673	0.052221	0.89377	459
'Region_Pac'	-0.16013	0.052221	-3.0665	459

Random effects covariance parameters (95% CIs):

Group: Date (52 Levels)

Name1	Name2	Type	Estimate	Lower	Upper
'(Intercept)'	'(Intercept)'	'std'	0.6443	0.588	0.701

Group: Error

Name	Estimate	Lower	Upper
'Res Std'	0.26627	0.24878	0.285

The  $p$ -values 7.3324e-06 and 2.1623e-10 respectively show that the fixed effects of the flu rates in regions `WNCentral` and `ESCentral` are significantly different relative to the flu rates in region `NE`.

The confidence limits for the standard deviation of the random-effects term,  $\sigma_b^2$ , do not include 0 (0.5297, 0.78368), which indicates that the random-effects term is significant. You can also test the significance of the random-effects terms using the `compare` method.

The estimated value of an observation is the sum of the fixed effects and the random-effect value at the grouping variable level corresponding to that observation. For example, the estimated best linear unbiased predictor (BLUP) of the flu rate for region `WNCentral` in week `10/9/2005` is

$$\begin{aligned}\hat{y}_{WNCentral,10/9/2005} &= \hat{\beta}_0 + \hat{\beta}_3 I[WNCentral] + \hat{b}_{10/9/2005} \\ &= 1.2233 + 0.23687 - 0.1718 \\ &= 1.28837.\end{aligned}$$

This is the fitted conditional response, since it includes contribution to the estimate from both the fixed and random effects. You can compute this value as follows.

```
beta = fixedEffects(lme);
[~,~,STATS] = randomEffects(lme); % Compute the random-effects statistics
STATS.Level = nominal(STATS.Level);
y_hat = beta(1) + beta(4) + STATS.Estimate(STATS.Level=='10/9/2005')
```

```
y_hat =
    1.2884
```

You can simply display the fitted value using the `fitted` method.

```
F = fitted(lme);
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')
```

```
ans =
```

# LinearMixedModel

---

```
1.2884
```

Compute the fitted marginal response for region WNCentral in week 10/9/2005.

```
F = fitted(lme, 'Conditional', false);  
F(flu2.Date == '10/9/2005' & flu2.Region == 'WNCentral')
```

```
ans =
```

```
1.4602
```

## Linear Mixed-Effects Model with a Random Slope

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower and cylinders, and potentially correlated random effect for intercept and acceleration grouped by model year. This model corresponds to

$$MPG_{im} = \beta_0 + \beta_1 Acc_i + \beta_2 HP + b_{0m} + b_{1m} Acc_{im} + \varepsilon_{im}, \quad m = 1, 2, 3,$$

with the random-effects terms having the following prior distribution

$$b_m = \begin{pmatrix} b_{0m} \\ b_{1m} \end{pmatrix} \sim N \left( \mathbf{0}, \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} \\ \sigma_{0,1} & \sigma_1^2 \end{pmatrix} \right),$$

where  $D(\theta)$  is the covariance matrix.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];  
Z = [ones(406,1) Acceleration];  
Model_Year = nominal(Model_Year);
```



```
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables. Use the 'fminunc' optimization algorithm.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...  
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...  
{{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'},...  
'FitMethod','REML')
```

```
lme =
```

```
Linear mixed-effects model fit by REML
```

```
Model information:
```

Number of observations	392
Fixed effects coefficients	3
Random effects coefficients	26
Covariance parameters	4

```
Formula:
```

```
y ~ Intercept + Acceleration + Horsepower + (Intercept + Acceleration | Model_Year)
```

```
Model fit statistics:
```

AIC	BIC	LogLikelihood	Deviance
2202.9	2230.7	-1094.5	2188.9

```
Fixed effects coefficients (95% CIs):
```

Name	Estimate	SE	tStat	DF	p
'Intercept'	50.064	2.3176	21.602	389	1.0e-04
'Acceleration'	-0.57897	0.13843	-4.1825	389	3.0e-05
'Horsepower'	-0.16958	0.0073242	-23.153	389	3.0e-05

```
Random effects covariance parameters (95% CIs):
```

```
Group: Model_Year (13 Levels)
```

Name1	Name2	Type	Estimate
-------	-------	------	----------

# LinearMixedModel

---

```
'Intercept'          'Intercept'          'std'          3.72
'Acceleration'       'Intercept'          'corr'         -0.8769
'Acceleration'       'Acceleration'       'std'          0.3593
```

Group: Error

```
Name          Estimate    Lower    Upper
'Res Std'     3.6913     3.4331   3.9688
```

The fixed effects coefficients display includes the estimate, standard errors (SE), and the 95% confidence interval limits (Lower and Upper). The  $p$ -values for (pValue) indicate that all three fixed-effects coefficients are significant.

The confidence intervals for the standard deviations and the correlation between the random effects for intercept and acceleration do not include zeros, hence they seem significant. Use the `compare` method to test for the random effects.

Display the covariance matrix of the estimated fixed-effects coefficients.

```
lme.CoefficientCovariance
```

```
ans =
```

```
 5.3711  -0.2809  -0.0126
-0.2809   0.0192   0.0005
-0.0126   0.0005   0.0001
```

The diagonal elements show the variances of the fixed-effects coefficient estimates. For example, the variance of the estimate of the intercept is 5.3711. Note that the standard errors of the estimates are the square roots of the variances. For example, the standard error of the intercept is 2.3176, which is `sqrt(5.3711)`.

The off-diagonal elements show the correlation between the fixed-effects coefficient estimates. For example, the correlation between the intercept and acceleration is  $-0.2809$  and the correlation between acceleration and horsepower is 0.0005.

Display the coefficient of determination for the model.

```
lme.Rsquared
```

```
ans =
```

```
  Ordinary: 0.7826
```

```
  Adjusted: 0.7815
```

The adjusted value is the R-squared value adjusted for the number of predictors in the model.

## See Also

```
fitlme | fitlmematrix
```

# linhpytest

---

**Purpose** Linear hypothesis test

**Syntax** `p = linhpytest(beta,COVB,c,H,dfe)`  
`[p,t,r] = linhpytest(...)`

**Description** `p = linhpytest(beta,COVB,c,H,dfe)` returns the  $p$  value  $p$  of a hypothesis test on a vector of parameters. `beta` is a vector of  $k$  parameter estimates. `COVB` is the  $k$ -by- $k$  estimated covariance matrix of the parameter estimates. `c` and `H` specify the null hypothesis in the form  $H*b = c$ , where `b` is the vector of unknown parameters estimated by `beta`. `dfe` is the degrees of freedom for the `COVB` estimate, or `Inf` if `COVB` is known rather than estimated.

`beta` is required. The remaining arguments have default values:

- `COVB = eye(k)`
- `c = zeros(k,1)`
- `H = eye(K)`
- `dfe = Inf`

If `H` is omitted, `c` must have  $k$  elements and it specifies the null hypothesis values for the entire parameter vector.

---

**Note** The following functions return outputs suitable for use as the `COVB` input argument to `linhpytest`: `nlinfit`, `coxphfit`, `glmfit`, `mnrfit`, `regstats`, `robustfit`. `nlinfit` returns `COVB` directly; the other functions return `COVB` in `stats.covb`.

---

`[p,t,r] = linhpytest(...)` also returns the test statistic `t` and the rank `r` of the hypothesis matrix `H`. If `dfe` is `Inf` or is not given, `t*r` is a chi-square statistic with `r` degrees of freedom. If `dfe` is specified as a finite value, `t` is an  $F$  statistic with `r` and `dfe` degrees of freedom.

`linhpytest` performs a test based on an asymptotic normal distribution for the parameter estimates. It can be used after any estimation

procedure for which the parameter covariances are available, such as `regstats` or `glmfit`. For linear regression, the  $p$ -values are exact. For other procedures, the  $p$ -values are approximate, and may be less accurate than other procedures such as those based on a likelihood ratio.

## Examples

Fit a multiple linear model to the data in `hald.mat`:

```
load hald
stats = regstats(heat,ingredients,'linear');
beta = stats.beta
beta =
    62.4054
     1.5511
     0.5102
     0.1019
    -0.1441
```

Perform an  $F$ -test that the last two coefficients are both 0:

```
SIGMA = stats.covb;
dfe = stats.fstat.dfe;
H = [0 0 0 1 0;0 0 0 0 1];
c = [0;0];
[p,F] = linhpytest(beta,SIGMA,c,H,dfe)
p =
    0.4668
F =
    0.8391
```

## See Also

`regstats` | `glmfit` | `robustfit` | `mnrfit` | `nlinfit` | `coxphfit`

# linkage

---

## Purpose

Agglomerative hierarchical cluster tree

## Syntax

```
Z = linkage(X)
Z = linkage(X,method)
Z = linkage(X,method,metric)
Z = linkage(X,method,pdist_inputs)
Z = linkage(X,method,metric,'savememory',value)
Z = linkage(Y)
Z = linkage(Y,method)
```

## Description

`Z = linkage(X)` returns a matrix `Z` that encodes a tree of hierarchical clusters of the rows of the real matrix `X`.

`Z = linkage(X,method)` creates the tree using the specified *method*, where *method* describes how to measure the distance between clusters.

`Z = linkage(X,method,metric)` performs clustering using the distance measure *metric* to compute distances between the rows of `X`.

`Z = linkage(X,method,pdist_inputs)` passes parameters to the `pdist` function, which is the function that computes the distance between rows of `X`.

`Z = linkage(X,method,metric,'savememory',value)` uses a memory-saving algorithm when *value* is 'true', and uses the standard algorithm when *value* is 'false'.

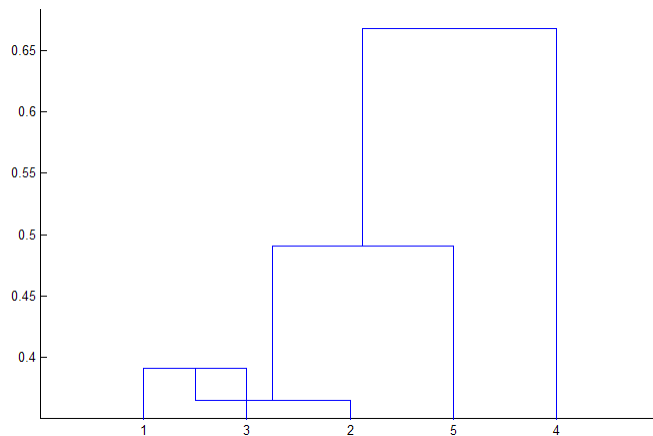
`Z = linkage(Y)` uses a vector representation `Y` of a distance matrix. `Y` can be a distance matrix as computed by `pdist`, or a more general dissimilarity matrix conforming to the output format of `pdist`.

`Z = linkage(Y,method)` creates the tree using the specified *method*, where *method* describes how to measure the distance between clusters.

## Tips

- Computing `linkage(Y)` can be slow when `Y` is a vector representation of the distance matrix. For the 'centroid', 'median', and 'ward' methods, `linkage` checks whether `Y` is a Euclidean distance. Avoid this time-consuming check by passing in `X` instead of `Y`.

- The centroid and median methods can produce a cluster tree that is not monotonic. This occurs when the distance from the union of two clusters,  $r$  and  $s$ , to a third cluster is less than the distance between  $r$  and  $s$ . In this case, in a dendrogram drawn with the default orientation, the path from a leaf to the root node takes some downward steps. To avoid this, use another method. The following image shows a nonmonotonic cluster tree.



In this case, cluster 1 and cluster 3 are joined into a new cluster, while the distance between this new cluster and cluster 2 is less than the distance between cluster 1 and cluster 3. This leads to a nonmonotonic tree.

- You can provide the output  $Z$  to other functions including `dendrogram` to display the tree, `cluster` to assign points to clusters, `inconsistent` to compute inconsistent measures, and `cophenet` to compute the cophenetic correlation coefficient.

## Input Arguments

### **X**

Matrix with two or more rows. The rows represent observations, the columns represent categories or dimensions.

### **method**

# linkage

---

Algorithm for computing distance between clusters.

Method	Description
'average'	Unweighted average distance (UPGMA)
'centroid'	Centroid distance (UPGMC), appropriate for Euclidean distances only
'complete'	Furthest distance
'median'	Weighted center of mass distance (WPGMC), appropriate for Euclidean distances only
'single'	Shortest distance
'ward'	Inner squared distance (minimum variance algorithm), appropriate for Euclidean distances only
'weighted'	Weighted average distance (WPGMA)

**Default:** 'single'

## **metric**

Any distance metric that the `pdist` function accepts.

Metric	Description
'euclidean'	Euclidean distance (default).
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between rows in <code>X</code> is scaled by dividing by the corresponding element of the standard deviation <code>S=nanstd(X)</code> . To specify another value for <code>S</code> , use <code>D=pdist(X, 'seuclidean', S)</code> .
'cityblock'	City block metric.



Metric	Description
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use <code>D = pdist(X, 'minkowski', P)</code> , where <code>P</code> is a scalar
'chebychev'	Chebychev distance (maximum coordinate difference).
'mahalanobis'	Mahalanobis distance, using the sample covariance of <code>X</code> as computed by <code>nancov</code> . To compute the distance with a different covariance, use <code>D = pdist(X, 'mahalanobis', C)</code> , where the matrix <code>C</code> is symmetric and positive definite.
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
custom distance function	<p>A distance function specified using <code>@</code>:</p> $D = \text{pdist}(X, @\text{distfun})$ <p>A distance function must be of form</p> $d2 = \text{distfun}(XI, XJ)$ <p>taking as arguments a 1-by-<math>n</math> vector <math>XI</math>, corresponding to a single row of <math>X</math>, and an <math>m2</math>-by-<math>n</math> matrix <math>XJ</math>, corresponding to multiple rows of <math>X</math>. <code>distfun</code> must accept a matrix <math>XJ</math></p>

# linkage

---

Metric	Description
	with an arbitrary number of rows. <code>distfun</code> must return an $m2$ -by-1 vector of distances <code>d2</code> , whose $k$ th element is the distance between $XI$ and $XJ(k, :)$ .

**Default:** 'euclidean'

## **pdist\_inputs**

A cell array of parameters accepted by the `pdist` function. For example, to set the *metric* to `minkowski` and use an exponent of 5, set `pdist_inputs` to {'minkowski', 5}.

## **savememory**

A string, either 'on' or 'off'. When applicable, the 'on' setting causes `linkage` to construct clusters without computing the distance matrix. `savememory` is applicable when:

- `linkage` is 'centroid', 'median', or 'ward'
- `distance` is 'euclidean' (default)

When `savememory` is 'on', `linkage` run time is proportional to the number of dimensions (number of columns of  $X$ ). When `savememory` is 'off', `linkage` memory requirement is proportional to  $N^2$ , where  $N$  is the number of observations. So choosing the best (least-time) setting for `savememory` depends on the problem dimensions, number of observations, and available memory. The default `savememory` setting is a rough approximation of an optimal setting.

**Default:** 'on' when  $X$  has 20 columns or fewer, or the computer does not have enough memory to store the distance matrix; otherwise 'off'

## **Y**

A vector of distances with the same format as the output of the `pdist` function:

- A row vector of length  $m(m-1)/2$ , corresponding to pairs of observations in a matrix  $X$  with  $m$  rows
- Distances arranged in the order  $(2,1)$ ,  $(3,1)$ , ...,  $(m,1)$ ,  $(3,2)$ , ...,  $(m,2)$ , ...,  $(m,m-1)$

$Y$  can be a more general dissimilarity matrix conforming to the output format of `pdist`.

## Output Arguments

### Z

$Z$  is a  $(m-1)$ -by-3 matrix, where  $m$  is the number of observations in the original data. Columns 1 and 2 of  $Z$  contain cluster indices linked in pairs to form a binary tree. The leaf nodes are numbered from 1 to  $m$ . Leaf nodes are the singleton clusters from which all higher clusters are built. Each newly-formed cluster, corresponding to row  $Z(I, :)$ , is assigned the index  $m+I$ .  $Z(I, 1:2)$  contains the indices of the two component clusters that form cluster  $m+I$ . There are  $m-1$  higher clusters which correspond to the interior nodes of the clustering tree.  $Z(I, 3)$  contains the linkage distances between the two clusters merged in row  $Z(I, :)$ .

For example, suppose there are 30 initial nodes and at step 12 cluster 5 and cluster 7 are combined. Suppose their distance at that time is 1.5. Then  $Z(12, :)$  will be  $[5, 7, 1.5]$ . The newly formed cluster will have index  $12 + 30 = 42$ . If cluster 42 appears in a later row, it means the cluster created at step 12 is being combined into some larger cluster.

## Definitions

### Linkages

The following notation is used to describe the linkages used by the various methods:

- Cluster  $r$  is formed from clusters  $p$  and  $q$ .
- $n_r$  is the number of objects in cluster  $r$ .
- $x_{ri}$  is the  $i$ th object in cluster  $r$ .

# linkage

---

- *Single linkage*, also called *nearest neighbor*, uses the smallest distance between objects in the two clusters:

$$d(r, s) = \min(\text{dist}(x_{ri}, x_{sj}), i \in (1, \dots, n_r), j \in (1, \dots, n_s))$$

- *Complete linkage*, also called *furthest neighbor*, uses the largest distance between objects in the two clusters:

$$d(r, s) = \max(\text{dist}(x_{ri}, x_{sj}), i \in (1, \dots, n_r), j \in (1, \dots, n_s))$$

- *Average linkage* uses the average distance between all pairs of objects in any two clusters:

$$d(r, s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} \text{dist}(x_{ri}, x_{sj})$$

- *Centroid linkage* uses the Euclidean distance between the centroids of the two clusters:

$$d(r, s) = \|\bar{x}_r - \bar{x}_s\|_2$$

where

$$\bar{x}_r = \frac{1}{n_r} \sum_{i=1}^{n_r} x_{ri}$$

- *Median linkage* uses the Euclidean distance between weighted centroids of the two clusters,

$$d(r, s) = \|\tilde{x}_r - \tilde{x}_s\|_2$$

where  $\tilde{x}_r$  and  $\tilde{x}_s$  are weighted centroids for the clusters  $r$  and  $s$ . If cluster  $r$  was created by combining clusters  $p$  and  $q$ ,  $\tilde{x}_r$  is defined recursively as

$$\tilde{x}_r = \frac{1}{2}(\tilde{x}_p + \tilde{x}_q)$$

- *Ward's linkage* uses the incremental sum of squares; that is, the increase in the total within-cluster sum of squares as a result of joining two clusters. The within-cluster sum of squares is defined as the sum of the squares of the distances between all objects in the cluster and the centroid of the cluster. The sum of squares measure is equivalent to the following distance measure  $d(r,s)$ , which is the formula linkage uses:

$$d(r,s) = \sqrt{\frac{2n_r n_s}{(n_r + n_s)}} \|\bar{x}_r - \bar{x}_s\|_2,$$

where

- $\|\cdot\|_2$  is Euclidean distance
- $\bar{x}_r$  and  $\bar{x}_s$  are the centroids of clusters  $r$  and  $s$
- $n_r$  and  $n_s$  are the number of elements in clusters  $r$  and  $s$

In some references the Ward linkage does not use the factor of 2 multiplying  $n_r n_s$ . The linkage function uses this factor so the distance between two singleton clusters is the same as the Euclidean distance.

- *Weighted average linkage* uses a recursive definition for the distance between two clusters. If cluster  $r$  was created by combining clusters  $p$  and  $q$ , the distance between  $r$  and another cluster  $s$  is defined as the average of the distance between  $p$  and  $s$  and the distance between  $q$  and  $s$ :

$$d(r,s) = \frac{(d(p,s) + d(q,s))}{2}$$

# linkage

---

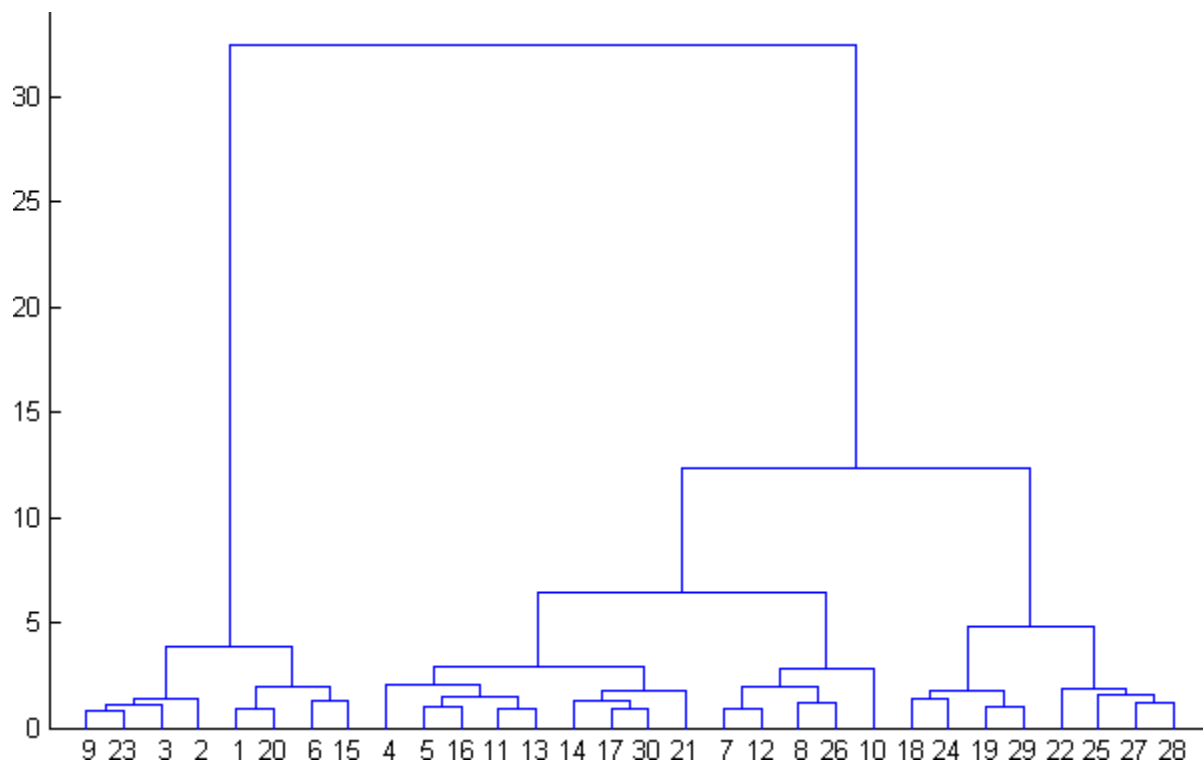
## Examples

Compute four clusters of the Fisher iris data using Ward linkage and ignoring species information, and see how the cluster assignments correspond to the three species.

```
load fisheriris
Z = linkage(meas,'ward','euclidean');
c = cluster(Z,'maxclust',4);
crosstab(c,species)
firstfive = Z(1:5,:) % first 5 rows of Z
dendrogram(Z)
```

```
ans =
     0     25     1
     0     24    14
     0      1    35
    50      0     0
```

```
firstfive =
 102.0000  143.0000         0
   8.0000   40.0000    0.1000
   1.0000   18.0000    0.1000
  10.0000   35.0000    0.1000
 129.0000  133.0000    0.1000
```

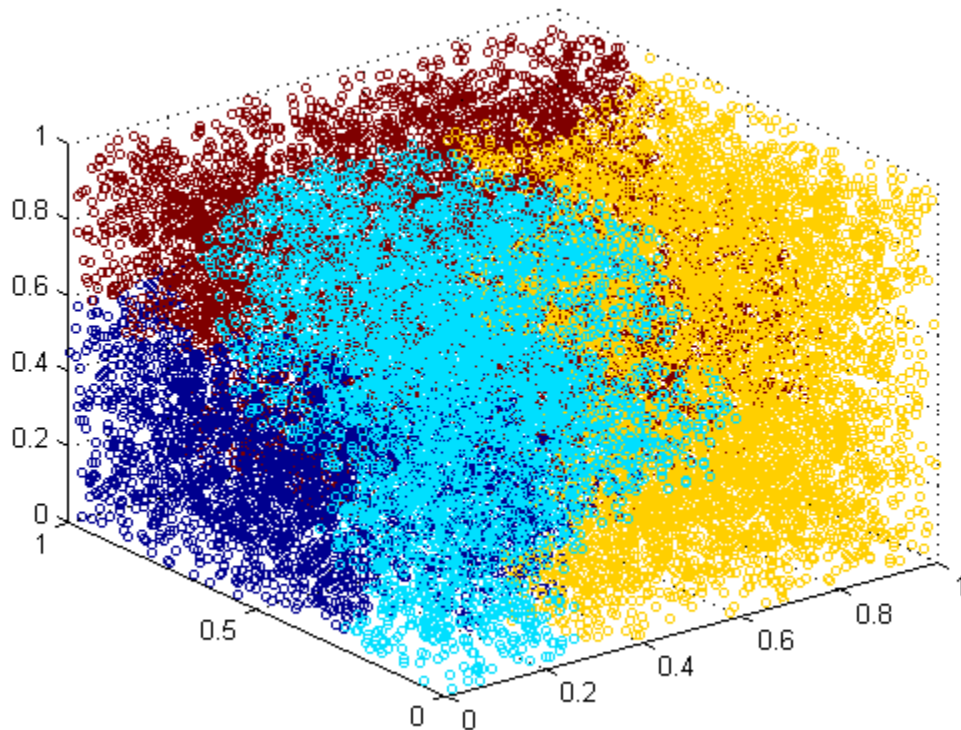


Create a hierarchical cluster tree for a data with 20000 observations using Ward's linkage. If you set `savememory` to `'off'`, you can get an out-of-memory error if your machine doesn't have enough memory to hold the distance matrix. Cluster the data into four groups and plot the result.

```
X = rand(20000,3);  
Z = linkage(X,'ward','euclidean','savememory','on');  
c = cluster(Z,'maxclust',4);  
scatter3(X(:,1),X(:,2),X(:,3),10,c)
```

# linkage

---



## See Also

[cluster](#) | [clusterdata](#) | [cophenet](#) | [dendrogram](#) | [inconsistent](#) | [kmeans](#) | [pdist](#) | [silhouette](#) | [squareform](#)



**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Logistic probability distribution object

**Description** prob.LogisticDistribution is an object consisting of parameters, a model description, and sample data for a logistic probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Logistic')` creates a logistic probability distribution object using the default parameter values.

`pd = makedist('Logistic','mu',mu,'sigma',sigma)` creates a logistic probability distribution object using the specified parameter values.

### Input Arguments

#### **mu - Mean**

0 (default) | scalar value

Mean of the logistic distribution, specified as a scalar value.

#### **Data Types**

single | double

#### **sigma - Scale parameter**

1 (default) | nonnegative scalar value

Scale parameter of the logistic distribution, specified as a nonnegative scalar value.

#### **Data Types**

single | double

**Properties** **mu**

Mean of the logistic distribution, stored as a scalar value.

# prob.LogisticDistribution

---

## Data Types

single | double

## **sigma**

Scale parameter of the logistic distribution, stored as a nonnegative scalar value.

## Data Types

single | double

## **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

## **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## **NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## **Data Types**

single | double

## **ParameterCovariance**

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## **Data Types**

single | double

## **ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## **Data Types**

char

## **ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## **Data Types**

logical

## **ParameterNames**

# prob.LogisticDistribution

---

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object

random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Logistic Distribution

The logistic distribution is used for growth models and in logistic regression. It has longer tails and a higher kurtosis than the normal distribution.

The logistic distribution uses the following parameters.

Parameter	Description	Support
mu	Mean	$-\infty < \mu < \infty$
sigma	Scale parameter	$\sigma \geq 0$

The probability density function (pdf) is

# prob.LogisticDistribution

---

$$f(x | \mu, \sigma) = \frac{\exp\left\{\frac{x - \mu}{\sigma}\right\}}{\sigma \left(1 + \exp\left\{\frac{x - \mu}{\sigma}\right\}\right)^2} ; \quad -\infty < x < \infty.$$

## Examples

### Create a Logistic Distribution Object Using Default Parameters

Create a logistic distribution object using the default parameter values.

```
pd = makedist('Logistic')
```

```
pd =
```

```
LogisticDistribution
```

```
Logistic distribution
```

```
mu = 0
```

```
sigma = 1
```

### Create a Logistic Distribution Object Using Specified Parameters

Create a logistic distribution object by specifying parameter values.

```
pd = makedist('Logistic', 'mu',2,'sigma',4)
```

```
pd =
```

```
LogisticDistribution
```

```
Logistic distribution
```

```
mu = 2
```

```
sigma = 4
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

```
7.2552
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Logistic Distribution” on page B-76
- Class Attributes
- Property Attributes

# prob.LoglogisticDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Logistic probability distribution object

**Description** prob.LoglogisticDistribution is an object consisting of parameters, a model description, and sample data for a loglogistic probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Loglogistic')` creates a loglogistic probability distribution object using the default parameter values.

`pd = makedist('Loglogistic','mu',mu,'sigma',sigma)` creates a loglogistic probability distribution object using the specified parameter values .

## Input Arguments

### **mu - Log mean**

0 (default) | positive scalar value

Log mean for the loglogistic distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **sigma - Log scale parameter**

1 (default) | positive scalar value

Log scale parameter for the loglogistic distribution, specified as a positive scalar value.

### **Data Types**

single | double



## Properties

### **mu**

Log mean for the loglogistic distribution, stored as a positive scalar value.

### **Data Types**

single | double

### **sigma**

Log scale parameter for the loglogistic distribution, stored as a positive scalar value.

### **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### **Data Types**

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

### **Data Types**

single | double

### **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

# prob.LoglogisticDistribution

---

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object

# prob.LoglogisticDistribution

---

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions **Loglogistic Distribution**

The loglogistic distribution is closely related to the logistic distribution. If  $x$  is distributed loglogistically with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed logistically with mean and standard deviation. This distribution is often used in survival analysis to model events that experience an initial rate increase, followed by a rate decrease.

The loglogistic distribution uses the following parameters.

Parameter	Description	Support
mu	Log mean	$\mu > 0$
sigma	Log scale parameter	$\sigma > 0$

The probability density function (pdf) is

$$f(x | \mu, \sigma) = \frac{1}{\sigma} \frac{1}{x} \frac{e^z}{(1 + e^z)^2} ; \quad x \geq 0,$$

where  $z = \frac{\log(x) - \mu}{\sigma}$ .

## Examples

### Create a Loglogistic Distribution Object Using Default Parameters

Create a loglogistic distribution object using the default parameter values.

```
pd = makedist('Loglogistic')
```

```
pd =
```

```
LoglogisticDistribution
```

```
Log-Logistic distribution
```

```
mu = 0
```

```
sigma = 1
```

### Create a Loglogistic Distribution Object Using Specified Parameters

Create a loglogistic distribution object by specifying the parameter values.

# prob.LoglogisticDistribution

---

```
pd = makedist('Loglogistic','mu',5,'sigma',2)
```

```
pd =
```

```
LoglogisticDistribution
```

```
Log-Logistic distribution
```

```
mu = 5
```

```
sigma = 2
```

Generate random numbers from the loglogistic distribution and compute their log values.

```
rng(19) % for reproducibility
```

```
x = random(pd,10000,1);
```

```
logx = log(x);
```

Compute the mean of the log values.

```
m = mean(logx)
```

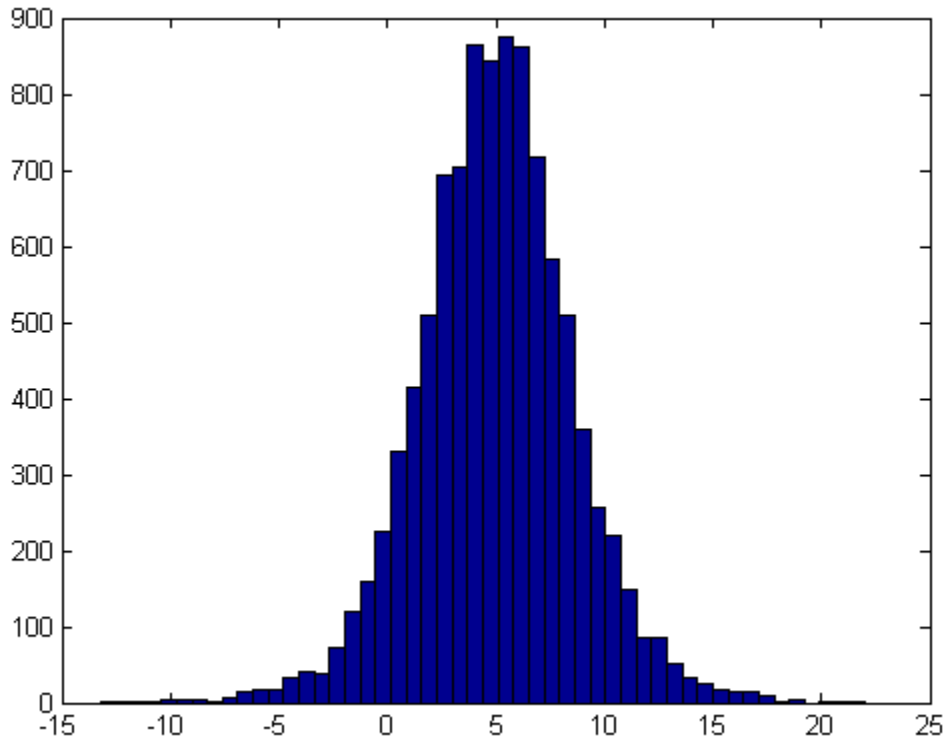
```
m =
```

```
4.9828
```

The mean of the log of  $x$  is equal to the  $\mu$  parameter of  $x$ , since  $x$  has a loglogistic distribution.

Plot  $\log x$ .

```
hist(logx,50)
```



The plot shows that the log values of  $x$  have a logistic distribution.

## See Also

`makedist` | `fitdist` | `dfittool`

## Concepts

- “Loglogistic Distribution” on page B-78
- Class Attributes
- Property Attributes

# logncdf

---

**Purpose** Lognormal cumulative distribution function

**Syntax**

```
p = logncdf(x,mu,sigma)
[p,plo,pup] = logncdf(x,mu,sigma,pcov,alpha)
[p,plo,pup] = logncdf( __ , 'upper' )
```

**Description** `p = logncdf(x,mu,sigma)` returns values at `x` of the lognormal cdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `x`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `x`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

`[p,plo,pup] = logncdf(x,mu,sigma,pcov,alpha)` returns confidence bounds for `p` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies 100(1 - `alpha`)% confidence bounds. The default value of `alpha` is 0.05. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p,plo,pup] = logncdf( __ , 'upper' )` returns the complement of the lognormal cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use 'upper' with any of the previous syntaxes.

`logncdf` computes confidence bounds for `p` using a normal approximation to the distribution of the estimate

$$\frac{x - \hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `p`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal cdf is



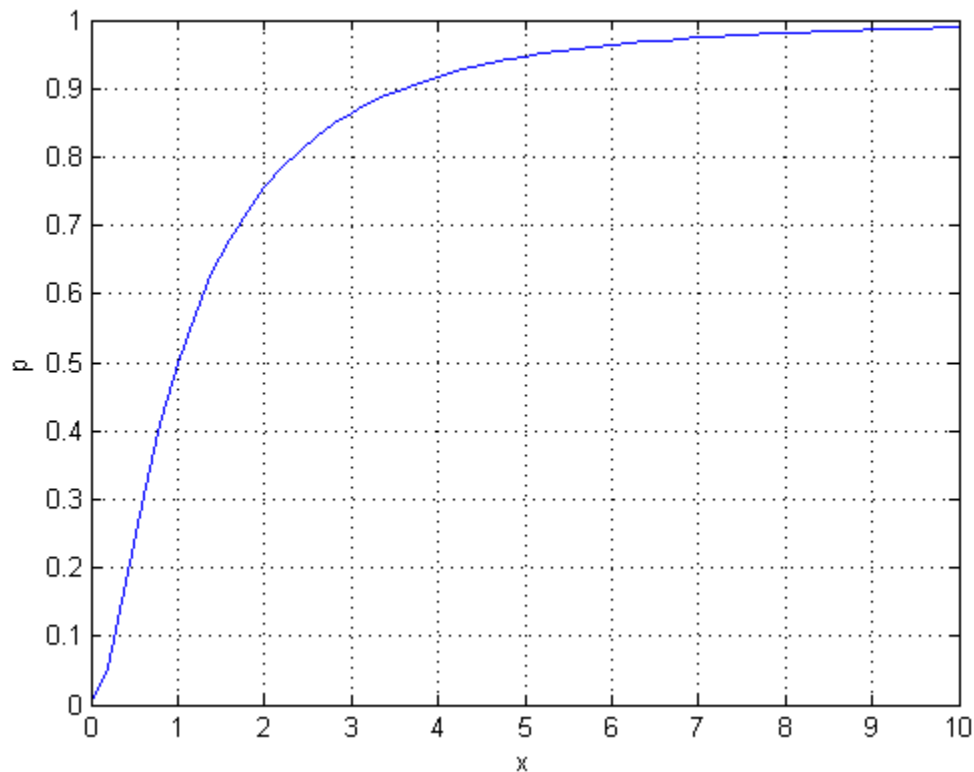
$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x \frac{e^{-\frac{(\ln(t)-\mu)^2}{2\sigma^2}}}{t} dt$$

**Examples****Compute Lognormal Distribution CDF**

```
x = (0:0.2:10);  
y = logncdf(x,0,1);  
plot(x,y);  
grid;  
xlabel('x');  
ylabel('p');
```

# logncdf

---



## References

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 102–105.

## See Also

[cdf](#) | [lognpdf](#) | [logninv](#) | [lognstat](#) | [lognfit](#) | [lognlike](#) | [lognrnd](#)

## How To

- “Lognormal Distribution” on page B-80

**Purpose**

Lognormal parameter estimates

**Syntax**

```
parmhat = lognfit(data)
[parmhat,parmci] = lognfit(data)
[parmhat,parmci] = lognfit(data,alpha)
[...] = lognfit(data,alpha,censoring)
[...] = lognfit(data,alpha,censoring,freq)
[...] = lognfit(data,alpha,censoring,freq,options)
```

**Description**

`parmhat = lognfit(data)` returns a vector of maximum likelihood estimates `parmhat(1) = mu` and `parmhat(2) = sigma` of parameters for a lognormal distribution fitting data. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution.

`[parmhat,parmci] = lognfit(data)` returns 95% confidence intervals for the parameter estimates `mu` and `sigma` in the 2-by-2 matrix `parmci`. The first column of the matrix contains the lower and upper confidence bounds for parameter `mu`, and the second column contains the confidence bounds for parameter `sigma`.

`[parmhat,parmci] = lognfit(data,alpha)` returns  $100(1 - \alpha)\%$  confidence intervals for the parameter estimates, where `alpha` is a value in the range (0 1) specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals.

`[...] = lognfit(data,alpha,censoring)` accepts a Boolean vector `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = lognfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

`[...] = lognfit(data,alpha,censoring,freq,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates when there is censoring. The lognormal fit function accepts an `options`

# lognfit

---

structure which can be created using the function `statset`. Enter `statset('lognfit')` to see the names and default values of the parameters that `lognfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

---

**Note** With no censoring, `lognfit` computes `sigma` using the square root of the unbiased estimator of the variance. With censoring, `sigma` is the maximum likelihood estimate.

---

## Examples

This example generates 100 independent samples of lognormally distributed data with  $\mu = 0$  and  $\sigma = 3$ . `parmhat` estimates  $\mu$  and  $\sigma$  and `parmci` gives 99% confidence intervals around `parmhat`. Notice that `parmci` contains the true values of  $\mu$  and  $\sigma$ .

```
data = lognrnd(0,3,100,1);
[parmhat,parmci] = lognfit(data,0.01)
parmhat =
    -0.2480    2.8902
parmci =
    -1.0071    2.4393
     0.5111    3.5262
```

## See Also

`mle` | `lognlike` | `lognpdf` | `logncdf` | `logninv` | `lognstat` | `lognrnd`

## How To

- “Lognormal Distribution” on page B-80

**Purpose**

Lognormal inverse cumulative distribution function

**Syntax**

```
X = logninv(P,mu,sigma)
[X,XLO,XUP] = logninv(P,mu,sigma,pcov,alpha)
```

**Description**

`X = logninv(P,mu,sigma)` returns values at `P` of the inverse lognormal cdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

`[X,XLO,XUP] = logninv(P,mu,sigma,pcov,alpha)` returns confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies  $100(1 - \alpha)\%$  confidence bounds. The default value of `alpha` is 0.05. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

`logninv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The lognormal inverse function is defined in terms of the lognormal cdf as

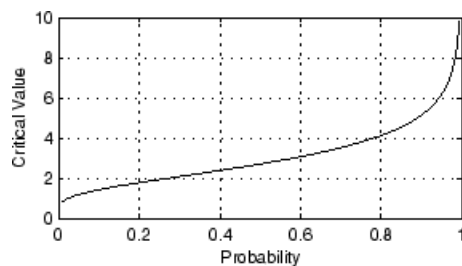
$$x = F^{-1}(p | \mu, \sigma) = \{x : F(x | \mu, \sigma) = p\}$$

where

$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x e^{-\frac{(\ln(t)-\mu)^2}{2\sigma^2}} dt$$

## Examples

```
p = (0.005:0.01:0.995);  
crit = logninv(p,1,0.5);  
plot(p,crit)  
xlabel('Probability'); ylabel('Critical Value'); grid
```



## References

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 102–105.

## See Also

icdf | logncdf | lognpdf | lognstat | lognfit | lognlike | lognrnd

## How To

- “Lognormal Distribution” on page B-80

---

<b>Purpose</b>	Lognormal negative log-likelihood
<b>Syntax</b>	<pre>nlogL = lognlike(params,data) [nlogL,avar] = lognlike(params,data) [...] = lognlike(params,data,censoring) [...] = lognlike(params,data,censoring,freq)</pre>
<b>Description</b>	<p><code>nlogL = lognlike(params,data)</code> returns the negative log-likelihood of data for the lognormal distribution with parameters <code>params</code>. <code>params(1)</code> is the mean of the associated normal distribution, <code>mu</code>, and <code>params(2)</code> is the standard deviation of the associated normal distribution, <code>sigma</code>. The values of <code>mu</code> and <code>sigma</code> are scalars, and the output <code>nlogL</code> is a scalar.</p> <p><code>[nlogL,avar] = lognlike(params,data)</code> returns the inverse of Fisher's information matrix. If the input parameter value in <code>params</code> is the maximum likelihood estimate, <code>avar</code> is its asymptotic variance. <code>avar</code> is based on the observed Fisher's information, not the expected information.</p> <p><code>[...] = lognlike(params,data,censoring)</code> accepts a Boolean vector, <code>censoring</code>, of the same size as <code>data</code>, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.</p> <p><code>[...] = lognlike(params,data,censoring,freq)</code> accepts a frequency vector, <code>freq</code>, of the same size as <code>data</code>. The vector <code>freq</code> typically contains integer frequencies for the corresponding elements in <code>data</code>, but can contain any nonnegative values. Pass in <code>[]</code> for <code>censoring</code> to use its default value.</p>
<b>See Also</b>	<code>lognfit</code>   <code>lognpdf</code>   <code>logncdf</code>   <code>logninv</code>   <code>lognstat</code>   <code>lognrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Lognormal Distribution” on page B-80</li></ul>

# prob.LognormalDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Lognormal probability distribution object

**Description** prob.LognormalDistribution is an object consisting of parameters, a model description, and sample data for a lognormal probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Lognormal')` creates a lognormal probability distribution object using the default parameter values.

`pd = makedist('Lognormal','mu',mu,'sigma',sigma)` creates a lognormal probability distribution object using the specified parameter values.

## Input Arguments

### **mu - Log mean**

0 (default) | scalar value

Log mean for the lognormal distribution, specified as a scalar value. `mu` is the mean of the log of  $x$ , when  $x$  has a lognormal distribution.

### **Data Types**

single | double

### **sigma - Log standard deviation**

1 (default) | nonnegative scalar value

Log standard deviation for the lognormal distribution, specified as a nonnegative scalar value. `sigma` is the standard deviation of the log of  $x$ , when  $x$  has a lognormal distribution.

### **Data Types**

single | double



## Properties

### **mu**

Log mean for the lognormal distribution, stored as a scalar value.

### **Data Types**

single | double

### **sigma**

Log standard deviation for the lognormal distribution, stored as a nonnegative scalar value.

### **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### **Data Types**

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

### **Data Types**

single | double

### **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

# prob.LognormalDistribution

---

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object

# prob.LognormalDistribution

---

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions **Lognormal Distribution**

The lognormal distribution is closely related to the normal distribution. If  $x$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ . The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(x)$  exists only when  $x$  is positive.

The lognormal distribution uses the following parameters.

Parameter	Description	Support
mu	Log mean	$-\infty < \mu < \infty$
sigma	Log standard deviation	$\sigma \geq 0$

The probability density function (pdf) of the lognormal distribution is

$$f(x | \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left\{-\frac{(\ln x - \mu)^2}{2\sigma^2}\right\} ; \quad x > 0.$$

## Examples

### Create a Lognormal Distribution Object Using Default Parameters

Create a lognormal distribution object using the default parameter values.

```
pd = makedist('Lognormal')
```

```
pd =
```

```
LognormalDistribution
```

```
Lognormal distribution
```

```
mu = 0
```

```
sigma = 1
```

### Create a Lognormal Distribution Object Using Specified Parameters

Create a lognormal distribution object by specifying the parameter values.

```
pd = makedist('Lognormal','mu',5,'sigma',2)
```

```
pd =
```

# prob.LognormalDistribution

---

```
LognormalDistribution
```

```
Lognormal distribution  
mu = 5  
sigma = 2
```

Compute the mean of the lognormal distribution.

```
mean(pd)
```

```
ans =
```

```
1.0966e+03
```

The mean of the lognormal distribution is not equal to the mu parameter.

Generate random numbers from the lognormal distribution and compute their log values.

```
rng(47); % for reproducibility  
x = random(pd,10000,1);  
logx = log(x);
```

Compute the mean of the log values.

```
m = mean(logx)
```

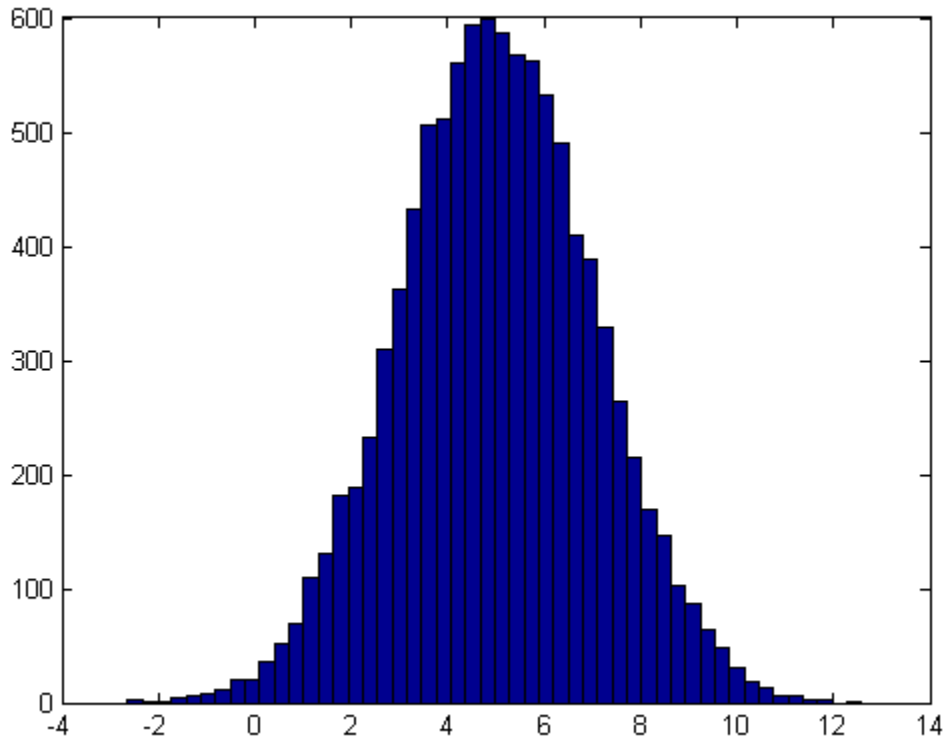
```
m =
```

```
4.9999
```

The mean of the log of x is equal to the mu parameter of x, since x has a lognormal distribution.

Plot logx.

```
hist(logx,50)
```



The plot shows that the log values of  $x$  are normally distributed with a mean equal to 5 and a standard deviation equal to 2.

## See Also

`makedist` | `fitdist` | `dfittool`

## Concepts

- “Lognormal Distribution” on page B-80
- Class Attributes
- Property Attributes

# lognpdf

---

**Purpose** Lognormal probability density function

**Syntax** `Y = lognpdf(X,mu,sigma)`

**Description** `Y = lognpdf(X,mu,sigma)` returns values at `X` of the lognormal pdf with distribution parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X`, `mu`, or `sigma` is expanded to a constant array with the same dimensions as the other inputs.

The lognormal pdf is

$$y = f(x | \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

The normal and lognormal distributions are closely related. If  $X$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(X)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ .

The mean  $m$  and variance  $v$  of a lognormal random variable are functions of  $\mu$  and  $\sigma$  that can be calculated with the `lognstat` function. They are:

$$m = \exp\left(\mu + \sigma^2 / 2\right)$$

$$v = \exp\left(2\mu + \sigma^2\right)\left(\exp\left(\sigma^2\right) - 1\right)$$

A lognormal distribution with mean  $m$  and variance  $v$  has parameters

$$\mu = \log\left(m^2 / \sqrt{v + m^2}\right)$$

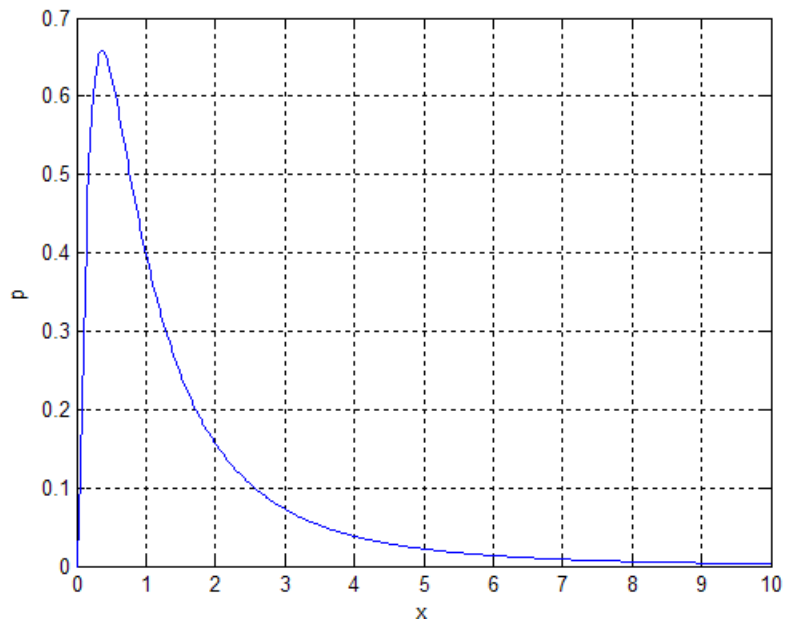
$$\sigma = \sqrt{\log\left(v / m^2 + 1\right)}$$



The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(X)$  exists only when  $X$  is positive.

## Examples

```
x = (0:0.02:10);  
y = lognpdf(x,0,1);  
plot(x,y); grid;  
xlabel('x'); ylabel('p')
```



## References

[1] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.

## See Also

pdf | logncdf | logninv | lognstat | lognfit | lognlike | lognrnd

## How To

- “Lognormal Distribution” on page B-80

# lognrnd

---

**Purpose** Lognormal random numbers

**Syntax**  
`R = lognrnd(mu,sigma)`  
`R = lognrnd(mu,sigma,m,n,...)`  
`R = lognrnd(mu,sigma,[m,n,...])`

**Description** `R = lognrnd(mu,sigma)` returns an array of random numbers generated from the lognormal distribution with parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = lognrnd(mu,sigma,m,n,...)` or `R = lognrnd(mu,sigma,[m,n,...])` generates an `m`-by-`n`-by-... array. The `mu`, `sigma` parameters can each be scalars or arrays of the same size as `R`.

The normal and lognormal distributions are closely related. If  $X$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(X)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ .

The mean  $m$  and variance  $v$  of a lognormal random variable are functions of  $\mu$  and  $\sigma$  that can be calculated with the `lognstat` function. They are:

$$m = \exp\left(\mu + \sigma^2 / 2\right)$$
$$v = \exp\left(2\mu + \sigma^2\right)\left(\exp\left(\sigma^2\right) - 1\right)$$

A lognormal distribution with mean  $m$  and variance  $v$  has parameters

$$\mu = \log\left(m^2 / \sqrt{v + m^2}\right)$$
$$\sigma = \sqrt{\log\left(v / m^2 + 1\right)}$$

**Examples**

Generate one million lognormally distributed random numbers with mean 1 and variance 2:

```
m = 1;  
v = 2;  
mu = log((m^2)/sqrt(v+m^2));  
sigma = sqrt(log(v/(m^2)+1));
```

```
[M,V]= lognstat(mu,sigma)
```

```
M =  
    1  
V =  
    2.0000
```

```
X = lognrnd(mu,sigma,1,1e6);
```

```
MX = mean(X)
```

```
MX =  
    0.9974
```

```
VX = var(X)
```

```
VX =  
    1.9776
```

**References**

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 102–105.

**See Also**

random | lognpdf | logncdf | logninv | lognstat | lognfit | lognlike | normrnd

**How To**

- “Lognormal Distribution” on page B-80

# lognstat

---

**Purpose** Lognormal mean and variance

**Syntax** `[M,V] = lognstat(mu,sigma)`

**Description** `[M,V] = lognstat(mu,sigma)` returns the mean of and variance of the lognormal distribution with parameters `mu` and `sigma`. `mu` and `sigma` are the mean and standard deviation, respectively, of the associated normal distribution. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

The normal and lognormal distributions are closely related. If  $X$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(X)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ .

The mean  $m$  and variance  $v$  of a lognormal random variable are functions of  $\mu$  and  $\sigma$  that can be calculated with the `lognstat` function. They are:

$$m = \exp\left(\mu + \sigma^2 / 2\right)$$
$$v = \exp\left(2\mu + \sigma^2\right)\left(\exp\left(\sigma^2\right) - 1\right)$$

A lognormal distribution with mean  $m$  and variance  $v$  has parameters

$$\mu = \log\left(m^2 / \sqrt{v + m^2}\right)$$
$$\sigma = \sqrt{\log\left(v / m^2 + 1\right)}$$

**Examples** Generate one million lognormally distributed random numbers with mean 1 and variance 2:

```
m = 1;  
v = 2;
```

```
mu = log((m^2)/sqrt(v+m^2));
sigma = sqrt(log(v/(m^2)+1));

[M,V]= lognstat(mu,sigma)
M =
    1
V =
    2.0000

X = lognrnd(mu,sigma,1,1e6);

MX = mean(X)
MX =
    0.9974
VX = var(X)
VX =
    1.9776
```

**References**

[1] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.

**See Also**

lognpdf | logncdf | logninv | lognfit | lognlike | lognrnd

**How To**

• “Lognormal Distribution” on page B-80

# CompactClassificationDiscriminant.logP

---

**Purpose** Log of the unconditional probability density

**Syntax** `lp = logP(obj, Xnew)`

**Description** `lp = logP(obj, Xnew)` returns the log of the unconditional probability density of each row of `Xnew`, computed using the discriminant analysis model `obj`.

**Input Arguments**

**obj**  
Discriminant analysis classifier, produced using `ClassificationDiscriminant.fit`.

**Xnew**  
Matrix where each row represents an observation, and each column represents a predictor. The number of columns in `Xnew` must equal the number of predictors in `obj`.

**Output Arguments**

**lp**  
Column vector with the same number of rows as `Xnew`. Each entry is the logarithm of the unconditional probability density of the corresponding row of `Xnew`.

## Definitions **Unconditional Probability Density**

The unconditional probability density of a point  $x$  of a discriminant analysis model is

$$P(x) = \sum_{k=1}^K P(x, k),$$

where  $P(x, k)$  is the conditional density of the model at  $x$  for class  $k$ , when the total number of classes is  $K$ .

The conditional density  $P(x, k)$  is

$$P(x,k) = P(k)P(x|k),$$

where  $P(k)$  is the prior probability of class  $k$ , and  $P(x|k)$  is the conditional density of  $x$  given class  $k$ . The conditional density function of the multivariate normal with mean  $\mu_k$  and covariance  $\Sigma_k$  at a point  $x$  is

$$P(x|k) = \frac{1}{(2\pi|\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

## Examples

### logP Calculation

Construct a discriminant analysis classifier for the Fisher iris data, and examine its prediction for an average measurement.

Load the Fisher iris data and construct a default discriminant analysis classifier.

```
load fisheriris
mdl = ClassificationDiscriminant.fit(meas,species);
```

Find the log probability of the discriminant model applied to an average iris.

```
logpaverage = logP(mdl,mean(meas))
```

```
logpaverage =
```

```
-1.7254
```

## See Also

CompactClassificationDiscriminant | mahal |

## Concepts

- “Discriminant Analysis” on page 15-3

# ClassificationKNN.loss

---

**Purpose** Loss of  $k$ -nearest neighbor classifier

**Syntax**  
`L = loss(md1,X,Y)`  
`L = loss(md1,X,Y,Name,Value)`

**Description** `L = loss(md1,X,Y)` returns a scalar representing how well `md1` classifies the data in `X`, when `Y` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `md1`.

`L = loss(md1,X,Y,Name,Value)` returns the loss with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**md1**  
*k*-nearest neighbor classifier, created by `ClassificationKNN.fit`.

**X**  
Matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

**Y**  
A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings, with the same number of rows as `X`. Each row of `Y` represents the classification of the corresponding row of `X`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'lossfun'**



Function handle or string representing a loss function. The built-in loss functions are:

- 'binodeviance' — See “Loss Functions” on page 21-1786.
- 'exponential' — See “Loss Functions” on page 21-1786.
- 'classiferror' — Fraction of misclassified observations. See “Loss Functions” on page 21-1786.
- 'mincost' — Smallest misclassification cost as given by the `mdl.Cost` matrix. See “Loss Functions” on page 21-1786.

You can write your own loss function using the syntax described in “Loss Functions” on page 21-1786.

**Default:** 'mincost'

## 'weights'

Numeric vector of length  $N$ , where  $N$  is the number of rows of  $X$ . `weights` are nonnegative. `loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class. When you supply `weights`, `loss` computes weighted classification loss.

**Default:** `ones(N,1)`

## Output Arguments

**L**

Classification error, a scalar. The meaning of the error depends on the values in `weights` and `lossfun`. See “Classification Error” on page 21-1791.

## Definitions

### Classification Error

The default classification error is the fraction of data  $X$  that `mdl` misclassifies, where  $Y$  represents the true classifications.

# ClassificationKNN.loss

---

The weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when `mdl` misclassifies the  $i$ th row of  $X$ , divided by the sum of the weights.

## Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

- 'classiferror' — Predict the label with the largest posterior probability. The loss is then the fraction of misclassified observations.
- 'mincost' — Predict the label with the smallest expected misclassification cost, with expectation taken over the posterior probability, and cost as given by the `Cost` property of the classifier (a matrix). The loss is then the true misclassification cost averaged over the observations.

To write your own loss function, create a function file in this form:

```
function loss = lossfun(C,S,W,COST)
```

- $N$  is the number of rows of  $X$ .
- $K$  is the number of classes in the classifier, represented in the `ClassNames` property.
- $C$  is an  $N$ -by- $K$  logical matrix, with one true per row for the true class. The index for each class is its position in the `ClassNames` property.

- $S$  is an  $N$ -by- $K$  numeric matrix.  $S$  is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from `predict`.
- $W$  is a numeric vector with  $N$  elements, the observation weights. If you pass  $W$ , the elements are normalized to sum to the prior probabilities in the respective classes.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, you can use  $COST = \text{ones}(K) - \text{eye}(K)$ , which means a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `ClassificationKNN.fit`.  $Cost(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $Cost(i, j) = 1$  if  $i \neq j$ , and  $Cost(i, j) = 0$  if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have  $K$  classes. You place the observations into a matrix `Xnew` with one observation per row. The command

```
[label,score,cost] = predict(mdl,Xnew)
```

# ClassificationKNN.loss

---

returns, among other outputs, a `cost` matrix of size `Nobs-by-K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{new}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new}(n))$  is the posterior probability of class  $i$  for observation  $X_{new}(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## Examples

### Loss Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(meas,species,'NumNeighbors',5);
```

Examine the loss of the classifier for a mean observation classified 'versicolor'.

```
X = mean(meas);  
Y = {'versicolor'};  
L = loss(mdl,X,Y)
```

```
L =
```

0

The classifier has no doubt that 'versicolor' is the correct classification (all five nearest neighbors classify as 'versicolor').

## See Also

[ClassificationKNN](#) | [edge](#) | [margin](#) |

## Related Examples

- “Examine the Quality of a KNN Classifier” on page 16-27
- “Predict Classification Based on a KNN Classifier” on page 16-28
- “Modify a KNN Classifier” on page 16-29

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9

# CompactClassificationDiscriminant.loss

---

**Purpose** Classification error

**Syntax**  
`L = loss(obj,X,Y)`  
`L = loss(obj,X,Y,Name,Value)`

**Description** `L = loss(obj,X,Y)` returns a scalar representing how well `obj` classifies the data in `X`, when `Y` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `obj`.

`L = loss(obj,X,Y,Name,Value)` returns the loss with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**obj**  
Discriminant analysis classifier of class `ClassificationDiscriminant` or `CompactClassificationDiscriminant`, typically constructed with `ClassificationDiscriminant.fit`.

**X**  
Matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `obj`.

**Y**  
Class labels, with the same data type as exists in `obj`. The number of elements of `Y` must equal the number of rows of `X`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## 'lossfun'

Function handle or string representing a loss function. Built-in loss functions:

- 'binodeviance' — See “Loss Functions” on page 21-1792.
- 'classiferror' — Fraction of misclassified observations. See “Loss Functions” on page 21-1792.
- 'exponential' — See “Loss Functions” on page 21-1792.
- 'mincost' — Smallest misclassification cost as given by the `obj.Cost` matrix. See “Loss Functions” on page 21-1792.

You can write your own loss function using the syntax described in “Loss Functions” on page 21-1792.

**Default:** 'mincost'

## 'weights'

Numeric vector of length  $N$ , where  $N$  is the number of rows of  $X$ . `weights` are nonnegative. `loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class. When you supply `weights`, `loss` computes weighted classification loss.

**Default:** `ones(N,1)`

## Output Arguments

**L**

Classification error, a scalar. The meaning of the error depends on the values in `weights` and `lossfun`. See “Classification Error” on page 21-1791.

## Definitions

### Classification Error

The default classification error is the fraction of data  $X$  that `obj` misclassifies, where  $Y$  represents the true classifications.

# CompactClassificationDiscriminant.loss

---

Weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when obj misclassifies the  $i$ th row of  $X$ , divided by the sum of the weights.

## Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

- 'classiferror' — Predict the label with the largest posterior probability. The loss is then the fraction of misclassified observations.
- 'mincost' — Predict the label with the smallest expected misclassification cost, with expectation taken over the posterior probability, and cost as given by the `COST` property of the classifier (a matrix). The loss is then the true misclassification cost averaged over the observations.

To write your own loss function, create a function file in this form:

```
function loss = lossfun(C,S,W,COST)
```

- $N$  is the number of rows of  $X$ .
- $K$  is the number of classes in the classifier, represented in the `ClassNames` property.
- $C$  is an  $N$ -by- $K$  logical matrix, with one true per row for the true class. The index for each class is its position in the `ClassNames` property.



- $S$  is an  $N$ -by- $K$  numeric matrix.  $S$  is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from `predict`.
- $W$  is a numeric vector with  $N$  elements, the observation weights. If you pass  $W$ , the elements are normalized to sum to the prior probabilities in the respective classes.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, you can use  $COST = \text{ones}(K) - \text{eye}(K)$ , which means a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## Posterior Probability

The posterior probability that a point  $z$  belongs to class  $j$  is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_j$  and covariance  $\Sigma_j$  at a point  $z$  is

$$P(x | k) = \frac{1}{(2\pi |\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k | x) = \frac{P(x | k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, the sum over  $k$  of  $P(x | k)P(k)$ .

## Prior Probability

The prior probability is one of three choices:

# CompactClassificationDiscriminant.loss

---

- 'uniform' — The prior probability of class  $k$  is one over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.
- Custom — The prior probability of class  $k$  is the  $k$ th element of the prior vector. See `ClassificationDiscriminant.fit`.

After creating a classifier `obj`, you can set the prior by dot addressing:

```
obj.Prior = v;
```

where `v` is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

## Cost

The matrix of expected costs per observation is defined in “Cost” on page 15-8.

## Examples

Compute the resubstituted classification error for the Fisher iris data:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
L = loss(obj,meas,species)
```

```
L =
    0.0200
```

## See Also

`ClassificationDiscriminant` | `edge` | `margin` | `predict`

## How To

- “Discriminant Analysis” on page 15-3

# CompactClassificationEnsemble.loss

---

## Purpose

Classification error

## Syntax

```
L = loss(ens,X,Y)
L = loss(ens,X,Y,Name,Value)
```

## Description

`L = loss(ens,X,Y)` returns the classification error for ensemble `ens` computed using matrix of predictors `X` and true class labels `Y`.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `ens`.

`L = loss(ens,X,Y,Name,Value)` computes classification error with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

Classification ensemble created with `fitensemble`, or a compact classification ensemble created with `compact`.

### **X**

Matrix of data to classify. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `ens`. `X` should have the same number of rows as the number of elements in `Y`.

### **Y**

Classification of `X`. `Y` should be of the same type as the classification used to train `ens`, and its number of elements should equal the number of rows of `X`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can

# CompactClassificationEnsemble.loss

---

specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

## 'learners'

Indices of weak learners in the ensemble ranging from 1 to ens.NTrained. loss uses only these learners for calculating loss.

**Default:** 1:NTrained

## 'lossfun '

Function handle or string representing a loss function. Built-in loss functions:

- 'binodeviance' — See “Loss Functions” on page 21-1797
- 'classiferror' — Fraction of misclassified data
- 'exponential' — See “Loss Functions” on page 21-1797

You can write your own loss function in the syntax described in “Loss Functions” on page 21-1797.

**Default:** 'classiferror'

## 'mode'

String representing the meaning of the output L:

- 'ensemble' — L is a scalar value, the loss for the entire ensemble.
- 'individual' — L is a vector with one element per trained learner.
- 'cumulative' — L is a vector in which element J is obtained by using learners 1:J from the input list of learners.

**Default:** 'ensemble'

## 'UseObsForLearner'

A logical matrix of size N-by-T, where:

- N is the number of rows of X.
- T is the number of weak learners in ens.

When `UseObsForLearner(i, j)` is true, learner j is used in predicting the class of row i of X.

**Default:** `true(N,T)`

## 'weights'

Vector of observation weights, with nonnegative entries. The length of `weights` must equal the number of rows in X. When you specify weights, `loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class.

**Default:** `ones(size(X,1),1)`

## Output Arguments

### L

Loss, by default the fraction of misclassified data. L can be a vector, and can mean different things, depending on the name-value pair settings.

## Definitions

### Classification Error

The default classification error is the fraction of the data X that ens misclassifies, where Y are the true classifications.

Weighted classification error is the sum of weight *i* times the Boolean value that is 1 when `tree` misclassifies the *i*th row of X, divided by the sum of the weights.

### Loss Functions

The built-in loss functions are:

# CompactClassificationEnsemble.loss

---

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'classiferror' — Fraction of misclassified data, weighted by  $w$ .
- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

To write your own loss function, create a function file of the form

```
function loss = lossfun(C,S,W,COST)
```

- $N$  is the number of rows of `ens.X`.
- $K$  is the number of classes in `ens`, represented in `ens.ClassNames`.
- $C$  is an  $N$ -by- $K$  logical matrix, with one true per row for the true class. The index for each class is its position in `tree.ClassNames`.
- $S$  is an  $N$ -by- $K$  numeric matrix.  $S$  is a matrix of posterior probabilities for classes with one row per observation, similar to the `score` output from `predict`.
- $W$  is a numeric vector with  $N$  elements, the observation weights.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. The default 'classiferror' gives a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## Examples

Create a compact classification ensemble for the ionosphere data, and find the fraction of training data that the ensemble misclassifies:

```
load ionosphere
ada = fitensemble(X,Y,'AdaBoostM1',100,'tree');
adb = compact(ada);
L = loss(adb,X,Y)
```

```
L =
    0.0085
```

## See Also

[loss](#) | [edge](#) | [margin](#) | [predict](#)

# CompactClassificationTree.loss

---

**Purpose** Classification error

**Syntax**

```
L = loss(tree,X,Y)
L = loss(tree,X,Y,Name,Value)
L = loss(tree,X,Y,'subtrees',subtreevector)
[L,se] = loss(tree,X,Y,'subtrees',subtreevector)
[L,se,NLeaf] = loss(tree,X,Y,'subtrees',subtreevector)
[L,se,NLeaf,bestlevel] = loss(tree,X,Y,'subtrees',
    subtreevector)
[L,...] =
loss(tree,X,Y,'subtrees',subtreevector,Name,Value)
```

**Description** `L = loss(tree,X,Y)` returns a scalar representing how well tree classifies the data in `X`, when `Y` contains the true classifications.

When computing the loss, `loss` normalizes the class probabilities in `Y` to the class probabilities used for training, stored in the `Prior` property of `tree`.

`L = loss(tree,X,Y,Name,Value)` returns the loss with additional options specified by one or more `Name,Value` pair arguments.

`L = loss(tree,X,Y,'subtrees',subtreevector)` returns a vector of classification errors for the trees in the pruning sequence `subtreevector`.

`[L,se] = loss(tree,X,Y,'subtrees',subtreevector)` returns the vector of standard errors of the classification errors.

---

**Note** `loss` returns `se` and further outputs only when the `lossfun` name-value pair is the default `'classiferror'`.

---

`[L,se,NLeaf] = loss(tree,X,Y,'subtrees',subtreevector)` returns the vector of numbers of leaf nodes in the trees of the pruning sequence.



`[L,se,NLeaf,bestlevel] = loss(tree,X,Y,'subtrees', subtreevector)` returns the best pruning level as defined in the `treesize` name-value pair. By default, `bestlevel` is the pruning level that gives loss within one standard deviation of minimal loss.

`[L,...] = loss(tree,X,Y,'subtrees',subtreevector,Name,Value)` returns loss statistics with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **tree**

A classification tree or compact classification tree constructed by `ClassificationTree.fit` or `compact`.

### **X**

Matrix of data to classify. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `tree`. `X` should have the same number of rows as the number of elements in `Y`.

### **Y**

Classification of `X`. `Y` should be of the same type as the classification used to train `tree`, and its number of elements should equal the number of rows of `X`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'lossfun'**

Function handle or string representing a loss function. Built-in loss functions:

# CompactClassificationTree.loss

---

- 'binodeviance' — See “Loss Functions” on page 21-1804
- 'classiferror' — Fraction of misclassified observations. See “Loss Functions” on page 21-1804.
- 'exponential' — See “Loss Functions” on page 21-1804
- 'mincost' — Smallest misclassification cost as given by the `tree.Cost` matrix. See “Loss Functions” on page 21-1804.

You can write your own loss function in the syntax described in “Loss Functions” on page 21-1804.

**Default:** 'mincost'

## 'weights'

A numeric vector of length  $N$ , where  $N$  is the number of rows of  $X$ . `weights` are nonnegative. `loss` normalizes the weights so that observation weights in each class sum to the prior probability of that class. When you supply `weights`, `loss` computes weighted classification loss.

**Default:** `ones(N,1)`

Name, Value arguments associated with pruning subtrees:

## 'subtrees'

A vector with integer values from 0 (full unpruned tree) to the maximal pruning level `max(tree.PruneList)`. You can set subtrees to 'all', meaning the entire pruning sequence.

**Default:** 0

## 'treesize'

One of the following strings:

- 'se' — loss returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in subtrees).
- 'min' — loss returns the element of subtrees with smallest loss, usually the smallest element of subtrees.

## Output Arguments

**L**

Classification error, a vector the length of subtrees. The meaning of the error depends on the values in `weights` and `lossfun`; see “Classification Error” on page 21-1803.

**se**

Standard error of loss, a vector the length of subtrees.

**NLeaf**

Number of leaves (terminal nodes) in the pruned subtrees, a vector the length of subtrees.

**bestlevel**

A scalar whose value depends on `treesize`:

- `treesize = 'se'` — loss returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in subtrees).
- `treesize = 'min'` — loss returns the element of subtrees with smallest loss, usually the smallest element of subtrees.

## Definitions

### Classification Error

The default classification error is the fraction of data  $X$  that `tree` misclassifies, where  $Y$  represents the true classifications.

Weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when `tree` misclassifies the  $i$ th row of  $X$ , divided by the sum of the weights.

# CompactClassificationTree.loss

---

## Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

- 'classiferror' — Predict the label with the largest posterior probability. The loss is then the fraction of misclassified observations.
- 'mincost' — Predict the label with the smallest expected misclassification cost, with expectation taken over the posterior probability, and cost as given by the `COST` property of the classifier (a matrix). The loss is then the true misclassification cost averaged over the observations.

To write your own loss function, create a function file in this form:

```
function loss = lossfun(C,S,W,COST)
```

- `N` is the number of rows of `X`.
- `K` is the number of classes in the classifier, represented in the `ClassNames` property.
- `C` is an `N`-by-`K` logical matrix, with one `true` per row for the true class. The index for each class is its position in the `ClassNames` property.
- `S` is an `N`-by-`K` numeric matrix. `S` is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from `predict`.

- $W$  is a numeric vector with  $N$  elements, the observation weights. If you pass  $W$ , the elements are normalized to sum to the prior probabilities in the respective classes.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, you can use  $COST = \text{ones}(K) - \text{eye}(K)$ , which means a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## True Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you create the classifier using the `ClassificationTree.fit` method.  $Cost(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $Cost(i, j)=1$  if  $i \neq j$ , and  $Cost(i, j)=0$  if  $i=j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier. Suppose you have  $K$  classes. You place the observations into a matrix `Xnew` with one observation per row.

The expected cost matrix `CE` has size `Nobs`-by- $K$ . Each row of `CE` contains the expected (average) cost of classifying the observation into each of the  $K$  classes.  $CE(n, k)$  is

# CompactClassificationTree.loss

---

$$\sum_{i=1}^K \hat{P}(i | X_{new(n)}) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new(n)})$  is the posterior probability of class  $i$  for observation  $X_{new(n)}$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## Score (tree)

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

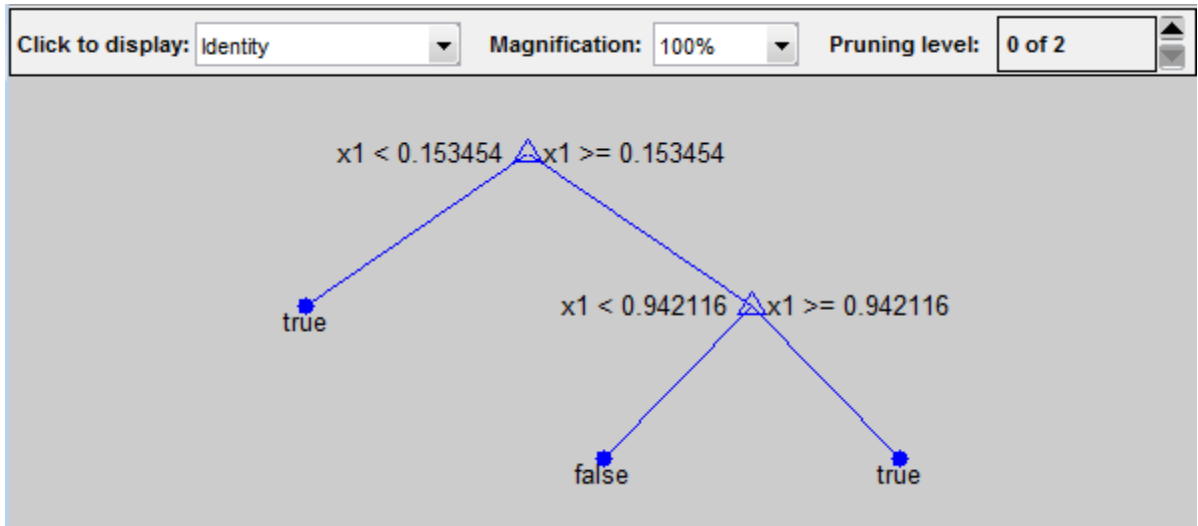
For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

1

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```

# CompactClassificationTree.loss

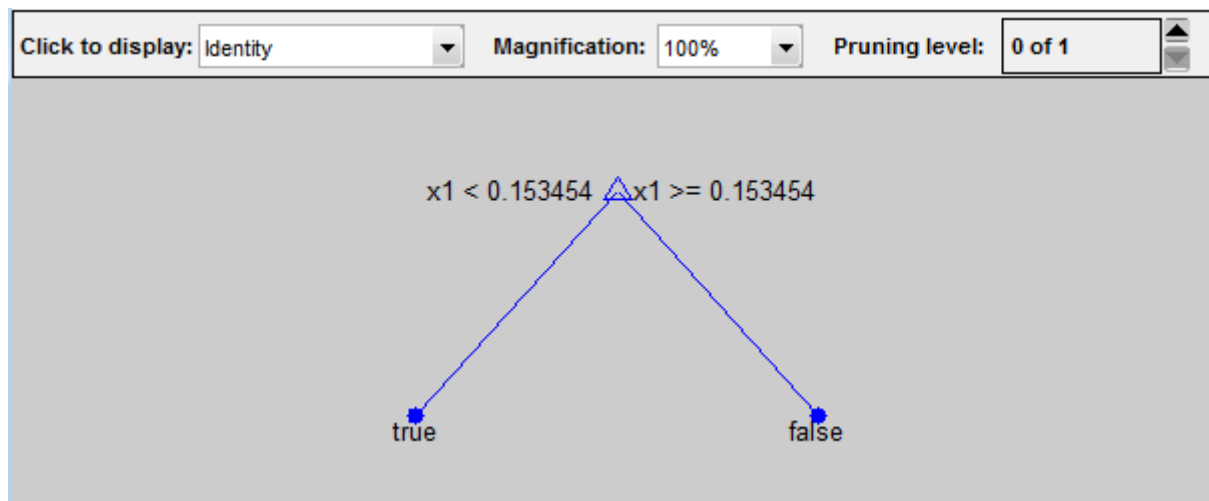


2

Prune the tree:

```
tree1 = prune(tree, 'level', 1);  
view(tree1, 'mode', 'graph')
```

# CompactClassificationTree.loss



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785
```



0.9059	0.0941	0.5469
0.9059	0.0941	0.9575
0.9059	0.0941	0.9649

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## Examples

Compute the resubstituted classification error for the ionosphere data:

```
load ionosphere
tree = ClassificationTree.fit(X,Y);
L = loss(tree,X,Y)
```

```
L =
    0.0114
```

## See Also

[margin](#) | [edge](#) | [predict](#)

# CompactRegressionEnsemble.loss

---

**Purpose** Regression error

**Syntax**  
`L = loss(ens,X,Y)`  
`L = loss(ens,X,Y,Name,Value)`

**Description** `L = loss(ens,X,Y)` returns the mean squared error between the predictions of `ens` to the data in `X`, compared to the true responses `Y`.  
`L = loss(ens,X,Y,Name,Value)` computes the error in prediction with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A regression ensemble created with `fitensemble`, or the compact method.

### **X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

NaN values in `X` are taken to be missing values. Observations with all missing values for `X` are not used in the calculation of loss.

### **Y**

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

NaN values in `Y` are taken to be missing values. Observations with missing values for `Y` are not used in the calculation of loss.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## 'learners'

Indices of weak learners in the ensemble ranging from 1 to `ens.NTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** `1:NTrained`

## 'lossfun'

Function handle for loss function, or the string `'mse'`, meaning mean squared error. If you pass a function handle `fun`, `loss` calls it as

```
fun(Y,Yfit,W)
```

where `Y`, `Yfit`, and `W` are numeric vectors of the same length.

- `Y` is the observed response.
- `Yfit` is the predicted response.
- `W` is the observation weights.

The returned value `fun(Y,Yfit,W)` should be a scalar.

**Default:** `'mse'`

## 'mode'

String representing the meaning of the output `L`:

- `'ensemble'` — `L` is a scalar value, the loss for the entire ensemble.
- `'individual'` — `L` is a vector with one element per trained learner.
- `'cumulative'` — `L` is a vector in which element `J` is obtained by using learners `1:J` from the input list of learners.

# CompactRegressionEnsemble.loss

---

**Default:** 'ensemble'

## 'UseObsForLearner'

A logical matrix of size N-by-NTrained, where N is the number of observations in `ens.X`, and NTrained is the number of weak learners. When `UseObsForLearner(I,J)` is true, `predict` uses learner J in predicting observation I.

**Default:** `true(N,NTrained)`

## 'weights'

Numeric vector of observation weights with the same number of elements as Y. The formula for loss with weights is in “Weighted Mean Squared Error” on page 21-1812.

**Default:** `ones(size(Y))`

## Output Arguments

**L**

Weighted mean squared error of predictions. The formula for loss is in “Weighted Mean Squared Error” on page 21-1812.

## Definitions

### Weighted Mean Squared Error

Let  $n$  be the number of rows of data,  $x_j$  be the  $j$ th row of data,  $y_j$  be the true response to  $x_j$ , and let  $f(x_j)$  be the response prediction of `ens` to  $x_j$ . Let  $w$  be the vector of weights (all one by default).

First the weights are divided by their sum so they add to one:  $w \rightarrow w/\sum w$ . The mean squared error  $L$  is

$$L = \sum_{j=1}^n w_j (f(x_j) - y_j)^2.$$

## Examples

Find the loss of an ensemble predictor of the `carsmall` data to find MPG as a function of engine displacement, horsepower, and vehicle weight:

```
load carsmall
X = [Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'LSBoost',100,'Tree');
L = loss(ens,X,MPG)
```

```
L =
    4.3904
```

## See Also

[predict](#) | [fitensemble](#)

# CompactRegressionTree.loss

---

**Purpose** Regression error

**Syntax**

```
L = loss(tree,X,Y)
[L,se] = loss(tree,X,Y)
[L,se,NLeaf] = loss(tree,X,Y)
[L,se,NLeaf,bestlevel] = loss(tree,X,Y)
L = loss(tree,X,Y,Name,Value)
```

**Description**

`L = loss(tree,X,Y)` returns the mean squared error between the predictions of `tree` to the data in `X`, compared to the true responses `Y`.

`[L,se] = loss(tree,X,Y)` returns the standard error of the loss.

`[L,se,NLeaf] = loss(tree,X,Y)` returns the number of leaves (terminal nodes) in the tree.

`[L,se,NLeaf,bestlevel] = loss(tree,X,Y)` returns the optimal pruning level for `tree`.

`L = loss(tree,X,Y,Name,Value)` computes the error in prediction with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **tree**

Regression tree created with `RegressionTree.fit`, or the `compact` method.

### **X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

### **Y**

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding

value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

## **'lossfun'**

Function handle for loss, or the string 'mse' representing mean-squared error. If you pass a function handle fun, loss calls fun as:

```
fun(Y,Yfit,W)
```

- Y is the vector of true responses.
- Yfit is the vector of predicted responses.
- W is the observation weights. If you pass W, the elements are normalized to sum to 1.

All the vectors have the same number of rows as Y.

**Default:** 'mse'

## **'subtrees'**

A vector with integer values from 0 (full unpruned tree) to the maximal pruning level `max(tree.PruneList)`. You can set subtrees to 'all', meaning the entire pruning sequence.

**Default:** 0

## **'treesize'**

A string, either:

- 'se' — loss returns `bestlevel` that corresponds to the smallest tree whose mean squared error (MSE) is within one standard error of the minimum MSE.

# CompactRegressionTree.loss

---

- 'min' — loss returns `bestlevel` that corresponds to the minimal MSE tree.

## 'weights'

Numeric vector of observation weights with the same number of elements as `Y`.

**Default:** `ones(size(Y))`

## Output Arguments

### **L**

Classification error, a vector the length of `subtrees`. The error for each tree is the mean squared error, weighted with `weights`. If you include `lossfun`, `L` reflects the loss calculated with `lossfun`.

### **se**

Standard error of loss, a vector the length of `subtrees`.

### **Nleaf**

Number of leaves (terminal nodes) in the pruned subtrees, a vector the length of `subtrees`.

### **bestlevel**

A scalar whose value depends on `treesize`:

- `treesize = 'se'` — loss returns the highest pruning level with loss within one standard deviation of the minimum (`L+se`, where `L` and `se` relate to the smallest value in `subtrees`).
- `treesize = 'min'` — loss returns the element of `subtrees` with smallest loss, usually the smallest element of `subtrees`.

## Definitions

### **Mean Squared Error**

The mean squared error  $m$  of the predictions  $f(X_n)$  with weight vector  $w$  is



$$m = \frac{\sum w_n (f(X_n) - Y_n)^2}{\sum w_n}.$$

## Examples

Find the loss of a regression tree predictor of the carsmall data to find MPG as a function of engine displacement, horsepower, and vehicle weight:

```
load carsmall
X = [Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
L = loss(tree,X,MPG)
```

```
L =
    4.8952
```

---

Find the pruning level that gives the optimal level of loss for the carsmall data:

```
load carsmall
X = [Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
[L,se,NLeaf,bestlevel] = loss(tree,X,MPG,'Subtrees','all');
bestlevel
```

```
bestlevel =
    4
```

## See Also

predict

# paretotails.lowerparams

---

**Purpose** Lower Pareto tails parameters

**Syntax** `params = lowerparams(obj)`

**Description** `params = lowerparams(obj)` returns the 2-element vector `params` of shape and scale parameters, respectively, of the lower tail of the Pareto tails object `obj`. `lowerparams` does not return a location parameter.

**Examples** Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);  
obj = paretotails(t,0.1,0.9);
```

```
lowerparams(obj)  
ans =  
    -0.1901    1.1898  
upperparams(obj)  
ans =  
    0.3646    0.5103
```

**See Also** `paretotails` | `upperparams`

**Purpose** Less than relation for handles

**Syntax** `h1 < h2`

**Description** `h1 < h2` performs element-wise comparisons between handle arrays `h1` and `h2`. `h1` and `h2` must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise `<` result.

If one of `h1` or `h2` is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.

`tf = lt(h1, h2)` stores the result in a logical array of the same dimensions.

**See Also** `grandstream | eq | ge | gt | le | ne`

# lsline

---

**Purpose** Add least-squares line to scatter plot

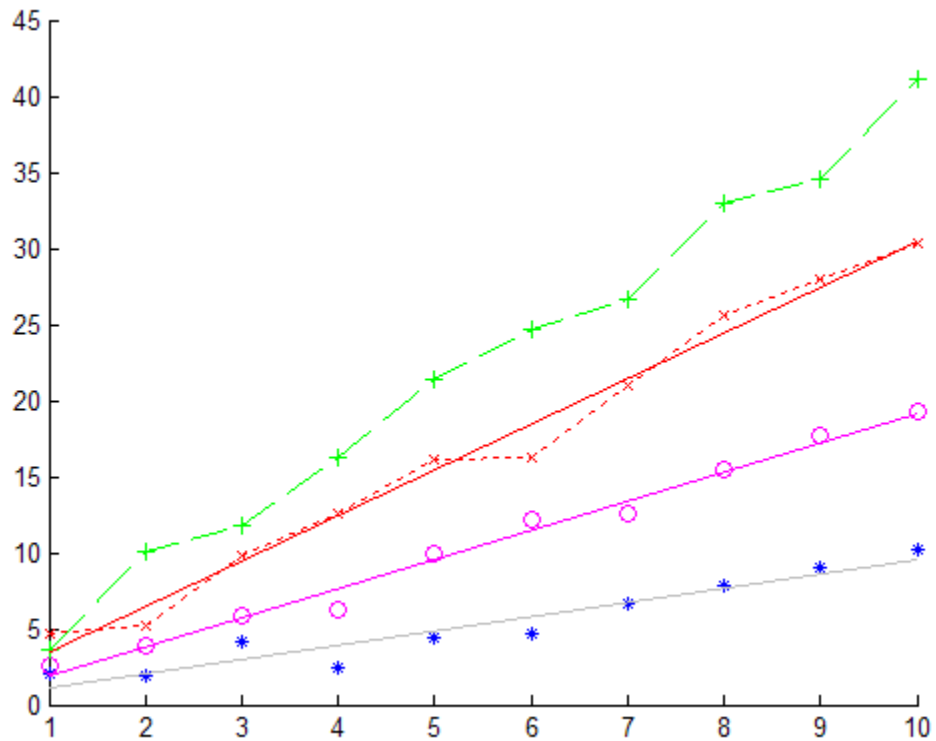
**Syntax** `lsline`  
`h = lsline`

**Description** `lsline` superimposes a least-squares line on each scatter plot in the current axes. Scatter plots are produced by the MATLAB `scatter` and `plot` functions. Data points connected with solid, dashed, or dash-dot lines ('-', '--', or '-.') are not considered to be scatter plots by `lsline`, and are ignored.

`h = lsline` returns a column vector of handles `h` to the least-squares lines.

**Examples** Use `lsline` together with scatter plots produced by `scatter` and various line styles of `plot`:

```
x = 1:10;  
  
y1 = x + randn(1,10);  
scatter(x,y1,25,'b','*')  
hold on  
  
y2 = 2*x + randn(1,10);  
plot(x,y2,'mo')  
  
y3 = 3*x + randn(1,10);  
plot(x,y3,'rx:')  
  
y4 = 4*x + randn(1,10);  
plot(x,y4,'g+--')  
  
lsline
```

**See Also**

scatter | plot | refline | refcurve | gline

# mad

---

**Purpose** Mean or median absolute deviation

**Syntax**

```
y = mad(X)
Y = mad(X,1)
Y = mad(X,0)
```

**Description** `y = mad(X)` returns the mean absolute deviation of the values in `X`. For vector input, `y` is `mean(abs(X-mean(X)))`. For a matrix input, `y` is a row vector containing the mean absolute deviation of each column of `X`. For  $N$ -dimensional arrays, `mad` operates along the first nonsingleton dimension of `X`.

`Y = mad(X,1)` returns the median absolute deviation of the values in `X`. For vector input, `y` is `median(abs(X-median(X)))`. For a matrix input, `y` is a row vector containing the median absolute deviation of each column of `X`. For  $N$ -dimensional arrays, `mad` operates along the first nonsingleton dimension of `X`.

`Y = mad(X,0)` is the same as `mad(X)`, and returns the mean absolute deviation of the values in `X`.

`mad(X,flag,dim)` computes absolute deviations along the dimension `dim` of `X`. `flag` is 0 or 1 to indicate mean or median absolute deviation, respectively.

`mad` treats NaNs as missing values and removes them.

For normally distributed data, multiply `mad` by one of the following factors to obtain an estimate of the normal scale parameter  $\sigma$ :

- `sigma = 1.253*mad(X,0)` — For mean absolute deviation
- `sigma = 1.4826*mad(X,1)` — For median absolute deviation

**Examples** The following compares the robustness of different scale estimates for normally distributed data in the presence of outliers:

```
x = normrnd(0,1,1,50);
xo = [x 10]; % Add outlier
```

```
r1 = std(xo)/std(x)
r1 =
    1.7385
```

```
r2 = mad(xo,0)/mad(x,0)
r2 =
    1.2306
```

```
r3 = mad(xo,1)/mad(x,1)
r3 =
    1.0602
```

## References

- [1] Mosteller, F., and J. Tukey. *Data Analysis and Regression*. Upper Saddle River, NJ: Addison-Wesley, 1977.
- [2] Sachs, L. *Applied Statistics: A Handbook of Techniques*. New York: Springer-Verlag, 1984, p. 253.

## See Also

std | range | iqr

# mahal

---

**Purpose** Mahalanobis distance

**Syntax** `d = mahal(Y,X)`

**Description** `d = mahal(Y,X)` computes the Mahalanobis distance (in squared units) of each observation in `Y` from the reference sample in matrix `X`. If `Y` is  $n$ -by- $m$ , where  $n$  is the number of observations and  $m$  is the dimension of the data, `d` is  $n$ -by-1. `X` and `Y` must have the same number of columns, but can have different numbers of rows. `X` must have more rows than columns.

For observation `I`, the Mahalanobis distance is defined by  $d(I) = (Y(I,:) - \mu) * \text{inv}(\text{SIGMA}) * (Y(I,:) - \mu)'$ , where `mu` and `SIGMA` are the sample mean and covariance of the data in `X`. `mahal` performs an equivalent, but more efficient, computation.

**Examples** Generate some correlated bivariate data in `X` and compare the Mahalanobis and squared Euclidean distances of observations in `Y`:

```
X = mvnrnd([0;0],[1 .9;.9 1],100);
Y = [1 1;1 -1;-1 1;-1 -1];

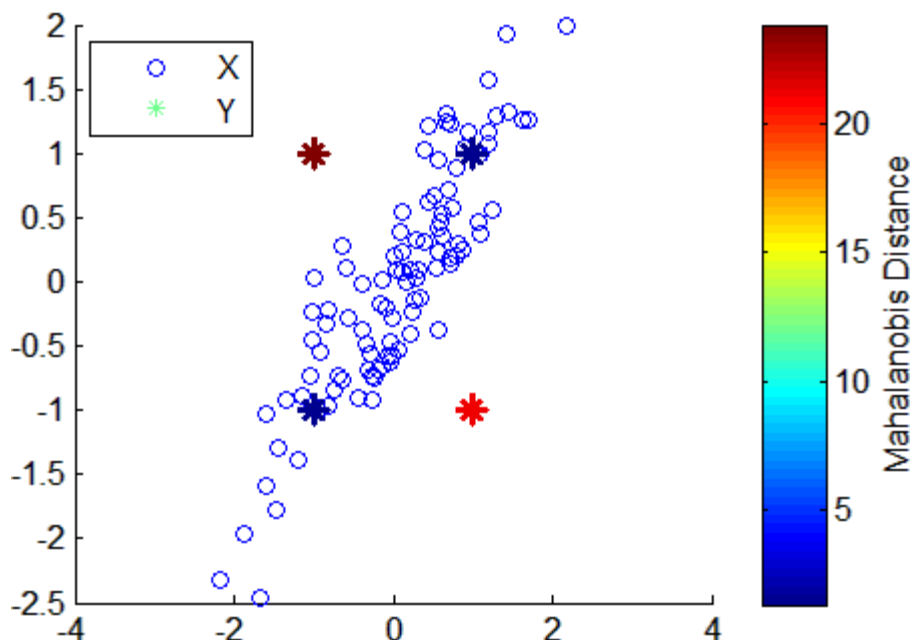
d1 = mahal(Y,X) % Mahalanobis
d1 =
    1.3592
   21.1013
   23.8086
    1.4727

d2 = sum((Y-repmat(mean(X),4,1)).^2, 2) % Squared Euclidean
d2 =
    1.9310
    1.8821
    2.1228
    2.0739

scatter(X(:,1),X(:,2))
```



```
hold on
scatter(Y(:,1),Y(:,2),100,d1,'*', 'LineWidth',2)
hb = colorbar;
ylabel(hb,'Mahalanobis Distance')
legend('X','Y','Location','NW')
```



The observations in Y with equal coordinate values are much closer to X in Mahalanobis distance than observations with opposite coordinate values, even though all observations are approximately equidistant from the mean of X in Euclidean distance. The Mahalanobis distance, by considering the covariance of the data and the scales of the different variables, is useful for detecting outliers in such cases.

## See Also

`pdist` | `mahal`

# CompactClassificationDiscriminant.mahal

---

**Purpose** Mahalanobis distance to class means

**Syntax** `M = mahal(obj,X)`  
`M = mahal(obj,X,Name,Value)`

**Description** `M = mahal(obj,X)` returns the squared Mahalanobis distances from observations in `X` to the class means in `obj`.  
`M = mahal(obj,X,Name,Value)` computes the squared Mahalanobis distance with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **obj**

Discriminant analysis classifier of class `ClassificationDiscriminant` or `CompactClassificationDiscriminant`, typically constructed with `ClassificationDiscriminant.fit`.

### **X**

Numeric matrix of size `n-by-p`, where `p` is the number of predictors in `obj`, and `n` is any positive integer.  `mahal`  computes the Mahalanobis distances from the rows of `X` to each of the `K` means of the classes in `obj`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'ClassLabels'**

Class labels consisting of `n` elements of `obj.Y`, where `n` is the number of rows of `X`.

## Output Arguments

**M**

Size and meaning of output **M** depends on whether the **ClassLabels** name-value pair is present:

- No **ClassLabels** — **M** is a numeric matrix of size  $n$ -by- $K$ , where  $K$  is the number of classes in **obj**, and  $n$  is the number of rows in **X**.  $M(i, j)$  is the squared Mahalanobis distance from the  $i$ th row of **X** to the mean of class  $j$ .
- **ClassLabels** exists — **M** is a column vector with  $n$  elements.  $M(i)$  is the squared Mahalanobis distance from the  $i$ th row of **X** to the mean for the class of the  $i$ th element of **ClassLabels**.

## Definitions

### Mahalanobis Distance

The Mahalanobis distance  $d(x,y)$  between  $n$ -dimensional points  $x$  and  $y$ , with respect to a given  $n$ -by- $n$  covariance matrix  $S$ , is

$$d(x,y) = \sqrt{(x-y)^T S^{-1} (x-y)}.$$

## Examples

Find the Mahalanobis distances from the mean of the Fisher iris data to the class means, using distinct covariance matrices for each class:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species,...
    'DiscrimType','quadratic');
mahadist = mahal(obj,mean(meas))

mahadist =
    220.0667    5.0254    30.5804
```

## See Also

CompactClassificationDiscriminant | mahal | gmdistribution

## How To

- “Discriminant Analysis” on page 15-3

# gmdistribution.mahal

---

**Purpose** Mahalanobis distance to component means

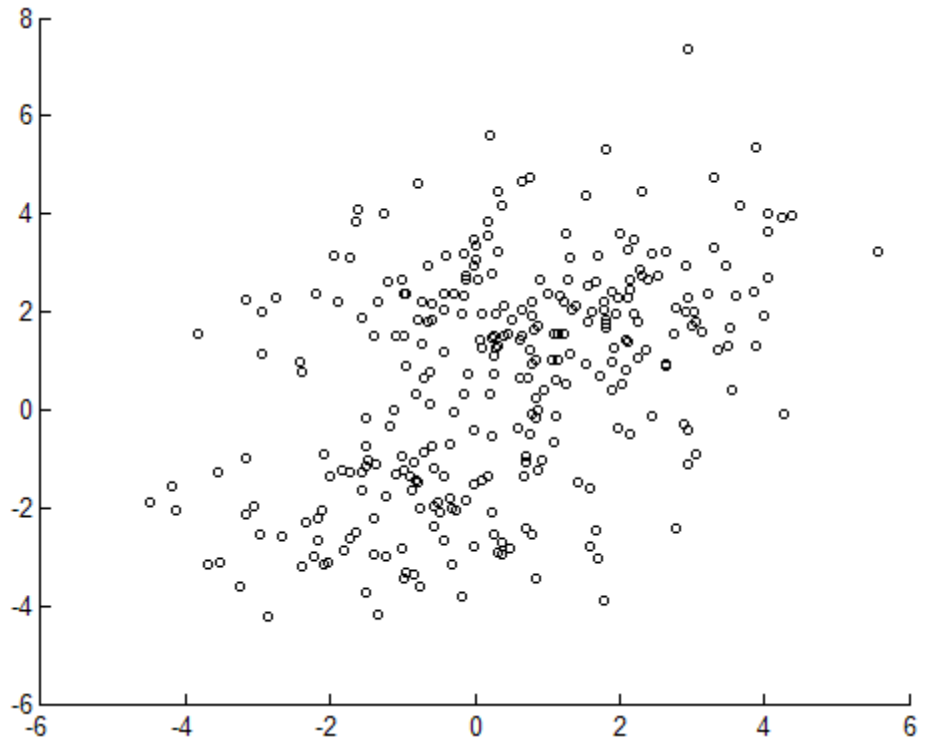
**Syntax** `D = mahal(obj,X)`

**Description** `D = mahal(obj,X)` computes the Mahalanobis distance (in squared units) of each observation in `X` to the mean of each of the  $k$  components of the Gaussian mixture distribution defined by `obj`. `obj` is an object created by `gmdistribution` or `fit`. `X` is an  $n$ -by- $d$  matrix, where  $n$  is the number of observations and  $d$  is the dimension of the data. `D` is  $n$ -by- $k$ , with `D(I,J)` the distance of observation `I` from the mean of component `J`.

**Examples** Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function:

```
MU1 = [1 2];  
SIGMA1 = [2 0; 0 .5];  
MU2 = [-3 -5];  
SIGMA2 = [1 0; 0 1];  
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];
```

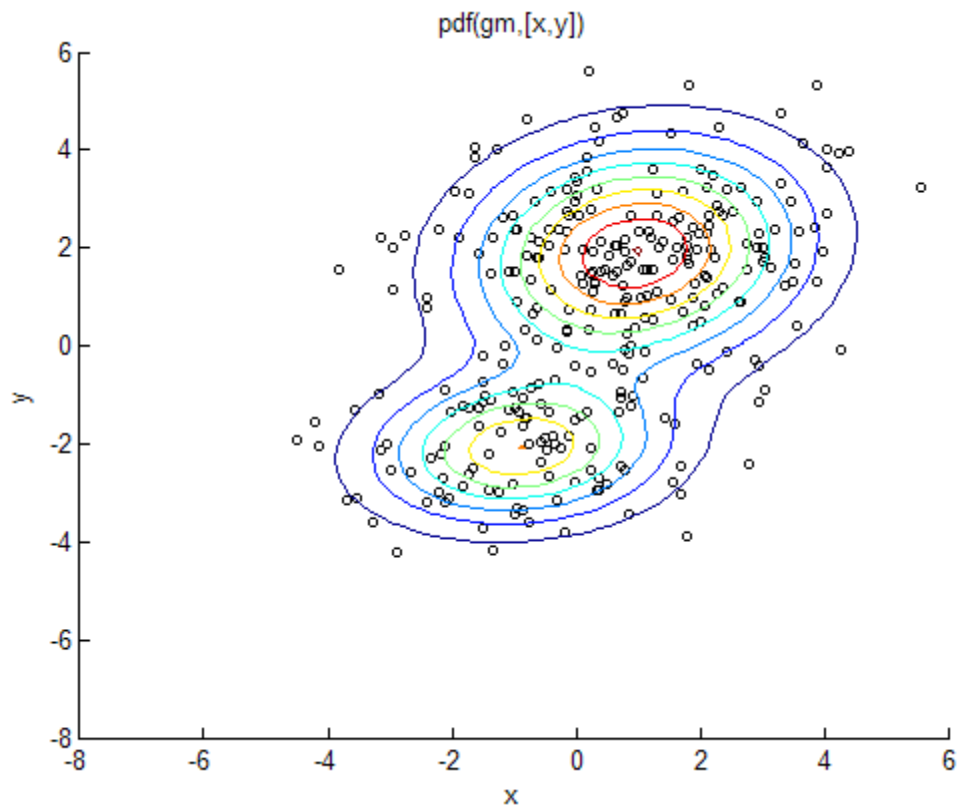
```
scatter(X(:,1),X(:,2),10,'.')  
hold on
```



Fit a two-component Gaussian mixture model:

```
obj = gmdistribution.fit(X,2);  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```

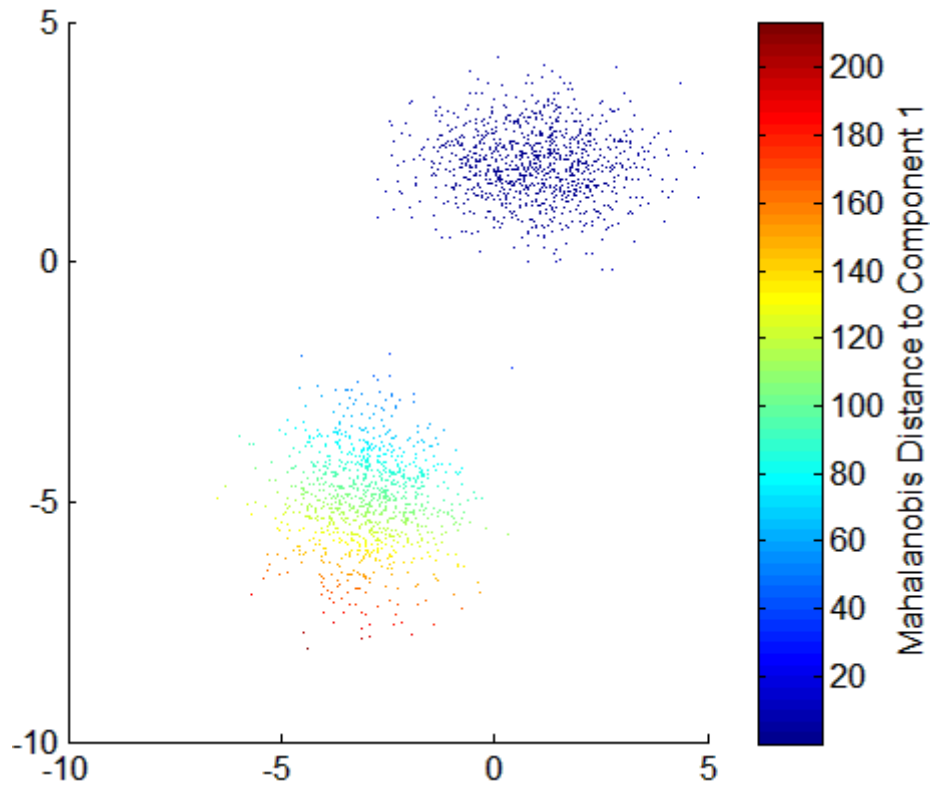
# gmdistribution.mahal



Compute the Mahalanobis distance of each point in  $X$  to the mean of each component of  $obj$ :

```
D = mahal(obj,X);
```

```
delete(h)  
scatter(X(:,1),X(:,2),10,D(:,1),'.')  
hb = colorbar;  
ylabel(hb,'Mahalanobis Distance to Component 1')
```



## See Also

[gmdistribution](#) | [cluster](#) | [posterior](#) | [mahal](#)

# maineffectsplot

---

**Purpose** Main effects plot for grouped data

**Syntax** `maineffectsplot(Y, GROUP)`  
`maineffectsplot(Y, GROUP, param1, val1, param2, val2, ...)`  
`[figh, AXESH] = maineffectsplot(...)`

**Description** `maineffectsplot(Y, GROUP)` displays main effects plots for the group means of matrix `Y` with groups defined by entries in the cell array `GROUP`. `Y` is a numeric matrix or vector. If `Y` is a matrix, the rows represent different observations and the columns represent replications of each observation. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or single-column cell array of strings. `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of rows as `Y`. The number of grouping variables must be greater than 1.

The display has one subplot per grouping variable, with each subplot showing the group means of `Y` as a function of one grouping variable.

`maineffectsplot(Y, GROUP, param1, val1, param2, val2, ...)` specifies one or more of the following name/value pairs:

- `'varnames'` — Grouping variable names in a character matrix or a cell array of strings, one per grouping variable. Default names are `'X1'`, `'X2'`, ... .
- `'statistic'` — String values that indicate whether the group mean or the group standard deviation should be plotted. Use `'mean'` or `'std'`. The default is `'mean'`. If the value is `'std'`, `Y` is required to have multiple columns.
- `'parent'` — A handle to the figure window for the plots. The default is the current figure window.

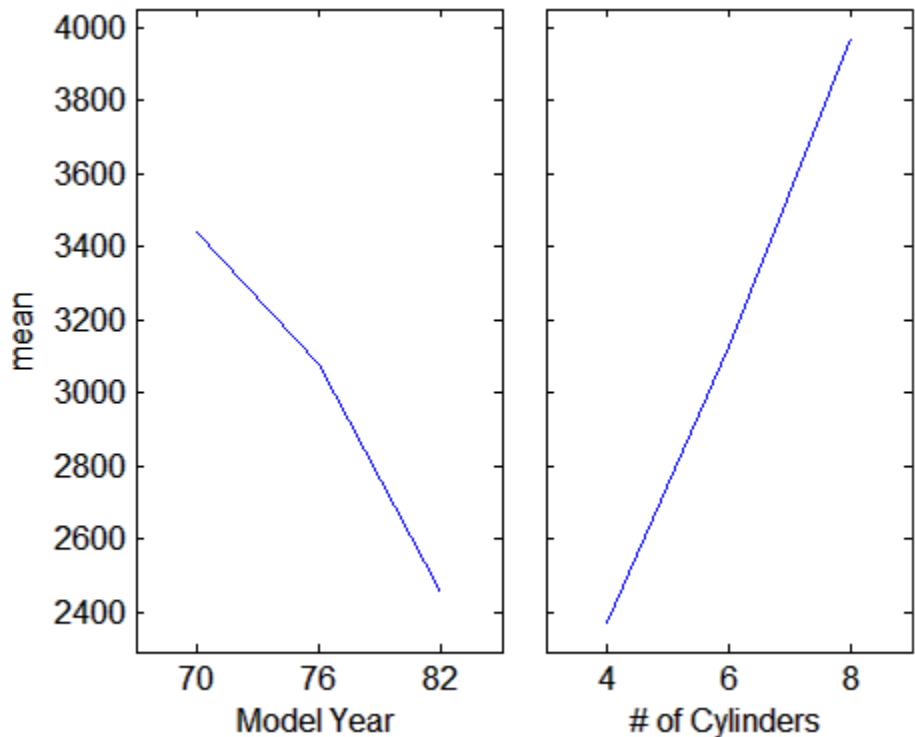
`[figh, AXESH] = maineffectsplot(...)` returns the handle `figh` to the figure window and an array of handles `AXESH` to the subplot axes.



## Examples

Display main effects plots for car weight with two grouping variables, model year and number of cylinders:

```
load carsmall;  
maineffectsplot(Weight,{Model_Year,Cylinders}, ...  
                'varnames',{'Model Year','# of Cylinders'})
```



## See Also

[interactionplot](#) | [multivarichart](#)

## How To

- “Grouping Variables” on page 2-51

# ClassificationDiscriminant.make

---

**Purpose** Construct discriminant analysis classifier from parameters

**Syntax**  
`cobj = ClassificationDiscriminant.make(Mu, Sigma)`  
`cobj = ClassificationDiscriminant.make(Mu, Sigma, Name, Value)`

**Description** `cobj = ClassificationDiscriminant.make(Mu, Sigma)` constructs a compact discriminant analysis classifier from the class means `Mu` and covariance matrix `Sigma`.

`cobj = ClassificationDiscriminant.make(Mu, Sigma, Name, Value)` constructs a compact classifier with additional options specified by one or more `Name, Value` pair arguments.

**Tips**

- You can change the discriminant type using dot addressing after constructing `cobj`:

```
cobj.DiscrimType = 'discrimType'
```

where *discrimType* is one of 'linear', 'quadratic', 'diagLinear', 'diagQuadratic', 'pseudoLinear', or 'pseudoQuadratic'. You can change between linear types or between quadratic types, but cannot change between a linear and a quadratic type.

- `cobj` is a linear classifier when `Sigma` is a matrix. `cobj` is a quadratic classifier when `Sigma` is a three-dimensional array.

## Input Arguments

### **Mu**

Matrix of class means of size  $K$ -by- $p$ , where  $K$  is the number of classes, and  $p$  is the number of predictors. Each row of `Mu` represents the mean of the multivariate normal distribution of the corresponding class. The class indices are in the `ClassNames` attribute.

### **Sigma**

Within-class covariance matrix or matrices.

- For a linear discriminant, `Sigma` is a symmetric, positive semidefinite matrix of size `p-by-p`, where `p` is the number of predictors.
- For a quadratic discriminant, `Sigma` is an array of size `p-by-p-by-K`, where `K` is the number of classes. For each `i`, `Sigma(:, :, i)` is a symmetric, positive semidefinite matrix.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### 'ClassNames'

Array of class names. Use any data type for a grouping variable: a numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings.

`ClassNames` names the classes, as ordered in `Mu`.

**Default:** `1:K`, where `K` is the number of classes (the number of rows of `Mu`)

### 'Cost'

Square matrix, where `Cost(i, j)` is the cost of classifying a point into class `j` if its true class is `i`. Alternatively, `Cost` can be a structure `S` having two fields: `S.ClassNames` containing the group names as a variable of the same type as `Y`, and `S.ClassificationCosts` containing the cost matrix.

**Default:** `Cost(i, j)=1` if `i~=j`, and `Cost(i, j)=0` if `i=j`

### 'PredictorNames'

# ClassificationDiscriminant.make

---

Cell array of names for the predictor variables, in the order in which they appear in X.

**Default:** {'x1', 'x2', ...}

## 'prior'

Prior probabilities for each class. Specify as one of:

- 'uniform', a string meaning all class prior probabilities are equal
- A vector (one scalar value for each class)
- A structure S with two fields:
  - S.ClassNames containing the class names as a variable of the same type as ClassNames
  - S.ClassProbs containing a vector of corresponding probabilities

**Default:** 'uniform'

## 'ResponseName'

Name of the response variable Y, a string.

**Default:** 'Y'

## Output Arguments

### cobj

Discriminant analysis classifier, of class CompactClassificationDiscriminant.

## Examples

Construct a compact linear discriminant analysis classifier from the means and covariances of the Fisher iris data:

```
load fisheriris
mu(1,:) = mean(meas(1:50,:));
```

```
mu(2,:) = mean(meas(51:100,:));
mu(3,:) = mean(meas(101:150,:));

mm1 = repmat(mu(1,:),50,1);
mm2 = repmat(mu(2,:),50,1);
mm3 = repmat(mu(3,:),50,1);
cc = meas;
cc(1:50,:) = cc(1:50,:) - mm1;
cc(51:100,:) = cc(51:100,:) - mm2;
cc(101:150,:) = cc(101:150,:) - mm3;
sigstar = cc' * cc / 147; % unbiased estimator of sigma
cpct = ClassificationDiscriminant.make(mu,sigstar,...
    'ClassNames',{'setosa','versicolor','virginica'})

cpct =
classreg.learning.classif.CompactClassificationDiscriminant:
    PredictorNames: {'x1' 'x2' 'x3' 'x4'}
    ResponseName: 'Y'
    ClassNames: {'setosa' 'versicolor' 'virginica'}
    ScoreTransform: 'none'
    DiscrimType: 'linear'
    Mu: [3x4 double]
    Coeffs: [3x3 struct]
```

## See Also

[ClassificationDiscriminant.fit](#) | [compact](#) | [CompactClassificationDiscriminant](#)

## How To

- “Discriminant Analysis” on page 15-3

# makedist

---

**Purpose** Create probability distribution object

**Syntax**  
`pd = makedist(distname)`  
`pd = makedist(distname,Name,Value)`

**Description** `pd = makedist(distname)` creates a probability distribution object for the distribution `distname`, using the default parameter values.

`pd = makedist(distname,Name,Value)` creates a probability distribution object with one or more distribution parameter values specified by name-value pair arguments.

## Input Arguments

### **distname - Distribution name**

string

Distribution name, specified as one of the following strings. The distribution specified by `distname` determines the class type of the returned probability distribution object.

Distribution Name	Description	Distribution Class
'Beta'	Beta distribution	prob.BetaDistribution
'Binomial'	Binomial distribution	prob.BinomialDistribution
'BirnbaumSaunders'	Birnbaum-Saunders distribution	prob.BirnbaumSaundersDistribution
'Burr'	Burr distribution	prob.BurrDistribution
'Exponential'	Exponential distribution	prob.ExponentialDistribution
'ExtremeValue'	Extreme Value distribution	prob.ExtremeValueDistribution
'Gamma'	Gamma distribution	prob.GammaDistribution
'GeneralizedExtremeValue'	Generalized Extreme Value distribution	prob.GeneralizedExtremeValueDistribution

<b>Distribution Name</b>	<b>Description</b>	<b>Distribution Class</b>
'GeneralizedPareto'	Generalized Pareto distribution	prob.GeneralizedParetoDistrib
'InverseGaussian'	Inverse Gaussian distribution	prob.InverseGaussianDistribut
'Logistic'	Logistic distribution	prob.LogisticDistribution
'Loglogistic'	Loglogistic distribution	prob.LoglogisticDistribution
'Lognormal'	Lognormal distribution	prob.LognormalDistribution
'Multinomial'	Multinomial distribution	prob.MultinomialDistribution
'Nakagami'	Nakagami distribution	prob.NakagamiDistribution
'NegativeBinomial'	Negative Binomial distribution	prob.NegativeBinomialDistribu
'Normal'	Normal distribution	prob.NormalDistribution
'PiecewiseLinear'	Piecewise Linear distribution	prob.PiecewiseLinearDistribut
'Poisson'	Poisson distribution	prob.PoissonDistribution
'Rayleigh'	Rayleigh distribution	prob.RayleighDistribution
'Rician'	Rician distribution	prob.RicianDistribution
'tLocationScale'	$t$ Location-Scale distribution	prob.tLocationScaleDistributi
'Triangular'	Triangular distribution	prob.TriangularDistribution
'Uniform'	Uniform distribution	prob.UniformDistribution
'Weibull'	Weibull distribution	prob.WeibullDistribution

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `makedist('Normal','mu',10)` specifies a normal distribution with parameter `mu` equal to 10, and parameter `sigma` equal to the default value of 1.

## Beta Distribution

### 'a' - First shape parameter

1 (default) | nonnegative scalar value

**Example:** `'a',3`

### Data Types

single | double

### 'b' - Second shape parameter

1 (default) | nonnegative scalar value

**Example:** `'b',5`

### Data Types

single | double

## Binomial Distribution

### 'N' - Number of trials

1 (default) | positive integer value

**Example:** `'N',25`

### Data Types

single | double

### 'p' - Probability of success



0.5 (default) | scalar value in the range [0,1]

**Example:** 'p',0.25

**Data Types**

single | double

**Birnbaum-Saunders Distribution**

**'beta' - Scale parameter**

1 (default) | positive scalar value

**Example:** 'beta',2

**Data Types**

single | double

**'gamma' - Shape parameter**

1 (default) | nonnegative scalar value

**Example:** 'gamma',0

**Data Types**

single | double

**Burr Distribution**

**'alpha' - Scale parameter**

1 (default) | positive scalar value

**Example:** 'alpha',2

**Data Types**

single | double

**'c' - First shape parameter**

1 (default) | positive scalar value

**Example:** 'c',2

**Data Types**

single | double

## **'k' - Second shape parameter**

1 (default) | positive scalar value

Example: 'k',5

### **Data Types**

single | double

## **Exponential Distribution**

### **'mu' - Mean parameter**

1 (default) | positive scalar value

Example: 'mu',5

### **Data Types**

single | double

## **Extreme Value Distribution**

### **'mu' - Location parameter**

0 (default) | scalar value

Example: 'mu',-2

### **Data Types**

single | double

### **'sigma' - Scale parameter**

1 (default) | nonnegative scalar value

Example: 'sigma',2

### **Data Types**

single | double

## **Gamma Distribution**

### **'a' - Shape parameter**

1 (default) | positive scalar value

**Example:** 'a',2

**Data Types**

single | double

**'b' - Scale parameter**

1 (default) | nonnegative scalar value

**Example:** 'b',0

**Data Types**

single | double

**Generalized Extreme Value Distribution**

**'k' - Shape parameter**

0 (default) | scalar value

**Example:** 'k',0

**Data Types**

single | double

**'sigma' - Scale parameter**

1 (default) | nonnegative scalar value

**Example:** 'sigma',2

**Data Types**

single | double

**'mu' - Location parameter**

0 (default) | scalar value

**Example:** 'mu',1

**Data Types**

single | double

**Generalized Pareto Distribution**

**'k' - Shape parameter**

1 (default) | scalar value

Example: 'k',0

**Data Types**

single | double

**'sigma' - Scale parameter**

1 (default) | nonnegative scalar value

Example: 'sigma',2

**Data Types**

single | double

**'theta' - Location parameter**

1 (default) | scalar value

Example: 'theta',2

**Data Types**

single | double

**Inverse Gaussian Distribution****'mu' - Scale parameter**

1 (default) | positive scalar value

Example: 'mu',2

**Data Types**

single | double

**'lambda' - Shape parameter**

1 (default) | positive scalar value

Example: 'lambda',4

**Data Types**

single | double

### Logistic Distribution

**'mu' - Mean**

0 (default) | scalar value

Example: 'mu',2

**Data Types**

single | double

**'sigma' - Scale parameter**

1 (default) | nonnegative scalar value

Example: 'sigma',4

**Data Types**

single | double

### Loglogistic Distribution

**'mu' - Log mean**

0 (default) | scalar value

Example: 'mu',2

**Data Types**

single | double

**'sigma' - Log scale parameter**

1 (default) | nonnegative scalar value

Example: 'sigma',4

**Data Types**

single | double

### Lognormal Distribution

**'mu' - Log mean**

0 (default) | scalar value

**Example:** 'mu',2

## **Data Types**

single | double

## **'sigma' - Log standard deviation**

1 (default) | nonnegative scalar value

**Example:** 'sigma',2

## **Data Types**

single | double

## **Multinomial Distribution**

### **'probabilities' - Outcome probabilities**

[0.500 0.500] (default) | vector of scalar values in the range [0,1]

**Example:** 'probabilities',[0.1 0.2 0.5 0.2]

## **Data Types**

single | double

## **Nakagami Distribution**

### **'mu' - Shape parameter**

1 (default) | positive scalar value

**Example:** 'mu',5

## **Data Types**

single | double

### **'omega' - Scale parameter**

1 (default) | positive scalar value

**Example:** 'omega',5

## **Data Types**

single | double

## Negative Binomial Distribution

### **'R' - Number of successes**

1 (default) | positive scalar value

Example: 'R',5

### **Data Types**

single | double

### **'p' - Probability of success**

0.5 (default) | scalar value in the range (0,1]

Example: 'p',0.1

### **Data Types**

single | double

## Normal Distribution

### **'mu' - Mean**

0 (default) | scalar value

Example: 'mu',2

### **Data Types**

single | double

### **'sigma' - Standard deviation**

1 (default) | nonnegative scalar value

Example: 'sigma',2

### **Data Types**

single | double

## Piecewise Linear Distribution

### **'x' - Data values**

1 (default) | vector of scalar values

**Example:** 'x',[1 2 3]

## **Data Types**

single | double

## **'Fx' - cdf values**

1 (default) | vector of scalar values

**Example:** 'Fx',[.2 .5 1]

## **Data Types**

single | double

## **Poisson Distribution**

### **'lambda' - Mean**

1 (default) | nonnegative scalar value

**Example:** 'lambda',5

## **Data Types**

single | double

## **Rayleigh Distribution**

### **'b' - Defining parameter**

1 (default) | positive scalar value

**Example:** 'b',3

## **Data Types**

single | double

## **Rician Distribution**

### **'s' - Noncentrality parameter**

1 (default) | nonnegative scalar value

**Example:** 's',0



**Data Types**

single | double

**'sigma' - Scale parameter**

1 (default) | positive scalar value

Example: 'sigma',2

**Data Types**

single | double

***t* Location-Scale Distribution****'mu' - Location parameter**

0 (default) | scalar value

Example: 'mu',-2

**Data Types**

single | double

**'sigma' - Scale parameter**

1 (default) | positive scalar value

Example: 'sigma',2

**Data Types**

single | double

**'nu' - Degrees of freedom**

5 (default) | positive scalar value

Example: 'nu',20

**Data Types**

single | double

**Triangular Distribution****'a' - Lower limit**

0 (default) | scalar value

**Example:** 'a', -2

**Data Types**

single | double

**'b' - Peak location**

0.5 (default) | scalar value greater than or equal to a

**Example:** 'b', 1

**Data Types**

single | double

**'c' - Upper limit**

1 (default) | scalar value greater than or equal to b

**Example:** 'c', 5

**Data Types**

single | double

**Uniform Distribution**

**'lower' - Lower parameter**

0 (default) | scalar value

**Example:** 'lower', -4

**Data Types**

single | double

**'upper' - Upper parameter**

1 (default) | scalar value greater than lower

**Example:** 'upper', 2

**Data Types**

single | double

**Weibull Distribution**

**'a' - Scale parameter**

1 (default) | positive scalar value

Example: 'a',2

**Data Types**

single | double

**'b' - Shape parameter**

1 (default) | positive scalar value

Example: 'b',5

**Data Types**

single | double

**Output Arguments****pd - Probability distribution**

probability distribution object

Probability distribution, returned as a probability distribution object of the type specified by `distname`.

**Examples****Create a Normal Distribution Object**

Create a normal distribution object using the default parameter values.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Compute the interquartile range of the distribution.

```
r = iqr(pd)
```

```
r =  
1.3490
```

## Create a Gamma Distribution Object

Create a gamma distribution object using the default parameter values.

```
pd = makedist('Gamma')  
pd =  
GammaDistribution  
  
Gamma distribution  
a = 1  
b = 1
```

Compute the mean of the gamma distribution.

```
mean = mean(pd)  
mean =  
1
```

## Specify Parameters for a Normal Distribution Object

Create a normal distribution object with parameter values  $\mu = 75$  and  $\sigma = 10$ .

```
pd = makedist('Normal','mu',75,'sigma',10)  
pd =  
NormalDistribution  
  
Normal distribution  
mu = 75  
sigma = 10
```

## Specify Parameters for a Gamma Distribution Object

Create a gamma distribution object with the parameter value  $a = 3$  and the default value  $b = 1$ .

```
pd = makedist('Gamma','a',3)
```

```
pd =
```

```
GammaDistribution
```

```
Gamma distribution
```

```
  a = 3
```

```
  b = 1
```

## Alternative Functionality

### App

The Distribution Fitting app opens a graphical user interface for you to import data from the workspace and interactively fit a probability distribution to that data. You can then save the distribution to the workspace as a probability distribution object. Open the Distribution Fitting app using `dfittool`, or click Distribution Fitting on the Apps tab.

## See Also

`fitdist` | `dfittool`

# manova1

---

**Purpose** One-way multivariate analysis of variance

**Syntax**  
`d = manova1(X,group)`  
`d = manova1(X,group,alpha)`  
`[d,p] = manova1(...)`  
`[d,p,stats] = manova1(...)`

**Description** `d = manova1(X,group)` performs a one-way Multivariate Analysis of Variance (MANOVA) for comparing the multivariate means of the columns of `X`, grouped by `group`. `X` is an  $m$ -by- $n$  matrix of data values, and each row is a vector of measurements on  $n$  variables for a single observation. `group` is a grouping variable defined as a categorical variable, vector, string array, or cell array of strings. Two observations are in the same group if they have the same value in the `group` array. The observations in each group represent a sample from a population.

The function returns `d`, an estimate of the dimension of the space containing the group means. `manova1` tests the null hypothesis that the means of each group are the same  $n$ -dimensional multivariate vector, and that any difference observed in the sample `X` is due to random chance. If `d = 0`, there is no evidence to reject that hypothesis. If `d = 1`, then you can reject the null hypothesis at the 5% level, but you cannot reject the hypothesis that the multivariate means lie on the same line. Similarly, if `d = 2` the multivariate means may lie on the same plane in  $n$ -dimensional space, but not on the same line.

`d = manova1(X,group,alpha)` gives control of the significance level, `alpha`. The return value `d` will be the smallest dimension having  $p > \alpha$ , where  $p$  is a  $p$ -value for testing whether the means lie in a space of that dimension.

`[d,p] = manova1(...)` also returns a `p`, a vector of  $p$ -values for testing whether the means lie in a space of dimension 0, 1, and so on. The largest possible dimension is either the dimension of the space, or one less than the number of groups. There is one element of `p` for each dimension up to, but not including, the largest.

If the  $i$ th  $p$ -value is near zero, this casts doubt on the hypothesis that the group means lie on a space of  $i-1$  dimensions. The choice of a

critical  $p$ -value to determine whether the result is judged statistically significant is left to the researcher and is specified by the value of the input argument `alpha`. It is common to declare a result significant if the  $p$ -value is less than 0.05 or 0.01.

`[d,p,stats] = manova1(...)` also returns `stats`, a structure containing additional MANOVA results. The structure contains the following fields.

Field	Contents
W	Within-groups sum of squares and cross-products matrix
B	Between-groups sum of squares and cross-products matrix
T	Total sum of squares and cross-products matrix
dfW	Degrees of freedom for W
dfB	Degrees of freedom for B
dfT	Degrees of freedom for T
lambda	Vector of values of Wilk's lambda test statistic for testing whether the means have dimension 0, 1, etc.
chisq	Transformation of lambda to an approximate chi-square distribution
chisqdf	Degrees of freedom for chisq
eigenval	Eigenvalues of $W^{-1}B$
eigenvec	Eigenvectors of $W^{-1}B$ ; these are the coefficients for the canonical variables C, and they are scaled so the within-group variance of the canonical variables is 1
canon	Canonical variables C, equal to $XC * \text{eigenvec}$ , where XC is X with columns centered by subtracting their means

Field	Contents
mdist	A vector of Mahalanobis distances from each point to the mean of its group
gmdist	A matrix of Mahalanobis distances between each pair of group means

The canonical variables  $C$  are linear combinations of the original variables, chosen to maximize the separation between groups. Specifically,  $C(:, 1)$  is the linear combination of the  $X$  columns that has the maximum separation between groups. This means that among all possible linear combinations, it is the one with the most significant  $F$  statistic in a one-way analysis of variance.  $C(:, 2)$  has the maximum separation subject to it being orthogonal to  $C(:, 1)$ , and so on.

You may find it useful to use the outputs from `manova1` along with other functions to supplement your analysis. For example, you may want to start with a grouped scatter plot matrix of the original variables using `gplotmatrix`. You can use `gscatter` to visualize the group separation using the first two canonical variables. You can use `manovacluster` to graph a dendrogram showing the clusters among the group means.

## Assumptions

The MANOVA test makes the following assumptions about the data in  $X$ :

- The populations for each group are normally distributed.
- The variance-covariance matrix is the same for each population.
- All observations are mutually independent.

## Examples

you can use `manova1` to determine whether there are differences in the averages of four car characteristics, among groups defined by the country where the cars were made.

```
load carbig
[d,p] = manova1([MPG Acceleration Weight Displacement],...
```



```
Origin)
d =
  3
p =
  0
  0.0000
  0.0075
  0.1934
```

There are four dimensions in the input matrix, so the group means must lie in a four-dimensional space. `manova1` shows that you cannot reject the hypothesis that the means lie in a 3-D subspace.

## References

[1] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

## See Also

`anova1` | `canoncorr` | `gscatter` | `gplotmatrix` | `manovacluster`

## How To

- “Grouping Variables” on page 2-51

# manovacluster

---

**Purpose** Dendrogram of group mean clusters following MANOVA

**Syntax**  
`manovacluster(stats)`  
`manovacluster(stats,method)`  
`H = manovacluster(stats,method)`

**Description** `manovacluster(stats)` generates a dendrogram plot of the group means after a multivariate analysis of variance (MANOVA). `stats` is the output `stats` structure from `manova1`. The clusters are computed by applying the single linkage method to the matrix of Mahalanobis distances between group means.

See `dendrogram` for more information on the graphical output from this function. The dendrogram is most useful when the number of groups is large.

`manovacluster(stats,method)` uses the specified method in place of single linkage. `method` can be any of the following character strings that identify ways to create the cluster hierarchy. (See `linkage` for additional information.)

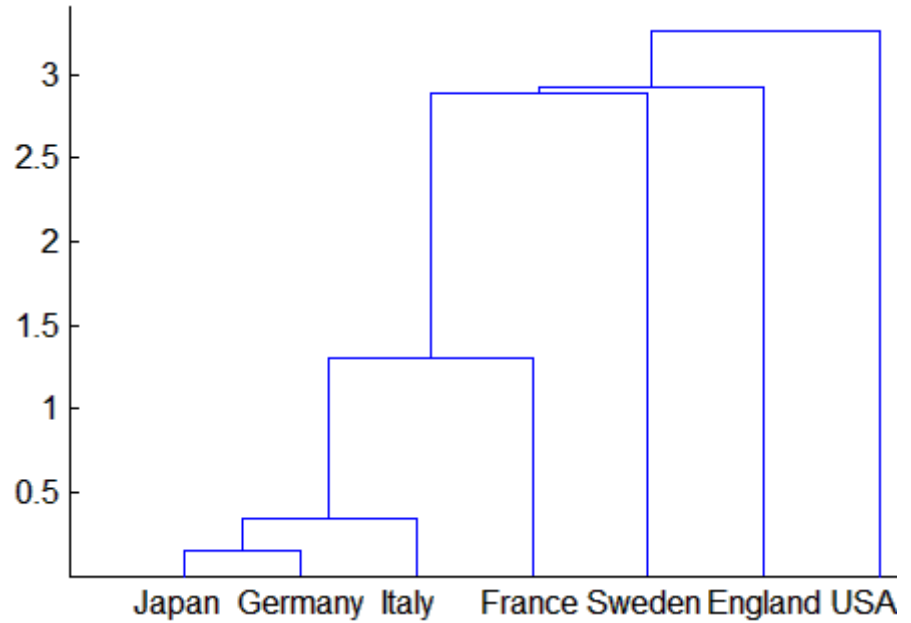
<b>Method</b>	<b>Description</b>
'single'	Shortest distance (default)
'complete'	Largest distance
'average'	Average distance
'centroid'	Centroid distance
'ward'	Incremental sum of squares

`H = manovacluster(stats,method)` returns a vector of handles to the lines in the figure.

**Examples** Let's analyze the larger car data set to determine which countries produce cars with the most similar characteristics.

```
load carbig
```

```
X = [MPG Acceleration Weight Displacement];  
[d,p,stats] = manova1(X,Origin);  
manovacluster(stats)
```



## See Also

[cluster](#) | [dendrogram](#) | [linkage](#) | [manova1](#)

# ClassificationKNN.margin

---

<b>Purpose</b>	Margin of $k$ -nearest neighbor classifier
<b>Syntax</b>	<code>m = margin mdl,X,Y</code>
<b>Description</b>	<code>m = margin mdl,X,Y</code> returns the classification margins for the matrix of predictors $X$ and class labels $Y$ . For the definition, see “Margin” on page 21-1860.
<b>Input Arguments</b>	<b>mdl</b> $k$ -nearest neighbor classifier, created by <code>ClassificationKNN.fit</code> . <b>X</b> Matrix of predictor values. Each column of $X$ represents one variable, and each row represents one observation. <b>Y</b> A numeric vector, vector of categorical variables (nominal or ordinal), logical vector, character array, or cell array of strings, with the same number of rows as $X$ . Each row of $Y$ represents the classification of the corresponding row of $X$ .
<b>Output Arguments</b>	<b>m</b> Numeric column vector of length <code>size(X,1)</code> . Each entry in $m$ represents the margin for the corresponding rows of $X$ and (true class) $Y$ , computed using <code>mdl</code> .
<b>Definitions</b>	<b>Margin</b> The classification <i>margin</i> is the difference between the classification <i>score</i> for the true class and maximal classification score for the false classes. <b>Score</b> The <i>score</i> of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more

detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 21-2608.

## Examples

### Margin Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(meas,species,'NumNeighbors',5);
```

Examine the margin of the classifier for a mean observation classified 'versicolor'.

```
X = mean(meas);  
Y = {'versicolor'};  
m = margin(mdl,X,Y)
```

```
m =
```

```
1
```

The classifier has no doubt that 'versicolor' is the correct classification (all five nearest neighbors classify as 'versicolor').

## See Also

[ClassificationKNN](#) | [edge](#) | [loss](#) |

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9

# CompactClassificationDiscriminant.margin

---

<b>Purpose</b>	Classification margins
<b>Syntax</b>	<code>m = margin(obj,X,Y)</code>
<b>Description</b>	<code>m = margin(obj,X,Y)</code> returns the classification margins for the matrix of predictors <i>X</i> and class labels <i>Y</i> . For the definition, see “Definitions” on page 21-1862.
<b>Input Arguments</b>	<b>obj</b> Discriminant analysis classifier of class <code>ClassificationDiscriminant</code> or <code>CompactClassificationDiscriminant</code> , typically constructed with <code>ClassificationDiscriminant.fit</code> . <b>X</b> Matrix where each row represents an observation, and each column represents a predictor. The number of columns in <i>X</i> must equal the number of predictors in <i>obj</i> . <b>Y</b> Class labels, with the same data type as exists in <i>obj</i> . The number of elements of <i>Y</i> must equal the number of rows of <i>X</i> .
<b>Output Arguments</b>	<b>m</b> Numeric column vector of length <code>size(X,1)</code> . Each entry in <i>m</i> represents the margin for the corresponding rows of <i>X</i> and (true class) <i>Y</i> , computed using <i>obj</i> .
<b>Definitions</b>	<b>Margin</b> The classification <i>margin</i> is the difference between the classification <i>score</i> for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix <i>X</i> . A high value of margin indicates a more reliable prediction than a low value.

## Score (discriminant analysis)

For discriminant analysis, the *score* of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 15-7.

## Examples

Compute the classification margin for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries:

```
load fisheriris
X = meas(:,1:2);
obj = ClassificationDiscriminant.fit(X,species);
M = margin(obj,X,species);
M(end-10:end)
```

```
ans =
    0.6551
    0.4838
    0.6551
   -0.5127
    0.5659
    0.4611
    0.4949
    0.1024
    0.2787
   -0.1439
   -0.4444
```

The classifier trained on all the data is better:

```
obj = ClassificationDiscriminant.fit(meas,species);
M = margin(obj,meas,species);
M(end-10:end)
```

```
ans =
    0.9983
    1.0000
```

# CompactClassificationDiscriminant.margin

---

0.9991  
0.9978  
1.0000  
1.0000  
0.9999  
0.9882  
0.9937  
1.0000  
0.9649

## See Also

`ClassificationDiscriminant` | `edge` | `loss` | `predict`

## How To

- “Discriminant Analysis” on page 15-3



# CompactClassificationEnsemble.margin

---

## Purpose

Classification margins

## Syntax

```
M = margin(ens,X,Y)
M = margin(ens,X,Y,Name,Value)
```

## Description

`M = margin(ens,X,Y)` returns the classification margin for the predictions of `ens` on data `X`, when the true classifications are `Y`.

`M = margin(ens,X,Y,Name,Value)` calculates margin with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

Classification ensemble created with `fitensemble`, or a compact classification ensemble created with `compact`.

### **X**

Matrix of data to classify. Each row of `X` represents one observation, and each column represents one predictor. `X` must have the same number of columns as the data used to train `ens`. `X` should have the same number of rows as the number of elements in `Y`.

### **Y**

Classification of `X`. `Y` should be of the same type as the classification used to train `ens`, and its number of elements should equal the number of rows of `X`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'learners'**

# CompactClassificationEnsemble.margin

---

Indices of weak learners in the ensemble ranging from 1 to `ens.NTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** `1:NTrained`

## 'UseObsForLearner'

A logical matrix of size N-by-T, where:

- N is the number of rows of X.
- T is the number of weak learners in `ens`.

When `UseObsForLearner(i, j)` is `true`, learner `j` is used in predicting the class of row `i` of X.

**Default:** `true(N,T)`

## Output Arguments

**M**

A numeric column vector with the same number of rows as X. Each row of M gives the classification margin for that row of X.

## Definitions

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix X.

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## Examples

Find the margin for classifying an average flower from the Fisheriris data as 'versicolor':

```
load fisheriris % X = meas, Y = species
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
flower = mean(meas);
predict(ens,flower)
```

```
ans =
    'versicolor'
```

```
margin(ens,mean(meas),'versicolor')
```

```
ans =
    3.2140
```

## See Also

[predict](#) | [edge](#) | [loss](#)

# CompactClassificationTree.margin

---

**Purpose** Classification margins

**Syntax** `m = margin(tree,X,Y)`

**Description** `m = margin(tree,X,Y)` returns the classification margins for the matrix of predictors `X` and class labels `Y`. For the definition, see “Margin” on page 21-1868.

## Input Arguments

### **tree**

A classification tree created by `ClassificationTree.fit`, or a compact classification tree created by `compact`.

### **X**

A matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `tree`.

### **Y**

Class labels, with the same data type as exists in `tree`.

## Output Arguments

### **m**

A numeric column vector of length `size(X,1)`. Each entry in `m` represents the margin for the corresponding rows of `X` and (true class) `Y`, computed using `tree`.

## Definitions

### **Margin**

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `X`.

### **Score (tree)**

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that

# CompactClassificationTree.margin

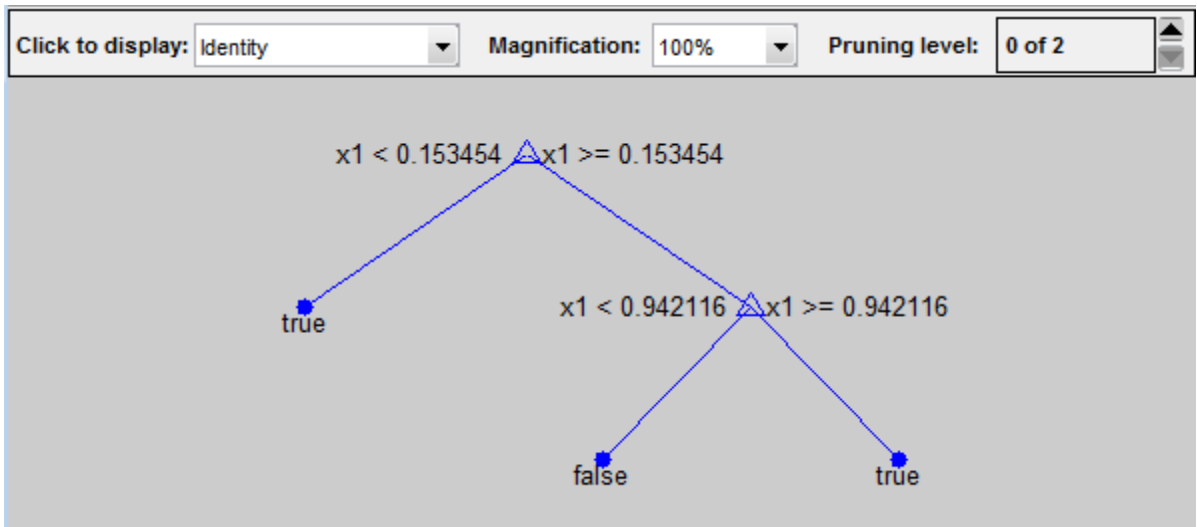
lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

1

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```

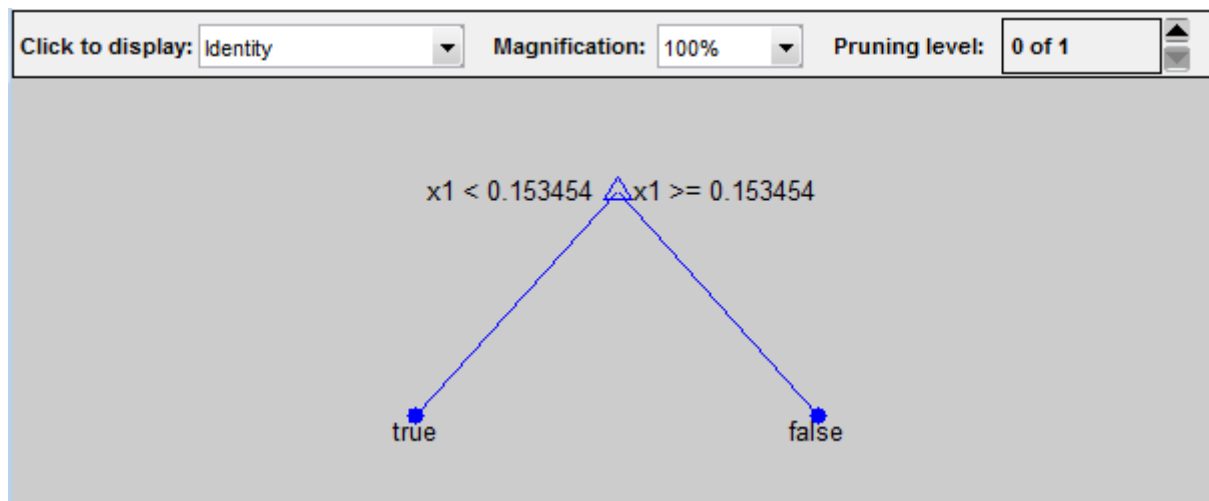


2

Prune the tree:

```
tree1 = prune(tree,'level',1);
view(tree1,'mode','graph')
```

# CompactClassificationTree.margin



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
0.9059    0.0941    0.8147  
0.9059    0.0941    0.9058  
0         1.0000    0.1270  
0.9059    0.0941    0.9134  
0.9059    0.0941    0.6324  
0         1.0000    0.0975  
0.9059    0.0941    0.2785
```

0.9059	0.0941	0.5469
0.9059	0.0941	0.9575
0.9059	0.0941	0.9649

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## Examples

Compute the classification margin for the Fisher iris data, trained on its first two columns of data, and view the last 10 entries:

```
load fisheriris
X = meas(:,1:2);
tree = ClassificationTree.fit(X,species);
M = margin(tree,X,species);
M(end-10:end)

ans =
    0.1111
    0.1111
    0.1111
   -0.2857
    0.6364
    0.6364
    0.1111
    0.7500
    1.0000
    0.6364
    0.2000
```

The classification tree trained on all the data is better:

```
tree = ClassificationTree.fit(meas,species);
M = margin(tree,meas,species);
M(end-10:end)
```

# CompactClassificationTree.margin

---

```
ans =  
  0.9565  
  0.9565  
  0.9565  
  0.9565  
  0.9565  
  0.9565  
  0.9565  
  0.9565  
  0.9565  
  0.9565  
  0.9565
```

## See Also

[predict](#) | [loss](#) | [edge](#)



## Purpose

Classification margin

## Syntax

```
mar = margin(B,X,Y)
mar = margin(B,X,Y, 'param1',val1, 'param2',val2, ...)
```

## Description

`mar = margin(B,X,Y)` computes the classification margins for predictors  $X$  given true response  $Y$ . The  $Y$  can be either a numeric vector, character matrix, cell array of strings, categorical vector or logical vector. `mar` is a numeric array of size `Nobs`-by-`NTrees`, where `Nobs` is the number of rows of  $X$  and  $Y$ , and `NTrees` is the number of trees in the ensemble  $B$ . For observation  $I$  and tree  $J$ , `mar(I,J)` is the difference between the score for the true class and the largest score for other classes. This method is available for classification ensembles only.

`mar = margin(B,X,Y, 'param1',val1, 'param2',val2, ...)` specifies optional parameter name/value pairs:

- 'mode' String indicating how the method computes errors. If set to 'cumulative' (default), `margin` computes cumulative errors and `mar` is an `Nobs`-by-`NTrees` matrix, where the first column gives error from `trees(1)`, second column gives error from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to 'individual', `mar` is a `Nobs`-by-`NTrees` matrix, where each element is an error from each tree in the ensemble. If set to 'ensemble', `mar` a single column of length `Nobs` showing the cumulative margins for the entire ensemble.
- 'trees' Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns a vector of length `NTrees` for 'cumulative' and 'individual' modes, where `NTrees` is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element

## CompactTreeBagger.margin

---

gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.

'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.

'useifort' Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

### See Also

`TreeBagger.margin`

## Purpose

Classification margin

## Syntax

```
mar = margin(B,X,Y)
mar = margin(B,X,Y, 'param1',val1, 'param2',val2, ...)
```

## Description

`mar = margin(B,X,Y)` computes the classification margins for predictors  $X$  given true response  $Y$ . The  $Y$  can be either a numeric vector, character matrix, cell array of strings, categorical vector or logical vector. `mar` is a numeric array of size `Nobs`-by-`NTrees`, where `Nobs` is the number of rows of  $X$  and  $Y$ , and `NTrees` is the number of trees in the ensemble  $B$ . For observation  $I$  and tree  $J$ , `mar(I,J)` is the difference between the score for the true class and the largest score for other classes. This method is available for classification ensembles only.

`mar = margin(B,X,Y, 'param1',val1, 'param2',val2, ...)` specifies optional parameter name/value pairs:

- 'mode' String indicating how the method computes errors. If set to 'cumulative' (default), `margin` computes cumulative errors and `mar` is an `Nobs`-by-`NTrees` matrix, where the first column gives error from `trees(1)`, second column gives error from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to 'individual', `mar` is a `Nobs`-by-`NTrees` matrix, where each element is an error from each tree in the ensemble. If set to 'ensemble', `mar` a single column of length `Nobs` showing the cumulative margins for the entire ensemble.
- 'trees' Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns a vector of length `NTrees` for 'cumulative' and 'individual' modes, where `NTrees` is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element

## TreeBagger.margin

---

gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.

- 'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. The method uses these weights to combine output from the specified trees by taking a weighted average instead of the simple non-weighted majority vote. You cannot use this argument in the 'individual' mode.
- 'useifort' Logical matrix of size Nobs-by-NTrees indicating which trees should be used to make predictions for each observation. By default the method uses all trees for all observations.

### See Also

`CompactTreeBagger.margin`

**Purpose** Convert matrix to dataset array

**Syntax**  
`ds = mat2dataset(X)`  
`ds = mat2dataset(X,Name,Value)`

**Description** `ds = mat2dataset(X)` converts a matrix to a dataset array.

`ds = mat2dataset(X,Name,Value)` performs the conversion using additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **X - Input matrix**

matrix

Input matrix to convert to a dataset array, specified as an  $M$ -by- $N$  numeric matrix. Each column of  $X$  becomes a variable in the output  $M$ -by- $N$  dataset array.

### **Data Types**

single | double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'NumCols',[1,1,2,1]` specifies that the 3rd and 4th columns of the input matrix should be combined into a single variable.

### **'VarNames' - Variable names for output dataset array**

cell array of strings

Variable names for the output dataset array, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings. You must provide a variable name for each variable in `ds`. The names must be valid MATLAB identifiers, and must be unique.

**Example:** 'VarNames', {'myVar1', 'myVar2', 'myVar3'}

## **'ObsNames' - Observation names for output dataset array**

cell array of strings

Observation names for the output dataset array, specified as the comma-separated pair consisting of 'ObsNames' and a cell array of strings. The names do not need to be valid MATLAB identifiers, but they must be unique.

## **'NumCols' - Number of columns for each variable**

vector of nonnegative integers

Number of columns for each variable in `ds`, specified as the comma-separated pair consisting of 'NumCols' and a vector of nonnegative integers. When the number of columns for a variable is greater than one, `mat2dataset` combines multiple columns in `X` into a single variable in `ds`. The vector you assign to `NumCols` must sum to `size(X,2)`.

For example, to convert a matrix with eight columns into a dataset array with five variables, specify a vector with five elements that sum to eight, such as 'NumCols', [1,1,3,1,2].

## **Output Arguments**

### **ds - Output dataset array**

dataset array

Output dataset array, returned by default with a variable for each column of `X`, and an observation for each row of `X`. If you specify `NumCols`, then the number of variables in `ds` is equal to the length of the specified vector of column numbers.

## **Examples**

### **Convert Matrix to Dataset Array**

Convert a matrix to a dataset array using the default options.

Load sample data.

```
load('fisheriris')
X = meas;
```

```
size(X)
```

```
ans =
```

```
    150     4
```

Convert the matrix to a dataset array.

```
ds = mat2dataset(X);
```

```
size(ds)
```

```
ans =
```

```
    150     4
```

```
ds(1:5,:)
```

```
ans =
```

X1	X2	X3	X4
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5	3.6	1.4	0.2

When you do not specify variable names, `mat2dataset` uses the matrix name and column numbers to create default variable names.

## Convert Matrix to Dataset Array with Variable Names

Load sample data.

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

```
X = meas;
```

```
size(X)
```

```
ans =
```

# mat2dataset

---

```
150    4
```

Convert the matrix to a dataset array, providing a variable name for each of the four column of X.

```
ds = mat2dataset(X, 'VarNames', {'SLength', ...  
'SWidth', 'PLength', 'PWidth'});  
size(ds)
```

```
ans =
```

```
150    4
```

```
ds(1:5,:)
```

```
ans =
```

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

## Create a Dataset Array with Multicolumn Variables

Convert a matrix to a dataset array containing multicolumn variables.

Load sample data.

```
load('fisheriris')  
X = meas;  
size(X)
```

```
ans =
```

```
150    4
```



Convert the matrix to a dataset array, combining the sepal measurements (the first two columns) into one variable named `SepalMeas`, and the petal measurements (third and fourth columns) into one variable names `PetalMeas`.

```
ds = mat2dataset(X,'NumCols',[2,2],...
'VarNames',{'SepalMeas','PetalMeas'});
ds(1:5,:)
```

```
ans =
```

SepalMeas		PetalMeas	
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2
4.6	3.1	1.5	0.2
5	3.6	1.4	0.2

The output dataset array has 150 observations and 2 variables.

```
size(ds)
```

```
ans =
```

```
150    2
```

## See Also

`cell2dataset` | `dataset` | `struct2dataset`

## Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Create a Dataset Array from a File” on page 2-71

## Concepts

- “Dataset Arrays” on page 2-135

# mdscale

---

**Purpose** Nonclassical multidimensional scaling

**Syntax**

```
Y = mdscale(D,p)
[Y,stress] = mdscale(D,p)
[Y,stress,disparities] = mdscale(D,p)
[...] = mdscale(D,p,'Name',value)
```

**Description** `Y = mdscale(D,p)` performs nonmetric multidimensional scaling on the  $n$ -by- $n$  dissimilarity matrix  $D$ , and returns  $Y$ , a configuration of  $n$  points (rows) in  $p$  dimensions (columns). The Euclidean distances between points in  $Y$  approximate a monotonic transformation of the corresponding dissimilarities in  $D$ . By default, `mdscale` uses Kruskal's normalized stress1 criterion.

You can specify  $D$  as either a full  $n$ -by- $n$  matrix, or in upper triangle form such as is output by `pdist`. A full dissimilarity matrix must be real and symmetric, and have zeros along the diagonal and non-negative elements everywhere else. A dissimilarity matrix in upper triangle form must have real, non-negative entries. `mdscale` treats NaNs in  $D$  as missing values, and ignores those elements. `Inf` is not accepted.

You can also specify  $D$  as a full similarity matrix, with ones along the diagonal and all other elements less than one. `mdscale` transforms a similarity matrix to a dissimilarity matrix in such a way that distances between the points returned in  $Y$  approximate  $\sqrt{1-D}$ . To use a different transformation, transform the similarities prior to calling `mdscale`.

`[Y,stress] = mdscale(D,p)` returns the minimized stress, i.e., the stress evaluated at  $Y$ .

`[Y,stress,disparities] = mdscale(D,p)` returns the disparities, that is, the monotonic transformation of the dissimilarities  $D$ .

`[...] = mdscale(D,p,'Name',value)` specifies one or more optional parameter name/value pairs that control further details of `mdscale`. Specify *Name* in single quotes. Available parameters are

- **Criterion**— The goodness-of-fit criterion to minimize. This also determines the type of scaling, either non-metric or metric, that mdscale performs. Choices for non-metric scaling are:
  - 'stress' — Stress normalized by the sum of squares of the inter-point distances, also known as stress1. This is the default.
  - 'sstress' — Squared stress, normalized with the sum of 4th powers of the inter-point distances.

Choices for metric scaling are:

- 'metricstress' — Stress, normalized with the sum of squares of the dissimilarities.
  - 'metricsstress' — Squared stress, normalized with the sum of 4th powers of the dissimilarities.
  - 'sammon' — Sammon's nonlinear mapping criterion. Off-diagonal dissimilarities must be strictly positive with this criterion.
  - 'strain' — A criterion equivalent to that used in classical multidimensional scaling.
- **Weights** — A matrix or vector the same size as D, containing nonnegative dissimilarity weights. You can use these to weight the contribution of the corresponding elements of D in computing and minimizing stress. Elements of D corresponding to zero weights are effectively ignored.

---

**Note** When you specify weights as a full matrix, its diagonal elements are ignored and have no effect, since the corresponding diagonal elements of D do not enter into the stress calculation.

---

- **Start** — Method used to choose the initial configuration of points for Y. The choices are

# mdscale

---

- 'cmdscale' — Use the classical multidimensional scaling solution. This is the default. 'cmdscale' is not valid when there are zero weights.
- 'random' — Choose locations randomly from an appropriately scaled  $p$ -dimensional normal distribution with uncorrelated coordinates.
- An  $n$ -by- $p$  matrix of initial locations, where  $n$  is the size of the matrix  $D$  and  $p$  is the number of columns of the output matrix  $Y$ . In this case, you can pass in `[]` for  $p$  and `mdscale` infers  $p$  from the second dimension of the matrix. You can also supply a 3-D array, implying a value for 'Replicates' from the array's third dimension.
- Replicates — Number of times to repeat the scaling, each with a new initial configuration. The default is 1.
- Options — Options for the iterative algorithm used to minimize the fitting criterion. Pass in an options structure created by `statset`. For example,

```
opts = statset(param1, val1, param2, val2, ...);  
[...] = mdscale(..., 'Options', opts)
```

The choices of `statset` parameters are

- 'Display' — Level of display output. The choices are 'off' (the default), 'iter', and 'final'.
- 'MaxIter' — Maximum number of iterations allowed. The default is 200.
- 'TolFun' — Termination tolerance for the stress criterion and its gradient. The default is  $1e-4$ .
- 'TolX' — Termination tolerance for the configuration location step size. The default is  $1e-4$ .

## Examples

```
load cereal.mat  
X = [Calories Protein Fat Sodium Fiber ...
```

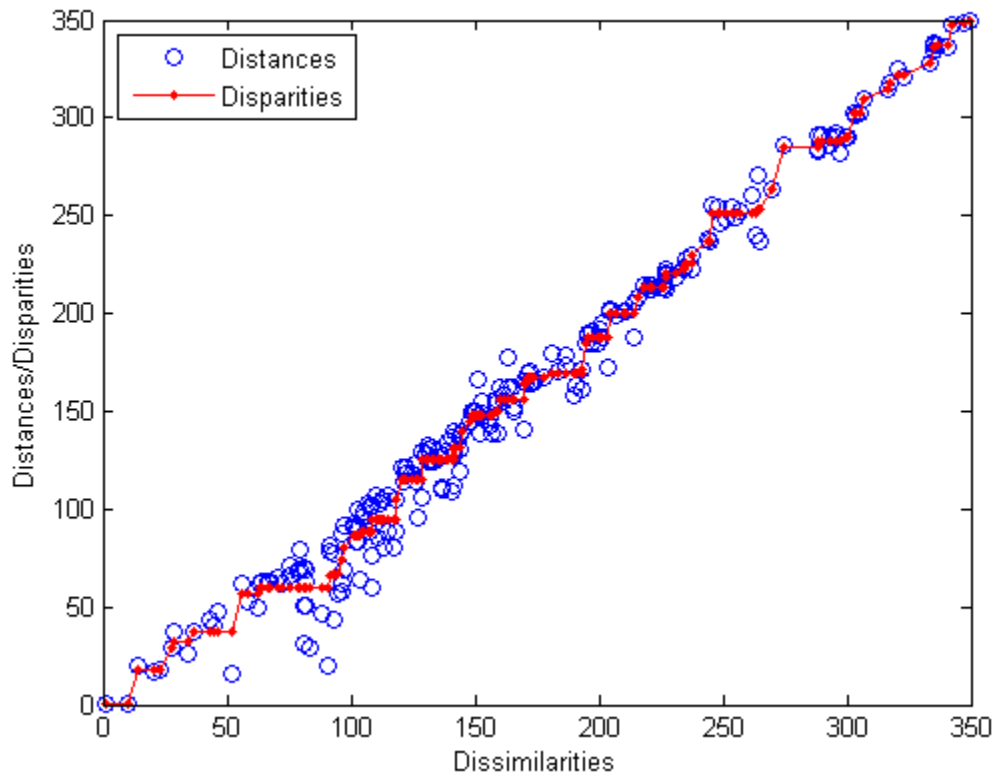
```
Carbo Sugars Shelf Potass Vitamins];

% Take a subset from a single manufacturer.
X = X(strcmp('K',cellstr(Mfg)),:);

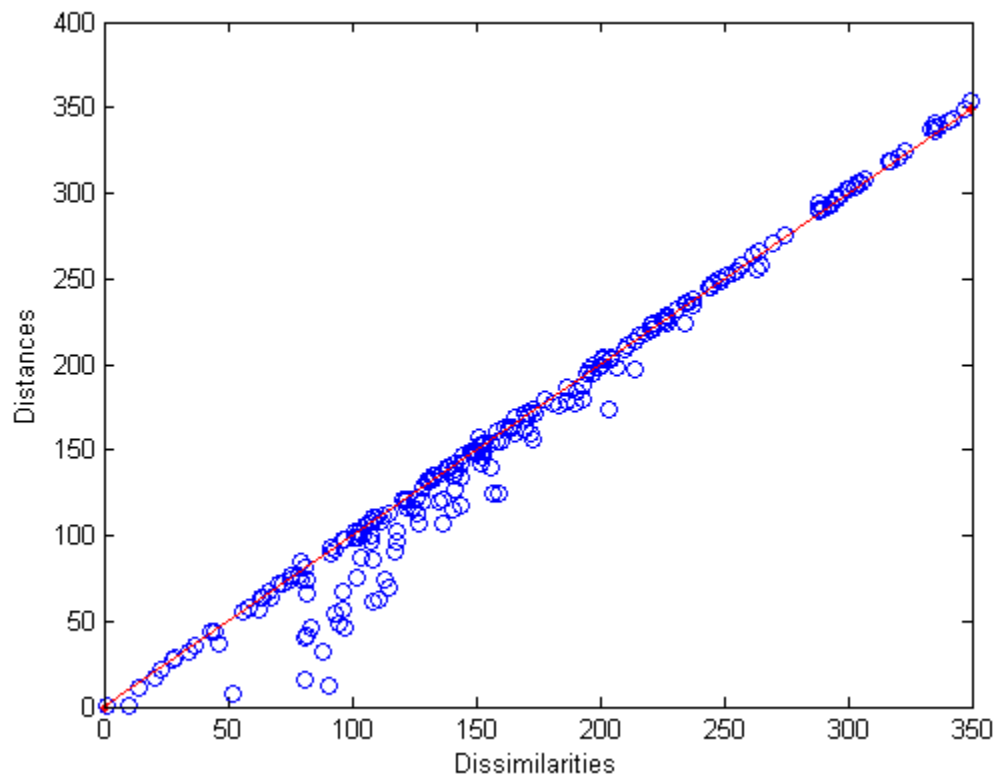
% Create a dissimilarity matrix.
dissimilarities = pdist(X);

% Use non-metric scaling to recreate the data in 2D,
% and make a Shepard plot of the results.
[Y, stress, disparities] = mdscale(dissimilarities, 2);
distances = pdist(Y);
[dum, ord] = sortrows([disparities(:) dissimilarities(:)]);
plot(dissimilarities, distances, 'bo', ...
dissimilarities(ord), disparities(ord), 'r.-');
xlabel('Dissimilarities'); ylabel('Distances/Disparities')
legend({'Distances' 'Disparities'}, 'Location', 'NW');
```

# mdscale



```
% Do metric scaling on the same dissimilarities.  
figure  
[Y, stress] = ...  
mdscale(dissimilarities, 2, 'criterion', 'metricsstress');  
distances = pdist(Y);  
plot(dissimilarities, distances, 'bo', ...  
[0 max(dissimilarities)], [0 max(dissimilarities)], 'r.-');  
xlabel('Dissimilarities'); ylabel('Distances')
```



**See Also** `cmdscale` | `pdist` | `statset`

# CompactTreeBagger.mdsProx

---

**Purpose** Multidimensional scaling of proximity matrix

**Syntax** `[SC,EIGEN] = mdsProx(B,X)`  
`[SC,EIGEN] = mdsProx(B,X,'param1',val1,'param2',val2,...)`

**Description** `[SC,EIGEN] = mdsProx(B,X)` applies classical multidimensional scaling to the proximity matrix computed for the data in the matrix `X`, and returns scaled coordinates `SC` and eigenvalues `EIGEN` of the scaling transformation. The method applies multidimensional scaling to the matrix of distances defined as `1-prox`, where `prox` is the proximity matrix returned by the proximity method.

You can supply the proximity matrix directly by using the `'data'` parameter.

`[SC,EIGEN] = mdsProx(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

- `'data'` Flag indicating how the method treats the `X` input argument. If set to `'predictors'` (default), `mdsProx` assumes `X` to be a matrix of predictors and used for computation of the proximity matrix. If set to `'proximity'`, the method treats `X` as a proximity matrix returned by the proximity method.
- `'colors'` If you supply this argument, `mdsProx` makes overlaid scatter plots of two scaled coordinates using specified colors for different classes. You must supply the colors as a string with one character for each color. If there are more classes in the data than characters in the supplied string, `mdsProx` only plots the first `C` classes, where `C` is the length of the string. For regression or if you do not provide the vector of true class labels, the method uses the first color for all observations in `X`.



- 'labels' Vector of true class labels for a classification ensemble. True class labels can be either a numeric vector, character matrix, or cell array of strings. If supplied, this vector must have as many elements as there are observations (rows) in X. This argument has no effect unless you also supply the 'colors' argument.
- 'mdscoords' Indices of the two scaled coordinates to plot. By default, mdsProx makes a scatter plot of the first and second scaled coordinates which correspond to the two largest eigenvalues. You can specify any other two or three indices not exceeding the dimensionality of the scaled data. This argument has no effect unless you also supply the 'colors' argument.

## See Also

[cmdscales](#) | [TreeBagger.mdsProx](#) | [proximity](#)

# TreeBagger.mdsProx

---

**Purpose** Multidimensional scaling of proximity matrix

**Syntax** `[S,E] = mdsProx(B)`  
`[S,E] = mdsProx(B, 'param1', val1, 'param2', val2, ...)`

**Description** `[S,E] = mdsProx(B)` returns scaled coordinates, S, and eigenvalues, E, for the proximity matrix in the ensemble B. An earlier call to `fillProximities(B)` must create the proximity matrix.

`[S,E] = mdsProx(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

- 'keep' Array of indices of observations in the training data to use for multidimensional scaling. By default, this argument is set to 'all'. If you provide numeric or logical indices, the method uses only the subset of the training data specified by these indices to compute the scaled coordinates and eigenvalues.
- 'colors' If you supply this argument, `mdsProx` makes overlaid scatter plots of two scaled coordinates using specified colors for different classes. You must supply the colors as a string with one character for each color. If there are more classes in the data than characters in the supplied string, `mdsProx` only plots the first C classes, where C is the length of the string. For regression or if you do not provide the vector of true class labels, the method uses the first color for all observations in X.
- 'mdscoords' Indices of the two scaled coordinates to plot. By default, `mdsProx` makes a scatter plot of the first and second scaled coordinates which correspond to the two largest eigenvalues. You can specify any other two or three indices not exceeding the dimensionality of the scaled data. This argument has no effect unless you also supply the 'colors' argument.

**See Also** `cmdscale` | `CompactTreeBagger.mdsProx` | `fillProximities`

<b>Purpose</b>	Mean of probability distribution object
<b>Syntax</b>	<code>m = mean(pd)</code>
<b>Description</b>	<code>m = mean(pd)</code> returns the mean <code>m</code> of the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Fit a probability distribution object to data using <code>fitdist</code> or the Distribution Fitting app.
<b>Output Arguments</b>	<b>m - Mean</b> scalar value  Mean of the probability distribution, returned as a scalar value.
<b>Examples</b>	<b>Mean of a Kernel Distribution</b>  Load the sample data. Create a probability distribution object by fitting a kernel distribution to the miles per gallon (MPG) data.  <pre>load carsmall; pd = fitdist(MPG,'Kernel')  pd =  KernelDistribution  Kernel = normal Bandwidth = 4.11428 Support = unbounded</pre> Compute the mean of the distribution. <pre>mean(pd)</pre>

# prob.KernelDistribution.mean

---

```
ans =
```

```
23.7181
```

## See Also

```
fitdist | dfittool
```

<b>Purpose</b>	Return mean of ProbDistUnivParam object	
<b>Syntax</b>	$M = \text{mean}(PD)$	
<b>Description</b>	$M = \text{mean}(PD)$ returns $M$ , the mean of the ProbDistUnivParam object $PD$ .	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivParam.
<b>Output Arguments</b>	$M$	The mean of the ProbDistUnivParam object $PD$ .
<b>See Also</b>	mean	

# prob.ParametricTruncatableDistribution.mean

---

<b>Purpose</b>	Mean of probability distribution object
<b>Syntax</b>	<code>m = mean(pd)</code>
<b>Description</b>	<code>m = mean(pd)</code> returns the mean <code>m</code> of the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code> .
<b>Output Arguments</b>	<b>m - Mean</b> scalar value  Mean of the probability distribution, returned as a scalar value.

## Examples **Mean of a Uniform Distribution**

Create a uniform distribution object

```
pd = makedist('Uniform','lower',-3,'upper',5)
```

```
pd =
```

```
UniformDistribution
```

```
Uniform distribution
```

```
Lower = -3
```

```
Upper = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1
```

**See Also** `makedist`

# prob.ToolboxFittableParametricDistribution.mean

---

**Purpose** Mean of probability distribution object

**Syntax** `m = mean(pd)`

**Description** `m = mean(pd)` returns the mean `m` of the probability distribution `pd`.

**Input Arguments** **pd - Probability distribution**  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**Output Arguments** **m - Mean**  
scalar value

Mean of the probability distribution, returned as a scalar value.

## Examples **Mean of a Fitted Distribution**

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x,'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]
```



```
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the mean of the fitted distribution.

```
m = mean(pd)
```

```
m =
```

```
75.0083
```

The mean of the normal distribution is equal to the parameter  $\mu$ .

## Mean of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull','a',5,'b',2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the mean of the distribution.

```
mean = mean(pd)
```

```
mean =
```

```
4.4311
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

# CompactTreeBagger.meanMargin

---

**Purpose** Mean classification margin

**Syntax**  
`mar = meanMargin(B,X,Y)`  
`mar = meanMargin(B,X,Y, 'param1',val1, 'param2',val2,...)`

**Description** `mar = meanMargin(B,X,Y)` computes average classification margins for predictors `X` given true response `Y`. The `Y` can be either a numeric vector, character matrix, cell array of strings, categorical vector or logical vector. `meanMargin` averages the margins over all observations (rows) in `X` for each tree. `mar` is a matrix of size 1-by-`NTrees`, where `NTrees` is the number of trees in the ensemble `B`. This method is available for classification ensembles only.

`mar = meanMargin(B,X,Y, 'param1',val1, 'param2',val2,...)` specifies optional parameter name/value pairs:

`'mode'` String indicating how `meanMargin` computes errors. If set to `'cumulative'` (default), is a vector of length `NTrees` where the first element gives mean margin from `trees(1)`, second column gives mean margins from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to `'individual'`, `mar` is a vector of length `NTrees`, where each element is a mean margin from each tree in the ensemble. If set to `'ensemble'`, `mar` is a scalar showing the cumulative mean margin for the entire ensemble.

`'trees'` Vector of indices indicating what trees to include in this calculation. By default, this argument is set to `'all'` and the method uses all trees. If `'trees'` is a numeric vector, the method returns a vector of length `NTrees` for `'cumulative'` and `'individual'` modes, where `NTrees` is the number of elements in the input vector, and a scalar for `'ensemble'` mode. For example, in the `'cumulative'` mode, the first element

gives mean margin from `trees(1)`, the second element gives mean margin from `trees(1:2)` etc.

'`treeweights`' Vector of tree weights. This vector must have the same length as the '`trees`' vector. `meanMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the '`individual`' mode.

## See Also

`TreeBagger.meanMargin`

# TreeBagger.meanMargin

---

**Purpose** Mean classification margin

**Syntax**  
`mar = meanMargin(B,X,Y)`  
`mar = meanMargin(B,X,Y, 'param1',val1, 'param2',val2,...)`

**Description** `mar = meanMargin(B,X,Y)` computes average classification margins for predictors `X` given true response `Y`. The `Y` can be either a numeric vector, character matrix, cell array of strings, categorical vector or logical vector. `meanMargin` averages the margins over all observations (rows) in `X` for each tree. `mar` is a matrix of size 1-by-`NTrees`, where `NTrees` is the number of trees in the ensemble `B`. This method is available for classification ensembles only.

`mar = meanMargin(B,X,Y, 'param1',val1, 'param2',val2,...)` specifies optional parameter name/value pairs:

`'mode'` String indicating how `meanMargin` computes errors. If set to `'cumulative'` (default), is a vector of length `NTrees` where the first element gives mean margin from `trees(1)`, second column gives mean margins from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to `'individual'`, `mar` is a vector of length `NTrees`, where each element is a mean margin from each tree in the ensemble . If set to `'ensemble'`, `mar` is a scalar showing the cumulative mean margin for the entire ensemble .

`'trees'` Vector of indices indicating what trees to include in this calculation. By default, this argument is set to `'all'` and the method uses all trees. If `'trees'` is a numeric vector, the method returns a vector of length `NTrees` for `'cumulative'` and `'individual'` modes, where `NTrees` is the number of elements in the input vector, and a scalar for `'ensemble'` mode. For example, in the `'cumulative'` mode, the first element

gives mean margin from `trees(1)`, the second element gives mean margin from `trees(1:2)` etc.

'`treeweights`' Vector of tree weights. This vector must have the same length as the '`trees`' vector. `meanMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the '`individual`' mode.

## See Also

`CompactTreeBagger.meanMargin`

# classregtree.meansurrvarassoc

---

**Purpose** Mean predictive measure of association for surrogate splits in decision tree

**Syntax** MA = meansurrvarassoc(T)  
MA = meansurrvarassoc(T,N)

**Description** MA = meansurrvarassoc(T) returns a  $p$ -by- $p$  matrix, MA, with predictive measures of association for  $p$  predictors. Element MA( $i,j$ ) is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of optimal splits on predictor  $i$ , including splits for which the predictive measure of association between predictors  $i$  and  $j$  is negative.

MA = meansurrvarassoc(T,N) takes an array N of node numbers and returns the predictive measure of association averaged over the specified nodes.

**See Also** classregtree | surrcutvar | surrvarassoc | surrcutcategories | surrcuttype | surrcutpoint | surrcutflip

# CompactClassificationTree.meanSurrVarAssoc

---

<b>Purpose</b>	Mean predictive measure of association for surrogate splits in decision tree
<b>Syntax</b>	<code>ma = meanSurrVarAssoc(tree)</code> <code>ma = meanSurrVarAssoc(tree,N)</code>
<b>Description</b>	<code>ma = meanSurrVarAssoc(tree)</code> returns a matrix of predictive measures of association for the predictors in <code>tree</code> . <code>ma = meanSurrVarAssoc(tree,N)</code> returns a matrix of predictive measures of association averaged over the nodes in vector <code>N</code> .
<b>Input Arguments</b>	<b>tree</b> A classification tree constructed with <code>ClassificationTree.fit</code> , or a compact regression tree constructed with <code>compact</code> . <b>N</b> Vector of node numbers in <code>tree</code> .
<b>Output Arguments</b>	<b>ma</b> <ul style="list-style-type: none"><li><code>ma = meanSurrVarAssoc(tree)</code> returns a P-by-P matrix, where P is the number of predictors in <code>tree</code>. <code>ma(i,j)</code> is the predictive measure of association between the optimal split on variable <code>i</code> and a surrogate split on variable <code>j</code>. See “Predictive Measure of Association” on page 21-1903.</li><li><code>ma = meanSurrVarAssoc(tree,N)</code> returns a P-by-P representing the predictive measure of association between variables averaged over nodes in the vector <code>N</code>. <code>N</code> contains node numbers from 1 to <code>max(tree.NumNodes)</code>.</li></ul>
<b>Definitions</b>	<b>Predictive Measure of Association</b> The predictive measure of association between the optimal split on variable <code>i</code> and a surrogate split on variable <code>j</code> is:

# CompactClassificationTree.meanSurrVarAssoc

---

$$\lambda_{i,j} = \frac{\min(P_L, P_R) - (1 - P_{L_i L_j} - P_{R_i R_j})}{\min(P_L, P_R)}.$$

Here

- $P_L$  and  $P_R$  are the node probabilities for the optimal split of node  $i$  into Left and Right nodes respectively.
- $P_{L_i L_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Left.
- $P_{R_i R_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Right.

Clearly,  $\lambda_{i,j}$  lies from  $-\infty$  to 1. Variable  $j$  is a worthwhile surrogate split for variable  $i$  if  $\lambda_{i,j} > 0$ .

Element  $ma(i, j)$  is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of optimal splits on predictor  $i$ , including splits for which the predictive measure of association between predictors  $i$  and  $j$  is negative.

## Examples

Find the mean predictive measure of association between the variables in the Fisher iris data:

```
load fisheriris
obj = ClassificationTree.fit(meas,species,'surrogate','on');
msva = meanSurrVarAssoc(obj)
```

```
msva =
    1.0000         0         0         0
         0    1.0000         0         0
    0.4633    0.2500    1.0000    0.5000
```



# CompactClassificationTree.meanSurrVarAssoc

---

```
0.2065    0.1413    0.4022    1.0000
```

Find the mean predictive measure of association averaged over the odd-numbered nodes in obj:

```
N = 1:2:obj.NumNodes;  
msva = meanSurrVarAssoc(obj,N)
```

```
msva =  
    1.0000         0         0         0  
         0    1.0000         0         0  
    0.7600    0.5000    1.0000    1.0000  
    0.4130    0.2826    0.8043    1.0000
```

## See Also

ClassificationTree

# CompactRegressionTree.meanSurrVarAssoc

---

**Purpose** Mean predictive measure of association for surrogate splits in decision tree

**Syntax** `ma = meanSurrVarAssoc(tree)`  
`ma = meanSurrVarAssoc(tree,N)`

**Description** `ma = meanSurrVarAssoc(tree)` returns a matrix of predictive measures of association for the predictors in `tree`.  
`ma = meanSurrVarAssoc(tree,N)` returns a matrix of predictive measures of association averaged over the nodes in vector `N`.

**Input Arguments** **tree**  
A regression tree constructed with `RegressionTree.fit`, or a compact regression tree constructed with `compact`.

**N**  
Vector of node numbers in `tree`.

**Output Arguments** **ma**

- `ma = meanSurrVarAssoc(tree)` returns a P-by-P matrix, where P is the number of predictors in `tree`. `ma(i,j)` is the predictive measure of association between the optimal split on variable `i` and a surrogate split on variable `j`. See “Predictive Measure of Association” on page 21-1906.
- `ma = meanSurrVarAssoc(tree,N)` returns a P-by-P representing the predictive measure of association between variables averaged over nodes in the vector `N`. `N` contains node numbers from 1 to `max(tree.NumNodes)`.

## **Definitions Predictive Measure of Association**

The predictive measure of association between the optimal split on variable `i` and a surrogate split on variable `j` is:

$$\lambda_{i,j} = \frac{\min(P_L, P_R) - (1 - P_{L_i L_j} - P_{R_i R_j})}{\min(P_L, P_R)}.$$

Here

- $P_L$  and  $P_R$  are the node probabilities for the optimal split of node  $i$  into Left and Right nodes respectively.
- $P_{L_i L_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Left.
- $P_{R_i R_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Right.

Clearly,  $\lambda_{i,j}$  lies from  $-\infty$  to 1. Variable  $j$  is a worthwhile surrogate split for variable  $i$  if  $\lambda_{i,j} > 0$ .

Element  $\text{ma}(i, j)$  is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of optimal splits on predictor  $i$ , including splits for which the predictive measure of association between predictors  $i$  and  $j$  is negative.

## Examples

Find the mean predictive measure of association between the Displacement, Horsepower, and Weight variables in the carsmall data:

```
load carsmall
X = [Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG, 'surrogate', 'on');
ma = meanSurrVarAssoc(tree)

ma =
    1.0000    0.2708    0.3854
    0.4764    1.0000    0.4568
```

# CompactRegressionTree.meanSurrVarAssoc

---

```
0.3472    0.2326    1.0000
```

Find the mean predictive measure of association averaged over the odd-numbered nodes in tree:

```
N = 1:2:tree.NumNodes;  
ma = meanSurrVarAssoc(tree,N)
```

```
ma =  
    1.0000    0.2500    0.3750  
    0.5910    1.0000    0.5861  
    0.5000    0.2361    1.0000
```

## See Also

[prune](#) | [RegressionTree](#)

<b>Purpose</b>	Return median of ProbDistUnivKernel object	
<b>Syntax</b>	$M = \text{median}(PD)$	
<b>Description</b>	$M = \text{median}(PD)$ returns $M$ , the median of the ProbDistUnivKernel object $PD$ .	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivKernel.
<b>Output Arguments</b>	$M$	The median of the ProbDistUnivKernel object $PD$ .
<b>See Also</b>	median	

# ProbDistUnivParam.median

---

**Purpose** Return median of ProbDistUnivParam object

**Syntax**  $M = \text{median}(PD)$

**Description**  $M = \text{median}(PD)$  returns  $M$ , the median of the ProbDistUnivParam object  $PD$ .

**Input Arguments**  $PD$  An object of the class ProbDistUnivParam.

**Output Arguments**  $M$  The median of the ProbDistUnivParam object  $PD$ .

**See Also** median

<b>Purpose</b>	Median of probability distribution object
<b>Syntax</b>	<code>m = median(pd)</code>
<b>Description</b>	<code>m = median(pd)</code> returns the median <code>m</code> for the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code> . Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using <code>fitdist</code> or the Distribution Fitting app.
<b>Output Arguments</b>	<b>m - Median</b> scalar value  Median of the probability distribution, returned as a scalar value. The value of <code>m</code> is the 50th percentile of the probability distribution.
<b>Examples</b>	<b>Median of a Fitted Distribution</b>  Load the sample data. Create a vector containing the first column of students' exam grade data.  <pre>load examgrades; x = grades(:,1);</pre> Create a normal distribution object by fitting it to the data. <pre>pd = fitdist(x,'Normal')</pre> <pre>pd =  NormalDistribution</pre>

# prob.TruncatableDistribution.median

---

```
Normal distribution
  mu = 75.0083 [73.4321, 76.5846]
  sigma = 8.7202 [7.7391, 9.98843]
```

Compute the median of the fitted distribution.

```
m = median(pd)
```

```
m =
```

```
75.0083
```

For a symmetrical distribution such as the normal distribution, the median will be equal to the mean,  $\mu$ .

## Median of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull','a',5,'b',2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the median of the distribution.

```
m = median(pd)
```

```
m =
```

```
4.1628
```

For a skewed distribution such as the Weibull distribution, the median and the mean may not be equal.



Calculate the mean of the Weibull distribution and compare it to the median.

```
mean = mean(pd)
```

```
mean =  
    4.4311
```

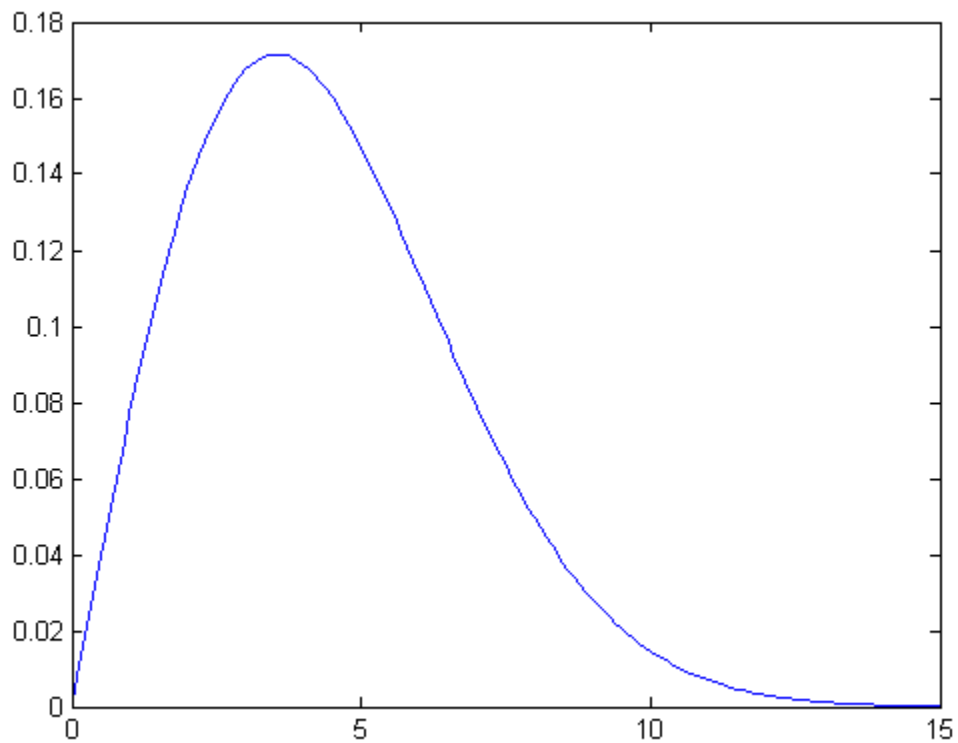
The mean of the distribution is greater than the median.

Plot the pdf to visualize the distribution.

```
x = [0:.1:15];  
pdf = pdf(pd,x);  
plot(x,pdf)
```

# prob.TruncatableDistribution.median

---



## See Also

`makedist` | `fitdist` | `median` | `mean` | `dfittool`

# TreeBagger.MergeLeaves property

---

- Purpose** Flag to merge leaves that do not improve risk
- Description** The MergeLeaves property is true if decision trees have their leaves with the same parent merged for splits that do not decrease the total risk, and false otherwise. The default value is false.
- See Also** `classregtree`

# categorical.mergelevels

---

**Purpose** Merge levels

**Syntax** B = mergelevels(A,oldlevels,newlevel)  
B = mergelevels(A,oldlevels)

**Description** B = mergelevels(A,oldlevels,newlevel) merges two or more levels of the categorical array A into a single new level. oldlevels is a cell array of strings or a 2-D character matrix that specifies the levels to be merged. Any elements of A that have levels in oldlevels are assigned the new level in the corresponding elements of B. newlevel is a character string that specifies the label for the new level. For ordinal arrays, the levels of A specified by oldlevels must be consecutive, and mergelevels inserts the new level to preserve the order of the levels.

B = mergelevels(A,oldlevels) merges two or more levels of A. For nominal arrays, mergelevels uses the first label in oldlevels as the label for the new level. For ordinal arrays, mergelevels uses the label corresponding to the lowest level in oldlevels as the label for the new level.

## **Examples Merge Levels in Nominal Array**

Load the sample data. Create a nominal array containing the iris species data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Create a nominal array containing the iris species data. Display the levels in the nominal array.

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

```
ans =
```

```
setosa    versicolor    virginica
```

Merge the 'setosa' and 'virginica' levels into a single new level called 'parent'.

```
species = mergelevels(species,...  
                       {'setosa','virginica'}, 'parent');
```

Display the new level labels.

```
getlabels(species)
```

```
ans =  
    'parent'    'versicolor'
```

# categorical.mergelevels

---

## Merge Levels in Ordinal Array

Create an ordinal array.

```
A = ordinal([1 2 3 2 1], {'lo', 'med', 'hi'})
```

```
A =  
    lo      med      hi      med      lo
```

Merge the 'lo' and 'med' levels into a single new level labeled 'bad'.

```
A = mergelevels(A, {'lo', 'med'}, 'bad')
```

```
A =  
    bad      bad      hi      bad      bad
```

Any element previous belonging to the levels 'lo' or 'med' now belongs to the level 'bad'.

### See Also

[addlevels](#) | [droplevels](#) | [islevel](#) | [reorderlevels](#) | [getlabels](#)

# TreeBagger.Method property

---

**Purpose**

Method used by trees (classification or regression)

**Description**

The Method property is 'classification' for classification ensembles and 'regression' for regression ensembles.

# mhsample

---

## Purpose

Metropolis-Hastings sample

## Syntax

```
smp1 = mhsample(start, nsamples, 'pdf', pdf, 'proppdf', proppdf,
               'proprnd', proprnd)
smp1 = mhsample(..., 'symmetric', sym)
smp1 = mhsample(..., 'burnin', K)
smp1 = mhsample(..., 'thin', m)
smp1 = mhsample(..., 'nchain', n)
[smp1, accept] = mhsample(...)
```

## Description

`smp1 = mhsample(start, nsamples, 'pdf', pdf, 'proppdf', proppdf, 'proprnd', proprnd)` draws `nsamples` random samples from a target stationary distribution `pdf` using the Metropolis-Hastings algorithm.

`start` is a row vector containing the start value of the Markov Chain, `nsamples` is an integer specifying the number of samples to be generated, and `pdf`, `proppdf`, and `proprnd` are function handles created using `@`. `proppdf` defines the proposal distribution density, and `proprnd` defines the random number generator for the proposal distribution. `pdf` and `proprnd` take one argument as an input with the same type and size as `start`. `proppdf` takes two arguments as inputs with the same type and size as `start`.

`smp1` is a column vector or matrix containing the samples. If the log density function is preferred, `'pdf'` and `'proppdf'` can be replaced with `'logpdf'` and `'logproppdf'`. The density functions used in Metropolis-Hastings algorithm are not necessarily normalized.

The proposal distribution  $q(x,y)$  gives the probability density for choosing  $x$  as the next point when  $y$  is the current point. It is sometimes written as  $q(x|y)$ .

If the `proppdf` or `logproppdf` satisfies  $q(x,y) = q(y,x)$ , that is, the proposal distribution is symmetric, `mhsample` implements Random Walk Metropolis-Hastings sampling. If the `proppdf` or `logproppdf` satisfies  $q(x,y) = q(x)$ , that is, the proposal distribution is independent of current values, `mhsample` implements Independent Metropolis-Hastings sampling.



`smp1 = mhsample(..., 'symmetric', sym)` draws `nsamples` random samples from a target stationary distribution pdf using the Metropolis-Hastings algorithm. `sym` is a logical value that indicates whether the proposal distribution is symmetric. The default value is false, which corresponds to the asymmetric proposal distribution. If `sym` is true, for example, the proposal distribution is symmetric, `proppdf` and `logproppdf` are optional.

`smp1 = mhsample(..., 'burnin', K)` generates a Markov chain with values between the starting point and the  $k^{\text{th}}$  point omitted in the generated sequence. Values beyond the  $k^{\text{th}}$  point are kept. `k` is a nonnegative integer with default value of 0.

`smp1 = mhsample(..., 'thin', m)` generates a Markov chain with `m-1` out of `m` values omitted in the generated sequence. `m` is a positive integer with default value of 1.

`smp1 = mhsample(..., 'nchain', n)` generates `n` Markov chains using the Metropolis-Hastings algorithm. `n` is a positive integer with a default value of 1. `smp1` is a matrix containing the samples. The last dimension contains the indices for individual chains.

`[smp1, accept] = mhsample(...)` also returns `accept`, the acceptance rate of the proposed distribution. `accept` is a scalar if a single chain is generated and is a vector if multiple chains are generated.

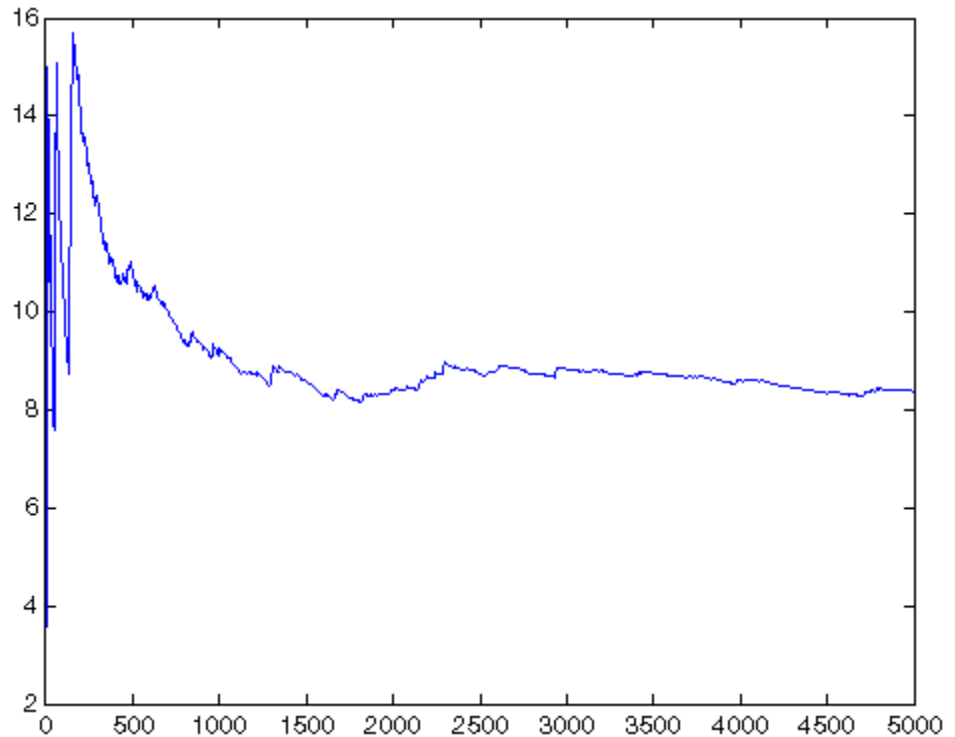
## Examples

Estimate the second order moment of a Gamma distribution using the Independent Metropolis-Hastings sampling.

```
alpha = 2.43;
beta = 1;
pdf = @(x)gampdf(x,alpha,beta); %target distribution
proppdf = @(x,y)gampdf(x,floor(alpha),floor(alpha)/alpha);
proprnd = @(x)sum(...
    exprnd(floor(alpha)/alpha,floor(alpha),1));
nsamples = 5000;
smp1 = mhsample(1,nsamples,'pdf',pdf,'proprnd',proprnd,...
    'proppdf',proppdf);
xxhat = cumsum(smp1.^2)./(1:nsamples)';
```

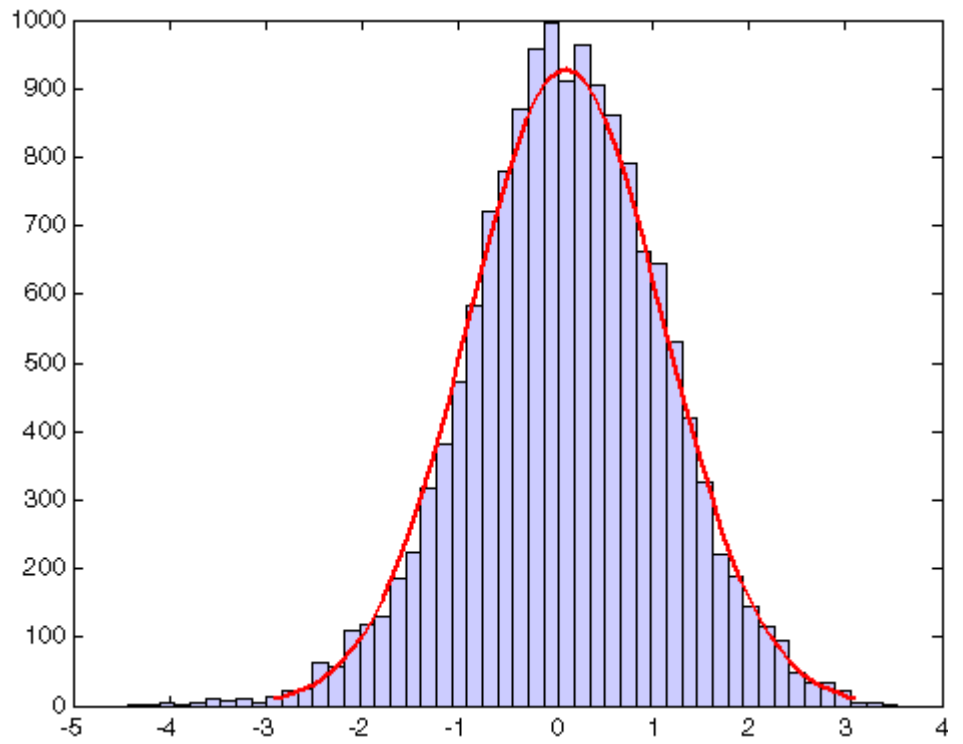
# mhsample

```
plot(1:nsamples,xxhat)
```



Generate random samples from  $N(0,1)$  using the Random Walk Metropolis-Hastings sampling.

```
delta = .5;
pdf = @(x) normpdf(x);
proppdf = @(x,y) unifpdf(y-x,-delta,delta);
proprnd = @(x) x + rand*2*delta - delta;
nsamples = 15000;
x = mhsample(1,nsamples,'pdf',pdf,'proprnd',proprnd,'symmetric',1);
histfit(x,50)
h = get(gca,'Children');
set(h(2),'FaceColor',[.8 .8 1])
```



**See Also**

`slicesample` | `rand`

# TreeBagger.MinLeaf property

---

**Purpose** Minimum number of observations per tree leaf

**Description** The MinLeaf property specifies the minimum number of observations per tree leaf. The default values are 1 for classification and 5 for regression. For `classregtree` training, the 'minparent' value is set to  $2 * \text{MinLeaf}$ .

**See Also** `classregtree`

**Purpose**

Maximum likelihood estimates

**Syntax**

```
phat = mle(data)
phat = mle(data, 'distribution', dist)

phat = mle(data, 'pdf', pdf, 'start', start)
phat = mle(data, 'pdf', pdf, 'start', start, 'cdf', cdf)

phat = mle(data, 'logpdf', logpdf, 'start', start)
phat =
mle(data, 'logpdf', logpdf, 'start', start, 'logsf', logsf)

phat = mle(data, 'nloglf', nloglf, 'start', start)

phat = mle( __, Name, Value)
[phat, pci] = mle( __ )
```

**Description**

`phat = mle(data)` returns maximum likelihood estimates (MLEs) for the parameters of a normal distribution, using the sample data in the vector `data`.

`phat = mle(data, 'distribution', dist)` returns parameter estimates for a distribution specified by `dist`.

`phat = mle(data, 'pdf', pdf, 'start', start)` returns parameter estimates for a custom distribution specified by the probability density function `pdf`. You must also specify the initial parameter values, `start`.

`phat = mle(data, 'pdf', pdf, 'start', start, 'cdf', cdf)` returns parameter estimates for a custom distribution specified by the probability density function `pdf` and custom cumulative distribution function `cdf`.

`phat = mle(data, 'logpdf', logpdf, 'start', start)` returns parameter estimates for a custom distribution specified by the log

probability density function `logpdf`. You must also specify the initial parameter values, `start`.

```
phat =  
mle(data, 'logpdf', logpdf, 'start', start, 'logsf', logsf) returns  
parameter estimates for a custom distribution specified by  
the log probability density function logpdf and custom log  
survival function logsf.
```

```
phat = mle(data, 'nloglf', nloglf, 'start', start) returns  
parameter estimates for the custom distribution specified by the  
negative loglikelihood function nloglf. You must also specify the initial  
parameter values, start.
```

```
phat = mle( ___, Name, Value) also returns the parameter estimates  
with additional options specified by one or more name-value pair  
arguments. You can use any of the input arguments in the previous  
syntaxes.
```

```
[phat, pci] = mle( ___ ) also returns the 95% confidence intervals  
for the parameters.
```

## Input Arguments

### **data** - Sample data

vector

Sample data `mle` uses to estimate the distribution parameters, specified as a vector.

### **Data Types**

single | double

### **dist** - Distribution type

'normal' (default) | string

Distribution type to estimate parameters for, specified as one of the following.

<b>dist</b>	<b>Description</b>	<b>Parameter 1</b>	<b>Parameter 2</b>	<b>Parameter 3</b>
'bernoulli'	“Bernoulli Distribution” on page B-3	p: probability of success for each trial	—	—
'beta' or 'Beta'	“Beta Distribution” on page B-5	a	b	—
'bino' or 'Binomial'	“Binomial Distribution” on page B-9	n: number of trials	p: probability of success for each trial	—
'birnbaumsaunders'	“Birnbaum-Saunders Distribution” on page B-13	$\beta$ : scale	$\gamma$ : shape	—
'burr' or 'Burr'	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale	c: first shape	k: second shape
'Discrete Uniform' or 'unid'	“Uniform Distribution (Discrete)” on page B-139	N: maximum observable value	—	—
'exp' or 'Exponential'	“Exponential Distribution” on page B-32	$\mu$ : mean	—	—
'ev' or 'Extreme Value'	“Extreme Value Distribution” on page B-35	$\mu$ : location	$\sigma$ : scale	—
'gam' or 'Gamma'	“Gamma Distribution” on page B-43	a: shape	b: scale	—
'gev' or 'Generalized Extreme Value'	“Generalized Extreme Value Distribution” on page B-48	k: shape	$\sigma$ : scale	$\mu$ : location

dist	Description	Parameter 1	Parameter 2	Parameter 3
'gp' or 'Generalized Pareto'	“Generalized Pareto Distribution” on page B-53	k: tail index (shape)	$\sigma$ : scale	$\theta$ : threshold
'geo' or 'Geometric'	“Geometric Distribution” on page B-57	p: probability	—	—
'inversegaussian'	“Inverse Gaussian Distribution” on page B-63	$\mu$ : scale	$\lambda$ : shape	—
'logistic'	“Logistic Distribution” on page B-76	$\mu$ : location	$\sigma$ : scale	—
'loglogistic'	“Loglogistic Distribution” on page B-78	$\mu$ : log location	$\sigma$ : log scale	—
'logn' or 'Lognormal'	“Lognormal Distribution” on page B-80	$\mu$ : log location	$\sigma$ : log scale	—
'nakagami'	“Nakagami Distribution” on page B-97	$\mu$ : shape	$\omega$ : scale	—
'nbin' or 'Negative Binomial'	“Negative Binomial Distribution” on page B-98	r: number of successes	p: probability of success in a single trial	—
'norm' or 'Normal'	“Normal Distribution” on page B-110	$\mu$ : location (mean)	$\sigma$ : scale (standard deviation)	—
'poiss' or 'Poisson'	“Poisson Distribution” on page B-118	$\lambda$ : mean	—	—
'rayl' or 'Rayleigh'	“Rayleigh Distribution” on page B-121	b: scale	—	—



dist	Description	Parameter 1	Parameter 2	Parameter 3
'rician'	“Rician Distribution” on page B-123	s: noncentrality	$\sigma$ : scale	—
'tlocationscale'	“t Location-Scale Distribution” on page B-126	$\mu$ : location	$\sigma$ : scale	$\nu$ : degrees of freedom
'unif' or 'Uniform'	“Uniform Distribution (Continuous)” on page B-133	a: lower endpoint (minimum)	b: upper endpoint (maximum)	—
'wbl' or 'Weibull'	“Weibull Distribution” on page B-141	a: scale	b: shape	—

**Example:** 'rician'

### pdf - Custom probability density function

function handle

Custom probability distribution function, specified as a function handle created using @.

This custom function accepts the vector data and one or more individual distribution parameters as input parameters, and returns a vector of probability density values.

For example, if the name of the custom probability density function is newpdf, then you can specify the function handle in mle as follows.

**Example:** @newpdf

### Data Types

function\_handle

### cdf - Custom cumulative distribution function

function handle

Custom cumulative distribution function, specified as a function handle created using @.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of cumulative probability values.

You must define `cdf` with `pdf` if data is censored and you use the 'censoring' name-value pair argument. If 'censoring' is not present, you do not have to specify `cdf` while using `pdf`.

For example, if the name of the custom cumulative distribution function is `newcdf`, then you can specify the function handle in `mle` as follows.

**Example:** `@newcdf`

### **Data Types**

`function_handle`

### **logpdf - Custom log probability density function**

`function handle`

Custom log probability density function, specified as a function handle created using @.

This custom function accepts the vector `data` and one or more individual distribution parameters as input parameters, and returns a vector of log probability values.

For example, if the name of the custom log probability density function is `customlogpdf`, then you can specify the function handle in `mle` as follows.

**Example:** `@customlogpdf`

### **Data Types**

`function_handle`

### **logsf - Custom log survival function**

`function handle`

Custom log survival function, specified as a function handle created using @.

This custom function accepts the vector data and one or more individual distribution parameters as input parameters, and returns a vector of log survival probability values.

You must define logsf with logpdf if data is censored and you use the 'censoring' name-value pair argument. If 'censoring' is not present, you do not have to specify logsf while using logpdf.

For example, if the name of the custom log survival function is logsurvival, then you can specify the function handle in mle as follows.

**Example:** @logsurvival

### Data Types

function\_handle

### nloglf - Custom negative loglikelihood function

function handle

Custom negative loglikelihood function, specified as a function handle created using @.

This custom function accepts the following input arguments.

params	Vector of distribution parameter values. mle detects the number of parameters from the number of
data	Vector of data.
cens	Boolean vector of censored values.
freq	Vector of integer data frequencies.

nloglf must accept all four arguments even if you do not use the 'censoring' or 'frequency' name-value pair arguments. You can write 'nloglf' to ignore cens and freq arguments in that case.

nloglf returns a scalar negative loglikelihood value and optionally, a negative loglikelihood gradient vector (see the 'GradObj' field in 'options').

If the name of the custom negative log likelihood function is `negloglik`, then you can specify the function handle in `mle` as follows.

**Example:** `@negloglik`

### Data Types

`function_handle`

### start - Initial parameter values

`scalar` | `vector`

Initial parameter values for the custom functions, specified as a scalar value or a vector of scalar values.

Use `start` when you fit custom distributions, that is, when you use `pdf` and `cdf`, `logpdf` and `logsf`, or `nloglf` input arguments.

**Example:** `0.05`

**Example:** `[100,2]`

### Data Types

`single` | `double`

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'censoring',Cens,'alpha',0.01,'options',Opt` specifies that `mle` estimates the parameters for the distribution of censored data specified by array `Cens`, computes the 99% confidence limits for the parameter estimates, and uses the algorithm control parameters specified by the structure `Opt`.

### 'censoring' - Indicator for censoring

`array of 0s (default)` | `array of 0s and 1s`

Indicator for censoring, specified as the comma-separated pair consisting of 'censoring' and a Boolean array of the same size as `data`. Use 1 for observations that are right censored and 0 for observations that are fully observed. The default is all observations are fully observed.

Censoring is not supported for all distributions.

For example, if the censored data information is in the binary array called `Censored`, then you can specify the censored data as follows.

**Example:** 'censoring',`Censored`

### Data Types

logical

### 'frequency' - Frequency of observations

array of 1s (default) | vector of nonnegative integer counts

Frequency of observations, specified as the comma-separated pair consisting of 'frequency' and an array containing nonnegative integer counts, which is the same size as `data`. The default is one observation per element of `data`.

For example, if the observation frequencies are stored in an array named `Freq`, you can specify the frequencies as follows.

**Example:** 'frequency',`Freq`

### Data Types

single | double

### 'alpha' - Confidence level

0.05 (default) | scalar value in the range (0,1)

Confidence level for the confidence interval of parameter estimates, `pci`, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range (0,1). The confidence level of `pci` is  $100(1-\alpha)\%$ . The default is 0.05 for 95% confidence.

For example, for 99% confidence limits, you can specify the confidence level as follows.

**Example:** 'alpha',0.01

## **Data Types**

single | double

## **'ntrials' - Number of trials**

scalar value | vector

Number of trials for the corresponding element of data, specified as the comma-separated pair consisting of 'ntrials' and a scalar or a vector of the same size as data.

Applies only to binomial distribution.

**Example:** 'ntrials',total

## **Data Types**

single | double

## **'options' - Fitting algorithm control parameters**

structure

Fitting algorithm control parameters, specified as the comma-separated pair consisting of 'options' and a structure returned by `statset`.

Not applicable to all distributions.

Use the 'options' name-value pair argument to control details of the maximum likelihood optimization when fitting a custom distribution. For parameter names and default values, type `statset('mlecustom')`. You can set the options under a new name and use that in the name-value pair argument. `mle` interprets the following `statset` parameters for custom distribution fitting.

Parameter	Value
'GradObj'	<p>Default is 'off'.</p> <p>'on' or 'off', indicating whether or not <code>fmincon</code> can expect the custom function provided with the <code>nloglf</code> input argument to return the gradient vector of the negative log-likelihood as a second output.</p>
'DerivStep'	<p>Default is <math>\text{eps}^{(1/3)}</math>.</p> <p>The relative difference, specified as a scalar or a vector the same size as <code>start</code>, used in finite difference derivative approximations when using <code>fmincon</code>, and 'GradObj' is 'off'.</p> <p><code>mle</code> ignores 'DerivStep' when using <code>fminsearch</code>.</p>
'FunValCheck'	<p>Default is 'on'.</p> <p>'on' or 'off', indicating whether or not <code>mle</code> should check the values returned by the custom distribution functions for validity.</p> <p>A poor choice of starting point can sometimes cause these functions to return NaNs, infinite values, or out-of-range values if they are written without suitable error checking.</p>
'TolBnd'	<p>Default is <math>1\text{e-}6</math>. An offset for upper and lower bounds when using <code>fmincon</code>.</p> <p><code>mle</code> treats upper and lower bounds as strict inequalities, that is, open bounds. With <code>fmincon</code>, this is approximated by creating closed bounds inset from the specified upper and lower bounds by <code>TolBnd</code>.</p>

Example: `'options', statset('mlecustom')`

**Data Types**  
struct

## **'lowerbound' - Lower bounds for distribution parameters**

$-\infty$  (default) | vector

Lower bounds for distribution parameters, specified as the comma-separated pair consisting of 'lowerbound' and a vector the same size as **start**.

This name-value pair argument is valid only when you use the pdf and cdf, logpdf and logcdf, or nloglf input arguments.

**Example:** 'lowerbound',0

### **Data Types**

single | double

## **'upperbound' - Upper bounds for distribution parameters**

$\infty$  (default) | vector

Upper bounds for distribution parameters, specified as the comma-separated pair consisting of 'upperbound' and a vector the same size as **start**.

This name-value pair argument is valid only when you use the pdf and cdf, logpdf and logsf, or nloglf input arguments.

**Example:** 'upperbound',1

### **Data Types**

single | double

## **'optimfun' - Optimization function**

'fminsearch' (default) | 'fmincon'

Optimization function mle uses in maximizing the likelihood, specified as the comma-separated pair consisting of 'optimfun' and either 'fminsearch' or 'fmincon'.

Default is 'fminsearch'.

You can only specify 'fmincon' if Optimization Toolbox is available.



The 'optimfun' name-value pair argument is valid only when you fit custom distributions, that is, when you use the pdf and cdf, logpdf and logsf, or nloglf input arguments.

**Example:** 'optimfun', 'fmincon'

## Output Arguments

### **phat** - Parameter estimates

scalar value | row vector

Parameter estimates, returned as a scalar value or a row vector.

### **pci** - Confidence intervals for parameter estimates

2-by- $k$  matrix

Confidence intervals for parameter estimates, returned as a column vector or a matrix depending on the number of parameters, hence the size of phat.

pci is a 2-by- $k$  matrix, where  $k$  is the number of parameters mle estimates. The first and second rows of the pci show the upper and lower confidence limits, respectively.

## Examples

### **Estimate Parameters of Burr Distribution**

Load the sample data.

```
load carbig
```

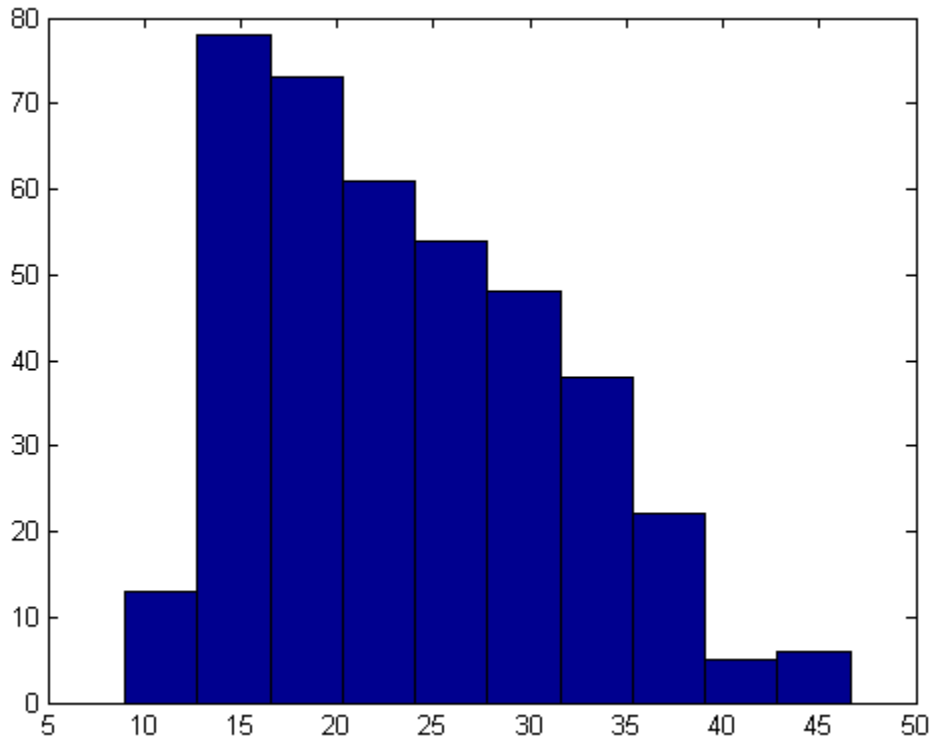
The variable MPG has the miles per gallon for different models of cars.

Draw a histogram of MPG data.

```
hist(MPG)
```

# mle

---



The distribution is somewhat right skewed. A symmetric distribution, such as normal distribution, might not be a good fit.

Estimate the parameters of the Burr Type XII distribution for the MPG data.

```
phat = mle(MPG, 'distribution', 'burr')
```

```
phat =
```

```
34.6447    3.7898    3.5722
```

The maximum likelihood estimates for the scale parameter  $\alpha$  is 34.6447. The estimates for the two shape parameters  $c$  and  $k$  of the Burr Type XII distribution are 3.7898 and 3.5722, respectively.

### Fit Custom Distribution to Censored Data

Navigate to a folder containing sample data.

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

Load the sample data.

```
load readmissiontimes
```

The data includes `ReadmissionTime`, which has readmission times for 100 patients. The column vector `Censored` has the censorship information for each patient, where 1 indicates a censored observation, and 0 indicates the exact readmission time is observed. This is simulated data.

Define a custom probability density and cumulative distribution function.

```
custpdf = @(data,lambda) lambda*exp(-lambda*data);
custcdf = @(data,lambda) 1-exp(-lambda*data);
```

Estimate the parameter, `lambda`, of the custom distribution for the censored sample data.

```
phat = mle(ReadmissionTime,'pdf',custpdf,...
'cdf',custcdf,'start',0.05,'Censoring',Censored)
phat
```

```
phat =
```

```
0.1096
```

## Fit Custom Log pdf and Survival Function

Navigate to a folder containing sample data.

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

Load the sample data.

```
load readmissiontimes
```

The data includes `ReadmissionTime`, which has readmission times for 100 patients. The column vector `Censored` has the censorship information for each patient, where 1 indicates a censored observation, and 0 indicates the exact readmission time is observed. This is simulated data.

Define a custom log probability density and survival function.

```
custlogpdf = @(data,lambda,k) log(k)-k*log(lambda)...
             +(k-1)*log(data)-(data/lambda).^k;
custlogsf = @(data,lambda,k) -(data/lambda).^k;
```

Estimate the parameters, `lambda` and `k`, of the custom distribution for the censored sample data.

```
phat = mle(ReadmissionTime,'logpdf',custlogpdf,...
           'logsf',custlogsf,'start',[1,0.75],'Censoring',Censored)
```

```
phat =
```

```
    9.2090    1.4223
```

The scale and shape parameters of the custom-defined distribution are 9.2090 and 1.4223, respectively.

## Fit Custom Log Negative Likelihood Function

Navigate to a folder containing sample data.

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

Load the sample data.

```
load readmissiontimes
```

The data includes `ReadmissionTime`, which has readmission times for 100 patients. This is simulated data.

Define a negative log likelihood function.

```
custnloglf=@(lambda,data,cens,freq) -length(data)*log(lambda)...
+ nansum(lambda*data);
```

Estimate the parameters of the defined distribution.

```
phat = mle(ReadmissionTime,'nloglf',custnloglf,'start',0.05)
```

```
phat =
```

```
0.1462
```

### **Estimate Probability of Success**

Generate 100 random observations from a binomial distribution with the number of trials,  $n = 20$ , and the probability of success,  $p = 0.75$ .

```
data = binornd(20,0.75,100,1);
```

Estimate the probability of success and 95% confidence limits using the simulated sample data.

```
[phat,pci] = mle(data,'distribution','binomial',...
                'alpha',.05,'ntrials',20)
```

```
phat =
```

```
0.7370
```

```
pci =
```

```
0.7171
```

0.7562

The estimate of probability of success is 0.737 and the lower and upper limits of the 95% confidence interval are 0.7171 and 0.7562. This interval covers the true value used to simulate the data.

## Definitions **Survival Function**

The survival function is the probability of survival as a function of time. It is also called the survivor function. It gives the probability that the survival time of an individual exceeds a certain value. Since the cumulative distribution function,  $F(t)$ , is the probability that the survival time is less than or equal to a given point in time, the survival function for a continuous distribution,  $S(t)$ , is the complement of the cumulative distribution function:  $S(t) = 1 - F(t)$ .

## Tips

When you supply distribution functions, `mle` computes the parameter estimates using an iterative maximization algorithm. With some models and data, a poor choice of starting point can cause `mle` to converge to a local optimum that is not the global maximizer, or to fail to converge entirely. Even in cases for which the log-likelihood is well-behaved near the global maximum, the choice of starting point is often crucial to convergence of the algorithm. In particular, if the initial parameter values are far from the MLEs, underflow in the distribution functions can lead to infinite log-likelihoods.

## See Also

`fitdist` | `mlecov` | `statset`

## Concepts

- “What Is Survival Analysis?” on page 12-2

**Purpose**

Asymptotic covariance of maximum likelihood estimators

**Syntax**

```
acov = mlecov(params,data,'pdf',pdf)
acov = mlecov(params,data,'pdf',pdf,'cdf',cdf)

acov = mlecov(params,data,'logpdf',logpdf)
acov = mlecov(params,data,'logpdf',logpdf,'logsf',logsf)

acov = mlecov(params,data,'nloglf',nloglf)

acov = mlecov( ____,Name,Value)
```

**Description**

`acov = mlecov(params,data,'pdf',pdf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom probability density function `pdf`.

`mlecov` computes a finite difference approximation to the Hessian of the log-likelihood at the maximum likelihood estimates `params`, given the observed data, and returns the negative inverse of that Hessian.

`acov = mlecov(params,data,'pdf',pdf,'cdf',cdf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom probability density function `pdf` and cumulative distribution function `cdf`.

`acov = mlecov(params,data,'logpdf',logpdf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom log probability density function `logpdf`.

`acov = mlecov(params,data,'logpdf',logpdf,'logsf',logsf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution

specified by the custom log probability density function `logpdf` and custom log survival function `logsf`.

`acov = mlecov(params,data,'nloglf',nloglf)` returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters for a distribution specified by the custom negative loglikelihood function `nloglf`.

`acov = mlecov( ___,Name,Value)` also returns an approximation to the asymptotic covariance matrix of the maximum likelihood estimators of the parameters with additional options specified by one or more name-value pair arguments. You can use any of the input arguments in the previous syntaxes.

## Input Arguments

### **params - Parameter estimates**

scalar value | vector

Parameter estimates, specified as a scalar value or vector of scalar values. These parameter estimates must be maximum likelihood estimates. For example, you can specify parameter estimates returned by `mle`.

### **Data Types**

single | double

### **data - Sample data**

vector

Sample data `mle` uses to estimate the distribution parameters, specified as a vector.

### **Data Types**

single | double

### **pdf - Custom probability density function**

function handle



Custom probability distribution function, specified as a function handle created using @.

This custom function accepts the vector data and one or more individual distribution parameters as input parameters, and returns a vector of probability density values.

For example, if the name of the custom probability density function is `newpdf`, then you can specify the function handle in `mlecov` as follows.

**Example:** `@newpdf`

### Data Types

function\_handle

### **cdf - Custom cumulative distribution function**

function handle

Custom cumulative distribution function, specified as a function handle created using @.

This custom function accepts the vector data and one or more individual distribution parameters as input parameters, and returns a vector of cumulative probability values.

You must define `cdf` with `pdf` if data is censored and you use the 'censoring' name-value pair argument. If 'censoring' is not present, you do not have to specify `cdf` while using `pdf`.

For example, if the name of the custom cumulative distribution function is `newcdf`, then you can specify the function handle in `mlecov` as follows.

**Example:** `@newcdf`

### Data Types

function\_handle

### **logpdf - Custom log probability density function**

function handle

Custom log probability density function, specified as a function handle created using @.

This custom function accepts the vector data and one or more individual distribution parameters as input parameters, and returns a vector of log probability values.

For example, if the name of the custom log probability density function is `customlogpdf`, then you can specify the function handle in `mlecov` as follows.

**Example:** `@customlogpdf`

### **Data Types**

`function_handle`

### **logsf - Custom log survival function**

`function handle`

Custom log survival function, specified as a function handle created using `@`.

This custom function accepts the vector data and one or more individual distribution parameters as input parameters, and returns a vector of log survival probability values.

You must define `logsf` with `logpdf` if data is censored and you use the `'censoring'` name-value pair argument. If `'censoring'` is not present, you do not have to specify `logsf` while using `logpdf`.

For example, if the name of the custom log survival function is `logsurvival`, then you can specify the function handle in `mlecov` as follows.

**Example:** `@logsurvival`

### **Data Types**

`function_handle`

### **nloglf - Custom negative loglikelihood function**

`function handle`

Custom negative loglikelihood function, specified as a function handle created using `@`.

This custom function accepts the following input arguments.

params	Vector of distribution parameter values
data	Vector of data
cens	Boolean vector of censored values
freq	Vector of integer data frequencies

`nloglf` must accept all four arguments even if you do not use the 'censoring' or 'frequency' name-value pair arguments. You can write 'nloglf' to ignore `cens` and `freq` arguments in that case.

`nloglf` returns a scalar negative loglikelihood value and optionally, a negative loglikelihood gradient vector (see the 'GradObj' field in 'options').

If the name of the custom negative log likelihood function is `negloglik`, then you can specify the function handle in `mlecov` as follows.

**Example:** `@negloglik`

### Data Types

`function_handle`

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** 'censoring', `cens`, 'options', `opt` specifies that `mlecov` reads the censored data information from the vector `cens` and performs according to the new options structure `opt`.

### 'censoring' - Indicator for censoring

array of 0s (default) | array of 0s and 1s

Indicator for censoring, specified as the comma-separated pair consisting of 'censoring' and a Boolean array of the same size as `data`. Use 1 for observations that are right censored and 0 for observations that are fully observed. The default is all observations are fully observed.

For censored data, you must use `cdf` with `pdf`, or `logsf` with `logpdf`, or `nloglf` must be defined to account for censoring.

For example, if the censored data information is in the binary array called `Censored`, then you can specify the censored data as follows.

**Example:** 'censoring',Censored

### Data Types

logical

### 'frequency' - Frequency of observations

array of 1s (default) | vector of nonnegative integer counts

Frequency of observations, specified as the comma-separated pair consisting of 'frequency' and an array containing nonnegative integer counts, which is the same size as `data`. The default is one observation per element of `data`.

For example, if the observation frequencies are stored in an array named `Freq`, you can specify the frequencies as follows.

**Example:** 'frequency',Freq

### Data Types

single | double

### 'options' - Numerical options

structure

Numerical options for the finite difference Hessian calculation, specified as the comma-separated pair consisting of 'options' and a structure returned by `statset`.

You can set the options under a new name and use it in the name-value pair argument. The applicable `statset` parameters are as follows.

Parameter	Value
'GradObj'	Default is 'off'.  'on' or 'off', indicating whether or not the function provided with the nloglf input argument can return the gradient vector of the negative log-likelihood as a second output.
'DerivStep'	Default is $\text{eps}^{(1/4)}$ .  Relative step size used in finite difference for Hessian calculations. It can be a scalar, or the same size as <code>params</code> . A smaller value than the default might be appropriate if 'GradObj' is 'on'.

**Example:** `'options', statset('mlecov')`

**Data Types**  
struct

## Output Arguments

**acov - Approximation to asymptotic covariance matrix**  
*p*-by-*p* matrix

Approximation to asymptotic covariance matrix, returned as a *p*-by-*p* matrix, where *p* is the number of parameters in `params`.

## Examples

### Custom Probability Density Function

Load the sample data.

```
load carbig
```

The vector `Weight` shows the weights of 406 cars.

In the MATLAB Editor, create a function that returns the probability density function (pdf) of a lognormal distribution. Save the file in your current working folder as `lognormpdf.m`.

```
function newpdf = lognormpdf(data,mu,sigma)
```

```
newpdf = exp((- (log(data) - mu).^2) / (2*sigma^2)) ./ (data*sigma*sqrt(2*pi));
```

Estimate the parameters, mu and sigma, of the custom-defined distribution.

```
phat = mle(Weight, 'pdf', @lognormpdf, 'start', [4.5 0.3])
```

```
phat =
```

```
    7.9600    0.2804
```

Compute the approximate covariance matrix of the parameter estimates.

```
acov = mlecov(phat, Weight, 'pdf', @lognormpdf)
```

```
acov =
```

```
    1.0e-03 *  
    0.1937    -0.0000  
   -0.0000    0.0968
```

Estimate the standard errors of estimates.

```
se = sqrt(diag(acov))
```

```
se =
```

```
    0.0139  
    0.0098
```

The standard error of the estimates of mu and sigma are 0.0139 and 0.0098, respectively.

## Custom Log Probability Density Function

In the MATLAB Editor, create a function that returns the log probability density function of a beta distribution. Save the file in your current working folder as `betalogpdf.m`.

```
function logpdf = betalogpdf(x,a,b)
logpdf = (a-1)*log(x)+(b-1)*log(1-x)-betaIn(a,b);
```

Generate sample data from a beta distribution with parameters 1.23 and 3.45 and estimate the parameters using the simulated data.

```
rng('default')
x = betarnd(1.23,3.45,25,1);
phat = mle(x,'dist','beta')
```

```
phat =

    1.1213    2.7182
```

Compute the approximate covariance matrix of the parameter estimates.

```
acov = mlecov(phat,x,'logpdf',@betalogpdf)
```

```
acov =

    0.0810    0.1646
    0.1646    0.6074
```

## Custom Log pdf and Survival Function

Navigate to a folder containing sample data.

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

Load the sample data.

```
load readmissiontimes
```

The sample data includes `ReadmissionTime`, which has readmission times for 100 patients. The column vector `Censored` has the censorship information for each patient, where 1 indicates a censored observation, and 0 indicates the exact readmission time is observed. This is simulated data.

Define a custom log probability density and survival function.

```
custlogpdf = @(data,lambda,k) log(k)-k*log(lambda)...  
             +(k-1)*log(data)-(data/lambda).^k;  
custlogsf = @(data,lambda,k) -(data/lambda).^k;
```

Estimate the parameters, `lambda` and `k`, of the custom distribution for the censored sample data.

```
phat = mle(ReadmissionTime,'logpdf',custlogpdf,...  
          'logsf',custlogsf,'start',[1,0.75],'Censoring',Censored)
```

```
phat =  
  
    9.2090    1.4223
```

The scale and shape parameters of the custom-defined distribution are 9.2090 and 1.4223, respectively.

Compute the approximate covariance matrix of the parameter estimates.

```
acov = mlecov(phat,ReadmissionTime,...  
             'logpdf',custlogpdf,'logsf',custlogsf,'Censoring',Censored)
```

```
acov =  
  
    0.5653    0.0102  
    0.0102    0.0163
```



## Custom Log Negative Likelihood Function

Navigate to a folder containing sample data.

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

Load the sample data.

```
load readmissiontimes
```

The sample data includes `ReadmissionTime`, which has readmission times for 100 patients. This is simulated data.

Define a negative log likelihood function.

```
custnloglf = @(lambda,data,cens,freq) -length(data)*log(lambda)...
+ nansum(lambda*data);
```

Estimate the parameters of the defined distribution.

```
phat = mle(ReadmissionTime,'nloglf',custnloglf,'start',0.05)
```

```
phat =
```

```
0.1462
```

Compute the variance of the parameter estimate.

```
acov = mlecov(phat,ReadmissionTime,'nloglf',custnloglf)
```

```
acov =
```

```
2.1374e-04
```

Compute the standard error.

```
sqrt(acov)
```

```
ans =
```

0.0146

## **Definitions**      **Survival Function**

The survival function is the probability of survival as a function of time. It is also called the survivor function. It gives the probability that the survival time of an individual exceeds a certain value. Since the cumulative distribution function,  $F(t)$ , is the probability that the survival time is less than or equal to a given point in time, the survival function for a continuous distribution,  $S(t)$ , is the complement of the cumulative distribution function:  $S(t) = 1 - F(t)$ .

## **See Also**      mle

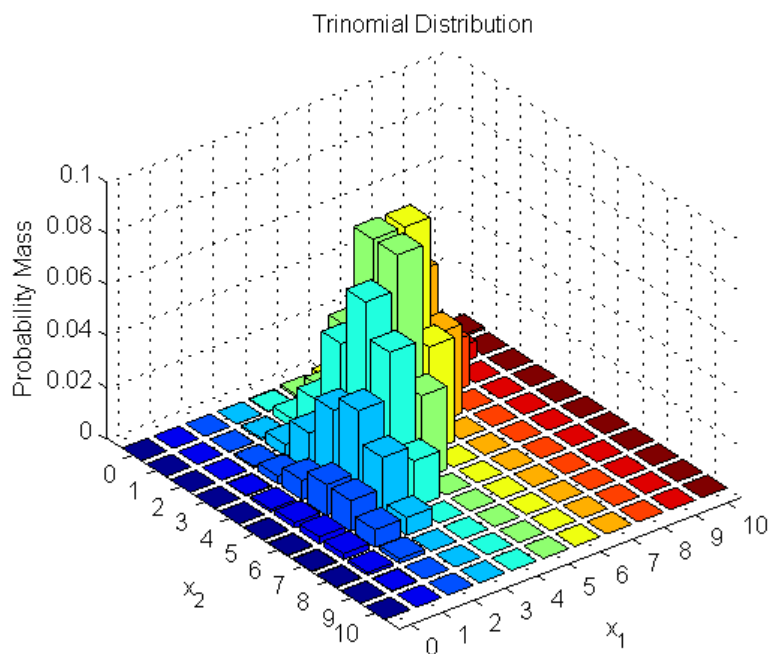
## **Concepts**      • “What Is Survival Analysis?” on page 12-2

<b>Purpose</b>	Multinomial probability density function
<b>Syntax</b>	<code>Y = mnpdf(X,PROB)</code>
<b>Description</b>	<code>Y = mnpdf(X,PROB)</code> returns the pdf for the multinomial distribution with probabilities <code>PROB</code> , evaluated at each row of <code>X</code> . <code>X</code> and <code>PROB</code> are $m$ -by- $k$ matrices or 1-by- $k$ vectors, where $k$ is the number of multinomial bins or categories. Each row of <code>PROB</code> must sum to one, and the sample sizes for each observation (rows of <code>X</code> ) are given by the row sums <code>sum(X,2)</code> . <code>Y</code> is an $m$ -by- $k$ matrix, and <code>mnpdf</code> computes each row of <code>Y</code> using the corresponding rows of the inputs, or replicates them if needed.

**Examples**

```
% Compute the distribution
p = [1/2 1/3 1/6]; % Outcome probabilities
n = 10; % Sample size
x1 = 0:n;
x2 = 0:n;
[X1,X2] = meshgrid(x1,x2);
X3 = n - (X1+X2);
Y = mnpdf([X1(:),X2(:),X3(:)],repmat(p,(n+1)^2,1));

% Plot the distribution
Y = reshape(Y,n+1,n+1);
bar3(Y)
set(gca,'XTickLabel',0:n)
set(gca,'YTickLabel',0:n)
xlabel('x_1')
ylabel('x_2')
zlabel('Probability Mass')
title('Trinomial Distribution')
```



Note that the visualization does not show  $x_3$ , which is determined by the constraint  $x_1 + x_2 + x_3 = n$ .

**See Also**

mnrnd

**How To**

- “Multinomial Distribution” on page B-83

**Purpose** Multinomial logistic regression

**Syntax**

```
B = mnrfit(X,Y)
B = mnrfit(X,Y,Name,Value)
[B,dev,stats] = mnrfit( ___ )
```

**Description** `B = mnrfit(X,Y)` returns a matrix, `B`, of coefficient estimates for a multinomial logistic regression of the nominal responses in `Y` on the predictors in `X`.

`B = mnrfit(X,Y,Name,Value)` returns a matrix, `B`, of coefficient estimates for a multinomial model fit with additional options specified by one or more `Name,Value` pair arguments.

For example, you can fit a nominal, an ordinal, or a hierarchical model, or change the link function.

`[B,dev,stats] = mnrfit( ___ )` also returns the deviance of the fit, `dev`, and the structure `stats` for any of the previous input arguments. `stats` contains model statistics such as degrees of freedom, standard errors for coefficient estimates, and residuals.

## Input Arguments

### **X - Observations on predictor variables**

*n*-by-*p* matrix

Observations on predictor variables, specified as an *n*-by-*p* matrix. `X` contains *n* observations for *p* predictors.

---

**Note** `mnrfit` automatically includes a constant term (intercept) in all models. Do not include a column of 1s in `X`.

---

### **Data Types**

single | double

## **Y - Response values**

*n*-by-*k* matrix | *n*-by-1 column vector

Response values, specified as a column vector or a matrix. Y can be one of the following:

- An *n*-by-*k* matrix, where  $Y(i,j)$  is the number of outcomes of the multinomial category *j* for the predictor combinations given by  $X(i,:)$ . In this case, the number of observations are made at each predictor combination.
- An *n*-by-1 column vector of scalar integers from 1 to *k* indicating the value of the response for each observation. In this case, all sample sizes are 1.

## **Data Types**

single | double

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

**Example:** 'model', 'ordinal', 'link', 'probit' specifies an ordinal model with a probit link function.

## **'model' - Type of model to fit**

'nominal' (default) | 'ordinal' | 'hierarchical'

Type of model to fit, specified as the comma-separated pair consisting of 'model' and one of the following.

'nominal'	Default. There is no ordering among the response categories.
'ordinal'	There is a natural ordering among the response categories.
'hierarchical'	The choice of response category is sequential/nested.

**Example:** 'model', 'ordinal'

### 'interactions' - Indicator for interaction between multinomial categories and coefficients

'on' | 'off'

Indicator for an interaction between the multinomial categories and coefficients, specified as the comma-separated pair consisting of 'interactions' and one of the following.

'on'	Default for nominal and hierarchical models. Fit a model with different coefficients across categories.
'off'	Default for ordinal models. Fit a model with a common set of coefficients for the predictor variables, across all multinomial categories. This is often described as <i>parallel regression</i> or the <i>proportional odds model</i> .

In all cases, the model has different intercepts across categories. The choice of 'interactions' determines the dimensions of the output array B.

**Example:** 'interactions', 'off'

### Data Types

logical

### 'link' - Link function

'logit' (default) | 'probit' | 'comploglog' | 'loglog'

Link function to use for ordinal and hierarchical models, specified as the comma-separated pair consisting of 'link' and one of the following.

'logit'	Default. $f(\gamma) = \ln(\gamma/(1 - \gamma))$
'probit'	$f(\gamma) = \Phi^{-1}(\gamma)$ — error term is normally distributed with variance 1
'comloglog'	Complementary log-log $f(\gamma) = \ln(-\ln(1 - \gamma))$
'loglog'	$f(\gamma) = \ln(-\ln(\gamma))$

The link function defines the relationship between response probabilities and the linear combination of predictors,  $X\beta$ . The link functions might be functions of cumulative or conditional probabilities based on whether the model is for an ordinal or a sequential/nested response. For example, for an ordinal model,  $\gamma$  represents the cumulative probability of being in categories 1 to  $j$  and the model with a logit link function as follows:

$$\ln\left(\frac{\gamma}{1 - \gamma}\right) = \ln\left(\frac{\pi_1 + \pi_2 + \dots + \pi_j}{\pi_{j+1} + \dots + \pi_k}\right) = \beta_{0j} + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p,$$

where  $k$  represents the last category.

You cannot specify the 'link' parameter for nominal models; these always use a multinomial logit link,

$$\ln\left(\frac{\pi_j}{\pi_r}\right) = \beta_{j0} + \beta_{j1} X_{j1} + \beta_{j2} X_{j2} + \dots + \beta_{jp} X_{jp}, \quad j = 1, \dots, k - 1,$$

where  $\pi$  stands for a categorical probability, and  $r$  corresponds to the reference category. mnrfit uses the last category as the reference category for nominal models.

**Example:** 'link', 'loglog'

**'estdisp' - Indicator for estimating dispersion parameter**

'off' (default) | 'on'



Indicator for estimating a dispersion parameter, specified as the comma-separated pair consisting of 'estdisp' and one of the following.

'off'	Default. Use the theoretical dispersion value of 1.
'on'	Estimate a dispersion parameter for the multinomial distribution in computing standard errors.

**Example:** 'estdisp', 'on'

## Output Arguments

### **B - Coefficient estimates**

vector | matrix

Coefficient estimates for a multinomial logistic regression of the responses in  $Y$ , returned as a vector or a matrix.

- If 'interaction' is 'off', then  $B$  is a  $k - 1 + p$  vector. The first  $k - 1$  rows of  $B$  correspond to the intercept terms, one for each  $k - 1$  multinomial categories, and the remaining  $p$  rows correspond to the predictor coefficients, which are common for all of the first  $k - 1$  categories.
- If 'interaction' is 'on', then  $B$  is a  $(p + 1)$ -by- $(k - 1)$  matrix. Each column of  $B$  corresponds to the estimated intercept term and predictor coefficients, one for each of the first  $k - 1$  multinomial categories.

The estimates for the  $k$ th category are taken to be zero as `mnrfit` takes the last category as the reference category.

### **dev - Deviance of the fit**

scalar value

Deviance of the fit, returned as a scalar value. It is twice the difference between the maximum achievable log likelihood and that attained under the fitted model. This corresponds to the sum of deviance residuals,

$$dev = 2 * \sum_i^n \sum_j^k y_{ij} * \log \left( \frac{y_{ij}}{\pi_{ij} * m_i} \right) = \sum_i^n rd_i,$$

where  $rd_i$  are the deviance residuals. For deviance residuals see `stats`.

## stats - Model statistics

structure

Model statistics, returned as a structure that contains the following fields.

beta	The coefficient estimates. These are the same as B.
dfc	Degrees of freedom for error <ul style="list-style-type: none"> <li>• If 'interactions' is 'off', then degrees of freedom is <math>n*(k - 1) - (k - 1 + p)</math>.</li> <li>• If 'interactions' is 'on', then degrees of freedom is <math>(n - p + 1)*(k - 1)</math>.</li> </ul>
sfit	Estimated dispersion parameter.
s	Theoretical or estimated dispersion parameter. <ul style="list-style-type: none"> <li>• If estdisp is 'off', then s is the theoretical dispersion parameter, 1.</li> <li>• If estdisp is 'on', then s is equal to the estimated dispersion parameter, sfit.</li> </ul>
estdisp	Indicator for a theoretical or estimated dispersion parameter.
se	Standard errors of coefficient estimates, B.
coeffcorr	Estimated correlation matrix for B.
covb	Estimated covariance matrix for B.
t	<i>t</i> statistics for B.
p	<i>p</i> -values for B.

resid	<p>Raw residuals. Observed minus fitted values,</p> $r_{ij} = y_{ij} - \hat{\pi}_{ij} * m_i, \quad \begin{cases} i = 1, \dots, n \\ j = 1, \dots, k \end{cases}$ <p>where <math>\pi_{ij}</math> is the categorical, cumulative or conditional probability, and <math>m_i</math> is the corresponding sample size.</p>
residp	<p>Pearson residuals, which are the raw residuals scaled by the estimated standard deviation:</p> $rp_{ij} = \frac{r_{ij}}{\hat{\sigma}_{ij}} = \frac{y_{ij} - \hat{\pi}_{ij} * m_i}{\sqrt{\hat{\pi}_{ij} * (1 - \hat{\pi}_{ij}) * m_i}}, \quad \begin{cases} i = 1, \dots, n \\ j = 1, \dots, k \end{cases}$ <p>where <math>\pi_{ij}</math> is the categorical, cumulative, or conditional probability, and <math>m_i</math> is the corresponding sample size.</p>
residd	<p>Deviance residuals:</p> $rd_i = 2 * \sum_j^k y_{ij} * \log \left( \frac{y_{ij}}{\pi_{ij} * m_i} \right), \quad i = 1, \dots, n.$ <p>where <math>\pi_{ij}</math> is the categorical, cumulative, or conditional probability, and <math>m_i</math> is the corresponding sample size.</p>

## Examples

### Multinomial Regression for Nominal Responses

Fit a multinomial regression for nominal outcomes and interpret the results.

Load the sample data.

```
load('fisheriris.mat')
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Define the nominal response variable.

```
sp = nominal(species);  
sp = double(sp);
```

Now in `sp`, 1, 2, and 3 indicate the species `setosa`, `versicolor`, and `virginica`, respectively.

Fit a multinomial regression model to predict the species using the measurements.

```
[B,dev,stats] = mnrfity(meas,sp);  
B
```

B =

13.3860	15.4492
2.4623	1.8196
5.2948	2.5700
-7.4916	-4.3714
-8.9322	-7.6467

This is a nominal model for the response category relative risks, with separate slopes on all four predictors, that is, each category of `meas`. The first row of `B` contains the intercept terms for the relative risk of the first two response categories, `setosa` and `versicolor` versus the reference category, `virginica`. The last four rows contain the slopes for the models for the first two categories. `mnrfit` accepts the third category as the reference category.

The models for the relative risk of an iris flower being a setosa versus a virginica, and the relative risk of an iris flower being a versicolor species versus a virginica species are respectively

$$\ln\left(\frac{\pi_{setosa}}{\pi_{virginica}}\right) = 13.38 + 2.46X_1 + 5.29X_2 - 7.49X_3 - 8.93X_4$$

and

$$\ln\left(\frac{\pi_{versicolor}}{\pi_{virginica}}\right) = 15.45 + 1.82X_1 + 2.57X_2 - 4.37X_3 - 7.65X_4.$$

The coefficients express the effects of the predictor variables on the relative risk or the log odds of being in one category versus the reference category.

For example, the estimated coefficient 2.46 indicates that the probability of being species 1 (setosa) compared to the probability of being species 3 (virginica) (the relative risk of being a setosa versus a virginica) increases  $\exp(2.46)$  times for each unit increase in  $X_1$ , the first measurement, given all else equal.

In terms of log odds, you can say that the relative log odds of being a setosa versus a virginica increases 2.46 times with a one-unit increase in  $X_1$  given all else is equal.

Check the statistical significance of the model coefficients.

```
stats.p
```

```
ans =
```

0.2457	0.0031
0.4543	0.1048
0.0773	0.0815
0.0258	0.0007
0.1856	0.0002

The  $p$ -value of 0.0258 indicates that the third measure is significant on the relative risk of being a setosa versus a virginica (species 1 compared to species 3). The  $p$ -values of 0.0007 and 0.0002 indicate that the third and fourth measures are significant on the relative risk of being a versicolor versus a virginica (species 2 compared to species 3).

Request the standard errors of coefficient estimates.

```
stats.se
```

```
ans =
```

```
    11.5316    5.2201
     3.2905    1.1218
     2.9976    1.4753
     3.3609    1.2869
     6.7474    2.0846
```

Calculate the 95% confidence limits for the coefficients.

```
LL = stats.beta - 1.96.*stats.se;
UL = stats.beta + 1.96.*stats.se;
```

Display the confidence intervals for the coefficients of the model for the relative risk of being a setosa versus a virginica (the first column of coefficients in B).

```
[LL(:,1) UL(:,1)]
```

```
ans =
```

```
   -9.2160   35.9880
   -3.9869    8.9116
   -0.5805   11.1701
  -14.0790   -0.9043
  -22.1570    4.2926
```

Find the confidence intervals for the coefficients of the model for the relative risk of being a versicolor versus a virginica (the second column of coefficients in B).

```
[LL(:,2) UL(:,2)]
```

```
ans =
```

```
    5.2177    25.6807
   -0.3791     4.0184
   -0.3216     5.4615
   -6.8938    -1.8490
  -11.7324    -3.5610
```

### Multinomial Regression for Ordinal Responses

Fit a multinomial regression model for categorical responses with natural ordering among categories.

Load the sample data and define the predictor variables.

```
load('carbig.mat')
X = [Acceleration Displacement Horsepower Weight];
```

The predictor variables are the acceleration, engine displacement, horsepower, and weight of the cars. The response variable is miles per gallon (mpg).

Create an ordinal response variable categorizing MPG into four levels from 9 to 48 mpg by labeling the response values in the range 9–19 as 1, 20–29 as 2, 30–39 as 3, and 40–48 as 4.

```
miles = ordinal(MPG,{'1','2','3','4'},[],[9,19,29,39,48]);
miles = double(miles);
```

Fit an ordinal response model for the response variable miles.

```
[B,dev,stats] = mnrfit(X,miles,'model','ordinal');
B
```

```
B =  
  
-16.6895  
-11.7208  
-8.0606  
0.1048  
0.0103  
0.0645  
0.0017
```

The first three elements of **B** are the intercept terms for the models, and the last four elements of **B** are the coefficients of the covariates, assumed common across all categories. This model corresponds to *parallel regression*, which is also called the *proportional odds* model, where there is a different intercept but common slopes among categories. You can specify this using the 'interactions', 'off' name-value pair argument, which is the default for ordinal models.

```
[B(1:3)'; repmat(B(4:end),1,3)]
```

```
ans =  
-16.6895 -11.7208 -8.0606  
0.1048 0.1048 0.1048  
0.0103 0.0103 0.0103  
0.0645 0.0645 0.0645  
0.0017 0.0017 0.0017
```

The link function in the model is logit ('link', 'logit'), which is the default for an ordinal model. The coefficients express the relative risk or log odds of the mpg of a car being less than or equal to one value versus greater than that value.

The proportional odds model in this example is



$$\ln\left(\frac{P(\text{mpg} \leq 19)}{P(\text{mpg} > 19)}\right) = -16.6895 + 0.1048X_A + 0.0103X_D + 0.0645X_H + 0.0017X_W$$

$$\ln\left(\frac{P(\text{mpg} \leq 29)}{P(\text{mpg} > 29)}\right) = -11.7208 + 0.1048X_A + 0.0103X_D + 0.0645X_H + 0.0017X_W$$

$$\ln\left(\frac{P(\text{mpg} \leq 39)}{P(\text{mpg} > 39)}\right) = -8.0606 + 0.1048X_A + 0.0103X_D + 0.0645X_H + 0.0017X_W$$

For example, the coefficient estimate of 0.1048 indicates that a unit change in acceleration would impact the odds of the mpg of a car being less than or equal to 19 versus more than 19, or being less than or equal to 29 versus greater than 29, or being less than or equal to 39 versus greater than 39, by a factor of  $\exp(0.1048)$  given all else is equal.

Assess the significance of the coefficients.

```
stats.p
```

```
ans =
```

```
0.0000
0.0000
0.0000
0.1899
0.0350
0.0000
0.0118
```

The  $p$ -values of 0.035, 0.0000, and 0.0118 for engine displacement, horsepower, and weight of a car, respectively, indicate that these factors are significant on the odds of mpg of a car being less than or equal to a certain value versus being greater than that value.

## **Hierarchical Multinomial Regression Model**

Fit a hierarchical multinomial regression model.

Navigate to the folder containing sample data.

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

Load the sample data.

```
load smoking
```

The data set `smoking` contains five variables: `sex`, `age`, `weight`, and `systolic` and `diastolic` blood pressure. `Sex` is a binary variable where 1 indicates female patients, and 0 indicates male patients.

Define the response variable.

```
Y = nominal(smoking.Smoker);
Y = double(Y);
```

The data in `Smoker` has four categories:

- 0: Nonsmoker, 0 cigarettes a day
- 1: Smoker, 1–5 cigarettes a day
- 2: Smoker, 6–10 cigarettes a day
- 3: Smoker, 11 or more cigarettes a day

Define the predictor variables.

```
X = [smoking.Sex smoking.Age smoking.Weight...
     smoking.SystolicBP smoking.DiastolicBP];
```

Fit a hierarchical multinomial model.

```
[B,dev,stats] = mnrfity(X,Y,'model','hierarchical');
B
```

B =

43.8148	5.9571	44.0712
1.8709	-0.0230	0.0662
0.0188	0.0625	0.1335
0.0046	-0.0072	-0.0130
-0.2170	0.0416	-0.0324
-0.2273	-0.1449	-0.4824

The first column of **B** includes the intercept and the coefficient estimates for the model of the relative risk of being a nonsmoker versus a smoker. The second column includes the parameter estimates for modeling the log odds of smoking 1–5 cigarettes a day versus more than five cigarettes a day given that a person is a smoker. Finally, the third column includes the parameter estimates for modeling the log odds of a person smoking 6–10 cigarettes a day versus more than 10 cigarettes a day given he/she smokes more than 5 cigarettes a day.

The coefficients differ across categories. You can specify this using the 'interactions', 'on' name-value pair argument, which is the default for hierarchical models. So, the model in this example is

$$\ln\left(\frac{P(y=0)}{P(y>0)}\right) = 43.8148 + 1.8709X_S + 0.0188X_A + 0.0046X_W - 0.2170X_{SBP} - 0.2273X_{DBP}$$

$$\ln\left(\frac{P(1 \leq y \leq 5)}{P(y > 5)}\right) = 5.9571 - 0.0230X_S + 0.0625X_A + 0.0072X_W + 0.0416X_{SBP} - 0.1449X_{DBP}$$

$$\ln\left(\frac{P(6 \leq y \leq 10)}{P(y > 10)}\right) = 44.0712 + 0.0662X_S + 0.1335X_A - 0.0130X_W - 0.0324X_{SBP} - 0.4824X_{DBP}$$

For example, the coefficient estimate of 1.8709 indicates that the likelihood of being a smoker versus a nonsmoker increases by  $\exp(1.8709) = 6.49$  times as the gender changes from female to male given everything else held constant.

Assess the statistical significance of the terms.

```
stats.p
```

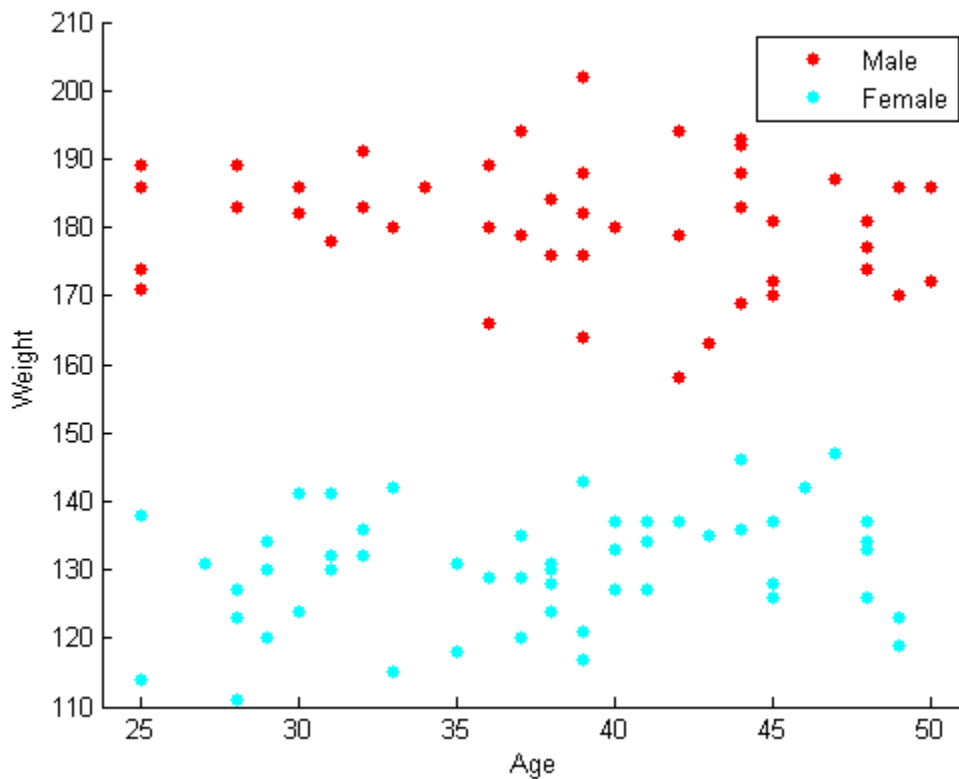
```
ans =
```

```
0.0000    0.5363    0.2149
0.3549    0.9912    0.9835
0.6850    0.2676    0.2313
0.9032    0.8523    0.8514
0.0009    0.5187    0.8165
0.0004    0.0483    0.0545
```

Sex, age, or weight don't appear significant on any level. The  $p$ -values of 0.0009 and 0.0004 indicate that both types of blood pressure are significant on the relative risk of a person being a smoker versus a nonsmoker. The  $p$ -value of 0.0483 shows that only diastolic blood pressure is significant on the odds of a person smoking 0–5 cigarettes a day versus more than 5 cigarettes a day. Similarly, the  $p$ -value of 0.0545 indicates that diastolic blood pressure is significant on the odds of a person smoking 6–10 cigarettes a day versus more than 10 cigarettes a day.

Check if any nonsignificant factors are correlated to each other. Draw a scatterplot of age versus weight grouped by sex.

```
figure()
gscatter(smoking.Age,smoking.Weight,smoking.Sex)
legend('Male','Female')
xlabel('Age')
ylabel('Weight')
```



The range of weight of an individual seems to differ according to gender. Age does not seem to have any obvious correlation with sex or weight. Age is insignificant and weight seems to be correlated with sex, so you can eliminate both and reconstruct the model.

Eliminate age and weight from the model and fit a hierarchical model with sex, systolic blood pressure, and diastolic blood pressure as the predictor variables.

```
X = double([smoking.Sex smoking.SystolicBP...
```

```
smoking.DiastolicBP]);  
[B,dev,stats] = mnrfits(X,Y,'model','hierarchical');  
B
```

```
B =
```

```
    44.8456    5.3230    25.0248  
    1.6045    0.2330    0.4982  
   -0.2161    0.0497    0.0179  
   -0.2222   -0.1358   -0.3092
```

Here, a coefficient estimate of 1.6045 indicates that the likelihood of being a nonsmoker versus a smoker increases by  $\exp(1.6045) = 4.97$  times as sex changes from male to female. A unit increase in the systolic blood pressure indicates an  $\exp(-.2161) = 0.8056$  decrease in the likelihood of being a nonsmoker versus a smoker. Similarly, a unit increase in the diastolic blood pressure indicates an  $\exp(-.2222) = 0.8007$  decrease in the relative rate of being a nonsmoker versus being a smoker.

Assess the statistical significance of the terms.

```
stats.p
```

```
ans =
```

```
    0.0000    0.4715    0.2325  
    0.0210    0.7488    0.6362  
    0.0010    0.4107    0.8899  
    0.0003    0.0483    0.0718
```

The  $p$ -values of 0.0210, 0.0010, and 0.0003 indicate that the terms sex and both types of blood pressure are significant on the relative risk of a person being a nonsmoker versus a smoker, given the other terms in the model. Based on the  $p$ -value of 0.0483, diastolic blood pressure appears significant on the relative risk of a person smoking 1–5 cigarettes versus more than 5 cigarettes a day, given that this person is a smoker. Because none of the  $p$ -values on the third column are less than 0.05,

you can say that none of the variables are statistically significant on the relative risk of a person smoking from 6–10 cigarettes versus more than 10 cigarettes, given that this person smokes more than 5 cigarettes a day.

## Algorithms

mnrfit treats NaNs in either X or Y as missing values, and ignores them.

## References

- [1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [2] Long, J. S. *Regression Models for Categorical and Limited Dependent Variables*. Sage Publications, 1997.
- [3] Dobson, A. J., and A. G. Barnett. *An Introduction to Generalized Linear Models*. Chapman and Hall/CRC. Taylor & Francis Group, 2008.

## See Also

fitglm | mnrvl | glmfit | glmval

## Concepts

- “Multinomial Distribution” on page B-83
- “Multinomial Models for Nominal Responses” on page 10-2
- “Multinomial Models for Ordinal Responses” on page 10-5
- “Hierarchical Multinomial Models” on page 10-9

# mnrnd

---

**Purpose** Multinomial random numbers

**Syntax**  
 $r = \text{mnrnd}(n, p)$   
 $R = \text{mnrnd}(n, p, m)$   
 $R = \text{mnrnd}(N, P)$

**Description**  $r = \text{mnrnd}(n, p)$  returns random values  $r$  from the multinomial distribution with parameters  $n$  and  $p$ .  $n$  is a positive integer specifying the number of trials (sample size) for each multinomial outcome.  $p$  is a 1-by- $k$  vector of multinomial probabilities, where  $k$  is the number of multinomial bins or categories.  $p$  must sum to one. (If  $p$  does not sum to one,  $r$  consists entirely of NaN values.)  $r$  is a 1-by- $k$  vector, containing counts for each of the  $k$  multinomial bins.

$R = \text{mnrnd}(n, p, m)$  returns  $m$  random vectors from the multinomial distribution with parameters  $n$  and  $p$ .  $R$  is a  $m$ -by- $k$  matrix, where  $k$  is the number of multinomial bins or categories. Each row of  $R$  corresponds to one multinomial outcome.

$R = \text{mnrnd}(N, P)$  generates outcomes from different multinomial distributions.  $P$  is a  $m$ -by- $k$  matrix, where  $k$  is the number of multinomial bins or categories and each of the  $m$  rows contains a different set of multinomial probabilities. Each row of  $P$  must sum to one. (If any row of  $P$  does not sum to one, the corresponding row of  $R$  consists entirely of NaN values.)  $N$  is a  $m$ -by-1 vector of positive integers or a single positive integer (replicated by `mnrnd` to a  $m$ -by-1 vector).  $R$  is a  $m$ -by- $k$  matrix. Each row of  $R$  is generated using the corresponding rows of  $N$  and  $P$ .

**Examples** Generate 2 random vectors with the same probabilities:

```
n = 1e3;  
p = [0.2, 0.3, 0.5];  
R = mnrd(n, p, 2)  
R =  
    215    282    503  
    194    303    503
```



Generate 2 random vectors with different probabilities:

```
n = 1e3;  
P = [0.2, 0.3, 0.5; ...  
     0.3, 0.4, 0.3;];  
R = mnrnd(n,P)  
R =  
    186    290    524  
    290    389    321
```

**See Also**

mnpdf

**How To**

- “Multinomial Distribution” on page B-83

# mnrval

---

**Purpose** Multinomial logistic regression values

**Syntax**

```
pihat = mnrvl(B,X)
[pihat,dlow,dhi] = mnrvl(B,X,stats)
[pihat,dlow,dhi] = mnrvl(B,X,stats,Name,Value)

yhat = mnrvl(B,X,ssize)
[yhat,dlow,dhi] = mnrvl(B,X,ssize,stats)
[yhat,dlow,dhi] = mnrvl(B,X,ssize,stats,Name,Value)
```

**Description** `pihat = mnrvl(B,X)` returns the predicted probabilities for the multinomial logistic regression model with predictors,  $X$ , and the coefficient estimates,  $B$ .

`pihat` is an  $n$ -by- $k$  matrix of predicted probabilities for each multinomial category.  $B$  is the vector or matrix that contains the coefficient estimates returned by `mnrfit`. And  $X$  is an  $n$ -by- $p$  matrix which contains  $n$  observations for  $p$  predictors.

---

**Note** `mnrval` automatically includes a constant term in all models. Do not enter a column of 1s in  $X$ .

---

`[pihat,dlow,dhi] = mnrvl(B,X,stats)` also returns 95% error bounds on the predicted probabilities, `pihat`, using the statistics in the structure, `stats`, returned by `mnrfit`.

The lower and upper confidence bounds for `pihat` are `pihat` minus `dlow` and `pihat` plus `dhi`, respectively. Confidence bounds are nonsimultaneous and only apply to the fitted curve, not to new observations.

`[pihat,dlow,dhi] = mnrvl(B,X,stats,Name,Value)` returns the predicted probabilities and 95% error bounds on the predicted probabilities `pihat`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the model type, link function, and the type of probabilities to return.

`yhat = mnrvl(B,X,ssize)` returns the predicted category counts for sample sizes, `ssize`.

`[yhat,dlow,dhi] = mnrvl(B,X,ssize,stats)` also computes 95% error bounds on the predicted counts `yhat`, using the statistics in the structure, `stats`, returned by `mnrfit`.

The lower and upper confidence bounds for `yhat` are `yhat` minus `dlo` and `yhat` plus `dhi`, respectively. Confidence bounds are nonsimultaneous and they apply to the fitted curve, not to new observations.

`[yhat,dlow,dhi] = mnrvl(B,X,ssize,stats,Name,Value)` returns the predicted category counts and 95% error bounds on the predicted counts `yhat`, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the model type, link function, and the type of predicted counts to return.

## Input Arguments

### **B - Coefficient estimates**

vector or matrix returned by `mnrfit`

Coefficient estimates for the multinomial logistic regression model, specified as a vector or matrix returned by `mnrfit`. It is a vector or matrix depending on the model and interactions.

**Example:** `B = mnrfit(X,y); pihat = mnrvl(B,X)`

### **Data Types**

single | double

### **X - Sample data**

matrix

Sample data on predictors, specified as an  $n$ -by- $p$ .  $X$  contains  $n$  observations for  $p$  predictors.

---

**Note** `mnrval` automatically includes a constant term in all models. Do not enter a column of 1s in  $X$ .

---

**Example:** `pihat = mnrfval(B,X)`

### Data Types

single | double

### stats - Model statistics

structure returned by `mnrfit`

Model statistics, specified as a structure returned by `mnrfit`. You must use the `stats` input argument in `mnrval` to compute the lower and upper error bounds on the category probabilities and counts.

**Example:** `[B,dev,stats] = mnrfval(X,y);[pihat,dlo,dhi] = mnrfval(B,X,stats)`

### ssize - Sample sizes

column vector of positive integers

Sample sizes to return the number of items in response categories for each combination of the predictor variables, specified as an  $n$ -by-1 column vector of positive integers.

For example, for a response variable having three categories, if an observation of the number of individuals in each category is  $y_1$ ,  $y_2$ , and  $y_3$ , respectively, then the sample size,  $m$ , for that observation is  $m = y_1 + y_2 + y_3$ .

If the sample sizes for  $n$  observations are in vector `sample`, then you can enter the sample sizes as follows.

**Example:** `yhat = mnrfval(B,X,sample)`

**Data Types**

single | double

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:**

`'model', 'ordinal', 'link', 'probit', 'type', 'cumulative'` specifies that `mnrval` returns the estimates for cumulative probabilities for an ordinal model with a probit link function.

**'model' - Type of multinomial model**

'nominal' (default) | 'ordinal' | 'hierarchical'

Type of multinomial model fit by `mnrfit`, specified as the comma-separated pair consisting of `'model'` and one of the following.

'nominal'	Default. Specify when there is no ordering among the response categories.
'ordinal'	Specify when there is a natural ordering among the response categories.
'hierarchical'	Specify when the choice of response category is sequential.

**Example:** `'model', 'ordinal'`

**'interactions' - Indicator for interaction between multinomial categories and coefficients**

'on' | 'off'

Indicator for an interaction between the multinomial categories and coefficients in the model fit by `mnrfit`, specified as the comma-separated pair consisting of `'interactions'` and one of the following.

'on'	Default for nominal and hierarchical models. Specify to fit a model with different intercepts and coefficients across categories.
'off'	Default for ordinal models. Specify to fit a model with different intercepts, but a common set of coefficients for the predictor variables, across all multinomial categories. This is often described as <i>parallel regression</i> or <i>proportional odds model</i> .

**Example:** 'interactions','off'

## Data Types

logical

## 'link' - Link function

'logit' (default) | 'probit' | 'comploglog' | 'loglog'

Link function `mnrfit` uses for ordinal and hierarchical models, specified as the comma-separated pair consisting of 'link' and one of the following.

'logit'	Default. $f(\gamma) = \ln(\gamma/(1 - \gamma))$
'probit'	$f(\gamma) = \Phi^{-1}(\gamma)$ — error term is normally distributed with variance 1
'comploglog'	Complementary log-log $f(\gamma) = \ln(-\ln(1 - \gamma))$
'loglog'	$f(\gamma) = \ln(-\ln(\gamma))$

The link function defines the relationship between response probabilities and the linear combination of predictors,  $X\beta$ .

$\gamma$  might be cumulative or conditional probabilities based on whether the model is for an ordinal or a sequential/nested response.

You cannot specify the 'link' parameter for nominal models; these always use a multinomial logit link,

$$\ln\left(\frac{\pi_j}{\pi_r}\right) = \beta_{j0} + \beta_{j1}X_{j1} + \beta_{j2}X_{j2} + \cdots + \beta_{jp}X_{jp}, \quad j = 1, \dots, k-1,$$

where  $\pi$  stands for a categorical probability, and  $r$  corresponds to the reference category,  $k$  is the total number of response categories,  $p$  is the number of predictor variables. `mnrfit` uses the last category as the reference category for nominal models.

**Example:** `'link','loglog'`

### 'type' - Type of probabilities or counts to estimate

`'category'` (default) | `'cumulative'` | `'conditional'`

Type of probabilities or counts to estimate, specified as the comma-separated pair including `'type'` and one of the following.

<code>'category'</code>	Default. Specify to return predictions and error bounds for the probabilities (or counts) of the $k$ multinomial categories.
<code>'cumulative'</code>	Specify to return predictions and confidence bounds for the cumulative probabilities (or counts) of the first $k - 1$ multinomial categories, as an $n$ -by- $(k - 1)$ matrix. The predicted cumulative probability for the $k$ th category is always 1.
<code>'conditional'</code>	Specify to return predictions and error bounds in terms of the first $k - 1$ conditional category probabilities (counts), i.e., the probability (count) for category $j$ , given an outcome in category $j$ or higher. When <code>'type'</code> is <code>'conditional'</code> , and you supply the sample size argument <code>ssize</code> , the predicted counts at each row of $X$ are conditioned on the corresponding element of <code>ssize</code> , across all categories.

**Example:** `'type','cumulative'`

## Output Arguments

### 'confidence' - Confidence level

0.95 (default) | Scalar value in the range (0,1)

Confidence level for the error bounds, specified as the comma-separated pair consisting of 'confidence' and a scalar value in the range (0,1).

For example, for 99% error bounds, you can specify the confidence as follows:

**Example:** 'confidence',0.99

### Data Types

single | double

### pihat - Probability estimates

$n$ -by- $(k - 1)$  matrix

Probability estimates for each multinomial category, returned as an  $n$ -by- $(k - 1)$  matrix, where  $n$  is the number of observations, and  $k$  is the number of response categories.

### yhat - Count estimates

$n$ -by- $k - 1$  matrix

Count estimates for the number in each response category, returned as an  $n$ -by- $k - 1$  matrix, where  $n$  is the number of observations, and  $k$  is the number of response categories.

### dlow - Lower error bound

column vector

Lower error bound to compute the lower confidence bound for pihat or yhat, returned as a column vector.

The lower confidence bound for pihat is pihat minus dlow. Similarly, the lower confidence bound for yhat is yhat minus dlow. Confidence bounds are nonsimultaneous and only apply to the fitted curve, not to new observations.

### dhi - Upper error bound

column vector



Upper error bound to compute the upper confidence bound for `pihat` or `yhat`, returned as a column vector.

The upper confidence bound for `pihat` is `pihat` plus `dhi`. Similarly, the upper confidence bound for `yhat` is `yhat` plus `dhi`. Confidence bounds are nonsimultaneous and only apply to the fitted curve, not to new observations.

## Examples

### Estimate Category Probabilities for Nominal Responses

Fit a multinomial regression for nominal outcomes and estimate the category probabilities.

Load the sample data.

```
load('fisheriris.mat')
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Define the nominal response variable.

```
sp = nominal(species);  
sp = double(sp);
```

Now in `sp`, 1, 2, and 3 indicate the species `setosa`, `versicolor`, and `virginica`, respectively.

Fit a nominal model to estimate the species using the flower measurements as the predictor variables.

```
[B,dev,stats] = mnrfits(meas,sp);
```

Estimate the probability of being a certain kind of species for an iris flower having the measurements (6.2, 3.7, 5.8, 0.2).

```
x = [6.2, 3.7, 5.8, 0.2];
```

```
pihat = mnrvl(B,meas);  
pihat  
  
pihat =  
  
    0.0017    0.9982    0.0001
```

The probability of an iris flower having the measurements (6.2, 3.7, 5.8, 0.2) being a setosa is 0.0017, a versicolor is 0.9982, and a virginica is 0.0001.

## **Estimate Upper and Lower Error Bounds for Probability Estimates of Ordinal Responses**

Fit a multinomial regression model for categorical responses with natural ordering among categories. Then estimate the upper and lower confidence bounds for the category probability estimates.

Load the sample data and define the predictor variables.

```
load('carbig.mat')  
X = [Acceleration Displacement Horsepower Weight];
```

The predictor variables are the acceleration, engine displacement, horsepower, and the weight of the cars. The response variable is miles per gallon (MPG).

Create an ordinal response variable categorizing MPG into four levels from 9 to 48 mpg.

```
miles = ordinal(MPG,{'1','2','3','4'},[],[9,19,29,39,48]);  
miles = double(miles);
```

Now in miles, 1 indicates the cars with miles per gallon from 9 to 19, and 2 indicates the cars with miles per gallon from 20 to 29. Similarly, 3 and 4 indicate the cars with miles per gallon from 30 to 39 and 40 to 48, respectively.

Fit a multinomial regression model for the response variable miles. For an ordinal model, the default 'link' is logit and the default 'interactions' is 'off'.

```
[B,dev,stats] = mnrfits(X,miles,'model','ordinal');
```

Compute the probability estimates and 95% error bounds for probability confidence intervals for miles per gallon of a car with  $x = (12, 113, 110, 2670)$ .

```
x = [12,113,110,2670];
[pihat,dlow,hi] = mnrval(B,x,stats,'model','ordinal');
pihat
```

```
pihat =
```

```
    0.0615    0.8426    0.0932    0.0027
```

Calculate the confidence bounds for the category probability estimates.

```
LL = pihat - dlow;
UL = pihat + hi;
[LL;UL]
```

```
ans =
```

```
    0.0073    0.7829    0.0283   -0.0003
    0.1157    0.9022    0.1580    0.0057
```

### **Estimate Category Counts and Error Bounds for Nominal Responses**

Fit a multinomial regression for nominal outcomes and estimate the category counts.

Load the sample data.

```
load('fisheriris.mat')
```

The column vector, `species`, consists of iris flowers of three different species, `setosa`, `versicolor`, and `virginica`. The double matrix `meas` consists of four types of measurements on the flowers, the length and width of sepals and petals in centimeters, respectively.

Define the nominal response variable.

```
sp = nominal(species);  
sp = double(sp);
```

Now in `sp`, 1, 2, and 3 indicate the species `setosa`, `versicolor`, and `virginica`, respectively.

Fit a nominal model to estimate the species based on the flower measurements.

```
[B,dev,stats] = mnrfit(meas,sp);
```

Estimate the number in each species category for a sample of 100 iris flowers all with the measurements (6.2, 3.7, 5.8, 0.2).

```
x = [6.2,3.7,5.8,0.2];  
yhat = mnrvl(B,x,18)
```

```
yhat =
```

```
    0.0314    17.9671    0.0016
```

Estimate the error bounds for the counts.

```
[yhat,dlow,hi] = mnrvl(B,x,18,stats,'model','nominal');
```

Calculate the confidence bounds for the category probability estimates.

```
LL = yhat - dlow;  
UL = yhat + hi;  
[LL;UL]
```

```
ans =
```

```
-0.7084    17.2272   -0.0115
 0.7711    18.7069    0.0146
```

### Plot the Count Estimates

Create sample data with one predictor variable and a categorical response variable with three categories.

```
x = [-3 -2 -1 0 1 2 3]';
Y = [1 11 13; 2 9 14; 6 14 5; 5 10 10; ...
     5 14 6; 7 13 5; 8 11 6];
[Y x]
```

```
ans =
```

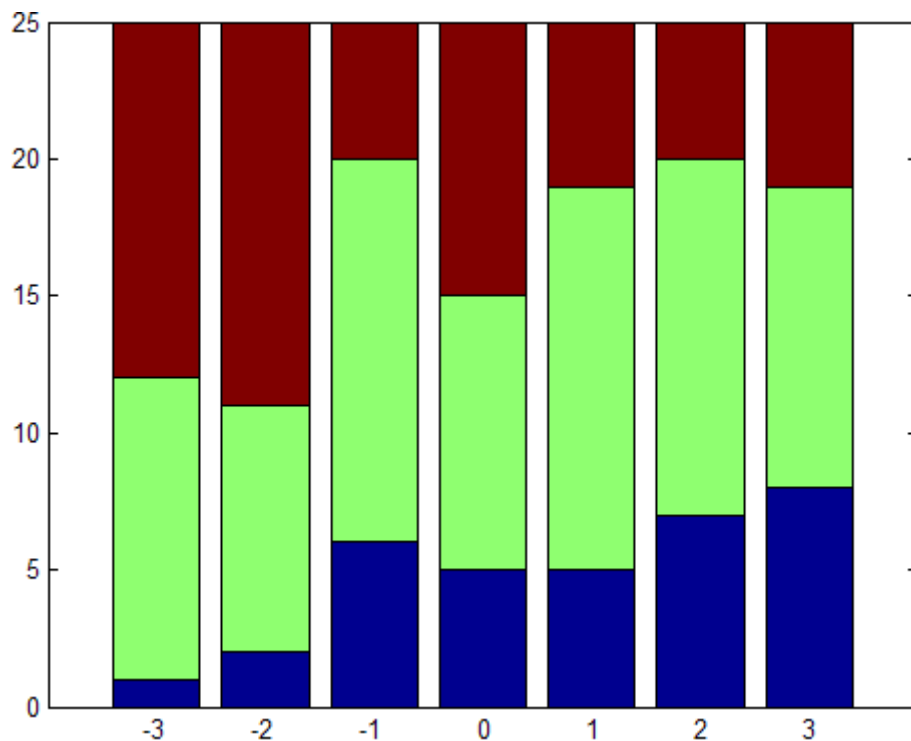
```
 1    11    13    -3
 2     9    14    -2
 6    14     5    -1
 5    10    10     0
 5    14     6     1
 7    13     5     2
 8    11     6     3
```

There are observations on seven different values of the predictor variable  $x$ . The response variable  $Y$  has three categories and the data shows how many of the 25 individuals are in each category of  $Y$  for each observation of  $x$ . For example, when  $x$  is  $-3$ , 1 of 25 individuals is observed in category 1, 11 observed in category 2, and 13 observed in category 3. Similarly, when  $x$  is 1, 5 of the individuals are observed in category 1, 14 are observed in category 2, and 6 are observed in category 3.

Plot the number in each category versus the  $x$  values, on a stacked bar graph.

```
bar(x,Y,'stacked');
ylim([0 25]);
```

# mnrval



Fit a nominal model for the individual response category probabilities, with separate slopes on the single predictor variable,  $x$ , for each category.

```
betaHatNom = mnrfity(x,Y,'model','nominal',...  
                    'interactions','on')
```

```
betaHatNom =
```

```
-0.6028    0.3832  
 0.4068    0.1948
```

The first row of `betaHatNom` contains the intercept terms for the first two response categories. The second row contains the slopes. `mnrfit` accepts the third category as the reference category and hence assumes the coefficients for the third category are zero.

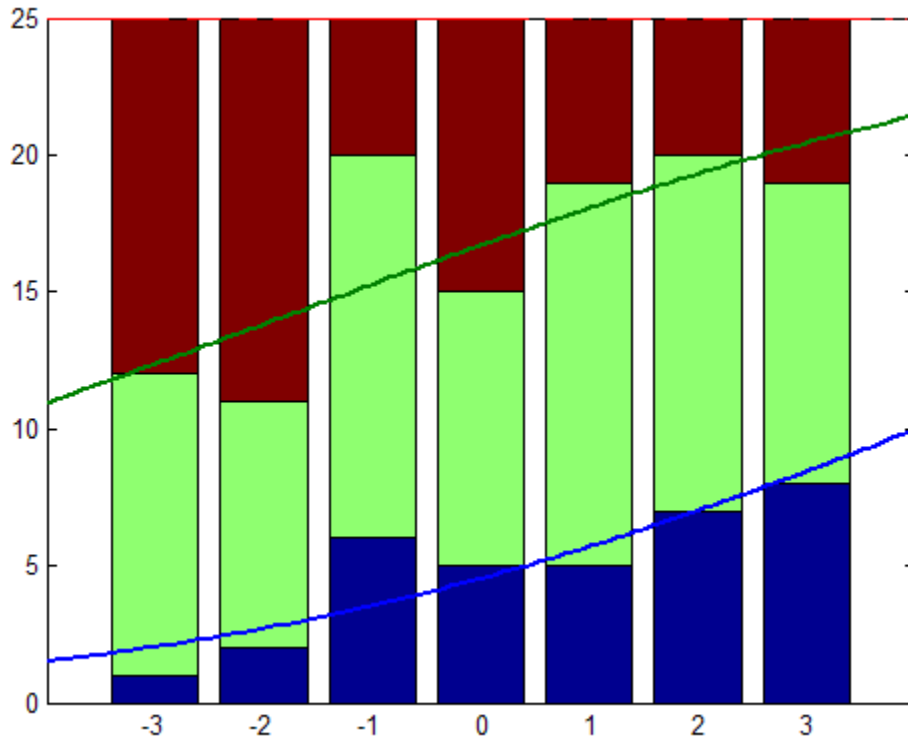
Compute the predicted probabilities for the three response categories.

```
xx = linspace(-4,4)';  
piHatNom = mnrfval(betaHatNom,xx,'model','nominal',...  
    'interactions','on');
```

The probability of being in the third category is simply  $1 - P(y = 1) - P(y = 2)$ .

Plot the estimated cumulative number in each category on the bar graph.

```
line(xx,cumsum(25*piHatNom,2),'LineWidth',2);
```



The cumulative probability for the third category is always 1.

Now, fit a "parallel" ordinal model for the cumulative response category probabilities, with a common slope on the single predictor variable,  $x$ , across all categories:

```
betaHatOrd = mnrfits(x,Y,'model','ordinal',...  
                    'interactions','off')
```

```
betaHatOrd =
```



```
-1.5001  
0.7266  
0.2642
```

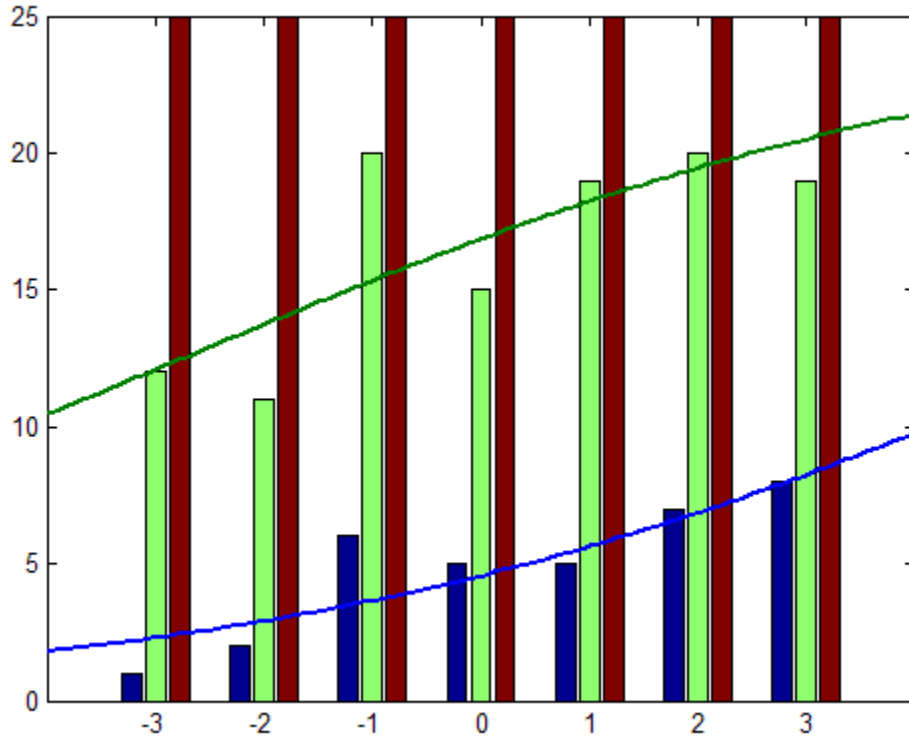
The first two elements of `betaHatOrd` are the intercept terms for the first two response categories. The last element of `betaHatOrd` is the common slope.

Compute the predicted cumulative probabilities for the first two response categories. The cumulative probability for the third category is always 1.

```
piHatOrd = mnrvl(betaHatOrd,xx,'type','cumulative',...  
                'model','ordinal','interactions','off');
```

Plot the estimated cumulative number on the bar graph of the observed cumulative number.

```
figure()  
bar(x,cumsum(Y,2),'grouped');  
ylim([0 25]);  
line(xx,25*piHatOrd,'LineWidth',2);
```



## References

[1] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## See Also

`fitglm` | `mnrfit` | `glmfit` | `glmval`

**Concepts**

- “Multinomial Models for Nominal Responses” on page 10-2
- “Multinomial Models for Ordinal Responses” on page 10-5
- “Hierarchical Multinomial Models” on page 10-9

# moment

---

**Purpose** Central moments

**Syntax** `m = moment(X,order)`  
`moment(X,order,dim)`

**Description** `m = moment(X,order)` returns the central sample moment of  $X$  specified by the positive integer order. For vectors, `moment(x,order)` returns the central moment of the specified order for the elements of  $x$ . For matrices, `moment(X,order)` returns central moment of the specified order for each column. For  $N$ -dimensional arrays, `moment` operates along the first nonsingleton dimension of  $X$ .

`moment(X,order,dim)` takes the moment along dimension `dim` of  $X$ .

**Tips** Note that the central first moment is zero, and the second central moment is the variance computed using a divisor of  $n$  rather than  $n - 1$ , where  $n$  is the length of the vector  $x$  or the number of rows in the matrix  $X$ .

The central moment of order  $k$  of a distribution is defined as

$$m_k = E(x - \mu)^k$$

where  $E(x)$  is the expected value of  $x$ .

## Examples

```
X = randn([6 5])
X =
    1.1650    0.0591    1.2460   -1.2704   -0.0562
    0.6268    1.7971   -0.6390    0.9846    0.5135
    0.0751    0.2641    0.5774   -0.0449    0.3967
    0.3516    0.8717   -0.3600   -0.7989    0.7562
   -0.6965   -1.4462   -0.1356   -0.7652    0.4005
    1.6961   -0.7012   -1.3493    0.8617   -1.3414

m = moment(X,3)
m =
   -0.0282    0.0571    0.1253    0.1460   -0.4486
```

**See Also**

kurtosis | mean | skewness | std | var

# gmdistribution.mu property

---

**Purpose**            Input matrix of means  $\mu$

**Description**     Input matrix of means  $\mu$ .

**Purpose** Multiple comparison test

**Syntax**

```
c = multcompare(stats)
c = multcompare(stats,param1,va11,param2,va12,...)
[c,m] = multcompare(...)
[c,m,h] = multcompare(...)
[c,m,h,gnames] = multcompare(...)
```

**Description** `c = multcompare(stats)` performs a multiple comparison test using the information in the `stats` structure, and returns a matrix `c` of pairwise comparison results. It also displays an interactive graph of the estimates with comparison intervals around them. See “Examples” on page 21-2004.

In a one-way analysis of variance, you compare the means of several groups to test the hypothesis that they are all the same, against the general alternative that they are not all the same. Sometimes this alternative may be too general. You may need information about which pairs of means are significantly different, and which are not. A test that can provide such information is called a *multiple comparison procedure*.

When you perform a simple t-test of one group mean against another, you specify a significance level that determines the cutoff value of the t statistic. For example, you can specify the value `alpha = 0.05` to insure that when there is no real difference, you will incorrectly find a significant difference no more than 5% of the time. When there are many group means, there are also many pairs to compare. If you applied an ordinary t-test in this situation, the alpha value would apply to each comparison, so the chance of incorrectly finding a significant difference would increase with the number of comparisons. Multiple comparison procedures are designed to provide an upper bound on the probability that *any* comparison will be incorrectly found significant.

The output `c` contains the results of the test in the form of a five-column matrix. Each row of the matrix represents one test, and there is one row for each pair of groups. The entries in the row indicate the means being compared, the estimated difference in means, and a confidence interval for the difference.

# multcompare

---

For example, suppose one row contains the following entries.

2.0000 5.0000 1.9442 8.2206 14.4971

These numbers indicate that the mean of group 2 minus the mean of group 5 is estimated to be 8.2206, and a 95% confidence interval for the true difference of the means is [1.9442, 14.4971].

In this example the confidence interval does not contain 0.0, so the difference is significant at the 0.05 level. If the confidence interval did contain 0.0, the difference would not be significant at the 0.05 level.

The `multcompare` function also displays a graph with each group mean represented by a symbol and an interval around the symbol. Two means are significantly different if their intervals are disjoint, and are not significantly different if their intervals overlap. You can use the mouse to select any group, and the graph will highlight any other groups that are significantly different from it.

`c = multcompare(stats,param1,va11,param2,va12,...)` specifies one or more of the parameter name/value pairs described in the following table.

Parameter	Values
'alpha'	Scalar between 0 and 1 that determines the confidence levels of the intervals in the matrix <code>c</code> and in the figure (default is 0.05). The confidence level is $100(1 - \alpha)\%$ .
'display'	Either 'on' (the default) to display a graph of the estimates with comparison intervals around them, or 'off' to omit the graph. See “Examples” on page 21-2004.
'ctype'	Specifies the type of critical value to use for the multiple comparison. “Values of <code>ctype</code> ” on page 21-2002 describes the allowed values for <code>ctype</code> .



Parameter	Values
'dimension'	A vector specifying the dimension or dimensions over which the population marginal means are to be calculated. Use only if you create <b>stats</b> with the function <b>anovan</b> . The default is 1 to compute over the first dimension. See “Dimension Parameter” on page 21-2004 for more information.
'estimate'	Specifies the estimate to be compared. The allowable values of estimate depend on the function that was the source of the <b>stats</b> structure, as described in “Values of estimate” on page 21-2003

`[c,m] = multcompare(...)` returns an additional matrix **m**. The first column of **m** contains the estimated values of the means (or whatever statistics are being compared) for each group, and the second column contains their standard errors.

`[c,m,h] = multcompare(...)` returns a handle **h** to the comparison graph. Note that the title of this graph contains instructions for interacting with the graph, and the *x*-axis label contains information about which means are significantly different from the selected mean. If you plan to use this graph for presentation, you may want to omit the title and the *x*-axis label. You can remove them using interactive features of the graph window, or you can use the following commands.

```
title('')
xlabel('')
```

`[c,m,h,gnames] = multcompare(...)` returns **gnames**, a cell array with one row for each group, containing the names of the groups.

The intervals shown in the graph are computed so that to a very close approximation, two estimates being compared are significantly different if their intervals are disjoint, and are not significantly different if their intervals overlap. (This is exact for multiple comparison of means from

anova1, if all means are based on the same sample size.) You can click on any estimate to see which means are significantly different from it.

## Values of ctype

The following table describes the allowed values for the parameter ctype.

Value	Description
'hsd' or 'tukey-kramer'	Use Tukey's honestly significant difference criterion. This is the default, and it is based on the Studentized range distribution. It is optimal for balanced one-way ANOVA and similar procedures with equal sample sizes. It has been proven to be conservative for one-way ANOVA with different sample sizes. According to the unproven Tukey-Kramer conjecture, it is also accurate for problems where the quantities being compared are correlated, as in analysis of covariance with unbalanced covariate values.
'lsd'	Use Tukey's least significant difference procedure. This procedure is a simple t-test. It is reasonable if the preliminary test (say, the one-way ANOVA $F$ statistic) shows a significant difference. If it is used unconditionally, it provides no protection against multiple comparisons.
'bonferroni'	Use critical values from the t distribution, after a Bonferroni adjustment to compensate for multiple comparisons. This procedure is conservative, but usually less so than the Scheffé procedure.

Value	Description
'dunn-sidak'	Use critical values from the t distribution, after an adjustment for multiple comparisons that was proposed by Dunn and proved accurate by Sidák. This procedure is similar to, but less conservative than, the Bonferroni procedure.
'scheffe'	Use critical values from Scheffé's S procedure, derived from the F distribution. This procedure provides a simultaneous confidence level for comparisons of all linear combinations of the means, and it is conservative for comparisons of simple differences of pairs.

### Values of estimate

The allowable values of the parameter 'estimate' depend on the function that was the source of the stats structure, according to the following table.

Source	Values
'anova1'	Ignored. Always compare the group means.
'anova2'	Either 'column' (the default) or 'row' to compare column or row means.
'anovan'	Ignored. Always compare the population marginal means as specified by the dim argument.
'aoctool'	Either 'slope', 'intercept', or 'pmm' to compare slopes, intercepts, or population marginal means. If the analysis of covariance model did not include separate slopes, then 'slope' is not allowed. If it did not include separate intercepts, then no comparisons are possible.

Source	Values
'friedman'	Ignored. Always compare average column ranks.
'kruskalwallis'	Ignored. Always compare average group ranks.

## Dimension Parameter

The dimension parameter is a vector specifying the dimension or dimensions over which the population marginal means are to be calculated. For example, if `dim = 1`, the estimates that are compared are the means for each value of the first grouping variable, adjusted by removing effects of the other grouping variables as if the design were balanced. If `dim = [1 3]`, population marginal means are computed for each combination of the first and third grouping variables, removing effects of the second grouping variable. If you fit a singular model, some cell means may not be estimable and any population marginal means that depend on those cell means will have the value `NaN`.

Population marginal means are described by Milliken and Johnson (1992) and by Searle, Speed, and Milliken (1980). The idea behind population marginal means is to remove any effect of an unbalanced design by fixing the values of the factors specified by `dim`, and averaging out the effects of other factors as if each factor combination occurred the same number of times. The definition of population marginal means does not depend on the number of observations at each factor combination. For designed experiments where the number of observations at each factor combination has no meaning, population marginal means can be easier to interpret than simple means ignoring other factors. For surveys and other studies where the number of observations at each combination does have meaning, population marginal means may be harder to interpret.

## Examples

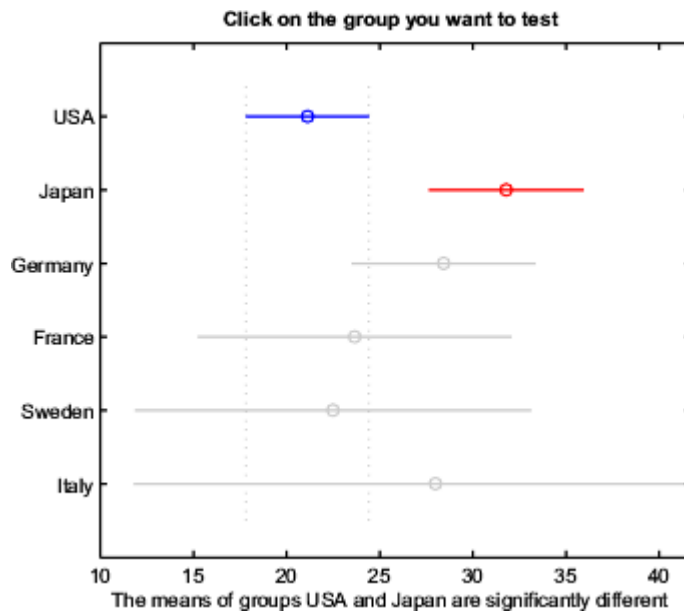
### Example 1

The following example performs a 1-way analysis of variance (ANOVA) and displays group means with their names.

```
load carsmall
```

```
[p,t,st] = anova1(MPG,Origin,'off');
[c,m,h,nms] = multcompare(st,'display','off');
[nms num2cell(m)]
ans =
    'USA'      [21.1328]  [0.8814]
    'Japan'    [31.8000]  [1.8206]
    'Germany'  [28.4444]  [2.3504]
    'France'   [23.6667]  [4.0711]
    'Sweden'   [22.5000]  [4.9860]
    'Italy'    [28.0000]  [7.0513]
```

multcompare also displays the following graph of the estimates with comparison intervals around them.



You can click the graphs of each country to compare its mean to those of other countries.

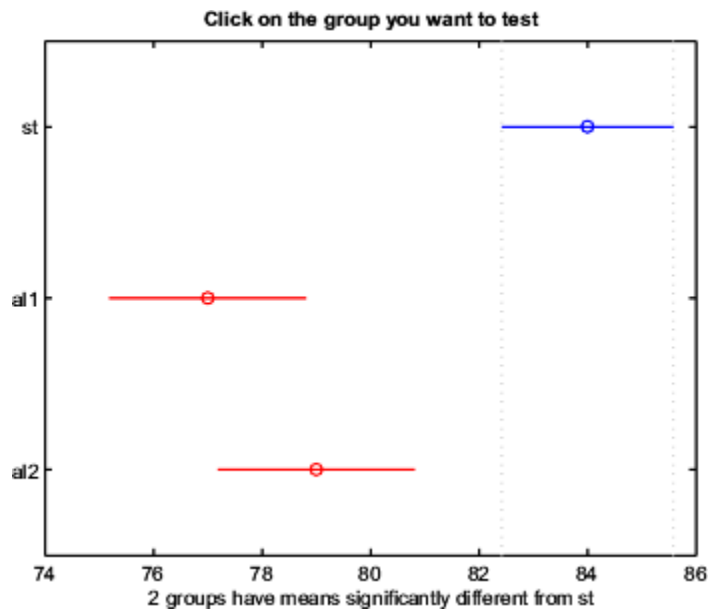
## Example 2

The following continues the example described in the `anova1` reference page, which is related to testing the material strength in structural beams. From the `anova1` output you found significant evidence that the three types of beams are not equivalent in strength. Now you can determine where those differences lie. First you create the data arrays and you perform one-way ANOVA.

```
strength = [82 86 79 83 84 85 86 87 74 82 ...  
            78 75 76 77 79 79 77 78 82 79];  
alloy = {'st','st','st','st','st','st','st','st',...  
         'al1','al1','al1','al1','al1','al1',...  
         'al2','al2','al2','al2','al2','al2'};  
[p,a,s] = anova1(strength,alloy);
```

Among the outputs is a structure that you can use as input to `multcompare`.

```
[c,m,h,nms] = multcompare(s);  
[nms num2cell(c)]  
ans =  
    'st'    [1]    [2]    [ 3.6064]    [ 7]    [10.3936]  
    'al1'   [1]    [3]    [ 1.6064]    [ 5]    [ 8.3936]  
    'al2'   [2]    [3]   [-5.6280]   [-2]    [ 1.6280]
```



The third row of the output matrix shows that the differences in strength between the two alloys is not significant. A 95% confidence interval for the difference is  $[-5.6, 1.6]$ , so you cannot reject the hypothesis that the true difference is zero.

The first two rows show that both comparisons involving the first group (steel) have confidence intervals that do not include zero. In other words, those differences are significant. The graph shows the same information.

## References

- [1] Hochberg, Y., and A. C. Tamhane. *Multiple Comparison Procedures*. Hoboken, NJ: John Wiley & Sons, 1987.
- [2] Milliken, G. A., and D. E. Johnson. *Analysis of Messy Data, Volume 1: Designed Experiments*. Boca Raton, FL: Chapman & Hall/CRC Press, 1992.

# multcompare

---

[3] Searle, S. R., F. M. Speed, and G. A. Milliken. "Population marginal means in the linear model: an alternative to least-squares means." *American Statistician*. 1980, pp. 216–221.

## See Also

[anova1](#) | [anova2](#) | [anovan](#) | [aoctool](#) | [friedman](#) | [kruskalwallis](#)



**Superclasses** ParametricTruncatableDistribution

**Purpose** Multinomial probability distribution object

**Description** `prob.MultinomialDistribution` is an object consisting of parameters and a model description for a multinomial probability distribution. Create a probability distribution object with specified parameters using `makedist`.

**Construction** `pd = makedist('Multinomial')` creates a multinomial probability distribution object using the default parameter values.

`pd = makedist('Multinomial','Probabilities',probabilities)` creates a multinomial distribution object using the specified parameter value.

## Input Arguments

### **probabilities - outcome probabilities**

[0.500 0.500] (default) | vector of scalar values in the range [0,1]

Outcome probabilities, specified as a vector of scalar values in the range [0,1]. Each vector element is the probability that a multinomial trial has a particular corresponding outcome. The values in `probabilities` must sum to 1.

### **Data Types**

single | double

## Properties

### **probabilities**

Outcome probabilities for the multinomial distribution, stored as a vector of scalar values in the range [0,1]. The values in `probabilities` must sum to 1.

### **Data Types**

single | double

### **DistributionName**

# prob.MultinomialDistribution

---

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated. If IsTruncated equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object

# prob.MultinomialDistribution

---

mean	Mean of probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions **Multinomial Distribution**

The multinomial distribution is a generalization of the binomial distribution. While the binomial distribution gives the probability of the number of “successes” in  $n$  independent trials of a two-outcome process, the multinomial distribution gives the probability of each combination of outcomes in  $n$  independent trials of a  $k$ -outcome process. The probability of each outcome in any one trial is given by the fixed probabilities  $p_1, \dots, p_k$ .

The multinomial distribution uses the following parameters.

Parameter	Description	Support
probabilities	Outcome probabilities	$0 \leq \text{probabilities}(i) \leq 1; \sum_{\text{all}(i)} \text{probabilities}(i) = 1$

The probability density function (pdf) is

$$f(x | n, p) = \frac{n!}{x_1! \dots x_k!} p_1^{x_1} \dots p_k^{x_k} \quad ; \quad \sum_1^k x_i = n, \sum_1^k p_i = 1,$$

where  $x = (x_1, \dots, x_k)$  gives the number of each  $k$  outcome in  $n$  trials of a process with fixed probabilities  $p = (p_1, \dots, p_k)$  of individual outcomes in any one trial.

## Examples

### Create a Multinomial Distribution Object Using Default Parameters

Create a multinomial distribution object using the default parameter values.

```
pd = makedist('Multinomial')
```

```
pd =
```

```
    MultinomialDistribution
```

```
    Probabilities:
```

```
    0.5000    0.5000
```

### Create a Multinomial Distribution Object Using Specified Parameters

Create a multinomial distribution object for a distribution with three possible outcomes. Outcome 1 has a probability of 1/2, outcome 2 has a probability of 1/3, and outcome 3 has a probability of 1/6.

```
pd = makedist('Multinomial','probabilities',[1/2 1/3 1/6])
```

```
pd =
```

```
    MultinomialDistribution
```

```
    Probabilities:
```

```
    0.5000    0.3333    0.1667
```

Generate a random outcome from the distribution.

```
rng('default'); % for reproducibility
```

```
r = random(pd)
```

```
r =
```

```
    2
```

# prob.MultinomialDistribution

---

The result of this trial is outcome 2. By default, the number of trials in each experiment,  $n$ , equals 1.

Generate random outcomes from the distribution when the number of trials in each experiment,  $n$ , equals 1, and the experiment is repeated ten times.

```
rng('default'); % for reproducibility
r = random(pd,10,1)
```

```
r =
```

```
2
3
1
3
2
1
1
2
3
3
```

Each element in the array is the outcome of an individual experiment that contains one trial.

Generate random outcomes from the distribution when the number of trials in each experiment,  $n$ , equals 5, and the experiment is repeated ten times.

```
rng('default'); % for reproducibility
r = random(pd,10,5)
```

```
r =
```

```
2    1    2    2    1
3    3    1    1    1
```

1	3	3	1	2
3	1	3	1	2
2	2	2	1	1
1	1	2	2	1
1	1	2	2	1
2	3	1	1	2
3	2	2	3	2
3	3	1	1	2

Each element in the resulting matrix is the outcome of one trial. The columns correspond to the five trials in each experiment, and the rows correspond to the ten experiments. For example, in the first experiment (corresponding to the first row), 2 of the 5 trials resulted in outcome 1, and 3 of the 5 trials resulted in outcome 2.

**See Also** `makedist`

**Concepts**

- “Multinomial Distribution” on page B-83
- Class Attributes
- Property Attributes

# multivarichart

---

**Purpose** Multivari chart for grouped data

**Syntax**  
`multivarichart(y, GROUP)`  
`multivarichart(Y)`  
`multivarichart(..., param1, val1, param2, val2, ...)`  
`[charthandle, AXESH] = multivarichart(...)`

**Description** `multivarichart(y, GROUP)` displays the multivari chart for the vector `y` grouped by entries in the cell array `GROUP`. Each cell of `GROUP` must contain a grouping variable that can be a categorical variable, numeric vector, character matrix, or single-column cell array of strings. `GROUP` can also be a matrix whose columns represent different grouping variables. Each grouping variable must have the same number of elements as `y`. The number of grouping variables must be 2, 3, or 4.

Each subplot of the plot matrix contains a multivari chart for the first and second grouping variables. The *x*-axis in each subplot indicates values of the first grouping variable. The legend at the bottom of the figure window indicates values of the second grouping variable. The subplot at position (*i*,*j*) is the multivari chart for the subset of `y` at the *i*th level of the third grouping variable and the *j*th level of the fourth grouping variable. If the third or fourth grouping variable is absent, it is considered to have only one level.

`multivarichart(Y)` displays the multivari chart for a matrix `Y`. The data in different columns represent changes in one factor. The data in different rows represent changes in another factor.

`multivarichart(..., param1, val1, param2, val2, ...)` specifies one or more of the following name/value pairs:

- 'varnames' — Grouping variable names in a character matrix or a cell array of strings, one per grouping variable. Default names are 'X1', 'X2', ... .
- 'plotorder' — A string with the value 'sorted' or a vector containing a permutation of the integers from 1 to the number of grouping variables.



If 'plotorder' is a string with value 'sorted', the grouping variables are rearranged in descending order according to the number of levels in each variable.

If 'plotorder' is a vector, it indicates the order in which each grouping variable should be plotted. For example, [2,3,1,4] indicates that the second grouping variable should be used as the x-axis of each subplot, the third grouping variable should be used as the legend, the first grouping variable should be used as the columns of the plot, and the fourth grouping variable should be used as the rows of the plot.

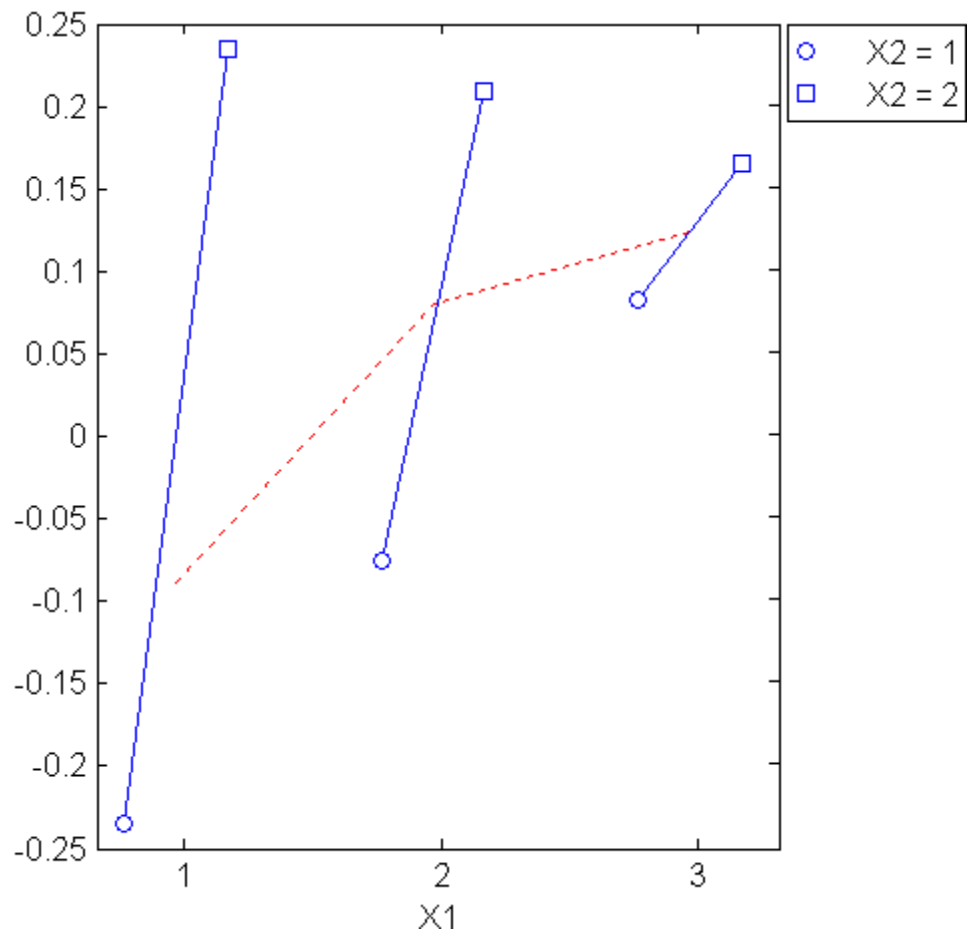
[charthandle,AXESH] = multivarichart(...) returns a handle charthandle to the figure window and a matrix AXESH of handles to the subplot axes.

## Examples

Display a multivari chart for data with two grouping variables:

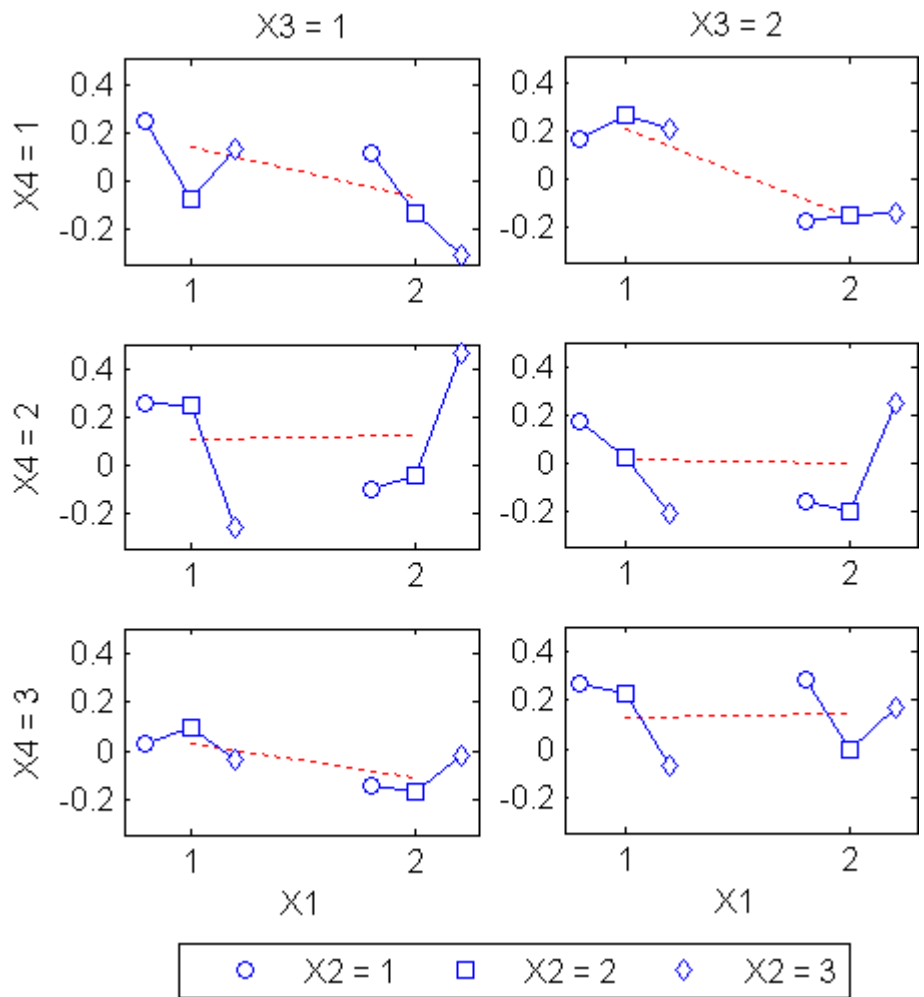
```
y = randn(100,1); % response
group = [ceil(3*rand(100,1)) ceil(2*rand(100,1))];
multivarichart(y,group)
```

# multivarichart



Display a multivari chart for data with four grouping variables:

```
y = randn(1000,1); % response
group = {ceil(2*rand(1000,1)),ceil(3*rand(1000,1)), ...
        ceil(2*rand(1000,1)),ceil(3*rand(1000,1))};
multivarichart(y,group)
```



**See Also**

maineffectsplot | interactionplot

**How To**

- “Grouping Variables” on page 2-51

# mvncdf

---

**Purpose** Multivariate normal cumulative distribution function

**Syntax**

```
y = mvncdf(X)
y = mvncdf(X,mu,SIGMA)
y = mvncdf(xl,xu,mu,SIGMA)
[y,err] = mvncdf(...)
[...] = mvncdf(...,options)
```

**Description** `y = mvncdf(X)` returns the cumulative probability of the multivariate normal distribution with zero mean and identity covariance matrix, evaluated at each row of  $X$ . Rows of the  $n$ -by- $d$  matrix  $X$  correspond to observations or points, and columns correspond to variables or coordinates.  $y$  is an  $n$ -by-1 vector.

`y = mvncdf(X,mu,SIGMA)` returns the cumulative probability of the multivariate normal distribution with mean  $\mu$  and covariance  $\text{SIGMA}$ , evaluated at each row of  $X$ .  $\mu$  is a 1-by- $d$  vector, and  $\text{SIGMA}$  is a  $d$ -by- $d$  symmetric, positive definite matrix.  $\mu$  can also be a scalar value, which `mvncdf` replicates to match the size of  $X$ . If the covariance matrix is diagonal, containing variances along the diagonal and zero covariances off the diagonal,  $\text{SIGMA}$  may also be specified as a 1-by- $d$  vector containing just the diagonal. Pass in the empty matrix `[]` for  $\mu$  to use as its default value when you want to only specify  $\text{SIGMA}$ .

The multivariate normal cumulative probability at  $X$  is defined as the probability that a random vector  $V$ , distributed as multivariate normal, will fall within the semi-infinite rectangle with upper limits defined by  $X$ , for example,  $\Pr\{V(1) \leq X(1), V(2) \leq X(2), \dots, V(d) \leq X(d)\}$ .

`y = mvncdf(xl,xu,mu,SIGMA)` returns the multivariate normal cumulative probability evaluated over the rectangle with lower and upper limits defined by  $x_l$  and  $x_u$ , respectively.

`[y,err] = mvncdf(...)` returns an estimate of the error in  $y$ . For bivariate and trivariate distributions, `mvncdf` uses adaptive quadrature on a transformation of the  $t$  density, based on methods developed by Drezner and Wesolowsky and by Genz, as described in the references. The default absolute error tolerance for these cases is  $1e-8$ . For four or more dimensions, `mvncdf` uses a quasi-Monte Carlo integration

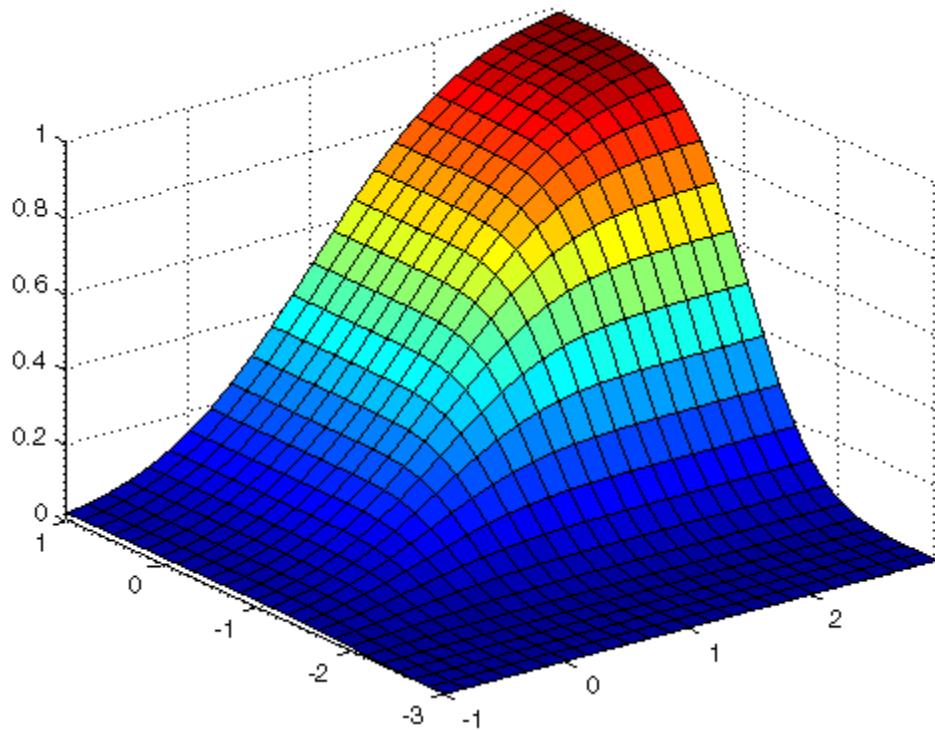
algorithm based on methods developed by Genz and Bretz, as described in the references. The default absolute error tolerance for these cases is  $1e-4$ .

`[...] = mvncdf(...,options)` specifies control parameters for the numerical integration used to compute `y`. This argument can be created by a call to `statset`. Choices of `statset` parameters:

- `'TolFun'` — Maximum absolute error tolerance. Default is  $1e-8$  when  $d < 4$ , or  $1e-4$  when  $d \geq 4$ .
- `'MaxFunEvals'` — Maximum number of integrand evaluations allowed when  $d \geq 4$ . Default is  $1e7$ . `'MaxFunEvals'` is ignored when  $d < 4$ .
- `'Display'` — Level of display output. Choices are `'off'` (the default), `'iter'`, and `'final'`. `'Display'` is ignored when  $d < 4$ .

## Examples

```
mu = [1 -1]; SIGMA = [.9 .4; .4 .3];  
[X1,X2] = meshgrid(linspace(-1,3,25)',linspace(-3,1,25)');  
X = [X1(:) X2(:)];  
p = mvncdf(X,mu,SIGMA);  
surf(X1,X2,reshape(p,25,25));
```



## References

- [1] Drezner, Z. “Computation of the Trivariate Normal Integral.” *Mathematics of Computation*. Vol. 63, 1994, pp. 289–294.
- [2] Drezner, Z., and G. O. Wesolowsky. “On the Computation of the Bivariate Normal Integral.” *Journal of Statistical Computation and Simulation*. Vol. 35, 1989, pp. 101–107.
- [3] Genz, A. “Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities.” *Statistics and Computing*. Vol. 14, No. 3, 2004, pp. 251–260.
- [4] Genz, A., and F. Bretz. “Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple

Contrasts.” *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.

[5] Genz, A., and F. Bretz. “Comparison of Methods for the Computation of Multivariate  $t$  Probabilities.” *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.

**See Also**

mvnpdf | mvnrnd

**How To**

- “Multivariate Normal Distribution” on page B-86

# mvnpdf

---

**Purpose** Multivariate normal probability density function

**Syntax**

```
y = mvnpdf(X)
y = mvnpdf(X,MU)
y = mvnpdf(X,MU,SIGMA)
```

**Description** `y = mvnpdf(X)` returns the  $n$ -by-1 vector `y`, containing the probability density of the multivariate normal distribution with zero mean and identity covariance matrix, evaluated at each row of the  $n$ -by- $d$  matrix `X`. Rows of `X` correspond to observations and columns correspond to variables or coordinates.

`y = mvnpdf(X,MU)` returns the density of the multivariate normal distribution with mean `mu` and identity covariance matrix, evaluated at each row of `X`. `MU` is a 1-by- $d$  vector, or an  $n$ -by- $d$  matrix. If `MU` is a matrix, the density is evaluated for each row of `X` with the corresponding row of `MU`. `MU` can also be a scalar value, which `mvnpdf` replicates to match the size of `X`.

`y = mvnpdf(X,MU,SIGMA)` returns the density of the multivariate normal distribution with mean `MU` and covariance `SIGMA`, evaluated at each row of `X`. `SIGMA` is a  $d$ -by- $d$  matrix, or a  $d$ -by- $d$ -by- $n$  array, in which case the density is evaluated for each row of `X` with the corresponding page of `SIGMA`, i.e., `mvnpdf` computes `y(i)` using `X(i,:)` and `SIGMA(:,:,i)`. If the covariance matrix is diagonal, containing variances along the diagonal and zero covariances off the diagonal, `SIGMA` may also be specified as a 1-by- $d$  vector or a 1-by- $d$ -by- $n$  array, containing just the diagonal. Specify `[]` for `MU` to use its default value when you want to specify only `SIGMA`.

If `X` is a 1-by- $d$  vector, `mvnpdf` replicates it to match the leading dimension of `mu` or the trailing dimension of `SIGMA`.

**Examples**

```
mu = [1 -1];
SIGMA = [.9 .4; .4 .3];
X = mvnrnd(mu,SIGMA,10);
p = mvnpdf(X,mu,SIGMA);
```



**See Also**

[mvncdf](#) | [mvnrnd](#) | [normpdf](#)

**How To**

- “Multivariate Normal Distribution” on page B-86

# mvregress

---

**Purpose** Multivariate linear regression

**Syntax**

```
beta = mvregress(X,Y)
beta = mvregress(X,Y,Name,Value)
[beta,Sigma] = mvregress( ___ )
[beta,Sigma,E,CovB,logL] = mvregress( ___ )
```

**Description** `beta = mvregress(X,Y)` returns the estimated coefficients for a multivariate normal regression of the  $d$ -dimensional responses in  $Y$  on the design matrices in  $X$ .

`beta = mvregress(X,Y,Name,Value)` returns the estimated coefficients using additional options specified by one or more name-value pair arguments. For example, you can specify the estimation algorithm, initial estimate values, or maximum number of iterations for the regression.

`[beta,Sigma] = mvregress( ___ )` also returns the estimated  $d$ -by- $d$  variance-covariance matrix of  $Y$ , using any of the input arguments from the previous syntaxes.

`[beta,Sigma,E,CovB,logL] = mvregress( ___ )` also returns a matrix of residuals  $E$ , estimated variance-covariance matrix of the regression coefficients  $CovB$ , and the value of the log likelihood objective function after the last iteration  $logL$ .

## Input Arguments

**X - Design matrices**  
matrix | cell array of matrices

Design matrices for the multivariate regression, specified as a matrix or cell array of matrices.  $n$  is the number of observations in the data,  $K$  is the number of regression coefficients to estimate,  $p$  is the number of predictor variables, and  $d$  is the number of dimensions in the response variable matrix  $Y$ .

- If  $d = 1$ , then specify  $X$  as a single  $n$ -by- $K$  design matrix.

- If  $d > 1$  and all  $d$  dimensions have the same design matrix, then you can specify  $X$  as a single  $n$ -by- $p$  design matrix (not in a cell array).
- If  $d > 1$  and all  $n$  observations have the same design matrix, then you can specify  $X$  as a cell array containing a single  $d$ -by- $K$  design matrix.
- If  $d > 1$  and all  $n$  observations do not have the same design matrix, then specify  $X$  as a cell array of length  $n$  containing  $d$ -by- $K$  design matrices.

To include a constant term in the regression model, each design matrix should contain a column of ones.

`mvregress` treats NaN values in  $X$  as missing values, and ignores rows in  $X$  with missing values.

### Data Types

single | double | cell

### Y - Response variables

matrix

Response variables, specified as an  $n$ -by- $d$  matrix.  $n$  is the number of observations in the data, and  $d$  is the number of dimensions in the response. When  $d = 1$ , `mvregress` treats the values in  $Y$  like  $n$  independent response values.

`mvregress` treats NaN values in  $Y$  as missing values, and handles them according to the estimation algorithm specified using the name-value pair argument `algorithm`.

### Data Types

single | double

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** 'algorithm', 'cwls', 'covar0', C specifies covariance-weighted least squares estimation using the covariance matrix C.

## 'algorithm' - Estimation algorithm

'mvn' | 'ecm' | 'cwls'

Estimation algorithm, specified as the comma-separated pair consisting of 'algorithm' and one of the following.

'mvn'	Ordinary multivariate normal maximum likelihood estimation.
'ecm'	Maximum likelihood estimation via the ECM algorithm.
'cwls'	Covariance-weighted least squares estimation.

The default algorithm depends on the presence of missing data.

- For complete data, the default is 'mvn'.
- If there are any missing responses (indicated by NaN), the default is 'ecm', provided the sample size is sufficient to estimate all parameters. Otherwise, the default algorithm is 'cwls'.

---

**Note** If algorithm has the value 'mvn', then mvregress removes observations with missing response values before estimation.

---

**Example:** 'algorithm', 'ecm'

## 'beta0' - Initial estimates for regression coefficients

vector

Initial estimates for the regression coefficients, specified as the comma-separated pair consisting of 'beta0' and a vector with  $K$  elements. The default value is a vector of 0s.

The `beta0` argument is not used if the estimation algorithm is `'mvn'`.

### **'covar0' - Initial estimate for variance-covariance matrix**

matrix

Initial estimate for the variance-covariance matrix, `Sigma`, specified as the comma-separated pair consisting of `'covar0'` and a symmetric, positive definite,  $d$ -by- $d$  matrix. The default value is the identity matrix.

If the estimation algorithm is `'cqls'`, then `mvregress` uses `covar0` as the weighting matrix at each iteration, without changing it.

### **'covtype' - Type of variance-covariance matrix**

`'full'` (default) | `'diagonal'`

Type of variance-covariance matrix to estimate for `Y`, specified as the comma-separated pair consisting of `'covtype'` and one of the following.

<code>'full'</code>	Estimate all $d(d + 1)/2$ variance-covariance elements.
<code>'diagonal'</code>	Estimate only the $d$ diagonal elements of the variance-covariance matrix.

**Example:** `'covtype','diagonal'`

### **'maxiter' - Maximum number of iterations**

100 (default) | positive integer

Maximum number of iterations for the estimation algorithm, specified as the comma-separated pair consisting of `'maxiter'` and a positive integer.

Iterations continue until estimates are within the convergence tolerances `tolbeta` and `tolobj`, or the maximum number of iterations specified by `maxiter` is reached. If both `tolbeta` and `tolobj` are 0, then `mvregress` performs `maxiter` iterations with no convergence tests.

**Example:** `'maxiter',50`

## **'outputfcn' - Function to evaluate each iteration**

function handle

Function to evaluate at each iteration, specified as the comma-separated pair consisting of 'outputfcn' and a function handle. The function must return a logical true or false. At each iteration, `mvregress` evaluates the function. If the result is true, iterations stop. Otherwise, iterations continue. For example, you could specify a function that plots or displays current iteration results, and returns true if you close the figure.

The function must accept three input arguments, in this order:

- Vector of current coefficient estimates
- Structure containing these three fields:

<code>Covar</code>	Current value of the variance-covariance matrix
<code>iteration</code>	Current iteration number
<code>fval</code>	Current value of the loglikelihood objective function

- Text string that takes these three values:

<code>'init'</code>	When the function is called during initialization
<code>'iter'</code>	When the function is called after an iteration
<code>'done'</code>	When the function is called after completion

## **'tolbeta' - Convergence tolerance for regression coefficients**

`sqrt(eps)` (default) | positive scalar value

Convergence tolerance for regression coefficients, specified as the comma-separated pair consisting of 'tolbeta' and a positive scalar value.

Let  $\mathbf{b}^t$  denote the estimate of the coefficient vector at iteration  $t$ , and  $\tau_\beta$  be the tolerance specified by `tolbeta`. The convergence criterion for regression coefficient estimation is

$$\|\mathbf{b}^t - \mathbf{b}^{t-1}\| < \tau_\beta \sqrt{K} (1 + \|\mathbf{b}^t\|),$$

where  $K$  is the length of  $\mathbf{b}^t$  and  $\|\mathbf{v}\|$  is the norm of a vector  $\mathbf{v}$ .

Iterations continue until estimates are within the convergence tolerances `tolbeta` and `tolobj`, or the maximum number of iterations specified by `maxiter` is reached. If both `tolbeta` and `tolobj` are 0, then `mvregress` performs `maxiter` iterations with no convergence tests.

**Example:** `'tolbeta', 1e-5`

### **'tolobj' - Convergence tolerance for loglikelihood objective function**

`eps^(3/4)` (default) | positive scalar value

Convergence tolerance for the loglikelihood objective function, specified as the comma-separated pair consisting of `'tolobj'` and a positive scalar value.

Let  $L^t$  denote the value of the loglikelihood objective function at iteration  $t$ , and  $\tau_\ell$  be the tolerance specified by `tolobj`. The convergence criterion for the objective function is

$$|L^t - L^{t-1}| < \tau_\ell (1 + |L^t|).$$

Iterations continue until estimates are within the convergence tolerances `tolbeta` and `tolobj`, or the maximum number of iterations specified by `maxiter` is reached. If both `tolbeta` and `tolobj` are 0, then `mvregress` performs `maxiter` iterations with no convergence tests.

**Example:** `'tolobj', 1e-5`

## **'varformat' - Format for parameter estimate variance-covariance matrix**

'beta' (default) | 'full'

Format for the parameter estimate variance-covariance matrix, `COVB`, specified as the comma-separated pair consisting of 'varformat' and one of the following.

'beta'	Return the variance-covariance matrix for only the regression coefficient estimates, <code>beta</code> .
'full'	Return the variance-covariance matrix for both the regression coefficient estimates, <code>beta</code> , and the variance-covariance matrix estimate, <code>Sigma</code> .

**Example:** 'varformat', 'full'

## **'vartype' - Type of variance-covariance matrix for parameter estimates**

'hessian' (default) | 'fisher'

Type of variance-covariance matrix for parameter estimates, specified as the comma-separated pair consisting of 'vartype' and either 'hessian' or 'fisher'.

- If the value is 'hessian', then `mvregress` uses the Hessian, or observed information, matrix to compute `COVB`.
- If the value is 'fisher', then `mvregress` uses the complete-data Fisher, or expected information, matrix to compute `COVB`.

The 'hessian' method takes into account the increase uncertainties due to missing data, while the 'fisher' method does not.

**Example:** 'vartype', 'fisher'



## Output Arguments

### **beta** - Estimated regression coefficients

column vector | matrix

Estimated regression coefficients, returned as a column vector or matrix.

- If you specify  $X$  as a single  $n$ -by- $K$  design matrix, then `mvregress` returns **beta** as a column vector of length  $K$ . For example, if  $X$  is a 20-by-5 design matrix, then **beta** is a 5-by-1 column vector.
- If you specify  $X$  as a cell array containing one or more  $d$ -by- $K$  design matrices, then `mvregress` returns **beta** as a column vector of length  $K$ . For example, if  $X$  is a cell array containing 2-by-10 design matrices, then **beta** is a 10-by-1 column vector.
- If you specify  $X$  as a single  $n$ -by- $p$  design matrix (not in a cell array), and  $Y$  has dimension  $d > 1$ , then `mvregress` returns **beta** as a  $p$ -by- $d$  matrix. For example, if  $X$  is a 20-by-5 design matrix, and  $Y$  has two dimensions such that  $d = 2$ , then **beta** is a 5-by-2 matrix, and the fitted  $Y$  values are  $X \times \text{beta}$ .

### **Sigma** - Estimated variance-covariance matrix

square matrix

Estimated variance-covariance matrix for the responses in  $Y$ , returned as a  $d$ -by- $d$  square matrix.

---

**Note** The estimated variance-covariance matrix, **Sigma**, is not the sample covariance matrix of the residual matrix, **E**.

---

### **E** - Residuals

matrix

Residuals for the fitted regression model, returned as an  $n$ -by- $d$  matrix.

If `algorithm` has the value 'ecm' or 'cwlsl', then `mvregress` computes the residual values corresponding to missing values in  $Y$  as

the difference between the conditionally imputed values and the fitted values.

---

**Note** If `algorithm` has the value 'mvn', then `mvregress` removes observations with missing response values before estimation.

---

## **CovB - Parameter estimate variance-covariance matrix**

square matrix

Parameter estimate variance-covariance matrix, returned as a square matrix.

- If `varformat` has the value 'beta' (default), then `CovB` is the estimated variance-covariance matrix of the coefficient estimates in `beta`.
- If `varformat` has the value 'full', then `CovB` is the estimated variance-covariance matrix of the combined estimates in `beta` and `Sigma`.

## **logL - Loglikelihood objective function value**

scalar value

Loglikelihood objective function value after the last iteration, returned as a scalar value.

## **Examples**

### **Multivariate Regression Model for Panel Data with Different Intercepts**

Fit a multivariate regression model to panel data, assuming different intercepts and common slopes.

Load the sample data.

```
load('flu')
```

The dataset array `flu` contains national CDC flu estimates, and nine separate regional estimates based on Google query data.

Extract the response and predictor data.

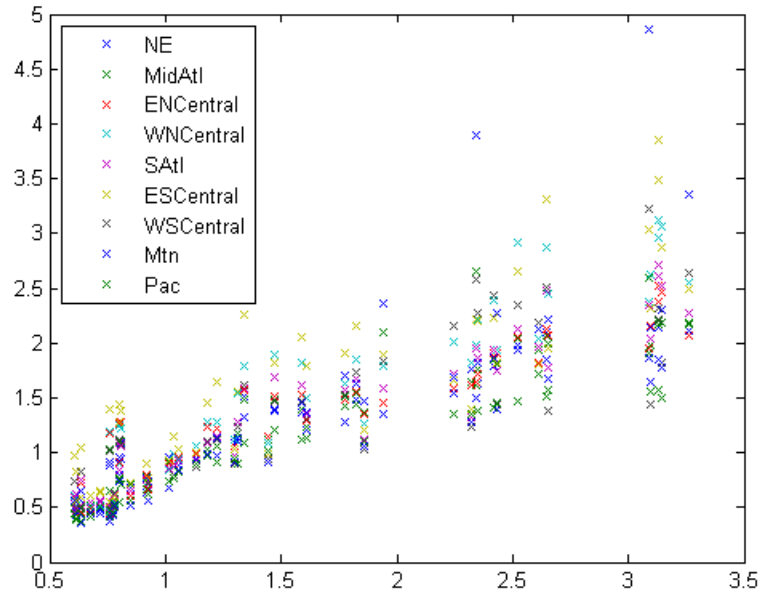
```
Y = double(flu(:,2:end-1));  
[n,d] = size(Y);  
x = flu.WtdILI;
```

The responses in  $Y$  are the nine regional flu estimates. Observations exist for every week over a one-year period, so  $n = 52$ . The dimension of the responses corresponds to the regions, so  $d = 9$ . The predictors in  $x$  are the weekly national flu estimates.

Plot the flu data, grouped by region.

```
figure;  
regions = flu.Properties.VarNames(2:end-1);  
plot(x,Y,'x')  
legend(regions,'Location','NorthWest')
```

# mvregress



Fit the multivariate regression model

$$y_{ij} = \alpha_j + \beta x_{ij} + \varepsilon_{ij}, \quad i = 1, \dots, n; \quad j = 1, \dots, d,$$

with between-region concurrent correlation

$$\text{COV}(\varepsilon_{ij}, \varepsilon_{ij'}) = \sigma_{jj'}, \quad j = 1, \dots, d.$$

There are  $K = 10$  regression coefficients to estimate: nine intercept terms and a common slope. The input argument  $X$  should be an  $n$ -element cell array of  $d$ -by- $K$  design matrices.

```
X = cell(n,1);  
for i=1:n  
    X{i} = [eye(d) repmat(x(i),d,1)];
```

```
end
[beta,Sigma] = mvregress(X,Y);
```

beta contains estimates of the  $K$ -dimensional coefficient vector

$$(\alpha_1, \alpha_2, \dots, \alpha_9, \beta)'$$

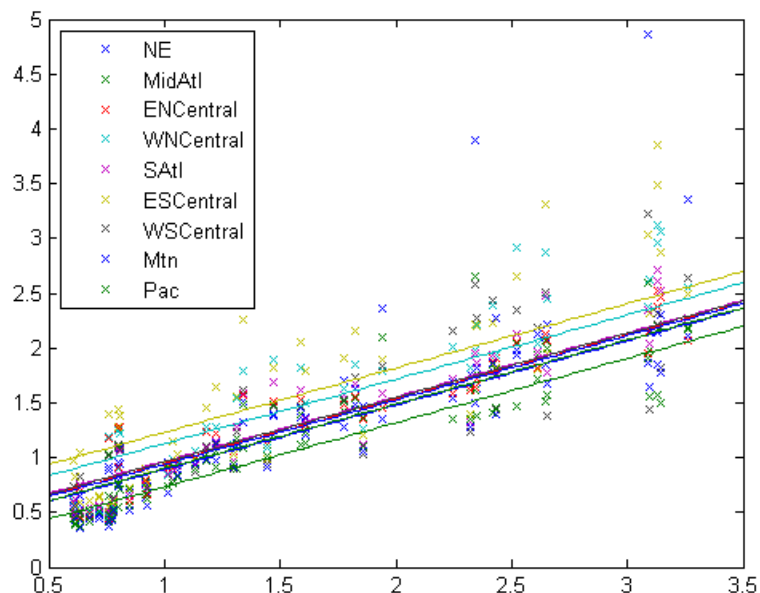
Sigma contains estimates of the  $d$ -by- $d$  variance-covariance matrix for the between-region concurrent correlations

$$\begin{pmatrix} \sigma_{11} & \cdots & \sigma_{1,9} \\ \vdots & \ddots & \vdots \\ \sigma_{9,1} & \cdots & \sigma_{9,9} \end{pmatrix}.$$

Plot the fitted regression model.

```
B = [beta(1:d)'; repmat(beta(end),1,d)];
xx = linspace(.5,3.5)';
fits = [ones(size(xx)),xx]*B;

figure;
h = plot(x,Y,'x',xx,fits,'-');
for i = 1:d
    set(h(d+i),'color',get(h(i),'color'));
end
legend(regions,'Location','NorthWest');
```



The plot shows that each regression line has a different intercept but the same slope. Upon visual inspection, some regression lines appear to fit the data better than others.

## Multivariate Regression for Panel Data with Different Slopes

Fit a multivariate regression model to panel data using least squares, assuming different intercepts and slopes.

Load the sample data.

```
load('flu');
```

The dataset array `flu` contains national CDC flu estimates, and nine separate regional estimates based on Google queries.

Extract the response and predictor data.

```

Y = double(flu(:,2:end-1));
[n,d] = size(Y);
x = flu.WtdILI;

```

The responses in  $Y$  are the nine regional flu estimates. Observations exist for every week over a one-year period, so  $n = 52$ . The dimension of the responses corresponds to the regions, so  $d = 9$ . The predictors in  $x$  are the weekly national flu estimates.

Fit the multivariate regression model

$$y_{ij} = \alpha_j + \beta_j x_{ij} + \varepsilon_{ij}, \quad i = 1, \dots, n; \quad j = 1, \dots, d,$$

with between-region concurrent correlation

$$COV(\varepsilon_{ij}, \varepsilon_{ij'}) = \sigma_{jj'}, \quad j = 1, \dots, d.$$

There are  $K = 18$  regression coefficients to estimate: nine intercept terms, and nine slope terms.  $X$  is an  $n$ -element cell array of  $d$ -by- $K$  design matrices.

```

X = cell(n,1);
for i=1:n
    X{i} = [eye(d) x(i)*eye(d)];
end
[beta,Sigma] = mvregress(X,Y,'algorithm','cwlsl');

```

$\beta$  contains estimates of the  $K$ -dimensional coefficient vector,

$$(\alpha_1, \alpha_2, \dots, \alpha_9, \beta_1, \beta_2, \dots, \beta_9)'$$

Plot the fitted regression model.

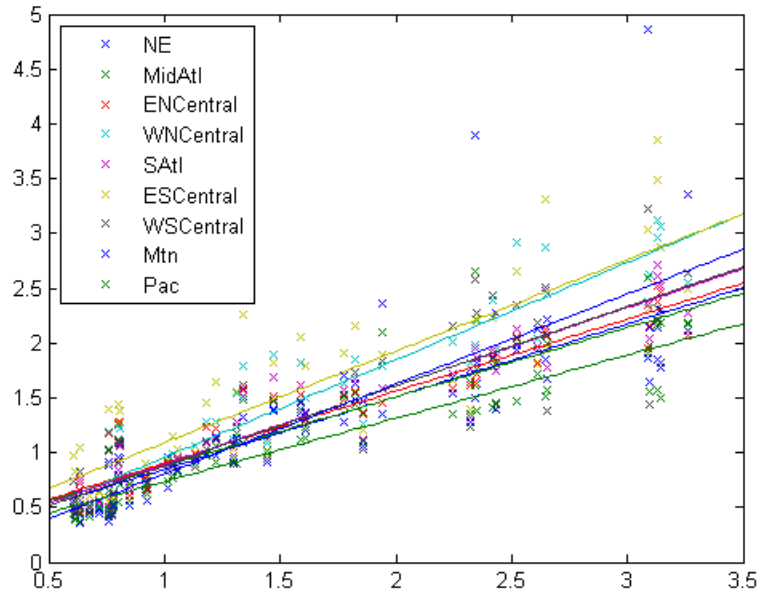
```

B = [beta(1:d)';beta(d+1:end)'];
xx = linspace(.5,3.5)';
fits = [ones(size(xx)),xx]*B;

```

# mvregress

```
figure;  
h = plot(x,Y,'x',xx,fits,'-');  
for i = 1:d  
    set(h(d+i),'color',get(h(i),'color'));  
end  
  
regions = flu.Properties.VarNames(2:end-1);  
legend(regions,'Location','NorthWest');
```



The plot shows that each regression line has a different intercept and slope.

## Multivariate Regression With a Single Design Matrix

Fit a multivariate regression model using a single  $n$ -by- $P$  design matrix for all response dimensions.



Load the sample data.

```
load('flu');
```

The dataset array `flu` contains national CDC flu estimates, and nine separate regional estimates based on Google queries.

Extract the response and predictor data.

```
Y = double(flu(:,2:end-1));
[n,d] = size(Y);
x = flu.WtdILI;
```

The responses in  $Y$  are the nine regional flu estimates. Observations exist for every week over a one-year period, so  $n = 52$ . The dimension of the responses corresponds to the regions, so  $d = 9$ . The predictors in  $x$  are the weekly national flu estimates.

Create an  $n$ -by- $P$  design matrix  $X$ . Add a column of ones to include a constant term in the regression.

```
X = [ones(size(x)), x];
```

Fit the multivariate regression model

$$y_{ij} = \alpha_j + \beta_j x_{ij} + \varepsilon_{ij}, \quad i = 1, \dots, n; \quad j = 1, \dots, d,$$

with between-region concurrent correlation

$$COV(\varepsilon_{ij}, \varepsilon_{ij'}) = \sigma_{jj'}, \quad j = 1, \dots, d.$$

There are 18 regression coefficients to estimate: nine intercept terms, and nine slope terms.

```
[beta, Sigma, E, CovB, logL] = mvregress(X, Y);
```

`beta` contains estimates of the  $P$ -by- $d$  coefficient matrix. `Sigma` contains estimates of the  $d$ -by- $d$  variance-covariance matrix for the between-region concurrent correlations. `E` is a matrix of the residuals.

## mvregress

---

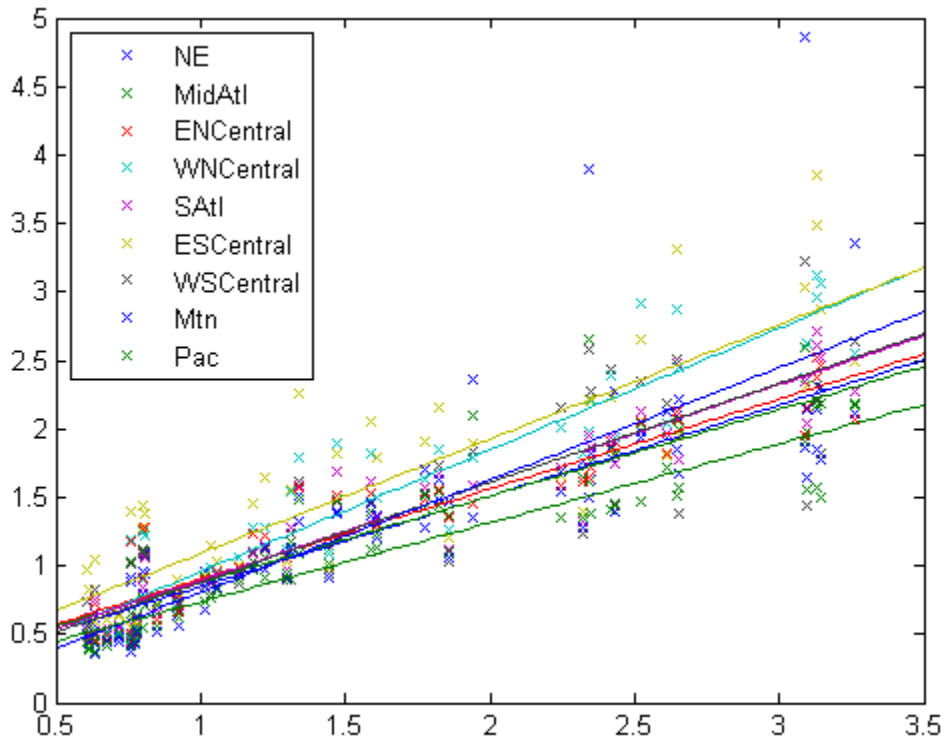
CovB is the estimated variance-covariance matrix of the regression coefficients. logL is the value of the log likelihood objective function after the last iteration.

Plot the fitted regression model.

```
B = beta;
xx = linspace(.5,3.5)';
fits = [ones(size(xx)),xx]*B;

figure;
h = plot(x,Y,'x', xx,fits,'-');
for i = 1:d
    set(h(d+i),'color',get(h(i),'color'));
end

regions = flu.Properties.VarNames(2:end-1);
legend(regions,'Location','NorthWest');
```



The plot shows that each regression line has a different intercept and slope.

## Definitions **Multivariate Normal Regression**

Multivariate normal regression is the regression of a  $d$ -dimensional response on a design matrix of predictor variables, with normally distributed errors. The errors can be heteroscedastic and correlated.

The model is

$$\mathbf{y}_i = \mathbf{X}_i\boldsymbol{\beta} + \mathbf{e}_i, \quad i = 1, \dots, n,$$

where

- $\mathbf{y}_i$  is a  $d$ -dimensional vector of responses.
- $\mathbf{X}_i$  is a design matrix of predictor variables.
- $\boldsymbol{\beta}$  is vector or matrix of regression coefficients.
- $\mathbf{e}_i$  is a  $d$ -dimensional vector of error terms, with multivariate normal distribution

$$\mathbf{e}_i \sim MVN_d(\mathbf{0}, \Sigma).$$

## Conditionally Imputed Values

The expectation/conditional maximization ('ecm') and covariance-weighted least squares ('cwlsl') estimation algorithms include imputation of missing response values.

Let  $\tilde{\mathbf{y}}$  denote missing observations. The conditionally imputed values are the expected value of the missing observation given the observed data,  $E(\tilde{\mathbf{y}} | \mathbf{y})$ .

The joint distribution of the missing and observed responses is a multivariate normal distribution,

$$\begin{pmatrix} \tilde{\mathbf{y}} \\ \mathbf{y} \end{pmatrix} \sim MVN \left\{ \begin{pmatrix} \tilde{\mathbf{X}}\boldsymbol{\beta} \\ \mathbf{X}\boldsymbol{\beta} \end{pmatrix}, \begin{pmatrix} \Sigma_{\tilde{\mathbf{y}}} & \Sigma_{\tilde{\mathbf{y}}\mathbf{y}} \\ \Sigma_{\mathbf{y}\tilde{\mathbf{y}}} & \Sigma_{\mathbf{y}} \end{pmatrix} \right\}.$$

Using properties of the multivariate normal distribution, the imputed conditional expectation is given by

$$E(\tilde{\mathbf{y}} | \mathbf{y}) = \tilde{\mathbf{X}}\boldsymbol{\beta} + \Sigma_{\tilde{\mathbf{y}}\mathbf{y}}\Sigma_{\mathbf{y}}^{-1}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}).$$

---

**Note** mvregress only imputes missing response values. Observations with missing values in the design matrix are removed.

---

## References

- [1] Little, Roderick J. A., and Donald B. Rubin. *Statistical Analysis with Missing Data*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2002.
- [2] Meng, Xiao-Li, and Donald B. Rubin. “Maximum Likelihood Estimation via the ECM Algorithm.” *Biometrika*. Vol. 80, No. 2, 1993, pp. 267–278.
- [3] Sexton, Joe, and A. R. Swensen. “ECM Algorithms that Converge at the Rate of EM.” *Biometrika*. Vol. 87, No. 3, 2000, pp. 651–662.
- [4] Dempster, A. P., N. M. Laird, and D. B. Rubin. “Maximum Likelihood from Incomplete Data via the EM Algorithm.” *Journal of the Royal Statistical Society*. Series B, Vol. 39, No. 1, 1977, pp. 1–37.

## See Also

manova1 | mvregresslike

## Related Examples

- “Set Up Multivariate Regression Problems” on page 13-16
- “Multivariate General Linear Model” on page 13-30
- “Fixed Effects Panel Model with Concurrent Correlation” on page 13-35
- “Longitudinal Analysis” on page 13-43

## Concepts

- “Multivariate Linear Regression” on page 13-3
- “Estimation of Multivariate Regression Models” on page 13-6

# mvregresslike

---

**Purpose** Negative log-likelihood for multivariate regression

**Syntax**  
`nlogL = mvregresslike(X,Y,b,SIGMA,alg)`  
`[nlogL,COVB] = mvregresslike(...)`  
`[nlogL,COVB] = mvregresslike(...,type,format)`

**Description** `nlogL = mvregresslike(X,Y,b,SIGMA,alg)` computes the negative log-likelihood `nlogL` for a multivariate regression of the  $d$ -dimensional multivariate observations in the  $n$ -by- $d$  matrix `Y` on the predictor variables in the matrix or cell array `X`, evaluated for the  $p$ -by-1 column vector `b` of coefficient estimates and the  $d$ -by- $d$  matrix `SIGMA` specifying the covariance of a row of `Y`. If  $d = 1$ , `X` can be an  $n$ -by- $p$  design matrix of predictor variables. For any value of  $d$ , `X` can also be a cell array of length  $n$ , with each cell containing a  $d$ -by- $p$  design matrix for one multivariate observation. If all observations have the same  $d$ -by- $p$  design matrix, `X` can be a single cell.

NaN values in `X` or `Y` are taken as missing. Observations with missing values in `X` are ignored. Treatment of missing values in `Y` depends on the algorithm specified by `alg`.

`alg` should match the algorithm used by `mvregress` to obtain the coefficient estimates `b`, and must be one of the following:

- 'ecm' — ECM algorithm
- 'cwlsl' — Least squares conditionally weighted by `SIGMA`
- 'mvm' — Multivariate normal estimates computed after omitting rows with any missing values in `Y`

`[nlogL,COVB] = mvregresslike(...)` also returns an estimated covariance matrix `COVB` of the parameter estimates `b`.

`[nlogL,COVB] = mvregresslike(...,type,format)` specifies the type and format of `COVB`.

`type` is either:

- 'hessian' — To use the Hessian or observed information. This method takes into account the increased uncertainties due to missing data. This is the default.
- 'fisher' — To use the Fisher or expected information. This method uses the complete data expected information, and does not include uncertainty due to missing data.

*format* is either:

- 'beta' — To compute COVB for b only. This is the default.
- 'full' — To compute COVB for both b and SIGMA.

## See Also

mvregress | manova1

## How To

- “Multivariate Normal Distribution” on page B-86

# mvnrnd

---

**Purpose** Multivariate normal random numbers

**Syntax**  
`R = mvnrnd(MU,SIGMA)`  
`r = mvnrnd(MU,SIGMA,cases)`

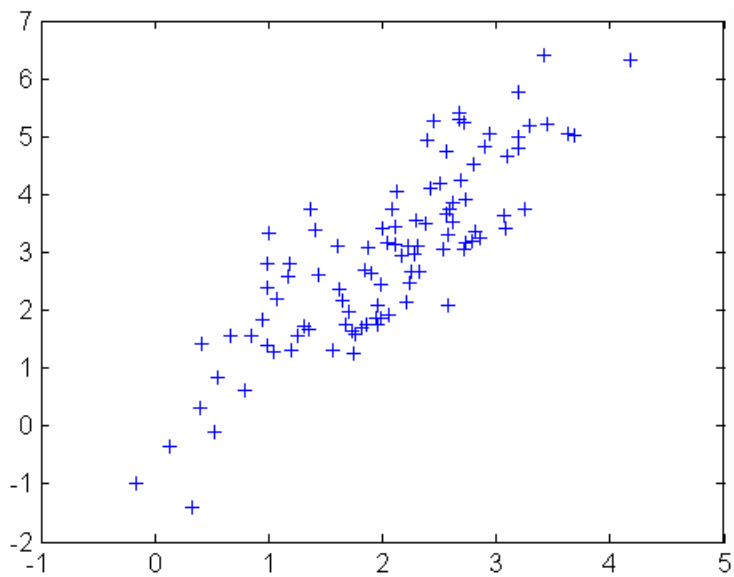
**Description** `R = mvnrnd(MU,SIGMA)` returns an  $n$ -by- $d$  matrix `R` of random vectors chosen from the multivariate normal distribution with mean `MU`, and covariance `SIGMA`. `MU` is an  $n$ -by- $d$  matrix, and `mvnrnd` generates each row of `R` using the corresponding row of `mu`. `SIGMA` is a  $d$ -by- $d$  symmetric positive semi-definite matrix, or a  $d$ -by- $d$ -by- $n$  array. If `SIGMA` is an array, `mvnrnd` generates each row of `R` using the corresponding page of `SIGMA`, i.e., `mvnrnd` computes `R(i,:)` using `MU(i,:)` and `SIGMA(:,:,i)`. If the covariance matrix is diagonal, containing variances along the diagonal and zero covariances off the diagonal, `SIGMA` may also be specified as a 1-by- $d$  vector or a 1-by- $d$ -by- $n$  array, containing just the diagonal. If `MU` is a 1-by- $d$  vector, `mvnrnd` replicates it to match the trailing dimension of `SIGMA`.

`r = mvnrnd(MU,SIGMA,cases)` returns a cases-by- $d$  matrix `R` of random vectors chosen from the multivariate normal distribution with a common 1-by- $d$  mean vector `MU`, and a common  $d$ -by- $d$  covariance matrix `SIGMA`.

**Examples**

```
mu = [2 3];  
SIGMA = [1 1.5; 1.5 3];  
r = mvnrnd(mu,SIGMA,100);  
plot(r(:,1),r(:,2),'+')
```



**See Also**

[mvnpdf](#) | [mvncdf](#) | [normrnd](#)

**How To**

- “Multivariate Normal Distribution” on page B-86

# mvtcdf

---

**Purpose** Multivariate  $t$  cumulative distribution function

**Syntax**

```
y = mvtcdf(X,C,DF)
y = mvtcdf(x1,xu,C,DF)
[y,err] = mvtcdf(...)
[...] = mvntdf(...,options)
```

**Description** `y = mvtcdf(X,C,DF)` returns the cumulative probability of the multivariate  $t$  distribution with correlation parameters **C** and degrees of freedom **DF**, evaluated at each row of **X**. Rows of the  $n$ -by- $d$  matrix **X** correspond to observations or points, and columns correspond to variables or coordinates. **y** is an  $n$ -by-1 vector.

**C** is a symmetric, positive definite,  $d$ -by- $d$  matrix, typically a correlation matrix. If its diagonal elements are not 1, `mvtcdf` scales **C** to correlation form. `mvtcdf` does not rescale **X**. **DF** is a scalar, or a vector with  $n$  elements.

The multivariate  $t$  cumulative probability at **X** is defined as the probability that a random vector **T**, distributed as multivariate  $t$ , will fall within the semi-infinite rectangle with upper limits defined by **X**, i.e.,  $\Pr\{T(1) \leq X(1), T(2) \leq X(2), \dots, T(d) \leq X(d)\}$ .

`y = mvtcdf(x1,xu,C,DF)` returns the multivariate  $t$  cumulative probability evaluated over the rectangle with lower and upper limits defined by **x1** and **xu**, respectively.

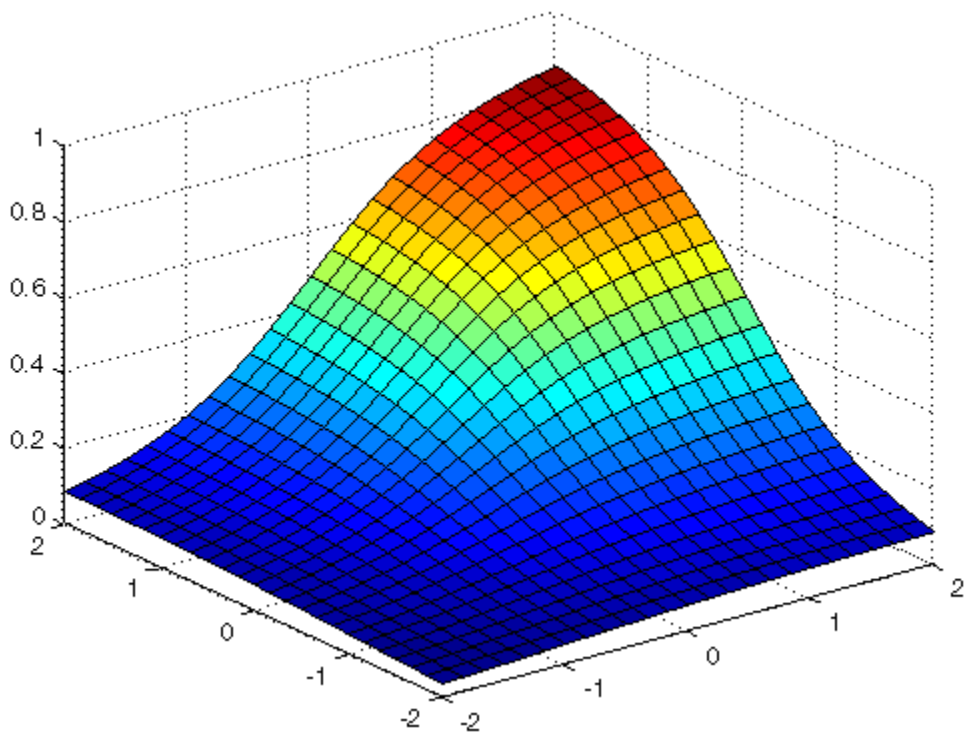
`[y,err] = mvtcdf(...)` returns an estimate of the error in **y**. For bivariate and trivariate distributions, `mvtcdf` uses adaptive quadrature on a transformation of the  $t$  density, based on methods developed by Genz, as described in the references. The default absolute error tolerance for these cases is  $1e-8$ . For four or more dimensions, `mvtcdf` uses a quasi-Monte Carlo integration algorithm based on methods developed by Genz and Bretz, as described in the references. The default absolute error tolerance for these cases is  $1e-4$ .

`[...] = mvntdf(...,options)` specifies control parameters for the numerical integration used to compute **y**. This argument can be created by a call to `statset`. Choices of `statset` parameters are:

- 'TolFun' — Maximum absolute error tolerance. Default is  $1e-8$  when  $d < 4$ , or  $1e-4$  when  $d \geq 4$ .
- 'MaxFunEvals' — Maximum number of integrand evaluations allowed when  $d \geq 4$ . Default is  $1e7$ . 'MaxFunEvals' is ignored when  $d < 4$ .
- 'Display' — Level of display output. Choices are 'off' (the default), 'iter', and 'final'. 'Display' is ignored when  $d < 4$ .

## Examples

```
C = [1 .4; .4 1]; df = 2;
[X1,X2] = meshgrid(linspace(-2,2,25)',linspace(-2,2,25)');
X = [X1(:) X2(:)];
p = mvtcdf(X,C,df);
surf(X1,X2,reshape(p,25,25));
```



## References

- [1] Genz, A. “Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities.” *Statistics and Computing*. Vol. 14, No. 3, 2004, pp. 251–260.
- [2] Genz, A., and F. Bretz. “Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple Contrasts.” *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.
- [3] Genz, A., and F. Bretz. “Comparison of Methods for the Computation of Multivariate t Probabilities.” *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.

**See Also**      mvtpdf | mvtrnd

**How To**      • “Multivariate  $t$  Distribution” on page B-91

# mvtpdf

---

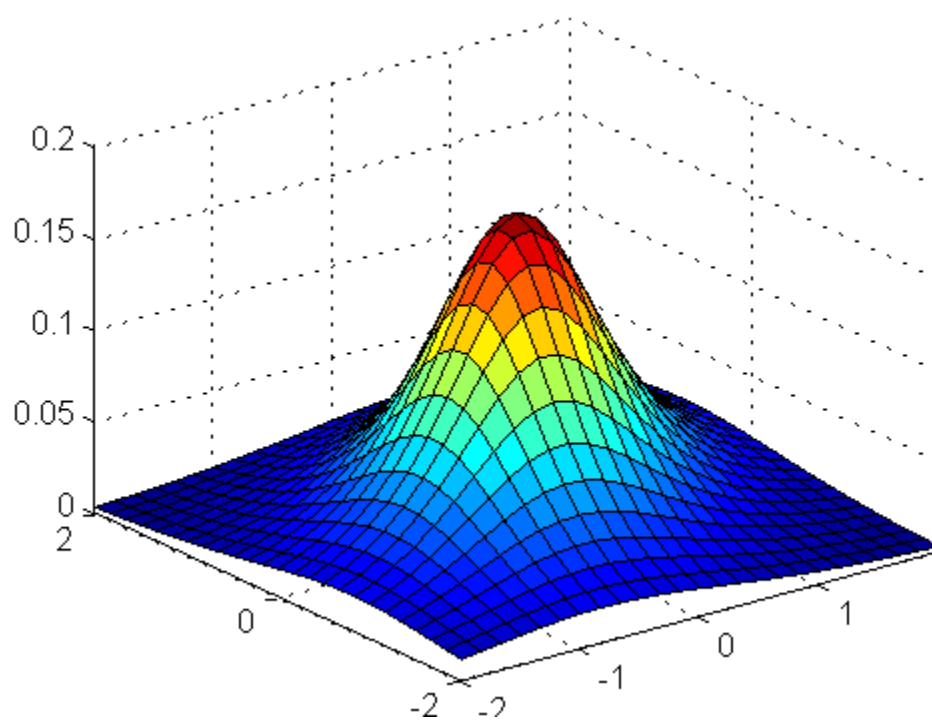
**Purpose** Multivariate  $t$  probability density function

**Syntax** `y = mvtpdf(X,C,df)`

**Description** `y = mvtpdf(X,C,df)` returns the probability density of the multivariate  $t$  distribution with correlation parameters `C` and degrees of freedom `df`, evaluated at each row of `X`. Rows of the  $n$ -by- $d$  matrix `X` correspond to observations or points, and columns correspond to variables or coordinates. `C` is a symmetric, positive definite,  $d$ -by- $d$  matrix, typically a correlation matrix. If its diagonal elements are not 1, `mvtpdf` scales `C` to correlation form. `mvtcdf` does not rescale `X`. `df` is a scalar, or a vector with  $n$  elements. `y` is an  $n$ -by-1 vector.

**Examples** Visualize a multivariate  $t$  distribution:

```
[X1,X2] = meshgrid(linspace(-2,2,25)',linspace(-2,2,25)');
X = [X1(:) X2(:)];
C = [1 .4; .4 1];
df = 2;
p = mvtpdf(X,C,df);
surf(X1,X2,reshape(p,25,25))
```

**See Also**

[mvtcdf](#) | [mvtrnd](#)

**How To**

- “Multivariate  $t$  Distribution” on page B-91

# mvtrnd

---

**Purpose** Multivariate  $t$  random numbers

**Syntax** `R = mvtrnd(C,df,cases)`  
`R = mvtrnd(C,df)`

**Description** `R = mvtrnd(C,df,cases)` returns a matrix of random numbers chosen from the multivariate  $t$  distribution, where `C` is a correlation matrix. `df` is the degrees of freedom and is either a scalar or is a vector with `cases` elements. If `p` is the number of columns in `C`, then the output `R` has `cases` rows and `p` columns.

Let `t` represent a row of `R`. Then the distribution of `t` is that of a vector having a multivariate normal distribution with mean 0, variance 1, and covariance matrix `C`, divided by an independent chi-square random value having `df` degrees of freedom. The rows of `R` are independent.

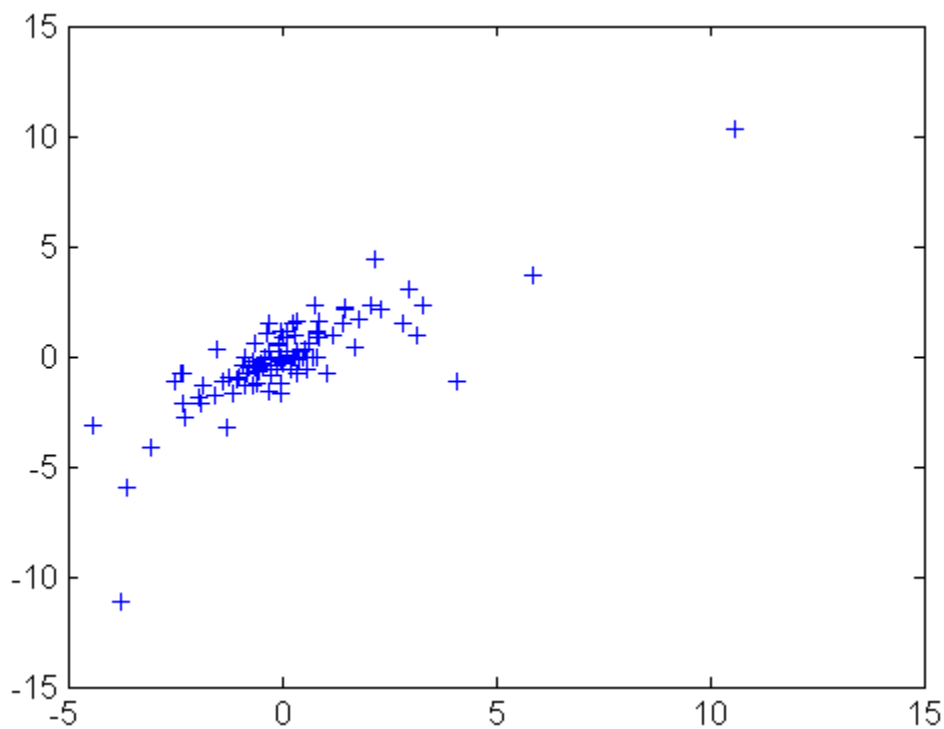
`C` must be a square, symmetric and positive definite matrix. If its diagonal elements are not all 1 (that is, if `C` is a covariance matrix rather than a correlation matrix), `mvtrnd` rescales `C` to transform it to a correlation matrix before generating the random numbers.

`R = mvtrnd(C,df)` returns a single random number from the multivariate  $t$  distribution.

**Examples**

```
SIGMA = [1 0.8;0.8 1];  
R = mvtrnd(SIGMA,3,100);  
plot(R(:,1),R(:,2),'+')
```



**See Also**

`mvtpdf` | `mvtcdf`

**How To**

- “Multivariate  $t$  Distribution” on page B-91

# cvpartition.N property

---

**Purpose**            Number of observations (including observations with missing group values)

**Description**        Number of observations (including observations with missing group values).

<b>Purpose</b>	Naive Bayes classifier	
<b>Description</b>	A NaiveBayes object defines a Naive Bayes classifier. A Naive Bayes classifier assigns a new observation to the most probable class, assuming the features are conditionally independent given the class value.	
<b>Construction</b>	NaiveBayes	Create NaiveBayes object
<b>Methods</b>	disp	Display NaiveBayes classifier object
	display	Display NaiveBayes classifier object
	fit	Create Naive Bayes classifier object by fitting training data
	posterior	Compute posterior probability of each class for test data
	predict	Predict class label for test data
	subsasgn	Subscripted reference for NaiveBayes object
	subsref	Subscripted reference for NaiveBayes object
<b>Properties</b>	CIIsNonEmpty	Flag for non-empty classes
	ClassLevels	Class levels
	Dist	Distribution names
	NClasses	Number of classes

# NaiveBayes

---

NDims	Number of dimensions
Params	Parameter estimates
Prior	Class priors

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## Examples

Predict the class label using the Naive Bayes classifier:

```
load fisheriris
```

Use the default Gaussian distribution and a confusion matrix:

```
O1 = NaiveBayes.fit(meas,species);  
C1 = O1.predict(meas);  
cMat1 = confusionmat(species,C1)
```

This returns:

```
cMat1 =  
  
    50     0     0  
     0    47     3  
     0     3    47
```

Use the Gaussian distribution for features 1 and 3 and use the kernel density estimation for features 2 and 4:

```
O2 = NaiveBayes.fit(meas,species,'dist',...  
{'normal','kernel','normal','kernel'});  
C2 = O2.predict(meas);  
cMat2 = confusionmat(species,C2)
```

This returns:

```
cMat2 =
```

50	0	0
0	47	3
0	3	47

## References

[1] Mitchell, T. (1997) Machine Learning, McGraw Hill.

[2] Vangelis M., Ion A., and Geogios P. Spam Filtering with Naive Bayes - Which Naive Bayes? (2006) Third Conference on Email and Anti-Spam.

[3] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers (1995) the Eleventh Conference on Uncertainty in Artificial Intelligence.

## How To

- “Naive Bayes Classification” on page 15-38
- “Grouping Variables” on page 2-51

# NaiveBayes

---

**Purpose** Create NaiveBayes object

**Description** You cannot create a NaiveBayes classifier by calling the constructor. Use `NaiveBayes.fit` to create a NaiveBayes classifier by fitting the object to training data.

**See Also** `fit`

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Nakagami probability distribution object

**Description** `prob.NakagamiDistribution` is an object consisting of parameters, a model description, and sample data for a Nakagami probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Nakagami')` creates a Nakagami probability distribution object using the default parameter values.

`pd = makedist('Nakagami','mu',mu,'omega',omega)` creates a Nakagami probability distribution object using the specified parameter values.

## Input Arguments

### **mu - Shape parameter**

1 (default) | positive scalar value

Shape parameter for the Nakagami distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **omega - Scale parameter**

1 (default) | positive scalar value

Scale parameter for the Nakagami distribution, specified as a positive scalar value.

### **Data Types**

single | double

# prob.NakagamiDistribution

---

## Properties

### **mu**

Shape parameter for the Nakagami distribution, stored as a positive scalar value.

### **Data Types**

single | double

### **omega**

Scale parameter for the Nakagami distribution, stored as a positive scalar value.

### **Data Types**

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### **Data Types**

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

### **Data Types**

single | double

### **IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.



If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

# prob.NakagamiDistribution

---

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Nakagami Distribution

The Nakagami distribution is commonly used in communication theory to model scattered signals that reach a receiver using multiple paths.

The Nakagami distribution uses the following parameters.

Parameter	Description	Support
mu	Shape parameter	$\mu > 0$
omega	Scale parameter	$\omega > 0$

# prob.NakagamiDistribution

---

The probability density function (pdf) is

$$f(x | \mu, \omega) = 2 \left( \frac{\mu^\mu}{\omega} \right) \frac{1}{\Gamma(\mu)} x^{(2\mu-1)} \exp\left\{ \frac{-\mu}{\omega} x^2 \right\} ; \quad x > 0,$$

where  $\Gamma(\cdot)$  is the Gamma function.

## Examples

### Create a Nakagami Distribution Object Using Default Parameters

Create a Nakagami distribution object using the default parameter values.

```
pd = makedist('Nakagami')
```

```
pd =
```

```
NakagamiDistribution
```

```
Nakagami distribution
```

```
mu = 1
```

```
omega = 1
```

### Create a Nakagami Distribution Object Using Specified Parameters

Create a Nakagami distribution object by specifying parameter values.

```
pd = makedist('Nakagami','mu',5,'omega',2)
```

```
pd =
```

```
NakagamiDistribution
```

```
Nakagami distribution
```

```
mu = 5
```

```
omega = 2
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1.3794
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Nakagami Distribution” on page B-97
- Class Attributes
- Property Attributes

# nancov

---

**Purpose** Covariance ignoring NaN values

**Syntax**  
`Y = nancov(X)`  
`Y = nancov(X1,X2)`  
`Y = nancov(...,1)`  
`Y = nancov(...,'pairwise')`

**Description** `Y = nancov(X)` is the covariance cov of `X`, computed after removing observations with NaN values.

For vectors `x`, `nancov(x)` is the sample variance of the remaining elements, once NaN values are removed. For matrices `X`, `nancov(X)` is the sample covariance of the remaining observations, once observations (rows) containing any NaN values are removed.

`Y = nancov(X1,X2)`, where `X1` and `X2` are matrices with the same number of elements, is equivalent to `nancov(X)`, where `X = [X1(:) X2(:)]`.

`nancov` removes the mean from each variable (column for matrix `X`) before calculating `Y`. If `n` is the number of remaining observations after removing observations with NaN values, `nancov` normalizes `Y` by either  $n - 1$  or  $n$ , depending on whether  $n > 1$  or  $n = 1$ , respectively. To specify normalization by  $n$ , use `Y = nancov(...,1)`.

`Y = nancov(...,'pairwise')` computes `Y(i,j)` using rows with no NaN values in columns `i` or `j`. The result `Y` may not be a positive definite matrix.

**Examples** Generate random data for two variables (columns) with random missing values:

```
X = rand(10,2);
p = randperm(numel(X));
X(p(1:5)) = NaN
X =
    0.8147    0.1576
         NaN         NaN
    0.1270    0.9572
```

```

0.9134      NaN
0.6324      NaN
0.0975     0.1419
0.2785     0.4218
0.5469     0.9157
0.9575     0.7922
0.9649      NaN

```

Establish a correlation between a third variable and the other two variables:

```

X(:,3) = sum(X,2)
X =

```

```

0.8147     0.1576     0.9723
      NaN      NaN      NaN
0.1270     0.9572     1.0842
0.9134      NaN      NaN
0.6324      NaN      NaN
0.0975     0.1419     0.2394
0.2785     0.4218     0.7003
0.5469     0.9157     1.4626
0.9575     0.7922     1.7497
0.9649      NaN      NaN

```

Compute the covariance matrix for the three variables after removing observations (rows) with NaN values:

```

Y = nancov(X)
Y =

```

```

0.1311     0.0096     0.1407
0.0096     0.1388     0.1483
0.1407     0.1483     0.2890

```

## See Also

NaN | cov | var | nanvar

# nanmax

---

**Purpose** Maximum ignoring NaN values

**Syntax**

```
y = nanmax(X)
Y = nanmax(X1,X2)
y = nanmax(X,[],dim)
[y,indices] = nanmax(...)
```

**Description** `y = nanmax(X)` is the maximum max of `X`, computed after removing NaN values.

For vectors `x`, `nanmax(x)` is the maximum of the remaining elements, once NaN values are removed. For matrices `X`, `nanmax(X)` is a row vector of column maxima, once NaN values are removed. For multidimensional arrays `X`, `nanmax` operates along the first nonsingleton dimension.

`Y = nanmax(X1,X2)` returns an array `Y` the same size as `X1` and `X2` with `Y(i,j) = nanmax(X1(i,j),X2(i,j))`. Scalar inputs are expanded to an array of the same size as the other input.

`y = nanmax(X,[],dim)` operates along the dimension `dim` of `X`.

`[y,indices] = nanmax(...)` also returns the row indices of the maximum values for each column in the vector `indices`.

**Examples** Find column maxima and their indices for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
[y,indices] = nanmax(X)
y =
     4     5    NaN
indices =
     3     2     1
```

**See Also** NaN | max | nanmin



**Purpose** Mean ignoring NaN values

**Syntax**  
`y = nanmean(X)`  
`y = nanmean(X,dim)`

**Description** `y = nanmean(X)` is the mean of `X`, computed after removing NaN values. For vectors `x`, `nanmean(x)` is the mean of the remaining elements, once NaN values are removed. For matrices `X`, `nanmean(X)` is a row vector of column means, once NaN values are removed. For multidimensional arrays `X`, `nanmean` operates along the first nonsingleton dimension. `y = nanmean(X,dim)` takes the mean along dimension `dim` of `X`.

---

**Note** If `X` contains a vector of all NaN values along some dimension, the vector is empty once the NaN values are removed, so the sum of the remaining elements is 0. Since the mean involves division by 0, its value is NaN. The output NaN is not a mean of NaN values.

---

**Examples** Find column means for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
y = nanmean(X)
y =
    3.5000    3.0000    NaN
```

**See Also** NaN | mean | nanmedian

# nanmedian

---

**Purpose** Median ignoring NaN values

**Syntax** `y = nanmedian(X)`  
`y = nanmedian(X,dim)`

**Description** `y = nanmedian(X)` is the median of `X`, computed after removing NaN values.

For vectors `x`, `nanmedian(x)` is the median of the remaining elements, once NaN values are removed. For matrices `X`, `nanmedian(X)` is a row vector of column medians, once NaN values are removed. For multidimensional arrays `X`, `nanmedian` operates along the first nonsingleton dimension.

`y = nanmedian(X,dim)` takes the mean along dimension `dim` of `X`.

**Examples** Find column medians for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
y = nanmedian(X)
y =
    3.5000    3.0000    NaN
```

**See Also** `NaN` | `median` | `nanmean`

**Purpose** Minimum ignoring NaN values

**Syntax**

```
y = nanmin(X)
Y = nanmin(X1,X2)
y = nanmin(X,[],dim)
[y,indices] = nanmin(...)
```

**Description** `y = nanmin(X)` is the minimum min of `X`, computed after removing NaN values.

For vectors `x`, `nanmin(x)` is the minimum of the remaining elements, once NaN values are removed. For matrices `X`, `nanmin(X)` is a row vector of column minima, once NaN values are removed. For multidimensional arrays `X`, `nanmin` operates along the first nonsingleton dimension.

`Y = nanmin(X1,X2)` returns an array `Y` the same size as `X1` and `X2` with `Y(i,j) = nanmin(X1(i,j),X2(i,j))`. Scalar inputs are expanded to an array of the same size as the other input.

`y = nanmin(X,[],dim)` operates along the dimension `dim` of `X`.

`[y,indices] = nanmin(...)` also returns the row indices of the minimum values for each column in the vector `indices`.

**Examples** Find column minima and their indices for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
[y,indices] = nanmin(X)
y =
     3     1    NaN
indices =
     2     1     1
```

**See Also** NaN | min | nanmax

# nanstd

---

**Purpose** Standard deviation ignoring NaN values

**Syntax**  
`y = nanstd(X)`  
`y = nanstd(X,1)`  
`y = nanstd(X,flag,dim)`

**Description** `y = nanstd(X)` is the standard deviation `std` of `X`, computed after removing NaN values.

For vectors `x`, `nanstd(x)` is the sample standard deviation of the remaining elements, once NaN values are removed. For matrices `X`, `nanstd(X)` is a row vector of column sample standard deviations, once NaN values are removed. For multidimensional arrays `X`, `nanstd` operates along the first nonsingleton dimension.

If  $n$  is the number of remaining observations after removing observations with NaN values, `nanstd` normalizes `y` by  $n-1$ . To specify normalization by  $n$ , use `y = nanstd(X,1)`.

`y = nanstd(X,flag,dim)` takes the standard deviation along the dimension `dim` of `X`. The `flag` is 0 or 1 to specify normalization by  $n-1$  or  $n$ , respectively, where  $n$  is the number of remaining observations after removing observations with NaN values.

**Examples** Find column standard deviations for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
y = nanstd(X)
y =
    0.7071    2.8284    NaN
```

**See Also** [NaN](#) | [std](#) | [nanvar](#) | [nanmean](#)

**Purpose** Sum ignoring NaN values

**Syntax**  
`y = nansum(X)`  
`y = nansum(X,dim)`

**Description** `y = nansum(X)` is the sum of `X`, computed after removing NaN values. For vectors `x`, `nansum(x)` is the sum of the remaining elements, once NaN values are removed. For matrices `X`, `nansum(X)` is a row vector of column sums, once NaN values are removed. For multidimensional arrays `X`, `nansum` operates along the first nonsingleton dimension. `y = nansum(X,dim)` takes the sum along dimension `dim` of `X`.

---

**Note** If `X` contains a vector of all NaN values along some dimension, the vector is empty once the NaN values are removed, so the sum of the remaining elements is 0. The output 0 is not a sum of NaN values.

---

**Examples** Find column sums for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
y = nansum(X)
y =
     7     6     0
```

**See Also** NaN | sum

**Purpose** Variance, ignoring NaN values

**Syntax**

```
y = nanvar(X)
y = nanvar(X,1)
y = nanvar(X,w)
y = nanvar(X,w,dim)
```

**Description** `y = nanvar(X)` is the variance `var` of `X`, computed after removing NaN values.

For vectors `x`, `nanvar(x)` is the sample variance of the remaining elements, once NaN values are removed. For matrices `X`, `nanvar(X)` is a row vector of column sample variances, once NaN values are removed. For multidimensional arrays `X`, `nanvar` operates along the first nonsingleton dimension.

`nancov` removes the mean from each variable (column for matrix `X`) before calculating `Y`. If  $n$  is the number of remaining observations after removing observations with NaN values, `nanvar` normalizes `y` by either  $n - 1$  or  $n$ , depending on whether  $n > 1$  or  $n = 1$ , respectively. To specify normalization by  $n$ , use `y = nanvar(X,1)`.

`y = nanvar(X,w)` computes the variance using the weight vector `w`. The length of `w` must equal the length of the dimension over which `nanvar` operates, and its elements must be nonnegative. Elements of `X` corresponding to NaN values of `w` are ignored.

`y = nanvar(X,w,dim)` takes the variance along the dimension `dim` of `X`. Set `w` to `[]` to use the default normalization by  $n - 1$ .

**Examples** Find column standard deviations for data with missing values:

```
X = magic(3);
X([1 6:9]) = repmat(NaN,1,5)
X =
    NaN     1    NaN
     3     5    NaN
     4    NaN    NaN
y = nanvar(X)
```

```
y =  
0.5000 8.0000 NaN
```

**See Also**

[NaN](#) | [var](#) | [nanstd](#) | [nanmean](#)

# nbincdf

---

**Purpose** Negative binomial cumulative distribution function

**Syntax**  
`y = nbincdf(x,R,p)`  
`y = nbincdf(x,R,p, 'upper')`

**Description** `y = nbincdf(x,R,p)` computes the negative binomial cdf at each of the values in `x` using the corresponding number of successes, `R` and probability of success in a single trial, `p`. `x`, `R`, and `p` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `y`. A scalar input for `x`, `R`, or `p` is expanded to a constant array with the same dimensions as the other inputs.

`y = nbincdf(x,R,p, 'upper')` returns the complement of the negative binomial cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The negative binomial cdf is

$$y = F(x | r, p) = \sum_{i=0}^x \binom{r+i-1}{i} p^r q^i I_{(0,1,\dots)}(i)$$

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `p` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbincdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the cdf is replaced by the equivalent expression

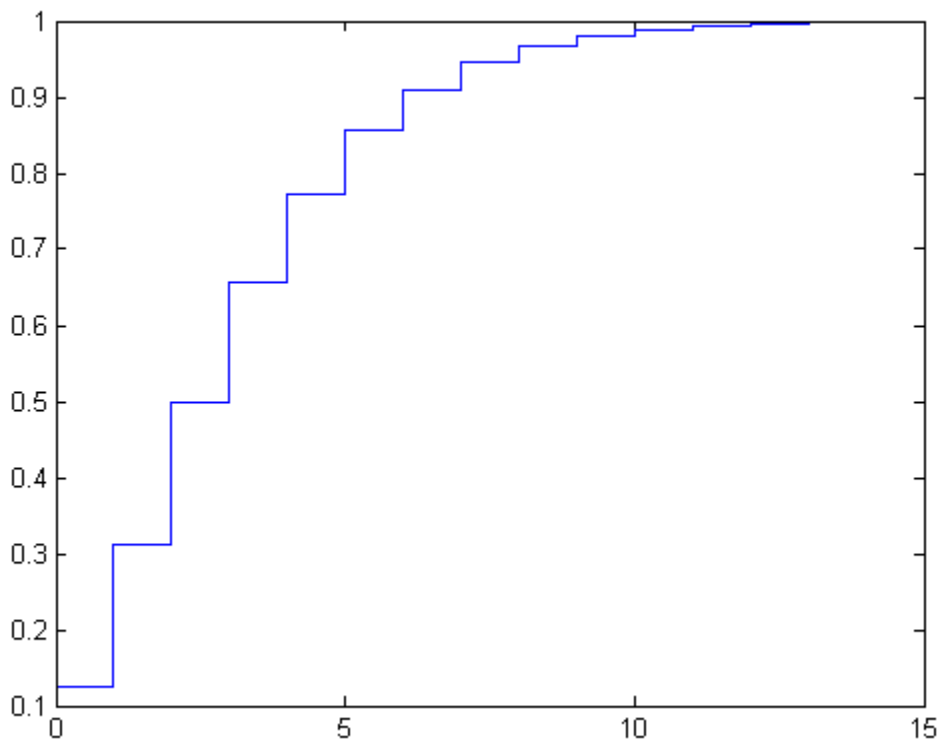
$$\frac{\Gamma(r+i)}{\Gamma(r)\Gamma(i+1)}$$

**Examples**      **Compute Negative Binomial Distribution CDF**

`x = (0:15);`



```
p = nbincdf(x,3,0.5);  
stairs(x,p)
```

**See Also**

[cdf](#) | [nbinpdf](#) | [nbininv](#) | [nbinstat](#) | [nbinfit](#) | [nbinrnd](#)

**How To**

- “Negative Binomial Distribution” on page B-98

# nbinfit

---

**Purpose** Negative binomial parameter estimates

**Syntax**

```
parmhat = nbinfit(data)
[parmhat,parmci] = nbinfit(data,alpha)
[...] = nbinfit(data,alpha,options)
```

**Description** `parmhat = nbinfit(data)` returns the maximum likelihood estimates (MLEs) of the parameters of the negative binomial distribution given the data in the vector `data`.

`[parmhat,parmci] = nbinfit(data,alpha)` returns MLEs and 100(1-alpha) percent confidence intervals. By default, `alpha = 0.05`, which corresponds to 95% confidence intervals.

`[...] = nbinfit(data,alpha,options)` accepts a structure, `options`, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The negative binomial fit function accepts an `options` structure which you can create using the function `statset`. Enter `statset('nbinfit')` to see the names and default values of the parameters that `nbinfit` accepts in the `options` structure. See the reference page for `statset` for more information about these options.

---

**Note** The variance of a negative binomial distribution is greater than its mean. If the sample variance of the data in `data` is less than its sample mean, `nbinfit` cannot compute MLEs. You should use the `poissfit` function instead.

---

**See Also** `nbincdf` | `nbininv` | `nbinpdf` | `nbinrnd` | `nbinstat` | `mle` | `statset`

**How To**

- “Negative Binomial Distribution” on page B-98

<b>Purpose</b>	Negative binomial inverse cumulative distribution function
<b>Syntax</b>	$X = \text{nbininv}(Y, R, P)$
<b>Description</b>	<p><math>X = \text{nbininv}(Y, R, P)</math> returns the inverse of the negative binomial cdf with corresponding number of successes, <math>R</math> and probability of success in a single trial, <math>P</math>. Since the binomial distribution is discrete, <code>nbininv</code> returns the least integer <math>X</math> such that the negative binomial cdf evaluated at <math>X</math> equals or exceeds <math>Y</math>. <math>Y</math>, <math>R</math>, and <math>P</math> can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of <math>X</math>. A scalar input for <math>Y</math>, <math>R</math>, or <math>P</math> is expanded to a constant array with the same dimensions as the other inputs.</p> <p>The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability <math>P</math> of success. The number of <i>extra</i> trials you must perform in order to observe a given number <math>R</math> of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, <code>nbininv</code> allows <math>R</math> to be any positive value, including nonintegers.</p>
<b>Examples</b>	<p>How many times would you need to flip a fair coin to have a 99% probability of having observed 10 heads?</p> <pre>flips = nbininv(0.99, 10, 0.5) + 10 flips =     33</pre> <p>Note that you have to flip at least 10 times to get 10 heads. That is why the second term on the right side of the equals sign is a 10.</p>
<b>See Also</b>	<code>icdf</code>   <code>nbincdf</code>   <code>nbinpdf</code>   <code>nbinstat</code>   <code>nbinfit</code>   <code>nbinrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Negative Binomial Distribution” on page B-98</li></ul>

# nbinpdf

---

**Purpose** Negative binomial probability density function

**Syntax** `Y = nbinpdf(X,R,P)`

**Description** `Y = nbinpdf(X,R,P)` returns the negative binomial pdf at each of the values in `X` using the corresponding number of successes, `R` and probability of success in a single trial, `P`. `X`, `R`, and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X`, `R`, or `P` is expanded to a constant array with the same dimensions as the other inputs. Note that the density function is zero unless the values in `X` are integers.

The negative binomial pdf is

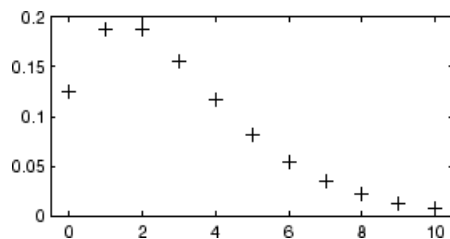
$$y = f(x | r, p) = \binom{r+x-1}{x} p^r q^x I_{(0,1,\dots)}(x)$$

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinpdf` allows `R` to be any positive value, including nonintegers. When `R` is noninteger, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

$$\frac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)}$$

**Examples**

```
x = (0:10);  
y = nbinpdf(x,3,0.5);  
plot(x,y, '+')  
set(gca, 'Xlim', [-0.5,10.5])
```

**See Also**

[pdf](#) | [nbincdf](#) | [nbininv](#) | [nbinstat](#) | [nbinfit](#) | [nbinrnd](#)

**How To**

- “Negative Binomial Distribution” on page B-98

# nbinrnd

---

**Purpose** Negative binomial random numbers

**Syntax**  
RND = nbinrnd(R,P)  
RND = nbinrnd(R,P,m,n,...)  
RND = nbinrnd(R,P,[m,n,...])

**Description** RND = nbinrnd(R,P) is a matrix of random numbers chosen from a negative binomial distribution with corresponding number of successes, R and probability of success in a single trial, P. R and P can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of RND. A scalar input for R or P is expanded to a constant array with the same dimensions as the other input.

RND = nbinrnd(R,P,m,n,...) or RND = nbinrnd(R,P,[m,n,...]) generates an m-by-n-by-... array. The R, P parameters can each be scalars or arrays of the same size as R.

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability P of success. The number of *extra* trials you must perform in order to observe a given number R of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, nbinrnd allows R to be any positive value, including nonintegers.

**Examples** Suppose you want to simulate a process that has a defect probability of 0.01. How many units might Quality Assurance inspect before finding three defective items?

```
r = nbinrnd(3,0.01,1,6)+3  
r =  
    496    142    420    396    851    178
```

**See Also** random | nbinpdf | nbincdf | nbininv | nbinstat | nbinfit

**How To** • “Negative Binomial Distribution” on page B-98

**Purpose** Negative binomial mean and variance

**Syntax** `[M,V] = nbinstat(R,P)`

**Description** `[M,V] = nbinstat(R,P)` returns the mean of and variance for the negative binomial distribution with corresponding number of successes, `R` and probability of success in a single trial, `P`. `R` and `P` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `R` or `P` is expanded to a constant array with the same dimensions as the other input.

The mean of the negative binomial distribution with parameters  $r$  and  $p$  is  $rq / p$ , where  $q = 1 - p$ . The variance is  $rq / p^2$ .

The simplest motivation for the negative binomial is the case of successive random trials, each having a constant probability `P` of success. The number of *extra* trials you must perform in order to observe a given number `R` of successes has a negative binomial distribution. However, consistent with a more general interpretation of the negative binomial, `nbinstat` allows `R` to be any positive value, including nonintegers.

## Examples

```
p = 0.1:0.2:0.9;
r = 1:5;
[R,P] = meshgrid(r,p);
[M,V] = nbinstat(R,P)
M =
    9.0000    18.0000    27.0000    36.0000    45.0000
    2.3333    4.6667    7.0000    9.3333    11.6667
    1.0000    2.0000    3.0000    4.0000    5.0000
    0.4286    0.8571    1.2857    1.7143    2.1429
    0.1111    0.2222    0.3333    0.4444    0.5556

V =
    90.0000   180.0000   270.0000   360.0000   450.0000
    7.7778   15.5556   23.3333   31.1111   38.8889
    2.0000    4.0000    6.0000    8.0000   10.0000
```

# nbinstat

---

0.6122	1.2245	1.8367	2.4490	3.0612
0.1235	0.2469	0.3704	0.4938	0.6173

## See Also

[nbinpdf](#) | [nbincdf](#) | [nbininv](#) | [nbinfit](#) | [nbinrnd](#)

## How To

- “Negative Binomial Distribution” on page B-98



**Purpose**

Noncentral  $F$  cumulative distribution function

**Syntax**

```
p = ncfcdf(x,nu1,nu2,delta)
p = ncfcdf(x,nu1,nu2,delta,'upper')
```

**Description**

`p = ncfcdf(x,nu1,nu2,delta)` computes the noncentral  $F$  cdf at each value in `x` using the corresponding numerator degrees of freedom in `nu1`, denominator degrees of freedom in `nu2`, and positive noncentrality parameters in `delta`. `nu1`, `nu2`, and `delta` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `p`. A scalar input for `x`, `nu1`, `nu2`, or `delta` is expanded to a constant array with the same dimensions as the other inputs.

`p = ncfcdf(x,nu1,nu2,delta,'upper')` returns the complement of the noncentral  $F$  cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The noncentral  $F$  cdf is

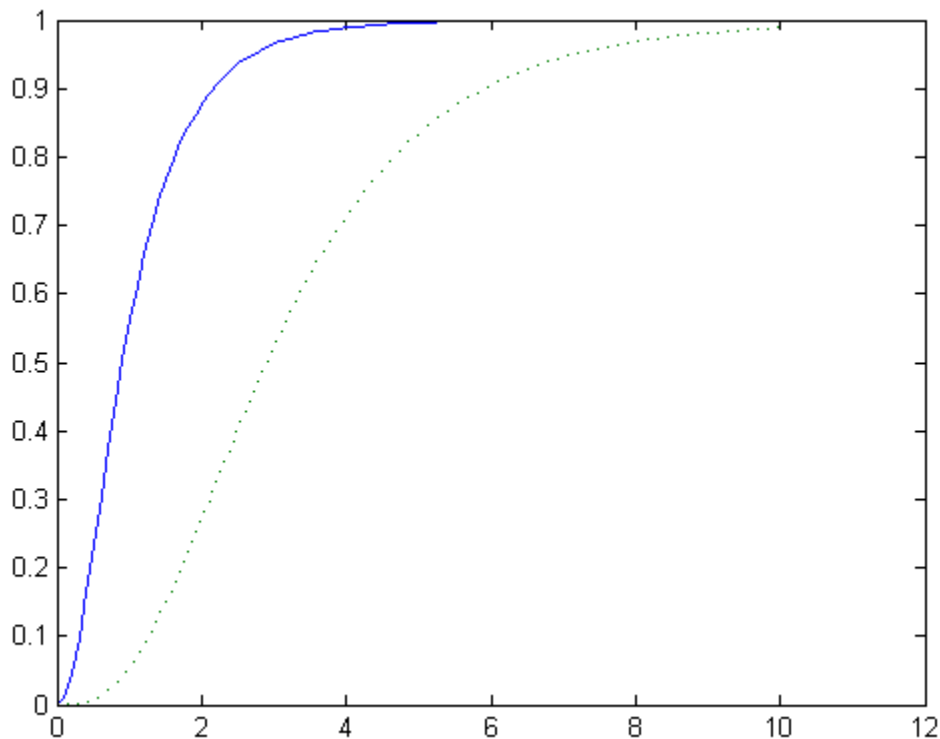
$$F(x | v_1, v_2, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) I\left(\frac{v_1 \cdot x}{v_2 + v_1 \cdot x} \middle| \frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

where  $I(x | a, b)$  is the incomplete beta function with parameters  $a$  and  $b$ .

**Examples****Compute Noncentral F Distribution cdf**

Compare the noncentral  $F$  cdf with  $\delta = 10$  to the  $F$  cdf with the same number of numerator and denominator degrees of freedom (5 and 20 respectively).

```
x = (0.01:0.1:10.01)';
p1 = ncfcdf(x,5,20,10);
p = fcdf(x,5,20);
plot(x,p,'-',x,p1,'-')
```



## References

[1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## See Also

[cdf](#) | [ncfpdf](#) | [ncfinv](#) | [ncfstat](#) | [ncfrnd](#)

## How To

- “Noncentral F Distribution” on page B-106

<b>Purpose</b>	Noncentral $F$ inverse cumulative distribution function
<b>Syntax</b>	$X = \text{ncfinv}(P, \text{NU1}, \text{NU2}, \text{DELTA})$
<b>Description</b>	$X = \text{ncfinv}(P, \text{NU1}, \text{NU2}, \text{DELTA})$ returns the inverse of the noncentral $F$ cdf with numerator degrees of freedom NU1, denominator degrees of freedom NU2, and positive noncentrality parameter DELTA for the corresponding probabilities in P. P, NU1, NU2, and DELTA can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of X. A scalar input for P, NU1, NU2, or DELTA is expanded to a constant array with the same dimensions as the other inputs.
<b>Examples</b>	<p>One hypothesis test for comparing two sample variances is to take their ratio and compare it to an <math>F</math> distribution. If the numerator and denominator degrees of freedom are 5 and 20 respectively, then you reject the hypothesis that the first variance is equal to the second variance if their ratio is less than that computed below.</p> <pre>critical = finv(0.95,5,20) critical =     2.7109</pre> <p>Suppose the truth is that the first variance is twice as big as the second variance. How likely is it that you would detect this difference?</p> <pre>prob = 1 - nfcdf(critical,5,20,2) prob =     0.1297</pre> <p>If the true ratio of variances is 2, what is the typical (median) value you would expect for the <math>F</math> statistic?</p> <pre>ncfinv(0.5,5,20,2) ans =     1.2786</pre>

## References

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## See Also

icdf | ncfcdf | ncfpdf | ncfstat | ncfrnd

## How To

• “Noncentral F Distribution” on page B-106

**Purpose** Noncentral  $F$  probability density function

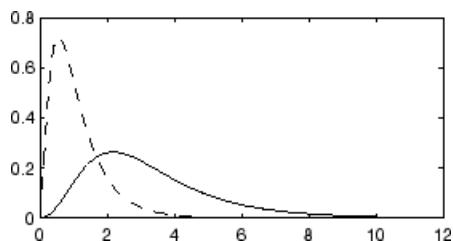
**Syntax**  $Y = \text{ncfpdf}(X, \text{NU1}, \text{NU2}, \text{DELTA})$

**Description**  $Y = \text{ncfpdf}(X, \text{NU1}, \text{NU2}, \text{DELTA})$  computes the noncentral  $F$  pdf at each of the values in  $X$  using the corresponding numerator degrees of freedom in  $\text{NU1}$ , denominator degrees of freedom in  $\text{NU2}$ , and positive noncentrality parameters in  $\text{DELTA}$ .  $X$ ,  $\text{NU1}$ ,  $\text{NU2}$ , and  $\text{DELTA}$  can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of  $Y$ . A scalar input for  $\text{NU1}$ ,  $\text{NU2}$ , or  $\text{DELTA}$  is expanded to a constant array with the same dimensions as the other inputs.

The  $F$  distribution is a special case of the noncentral  $F$  where  $\delta = 0$ . As  $\delta$  increases, the distribution flattens like the plot in the example.

**Examples** Compare the noncentral  $F$  pdf with  $\delta = 10$  to the  $F$  pdf with the same number of numerator and denominator degrees of freedom (5 and 20 respectively).

```
x = (0.01:0.1:10.01)';
p1 = ncfpdf(x,5,20,10);
p = fpdf(x,5,20);
plot(x,p,'-',x,p1,'-')
```



**References** [1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

**See Also** pdf | ncfcdf | ncfinv | ncfstat | ncfrnd

## **How To**

- “Noncentral F Distribution” on page B-106

**Purpose**

Noncentral  $F$  random numbers

**Syntax**

```
R = ncfrnd(NU1,NU2,DELTA)
R = ncfrnd(NU1,NU2,DELTA,m,n,...)
R = ncfrnd(NU1,NU2,DELTA,[m,n,...])
```

**Description**

`R = ncfrnd(NU1,NU2,DELTA)` returns a matrix of random numbers chosen from the noncentral  $F$  distribution with corresponding numerator degrees of freedom in `NU1`, denominator degrees of freedom in `NU2`, and positive noncentrality parameters in `DELTA`. `NU1`, `NU2`, and `DELTA` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `NU1`, `NU2`, or `DELTA` is expanded to a constant matrix with the same dimensions as the other inputs.

`R = ncfrnd(NU1,NU2,DELTA,m,n,...)` or `R = ncfrnd(NU1,NU2,DELTA,[m,n,...])` generates an `m`-by-`n`-by-... array. The `NU1`, `NU2`, `DELTA` parameters can each be scalars or arrays of the same size as `R`.

**Examples**

Compute six random numbers from a noncentral  $F$  distribution with 10 numerator degrees of freedom, 100 denominator degrees of freedom and a noncentrality parameter,  $\delta$ , of 4.0. Compare this to the  $F$  distribution with the same degrees of freedom.

```
r = ncfrnd(10,100,4,1,6)
r =
    2.5995    0.8824    0.8220    1.4485    1.4415    1.4864
```

```
r1 = frnd(10,100,1,6)
r1 =
    0.9826    0.5911    1.0967    0.9681    2.0096    0.6598
```

**References**

[1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

# ncfrnd

---

## **See Also**

random | ncfpdf | ncfcdf | ncfinv | ncfstat

## **How To**

- “Noncentral F Distribution” on page B-106



**Purpose**

Noncentral  $F$  mean and variance

**Syntax**

$[M, V] = \text{ncfstat}(NU1, NU2, DELTA)$

**Description**

$[M, V] = \text{ncfstat}(NU1, NU2, DELTA)$  returns the mean of and variance for the noncentral  $F$  pdf with corresponding numerator degrees of freedom in  $NU1$ , denominator degrees of freedom in  $NU2$ , and positive noncentrality parameters in  $DELTA$ .  $NU1$ ,  $NU2$ , and  $DELTA$  can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $NU1$ ,  $NU2$ , or  $DELTA$  is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral  $F$  distribution with parameters  $\nu_1$ ,  $\nu_2$ , and  $\delta$  is

$$\frac{\nu_2(\delta + \nu_1)}{\nu_1(\nu_2 - 2)}$$

where  $\nu_2 > 2$ .

The variance is

$$2 \left( \frac{\nu_2}{\nu_1} \right)^2 \left[ \frac{(\delta + \nu_1)^2 + (2\delta + \nu_1)(\nu_2 - 2)}{(\nu_2 - 2)^2(\nu_2 - 4)} \right]$$

where  $\nu_2 > 4$ .

**Examples**

```
[m,v]= ncfstat(10,100,4)
m =
    1.4286
v =
    0.4252
```

**References**

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 73–74.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 189–200.

## See Also

[ncfpdf](#) | [ncfcdf](#) | [ncfinv](#) | [ncfrnd](#)

## How To

- “Noncentral F Distribution” on page B-106

# NaiveBayes.NClasses property

---

**Purpose**

Number of classes

**Description**

The NClasses property specifies the number of classes in the grouping variable used to create the Naive Bayes classifier.

# gmdistribution.NComponents property

---

**Purpose**            Number  $k$  of mixture components

**Description**      The number  $k$  of mixture components.

**Purpose** Noncentral  $t$  cumulative distribution function

**Syntax**

```
p = nctcdf(x,nu,delta)
p = nctcdf(x,nu,delta,'upper')
```

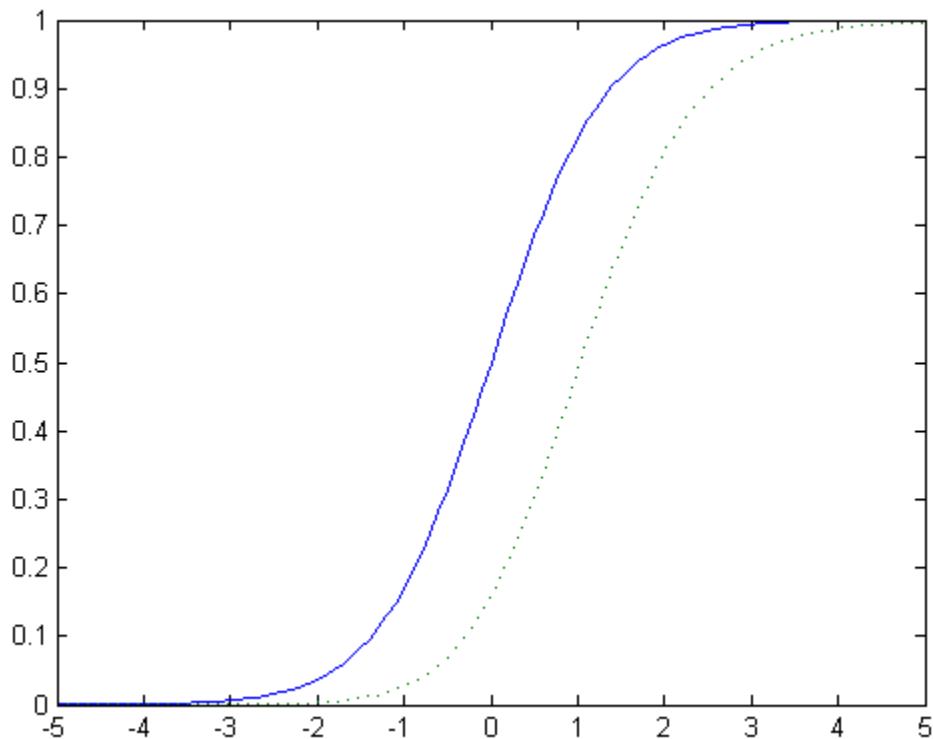
**Description** `p = nctcdf(x,nu,delta)` computes the noncentral  $t$  cdf at each value in `x` using the corresponding degrees of freedom in `nu` and noncentrality parameters in `delta`. `x`, `nu`, and `delta` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `p`. A scalar input for `x`, `nu`, or `delta` is expanded to a constant array with the same dimensions as the other inputs.

`p = nctcdf(x,nu,delta,'upper')` returns the complement of the noncentral  $t$  cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

### Examples **Compute Noncentral $t$ Distribution cdf**

Compare the noncentral  $t$  cdf with `DELTA = 1` to the  $t$  cdf with the same number of degrees of freedom (10).

```
x = (-5:0.1:5)';
p1 = nctcdf(x,10,1);
p = tcdf(x,10);
plot(x,p,'-',x,p1,':')
```



## References

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

## See Also

`cdf` | `nctpdf` | `nctinv` | `nctstat` | `nctrnd`

**How To**

- “Noncentral t Distribution” on page B-108

# nctinv

---

**Purpose** Noncentral  $t$  inverse cumulative distribution function

**Syntax** `X = nctinv(P,NU,DELTA)`

**Description** `X = nctinv(P,NU,DELTA)` returns the inverse of the noncentral  $t$  cdf with `NU` degrees of freedom and noncentrality parameter `DELTA` for the corresponding probabilities in `P`. `P`, `NU`, and `DELTA` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `X`. A scalar input for `P`, `NU`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

**Examples**

```
x = nctinv([0.1 0.2],10,1)
x =
    -0.2914    0.1618
```

**References**

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

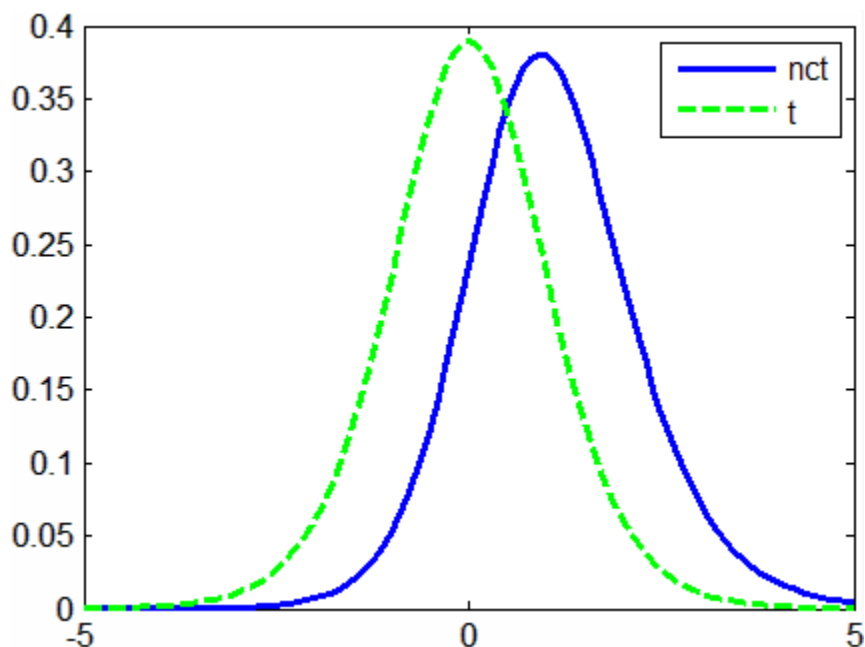
**See Also** `icdf` | `nctcdf` | `nctpdf` | `nctstat` | `nctrnd`

**How To**

- “Noncentral  $t$  Distribution” on page B-108



<b>Purpose</b>	Noncentral $t$ probability density function
<b>Syntax</b>	$Y = \text{nctpdf}(X,V,DELTA)$
<b>Description</b>	$Y = \text{nctpdf}(X,V,DELTA)$ computes the noncentral $t$ pdf at each of the values in $X$ using the corresponding degrees of freedom in $V$ and noncentrality parameters in $DELTA$ . Vector or matrix inputs for $X$ , $V$ , and $DELTA$ must have the same size, which is also the size of $Y$ . A scalar input for $X$ , $V$ , or $DELTA$ is expanded to a constant matrix with the same dimensions as the other inputs.
<b>Examples</b>	<p>Compare the noncentral <math>t</math> pdf with <math>DELTA = 1</math> to the <math>t</math> pdf with the same number of degrees of freedom (10):</p> <pre>x = (-5:0.1:5)'; nct = nctpdf(x,10,1); t = tpdf(x,10);  plot(x,nct,'b-','LineWidth',2) hold on plot(x,t,'g--','LineWidth',2) legend('nct','t')</pre>



## References

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

## See Also

pdf | nctcdf | nctinv | nctstat | nctrnd

## How To

- “Noncentral t Distribution” on page B-108

<b>Purpose</b>	Noncentral $t$ random numbers
<b>Syntax</b>	$R = \text{nctrnd}(V, \text{DELTA})$ $R = \text{nctrnd}(V, \text{DELTA}, m, n, \dots)$ $R = \text{nctrnd}(V, \text{DELTA}, [m, n, \dots])$
<b>Description</b>	<p><math>R = \text{nctrnd}(V, \text{DELTA})</math> returns a matrix of random numbers chosen from the noncentral <math>T</math> distribution using the corresponding degrees of freedom in <math>V</math> and noncentrality parameters in <math>\text{DELTA}</math>. <math>V</math> and <math>\text{DELTA}</math> can be vectors, matrices, or multidimensional arrays. A scalar input for <math>V</math> or <math>\text{DELTA}</math> is expanded to a constant array with the same dimensions as the other input.</p> <p><math>R = \text{nctrnd}(V, \text{DELTA}, m, n, \dots)</math> or <math>R = \text{nctrnd}(V, \text{DELTA}, [m, n, \dots])</math> generates an <math>m</math>-by-<math>n</math>-by-... array. The <math>V</math>, <math>\text{DELTA}</math> parameters can each be scalars or arrays of the same size as <math>R</math>.</p>
<b>Examples</b>	<pre>nctrnd(10, 1, 5, 1) ans =     1.6576     1.0617     1.4491     0.2930     3.6297</pre>
<b>References</b>	<p>[1] Evans, M., N. Hastings, and B. Peacock. <i>Statistical Distributions</i>. 2nd ed., Hoboken, NJ: John Wiley &amp; Sons, Inc., 1993, pp. 147–148.</p> <p>[2] Johnson, N., and S. Kotz. <i>Distributions in Statistics: Continuous Univariate Distributions-2</i>. Hoboken, NJ: John Wiley &amp; Sons, Inc., 1970, pp. 201–219.</p>
<b>See Also</b>	random   nctpdf   nctcdf   nctinv   nctstat
<b>How To</b>	• “Noncentral $t$ Distribution” on page B-108

# nctstat

---

**Purpose** Noncentral  $t$  mean and variance

**Syntax** `[M,V] = nctstat(NU,DELTA)`

**Description** `[M,V] = nctstat(NU,DELTA)` returns the mean of and variance for the noncentral  $t$  pdf with  $NU$  degrees of freedom and noncentrality parameter  $DELTA$ .  $NU$  and  $DELTA$  can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of  $M$  and  $V$ . A scalar input for  $NU$  or  $DELTA$  is expanded to a constant array with the same dimensions as the other input.

The mean of the noncentral  $t$  distribution with parameters  $\nu$  and  $\delta$  is

$$\frac{\delta(\nu/2)^{1/2}\Gamma((\nu-1)/2)}{\Gamma(\nu/2)}$$

where  $\nu > 1$ .

The variance is

$$\frac{\nu}{(\nu-2)}(1+\delta^2) - \frac{\nu}{2}\delta^2 \left[ \frac{\Gamma((\nu-1)/2)}{\Gamma(\nu/2)} \right]^2$$

where  $\nu > 2$ .

**Examples** `[m,v] = nctstat(10,1)`

`m =`  
`1.0837`

`v =`  
`1.3255`

**References** [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 147–148.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 201–219.

**See Also**

nctpdf | nctcdf | nctinv | nctrnd

**How To**

- “Noncentral t Distribution” on page B-108

**Purpose** Noncentral chi-square cumulative distribution function

**Syntax**  
`p = ncx2cdf(x,v,delta)`  
`p = ncx2cdf(x,v,delta,'upper')`

**Description** `p = ncx2cdf(x,v,delta)` computes the noncentral chi-square cdf at each value in `x` using the corresponding degrees of freedom in `v` and positive noncentrality parameters in `delta`. `x`, `v`, and `delta` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `p`. A scalar input for `x`, `v`, or `delta` is expanded to a constant array with the same dimensions as the other inputs.

`p = ncx2cdf(x,v,delta,'upper')` returns the complement of the noncentral chi-square cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

The noncentral chi-square cdf is

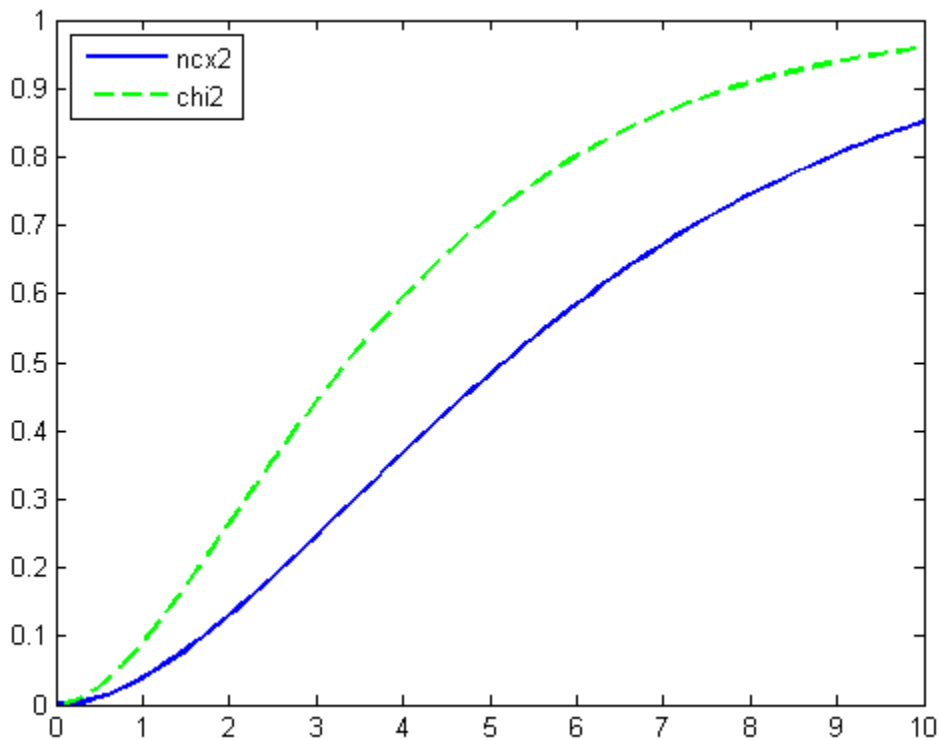
$$F(x | v, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) \Pr[\chi_{v+2j}^2 \leq x]$$

## Examples **Compute Noncentral Chi-Square cdf**

Compare the noncentral chi-square cdf with `DELTA = 2` to the chi-square cdf with the same number of degrees of freedom (4):

```
x = (0:0.1:10)';  
ncx2 = ncx2cdf(x,4,2);  
chi2 = chi2cdf(x,4);  
  
plot(x,ncx2,'b-','LineWidth',2)
```

```
hold on
plot(x,chi2,'g--','LineWidth',2)
legend('ncx2','chi2','Location','NW')
```



## References

[1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

## See Also

[cdf](#) | [ncx2pdf](#) | [ncx2inv](#) | [ncx2stat](#) | [ncx2rnd](#)

## How To

- “Noncentral Chi-Square Distribution” on page B-104



<b>Purpose</b>	Noncentral chi-square inverse cumulative distribution function
<b>Syntax</b>	<code>X = ncx2inv(P,V,DELTA)</code>
<b>Description</b>	<code>X = ncx2inv(P,V,DELTA)</code> returns the inverse of the noncentral chi-square cdf using the corresponding degrees of freedom in <code>V</code> and positive noncentrality parameters in <code>DELTA</code> , at the corresponding probabilities in <code>P</code> . <code>P</code> , <code>V</code> , and <code>DELTA</code> can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of <code>X</code> . A scalar input for <code>P</code> , <code>V</code> , or <code>DELTA</code> is expanded to a constant array with the same dimensions as the other inputs.
<b>Algorithms</b>	<code>ncx2inv</code> uses Newton's method to converge to the solution.
<b>Examples</b>	<pre>ncx2inv([0.01 0.05 0.1],4,2) ans =     0.4858    1.1498    1.7066</pre>
<b>References</b>	<p>[1] Evans, M., N. Hastings, and B. Peacock. <i>Statistical Distributions</i>. 2nd ed., Hoboken, NJ: John Wiley &amp; Sons, Inc., 1993, pp. 50–52.</p> <p>[2] Johnson, N., and S. Kotz. <i>Distributions in Statistics: Continuous Univariate Distributions-2</i>. Hoboken, NJ: John Wiley &amp; Sons, Inc., 1970, pp. 130–148.</p>
<b>See Also</b>	<code>icdf</code>   <code>ncx2cdf</code>   <code>ncx2pdf</code>   <code>ncx2stat</code>   <code>ncx2rnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Noncentral Chi-Square Distribution” on page B-104</li></ul>

**Purpose** Noncentral chi-square probability density function

**Syntax** `Y = ncx2pdf(X,V,DELTA)`

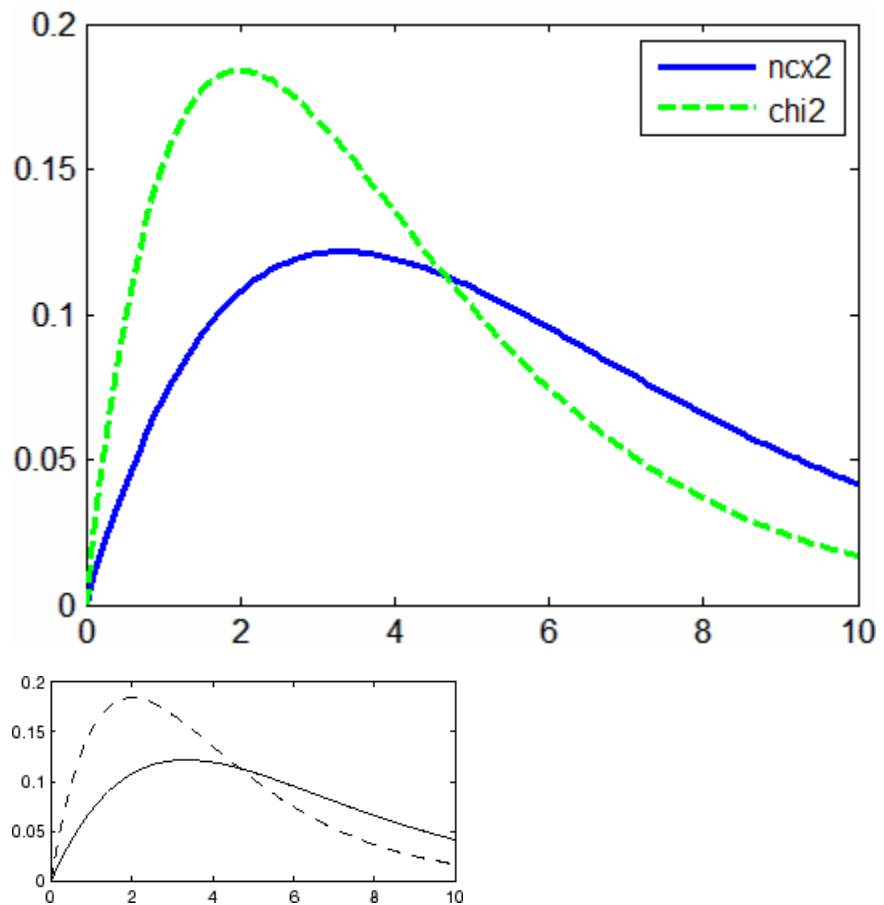
**Description** `Y = ncx2pdf(X,V,DELTA)` computes the noncentral chi-square pdf at each of the values in `X` using the corresponding degrees of freedom in `V` and positive noncentrality parameters in `DELTA`. Vector or matrix inputs for `X`, `V`, and `DELTA` must have the same size, which is also the size of `Y`. A scalar input for `X`, `V`, or `DELTA` is expanded to a constant array with the same dimensions as the other inputs.

Some texts refer to this distribution as the generalized Rayleigh, Rayleigh-Rice, or Rice distribution.

**Examples** Compare the noncentral chi-square pdf with `DELTA = 2` to the chi-square pdf with the same number of degrees of freedom (4):

```
x = (0:0.1:10)';
ncx2 = ncx2pdf(x,4,2);
chi2 = chi2pdf(x,4);

plot(x,ncx2,'b-','LineWidth',2)
hold on
plot(x,chi2,'g--','LineWidth',2)
legend('ncx2','chi2')
```



## References

[1] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

## See Also

[pdf](#) | [ncx2cdf](#) | [ncx2inv](#) | [ncx2stat](#) | [ncx2rnd](#)

## How To

- “Noncentral Chi-Square Distribution” on page B-104

# ncx2rnd

---

**Purpose** Noncentral chi-square random numbers

**Syntax**  
R = ncx2rnd(V,DELTA)  
R = ncx2rnd(V,DELTA,m,n,...)  
R = ncx2rnd(V,DELTA,[m,n,...])

**Description** R = ncx2rnd(V,DELTA) returns a matrix of random numbers chosen from the noncentral chi-square distribution using the corresponding degrees of freedom in V and positive noncentrality parameters in DELTA. V and DELTA can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of R. A scalar input for V or DELTA is expanded to a constant array with the same dimensions as the other input.

R = ncx2rnd(V,DELTA,m,n,...) or R = ncx2rnd(V,DELTA,[m,n,...]) generates an m-by-n-by-... array. The V, DELTA parameters can each be scalars or arrays of the same size as R.

**Examples**

```
ncx2rnd(4,2,6,3)
ans =
    6.8552    5.9650   11.2961
    5.2631    4.2640    5.9495
    9.1939    6.7162    3.8315
   10.3100    4.4828    7.1653
    2.1142    1.9826    4.6400
    3.8852    5.3999    0.9282
```

**References** [1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52.

[2] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148.

**See Also** random | ncx2pdf | ncx2cdf | ncx2inv | ncx2stat

**How To** • “Noncentral Chi-Square Distribution” on page B-104

<b>Purpose</b>	Noncentral chi-square mean and variance
<b>Syntax</b>	$[M, V] = \text{ncx2stat}(NU, DELTA)$
<b>Description</b>	<p><math>[M, V] = \text{ncx2stat}(NU, DELTA)</math> returns the mean of and variance for the noncentral chi-square pdf with <math>NU</math> degrees of freedom and noncentrality parameter <math>DELTA</math>. <math>NU</math> and <math>DELTA</math> can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of <math>M</math> and <math>V</math>. A scalar input for <math>NU</math> or <math>DELTA</math> is expanded to a constant array with the same dimensions as the other input.</p> <p>The mean of the noncentral chi-square distribution with parameters <math>v</math> and <math>\delta</math> is <math>v+\delta</math>, and the variance is <math>2(v+2\delta)</math>.</p>
<b>Examples</b>	<pre>[m, v] = ncx2stat(4, 2) m =     6 v =    16</pre>
<b>References</b>	<p>[1] Evans, M., N. Hastings, and B. Peacock. <i>Statistical Distributions</i>. 2nd ed., Hoboken, NJ: John Wiley &amp; Sons, Inc., 1993, pp. 50–52.</p> <p>[2] Johnson, N., and S. Kotz. <i>Distributions in Statistics: Continuous Univariate Distributions-2</i>. Hoboken, NJ: John Wiley &amp; Sons, Inc., 1970, pp. 130–148.</p>
<b>See Also</b>	<a href="#">ncx2pdf</a>   <a href="#">ncx2cdf</a>   <a href="#">ncx2inv</a>   <a href="#">ncx2rnd</a>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Noncentral Chi-Square Distribution” on page B-104</li></ul>

# categorical.ndim

---

**Purpose**            Number of dimensions of categorical array

**Syntax**            `n = ndims(A)`

**Description**        `n = ndims(A)` returns the number of dimensions in the categorical array `A`. The number of dimensions in an array is always greater than or equal to 2. Trailing singleton dimensions are ignored. Put simply, `ndims(A)` is `length(size(A))`.

**See Also**            `size`

# gmdistribution.NDimensions property

---

**Purpose** Dimension  $d$  of multivariate Gaussian distributions

**Description** The dimension  $d$  of the multivariate Gaussian distributions.

# dataset.ndims

---

**Purpose**            Number of dimensions of dataset array

**Syntax**            `n = ndims(A)`

**Description**        `n = ndims(A)` returns the number of dimensions in the dataset A. The number of dimensions in an array is always 2.

**See Also**            `size`



**Purpose** Number of dimensions in matrix

**Syntax** `n = ndims(p)`

**Description** `n = ndims(p)` returns the number of dimensions in the matrix that is created by the syntax `p(:, :)`. Since this is always a 2-D matrix, `n` is always equal to 2.

**See Also** `grandset` | `size`

# NaiveBayes.NDims property

---

**Purpose**            Number of dimensions

**Description**        The NDims property specifies the number of dimensions, which is equal to the number of features in the training data used to create the Naive Bayes classifier.

---

<b>Purpose</b>	Not equal relation for handles
<b>Syntax</b>	<code>h1 ~= h2</code>
<b>Description</b>	<p>Handles are equal if they are handles for the same object and are unequal otherwise.</p> <p><code>h1 ~= h2</code> performs element-wise comparisons between handle arrays <code>h1</code> and <code>h2</code>. <code>h1</code> and <code>h2</code> must be of the same dimensions unless one is a scalar. The result is a logical array of the same dimensions, where each element is an element-wise <code>~=</code> result.</p> <p>If one of <code>h1</code> or <code>h2</code> is scalar, scalar expansion is performed and the result will match the dimensions of the array that is not scalar.</p> <p><code>tf = ne(h1, h2)</code> stores the result in a logical array of the same dimensions.</p>
<b>See Also</b>	<code>grandstream</code>   <code>eq</code>   <code>ge</code>   <code>gt</code>   <code>le</code>   <code>lt</code>

# prob.NegativeBinomialDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Negative binomial distribution object

**Description** prob.NegativeBinomialDistribution is an object consisting of parameters, a model description, and sample data for a negative binomial probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('NegativeBinomial')` creates a negative binomial probability distribution object using the default parameter values.

`pd = makedist('NegativeBinomial', 'R', r, 'p', p)` creates a negative binomial probability distribution object using the specified parameter values.

## Input Arguments

### **r - Number of successes**

1 (default) | positive scalar value

Number of successes for the negative binomial distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **p - Probability of success**

0.5 (default) | positive scalar value in the range (0,1]

Probability of success of any individual trial for the negative binomial distribution, specified as a positive scalar value in the range (0,1].

### **Data Types**

single | double

## Properties

**r**

Number of successes for the negative binomial distribution, stored as a positive scalar value.

### Data Types

single | double

**p**

Probability of success of any individual trial for the negative binomial distribution, specified as a positive scalar value in the range (0,1].

### Data Types

single | double

### DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### Data Types

char

### InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

### Data Types

single | double

### IsTruncated

# prob.NegativeBinomialDistribution

---

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames`

# prob.NegativeBinomialDistribution

---

array is not fixed. If 1, the corresponding parameter in the ParameterNames array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object

# prob.NegativeBinomialDistribution

---

median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Negative Binomial Distribution

The negative binomial distribution models the number of failures  $x$  before a specified number of successes is reached in a series of independent, identical trials. This distribution can also model count data, in which case  $r$  does not need to be an integer value.

The negative binomial distribution uses the following parameters.



# prob.NegativeBinomialDistribution

Parameter	Description	Support
R	Number of successes	$r > 0$
p	Probability of success	$0 < p \leq 1$

The probability density function (pdf) when  $r$  is an integer is

$$f(x|r,p) = \binom{r+x-1}{x} p^r q^x \quad ; \quad x = 1, 2, \dots, \infty,$$

where  $q = 1 - p$  and  $\binom{r+x-1}{x}$  is the binomial coefficient.

When  $r$  is not an integer, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

$$\frac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)},$$

where  $\Gamma(\cdot)$  is the Gamma function.

## Examples

### Create a Negative Binomial Distribution Object Using Default Parameters

Create a negative binomial distribution object using the default parameter values.

```
pd = makedist('NegativeBinomial')
```

```
pd =
```

```
NegativeBinomialDistribution
```

```
Negative Binomial distribution
```

```
R = 1
```

```
P = 0.5
```

# prob.NegativeBinomialDistribution

---

## Create a Negative Binomial Distribution Object Using Specified Parameters

Create a negative binomial distribution object by specifying the parameter values.

```
pd = makedist('NegativeBinomial','r',5,'p',.1)
```

```
pd =
```

```
    NegativeBinomialDistribution
```

```
    Negative Binomial distribution
```

```
    R =    5
```

```
    P = 0.1
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
    45
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Negative Binomial Distribution” on page B-98
- Class Attributes
- Property Attributes

<b>Purpose</b>	Negative loglikelihood
<b>Syntax</b>	<code>nll = negloglik(pd)</code>
<b>Description</b>	<code>nll = negloglik(pd)</code> returns the value of the negative log likelihood function for the data used to fit the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Fit a probability distribution object to data using <code>fitdist</code> or the Distribution Fitting app.
<b>Output Arguments</b>	<b>nll - Negative log likelihood</b> scalar value  Negative log likelihood value for the data used to fit the distribution, returned as a scalar value.
<b>Examples</b>	<b>Negative Loglikelihood for a Kernel Distribution</b>  Load the sample data. Fit a kernel distribution to the miles per gallon (MPG) data.  <pre>load carsmall; pd = fitdist(MPG,'Kernel')</pre> <pre>pd =  KernelDistribution  Kernel = normal Bandwidth = 4.11428 Support = unbounded</pre> Compute the negative loglikelihood.

# prob.KernelDistribution.negloglik

---

```
nll = negloglik(pd)
```

```
nll =
```

```
327.3139
```

## **See Also**

```
fitdist | dfittool
```

# prob.ToolboxFittableParametricDistribution.negloglik

<b>Purpose</b>	Negative loglikelihood of probability distribution object
<b>Syntax</b>	<code>nll = negloglik(pd)</code>
<b>Description</b>	<code>nll = negloglik(pd)</code> returns the value of the negative loglikelihood function for the data used to fit the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code> . Alternatively, create a probability distribution object by fitting it to data using <code>fitdist</code> or the Distribution Fitting app.
<b>Output Arguments</b>	<b>nll - Negative log likelihood</b> scalar value  Negative log likelihood value for the data used to fit the distribution, returned as a scalar value.
<b>Examples</b>	<b>Negative Log Likelihood for a Fitted Distribution</b>  Load the sample data.  <code>load carsmall;</code>  Create a Weibull distribution object by fitting it to the mile per gallon (MPG) data.  <code>pd = fitdist(MPG,'Weibull')</code>  <code>pd =</code>  <code>WeibullDistribution</code>  <code>Weibull distribution</code>

# prob.ToolboxFittableParametricDistribution.negloglik

---

```
A = 26.5079 [24.8333, 28.2954]
B = 3.27193 [2.79441, 3.83104]
```

Compute the negative log likelihood for the fitted Weibull distribution.

```
wll = negloglik(p)
```

```
wll =
```

```
327.4942
```

## See Also

```
makedist | fitdist | dfittool
```

**Purpose** Nearest neighbor search object

**Description** NeighborSearcher is an abstract class used for nearest neighbor search. You cannot create instances of this class directly. Instead, create an instance of a derived class such as ExhaustiveSearcher or KDTreeSearcher either by calling the derived class constructor or by calling the function createns.

**Construction** NeighborSearcher is an abstract class. You cannot create instances of this class directly. You can construct an object in a subclass, such as KDTreeSearcher or ExhaustiveSearcher, either by calling the subclass constructors or by using the createns function.

**Properties** **x**

A matrix used to create the object.

**Distance**

A string specifying a built-in distance metric (applies to both ExhaustiveSearcher and KDTreeSearcher) or a function handle (only applies to ExhaustiveSearcher) that you provide when you create the object. This property is the default distance metric used when you call the knnsearch method to find nearest neighbors for future query points.

**DistParameter**

Specifies the additional parameter for the chosen distance metric. The value is:

- If 'Distance' is 'minkowski': A positive scalar indicating the exponent of the Minkowski distance. (Applies for both ExhaustiveSearcher and KDTreeSearcher.)
- If 'Distance' is 'mahalanobis': A positive definite matrix representing the covariance matrix used for computing the Mahalanobis distance. (Only applies for ExhaustiveSearcher.)

# NeighborSearcher

---

- If 'Distance' is 'seuclidean': A vector representing the scale value of the data when computing the 'seuclidean' distance. (Only applies for ExhaustiveSearcher.)
- Otherwise: Empty.

## See Also

`createns` | `KDTreeSearcher` | `ExhaustiveSearcher`



<b>Purpose</b>	Generate quasi-random point set
<b>Syntax</b>	<code>X = net(p,n)</code>
<b>Description</b>	<p><code>X = net(p,n)</code> returns the first <math>n</math> points <math>X</math> from the point set <math>p</math> of the <code>grandset</code> class. <math>X</math> is <math>n</math>-by-<math>d</math>, where <math>d</math> is the dimension of the point set.</p> <p>Objects <math>p</math> of the <code>@grandset</code> class encapsulate properties of a specified quasi-random sequence. Values of the point set are not generated and stored in memory until <math>p</math> is accessed using <code>net</code> or parenthesis indexing.</p>
<b>Examples</b>	<p>Use <code>haltonset</code> to generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:</p> <pre>p = haltonset(3,'Skip',1e3,'Leap',1e2) p =     Halton point set in 3 dimensions (8.918019e+013 points)     Properties:         Skip : 1000         Leap : 100         ScrambleMethod : none</pre> <p>Use <code>scramble</code> to apply reverse-radix scrambling:</p> <pre>p = scramble(p,'RR2') p =     Halton point set in 3 dimensions (8.918019e+013 points)     Properties:         Skip : 1000         Leap : 100         ScrambleMethod : RR2</pre> <p>Use <code>net</code> to generate the first four points:</p> <pre>X0 = net(p,4) X0 =     0.0928    0.6950    0.0029     0.6958    0.2958    0.8269</pre>

```
0.3013    0.6497    0.4141
0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

## See Also

[haltonset](#) | [sobolset](#) | [grandstream](#)

# CompactClassificationDiscriminant.nLinearCoeffs

---

<b>Purpose</b>	Number of nonzero linear coefficients
<b>Syntax</b>	<code>ncoeffs = nLinearCoeffs(obj)</code> <code>ncoeffs = nLinearCoeffs(obj,delta)</code>
<b>Description</b>	<code>ncoeffs = nLinearCoeffs(obj)</code> returns the number of nonzero linear coefficients in the linear discriminant model <code>obj</code> . <code>ncoeffs = nLinearCoeffs(obj,delta)</code> returns the number of nonzero linear coefficients for threshold parameter <code>delta</code> .
<b>Input Arguments</b>	<b>obj</b> Discriminant analysis classifier, produced using <code>ClassificationDiscriminant.fit</code> . <b>delta</b> Scalar or vector value of the Delta parameter. See “Gamma and Delta” on page 21-2139.
<b>Output Arguments</b>	<b>ncoeffs</b> Nonnegative integer, the number of nonzero coefficients in the discriminant analysis model <code>obj</code> . If you call <code>nLinearCoeffs</code> with a <code>delta</code> argument, <code>ncoeffs</code> is the number of nonzero linear coefficients for threshold parameter <code>delta</code> . If <code>delta</code> is a vector, <code>ncoeffs</code> is a vector with the same number of elements. If <code>obj</code> is a quadratic discriminant model, <code>ncoeffs</code> is the number of predictors in <code>obj</code> .
<b>Definitions</b>	<b>Gamma and Delta</b> Regularization is the process of finding a small set of predictors that yield an effective predictive model. For linear discriminant analysis, there are two parameters, $\gamma$ and $\delta$ , that control regularization as follows. <code>cvshrink</code> helps you select appropriate values of the parameters.

# Compact Classification Discriminant.nLinear Coeffs

---

Let  $\Sigma$  represent the covariance matrix of the data  $X$ , and let  $\hat{X}$  be the centered data (the data  $X$  minus the mean by class). Define

$$D = \text{diag}(\hat{X}^T * \hat{X}).$$

The regularized covariance matrix  $\tilde{\Sigma}$  is

$$\tilde{\Sigma} = (1 - \gamma)\Sigma + \gamma D.$$

Whenever  $\gamma \geq \text{MinGamma}$ ,  $\tilde{\Sigma}$  is nonsingular.

Let  $\mu_k$  be the mean vector for those elements of  $X$  in class  $k$ , and let  $\mu_0$  be the global mean vector (the mean of the rows of  $X$ ). Let  $C$  be the correlation matrix of the data  $X$ , and let  $\tilde{C}$  be the regularized correlation matrix:

$$\tilde{C} = (1 - \gamma)C + \gamma I,$$

where  $I$  is the identity matrix.

The linear term in the regularized discriminant analysis classifier for a data point  $x$  is

$$(x - \mu_0)^T \tilde{\Sigma}^{-1} (\mu_k - \mu_0) = [(x - \mu_0)^T D^{-1/2}] [\tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0)].$$

The parameter  $\delta$  enters into this equation as a threshold on the final term in square brackets. Each component of the vector

$[\tilde{C}^{-1} D^{-1/2} (\mu_k - \mu_0)]$  is set to zero if it is smaller in magnitude than the threshold  $\delta$ . Therefore, for class  $k$ , if component  $j$  is thresholded to zero, component  $j$  of  $x$  does not enter into the evaluation of the posterior probability.

The `DeltaPredictor` property is a vector related to this threshold. When  $\delta \geq \text{DeltaPredictor}(i)$ , all classes  $k$  have

$$|\tilde{C}^{-1}D^{-1/2}(\mu_k - \mu_0)| \leq \delta.$$

Therefore, when  $\delta \geq \text{DeltaPredictor}(i)$ , the regularized classifier does not use predictor  $i$ .

## Examples

### Find the Number of Nonzero Coefficients in a Discriminant Analysis Classifier

Find the number of nonzero coefficients in a discriminant analysis classifier for various `Delta` values.

Create a discriminant analysis classifier from the `fishseriris` data.

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
```

Find the number of nonzero coefficients in `obj`.

```
ncoeffs = nLinearCoeffs(obj)
```

```
ncoeffs =
```

```
4
```

Find the number of nonzero coefficients for `delta = 1, 2, 4, and 8`.

```
delta = [1 2 4 8];
ncoeffs = nLinearCoeffs(obj,delta)
```

```
ncoeffs =
```

```
4
```

```
4
```

```
3
```

```
0
```

The `DeltaPredictor` property gives the values of `delta` where the number of nonzero coefficients changes.

# CompactClassificationDiscriminant.nLinearCoeffs

---

```
ncoeffs2 = nLinearCoeffs(obj,obj.DeltaPredictor)
```

```
ncoeffs2 =
```

```
4
```

```
3
```

```
1
```

```
2
```

## See Also

[CompactClassificationDiscriminant | cvshrink | ClassificationDiscriminant.fit |](#)

## Concepts

- “Discriminant Analysis” on page 15-3

**Purpose**

Nonlinear regression

**Syntax**

```
beta = nlinfit(X,Y,modelfun,beta0)
beta = nlinfit(X,Y,modelfun,beta0,options)
beta = nlinfit(___,Name,Value)
[beta,R,J,CovB,MSE,ErrorModelInfo] = nlinfit(___)
```

**Description**

`beta = nlinfit(X,Y,modelfun,beta0)` returns a vector of estimated coefficients for the nonlinear regression of the responses in `Y` on the predictors in `X` using the model specified by `modelfun`. The coefficients are estimated using iterative least squares estimation, with initial values specified by `beta0`.

`beta = nlinfit(X,Y,modelfun,beta0,options)` fits the nonlinear regression using the algorithm control parameters in the structure `options`. You can return any of the output arguments in the previous syntaxes.

`beta = nlinfit(___,Name,Value)` uses additional options specified by one or more name-value pair arguments. For example, you can specify observation weights or a nonconstant error model. You can use any of the input arguments in the previous syntaxes.

`[beta,R,J,CovB,MSE,ErrorModelInfo] = nlinfit(___)` additionally returns the residuals, `R`, the Jacobian of `modelfun`, `J`, the estimated variance-covariance matrix for the estimated coefficients, `CovB`, an estimate of the variance of the error term, `MSE`, and a structure containing details about the error model, `ErrorModelInfo`.

**Input Arguments****X - Predictor variables**

matrix

Predictor variables for the nonlinear regression function, specified as a matrix. Typically, `X` is a design matrix of predictor (independent

variable) values, with one row for each value in Y, and one column for each coefficient. However, X can be any array that `modelfun` can accept.

### Data Types

`single` | `double`

### Y - Response values

vector

Response values (dependent variable) for fitting the nonlinear regression function, specified as a vector with the same number of rows as X.

### Data Types

`single` | `double`

### modelfun - Nonlinear regression model function

function handle

Nonlinear regression model function, specified as a function handle. `modelfun` must accept two input arguments, a coefficient vector and an array X—in that order—and return a vector of fitted response values.

For example, to specify the hougen nonlinear regression function, use the function handle `@hougen`.

### Data Types

function\_handle

### beta0 - Initial coefficient values

vector

Initial coefficient values for the least squares estimation algorithm, specified as a vector.

---

**Note** Poor starting values can lead to a solution with large residual error.

---



## Data Types

single | double

## options - Estimation algorithm options

structure created using `statset`

Estimation algorithm options, specified as a structure you create using `statset`. The following `statset` parameters are applicable to `nlinfit`.

## DerivStep - Relative difference for finite difference gradient

$\text{eps}^{(1/3)}$  (default) | positive scalar value | vector

Relative difference for the finite difference gradient calculation, specified as a positive scalar value, or a vector the same size as `beta`. Use a vector to specify a different relative difference for each coefficient.

## Display - Level of output display

'off' (default) | 'iter' | 'final'

Level of output display during estimation, specified as one of 'off', 'iter', or 'final'. If you specify 'iter', output is displayed at each iteration. If you specify 'final', output is displayed after the final iteration.

## FunValCheck - Indicator for whether to check for invalid values

'on' (default) | 'off'

Indicator for whether to check for invalid values such as NaN or Inf from the objective function, specified as 'on' or 'off'.

## MaxIter - Maximum number of iterations

100 (default) | positive integer

Maximum number of iterations for the estimation algorithm, specified as a positive integer. Iterations continue until estimates are within the convergence tolerance, or the maximum number of iterations specified by `MaxIter` is reached.

## RobustWgtFun - Weight function

string | function handle | []

Weight function for robust fitting, specified as a valid string or function handle.

**Note** RobustWgtFun must have value [] when you use observation weights, W.

The following table describes the possible string values. Let  $r$  denote normalized residuals and  $w$  denote robust weights. The indicator function  $I[x]$  is equal to 1 if the expression  $x$  is true, and 0 otherwise.

Weight Function	Equation	Default Tuning Constant
' ' (default)	No robust fitting	—
'andrews'	$w = I[ r  < \pi] \times \sin(r) / r$	1.339
'bisquare'	$w = I[ r  < 1] \times (1 - r^2)^2$	4.685
'cauchy'	$w = \frac{1}{(1 + r^2)}$	2.385
'fair'	$w = \frac{1}{(1 +  r )}$	1.400

Weight Function	Equation	Default Tuning Constant
'huber'	$w = \frac{1}{\max(1,  r )}$	1.345
'logistic'	$w = \frac{\tanh(r)}{r}$	1.205
'talwar'	$w = I[ r  < 1]$	2.795
'welsch'	$w = \exp\{-r^2\}$	2.985

You can alternatively specify a function handle that accepts a vector of normalized residuals as input, and returns a vector of robust weights as output. If you use a function handle, you must provide a `Tune` constant.

### **Tune - Tuning constant**

positive scalar value

Tuning constant for robust fitting, specified as a positive scalar value. The tuning constant is used to normalize residuals before applying a robust weight function. The default tuning constant depends on the function specified by `RobustWgtFun`.

If you use a function handle to specify `RobustWgtFun`, then you must specify a value for `Tune`.

### **TolFun - Termination tolerance on residual sum of squares**

1e-8 (default) | positive scalar value

Termination tolerance for the residual sum of squares, specified as a positive scalar value. Iterations continue until estimates are within the

convergence tolerance, or the maximum number of iterations specified by `MaxIter` is reached.

### **TolX - Termination tolerance on estimated coefficients**

1e-8 (default) | positive scalar value

Termination tolerance on the estimated coefficients, `beta`, specified as a positive scalar value. Iterations continue until estimates are within the convergence tolerance, or the maximum number of iterations specified by `MaxIter` is reached.

### **Robust - Indicator for robust fitting**

'off' (default) | 'on'

Indicator for robust fitting, specified as 'off' or 'on'.

---

**Note** Robust will be removed in a future software release. Use `RobustWgtFun` for robust fitting.

---

### **WgtFun - Weight function for robust fitting**

string | function handle

Weight function for robust fitting, specified as a string indicating a weight function, or a function handle. `WgtFun` is valid only when `Robust` has value 'on'.

---

**Note** `WgtFun` will be removed in a future software release. Use `RobustWgtFun` instead.

---

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can

specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'ErrorModel', 'proportional', 'ErrorParameters', 0.5` specifies a proportional error model, with initial value 0.5 for the error parameter estimation

### 'ErrorModel' - Form of error term

`'constant'` (default) | `'proportional'` | `'combined'`

Form of the error term, specified as the comma-separated pair consisting of `'ErrorModel'` and one of the following strings indicating the error model. Each model defines the error using a standard mean-zero and unit-variance variable  $e$  in combination with independent components: the function value  $f$ , and one or two parameters  $a$  and  $b$ .

<code>'constant'</code> (default)	$y = f + ae$
<code>'proportional'</code>	$y = f + bfe$
<code>'combined'</code>	$y = f + (a + b f )e$

The only allowed error model when using `Weights` is `'constant'`.

---

**Note** `options.RobustWgtFun` must have value `[]` when using an error model other than `'constant'`.

---

### 'ErrorParameters' - Initial estimates for error model parameters

1 or `[1, 1]` (default) | scalar value | two-element vector

Initial estimates for the error model parameters in the chosen `ErrorModel`, specified as the comma-separated pair consisting of `'ErrorParameters'` and a scalar value or two-element vector.

Error Model	Parameters	Default Values
'constant'	$a$	1
'proportional'	$b$	1
'combined'	$a, b$	[1,1]

For example, if 'ErrorModel' has the value 'combined', you can specify the starting value 1 for  $a$  and the starting value 2 for  $b$  as follows.

**Example:** 'ErrorParameters', [1,2]

You can only use the 'constant' error model when using **Weights**.

---

**Note** options.RobustWgtFun must have value [] when using an error model other than 'constant'.

---

## Data Types

double | single

## 'Weights' - Observation weights

vector | function handle

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of real positive weights or a function handle. You can use observation weights to down-weight the observations that you want to have less influence on the fitted model.

- If  $W$  is a vector, then it must be the same size as  $Y$ .
- If  $W$  is a function handle, then it must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

---

**Note** options.RobustWgtFun must have value [] when you use observation weights.

---

## Output Arguments

### Data Types

double | single | function\_handle

### beta - Estimated regression coefficients

vector

Estimated regression coefficients, returned as a vector. The number of elements in `beta` equals the number of elements in `beta0`.

Let  $f(X_i, \mathbf{b})$  denote the nonlinear function specified by `modelfun`, where  $\mathbf{x}_i$  are the predictors for observation  $i$ ,  $i = 1, \dots, N$ , and  $\mathbf{b}$  are the regression coefficients. The vector of coefficients returned in `beta` minimizes the weighted least squares equation,

$$\sum_{i=1}^N w_i [y_i - f(\mathbf{x}_i, \mathbf{b})]^2.$$

For unweighted nonlinear regression, all of the weight terms are equal to 1.

### R - Residuals

vector

Residuals for the fitted model, returned as a vector.

- If you specify observation weights using the name-value pair argument `Weights`, then `R` contains weighted residuals.
- If you specify an error model other than 'constant' using the name-value pair argument `ErrorModel`, then you can no longer interpret `R` as model fit residuals.

### J - Jacobian

matrix

Jacobian of the nonlinear regression model, `modelfun`, returned as an  $N$ -by- $p$  matrix, where  $N$  is the number of observations and  $p$  is the number of estimated coefficients.

- If you specify observation weights using the name-value pair argument `Weights`, then `J` is the weighted model function Jacobian.
- If you specify an error model other than 'constant' using the name-value pair argument `ErrorModel`, then you can no longer interpret `J` as the model function Jacobian.

## **CovB - Estimated variance-covariance matrix**

matrix

Estimated variance-covariance matrix for the fitted coefficients, `beta`, returned as a  $p$ -by- $p$  matrix, where  $p$  is the number of estimated coefficients. If the model Jacobian, `J`, has full column rank, then  $\text{CovB} = \text{inv}(J' * J) * \text{MSE}$ , where `MSE` is the mean squared error.

## **MSE - Mean squared error**

scalar value

Mean squared error (MSE) of the fitted model, returned as a scalar value. MSE is an estimate of the variance of the error term. If the model Jacobian, `J`, has full column rank, then  $\text{MSE} = (R' * R) / (N - p)$ , where  $N$  is the number of observations, and  $p$  is the number of estimated coefficients.

## **ErrorModelInfo - Information about error model fit**

structure

Information about the error model fit, returned as a structure with the following fields:

<code>ErrorModel</code>	Chosen error model
<code>ErrorParameters</code>	Estimated error parameters
<code>ErrorVariance</code>	Function handle that accepts an $N$ -by- $p$ matrix, <code>X</code> , and returns an $N$ -by-1 vector of error variances using the estimated error model



MSE	Mean squared error
ScheffeSimPred	Scheffé parameter for simultaneous prediction intervals when using the estimated error model
WeightFunction	Logical with value true if you used a custom weight function previously in nlinfit
FixedWeights	Logical with value true if you used fixed weights previously in nlinfit
RobustWeightFunction	Logical with value true if you used robust fitting previously in

## Examples

### Nonlinear Regression Model Using Default Options

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the initial values in beta0.

```
beta = nlinfit(X,y,@hougen,beta0)
```

```
beta =  
  
    1.2526  
    0.0628  
    0.0400  
    0.1124  
    1.1914
```

## Nonlinear Regression Using Robust Options

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp\{-b_3 x\} + \varepsilon,$$

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.1.

```
modelfun = @(b,x)(b(1)+b(2)*exp(-b(3)*x));
```

```
rng('default') % for reproducibility  
b = [1;3;2];  
x = exprnd(2,100,1);  
y = modelfun(b,x) + normrnd(0,0.1,100,1);
```

Set robust fitting options.

```
opts = statset('nlinfit');  
opts.RobustWgtFun = 'bisquare';
```

Fit the nonlinear model using the robust fitting options.

```
beta0 = [2;2;2];  
beta = nlinfit(x,y,modelfun,beta0,opts)
```

```
beta =  
  
    1.0041  
    3.0997  
    2.1483
```

## Nonlinear Regression Using Observation Weights

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;
```

```
beta0 = S.beta;
```

Specify a vector of known observation weights.

```
W = [8 2 1 6 12 9 12 10 10 12 2 10 8]';
```

Fit the Hougen-Watson model to the rate data using the specified observation weights.

```
[beta,R,J,CovB] = nlinfit(X,y,@hougen,beta0,'Weights',W);  
beta
```

```
beta =  
  
    2.2068  
    0.1077  
    0.0766  
    0.1818  
    0.6516
```

Display the coefficient standard errors.

```
sqrt(diag(CovB))
```

```
ans =  
  
    2.5721  
    0.1251  
    0.0950  
    0.2043  
    0.7735
```

## **Nonlinear Regression Using Weights Function Handle**

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;
```

```
beta0 = S.beta;
```

Specify a function handle for observation weights. The function accepts the model fitted values as input, and returns a vector of weights.

```
a = 1; b = 1;  
weights = @(yhat) 1./((a + b*abs(yhat)).^2);
```

Fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
[beta,R,J,CovB] = nlinfit(X,y,@hougen,beta0,'Weights',weights);  
beta
```

```
beta =  
  
    0.8308  
    0.0409  
    0.0251  
    0.0801  
    1.8261
```

Display the coefficient standard errors.

```
sqrt(diag(CovB))  
  
ans =  
  
    0.5822  
    0.0297  
    0.0197  
    0.0578  
    1.2810
```

## **Nonlinear Regression Using Nonconstant Error Model**

Load sample data.

```
S = load('reaction');
```

```
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error model.

```
[beta,R,J,CovB,MSE,ErrorModelInfo] = nlinfit(X,y,@hougen,beta0,'ErrorModelInfo',beta)
```

```
beta =
```

```
    1.2526
    0.0628
    0.0400
    0.1124
    1.1914
```

Display the error model information.

```
ErrorModelInfo
```

```
ErrorModelInfo =
```

```
    ErrorModel: 'combined'
    ErrorParameters: [0.1517 5.6783e-08]
    ErrorVariance: @(x)mse*(errorparam(1)+errorparam(2)*abs(mod
    MSE: 1.6245
    ScheffeSimPred: 6
    WeightFunction: 0
    FixedWeights: 0
    RobustWeightFunction: 0
```

## Tips

- To produce error estimates on predictions, use the optional output arguments R, J, CovB, or MSE as inputs to `nlpredci`.
- To produce error estimates on the estimated coefficients, beta, use the optional output arguments R, J, CovB, or MSE as inputs to `nlparci`.

- If you use the robust fitting option, `RobustWgtFun`, you must use `CovB`—and might need `MSE`—as inputs to `nlpredci` or `nlparci` to ensure that the confidence intervals take the robust fit properly into account.

## Algorithms

- `nlinfit` treats NaN values in `Y` or `modelfun(beta0,X)` as missing data, and ignores the corresponding observations.
- For nonrobust estimation, `nlinfit` uses the Levenberg-Marquardt nonlinear least squares algorithm [1].
- For robust estimation, `nlinfit` uses an iterative reweighted least squares algorithm ([2], [3]). At each iteration, the robust weights are recalculated based on each observation's residual from the previous iteration. These weights downweight outliers, so that their influence on the fit is decreased. Iterations continue until the weights converge.
- When you specify a function handle for observation weights, the weights depend on the fitted model. In this case, `nlinfit` uses an iterative generalized least squares algorithm to fit the nonlinear regression model.

## Definitions

### Weighted Residuals

A *weighted residual* is a residual multiplied by the square root of the corresponding observation weight.

Given estimated regression coefficients,  $\mathbf{b}$ , the residual for observation  $i$  is

$$r_i = y_i - f(\mathbf{x}_i, \mathbf{b}),$$

where  $y_i$  is the observed response and  $f(\mathbf{x}_i, \mathbf{b})$  is the fitted response at predictors  $\mathbf{x}_i$ .

When you fit a weighted nonlinear regression with weights  $w_i$ ,  $i = 1, \dots, N$ , `nlinfit` returns the weighted residuals,

$$r_i^* = \sqrt{w_i} (y_i - f(\mathbf{x}_i, \mathbf{b})).$$

## Weighted Model Function Jacobian

The *weighted model function Jacobian* is the nonlinear model Jacobian multiplied by the square root of the observation weight matrix.

Given estimated regression coefficients,  $\mathbf{b}$ , the estimated model Jacobian,  $\mathbf{J}$ , for the nonlinear function  $f(\mathbf{x}_i, \mathbf{b})$  has elements

$$\mathbf{J}_{ij} = \frac{\partial f(\mathbf{x}_i, \mathbf{b})}{\partial b_j},$$

where  $b_j$  is the  $j$ th element of  $\mathbf{b}$ .

When you fit a weighted nonlinear regression with diagonal weights matrix  $\mathbf{W}$ , nlinfit returns the weighted Jacobian matrix,

$$\mathbf{J}^* = \mathbf{W}^{1/2}\mathbf{J}.$$

## References

- [1] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.
- [2] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.
- [3] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.

## See Also

fitnlm | nlparci | nlpredci | nlintool

# nlinfit

---

## **Concepts**

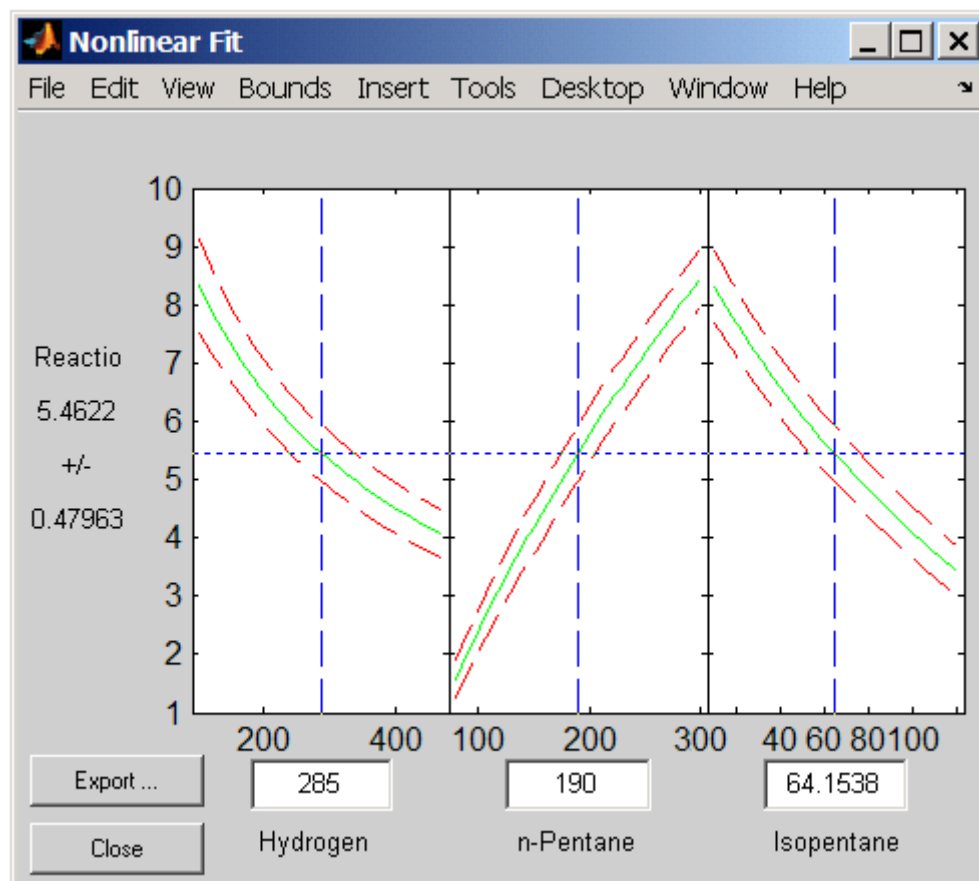
- “Nonlinear Regression” on page 11-2



---

<b>Purpose</b>	Interactive nonlinear regression
<b>Syntax</b>	<pre>nlintool(X,y,fun,beta0) nlintool(X,y,fun,beta0,alpha) nlintool(X,y,fun,beta0,alpha,'xname','yname')</pre>
<b>Description</b>	<p><code>nlintool(X,y,fun,beta0)</code> is a graphical user interface to the <code>nlinfit</code> function, and uses the same input arguments. The interface displays plots of the fitted response against each predictor, with the other predictors held fixed. The fixed values are in the text boxes below each predictor axis. Change the fixed values by typing in a new value or by dragging the vertical lines in the plots to new positions. When you change the value of a predictor, all plots update to display the model at the new point in predictor space. Dashed red curves show 95% simultaneous confidence bands for the function.</p> <p><code>nlintool(X,y,fun,beta0,alpha)</code> shows <math>100(1-\alpha)\%</math> confidence bands. These are simultaneous confidence bounds for the function value. Using the <b>Bounds</b> menu you can switch between simultaneous and non-simultaneous bounds, and between bounds on the function and bounds for predicting a new observation.</p> <p><code>nlintool(X,y,fun,beta0,alpha,'xname','yname')</code> labels the plots using the string matrix <code>'xname'</code> for the predictors and the string <code>'yname'</code> for the response.</p>
<b>Examples</b>	<p>The data in <code>reaction.mat</code> are partial pressures of three chemical reactants and the corresponding reaction rates. The function <code>hougen</code> implements the nonlinear Hougen-Watson model for reaction rates. The following fits the model to the data:</p> <pre>load reaction nlintool(reactants,rate,@hougen,beta,0.01,xn,yn)</pre>

# nlintool



**See Also** [nlinfit](#) | [polytool](#) | [rstool](#)

**Purpose**

Nonlinear mixed-effects estimation

**Syntax**

```
beta = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI] = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI,stats] = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI,stats,B] = nlmefit(X,y,group,V,fun,beta0)
[beta,PSI,stats,B] = nlmefit(X,y,group,V,fun,beta0,'Name',
    value)
```

**Description**

`beta = nlmefit(X,y,group,V,fun,beta0)` fits a nonlinear mixed-effects regression model and returns estimates of the fixed effects in `beta`. By default, `nlmefit` fits a model in which each parameter is the sum of a fixed and a random effect, and the random effects are uncorrelated (their covariance matrix is diagonal).

`X` is an  $n$ -by- $h$  matrix of  $n$  observations on  $h$  predictors.

`y` is an  $n$ -by-1 vector of responses.

`group` is a grouping variable indicating  $m$  groups in the observations. `group` is a categorical variable, a numeric vector, a character matrix with rows for group names, or a cell array of strings. For more information on grouping variables, see “Grouping Variables” on page 2-51.

`V` is an  $m$ -by- $g$  matrix or cell array of  $g$  group-specific predictors. These are predictors that take the same value for all observations in a group. The rows of `V` are assigned to groups using `grp2idx`, according to the order specified by `grp2idx(group)`. Use a cell array for `V` if group predictors vary in size across groups. Use `[]` for `V` if there are no group-specific predictors.

`fun` is a handle to a function that accepts predictor values and model parameters and returns fitted values. `fun` has the form

```
yfit = modelfun(PHI,XFUN,VFUN)
```

The arguments are:

- `PHI` — A 1-by- $p$  vector of model parameters.

- XFUN — A  $k$ -by- $h$  array of predictors, where:
  - $k = 1$  if XFUN is a single row of X.
  - $k = n_i$  if XFUN contains the rows of X for a single group of size  $n_i$ .
  - $k = n$  if XFUN contains all rows of X.
- VFUN — Group-specific predictors given by one of:
  - A 1-by- $g$  vector corresponding to a single group and a single row of V.
  - An  $n$ -by- $g$  array, where the  $j$ th row is  $V(I,:)$  if the  $j$ th observation is in group I.

If V is empty, nlmefit calls modelfun with only two inputs.

- yfit — A  $k$ -by-1 vector of fitted values

When either PHI or VFUN contains a single row, it corresponds to all rows in the other two input arguments.

---

**Note** If modelfun can compute yfit for more than one vector of model parameters per call, use the 'Vectorization' parameter (described later) for improved performance.

---

beta0 is a  $q$ -by-1 vector with initial estimates for  $q$  fixed effects. By default,  $q$  is the number of model parameters  $p$ .

nlmefit fits the model by maximizing an approximation to the marginal likelihood with random effects integrated out, assuming that:

- Random effects are multivariate normally distributed and independent between groups.
- Observation errors are independent, identically normally distributed, and independent of the random effects.

`[beta,PSI] = nlmefit(X,y,group,V,fun,beta0)` also returns `PSI`, an  $r$ -by- $r$  estimated covariance matrix for the random effects. By default,  $r$  is equal to the number of model parameters  $p$ .

`[beta,PSI,stats] = nlmefit(X,y,group,V,fun,beta0)` also returns `stats`, a structure with fields:

- `dfe` — The error degrees of freedom for the model
- `logl` — The maximized loglikelihood for the fitted model
- `rmse` — The square root of the estimated error variance (computed on the log scale for the exponential error model)
- `errorparam` — The estimated parameters of the error variance model
- `aic` — The Akaike information criterion, calculated as  $\text{aic} = -2 * \text{logl} + 2 * \text{numParam}$ , where `numParam` is the number of fitting parameters, including the degree of freedom for covariance matrix of the random effects, the number of fixed effects and the number of parameters of the error model, and `logl` is a field in the `stats` structure
- `bic` — The Bayesian information criterion, calculated as  $\text{bic} = -2 * \text{logl} + \log(M) * \text{numParam}$ 
  - `M` is the number of groups.
  - `numParam` and `logl` are defined as in `aic`.Note that some literature suggests that the computation of `bic` should be ,  $\text{bic} = -2 * \text{logl} + \log(N) * \text{numParam}$ , where `N` is the number of observations.
- `covb` — The estimated covariance matrix of the parameter estimates
- `sebeta` — The standard errors for `beta`
- `ires` — The population residuals ( $y - y_{\text{population}}$ ), where `y_population` is the individual predicted values
- `pres` — The population residuals ( $y - y_{\text{population}}$ ), where `y_population` is the population predicted values
- `iwres` — The individual weighted residuals

- `pwres` — The population weighted residuals
- `cwres` — The conditional weighted residuals

`[beta,PSI,stats,B] = nlmefit(X,y,group,V,fun,beta0)` also returns `B`, an  $r$ -by- $m$  matrix of estimated random effects for the  $m$  groups. By default,  $r$  is equal to the number of model parameters  $p$ .

`[beta,PSI,stats,B] = nlmefit(X,y,group,V,fun,beta0,'Name',value)` specifies one or more optional parameter name/value pairs. Specify *Name* inside single quotes.

Use the following parameters to fit a model different from the default. (The default model is obtained by setting both `FEConstDesign` and `REConstDesign` to `eye(p)`, or by setting both `FEPARAMSSelect` and `REPARAMSSelect` to `1:p`.) Use at most one parameter with an 'FE' prefix and one parameter with an 'RE' prefix. The `nlmefit` function requires you to specify at least one fixed effect and one random effect.

Parameter	Value
<code>FEPARAMSSelect</code>	A vector specifying which elements of the parameter vector <code>PHI</code> include a fixed effect, given as a numeric vector of indices from 1 to $p$ or as a 1-by- $p$ logical vector. If $q$ is the specified number of elements, then the model includes $q$ fixed effects.
<code>FEConstDesign</code>	A $p$ -by- $q$ design matrix <code>ADESIGN</code> , where <code>ADESIGN*beta</code> are the fixed components of the $p$ elements of <code>PHI</code> .
<code>FEGroupDesign</code>	A $p$ -by- $q$ -by- $m$ array specifying a different $p$ -by- $q$ fixed-effects design matrix for each of the $m$ groups.
<code>FEObsDesign</code>	A $p$ -by- $q$ -by- $n$ array specifying a different $p$ -by- $q$ fixed-effects design matrix for each of the $n$ observations.

Parameter	Value
REParamsSelect	A vector specifying which elements of the parameter vector PHI include a random effect, given as a numeric vector of indices from 1 to $p$ or as a 1-by- $p$ logical vector. The model includes $r$ random effects, where $r$ is the specified number of elements.
REConstDesign	A $p$ -by- $r$ design matrix BDESIGN, where BDESIGN*B are the random components of the $p$ elements of PHI.
REGroupDesign	A $p$ -by- $r$ -by- $m$ array specifying a different $p$ -by- $r$ random-effects design matrix for each of $m$ groups.
REObsDesign	A $p$ -by- $r$ -by- $n$ array specifying a different $p$ -by- $r$ random-effects design matrix for each of $n$ observations.

Use the following parameters to control the iterative algorithm for maximizing the likelihood:

Parameter	Value
RefineBeta0	Determines whether nlmefit makes an initial refinement of beta0 by first fitting modelfun without random effects and replacing beta0 with beta. Choices are 'on' and 'off'. The default value is 'on'.
ErrorModel	A string specifying the form of the error term. Default is 'constant'. Each model defines the error using a standard normal (Gaussian) variable $e$ , the function value $f$ , and one or two parameters $a$ and $b$ . Choices are: <ul style="list-style-type: none"> <li>'constant': <math>y = f + a*e</math></li> </ul>

Parameter	Value
	<ul style="list-style-type: none"> <li>• 'proportional': <math>y = f + b \cdot f^a \cdot e</math></li> <li>• 'combined': <math>y = f + (a + b \cdot f)^a \cdot e</math></li> <li>• 'exponential': <math>y = f \cdot \exp(a \cdot e)</math>, or equivalently <math>\log(y) = \log(f) + a \cdot e</math></li> </ul> <p>If this parameter is given, the output <code>stats.errorparam</code> field has the value</p> <ul style="list-style-type: none"> <li>• <math>a</math> for 'constant' and 'exponential'</li> <li>• <math>b</math> for 'proportional'</li> <li>• <math>[a \ b]</math> for 'combined'</li> </ul>
ApproximationType	<p>The method used to approximate the likelihood of the model. Choices are:</p> <ul style="list-style-type: none"> <li>• 'LME' — Use the likelihood for the linear mixed-effects model at the current conditional estimates of <code>beta</code> and <code>B</code>. This is the default.</li> <li>• 'RELME' — Use the restricted likelihood for the linear mixed-effects model at the current conditional estimates of <code>beta</code> and <code>B</code>.</li> <li>• 'FO' — First-order Laplacian approximation without random effects.</li> <li>• 'FOCE' — First-order Laplacian approximation at the conditional estimates of <code>B</code>.</li> </ul>



Parameter	Value
Vectorization	<p data-bbox="805 319 1277 409">Indicates acceptable sizes for the PHI, XFUN, and VFUN input arguments to <code>modelfun</code>. Choices are:</p> <ul data-bbox="805 447 1332 1407" style="list-style-type: none"><li data-bbox="805 447 1332 795">• 'SinglePhi' — <code>modelfun</code> can only accept a single set of model parameters at a time, so PHI must be a single row vector in each call. <code>nlmefit</code> calls <code>modelfun</code> in a loop, if necessary, with a single PHI vector and with XFUN containing rows for a single observation or group at a time. VFUN may be a single row that applies to all rows of XFUN, or a matrix with rows corresponding to rows in XFUN. This is the default.</li><li data-bbox="805 815 1332 1072">• 'SingleGroup' — <code>modelfun</code> can only accept inputs corresponding to a single group in the data, so XFUN must contain rows of X from a single group in each call. Depending on the model, PHI is a single row that applies to the entire group or a matrix with one row for each observation. VFUN is a single row.</li><li data-bbox="805 1093 1332 1407">• 'Full' — <code>modelfun</code> can accept inputs for multiple parameter vectors and multiple groups in the data. Either PHI or VFUN may be a single row that applies to all rows of XFUN or a matrix with rows corresponding to rows in XFUN. This option can improve performance by reducing the number of calls to <code>modelfun</code>, but may require <code>modelfun</code> to perform singleton expansion on PHI or V.</li></ul>

Parameter	Value
CovParameterization	Specifies the parameterization used internally for the scaled covariance matrix. Choices are 'chol' for the Cholesky factorization or 'logm' the matrix logarithm. The default is 'logm'.
CovPattern	<p>Specifies an <math>r</math>-by-<math>r</math> logical or numeric matrix <math>P</math> that defines the pattern of the random-effects covariance matrix <math>\Psi</math>. <code>nlmefit</code> estimates the variances along the diagonal of <math>\Psi</math> and the covariances specified by nonzeros in the off-diagonal elements of <math>P</math>. Covariances corresponding to zero off-diagonal elements in <math>P</math> are constrained to be zero. If <math>P</math> does not specify a row-column permutation of a block diagonal matrix, <code>nlmefit</code> adds nonzero elements to <math>P</math> as needed. The default value of <math>P</math> is <math>\text{eye}(r)</math>, corresponding to uncorrelated random effects.</p> <p>Alternatively, <math>P</math> may be a 1-by-<math>r</math> vector containing values in <math>1:r</math>, with equal values specifying groups of random effects. In this case, <code>nlmefit</code> estimates covariances only within groups, and constrains covariances across groups to be zero.</p>

Parameter	Value
ParamTransform	<p>A vector of <math>p</math>-values specifying a transformation function <math>f()</math> for each of the <math>P</math> parameters: <math>XB = ADESIGN*BETA + BDESIGN*B</math> <math>PHI = f(XB)</math>. Each element of the vector must be one of the following integer codes specifying the transformation for the corresponding value of <math>PHI</math>:</p> <ul style="list-style-type: none"> <li>• 0: <math>PHI = XB</math> (default for all parameters)</li> <li>• 1: <math>\log(PHI) = XB</math></li> <li>• 2: <math>\text{probit}(PHI) = XB</math></li> <li>• 3: <math>\text{logit}(PHI) = XB</math></li> </ul>
Options	<p>A structure of the form returned by <code>statset</code>. <code>nlmefit</code> uses the following <code>statset</code> parameters:</p> <ul style="list-style-type: none"> <li>• 'DerivStep' — Relative difference used in finite difference gradient calculation. May be a scalar, or a vector whose length is the number of model parameters <math>p</math>. The default is <math>\text{eps}^{(1/3)}</math>.</li> <li>• 'Display' — Level of iterative display during estimation. Choices are: <ul style="list-style-type: none"> <li>▪ 'off' (default) — Displays no information</li> <li>▪ 'final' — Displays information after the final iteration</li> <li>▪ 'iter' — Displays information at each iteration</li> </ul> </li> <li>• 'FunValCheck' — Check for invalid values, such as NaN or Inf, from</li> </ul>

Parameter	Value
	<p>modelfun. Choices are 'on' and 'off'. The default is 'on'.</p> <ul style="list-style-type: none"> <li>• 'MaxIter' — Maximum number of iterations allowed. The default is 200.</li> <li>• 'OutputFcn' — Function handle specified using @, a cell array with function handles or an empty array (default). The solver calls all output functions after each iteration.</li> <li>• 'TolFun' — Termination tolerance on the loglikelihood function. The default is 1e-4.</li> <li>• 'TolX' — Termination tolerance on the estimated fixed and random effects. The default is 1e-4.</li> </ul>
OptimFun	<p>Specifies the optimization function used in maximizing the likelihood. Choices are 'fminsearch' to use fminsearch or 'fminunc' to use fminunc. The default is 'fminsearch'. You can specify 'fminunc' only if Optimization Toolbox software is installed.</p>

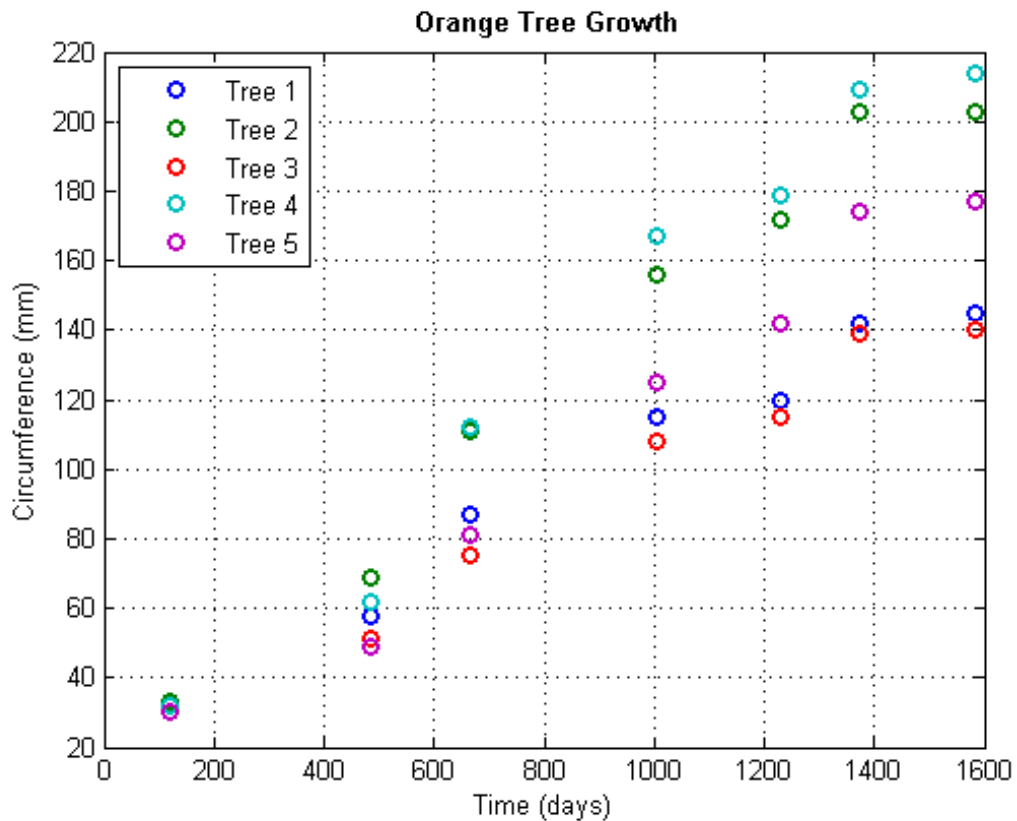
## Examples

### Nonlinear Mixed-Effects Model

Enter and display data on the growth of five orange trees.

```
CIRC = [30 58 87 115 120 142 145;
        33 69 111 156 172 203 203;
        30 51 75 108 115 139 140;
        32 62 112 167 179 209 214;
        30 49 81 125 142 174 177];
time = [118 484 664 1004 1231 1372 1582];
```

```
h = plot(time,CIRC','o','LineWidth',2);
xlabel('Time (days)')
ylabel('Circumference (mm)')
title('\bf Orange Tree Growth')
legend([repmat('Tree ',5,1),num2str((1:5)')],...
       'Location','NW')
grid on
hold on
```



Use an anonymous function to specify a logistic growth model.

```
model = @(PHI,t)(PHI(:,1))./(1+exp(-(t-PHI(:,2))./PHI(:,3))));
```

Fit the model using `nlmefit` with default settings (that is, assuming each parameter is the sum of a fixed and a random effect, with no correlation among the random effects):

```
TIME = repmat(time,5,1);  
NUMS = repmat((1:5)',size(time));  
  
beta0 = [100 100 100];  
[beta1,PSI1,stats1] = nlmefit(TIME(:),CIRC(:),NUMS(:),...  
                             [],model,beta0)
```

```
beta1 =
```

```
191.3189  
723.7608  
346.2517
```

```
PSI1 =
```

```
962.1534    0    0  
    0    0.0000    0  
    0    0    297.9879
```

```
stats1 =
```

```
    dfe: 28  
    logl: -131.5457  
    mse: 59.7882  
    rmse: 7.9016  
    errorparam: 7.7323
```

```
aic: 277.0913
bic: 274.3574
covb: [3x3 double]
sebeta: [15.2249 33.1579 26.8235]
ires: [35x1 double]
pres: [35x1 double]
iwres: [35x1 double]
pwres: [35x1 double]
cwres: [35x1 double]
```

The negligible variance of the second random effect,  $\text{PSI1}(2,2)$ , suggests that it can be removed to simplify the model.

```
[beta2,PSI2,stats2,b2] = nlmefit(TIME(:),CIRC(:),...
    NUMS(:),[],model,beta0,'REParamsSelect',[1 3])
```

```
beta2 =
```

```
191.3193
723.7627
346.2529
```

```
PSI2 =
```

```
962.4853      0
      0 297.9475
```

```
stats2 =
```

```
dfc: 29
logl: -131.5456
mse: 59.7850
rmse: 7.7642
```

```
errorparam: 7.7321
      aic: 275.0913
      bic: 272.7479
      covb: [3x3 double]
      sebeta: [15.2270 33.1574 26.8228]
      ires: [35x1 double]
      pres: [35x1 double]
      iwres: [35x1 double]
      pwres: [35x1 double]
      cwres: [35x1 double]
```

b2 =

```
-28.5263   31.6066  -36.5079   39.0750  -5.6473
   9.9967  -0.7622   6.0038  -9.4567  -5.7816
```

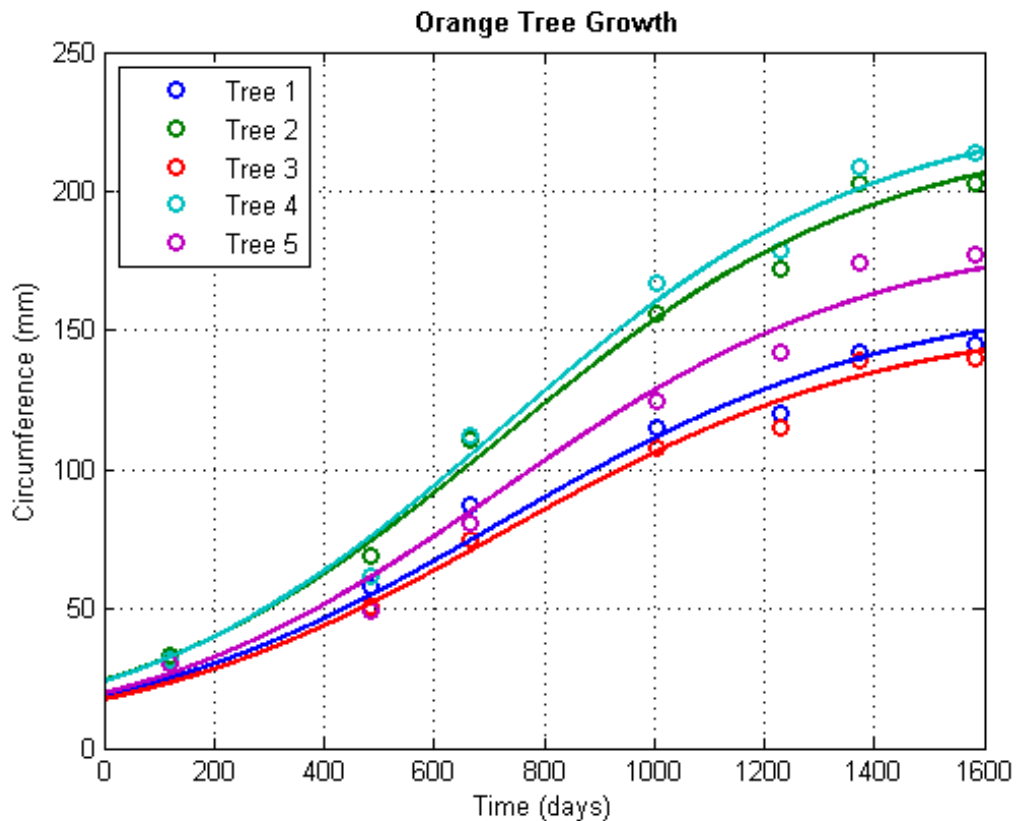
The loglikelihood `logl` is unaffected, and both the Akaike and Bayesian information criteria ( `aic` and `bic` ) are reduced, supporting the decision to drop the second random effect from the model.

Use the estimated fixed effects in `beta2` and the estimated random effects for each tree in `b2` to plot the model through the data.

```
PHI = repmat(beta2,1,5) + ...           % Fixed effects
      [b2(1,:);zeros(1,5);b2(2,:)];    % Random effects

colors = get(h,'Color');
tplot = 0:0.1:1600;
for I = 1:5
    fitted_model=@(t)(PHI(1,I))./(1+exp(-(t-PHI(2,I))./ ...
    PHI(3,I)));
    plot(tplot,fitted_model(tplot),'Color',colors{I}, ...
        'LineWidth',2)
end
```





## References

- [1] Lindstrom, M. J., and D. M. Bates. "Nonlinear mixed-effects models for repeated measures data." *Biometrics*. Vol. 46, 1990, pp. 673–687.
- [2] Davidian, M., and D. M. Giltinan. *Nonlinear Models for Repeated Measurements Data*. New York: Chapman & Hall, 1995.
- [3] Pinheiro, J. C., and D. M. Bates. "Approximations to the log-likelihood function in the nonlinear mixed-effects model." *Journal of Computational and Graphical Statistics*. Vol. 4, 1995, pp. 12–35.

[4] Demidenko, E. *Mixed Models: Theory and Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 2004.

## See Also

`nlinfit` | `nlpredci` | `nlmefitsa`

## How To

- “Mixed-Effects Models” on page 11-23
- “Grouping Variables” on page 2-51

**Purpose**

Fit nonlinear mixed-effects model with stochastic EM algorithm

**Syntax**

```
[BETA,PSI,STATS,B] = nlmefitsa(X,Y,GROUP,V,MODELFUN,BETA0)
[BETA,PSI,STATS,B] = nlmefitsa(X,Y,GROUP,V,MODELFUN,BETA0,
    'Name',Value)
```

**Description**

[BETA,PSI,STATS,B] = nlmefitsa(X,Y,GROUP,V,MODELFUN,BETA0) fits a nonlinear mixed-effects regression model and returns estimates of the fixed effects in BETA. By default, nlmefitsa fits a model where each model parameter is the sum of a corresponding fixed and random effect, and the covariance matrix of the random effects is diagonal, i.e., uncorrelated random effects.

The BETA, PSI, and other values this function returns are the result of a random (Monte Carlo) simulation designed to converge to the maximum likelihood estimates of the parameters. Because the results are random, it is advisable to examine the plot of simulation to results to be sure that the simulation has converged. It may also be helpful to run the function multiple times, using multiple starting values, or use the 'Replicates' parameter to perform multiple simulations.

[BETA,PSI,STATS,B] = nlmefitsa(X,Y,GROUP,V,MODELFUN,BETA0, 'Name',Value) accepts one or more comma-separated parameter name/value pairs. Specify *Name* inside single quotes.

**Input Arguments****Definitions:**

In the following list of arguments, the following variable definitions apply:

- *n* — number of observations
- *h* — number of predictor variables
- *m* — number of groups
- *g* — number of group-specific predictor variables
- *p* — number of parameters
- *f* — number of fixed effects

## **X**

An  $n$ -by- $h$  matrix of  $n$  observations on  $h$  predictor variables.

## **Y**

An  $n$ -by-1 vector of responses.

## **GROUP**

A grouping variable indicating to which of  $m$  groups each observation belongs. **GROUP** can be a categorical variable, a numeric vector, a character matrix with rows for group names, or a cell array of strings.

## **V**

An  $m$ -by- $g$  matrix of  $g$  group-specific predictor variables for each of the  $m$  groups in the data. These are predictor values that take on the same value for all observations in a group. Rows of **V** are ordered according to `GRP2IDX(GROUP)`. Use an  $m$ -by- $g$  cell array for **V** if any of the group-specific predictor values vary in size across groups. Specify `[]` for **V** if there are no group predictors.

## **MODELFUN**

A handle to a function that accepts predictor values and model parameters, and returns fitted values. **MODELFUN** has the form `YFIT = MODELFUN(PHI, XFUN, VFUN)` with input arguments

- **PHI** — A 1-by- $p$  vector of model parameters.
- **XFUN** — An  $l$ -by- $h$  array of predictor variables where
  - $l$  is 1 if **XFUN** is a single row of **X**
  - $l$  is  $n_i$  if **XFUN** contains the rows of **X** for a single group of size  $n_i$
  - $l$  is  $n$  if **XFUN** contains all rows of **X**.
- **VFUN** — Either
  - A 1-by- $g$  vector of group-specific predictors for a single group, corresponding to a single row of **V**

- An  $n$ -by- $g$  matrix, where the  $k$ -th row of VFUN is  $V(i,:)$  if the  $k$ -th observation is in group  $i$ .

If  $V$  is empty, nlmefitsa calls MODELFUN with only two inputs.

MODELFUN returns an  $l$ -by-1 vector of fitted values YFIT. When either PHI or VFUN contains a single row, that one row corresponds to all rows in the other two input arguments. For improved performance, use the 'Vectorization' parameter name/value pair (described below) if MODELFUN can compute YFIT for more than one vector of model parameters in one call.

### **BETA0**

An  $f$ -by-1 vector with initial estimates for the  $f$  fixed effects. By default,  $f$  is equal to the number of model parameters  $p$ . BETA0 can also be an  $f$ -by-REPS matrix, and the estimation is repeated REPS times using each column of BETA0 as a set of starting values.

### **Name-Value Pair Arguments**

By default, nlmefitsa fits a model where each model parameter is the sum of a corresponding fixed and random effect. Use the following parameter name/value pairs to fit a model with a different number of or dependence on fixed or random effects. Use at most one parameter name with an 'FE' prefix and one parameter name with an 'RE' prefix. Note that some choices change the way nlmefitsa calls MODELFUN, as described further below.

#### **'FEParamsSelect'**

A vector specifying which elements of the model parameter vector PHI include a fixed effect, as a numeric vector with elements in  $1:p$ , or as a 1-by- $p$  logical vector. The model will include  $f$  fixed effects, where  $f$  is the specified number of elements.

#### **'FEConstDesign'**

A  $p$ -by- $f$  design matrix ADESIGN, where ADESIGN\*BETA are the fixed components of the  $p$  elements of PHI.

## **'FEGroupDesign'**

A  $p$ -by- $f$ -by- $m$  array specifying a different  $p$ -by- $f$  fixed effects design matrix for each of the  $m$  groups.

## **'REParamsSelect'**

A vector specifying which elements of the model parameter vector PHI include a random effect, as a numeric vector with elements in  $1:p$ , or as a 1-by- $p$  logical vector. The model will include  $r$  random effects, where  $r$  is the specified number of elements.

## **'REConstDesign'**

A  $p$ -by- $r$  design matrix BDESIGN, where BDESIGN\*B are the random components of the  $p$  elements of PHI. This matrix must consist of 0s and 1s, with at most one 1 per row.

The default model is equivalent to setting both FEConstDesign and REConstDesign to  $\text{eye}(p)$ , or to setting both FEParamsSelect and REParamsSelect to  $1:p$ .

Additional optional parameter name/value pairs control the iterative algorithm used to maximize the likelihood:

## **'CovPattern'**

Specifies an  $r$ -by- $r$  logical or numeric matrix PAT that defines the pattern of the random effects covariance matrix PSI. nlmefitsa computes estimates for the variances along the diagonal of PSI as well as covariances that correspond to non-zeroes in the off-diagonal of PAT. nlmefitsa constrains the remaining covariances, i.e., those corresponding to off-diagonal zeroes in PAT, to be zero. PAT must be a row-column permutation of a block diagonal matrix, and nlmefitsa adds non-zero elements to PAT as needed to produce such a pattern. The default value of PAT is  $\text{eye}(r)$ , corresponding to uncorrelated random effects.

Alternatively, specify PAT as a 1-by- $r$  vector containing values in  $1:r$ . In this case, elements of PAT with equal values define groups of random

effects, `nlmefitsa` estimates covariances only within groups, and constrains covariances across groups to be zero.

#### **'Cov0'**

Initial value for the covariance matrix `PSI`. Must be an  $r$ -by- $r$  positive definite matrix. If empty, the default value depends on the values of `BETA0`.

#### **'ComputeStdErrors'**

`true` to compute standard errors for the coefficient estimates and store them in the output `STATS` structure, or `false` (default) to omit this computation.

#### **'ErrorModel'**

A string specifying the form of the error term. Default is `'constant'`. Each model defines the error using a standard normal (Gaussian) variable  $e$ , the function value  $f$ , and one or two parameters  $a$  and  $b$ . Choices are

- `'constant'` —  $y = f + a*e$
- `'proportional'` —  $y = f + b*f*e$
- `'combined'` —  $y = f + (a+b*f)*e$
- `'exponential'` —  $y = f*\exp(a*e)$ , or equivalently  $\log(y) = \log(f) + a*e$

If this parameter is given, the output `STATS.errorparam` field has the value

- $a$  for `'constant'` and `'exponential'`
- $b$  for `'proportional'`
- $[a\ b]$  for `'combined'`

#### **'ErrorParameters'**

A scalar or two-element vector specifying starting values for parameters of the error model. This specifies the  $a$ ,  $b$ , or  $[a\ b]$  values depending on the `ErrorModel` parameter.

### **'LogLikMethod'**

Specifies the method for approximating the loglikelihood. Choices are:

- 'is' — Importance sampling
- 'gq' — Gaussian quadrature
- 'lin' — Linearization
- 'none' — Omit the loglikelihood approximation (default)

### **'NBurnIn'**

Number of initial burn-in iterations during which the parameter estimates are not recomputed. Default is 5.

### **'NChains'**

Number  $c$  of "chains" simulated. Default is 1. Setting  $c > 1$  causes  $c$  simulated coefficient vectors to be computed for each group during each iteration. Default depends on the data, and is chosen to provide about 100 groups across all chains.

### **'NIterations'**

Number of iterations. This can be a scalar or a three-element vector. Controls how many iterations are performed for each of three phases of the algorithm:

- 1** simulated annealing
- 2** full step size
- 3** reduced step size



Default is [150 150 100]. A scalar is distributed across the three phases in the same proportions as the default.

### **'NMCIterations'**

Number of Markov Chain Monte Carlo (MCMC) iterations. This can be a scalar or a three-element vector. Controls how many of three different types of MCMC updates are performed during each phase of the main iteration:

- 1** full multivariate update
- 2** single coordinate update
- 3** multiple coordinate update

Default is [2 2 2]. A scalar value is treated as a three-element vector with all elements equal to the scalar.

### **'OptimFun'**

Either 'fminsearch' or 'fminunc', specifying the optimization function to be used during the estimation process. Default is 'fminsearch'. Use of 'fminunc' requires Optimization Toolbox.

### **'Options'**

A structure created by a call to `statset`. `nlmefitsa` uses the following `statset` parameters:

- 'DerivStep' — Relative difference used in finite difference gradient calculation. May be a scalar, or a vector whose length is the number of model parameters  $p$ . The default is  $\text{eps}^{(1/3)}$ .
- Display — Level of display during estimation.
  - 'off' (default) — Displays no information
  - 'final' — Displays information after the final iteration of the estimation algorithm

- 'iter' — Displays information at each iteration
- FunValCheck
  - 'on' (sdefault) — Check for invalid values (such as NaN or Inf) from MODELFUN
  - 'off' — Skip this check
- OutputFcn — Function handle specified using @, a cell array with function handles or an empty array. nlmefitsa calls all output functions after each iteration. See nlmefitoutputfcn.m (the default output function for nlmefitsa) for an example of an output function.

### **'ParamTransform'**

A vector of  $p$ -values specifying a transformation function  $f()$  for each of the  $p$  parameters:

$$XB = ADESIGN*BETA + BDESIGN*B$$
$$PHI = f(XB)$$

Each element of the vector must be one of the following integer codes specifying the transformation for the corresponding value of PHI:

- 0:  $PHI = XB$  (default for all parameters)
- 1:  $\log(PHI) = XB$
- 2:  $\text{probit}(PHI) = XB$
- 3:  $\text{logit}(PHI) = XB$

### **'Replicates'**

Number REPS of estimations to perform starting from the starting values in the vector BETA0. If BETA0 is a matrix, REPS must match the number of columns in BETA0. Default is the number of columns in BETA0.

### **'Vectorization'**

Determines the possible sizes of the PHI, XFUN, and VFUN input arguments to MODELFUN. Possible values are:

- 'SinglePhi' — MODELFUN is a function (such as an ODE solver) that can only compute YFIT for a single set of model parameters at a time, i.e., PHI must be a single row vector in each call. nlmefitsa calls MODELFUN in a loop if necessary using a single PHI vector and with XFUN containing rows for a single observation or group at a time. VFUN may be a single row that applies to all rows of XFUN, or a matrix with rows corresponding to rows in XFUN.
- 'SingleGroup' — MODELFUN can only accept inputs corresponding to a single group in the data, i.e., XFUN must contain rows of X from a single group in each call. Depending on the model, PHI is a single row that applies to the entire group, or a matrix with one row for each observation. VFUN is a single row.
- 'Full' — MODELFUN can accept inputs for multiple parameter vectors and multiple groups in the data. Either PHI or VFUN may be a single row that applies to all rows of XFUN, or a matrix with rows corresponding to rows in XFUN. Using this option can improve performance by reducing the number of calls to MODELFUN, but may require MODELFUN to perform singleton expansion on PHI or V.

The default for 'Vectorization' is 'SinglePhi'. In all cases, if V is empty, nlmefitsa calls MODELFUN with only two inputs.

## Output Arguments

### BETA

Estimates of the fixed effects

### PSI

An  $r$ -by- $r$  estimated covariance matrix for the random effects. By default,  $r$  is equal to the number of model parameters  $p$ .

### STATS

A structure with the following fields:

- logl — The maximized loglikelihood for the fitted model; empty if the LogLikMethod parameter has its default value of 'none'

- `rmse` — The square root of the estimated error variance (computed on the log scale for the exponential error model)
- `errorparam` — The estimated parameters of the error variance model
- `aic` — The Akaike information criterion (empty if `logl` is empty), calculated as  $\text{aic} = -2 * \text{logl} + 2 * \text{numParam}$ , where
  - `logl` is the maximized loglikelihood.
  - `numParam` is the number of fitting parameters, including the degree of freedom for covariance matrix of the random effects, the number of fixed effects and the number of parameters of the error model.
- `bic` — The Bayesian information criterion (empty if `logl` is empty), calculated as  $\text{bic} = -2 * \text{logl} + \log(M) * \text{numParam}$ 
  - `M` is the number of groups.
  - `logl` and `numParam` are defined as in `aic`.  
Note that some literature suggests that the computation of `bic` should be ,  $\text{bic} = -2 * \text{logl} + \log(N) * \text{numParam}$ , where `N` is the number of observations. To adjust the value of the output you can redefine `bic` as follows:  $\text{bic} = \text{bic} - \text{numel}(\text{unique}(\text{group})) + \text{numel}(Y)$
- `sebeta` — The standard errors for BETA (empty if the `ComputeStdErrors` parameter has its default value of false)
- `covb` — The estimated covariance of the parameter estimates (empty if `ComputeStdErrors` is false)
- `dfe` — The error degrees of freedom
- `pres` — The population residuals ( $y - y_{\text{population}}$ ), where `y_population` is the population predicted values
- `ires` — The population residuals ( $y - y_{\text{population}}$ ), where `y_population` is the individual predicted values
- `pwres` — The population weighted residuals
- `cwres` — The conditional weighted residuals
- `iwres` — The individual weighted residuals

**Examples****Nonlinear Mixed-Effects Model with Stochastic EM Algorithm**

Load the sample data.

```
load indomethacin
```

Fit a model to data on concentrations of the drug indomethacin in the bloodstream of six subjects over eight hours.

```
model = @(phi,t)(phi(:,1).*exp(-phi(:,2).*t)+phi(:,3).*exp(-phi(:,4).*t));
phi0 = [1 1 1 1];
xform = [0 1 0 1]; % log transform for 2nd and 4th parameters
[beta,PSI,stats,br] = nlmefitsa(time,concentration,...
    subject,[],model,phi0,'ParamTransform',xform)
```

```
beta =
```

```
    0.8563
   -0.7950
    2.7744
    1.0772
```

```
PSI =
```

```
    0.0529         0         0         0
         0    0.0220         0         0
         0         0    0.4762         0
         0         0         0    0.0120
```

```
stats =
```

```
    logl: []
     aic: []
     bic: []
  sebeta: []
```

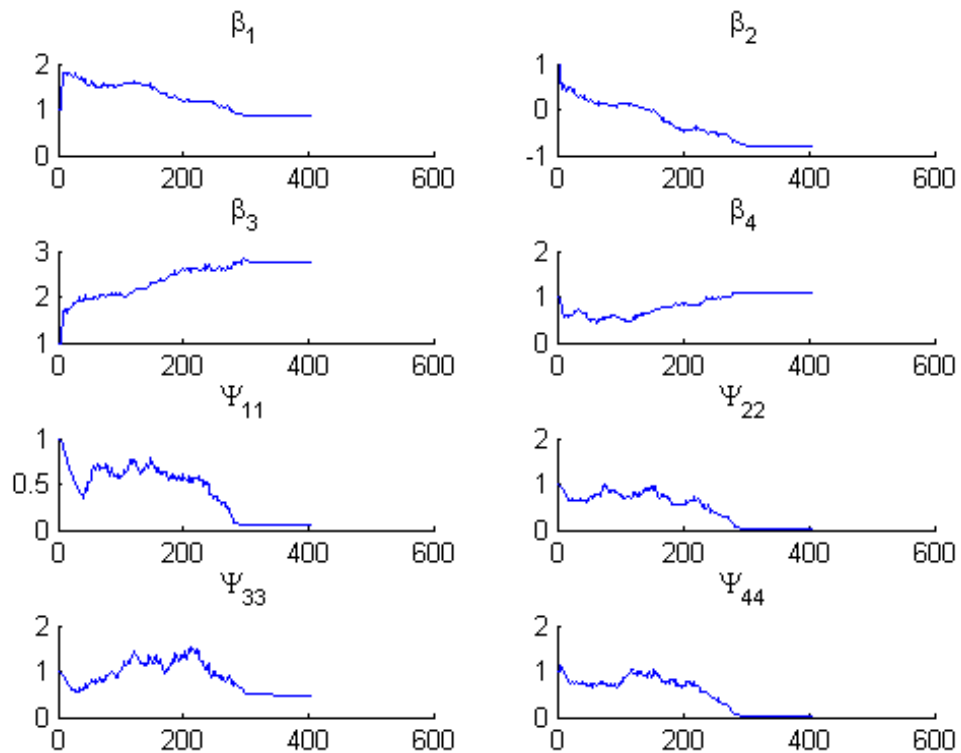
# nlmefitsa

---

```
    dfe: 57
   covb: []
errorparam: 0.0809
   rmse: 0.0775
   ires: [66x1 double]
   pres: [66x1 double]
  iwres: [66x1 double]
  pwres: [66x1 double]
  cwres: [66x1 double]
```

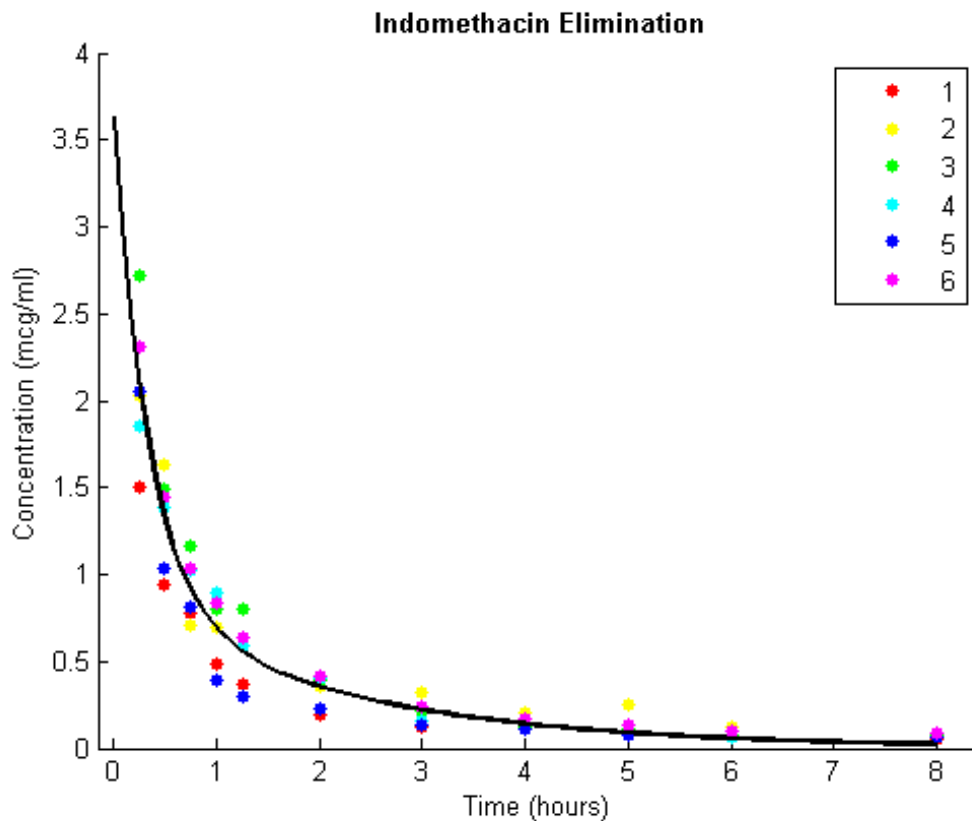
br =

-0.2255	0.0063	0.1600	0.1773	-0.3269	0.1157
0.0350	-0.1384	0.0058	0.0431	0.0093	-0.0453
-0.7557	-0.0550	0.8736	-0.7875	0.5304	0.1727
-0.0010	-0.0198	0.0137	-0.0757	0.0478	-0.0076



Plot the data along with an overall population fit

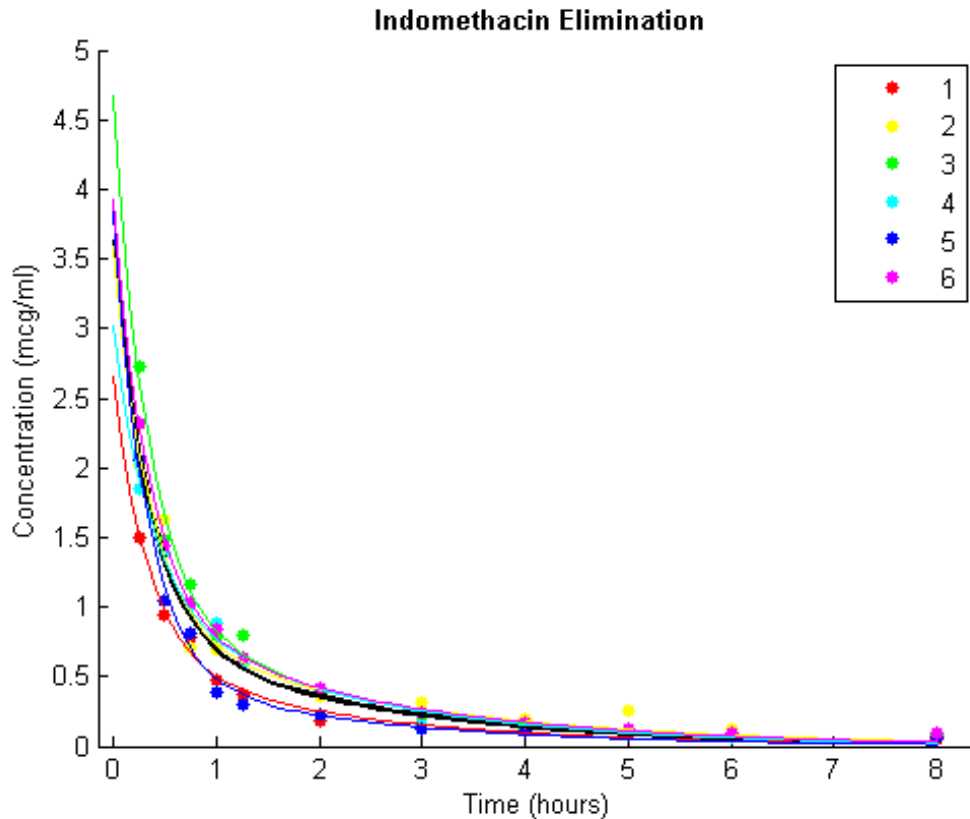
```
clf
phi = [beta(1), exp(beta(2)), beta(3), exp(beta(4))];
h = gscatter(time,concentration,subject);
xlabel('Time (hours)')
ylabel('Concentration (mcg/ml)')
title('{\bf Indomethacin Elimination}')
xx = linspace(0,8);
line(xx,model(phi,xx),'linewidth',2,'color','k')
```



Plot individual curves based on random-effect estimates.

```
for j=1:6
    phir = [beta(1)+br(1,j), exp(beta(2)+br(2,j)), ...
            beta(3)+br(3,j), exp(beta(4)+br(4,j))];
    line(xx,model(phir,xx),'color',get(h(j),'color'))
end
```





## Algorithms

In order to estimate the parameters of a nonlinear mixed effects model, we would like to choose the parameter values that maximize a likelihood function. These values are called the maximum likelihood estimates. The likelihood function can be written in the form

$$p(y | \beta, \sigma^2, \Sigma) = \int p(y | \beta, b, \sigma^2) p(b | \Sigma) db$$

where

- $y$  is the response data
- $\beta$  is the vector of population coefficients
- $\sigma^2$  is the residual variance
- $\Sigma$  is the covariance matrix for the random effects
- $b$  is the set of unobserved random effects

Each  $p()$  function on the right-hand-side is a normal (Gaussian) likelihood function that may depend on covariates.

Since the integral does not have a closed form, it is difficult to find parameters that maximize it. Delyon, Lavielle, and Moulines [1] proposed to find the maximum likelihood estimates using an Expectation-Maximization (EM) algorithm in which the E step is replaced by a stochastic procedure. They called their algorithm SAEM, for Stochastic Approximation EM. They demonstrated that this algorithm has desirable theoretical properties, including convergence under practical conditions and convergence to a local maximum of the likelihood function. Their proposal involves three steps:

- 1** Simulation: Generate simulated values of the random effects  $b$  from the posterior density  $p(b | \Sigma)$  given the current parameter estimates.
- 2** Stochastic approximation: Update the expected value of the loglikelihood function by taking its value from the previous step, and moving part way toward the average value of the loglikelihood calculated from the simulated random effects.
- 3** Maximization step: Choose new parameter estimates to maximize the loglikelihood function given the simulated values of the random effects.

## References

- [1] Delyon, B., M. Lavielle, and E. Moulines, *Convergence of a stochastic approximation version of the EM algorithm*, Annals of Statistics, 27, 94-128, 1999.

[2] Mentré, France, and Marc Lavielle, *Stochastic EM algorithms in population PKPD analyses*, 2008.

**See Also**

`nlinfit` | `nlpredci` | `nlmefit`

**How To**

- “Mixed-Effects Models” on page 11-23
- “Grouping Variables” on page 2-51

# gmdistribution.NlogL property

---

**Purpose** Negative of log-likelihood

**Description** The negative of the log-likelihood of the data.

---

**Note** This property applies only to gmdistribution objects constructed with `fit`.

---

# ProbDistParametric.NLogL property

---

<b>Purpose</b>	Read-only value specifying negative log likelihood for input data to ProbDistParametric object
<b>Description</b>	NLogL is a read-only property of the ProbDistParametric class. NLogL is a value specifying the negative log likelihood for input data used to fit a distribution represented by a ProbDistParametric object.
<b>Values</b>	The value is a numeric scalar for a distribution fit to input data, that is, a distribution created using the <code>fitdist</code> function. This property is empty for distributions created without fitting to data, that is, by using the <code>ProbDistUnivParam.ProbDistUnivParam</code> constructor. Use this information to view and compare the negative log likelihood for input data supplied to create distributions.

# ProbDistUnivKernel.NLogL property

---

<b>Purpose</b>	Read-only value specifying negative log likelihood for input data to ProbDistUnivKernel object
<b>Description</b>	NLogL is a read-only property of the ProbDistUnivKernel class. NLogL is a value specifying the negative log likelihood for input data used to fit a distribution represented by a ProbDistUnivKernel object.
<b>Values</b>	The value is a numeric scalar for a distribution fit to input data, that is, a distribution created using the <code>fitdist</code> function. Use this information to view and compare the negative log likelihood for input data used to create distributions.

**Purpose**

Nonlinear regression parameter confidence intervals

**Syntax**

```
ci = nlparci(beta,resid,'covar',sigma)
ci = nlparci(beta,resid,'jacobian',J)
ci = nlparci(...,'alpha',alpha)
```

**Description**

`ci = nlparci(beta,resid,'covar',sigma)` returns the 95% confidence intervals `ci` for the nonlinear least squares parameter estimates `beta`. Before calling `nlparci`, use `nlinfit` to fit a nonlinear regression model and get the coefficient estimates `beta`, residuals `resid`, and estimated coefficient covariance matrix `sigma`.

`ci = nlparci(beta,resid,'jacobian',J)` is an alternative syntax that also computes 95% confidence intervals. `J` is the Jacobian computed by `nlinfit`. If the `'robust'` option is used with `nlinfit`, use the `'covar'` input rather than the `'jacobian'` input so that the required `sigma` parameter takes the robust fitting into account.

`ci = nlparci(...,'alpha',alpha)` returns  $100(1-\alpha)\%$  confidence intervals.

`nlparci` treats NaNs in `resid` or `J` as missing values, and ignores the corresponding observations.

The confidence interval calculation is valid for systems where the length of `resid` exceeds the length of `beta` and `J` has full column rank. When `J` is ill-conditioned, confidence intervals may be inaccurate.

**Examples****Fit to exponential decay**

Suppose you have data, and want to fit a model of the form

$$y_i = a_1 + a_2 \exp(-a_3 x_i) + \varepsilon_i.$$

Here the  $a_i$  are the parameters you want to estimate,  $x_i$  are the data points, the  $y_i$  are the responses, and the  $\varepsilon_i$  are noise terms.

**1** Write a function handle that represents the model:

```
mdl = @(a,x)(a(1) + a(2)*exp(-a(3)*x));
```

- 2** Generate synthetic data with parameters  $a = [1;3;2]$ , with the  $x$  data points distributed exponentially with parameter 2, and normally distributed noise with standard deviation 0.1:

```
rng(9845,'twister') % for reproducibility
a = [1;3;2];
x = exprnd(2,100,1);
epsn = normrnd(0,0.1,100,1);
y = mdl(a,x) + epsn;
```

- 3** Fit the model to data starting from the arbitrary guess  $a_0 = [2;2;2]$ :

```
a0 = [2;2;2];
[ahat,r,J,cov,mse] = nlinfit(x,y,mdl,a0);
ahat
```

```
ahat =
    1.0153
    3.0229
    2.1070
```

- 4** Check whether  $[1;3;2]$  is in a 95% confidence interval using the Jacobian argument in `nlparci`:

```
ci = nlparci(ahat,r,'Jacobian',J)
```

```
ci =
    0.9869    1.0438
    2.9401    3.1058
    1.9963    2.2177
```

- 5** You can obtain the same result using the covariance argument:

```
ci = nlparci(ahat,r,'covar',cov)
```

```
ci =
    0.9869    1.0438
```



2.9401    3.1058  
1.9963    2.2177

**See Also**    `nlinfit` | `nlpredci`

# nlpredci

---

## Purpose

Nonlinear regression prediction confidence intervals

## Syntax

```
[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Covar',CovB)
[Ypred,delta] =
nlpredci(modelfun,X,beta,R,'Covar',CovB,Name,
        Value)
```

```
[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Jacobian',J)
[Ypred,delta] =
nlpredci(modelfun,X,beta,R,'Jacobian',J,Name,
        Value)
```

## Description

[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Covar',CovB) returns predictions, Ypred, and 95% confidence interval half-widths, delta, for the nonlinear regression model modelfun at input values X. Before calling nlpredci, use nlinfit to fit modelfun and get the estimated coefficients, beta, residuals, R, and variance-covariance matrix, CovB.

[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Covar',CovB,Name, Value) uses additional options specified by one or more name-value pair arguments.

[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Jacobian',J) returns predictions, Ypred, and 95% confidence interval half-widths, delta, for the nonlinear regression model modelfun at input values X. Before calling nlpredci, use nlinfit to fit modelfun and get the estimated coefficients, beta, residuals, R, and Jacobian, J.

If you use a robust option with nlinfit, then you should use the Covar syntax rather than the Jacobian syntax. The variance-covariance matrix, CovB, is required to properly take the robust fitting into account.

[Ypred,delta] = nlpredci(modelfun,X,beta,R,'Jacobian',J,Name, Value) uses additional options specified by one or more name-value pair arguments.

## Input Arguments

### **modelfun - Nonlinear regression model function**

function handle

Nonlinear regression model function, specified as a function handle. `modelfun` must accept two input arguments, a coefficient vector and an array  $X$ —in that order—and return a vector of fitted response values.

For example, to specify the `hougen` nonlinear regression function, use the function handle `@hougen`.

### **Data Types**

function\_handle

### **X - Input values for predictions**

matrix

Input values for predictions, specified as a matrix. `nlpredci` makes a prediction for the covariates in each row of  $X$ . There should be a column in  $X$  for each coefficient in the model.

### **Data Types**

single | double

### **beta - Estimated regression coefficients**

vector returned by `nlinfit`

Estimated regression coefficients, specified as the vector of fitted coefficients returned by a previous call to `nlinfit`.

### **Data Types**

single | double

### **R - Residuals**

vector returned by `nlinfit`

Residuals for the fitted `modelfun`, specified as the vector of residuals returned by a previous call to `nlinfit`.

### **CovB - Estimated variance-covariance matrix**

matrix returned by `nlinfit`

Estimated variance-covariance matrix for the fitted coefficients, `beta`, specified as the variance-covariance matrix returned by a previous call to `nlinf`.

## **J - Estimated Jacobian**

matrix returned by `nlinf`

Estimated Jacobian of the nonlinear regression model, `modelfun`, specified as the Jacobian matrix returned by a previous call to `nlinf`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'Alpha', 0.1, 'PredOpt', 'observation'` specifies 90% prediction intervals for new observations.

## **'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level for the confidence interval, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1). If `Alpha` has value  $\alpha$ , then `nlpredci` returns intervals with  $100 \times (1 - \alpha)\%$  confidence level.

The default confidence level is 95% ( $\alpha = 0.05$ ).

**Example:** `'Alpha', 0.1`

## **Data Types**

single | double

## **'ErrorModelInfo' - Information about error model fit**

structure returned by `nlinf`

Information about the error model fit, specified as the comma-separated pair consisting of 'ErrorModelInfo' and a structure returned by a previous call to `nlinfit`.

`ErrorModelInfo` only has an effect on the returned prediction interval when `PredOpt` has the value 'observation'. If you do not use `ErrorModelInfo`, then `nlpredci` assumes the error variance model is 'constant'.

The error model structure returned by `nlinfit` has the following fields:

<code>ErrorModel</code>	Chosen error model
<code>ErrorParameters</code>	Estimated error parameters
<code>ErrorVariance</code>	Function handle that accepts an $N$ -by- $p$ matrix, $X$ , and returns an $N$ -by-1 vector of error variances using the estimated error model
<code>MSE</code>	Mean squared error
<code>ScheffeSimPred</code>	Scheffé parameter for simultaneous prediction intervals when using the estimated error model
<code>WeightFunction</code>	Logical with value <code>true</code> if you used a custom weight function previously in <code>nlinfit</code>
<code>FixedWeights</code>	Logical with value <code>true</code> if you used fixed weights previously in <code>nlinfit</code>
<code>RobustWeightFunction</code>	Logical with value <code>true</code> if you used robust fitting previously in

### 'MSE' - Mean squared error

MSE returned by `nlinfit`

Mean squared error (MSE) for the fitted nonlinear regression model, specified as the comma-separated pair consisting of 'MSE' and the MSE value returned by a previous call to `nlinfit`.

If you use a robust option with `nlinfit`, then you must specify the MSE when predicting new observations to properly take the robust fitting into account. If you do not specify the MSE, then `nlpredci` computes the MSE from the residuals, `R`, and does not take the robust fitting into account.

For example, if `mse` is the MSE value returned by `nlinfit`, then you can specify `'MSE',mse`.

### Data Types

`single` | `double`

### 'PredOpt' - Prediction interval to compute

`'curve'` (default) | `'observation'`

Prediction interval to compute, specified as the comma-separated pair consisting of `'PredOpt'` and either `'curve'` or `'observation'`.

- If you specify the value `'curve'`, then `nlpredci` returns confidence intervals for the estimated curve (function value) at the observations `X`.
- If you specify the value `'observation'`, then `nlpredci` returns prediction intervals for new observations at `X`.

If you specify `'observation'` after using a robust option with `nlinfit`, then you must also specify a value for MSE to provide the robust estimate of the mean squared error.

**Example:** `'PredOpt','observation'`

### Data Types

`char`

### 'SimOpt' - Indicator for specifying simultaneous bounds

`'off'` (default) | `'on'`

Indicator for specifying simultaneous bounds, specified as the comma-separated pair consisting of `'SimOpt'` and either `'off'` or `'on'`. Use the value `'off'` to compute nonsimultaneous bounds, and `'on'` for simultaneous bounds.

**'Weights' - Observation weights**

vector | function handle

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of positive scalar values or a function handle. The default is no weights.

- If you specify a vector of weights, then it must have the same number of elements as the number of observations (rows) in X.
- If you specify a function handle for the weights, then it must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights, W, nlpredci estimates the error variance at observation i by  $mse * (1/W(i))$ , where mse is the mean squared error value specified using MSE.

**Example:** 'Weights',@WFun

**Data Types**

double | single | function\_handle

**Output Arguments****Ypred - Predicted responses**

vector

Predicted responses, returned as a vector with the same number of rows as X.

**delta - Confidence interval half-widths**

vector

Confidence interval half-widths, returned as a vector with the same number of rows as X. By default, delta contains the half-widths for nonsimultaneous 95% confidence intervals for modelfun at the observations in X. You can compute the lower and upper bounds of the confidence intervals as Ypred-delta and Ypred+delta, respectively.

If 'PredOpt' has value 'observation', then delta contains the half-widths for prediction intervals of new observations at the values in X.

## Examples

### Confidence Interval for Nonlinear Regression Curve

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the initial values in beta0.

```
[beta,R,J] = nlinfit(X,y,@hougen,beta0);
```

Obtain the predicted response and 95% confidence interval half-width for the value of the curve at average reactant levels.

```
[ypred,delta] = nlpredci(@hougen,mean(X),beta,R,'Jacobian',J)
```

```
ypred =
```

```
5.4622
```

```
delta =
```

```
0.1921
```

Compute the 95% confidence interval for the value of the curve.

```
[ypred-delta,ypred+delta]
```

```
ans =
```

```
5.2702 5.6543
```

### Prediction Interval for New Observation

Load sample data.



```
S = load('reaction');
X = S.reactants;
y = S.rate;
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the initial values in beta0.

```
[beta,R,J] = nlinfit(X,y,@hougen,beta0);
```

Obtain the predicted response and 95% prediction interval half-width for a new observation with reactant levels [100,100,100].

```
[ypred,delta] = nlpredci(@hougen,[100,100,100],beta,R,'Jacobian',J,...
                        'PredOpt','observation')
```

```
ypred =
```

```
    1.8346
```

```
delta =
```

```
    0.5101
```

Compute the 95% prediction interval for the new observation.

```
[ypred-delta,ypred+delta]
```

```
ans =
```

```
    1.3245    2.3447
```

### **Simultaneous Confidence Intervals for Robust Fit Curve**

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp\{-b_3 x\} + \varepsilon,$$

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x)(b(1)+b(2)*exp(-b(3)*x));
```

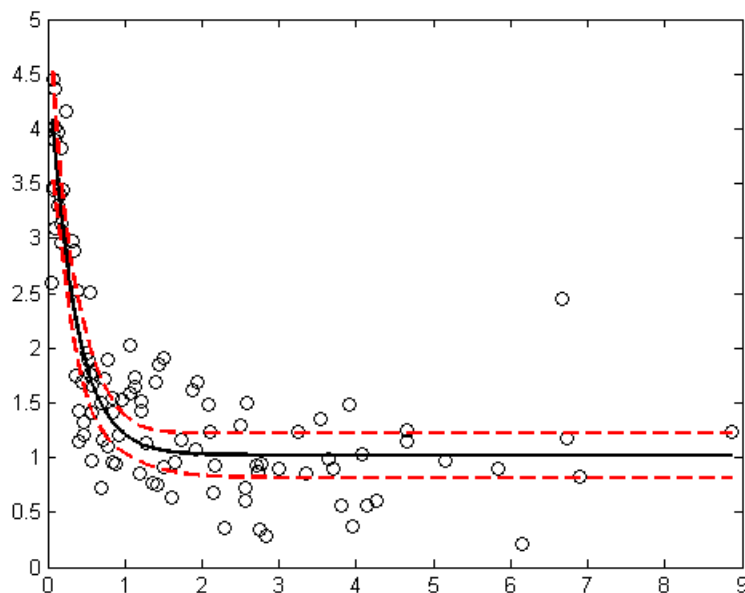
```
rng('default') % for reproducibility  
b = [1;3;2];  
x = exprnd(2,100,1);  
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

Fit the nonlinear model using robust fitting options.

```
opts = statset('nlinfit');  
opts.RobustWgtFun = 'bisquare';  
beta0 = [2;2;2];  
[beta,R,J,CovB,MSE] = nlinfit(x,y,modelfun,beta0,opts);
```

Plot the fitted regression model and simultaneous 95% confidence bounds.

```
xrange = min(x):.01:max(x);  
[ypred,delta] = nlpredci(modelfun,xrange,beta,R,'Covar',CovB,...  
                        'MSE',MSE,'SimOpt','on');  
lower = ypred - delta;  
upper = ypred + delta;  
  
figure()  
plot(x,y,'ko') % observed data  
hold on  
plot(xrange,ypred,'k','LineWidth',2)  
plot(xrange,[lower;upper],'r--','LineWidth',1.5)
```



### Confidence Interval Using Observation Weights

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Specify a function handle for observation weights, then fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
a = 1; b = 1;  
weights = @(yhat) 1./((a + b*abs(yhat)).^2);  
[beta,R,J,CovB] = nlinfit(X,y,@hougen,beta0,'Weights',weights);
```

# nlpredci

---

Compute the 95% prediction interval for a new observation with reactant levels [100,100,100] using the observation weight function.

```
[ypred,delta] = nlpredci(@hougen,[100,100,100],beta,R,'Jacobian',J,...  
                        'PredOpt','observation','Weights',weights);  
[ypred-delta,ypred+delta]
```

```
ans =
```

```
    1.5264    2.1033
```

## Confidence Interval Using Nonconstant Error Model

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Fit the Hougen-Watson model to the rate data using the combined error variance model.

```
[beta,R,J,CovB,MSE,S] = nlinfit(X,y,@hougen,beta0,'ErrorModel','combined')
```

Compute the 95% prediction interval for a new observation with reactant levels [100,100,100] using the fitted error variance model.

```
[ypred,delta] = nlpredci(@hougen,[100,100,100],beta,R,'Jacobian',J,...  
                        'PredOpt','observation','ErrorModelInfo',S);  
[ypred-delta,ypred+delta]
```

```
ans =
```

```
    1.3245    2.3447
```

## Algorithms

- `nlpredci` treats NaN values in the residuals, R, or the Jacobian, J, as missing values, and ignores the corresponding observations.

- If the Jacobian,  $J$ , does not have full column rank, then some of the model parameters might be nonidentifiable. In this case, `nlpredci` tries to construct confidence intervals for estimable predictions, and returns NaN for those that are not.
- To compute confidence intervals for complex parameters or data, you need to split the problem into its real and imaginary parts. When calling `nlinfit`:
  - 1 Define your parameter vector `beta` as the concatenation of the real and imaginary parts of the original parameter vector.
  - 2 Concatenate the real and imaginary parts of the response vector `Y` as a single vector.
  - 3 Modify your model function `modelfun` to accept `X` and the purely real parameter vector, and return a concatenation of the real and imaginary parts of the fitted values.

With the problem formulated this way, `nlinfit` computes real estimates, and confidence intervals are feasible.

## Definitions

### Confidence Intervals for Estimable Predictions

When the estimated model Jacobian is not of full rank, then it might not be possible to construct sensible confidence intervals at all prediction points. In this case, `nlpredci` still tries to construct confidence intervals for any *estimable* prediction points.

For example, suppose you fit the linear function

$f(\mathbf{x}_i, \beta) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3}$  at the points in the design matrix

$$\mathbf{X} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}.$$

The estimated Jacobian at the values in  $\mathbf{X}$  is the design matrix itself,  $\mathbf{J} = \mathbf{X}$ . Thus, the Jacobian is not of full rank:

```
rng('default') % For reproducibility
y = randn(6,1);

linfun = @(b,x) x*b;
beta0 = [1;1;1];
X = [repmat([1 1 0],3,1); repmat([1 0 1],3,1)];

[beta,R,J] = nlinfit(X,y,linfun,beta0);
```

```
Warning: The Jacobian at the solution is ill-conditioned, and
some model parameters may not be estimated well (they are not
identifiable). Use caution in making predictions.
> In nlinfit at 283
```

In this example, `nlpredci` can only compute prediction intervals at points that satisfy the linear relationship

$$x_{i1} = x_{i2} + x_{i3}.$$

If you try to compute confidence intervals for predictions at nonidentifiable points, `nlpredci` returns NaN for the corresponding interval half-widths:

```
xpred = [1 1 1;0 1 -1;2 1 1];
[ypred,delta] = nlpredci(linfun,xpred,beta,R,'Jacobian',J)

ypred =

    -0.0035
     0.0798
    -0.0047

delta =
```

```
      NaN  
      3.8102  
      3.8102
```

Here, the first element of `delta` is NaN because the first row in `xpred` does not satisfy the required linear dependence, and is therefore not an estimable contrast.

## References

- [1] Lane, T. P. and W. H. DuMouchel. “Simultaneous Confidence Intervals in Multiple Regression.” *The American Statistician*. Vol. 48, No. 4, 1994, pp. 315–321.
- [2] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

## See Also

`nlinfit` | `nlparciNonLinearModel` |

# nnmf

---

**Purpose** Nonnegative matrix factorization

**Syntax**  
 $[W,H] = \text{nnmf}(A,k)$   
 $[W,H] = \text{nnmf}(A,k,param1,val1,param2,val2,...)$   
 $[W,H,D] = \text{nnmf}(...)$

**Description**  $[W,H] = \text{nnmf}(A,k)$  factors the nonnegative  $n$ -by- $m$  matrix  $A$  into nonnegative factors  $W$  ( $n$ -by- $k$ ) and  $H$  ( $k$ -by- $m$ ). The factorization is not exact;  $W*H$  is a lower-rank approximation to  $A$ . The factors  $W$  and  $H$  are chosen to minimize the root-mean-squared residual  $D$  between  $A$  and  $W*H$ :

$$D = \sqrt{\text{norm}(A-W*H, 'fro') / (N*M)}$$

The factorization uses an iterative method starting with random initial values for  $W$  and  $H$ . Because the root-mean-squared residual  $D$  may have local minima, repeated factorizations may yield different  $W$  and  $H$ . Sometimes the algorithm converges to a solution of lower rank than  $k$ , which may indicate that the result is not optimal.

$W$  and  $H$  are normalized so that the rows of  $H$  have unit length. The columns of  $W$  are ordered by decreasing length.

$[W,H] = \text{nnmf}(A,k,param1,val1,param2,val2,...)$  specifies optional parameter name/value pairs from the following table.

Parameter	Value
'algorithm'	Either 'als' (the default) to use an alternating least-squares algorithm, or 'mult' to use a multiplicative update algorithm.  In general, the 'als' algorithm converges faster and more consistently. The 'mult' algorithm is more sensitive to initial values, which makes it a



Parameter	Value
	good choice when using 'replicates' to find W and H from multiple random starting values.
'w0'	An $n$ -by- $k$ matrix to be used as the initial value for W.
'h0'	A $k$ -by- $m$ matrix to be used as the initial value for H.
'options'	<p>An options structure as created by the <code>statset</code> function. <code>nnmf</code> uses the following fields of the options structure:</p> <ul style="list-style-type: none"> <li>• <b>Display</b> — Level of display. Choices: <ul style="list-style-type: none"> <li>▪ 'off' (default) — No display</li> <li>▪ 'final' — Display final result</li> <li>▪ 'iter' — Iterative display of intermediate results</li> </ul> </li> <li>• <b>MaxIter</b> — Maximum number of iterations. Default is 100. Unlike in optimization settings, reaching <b>MaxIter</b> iterations is treated as convergence.</li> <li>• <b>TolFun</b> — Termination tolerance on change in size of the residual. Default is <math>1e-4</math>.</li> <li>• <b>TolX</b> — Termination tolerance on relative change in the elements of W and H. Default is <math>1e-4</math>.</li> <li>• <b>UseParallel</b> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <b>UseSubstreams</b> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <b>Streams</b></li> </ul>

Parameter	Value
	<p>to a type allowing substreams: 'mlfg6331_64' or 'mrg32k3a'.</p> <ul style="list-style-type: none"> <li>Streams — A RandStream object or cell array of such objects. If you do not specify Streams, nnmf uses the default stream or streams. If you choose to specify Streams, use a single object except in the case <ul style="list-style-type: none"> <li>You have an open Parallel pool</li> <li>UseParallel is true</li> <li>UseSubstreams is false</li> </ul> </li> </ul> <p>In that case, use a cell array the same size as the Parallel pool.</p>
'replicates'	<p>The number of times to repeat the factorization, using new random starting values for W and H, except at the first replication if 'w0' and 'h0' are given. This is most beneficial with the 'mult' algorithm. The default is 1.</p>

[W,H,D] = nnmf(...) also returns D, the root mean square residual.

## Examples

### Example 1

Compute a nonnegative rank-two approximation of the measurements of the four variables in Fisher's iris data:

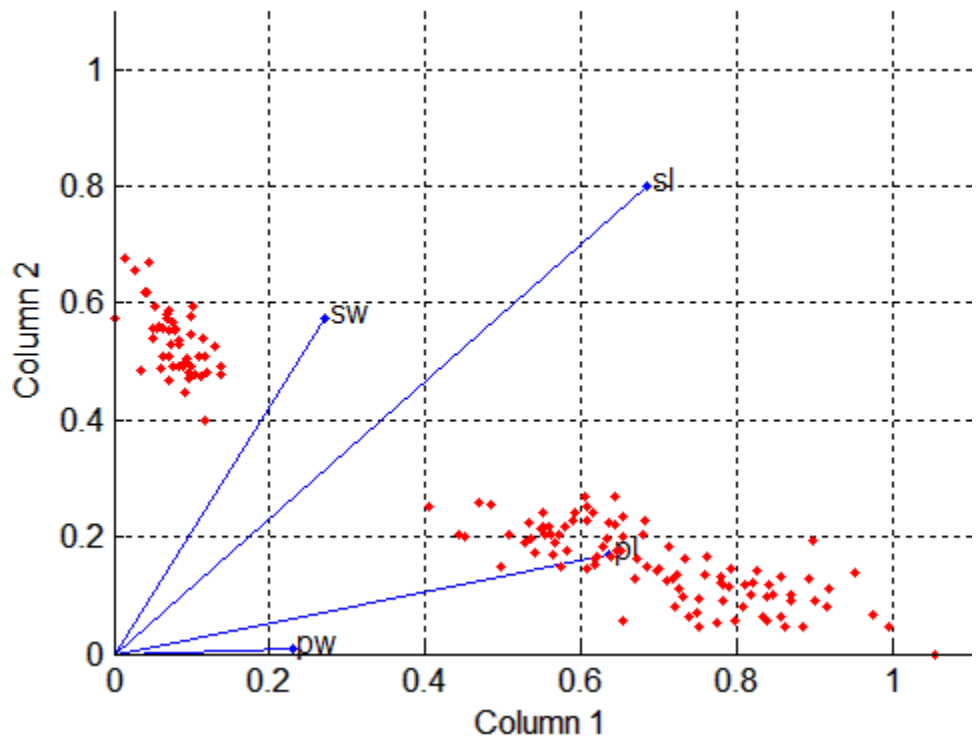
```
load fisheriris
[W,H] = nnmf(meas,2);
H
H =
    0.6852    0.2719    0.6357    0.2288
    0.8011    0.5740    0.1694    0.0087
```

The first and third variables in meas (sepal length and petal length, with coefficients 0.6852 and 0.6357, respectively) provide relatively

strong weights to the first column of  $W$ . The first and second variables in `meas` (sepal length and sepal width, with coefficients 0.8011 and 0.5740) provide relatively strong weights to the second column of  $W$ .

Create a biplot of the data and the variables in `meas` in the column space of  $W$ :

```
biplot(H', 'scores', W, 'varlabels', {'sl', 'sw', 'pl', 'pw'});  
axis([0 1.1 0 1.1])  
xlabel('Column 1')  
ylabel('Column 2')
```



## Example 2

Starting from a random array  $X$  with rank 20, try a few iterations at several replicates using the multiplicative algorithm:

```
X = rand(100,20)*rand(20,50);
opt = statset('MaxIter',5,'Display','final');
[WO,H0] = nnmf(X,5,'replicates',10,...
               'options',opt,...
               'algorithm','mult');
```

rep	iteration	rms resid	delta x
1	5	0.560887	0.0245182
2	5	0.66418	0.0364471
3	5	0.609125	0.0358355
4	5	0.608894	0.0415491
5	5	0.619291	0.0455135
6	5	0.621549	0.0299965
7	5	0.640549	0.0438758
8	5	0.673015	0.0366856
9	5	0.606835	0.0318931
10	5	0.633526	0.0319591

Final root mean square residual = 0.560887

Continue with more iterations from the best of these results using alternating least squares:

```
opt = statset('Maxiter',1000,'Display','final');
[W,H] = nnmf(X,5,'w0',WO,'h0',H0,...
              'options',opt,...
              'algorithm','als');
```

rep	iteration	rms resid	delta x
1	80	0.256914	9.78625e-005

Final root mean square residual = 0.256914

## References

[1] Berry, M. W., et al. "Algorithms and Applications for Approximate Nonnegative Matrix Factorization." *Computational Statistics and Data Analysis*. Vol. 52, No. 1, 2007, pp. 155–173.

**See Also**

pca | factoran | statset

# classregtree.nodeclass

---

**Purpose** Class values of nodes of classification tree

**Syntax** NAME=nodeclass(T)  
NAME=nodeclass(T,J)  
[NAME,ID]=nodeclass(...)

**Description** NAME=nodeclass(T) returns an  $n$ -element cell array with the names of the most probable classes in each node of the tree T, where  $n$  is the number of nodes in the tree. Every element of this array is a string equal to one of the class names returned by `classname(T)`. For regression trees, `nodeclass` returns an empty cell array.

NAME=nodeclass(T,J) takes an array J of node numbers and returns the class names for the specified nodes.

[NAME,ID]=nodeclass(...) also returns a numeric array with the class index for each node. The class index is determined by the order of classes `classname` returns.

**See Also** classregtree | classname | numnodes

**Purpose** Return vector of node errors

**Syntax**  
`e = nodeerr(t)`  
`e = nodeerr(t,nodes)`

**Description** `e = nodeerr(t)` returns an  $n$ -element vector `e` of the errors of the nodes in the tree `t`, where  $n$  is the number of nodes. For a regression tree, the error `e(i)` for node `i` is the variance of the observations assigned to node `i`. For a classification tree, `e(i)` is the misclassification probability for node `i`.

`e = nodeerr(t,nodes)` takes a vector `nodes` of node numbers and returns the errors for the specified nodes.

The error `e` is the so-called *resubstitution error* computed by applying the tree to the same data used to create the tree. This error is likely to under estimate the error you would find if you applied the tree to new data. The `test` function provides options to compute the error (or cost) using cross-validation or a test sample.

**Examples** Create a classification tree for Fisher's iris data:

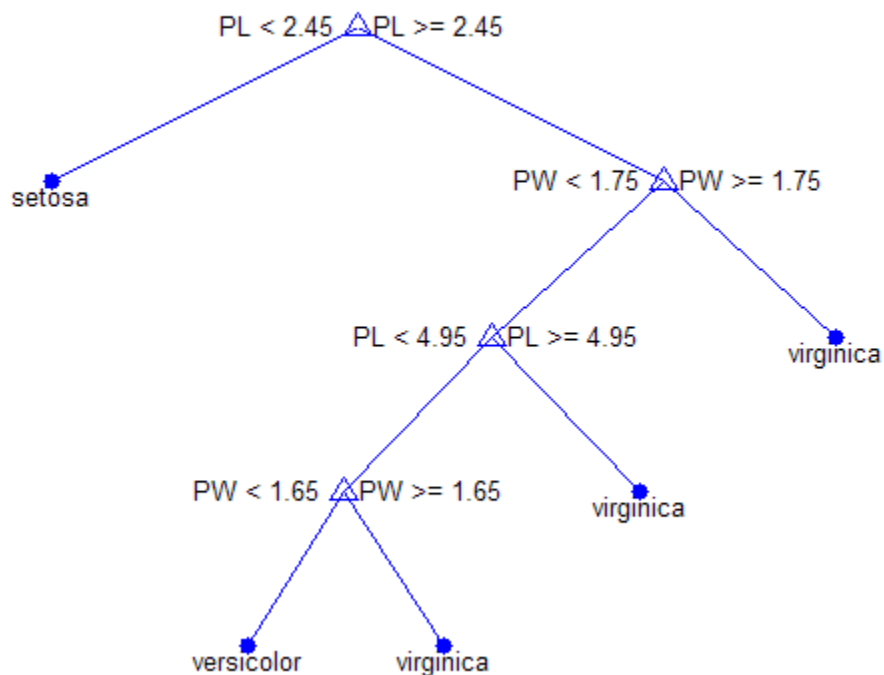
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica
```

# classregtree.nodeerr

view(t)

Click to display:  Magnification:  Pruning level:



e = nodeerr(t)

e =  
0.6667  
0  
0.5000  
0.0926  
0.0217  
0.0208



```
0.3333
      0
      0
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `numnodes` | `test`

# classregtree.nodemean

---

**Purpose** Mean values of nodes of regression tree

**Syntax** NM = nodemean(T)  
NM = nodemean(T,J)

**Description** NM = nodemean(T) returns an  $n$ -element numeric array with mean values in each node of the tree T, where  $n$  is the number of nodes in the tree. Every element of this array is computed by averaging true Y values over all observations in the node. For classification trees, nodemean returns an empty numeric array.

NM = nodemean(T,J) takes an array J of node numbers and returns the mean values for the specified nodes.

**See Also** classregtree | numnodes

## Purpose

Node probabilities

## Syntax

```
p = nodeprob(t)
p = nodeprob(t,nodes)
```

## Description

`p = nodeprob(t)` returns an  $n$ -element vector `p` of the probabilities of the nodes in the tree `t`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. For a classification tree, this proportion is adjusted for any prior probabilities assigned to each class.

`p = nodeprob(t,nodes)` takes a vector `nodes` of node numbers and returns the probabilities for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

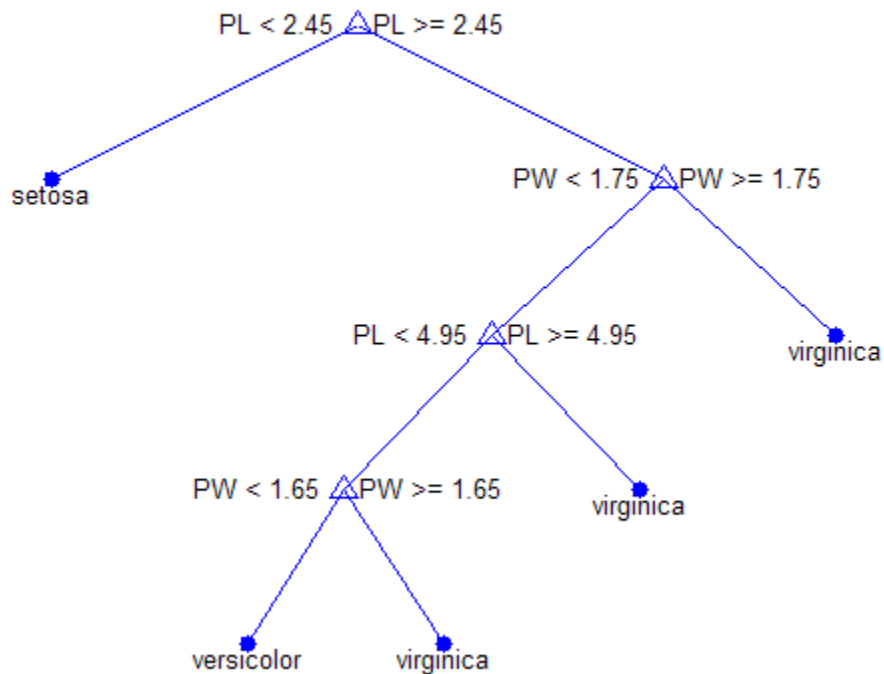
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

# classregtree.nodeprob

Click to display:  Magnification:  Pruning level:



p = nodeprob(t)

p =  
1.0000  
0.3333  
0.6667  
0.3600  
0.3067  
0.3200  
0.0400  
0.3133

0.0067

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree | nodesize | numnodes

# classregtree.nodesize

---

**Purpose** Return node size

**Syntax** `sizes = nodesize(t)`  
`sizes = nodesize(t,nodes)`

**Description** `sizes = nodesize(t)` returns an  $n$ -element vector `sizes` of the sizes of the nodes in the tree `t`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

`sizes = nodesize(t,nodes)` takes a vector `nodes` of node numbers and returns the sizes for the specified nodes.

**Examples** Create a classification tree for Fisher's iris data:

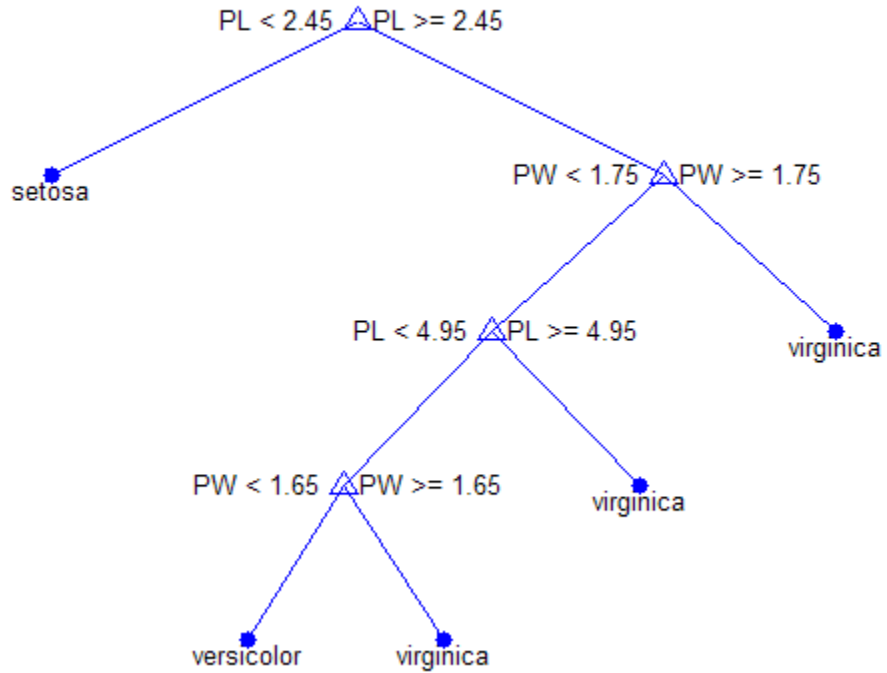
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

# classregtree.nodesize

Click to display:  Magnification:  Pruning level:



```
sizes = nodesize(t)
sizes =
    150
     50
    100
     54
     46
     48
     6
     47
```

# classregtree.nodesize

---

1

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree | numnodes



**Purpose** Notify listeners of event

**Syntax** `notify(h, 'eventname')`  
`notify(h, 'eventname', data)`

**Description** `notify(h, 'eventname')` notifies listeners added to the event named `eventname` on handle object array `h` that the event is taking place. `h` is the array of handles to objects triggering the event, and `eventname` must be a string.

`notify(h, 'eventname', data)` provides a way of encapsulating information about an event which can then be accessed by each registered listener. `data` must belong to the `event.EventData` class.

**See Also** `addlistener` | `event.EventData` | `events` | `grandstream`

# nominal

---

**Superclasses**    categorical

**Purpose**    Arrays for nominal categorical data

**Description**    Nominal arrays are used to store discrete values that are not numeric and that do not have an ordering. A nominal array provides efficient storage and convenient manipulation of such data, while also maintaining meaningful labels for the values. Nominal arrays are often used as grouping variables.

You can subscript, concatenate, reshape, etc. nominal arrays much like ordinary numeric arrays. You can test equality between elements of two nominal arrays, or between a nominal array and a single string representing a nominal value.

**Construction**    Use the `nominal` constructor to create a nominal array from a numeric, logical, or character array, or from a cell array of strings.

<code>nominal</code>	Construct nominal categorical array
----------------------	-------------------------------------

**Methods**    Each nominal array carries along a list of possible values that it can store, known as its levels. The list is created when you create a nominal array, and you can access it using the `getlevels` method, or modify it using the `addlevels`, `mergelevels`, or `droplevels` methods. Assignment to the array will also add new levels automatically if the values assigned are not already levels of the array.

You can change the order of the list of levels for a nominal array using the `reorderlevels` method, however, that order has no significance for the values in the array. The order is used only for display purposes, or when you convert the nominal array to numeric values using methods such as `double` or `subsindex`, or compare two arrays using `isequal`. If you need to work with values that have a mathematical ordering, you should use an `ordinal` array instead.

The following table lists operations available for nominal arrays.

---

summary	Summary statistics for nominal array
---------	--------------------------------------

### Inherited Methods

Methods in the following table are inherited from `categorical`.

addlevels	Add levels to categorical array
cat	Concatenate categorical arrays
cellstr	Convert categorical array to cell array of strings
char	Convert categorical array to character array
circshift	Shift categorical array circularly
ctranspose	Transpose categorical matrix
disp	Display categorical array
display	Display categorical array
double	Convert categorical array to double array
droplevels	Drop levels
end	Last index in indexing expression for categorical array
flipdim	Flip categorical array along specified dimension
fliplr	Flip categorical matrix in left/right direction
flipud	Flip categorical matrix in up/down direction
getlabels	Access categorical array labels
getlevels	Get categorical array levels

# nominal

---

hist	Plot histogram of categorical data
horzcat	Horizontal concatenation for categorical arrays
int16	Convert categorical array to signed 16-bit integer array
int32	Convert categorical array to signed 32-bit integer array
int64	Convert categorical array to signed 64-bit integer array
int8	Convert categorical array to signed 8-bit integer array
intersect	Set intersection for categorical arrays
ipermute	Inverse permute dimensions of categorical array
isempty	True for empty categorical array
isequal	True if categorical arrays are equal
islevel	Test for levels
ismember	True for elements of categorical array in set
isscalar	True if categorical array is scalar
isundefined	Test for undefined elements
isvector	True if categorical array is vector
length	Length of categorical array
levelcounts	Element counts by level
mergelevels	Merge levels

---

<code>ndims</code>	Number of dimensions of categorical array
<code>numel</code>	Number of elements in categorical array
<code>permute</code>	Permute dimensions of categorical array
<code>reorderlevels</code>	Reorder levels
<code>repmat</code>	Replicate and tile categorical array
<code>reshape</code>	Resize categorical array
<code>rot90</code>	Rotate categorical matrix 90 degrees
<code>setdiff</code>	Set difference for categorical arrays
<code>setlabels</code>	Label levels
<code>setxor</code>	Set exclusive-or for categorical arrays
<code>shiftdim</code>	Shift dimensions of categorical array
<code>single</code>	Convert categorical array to single array
<code>size</code>	Size of categorical array
<code>squeeze</code>	Squeeze singleton dimensions from categorical array
<code>subsasgn</code>	Subscripted assignment for categorical array
<code>subsindex</code>	Subscript index for categorical array

# nominal

---

suboref	Subscripted reference for categorical array
times	Product of categorical arrays
transpose	Transpose categorical matrix
uint16	Convert categorical array to unsigned 16-bit integers
uint32	Convert categorical array to unsigned 32-bit integers
uint64	Convert categorical array to unsigned 64-bit integers
uint8	Convert categorical array to unsigned 8-bit integers
union	Set union for categorical arrays
unique	Unique values in categorical array
vertcat	Vertical concatenation for categorical arrays

## Properties

## Inherited Properties

Properties in the following table are inherited from `categorical`.

labels	Text labels for levels
undeflabel	Text label for undefined levels

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## Examples

### Create and Compare Nominal Arrays

Create a nominal array from string data in a cell array.

```
colors = nominal({'r' 'b' 'g';'g' 'r' 'b';'b' 'r' 'g'},...  
                {'blue' 'green' 'red'})
```

```
colors =
```

red	blue	green
green	red	blue
blue	red	green

Identify the elements in `colors` that are members of the level 'red'.

```
colors == 'red'
```

```
ans =
```

1	0	0
0	1	0
0	1	0

A value of 1 in the resulting array `ans` indicates that the corresponding element of `colors` is a member of 'red'.

Identify the elements of `colors` that are members of either 'red' or 'blue'.

```
ismember(colors,{'red' 'blue'})
```

```
ans =
```

1	1	0
0	1	1
1	1	0

A value of 1 in the resulting array `ans` indicates that the corresponding element of `colors` is a member of either 'red' or 'blue'.

Create a second nominal array `colors2` by flipping `colors` from left to right.

# nominal

---

```
colors2 = fliplr(colors)
```

```
colors2 =
```

```
    green    blue    red  
    blue    red    green  
    green    red    blue
```

Compare the elements of `colors` and `colors2`.

```
colors == colors2
```

```
ans =
```

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

A value of 1 in the resulting array `ans` indicates that the corresponding element of `colors` is a member of the same level as the corresponding element of `colors2`.

## See Also

`histc` | `ordinal`



**Purpose**

Construct nominal categorical array

**Syntax**

```
B = nominal(A)
B = nominal(A,labels)
B = nominal(A,labels,levels)
B = nominal(A,labels,[],edges)
```

**Description**

`B = nominal(A)` creates a nominal array `B` from the array `A`. `A` can be numeric, logical, character, categorical, or a cell array of strings. `nominal` creates the levels of `B` from the sorted unique values in `A`, and creates default labels for them.

`B = nominal(A,labels)` labels the levels in `B` using the character array or cell array of strings `labels`. `nominal` assigns labels to levels in `B` in order according to the sorted unique values in `A`.

`B = nominal(A,labels,levels)` creates a nominal array with possible levels defined by `levels`. `levels` is a vector whose values can be compared to those in `A` using the equality operator. `nominal` assigns labels to each level from the corresponding elements of `labels`. If `A` contains any values not present in `levels`, the levels of the corresponding elements of `B` are undefined.

`B = nominal(A,labels,[],edges)` creates a nominal array by binning the numeric array `A` with bin edges given by the numeric vector `edges`. The uppermost bin includes values equal to the right-most edge. `nominal` assigns labels to each level in `B` from the corresponding elements of `labels`. `edges` must have one more element than `labels`.

By default, an element of `B` is undefined if the corresponding element of `A` is NaN (when `A` is numeric), an empty string (when `A` is character), or undefined (when `A` is categorical). `nominal` treats such elements as “undefined” or “missing” and does not include entries for them among the possible levels for `B`. To create an explicit level for such elements instead of treating them as undefined, you must use the `levels` input, and include NaN, the empty string, or an undefined element.

You may include duplicate labels in `labels` in order to merge multiple values in `A` into a single level in `B`.

# nominal

---

## Examples

### Create and Label Nominal Arrays

Load the sample data.

```
load fisheriris;
```

Create a nominal array and display a summary of the data.

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

```
      setosa      50  
versicolor      50  
virginica        50
```

Create a nominal array from characters, and provide explicit labels.

```
colors1 = nominal({'r' 'b' 'g'; 'g' 'r' 'b'; 'b' 'r' 'g'}, ...  
                 {'blue' 'green' 'red'})
```

```
colors1 =
```

```
      red      blue      green  
green      red      blue  
blue       red      green
```

Create a nominal array from characters, and provide both explicit labels and an explicit order for display.

```
colors2 = nominal({'r' 'b' 'g'; 'g' 'r' 'b'; 'b' 'r' 'g'}, ...  
                 {'red' 'green' 'blue'}, {'r' 'g' 'b'})
```

```
colors2 =
```

```
      red      blue      green  
green      red      blue  
blue       red      green
```

Create a nominal array from integer data, merging odd and even values into only two nominal levels. Provide explicit labels.

```
toss = nominal(randi([1 4],5,2),{'odd' 'even' 'odd' 'even'},1:4)
```

```
toss =
```

```

    even    odd
    even    even
    odd     odd
    even    even
    odd     even

```

### Change Nominal Array Levels

Load the sample data.

```
load hospital;
```

Make the {0,1}-valued variable `Smoker` nominal, and change the labels to 'No' and 'Yes'.

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

Add new levels to `Smoker` as placeholders for more detailed histories of smokers.

```
hospital.Smoker = addlevels(hospital.Smoker,...
    {'0-5 Years','5-10 Years','LongTerm'});
```

Assuming the nonsmokers have never smoked, relabel the 'No' level.

```
hospital.Smoker = setlabels(hospital.Smoker,'Never','No');
```

Drop the undifferentiated 'Yes' level from `Smoker`.

```
hospital.Smoker = droplevels(hospital.Smoker,'Yes');
```

Note that smokers now have an undefined level.

# nominal

---

Set each smoker to one of the new levels, by observation name.

```
hospital.Smoker('YPL-320') = '5-10 Years';
```

## See Also

[histc](#) | [ordinal](#)

**Purpose**

Nonlinear regression model class

**Description**

An object comprising training data, model description, diagnostic information, and fitted coefficients for a nonlinear regression. Predict model responses with the `predict` or `feval` methods.

**Construction**

`nlm = fitnlm(ds,modelfun,beta0)` or `nlm = fitnlm(X,y,modelfun,beta0)` create a nonlinear model of a dataset array `ds`, or of the responses `y` to a data matrix `X`. For details, see `fitnlm`.

**Input Arguments****ds - Input data**

dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

**Data Types**

single | double | logical

**X - Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor

# NonLinearModel

---

variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

## Data Types

single | double | logical

## **y** - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

## Data Types

single | double

## **modelfun** - Functional form of the model

function handle | string of the form ' $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ '

Functional form of the model, specified as either of the following.

- Function handle `@modelfun` or `@(b,x)modelfun`, where
  - $b$  is a coefficient vector with the same number of elements as `beta0`.
  - $x$  is a matrix with the same number of columns as  $X$  or the number of predictor variable columns of `ds`.

`modelfun(b,x)` returns a column vector that contains the same number of rows as  $x$ . Each row of the vector is the result of evaluating `modelfun` on the corresponding row of  $x$ . In other words, `modelfun` is a vectorized function, one that operates on all data rows and returns all evaluations in one function call.

- String of the form ' $y \sim f(b_1, b_2, \dots, b_j, x_1, x_2, \dots, x_k)$ ', where  $f$  represents a scalar function of the scalar coefficient variables  $b_1, \dots, b_j$  and the scalar data variables  $x_1, \dots, x_k$ .

## **beta0 - Coefficients**

numeric vector

Coefficients for the nonlinear model, specified as a numeric vector. NonLinearModel starts its search for optimal coefficients from beta0.

## **Data Types**

single | double

## **Properties**

### **CoefficientCovariance**

Covariance matrix of coefficient estimates.

### **CoefficientNames**

Cell array of strings containing a label for each coefficient.

### **Coefficients**

Table of coefficient values in a dataset array. Coefficients has one row for each coefficient and these columns:

- Estimate — Estimated coefficient value
- SE — Standard error of the estimate
- tStat —  $t$  statistic for a test that the coefficient is zero
- pValue —  $p$ -value for the  $t$  statistic

To obtain any of these columns as a vector, index into the property using dot notation. For example, in mdl the estimated coefficient vector is

```
beta = mdl.Coefficients.Estimate
```

Use coefTest to perform other tests on the coefficients.

### **Diagnostics**

# NonLinearModel

---

Dataset array with diagnostics helpful in finding outliers and influential observations. The structure contains the following fields.

Field	Meaning	Utility
Leverage	Diagonal elements of HatMatrix	Leverage indicates to what extent the predicted value for an observation is determined by the observed value for that observation. A value close to 1 indicates that the prediction is largely determined by that observation, with little contribution from the other observations. A value close to 0 indicates the fit is largely determined by the other observations. For a model with P coefficients and N observations, the average value of Leverage is P/N. An observation with Leverage larger than $2 \cdot P/N$ can be regarded as having high leverage.
CooksDistance	Cook's measure of scaled change in fitted values	CooksDistance is a measure of scaled change in fitted values. An observation with CooksDistance larger than three times the mean Cook's distance can be an outlier.
HatMatrix	Projection matrix to compute fitted from observed responses	HatMatrix is an N-by-N matrix such that $Fitted = HatMatrix \cdot Y$ , where Y is the response vector and Fitted is the vector of fitted response values.

## DFE

Degrees of freedom for error (residuals), equal to the number of observations minus the number of estimated coefficients.

## Fitted

Vector of predicted values based on the training data. `fitnlm` attempts to make Fitted as close as possible to the response data.

## Formula



Object that represents the mathematical form of the model.

## **Iterative**

Structure with information about the fitting process. Fields:

- `InitialCoefs` — Initial coefficient values (the `beta0` vector)
- `IterOpts` — Options included in the `Options` name-value pair argument for `fitnlm`.

## **LogLikelihood**

Log likelihood of the model distribution at the response values, with mean fitted from the model, and other parameters estimated as part of the model fit.

## **ModelCriterion**

AIC and other information criteria for comparing models. A structure with fields:

- `AIC` — Akaike information criterion
- `AICc` — Akaike information criterion corrected for sample size
- `BIC` — Bayesian information criterion
- `CAIC` — Consistent Akaike information criterion

To obtain any of these values as a scalar, index into the property using dot notation. For example, in a model `mdl`, the AIC value `aic` is:

```
aic = mdl.ModelCriterion.AIC
```

## **MSE**

Mean squared error, a scalar that is an estimate of the variance of the error term in the model.

## **NumCoefficients**

# NonLinearModel

---

Number of coefficients in the fitted model, a scalar. `NumCoefficients` is the same as `NumEstimatedCoefficients` for `NonLinearModel` objects. `NumEstimatedCoefficients` is equal to the degrees of freedom for regression.

## **NumEstimatedCoefficients**

Number of estimated coefficients in the fitted model, a scalar. `NumEstimatedCoefficients` is the same as `NumCoefficients` for `NonLinearModel` objects. `NumEstimatedCoefficients` is equal to the degrees of freedom for regression.

## **NumPredictors**

Number of variables `fitnlm` used as predictors for fitting.

## **NumVariables**

Number of variables in the data. `NumVariables` is the number of variables in the original dataset when the fit is based on a dataset, or the total number of columns in the predictor matrix and response vector when the fit is based on those arrays. It includes variables, if any, that are not used as predictors or as the response.

## **ObservationInfo**

Dataset with the same number of rows as the input data (`ds` or `X`).

<b>Field</b>	<b>Description</b>
Weights	Observation weights. Default is all 1.
Excluded	Logical value, 1 indicates an observation that you excluded from the fit with the <code>Exclude</code> name-value pair.
Missing	Logical value, 1 indicates a missing value in the input. Missing values are not used in the fit.
Subset	Logical value, 1 indicates the observation is not excluded or missing, so is used in the fit.

## ObservationNames

Cell array of strings containing the names of the observations used in the fit.

- If the fit is based on a dataset containing observation names, `ObservationNames` uses those names.
- Otherwise, `ObservationNames` is an empty cell array

## PredictorNames

Cell array of strings, the names of the predictors used in fitting the model.

## Residuals

Dataset array containing a table of residuals, with one row for each observation and these variables.

Field	Description
Raw	Observed minus fitted values.
Pearson	Raw residuals divided by RMSE.
Standardized	Raw residuals divided by their estimated standard deviation.
Studentized	Residual divided by an independent estimate of the residual standard deviation. The residual for observation $i$ is divided by an estimate of the error standard deviation based on all observations except for observation $i$ .

To obtain any of these columns as a vector, index into the property using dot notation. For example, in a model `mdl`, the ordinary raw residual vector `r` is:

```
r = mdl.Residuals.Raw
```

# NonLinearModel

---

Rows not used in the fit because of missing values (in `ObservationInfo.Missing`) contain NaN values.

Rows not used in the fit because of excluded values (in `ObservationInfo.Excluded`) contain NaN values, with the following exceptions:

- `raw` contains the difference between the observed and predicted values.
- `standardized` is the residual, standardized in the usual way.
- `studentized` matches the standardized values because this residual is not used in the estimate of the residual standard deviation.

## ResponseName

String giving naming the response variable.

## RMSE

Root mean squared error, a scalar that is an estimate of the standard deviation of the error term in the model.

## Robust

Structure that is empty unless `fitnlm` constructed the model using robust regression.

Field	Description
WgtFun	Robust weighting function, such as 'bisquare' (see <code>robustfit</code> )
Tune	Value specified for tuning parameter (can be [ ])
Weights	Vector of weights used in final iteration of robust fit

## Rsquared

Proportion of total sum of squares explained by the model. The ordinary R-squared value relates to the SSR and SST properties:

$$Rsquared = SSR/SST = 1 - SSE/SST.$$

Rsquared is a structure with two fields:

- Ordinary — Ordinary (unadjusted) R-squared
- Adjusted — R-squared adjusted for the number of coefficients

To obtain any of these values as a scalar, index into the property using dot notation. For example, the adjusted R-squared value in `mdl` is

```
r2 = mdl.Rsquared.Adjusted
```

## **SSE**

Sum of squared errors (residuals).

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

## **SSR**

Regression sum of squares, the sum of squared deviations of the fitted values from their mean.

The Pythagorean theorem implies

$$SST = SSE + SSR.$$

## **SST**

Total sum of squares, the sum of squared deviations of  $y$  from  $\text{mean}(y)$ .

The Pythagorean theorem implies

# NonLinearModel

---

$$\text{SST} = \text{SSE} + \text{SSR}.$$

## VariableInfo

Dataset array containing metadata about `Variables`. There is one row for each term in the model, and the following columns.

Field	Description
Class	String giving variable class, such as 'double'
Range	Cell array giving variable range: <ul style="list-style-type: none"><li>• Continuous variable — Two-element vector <math>[min, max]</math>, the minimum and maximum values</li><li>• Categorical variable — Cell array of distinct variable values</li></ul>
InModel	Logical vector, where true indicates the variable is in the model
IsCategorical	Logical vector, where true indicates a categorical variable

## VariableNames

Cell array of strings containing names of the variables in the fit.

- If the fit is based on a dataset, this property provides the names of the variables in that dataset.
- If the fit is based on a predictor matrix and response vector, `VariableNames` is the values in the `VarNames` name-value pair of the fitting method.
- Otherwise the variables have the default fitting names.

## Variables

Dataset array containing the data, both observations and responses, that the fitting function used to construct the fit. If

the fit is based on a dataset array, `Variables` is a copy of that dataset. Otherwise, `Variables` is a dataset created from the input data matrix  $X$  and response vector  $y$ .

## Methods

<code>coefCI</code>	Confidence intervals of coefficient estimates of nonlinear regression model
<code>coefTest</code>	Linear hypothesis test on nonlinear regression model coefficients
<code>disp</code>	Display nonlinear regression model
<code>feval</code>	Evaluate nonlinear regression model prediction
<code>fit</code>	Fit nonlinear regression model
<code>plotDiagnostics</code>	Plot diagnostics of nonlinear regression model
<code>plotResiduals</code>	Plot residuals of nonlinear regression model
<code>plotSlice</code>	Plot of slices through fitted nonlinear regression surface
<code>predict</code>	Predict response of nonlinear regression model
<code>random</code>	Simulate responses for nonlinear regression model

## Definitions

### Hat Matrix

The *hat matrix*  $H$  is defined in terms of the data matrix  $X$  and the Jacobian matrix  $J$ :

# NonLinearModel

---

$$J_{i,j} = \left. \frac{\partial f}{\partial \beta_j} \right|_{x_i, \beta}$$

Here  $f$  is the nonlinear model function, and  $\beta$  is the vector of model coefficients.

The Hat Matrix  $H$  is

$$H = J(J^T J)^{-1} J^T.$$

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$
$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

## Leverage

The *leverage* of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where



- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $MSE$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{pMSE} \left( \frac{h_{ii}}{(1-h_{ii})^2} \right),$$

where  $e_i$  is the  $i$ th residual.

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Nonlinear Model

Create a nonlinear model for auto mileage based on the carbig data. Predict the mileage of an average car.

Load the data and create a nonlinear model.

```
load carbig
X = [Horsepower,Weight];
y = MPG;
modelfun = @(b,x)b(1) + b(2)*x(:,1).^b(3) + ...
    b(4)*x(:,2).^b(5);
beta0 = [-50 500 -1 500 -1];
mdl = fitnlm(X,y,modelfun,beta0)
```

```
mdl =
```

# NonLinearModel

---

```
Nonlinear regression model:  
y ~ b1 + b2*x1^b3 + b4*x2^b5
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
b1	-49.383	119.97	-0.41164	0.68083
b2	376.43	567.05	0.66384	0.50719
b3	-0.78193	0.47168	-1.6578	0.098177
b4	422.37	776.02	0.54428	0.58656
b5	-0.24127	0.48325	-0.49926	0.61788

```
Number of observations: 392, Error degrees of freedom: 387  
Root Mean Squared Error: 3.96  
R-Squared: 0.745, Adjusted R-Squared 0.743  
F-statistic vs. constant model: 283, p-value = 1.79e-113
```

Find the predicted mileage of an average auto. The data contain some observations with NaN, so compute the mean using `nanmean`.

```
Xnew = nanmean(X)
```

```
Xnew =  
1.0e+03 *  
0.1051    2.9794
```

```
MPGnew = predict mdl, Xnew)
```

```
MPGnew =  
  
21.8073
```

## See Also

```
GeneralizedLinearModel | LinearModel | nlinfit |  
fitnlmNonLinearModel.predict |
```

## Related Examples

- “Nonlinear Regression Workflow” on page 11-16

## Concepts

- “Nonlinear Regression” on page 11-2

# prob.NormalDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Normal probability distribution object

**Description** `prob.NormalDistribution` is an object consisting of parameters, a model description, and sample data for a normal probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction**

`pd = makedist('Normal')` creates a normal probability distribution object using the default parameter values.

`pd = makedist('Normal','mu',mu,'sigma',sigma)` creates a normal distribution object using the specified parameter values.

## Input Arguments

### **mu - Mean**

0 (default) | scalar value

Mean of the normal distribution, specified as a scalar value.

### **Data Types**

single | double

### **sigma - Standard deviation**

1 (default) | nonnegative scalar value

Standard deviation of the normal distribution, specified as a nonnegative scalar value.

### **Data Types**

single | double

**Properties** `mu`

Mean of the normal distribution, stored as a scalar value.

**Data Types**

single | double

**sigma**

Standard deviation of the normal distribution, stored as a nonnegative scalar value.

**Data Types**

single | double

**DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

**Data Types**

char

**InputData**

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

**Data Types**

single | double

**IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

**Data Types**

logical

# prob.NormalDistribution

---

## **NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## **Data Types**

single | double

## **ParameterCovariance**

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## **Data Types**

single | double

## **ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## **Data Types**

char

## **ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## **Data Types**

logical

## **ParameterNames**

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object

# prob.NormalDistribution

---

random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Normal Distribution

The normal distribution, sometimes called the Gaussian distribution, is a two-parameter family of curves. The usual justification for using the normal distribution for modeling is the Central Limit theorem, which states (roughly) that the sum of independent samples from any distribution with finite mean and variance converges to the normal distribution as the sample size goes to infinity.

The normal distribution uses the following parameters.



Parameter	Description	Support
mu	Mean	$-\infty < \mu < \infty$
sigma	Standard deviation	$\sigma \geq 0$

The probability density function (pdf) is

$$f(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad -\infty < x < \infty.$$

## Examples

### Create a Normal Distribution Object Using Default Parameters

Create a normal distribution object using the default parameter values.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

### Create a Normal Distribution Object Using Specified Parameters

Create a normal distribution object by specifying the parameter values.

```
pd = makedist('Normal', 'mu', 75, 'sigma', 10)
```

```
pd =
```

```
NormalDistribution
```

# prob.NormalDistribution

---

```
Normal distribution
  mu = 75
  sigma = 10
```

Compute the interquartile range of the distribution.

```
r = iqr(pd)
```

```
r =
```

```
13.4898
```

## Fit a Normal Distribution Object

Load the sample data. Create a vector containing the first column of students' exam grades data.

```
load examgrades;
x = grades(:,1);
```

Create a normal distribution object by fitting it to the data.

```
pd = fitdist(x,'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
  mu = 75.0083    [73.4321, 76.5846]
  sigma =  8.7202    [7.7391, 9.98843]
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Normal Distribution” on page B-110
- Class Attributes
- Property Attributes

**Purpose**

Normal cumulative distribution function

**Syntax**

```
p = normcdf(x)
p = normcdf(x,mu,sigma)
[p,plo,pup] = normcdf(x,mu,sigma,pcov,alpha)
[p,plo,pup] = normcdf( __ , 'upper' )
```

**Description**

`p = normcdf(x)` returns the standard normal cdf at each value in `x`. The standard normal distribution has parameters `mu = 0` and `sigma = 1`. `x` can be a vector, matrix, or multidimensional array.

`p = normcdf(x,mu,sigma)` returns the normal cdf at each value in `x` using the specified values for the mean `mu` and standard deviation `sigma`. `x`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive.

`[p,plo,pup] = normcdf(x,mu,sigma,pcov,alpha)` returns confidence bounds for `p` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies  $100(1 - \alpha)\%$  confidence bounds. The default value of `alpha` is 0.05. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p,plo,pup] = normcdf( __ , 'upper' )` returns the complement of the normal cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use 'upper' with any of the previous syntaxes.

The function `normcdf` computes confidence bounds for `p` using a normal approximation to the distribution of the estimate

$$\frac{X - \hat{\mu}}{\hat{\sigma}}$$

and then transforming those bounds to the scale of the output `p`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller

# normcdf

---

samples other methods of computing the confidence bounds might be more accurate.

The normal cdf is

$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$

The result,  $p$ , is the probability that a single observation from a normal distribution with parameters  $\mu$  and  $\sigma$  will fall in the interval  $(-\infty, x]$ .

The *standard normal* distribution has  $\mu = 0$  and  $\sigma = 1$ .

## Examples

### Compute Normal Distribution cdf

What is the probability that an observation from a standard normal distribution will fall on the interval  $[-1, 1]$ ?

```
p = normcdf([-1 1]);  
p(2) - p(1)
```

```
ans =  
    0.6827
```

More generally, about 68% of the observations from a normal distribution fall within one standard deviation,  $\sigma$ , of the mean,  $\mu$ .

## See Also

[cdf](#) | [normpdf](#) | [norminv](#) | [normstat](#) | [normfit](#) | [normlike](#) | [normrnd](#)

## How To

- “Normal Distribution” on page B-110

**Purpose**

Normal parameter estimates

**Syntax**

```
[muhat,sigmahat] = normfit(data)
[muhat,sigmahat,muci,sigmaci] = normfit(data)
[muhat,sigmahat,muci,sigmaci] = normfit(data,alpha)
[...] = normfit(data,alpha,censoring)
[...] = normfit(data,alpha,censoring,freq)
[...] = normfit(data,alpha,censoring,freq,options)
```

**Description**

`[muhat,sigmahat] = normfit(data)` returns an estimate of the mean  $\mu$  in `muhat`, and an estimate of the standard deviation  $\sigma$  in `sigmahat`, of the normal distribution given the data in `data`.

`[muhat,sigmahat,muci,sigmaci] = normfit(data)` returns 95% confidence intervals for the parameter estimates on the mean and standard deviation in the arrays `muci` and `sigmaci`, respectively. The first row of `muci` contains the lower bounds of the confidence intervals for  $\mu$  the second row contains the upper bounds. The first row of `sigmaci` contains the lower bounds of the confidence intervals for  $\sigma$ , and the second row contains the upper bounds.

`[muhat,sigmahat,muci,sigmaci] = normfit(data,alpha)` returns  $100(1 - \alpha)$  % confidence intervals for the parameter estimates, where `alpha` is a value in the range `[0 1]` specifying the width of the confidence intervals. By default, `alpha` is 0.05, which corresponds to 95% confidence intervals.

`[...] = normfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly. `data` must be a vector in order to pass in the argument `censoring`.

`[...] = normfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. Typically, `freq` contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

# normfit

---

[...] = normfit(data,alpha,censoring,freq,options) accepts a structure, options, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates when there is censoring. The normal fit function accepts an options structure which you can create using the function statset. Enter statset('normfit') to see the names and default values of the parameters that normfit accepts in the options structure. See the reference page for statset for more information about these options.

---

**Note** With no censoring, normfit computes muhat using the sample mean and sigmahat using the square root of the unbiased estimator of the variance. With censoring, both muhat and sigmahat are the maximum likelihood estimates.

---

## Examples

In this example the data is a two-column random normal matrix. Both columns have  $\mu = 10$  and  $\sigma = 2$ . Note that the confidence intervals below contain the "true values."

```
data = normrnd(10,2,100,2);
[mu,sigma,muci,sigmaci] = normfit(data)
mu =
    10.1455    10.0527
sigma =
    1.9072    2.1256
muci =
    9.7652    9.6288
    10.5258    10.4766
sigmaci =
    1.6745    1.8663
    2.2155    2.4693
```

## See Also

mle | normlike | normpdf | normcdf | norminv | normstat | normrnd

## How To

- “Normal Distribution” on page B-110

**Purpose**

Normal inverse cumulative distribution function

**Syntax**

```
X = norminv(P,mu,sigma)
[X,XLO,XUP] = norminv(P,mu,sigma,pcov,alpha)
```

**Description**

`X = norminv(P,mu,sigma)` computes the inverse of the normal cdf using the corresponding mean `mu` and standard deviation `sigma` at the corresponding probabilities in `P`. `P`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive, and the values in `P` must lie in the interval `[0 1]`.

`[X,XLO,XUP] = norminv(P,mu,sigma,pcov,alpha)` produces confidence bounds for `X` when the input parameters `mu` and `sigma` are estimates. `pcov` is the covariance matrix of the estimated parameters. `alpha` specifies `100(1 - alpha)%` confidence bounds. The default value of `alpha` is `0.05`. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `norminv` computes confidence bounds for `P` using a normal approximation to the distribution of the estimate

$$\hat{\mu} + \hat{\sigma}q$$

where  $q$  is the  $P$ th quantile from a normal distribution with mean 0 and standard deviation 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds may be more accurate.

The normal inverse function is defined in terms of the normal cdf as

$$x = F^{-1}(p | \mu, \sigma) = \{x : F(x | \mu, \sigma) = p\}$$

where

# norminv

---

$$p = F(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt$$

The result,  $x$ , is the solution of the integral equation above where you supply the desired probability,  $p$ .

## Examples

Find an interval that contains 95% of the values from a standard normal distribution.

```
x = norminv([0.025 0.975],0,1)
x =
-1.9600  1.9600
```

Note that the interval  $x$  is not the only such interval, but it is the shortest.

```
x1 = norminv([0.01 0.96],0,1)
x1 =
-2.3263  1.7507
```

The interval  $x1$  also contains 95% of the probability, but it is longer than  $x$ .

## See Also

[icdf](#) | [normcdf](#) | [normpdf](#) | [normstat](#) | [normfit](#) | [normlike](#) | [normrnd](#)

## How To

- “Normal Distribution” on page B-110



---

<b>Purpose</b>	Normal negative log-likelihood
<b>Syntax</b>	<pre>nlogL = normlike(params,data) [nlogL,AVAR] = normlike(params,data) [...] = normlike(param,data,censoring) [...] = normlike(param,data,censoring,freq)</pre>
<b>Description</b>	<p><code>nlogL = normlike(params,data)</code> returns the negative of the normal log-likelihood function. <code>params(1)</code> is the mean, <code>mu</code>, and <code>params(2)</code> is the standard deviation, <code>sigma</code>.</p> <p><code>[nlogL,AVAR] = normlike(params,data)</code> also returns the inverse of Fisher's information matrix, <code>AVAR</code>. If the input parameter values in <code>params</code> are the maximum likelihood estimates, the diagonal elements of <code>AVAR</code> are their asymptotic variances. <code>AVAR</code> is based on the observed Fisher's information, not the expected information.</p> <p><code>[...] = normlike(param,data,censoring)</code> accepts a Boolean vector, <code>censoring</code>, of the same size as <code>data</code>, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.</p> <p><code>[...] = normlike(param,data,censoring,freq)</code> accepts a frequency vector, <code>freq</code>, of the same size as <code>data</code>. The vector <code>freq</code> typically contains integer frequencies for the corresponding elements in <code>data</code>, but can contain any nonnegative values. Pass in <code>[]</code> for <code>censoring</code> to use its default value.</p> <p><code>normlike</code> is a utility function for maximum likelihood estimation.</p>
<b>See Also</b>	<code>normfit</code>   <code>normpdf</code>   <code>normcdf</code>   <code>norminv</code>   <code>normstat</code>   <code>normrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Normal Distribution” on page B-110</li></ul>

# normpdf

---

**Purpose** Normal probability density function

**Syntax**  
`Y = normpdf(X,mu,sigma)`  
`Y = normpdf(X)`  
`Y = normpdf(X,mu)`

**Description** `Y = normpdf(X,mu,sigma)` computes the pdf at each of the values in `X` using the normal distribution with mean `mu` and standard deviation `sigma`. `X`, `mu`, and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `sigma` must be positive.

The normal pdf is

$$y = f(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The *likelihood function* is the pdf viewed as a function of the parameters. Maximum likelihood estimators (MLEs) are the values of the parameters that maximize the likelihood function for a fixed value of `x`.

The *standard normal* distribution has  $\mu = 0$  and  $\sigma = 1$ .

If `x` is standard normal, then `xσ + μ` is also normal with mean  $\mu$  and standard deviation  $\sigma$ . Conversely, if `y` is normal with mean  $\mu$  and standard deviation  $\sigma$ , then `x = (y-μ) / σ` is standard normal.

`Y = normpdf(X)` uses the standard normal distribution (`mu = 0`, `sigma = 1`).

`Y = normpdf(X,mu)` uses the normal distribution with unit standard deviation (`sigma = 1`).

## Examples

```
mu = [0:0.1:2];  
[y i] = max(normpdf(1.5,mu,1));  
MLE = mu(i)  
MLE =
```

1.5000

**See Also**

pdf | normcdf | norminv | normstat | normfit | normlike | normrnd  
| mvnpdf

**How To**

- “Normal Distribution” on page B-110

# normplot

---

**Purpose** Normal probability plot

**Syntax** `h = normplot(X)`

**Description** `h = normplot(X)` displays a normal probability plot of the data in `X`. For matrix `X`, `normplot` displays a line for each column of `X`. `h` is a handle to the plotted lines.

The plot has the sample data displayed with the plot symbol '+' . Superimposed on the plot is a line joining the first and third quartiles of each column of `X` (a robust linear fit of the sample order statistics.) This line is extrapolated out to the ends of the sample to help evaluate the linearity of the data.

The purpose of a normal probability plot is to graphically assess whether the data in `X` could come from a normal distribution. If the data are normal the plot will be linear. Other distribution types will introduce curvature in the plot. `normplot` uses midpoint probability plotting positions. Use `probplot` when the data included censored observations.

## **Examples** **Generate a Normal Probability Plot**

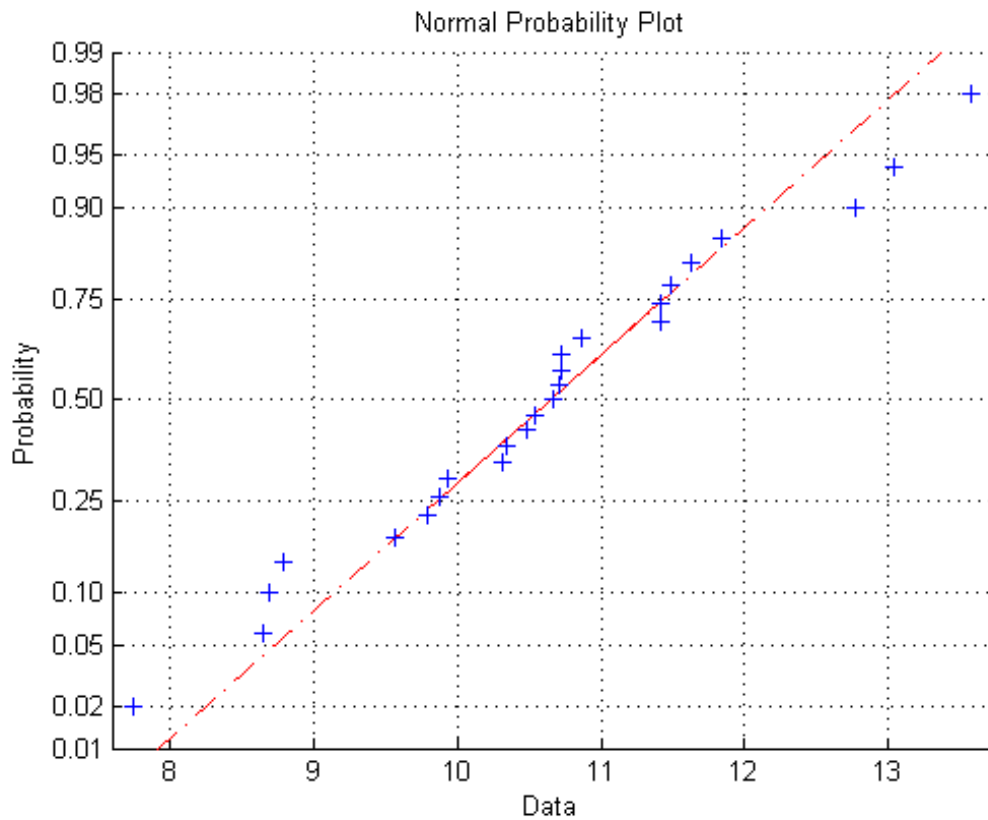
This example shows how to generate a normal probability plot using sample data.

Generate random sample data from a normal distribution with  $\mu = 10$  and  $\sigma = 1$ .

```
rng('default'); % For reproducibility
x = normrnd(10,1,25,1);
```

Create a normal probability plot of the sample data.

```
normplot(x)
```



The plot indicates that the data follows a normal distribution.

### See Also

`cdfplot` | `wblplot` | `probplot` | `hist` | `normfit` | `norminv` | `normpdf`  
| `normspec` | `normstat` | `normcdf` | `normrnd` | `normlike`

### How To

- “Normal Distribution” on page B-110

# normrnd

---

**Purpose** Normal random numbers

**Syntax**  
`R = normrnd(mu,sigma)`  
`R = normrnd(mu,sigma,m,n,...)`  
`R = normrnd(mu,sigma,[m,n,...])`

**Description** `R = normrnd(mu,sigma)` generates random numbers from the normal distribution with mean parameter `mu` and standard deviation parameter `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `R`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

`R = normrnd(mu,sigma,m,n,...)` or `R = normrnd(mu,sigma,[m,n,...])` generates an `m`-by-`n`-by-... array. The `mu`, `sigma` parameters can each be scalars or arrays of the same size as `R`.

**Examples**

```
n1 = normrnd(1:6,1./(1:6))
n1 =
    2.1650    2.3134    3.0250    4.0879    4.8607    6.2827

n2 = normrnd(0,1,[1 5])
n2 =
    0.0591    1.7971    0.2641    0.8717   -1.4462

n3 = normrnd([1 2 3;4 5 6],0.1,2,3)
n3 =
    0.9299    1.9361    2.9640
    4.1246    5.0577    5.9864
```

**See Also** `random` | `normpdf` | `normcdf` | `norminv` | `normstat` | `normfit` | `normlike` | `mvnrnd` | `lognrnd`

**How To** • “Normal Distribution” on page B-110

**Purpose** Normal density plot between specifications

**Syntax**

```
normspec(specs)
normspec(specs,mu,sigma)
normspec(specs,mu,sigma,region)
p = normspec(...)
[p,h] = normspec(...)
```

**Description**

`normspec(specs)` plots the standard normal density, shading the portion inside the specification limits given by the two-element vector `specs`. Set `specs(1)` to `-Inf` if there is no lower limit; set `specs(2)` to `Inf` if there is no upper limit.

`normspec(specs,mu,sigma)` shades the portion inside the specification limits of a normal density with parameters `mu` and `sigma`. The defaults are `mu = 0` and `sigma = 1`.

`normspec(specs,mu,sigma,region)` shades the *region* either 'inside' or 'outside' the specification limits. The default is 'inside'.

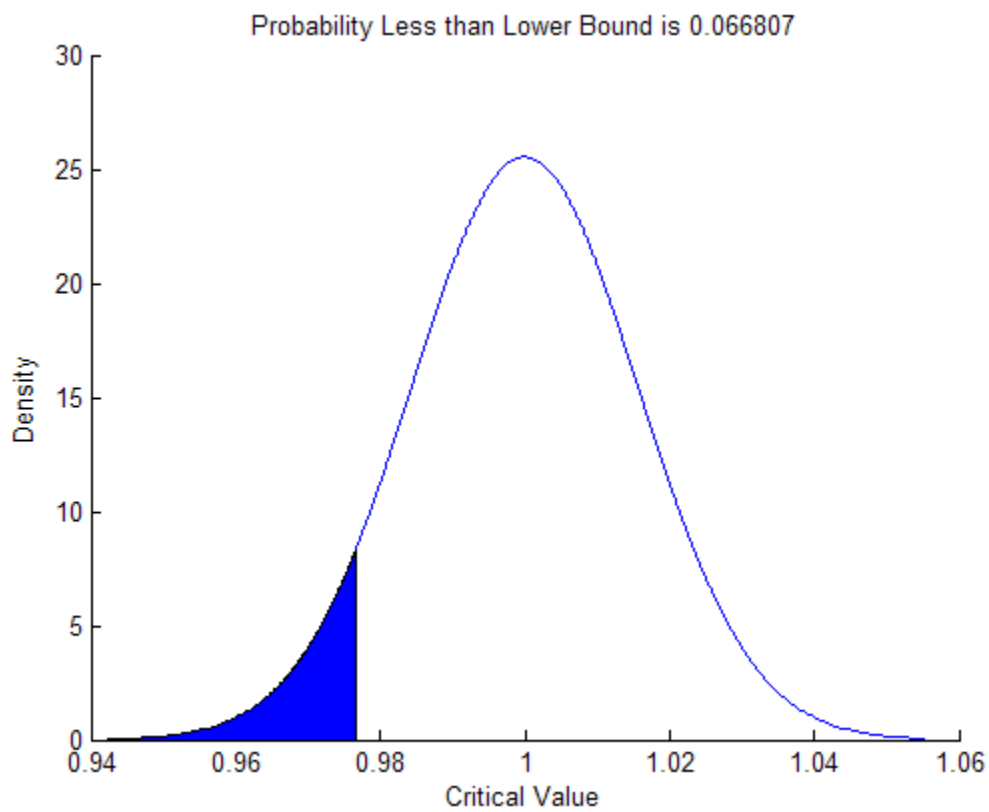
`p = normspec(...)` also returns the probability, `p`, of the shaded area.

`[p,h] = normspec(...)` also returns a handle `h` to the line objects.

**Examples**

A production process fills cans of paint. The average amount of paint in any can is 1 gallon, but variability in the process produces a standard deviation of 2 ounces (2/128 gallons). What is the probability that cans will be filled under specification by 3 or more ounces?

```
p = normspec([1-3/128,Inf],1,2/128,'outside')
p =
    0.0668
```



## See Also

`capaplot` | `histfit`

## How To

- “Normal Distribution” on page B-110



**Purpose** Normal mean and variance

**Syntax** `[M,V] = normstat(mu,sigma)`

**Description** `[M,V] = normstat(mu,sigma)` returns the mean of and variance for the normal distribution using the corresponding mean `mu` and standard deviation `sigma`. `mu` and `sigma` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `M` and `V`. A scalar input for `mu` or `sigma` is expanded to a constant array with the same dimensions as the other input.

The mean of the normal distribution with parameters  $\mu$  and  $\sigma$  is  $\mu$ , and the variance is  $\sigma^2$ .

**Examples**

```
n = 1:5;
[m,v] = normstat(n'*n,n'*n)
m =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
     4     8    12    16    20
     5    10    15    20    25

v =
     1     4     9    16    25
     4    16    36    64   100
     9    36    81   144   225
    16    64   144   256   400
    25   100   225   400   625
```

**See Also** `normpdf` | `normcdf` | `norminv` | `normfit` | `normlike` | `normrnd`

**How To** • “Normal Distribution” on page B-110

# piecewisedistribution.nsegments

---

**Purpose**            Number of segments

**Syntax**            `n = nsegments(obj)`

**Description**        `n = nsegments(obj)` returns the number of segments `n` in the piecewise distribution object `obj`.

**Examples**            Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);  
obj = paretotails(t,0.1,0.9);
```

```
n = nsegments(obj)  
n =  
    3
```

**See Also**            `paretotails` | `boundary` | `segment`

<b>Purpose</b>	Number of decision trees in ensemble
<b>Description</b>	The NTrees property is a scalar equal to the number of decision trees in the ensemble.
<b>See Also</b>	Trees

# ProbDistParametric.NumParams property

---

<b>Purpose</b>	Read-only value specifying number of parameters of ProbDistParametric object
<b>Description</b>	NumParams is a read-only property of the ProbDistParametric class. NumParams is a value specifying the number of parameters of a distribution represented by a ProbDistParametric object.
<b>Values</b>	This value is an integer that counts both the specified parameters and parameters that are fit to the data. Use this information to view and compare the number of parameters supplied to create distributions.

**Purpose** Number of elements in dataset array

**Syntax** `n = numel(A)`  
`n = numel(A, varargin)`

**Description** `n = numel(A)` returns 1. To find the number of elements, `n`, in the dataset array `A`, use `prod(size(A))` or `numel(A, ':', ':')`.  
`n = numel(A, varargin)` returns the number of subscripted elements, `n`, in `A(index1, index2, ..., indexn)`, where `varargin` is a cell array whose elements are `index1, index2, ..., indexn`.

**See Also** `length` | `size`

# categorical.numel

---

**Purpose**            Number of elements in categorical array

**Syntax**            `n = numel(A)`  
                      `n = numel(A, varargin)`

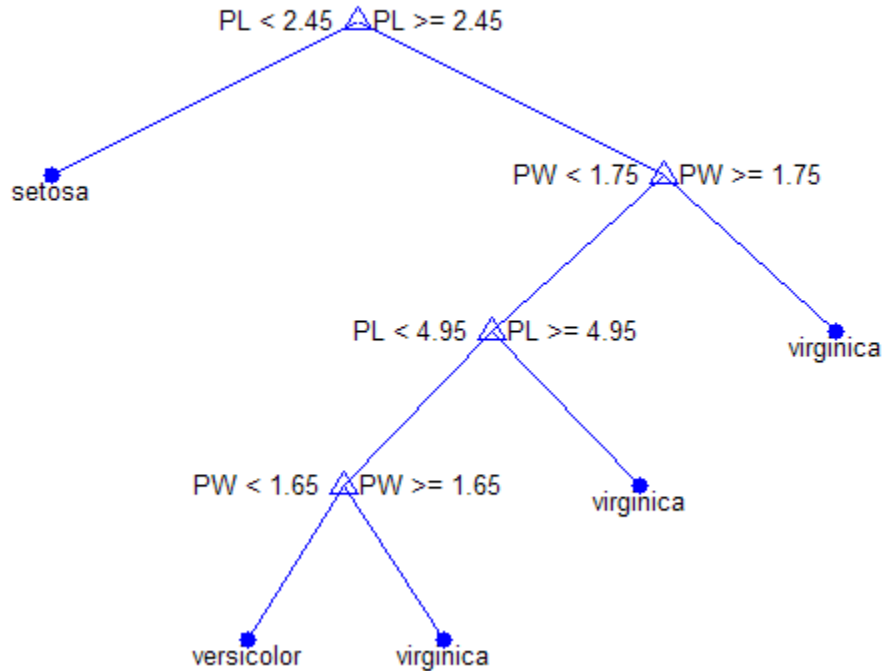
**Description**        `n = numel(A)` returns the number of elements in the categorical array `A`.  
  
`n = numel(A, varargin)` returns the number of subscripted elements, `n`, in `A(index1, index2, ..., indexN)`, where `varargin` is a cell array whose elements are `index1, index2, ... indexN`.

**See Also**            `size`

<b>Purpose</b>	Number of nodes
<b>Syntax</b>	<code>n = numnodes(t)</code>
<b>Description</b>	<code>n = numnodes(t)</code> returns the number of nodes <code>n</code> in the tree <code>t</code> .
<b>Examples</b>	<p>Create a classification tree for Fisher's iris data:</p> <pre>load fisheriris;  t = classregtree(meas,species,...                 'names',{'SL' 'SW' 'PL' 'PW'})  t= Decision tree for classification 1  if PL&lt;2.45 then node 2 elseif PL&gt;=2.45 then node 3 else setosa 2  class = setosa 3  if PW&lt;1.75 then node 4 elseif PW&gt;=1.75 then node 5 else versicolor 4  if PL&lt;4.95 then node 6 elseif PL&gt;=4.95 then node 7 else versicolor 5  class = virginica 6  if PW&lt;1.65 then node 8 elseif PW&gt;=1.65 then node 9 else versicolor 7  class = virginica 8  class = versicolor 9  class = virginica view(t)</pre>

# classregtree.numnodes

Click to display:  Magnification:  Pruning level:



n = numnodes(t)

n =

9

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree



# cvpartition.NumTestSets property

---

**Purpose**

Number of test sets

**Description**

Value is the number of folds in partitions of type 'kfold' and 'leaveout'.

Value is 1 in partitions of type 'holdout' and 'resubstitution'.

# TreeBagger.NVarToSample property

---

**Purpose** Number of variables for random feature selection

**Description** The `NVarToSample` property specifies the number of predictor or feature variables to select at random for each decision split. By default, it is set to the square root of the total number of variables for classification and one third of the total number of variables for regression. Setting this argument to any valid value except 'all' invokes Breiman's "random forest" algorithm.

**See Also** `classregtree`

# dataset.ObsNames property

---

**Purpose**

Cell array of nonempty, distinct strings giving names of observations in data set

**Description**

A cell array of nonempty, distinct strings giving the names of the observations in the data set. This property may be empty, but if not empty, the number of strings must equal the number of observations.

# optimalleaforder

---

**Purpose** Optimal leaf ordering for hierarchical clustering

**Syntax** `leafOrder = optimalleaforder(tree,D)`  
`leafOrder = optimalleaforder(tree,D,Name,Value)`

**Description** `leafOrder = optimalleaforder(tree,D)` returns an optimal leaf ordering for the hierarchical binary cluster tree, `tree`, using the distances, `D`. An optimal leaf ordering of a binary tree maximizes the sum of the similarities between adjacent leaves by flipping tree branches without dividing the clusters.

`leafOrder = optimalleaforder(tree,D,Name,Value)` returns the optimal leaf ordering using one or more name-value pair arguments.

## Input Arguments

### **tree - Hierarchical binary cluster tree**

matrix returned by `linkage`

Hierarchical binary cluster tree, specified as an  $(M - 1)$ -by-3 matrix that you generate using `linkage`, where  $M$  is the number of leaves.

### **D - Distances**

matrix | vector

Distances for determining similarities between leaves, specified as a matrix or vector of distances. For example, you can generate distances using `pdist`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'Criteria', 'group', 'Transformation', 'inverse'` specifies that the sum of similarities be maximized between every leaf

and all other leaves in adjacent clusters, using an inverse similarity transformation.

## 'Criteria' - Optimization criterion

'adjacent' (default) | 'group'

Optimization criterion for determining an optimal leaf ordering, specified as the comma-separated pair consisting of 'criteria' and one of these strings:

'adjacent'	Maximize the sum of similarities between adjacent leaves.
'group'	Maximize the sum of similarities between every leaf and all other leaves in the adjacent clusters at the same level of the dendrogram.

**Example:** 'Criteria', 'group'

## Data Types

char

## 'Transformation' - Method for transforming distances to similarities

'linear' (default) | 'inverse' | function handle

Method for transforming distances to similarities, specified as the comma-separated pair consisting of 'Transformation' and one of 'linear', 'inverse', or a function handle.

Let  $d_{i,j}$  and  $Sim_{i,j}$  denote the distance and similarity between leaves  $i$  and  $j$ , respectively. The included similarity transformations are:

'linear'	$Sim_{i,j} = \max_{i,j} (d_{i,j}) - d_{i,j}$
'inverse'	$Sim_{i,j} = 1/d_{i,j}$

To use a custom transformation function, specify a handle to a function that accepts a matrix of distances, D, and returns a matrix of similarities, S. The function should be monotonic decreasing in the

# optimalleaforder

---

range of distance values.  $S$  must have the same size as  $D$ , with  $S(i, j)$  being the similarity computed based on  $D(i, j)$ .

**Example:** 'Transformation',@myTransform

## Data Types

char | function\_handle

## Output Arguments

### leafOrder - Optimal leaf order

vector

Optimal leaf order, returned as a length- $M$  vector, where  $M$  is the number of leaves. `leafOrder` is a permutation of the vector  $1:M$ , giving an optimal leaf ordering based on the specified distances and similarity transformation.

## Examples

### Plot Dendrogram With Optimal Leaf Order

Create a hierarchical binary cluster tree using `linkage`. Then, compare the dendrogram plot with the default ordering to a dendrogram with an optimal leaf ordering.

Generate sample data.

```
rng('default') % For reproducibility
X = rand(10,2);
```

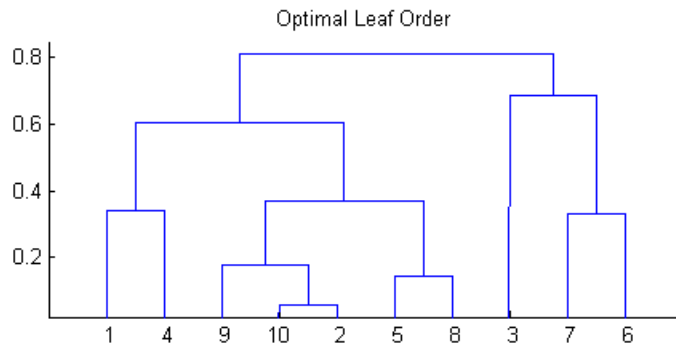
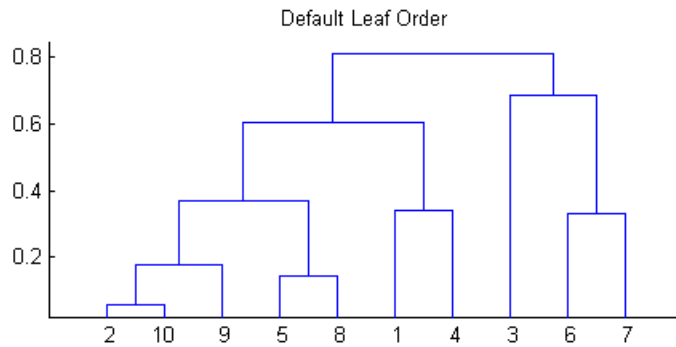
Create a distance vector and a hierarchical binary clustering tree. Use the distances and clustering tree to determine an optimal leaf order.

```
D = pdist(X);
tree = linkage(D, 'average');
leafOrder = optimalleaforder(tree,D);
```

Plot the dendrogram with the default ordering and the dendrogram with the optimal leaf ordering.

```
figure()
subplot(2,1,1)
dendrogram(tree)
```

```
title('Default Leaf Order')  
  
subplot(2,1,2)  
dendrogram(tree,'reorder',leafOrder)  
title('Optimal Leaf Order')
```



The order of the leaves in the bottom figure corresponds to the elements in `leafOrder`.

`leafOrder`

# optimalleaforder

---

```
leafOrder =  
    1    4    9   10    2    5    8    3    7    6
```

## Optimal Leaf Order Using Inverse Distance Similarity

Generate sample data.

```
rng('default') % For reproducibility  
X = rand(10,2);
```

Create a distance vector and a hierarchical binary clustering tree.

```
D = pdist(X);  
tree = linkage(D,'average');
```

Use the inverse distance similarity transformation to determine an optimal leaf order.

```
leafOrder = optimalleaforder(tree,D,'Transformation','inverse')
```

```
leafOrder =  
    1    4    9   10    2    5    8    3    7    6
```

## References

[1] Bar-Joseph, Z., Gifford, D.K., and Jaakkola, T.S. (2001). Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics* 17, Suppl 1:S22–9. PMID: 11472989.

## See Also

[dendrogram](#) | [linkage](#) | [pdist](#)



**Purpose** Out-of-bag classification edge

**Syntax**  
edge = oobEdge(ens)  
edge = oobEdge(ens,Name,Value)

**Description**  
edge = oobEdge(ens) returns out-of-bag classification edge for ens.  
edge = oobEdge(ens,Name,Value) computes classification edge with additional options specified by one or more Name,Value pair arguments. You can specify several name-value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

**Input Arguments**  
**ens**  
A classification bagged ensemble, constructed with fitensemble.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

#### **'learners'**

Indices of weak learners in the ensemble ranging from 1 to ens.NTrained. oobEdge uses only these learners for calculating loss.

**Default:** 1:NTrained

#### **'mode'**

String representing the meaning of the output L:

- 'ensemble' — L is a scalar value, the loss for the entire ensemble.

# ClassificationBaggedEnsemble.oobEdge

---

- 'individual' — L is a vector with one element per trained learner.
- 'cumulative' — L is a vector in which element J is obtained by using learners 1:J from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

### edge

Classification edge, a weighted average of the classification margin.

## Definitions

### Edge

The *edge* is the weighted mean value of the classification margin. The weights are the class probabilities in `ens.Prior`.

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

### Out of Bag

*Bagging*, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas. `fitensemble` obtains each bootstrap replica by randomly selecting N observations out of N with replacement, where N is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

Drawing N out of N observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed

prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## Examples

Find the out-of-bag edge for a bagged ensemble from the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'Bag',100,...
    'Tree','type','classification');
edge = oobEdge(ens)

edge =
    0.8730
```

## See Also

[oobMargin](#) | [oobPredict](#) | [oobLoss](#)

# TreeBagger.oobError

---

**Purpose** Out-of-bag error

**Syntax**  
`err = oobError(B)`  
`err = oobError(B, 'param1', val1, 'param2', val2, ...)`

**Description** `err = oobError(B)` computes the misclassification probability (for classification trees) or mean squared error (for regression trees) for out-of-bag observations in the training data, using the trained bagger `B`. `err` is a vector of length `NTrees`, where `NTrees` is the number of trees in the ensemble.

`err = oobError(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

`'mode'` String indicating how `oobError` computes errors. If set to `'cumulative'` (default), the method computes cumulative errors and `err` is a vector of length `NTrees`, where the first element gives error from `trees(1)`, second element gives error from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to `'individual'`, `err` is a vector of length `NTrees`, where each element is an error from each tree in the ensemble. If set to `'ensemble'`, `err` is a scalar showing the cumulative error for the entire ensemble.

`'trees'` Vector of indices indicating what trees to include in this calculation. By default, this argument is set to `'all'` and the method uses all trees. If `'trees'` is a numeric vector, the method returns a vector of length `NTrees` for `'cumulative'` and `'individual'` modes, where `NTrees` is the number of elements in the input vector, and a scalar for `'ensemble'` mode. For example, in the `'cumulative'` mode, the

first element gives error from `trees(1)`, the second element gives error from `trees(1:2)` etc.

'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. `oobError` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

## See Also

`CompactTreeBagger.error`

# TreeBagger.OOBIndices property

---

**Purpose** Indicator matrix for out-of-bag observations

**Description** The `OOBIndices` property is a logical array of size `Nobs-by-NTrees` where `Nobs` is the number of observations in the training data and `NTrees` is the number of trees in the ensemble. The  $(I, J)$  element is true if observation `I` is out-of-bag for tree `J` and false otherwise. In other words, a true value means observation `I` was not selected for the training data used to grow tree `J`.

**See Also** `classregtree`

# TreeBagger.OOBInstanceWeight property

---

**Purpose**

Count of out-of-bag trees for each observation

**Description**

The `OOBInstanceWeight` property is a numeric array of size `Nobs-by-1` containing the number of trees used for computing out-of-bag response for each observation. `Nobs` is the number of observations in the training data used to create the ensemble.

# ClassificationBaggedEnsemble.oobLoss

---

**Purpose** Out-of-bag classification error

**Syntax**  
L = oobloss(ens)  
L = oobloss(ens,Name,Value)

**Description** L = oobloss(ens) returns the classification error for ens computed for out-of-bag data.

L = oobloss(ens,Name,Value) computes error with additional options specified by one or more Name,Value pair arguments. You can specify several name-value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

## Input Arguments

**ens**  
A classification bagged ensemble, constructed with fitensemble.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

#### 'learners'

Indices of weak learners in the ensemble ranging from 1 to NTrained. oobLoss uses only these learners for calculating loss.

**Default:** 1:NTrained

#### 'lossfun'

Function handle or string representing a loss function. Built-in loss functions:

- 'binodeviance' — See “Loss Functions” on page 21-2306
- 'classiferror' — Fraction of misclassified data



- 'exponential' — See “Loss Functions” on page 21-2306

You can write your own loss function in the syntax described in “Loss Functions” on page 21-2306.

**Default:** 'classiferror'

## 'mode'

String representing the meaning of the output L:

- 'ensemble' — L is a scalar value, the loss for the entire ensemble.
- 'individual' — L is a vector with one element per trained learner.
- 'cumulative' — L is a vector in which element J is obtained by using learners 1:J from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**L**

Classification error of the out-of-bag observations, a scalar. L can be a vector, or can represent a different quantity, depending on the name-value settings.

## Definitions

### Out of Bag

*Bagging*, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas. `fitensemble` obtains each bootstrap replica by randomly selecting N observations out of N with replacement, where N is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

# ClassificationBaggedEnsemble.oobLoss

---

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'classiferror' — Fraction of misclassified data, weighted by  $w$ .
- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

To write your own loss function, create a function file of the form

```
function loss = lossfun(C,S,W,COST)
```

- $N$  is the number of rows of  $X$ .
- $K$  is the number of classes in `tree`, represented in `tree.ClassNames`.
- $C$  is an  $N$ -by- $K$  logical matrix, with one true per row for the true class. The index for each class is its position in `tree.ClassNames`.

# ClassificationBaggedEnsemble.oobLoss

---

- $S$  is an  $N$ -by- $K$  numeric matrix.  $S$  is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from `predict`.
- $W$  is a numeric vector with  $N$  elements, the observation weights.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. The default `'classiferror'` cost function uses a cost of 0 for correct classification, and 1 for misclassification. In other words, `'classiferror'` uses  $COST = \text{ones}(K) - \text{eye}(K)$ .
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## Examples

Find the out-of-bag error for a bagged ensemble from the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'Bag',100,...
    'Tree','type','classification');
L = oobLoss(ens)

L =
    0.0467
```

## See Also

`loss` | `oobEdge` | `oobMargin` | `oobPredict`

# RegressionBaggedEnsemble.oobLoss

---

**Purpose** Out-of-bag regression error

**Syntax**  
L = oobLoss(ens)  
L = oobLoss(ens,Name,Value)

**Description** L = oobLoss(ens) returns the mean squared error for ens computed for out-of-bag data.

L = oobLoss(ens,Name,Value) computes error with additional options specified by one or more Name,Value pair arguments. You can specify several name-value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

## Input Arguments

**ens**  
A regression bagged ensemble, constructed with fitensemble.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

### 'learners'

Indices of weak learners in the ensemble ranging from 1 to NTrained. oobLoss uses only these learners for calculating loss.

**Default:** 1:NTrained

### 'lossfun'

Function handle for loss function, or the string 'mse', meaning mean squared error. If you pass a function handle fun, oobLoss calls it as

FUN(Y,Yfit,W)

where  $Y$ ,  $Y_{fit}$ , and  $W$  are numeric vectors of the same length.  $Y$  is the observed response,  $Y_{fit}$  is the predicted response, and  $W$  is the observation weights.

**Default:** 'mse'

## 'mode'

String representing the meaning of the output  $L$ :

- 'ensemble' —  $L$  is a scalar value, the loss for the entire ensemble.
- 'individual' —  $L$  is a vector with one element per trained learner.
- 'cumulative' —  $L$  is a vector in which element  $J$  is obtained by using learners  $1:J$  from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

### $L$

Mean squared error of the out-of-bag observations, a scalar.  $L$  can be a vector, or can represent a different quantity, depending on the name-value settings.

## Definitions

### Out of Bag

*Bagging*, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas. `fitensemble` obtains each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag"

# RegressionBaggedEnsemble.oobLoss

---

observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## Examples

Compute the out-of-bag error for the `carsmall` data:

```
load carsmall
X = [Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'bag',100,'Tree',...
    'type','regression');
L = oobLoss(ens)

L =
    17.0665
```

## See Also

`oobPredict` | `loss`

# ClassificationBaggedEnsemble.oobMargin

---

<b>Purpose</b>	Out-of-bag classification margins
<b>Syntax</b>	<code>margin = oobMargin(ens)</code> <code>margin = oobMargin(ens,Name,Value)</code>
<b>Description</b>	<code>margin = oobMargin(ens)</code> returns out-of-bag classification margins. <code>margin = oobMargin(ens,Name,Value)</code> calculates margins with additional options specified by one or more <code>Name,Value</code> pair arguments.
<b>Input Arguments</b>	<b>ens</b> A classification bagged ensemble, constructed with <code>fitensemble</code> . <b>Name-Value Pair Arguments</b> Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> . <b>'learners'</b> Indices of weak learners in the ensemble ranging from 1 to <code>ens.NTrained</code> . <code>oobEdge</code> uses only these learners for calculating loss. <b>Default:</b> <code>1:NTrained</code>
<b>Output Arguments</b>	<b>margin</b> A numeric column vector of length <code>size(ens.X,1)</code> .
<b>Definitions</b>	<b>Out of Bag</b> <i>Bagging</i> , which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, <code>fitensemble</code> generates many bootstrap replicas of the dataset and

# ClassificationBaggedEnsemble.oobMargin

---

grows decision trees on these replicas. `fitensemble` obtains each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

## Examples

Find the out-of-bag margin for a bagged ensemble from the Fisher iris data: Find how many elements of margin are equal to 1.

```
load fisheriris
ens = fitensemble(meas,species,'Bag',100,...
    'Tree','type','classification');
margin = oobMargin(ens);
sum(margin == 1)

ans =
    108
```

## See Also

`oobPredict` | `oobLoss` | `oobEdge` | `margin`



## Purpose

Out-of-bag margins

## Syntax

```
mar = oobMargin(B)
mar = oobMargin(B, 'param1', val1, 'param2', val2, ...)
```

## Description

`mar = oobMargin(B)` computes an `Nobs`-by-`NTrees` matrix of classification margins for out-of-bag observations in the training data, using the trained bagger `B`.

`mar = oobMargin(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

'mode'	String indicating how <code>oobMargin</code> computes errors. If set to 'cumulative' (default), the method computes cumulative margins and <code>mar</code> is an <code>Nobs</code> -by- <code>NTrees</code> matrix, where the first column gives margins from <code>trees(1)</code> , second column gives margins from <code>trees(1:2)</code> etc, up to <code>trees(1:NTrees)</code> . If set to 'individual', <code>mar</code> is an <code>Nobs</code> -by- <code>NTrees</code> matrix, where each column gives margins from each tree in the ensemble. If set to 'ensemble', <code>mar</code> is a single column of length <code>Nobs</code> showing the cumulative margins for the entire ensemble.
'trees'	Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns an <code>Nobs</code> -by- <code>NTrees</code> matrix for 'cumulative' and 'individual' modes, where <code>NTrees</code> is the number of elements in the input vector, and a single column for 'ensemble' mode. For example, in the 'cumulative' mode, the first column gives

# TreeBagger.oobMargin

---

margins from `trees(1)`, the second column gives margins from `trees(1:2)` etc.

'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. `oobMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

## See Also

`CompactTreeBagger.margin`

## Purpose

Out-of-bag mean margins

## Syntax

```
mar = oobMeanMargin(B)
mar = oobMeanMargin(B, 'param1', val1, 'param2', val2, ...)
```

## Description

`mar = oobMeanMargin(B)` computes average classification margins for out-of-bag observations in the training data, using the trained bagger `B`. `oobMeanMargin` averages the margins over all out-of-bag observations. `mar` is a row-vector of length `NTrees`, where `NTrees` is the number of trees in the ensemble.

`mar = oobMeanMargin(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

- 'mode' String indicating how `oobMargin` computes errors. If set to 'cumulative' (default), is a vector of length `NTrees` where the first element gives mean margin from `trees(1)`, second column gives mean margins from `trees(1:2)` etc, up to `trees(1:NTrees)`. If set to 'individual', `mar` is a vector of length `NTrees`, where each element is a mean margin from each tree in the ensemble. If set to 'ensemble', `mar` is a scalar showing the cumulative mean margin for the entire ensemble.
- 'trees' Vector of indices indicating what trees to include in this calculation. By default, this argument is set to 'all' and the method uses all trees. If 'trees' is a numeric vector, the method returns a vector of length `NTrees` for 'cumulative' and 'individual' modes, where `NTrees` is the number of elements in the input vector, and a scalar for 'ensemble' mode. For example, in the 'cumulative' mode, the first element gives mean margin from `trees(1)`, the second element gives mean margin from `trees(1:2)` etc.
- 'treeweights' Vector of tree weights. This vector must have the same length as the 'trees' vector. `oobMeanMargin` uses these weights to combine output from the specified trees by taking a weighted average instead of the simple nonweighted majority vote. You cannot use this argument in the 'individual' mode.

# TreeBagger.oobMeanMargin

---

## See Also

`CompactTreeBagger.meanMargin`

# TreeBagger.OOBPermutedVarCountRaiseMargin property

---

**Purpose**

Variable importance for raising margin

**Description**

The `OOBPermutedVarCountRaiseMargin` property is a numeric array of size 1-by-Nvars containing a measure of variable importance for each predictor. For any variable, the measure is the difference between the number of raised margins and the number of lowered margins if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

# TreeBagger.OOBPermutedVarDeltaError property

---

**Purpose** Variable importance for prediction error

**Description** The OOBPermutedVarDeltaError property is a numeric array of size 1-by-*Nvars* containing a measure of importance for each predictor variable (feature). For any variable, the measure is the increase in prediction error if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble.

# TreeBagger.OOBPermutedVarDeltaMeanMargin property

---

**Purpose** Variable importance for classification margin

**Description** The `OOBPermutedVarDeltaMeanMargin` property is a numeric array of size 1-by-Nvars containing a measure of importance for each predictor variable (feature). For any variable, the measure is the decrease in the classification margin if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

# ClassificationBaggedEnsemble.oobPredict

---

<b>Purpose</b>	Predict out-of-bag response of ensemble
<b>Syntax</b>	<pre>[label,score] = oobPredict(ens) [label,score] = oobPredict(ens,Name,Value)</pre>
<b>Description</b>	<p>[label,score] = oobPredict(ens) returns class labels and scores for ens for out-of-bag data.</p> <p>[label,score] = oobPredict(ens,Name,Value) computes labels and scores with additional options specified by one or more Name,Value pair arguments.</p>
<b>Input Arguments</b>	<p><b>ens</b></p> <p>A classification bagged ensemble, constructed with fitensemble.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.</p> <p><b>'learners'</b></p> <p>Indices of weak learners in the ensemble ranging from 1 to ens.NTrained. oobEdge uses only these learners for calculating loss.</p> <p><b>Default:</b> 1:NTrained</p>
<b>Output Arguments</b>	<p><b>label</b></p> <p>Classification labels of the same data type as the training data Y. There are N elements or rows, where N is the number of training observations. The label is the class with the highest score. In case of a tie, the label is earliest in ens.ClassNames.</p> <p><b>score</b></p>



An N-by-K numeric matrix for N observations and K classes. A high score indicates that an observation is likely to come from this class. Scores are in the range 0 to 1.

## Definitions

### Out of Bag

*Bagging*, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas. `fitensemble` obtains each bootstrap replica by randomly selecting N observations out of N with replacement, where N is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

Drawing N out of N observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

# ClassificationBaggedEnsemble.oobPredict

---

## Examples

Find the out-of-bag predictions and scores for the Fisher iris data. Find the scores in the range (0.2,0.8); these are the scores where there is notable uncertainty in the resulting classifications.

```
load fisheriris
ens = fitensemble(meas,species,'Bag',100,...
    'Tree','type','classification');
[label score] = oobPredict(ens);
unsure = ( (score > .2) & (score < .8));
sum(sum(unsure)) % How many uncertain predictions?

ans =
    16
```

## See Also

[oobMargin](#) | [oobPredict](#) | [oobLoss](#) | [oobEdge](#) | [predict](#)

**Purpose** Predict out-of-bag response of ensemble

**Syntax**  
`Yfit = oobPredict(ens)`  
`Yfit = oobPredict(ens,Name,Value)`

**Description** `Yfit = oobPredict(ens)` returns the predicted responses for the out-of-bag data in `ens`.

`Yfit = oobPredict(ens,Name,Value)` predicts responses with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**ens**

A regression bagged ensemble, constructed with `fitensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### 'learners'

Indices of weak learners in the ensemble ranging from 1 to `NTrained`. `oobLoss` uses only these learners for calculating loss.

**Default:** `1:NTrained`

## Output Arguments

**Yfit**

A vector of predicted responses for out-of-bag data. `Yfit` has `size(ens.X,1)` elements.

You can find the indices of out-of-bag observations for weak learner `L` with the command

`~ens.UseObsForLearner(:,L)`

# RegressionBaggedEnsemble.oobPredict

---

## Definitions

### Out of Bag

*Bagging*, which stands for “bootstrap aggregation”, is a type of ensemble learning. To bag a weak learner such as a decision tree on a dataset, `fitensemble` generates many bootstrap replicas of the dataset and grows decision trees on these replicas. `fitensemble` obtains each bootstrap replica by randomly selecting  $N$  observations out of  $N$  with replacement, where  $N$  is the dataset size. To find the predicted response of a trained ensemble, `predict` take an average over predictions from individual trees.

Drawing  $N$  out of  $N$  observations with replacement omits on average 37% ( $1/e$ ) of observations for each decision tree. These are "out-of-bag" observations. For each observation, `oobLoss` estimates the out-of-bag prediction by averaging over predictions from all trees in the ensemble for which this observation is out of bag. It then compares the computed prediction against the true response for this observation. It calculates the out-of-bag error by comparing the out-of-bag predicted responses against the true responses for all observations used for training. This out-of-bag average is an unbiased estimator of the true ensemble error.

## Examples

Compute out-of-bag predictions for the `carsmall` data. Look at the first three terms of the fit:

```
load carsmall
X = [Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'bag',100,'Tree',...
    'type','regression');
Yfit = oobPredict(ens);
Yfit(1:3) % first three terms

ans =
    15.7964
    14.7162
    14.8062
```

## See Also

`oobLoss` | `predict`

**Purpose** Ensemble predictions for out-of-bag observations

**Syntax** `Y = oobPredict(B)`  
`Y = oobPredict(B, 'param1', val1, 'param2', val2, ...)`

**Description** `Y = oobPredict(B)` computes predicted responses using the trained bagger `B` for out-of-bag observations in the training data. The output has one prediction for each observation in the training data. The returned `Y` is a cell array of strings for classification and a numeric array for regression.

`Y = oobPredict(B, 'param1', val1, 'param2', val2, ...)` specifies optional parameter name/value pairs:

`'trees'` Array of tree indices to use for computation of responses. Default is `'all'`.

`'treeweights'` Array of `N`Tree weights for weighting votes from the specified trees.

**See Also** `CompactTreeBagger.predict` | `OOBIndices`

# ordinal

---

**Superclasses**    categorical

**Purpose**    Arrays for ordinal categorical data

**Description**    Ordinal arrays are used to store discrete values that have an ordering but are not numeric. An ordinal array provides efficient storage and convenient manipulation of such data, while also maintaining meaningful labels for the values. Ordinal arrays are often used as grouping variables.

Like a numerical array, an ordinal array can have any size or dimension. You can subscript, concatenate, reshape, sort, etc. ordinal arrays, much like ordinary numeric arrays. You can make comparisons between elements of two ordinal arrays, or between an ordinal array and a single string representing a ordinal value.

**Construction**    Use the `ordinal` constructor to create an ordinal array from a numeric, logical, or character array, or from a cell array of strings.

<code>ordinal</code>	Construct ordinal categorical array
----------------------	-------------------------------------

**Methods**    Each ordinal array carries along a list of possible values that it can store, known as its levels. The list is created when you create an ordinal array, and you can access it using the `getlevels` method, or modify it using the `addlevels`, `mergelevels`, or `droplevels` methods. Assignment to the array will also add new levels automatically if the values assigned are not already levels of the array. The ordering on values stored in an ordinal array is defined by the order of the list of levels. You can change that order using the `reorderlevels` method.

The following table lists operations available for ordinal arrays.

sort	Sort elements of ordinal array
sortrows	Sort rows
summary	Summary statistics for ordinal array

### Inherited Methods

Methods in the following table are inherited from categorical.

addlevels	Add levels to categorical array
cat	Concatenate categorical arrays
cellstr	Convert categorical array to cell array of strings
char	Convert categorical array to character array
circshift	Shift categorical array circularly
ctranspose	Transpose categorical matrix
disp	Display categorical array
display	Display categorical array
double	Convert categorical array to double array
droplevels	Drop levels
end	Last index in indexing expression for categorical array
flipdim	Flip categorical array along specified dimension
fliplr	Flip categorical matrix in left/right direction
flipud	Flip categorical matrix in up/down direction

<code>getlabels</code>	Access categorical array labels
<code>getlevels</code>	Get categorical array levels
<code>hist</code>	Plot histogram of categorical data
<code>horzcat</code>	Horizontal concatenation for categorical arrays
<code>int16</code>	Convert categorical array to signed 16-bit integer array
<code>int32</code>	Convert categorical array to signed 32-bit integer array
<code>int64</code>	Convert categorical array to signed 64-bit integer array
<code>int8</code>	Convert categorical array to signed 8-bit integer array
<code>intersect</code>	Set intersection for categorical arrays
<code>ipermute</code>	Inverse permute dimensions of categorical array
<code>isempty</code>	True for empty categorical array
<code>isequal</code>	True if categorical arrays are equal
<code>islevel</code>	Test for levels
<code>ismember</code>	True for elements of categorical array in set
<code>isscalar</code>	True if categorical array is scalar
<code>isundefined</code>	Test for undefined elements
<code>isvector</code>	True if categorical array is vector
<code>length</code>	Length of categorical array
<code>levelcounts</code>	Element counts by level



---

<code>mergelevels</code>	Merge levels
<code>ndims</code>	Number of dimensions of categorical array
<code>numel</code>	Number of elements in categorical array
<code>permute</code>	Permute dimensions of categorical array
<code>reorderlevels</code>	Reorder levels
<code>repmat</code>	Replicate and tile categorical array
<code>reshape</code>	Resize categorical array
<code>rot90</code>	Rotate categorical matrix 90 degrees
<code>setdiff</code>	Set difference for categorical arrays
<code>setlabels</code>	Label levels
<code>setxor</code>	Set exclusive-or for categorical arrays
<code>shiftdim</code>	Shift dimensions of categorical array
<code>single</code>	Convert categorical array to single array
<code>size</code>	Size of categorical array
<code>squeeze</code>	Squeeze singleton dimensions from categorical array
<code>subsasgn</code>	Subscripted assignment for categorical array
<code>subsindex</code>	Subscript index for categorical array

# ordinal

---

suboref	Subscripted reference for categorical array
times	Product of categorical arrays
transpose	Transpose categorical matrix
uint16	Convert categorical array to unsigned 16-bit integers
uint32	Convert categorical array to unsigned 32-bit integers
uint64	Convert categorical array to unsigned 64-bit integers
uint8	Convert categorical array to unsigned 8-bit integers
union	Set union for categorical arrays
unique	Unique values in categorical array
vertcat	Vertical concatenation for categorical arrays

## Properties

## Inherited Properties

Properties in the following table are inherited from `categorical`.

labels	Text labels for levels
undeflabel	Text label for undefined levels

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## Examples

### Create and Compare Ordinal Arrays

Create an ordinal array from integer data.

```
quality = ordinal([1 2 3; 3 2 1; 2 1 3],{'low' 'medium' 'high'})
```

```
quality =
```

low	medium	high
high	medium	low
medium	low	high

Identify the elements in `quality` that are members of a level that is greater than or equal to `'medium'`.

```
quality >= 'medium'
```

```
ans =
```

0	1	1
1	1	0
1	0	1

A value of 1 in the resulting array `ans` indicates that the corresponding element of `quality` is a member of a level that is greater than or equal to `'medium'`.

Identify the elements of `quality` that are members of either `'low'` or `'high'`.

```
ismember(quality,{'low' 'high'})
```

```
ans =
```

1	0	1
1	0	1
0	1	1

A value of 1 in the resulting array `ans` indicates that the corresponding element of `quality` is a member of either `'low'` or `'high'`.

Create a second nominal array `quality2` by flipping `quality` from left to right.

# ordinal

---

```
quality2 = fliplr(quality)
```

```
quality2 =
```

```
      high      medium      low
low      medium      high
high      low      medium
```

Compare the elements of `quality` and `quality2`.

```
quality == quality2
```

```
ans =
```

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

A value of 1 in the resulting array `ans` indicates that the corresponding element of `quality` is a member of the same level as the corresponding element of `quality2`.

## References

[1] Johnson, N. L., S. Kotz, and A. W. Kemp, *Univariate Discrete Distributions*, 2nd edition, Wiley, 1992, pp. 124-130.

## See Also

`histc` | `nominal`

**Purpose**

Construct ordinal categorical array

**Syntax**

```
B = ordinal(A)
B = ordinal(A,labels)
B = ordinal(A,labels,levels)
B = ordinal(A,labels,[],edges)
```

**Description**

`B = ordinal(A)` creates an ordinal array `B` from the array `A`. `A` can be numeric, logical, character, categorical, or a cell array of strings. `ordinal` creates the levels of `B` from the sorted unique values in `A`, and creates default labels for them.

`B = ordinal(A,labels)` labels the levels in `B` using the character array or cell array of strings `labels`. `ordinal` assigns labels to levels in `B` in order according to the sorted unique values in `A`.

`B = ordinal(A,labels,levels)` creates an ordinal array with possible levels defined by `levels`. `levels` is a vector whose values can be compared to those in `A` using the equality operator. `ordinal` assigns labels to each level from the corresponding elements of `labels`. If `A` contains any values not present in `levels`, the levels of the corresponding elements of `B` are undefined. Use `[]` for `labels` to allow `ordinal` to create default labels.

`B = ordinal(A,labels,[],edges)` creates an ordinal array by binning the numeric array `A`, with bin edges given by the numeric vector `edges`. The uppermost bin includes values equal to the right-most edge. `ordinal` assigns labels to each level in `B` from the corresponding elements of `labels`. `edges` must have one more element than `labels`.

By default, an element of `B` is undefined if the corresponding element of `A` is NaN (when `A` is numeric), an empty string (when `A` is character), or undefined (when `A` is categorical). `ordinal` treats such elements as “undefined” or “missing” and does not include entries for them among the possible levels for `B`. To create an explicit level for such elements instead of treating them as undefined, you must use the `levels` input, and include NaN, the empty string, or an undefined element.

# ordinal

---

You may include duplicate labels in `labels` in order to merge multiple values in A into a single level in B.

## Examples **Create and Label Ordinal Arrays**

Create an ordinal array from integer data, and provide explicit labels.

```
quality1 = ordinal([1 2 3; 3 2 1; 2 1 3],...  
                  {'low' 'medium' 'high'})
```

```
quality1 =
```

```
      low      medium      high  
high      medium      low  
medium      low      high
```

Create an ordinal array from integer data, and provide both explicit labels and an explicit order.

```
quality2 = ordinal([1 2 3; 3 2 1; 2 1 3],...  
                  {'high' 'medium' 'low'},[3 2 1])
```

```
quality2 =
```

```
      low      medium      high  
high      medium      low  
medium      low      high
```

Create an ordinal array by binning floating point values.

```
size = ordinal(rand(5,2),{'small' 'medium' 'large'},...  
                [],[0 1/3 2/3 1])
```

```
size =
```

```
      small      small  
      large      medium  
      large      large
```

```
    medium    large
    large     large
```

## Create Ordinal Array Using Sample Data

Load the sample data.

```
load fisheriris;
```

Create an ordinal array from the measurements in Fisher's iris data, ignoring decimal lengths.

```
m = floor(min(meas(:)));
M = floor(max(meas(:)));
labels = num2str((m:M)');
edges = m:M+1;
cms = ordinal(meas,labels,[],edges);
```

Display the first five rows of `meas`, which contains actual flower measurements.

```
meas(1:5,:)
```

```
ans =
```

```
    5.1000    3.5000    1.4000    0.2000
    4.9000    3.0000    1.4000    0.2000
    4.7000    3.2000    1.3000    0.2000
    4.6000    3.1000    1.5000    0.2000
    5.0000    3.6000    1.4000    0.2000
```

Display the first five rows of the ordinal array `cms`.

```
cms(1:5,:)
```

```
ans =
```

```
    5     3     1     0
    4     3     1     0
```

# ordinal

---

```
4     3     1     0
4     3     1     0
5     3     1     0
```

## Create Groups in Ordinal Array

Load the sample data.

```
load hospital;
```

Create an age group ordinal array from the patients' age data.

```
edges = 0:10:100;
labels = strcat(num2str((0:10:90)', '%d'), {'s'});
AgeGroup = ordinal(hospital.Age, labels, [], edges);
```

Display the first five elements of the ordinal array AgeGroup.

```
AgeGroup(1:5)
```

```
ans =
```

```
30s
40s
30s
40s
40s
```

## See Also

[histc](#) | [nominal](#)



## Purpose

Outlier measure for data

## Syntax

```
out = outlierMeasure(B,X)
out = outlierMeasure(B,X,'param1',val1,'param2',val2,...)
```

## Description

`out = outlierMeasure(B,X)` computes outlier measures for predictors `X` using trees in the ensemble `B`. The method computes the outlier measure for a given observation by taking an inverse of the average squared proximity between this observation and other observations. `outlierMeasure` then normalizes these outlier measures by subtracting the median of their distribution, taking the absolute value of this difference, and dividing by the median absolute deviation. A high value of the outlier measure indicates that this observation is an outlier.

You can supply the proximity matrix directly by using the `'data'` parameter.

`out = outlierMeasure(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

<code>'data'</code>	Flag indicating how to treat the <code>X</code> input argument. If set to <code>'predictors'</code> (default), the method assumes <code>X</code> is a matrix of predictors and uses it for computation of the proximity matrix. If set to <code>'proximity'</code> , the method treats <code>X</code> as a proximity matrix returned by the <code>proximity</code> method. If you do not supply the proximity matrix, <code>outlierMeasure</code> computes it internally. If you use the <code>proximity</code> method to compute a proximity matrix, supplying it as input to <code>outlierMeasure</code> reduces computing time.
<code>'labels'</code>	Vector of true class labels. True class labels can be either a numeric vector, character matrix, or cell array of strings. When you supply this parameter, the method performs the outlier calculation for any observations using only other observations from the same class. This parameter must specify one label for each observation (row) in <code>X</code> .

# CompactTreeBagger.outlierMeasure

---

## See Also

`proximity`

# TreeBagger.OutlierMeasure property

---

**Purpose** Measure for determining outliers

**Description** The OutlierMeasure property is a numeric array of size Nobs-by-1, where Nobs is the number of observations in the training data, containing outlier measures for each observation.

**See Also** CompactTreeBagger.outlierMeasure

# parallelcoords

---

**Purpose** Parallel coordinates plot

**Syntax**

```
parallelcoords(X)
parallelcoords(X,...,'Standardize','on')
parallelcoords(X,...,'Standardize','PCA')
parallelcoords(X,...,'Standardize','PCAStd')
parallelcoords(X,...,'Quantile',alpha)
parallelcoords(X,...,'Group',group)
parallelcoords(X,...,'Labels',labels)
parallelcoords(X,...,PropertyName,PropertyValue,...)
h = parallelcoords(X,...)
parallelcoords(axes,...)
```

**Description** `parallelcoords(X)` creates a parallel coordinates plot of the multivariate data in the  $n$ -by- $p$  matrix  $X$ . Rows of  $X$  correspond to observations, columns to variables. A parallel coordinates plot is a tool for visualizing high dimensional data, where each observation is represented by the sequence of its coordinate values plotted against their coordinate indices. `parallelcoords` treats NaNs in  $X$  as missing values and does not plot those coordinate values.

`parallelcoords(X,...,'Standardize','on')` scales each column of  $X$  to have mean 0 and standard deviation 1 before making the plot.

`parallelcoords(X,...,'Standardize','PCA')` creates a parallel coordinates plot from the principal component scores of  $X$ , in order of decreasing eigenvalues. `parallelcoords` removes rows of  $X$  containing missing values (NaNs) for principal components analysis (PCA) standardization.

`parallelcoords(X,...,'Standardize','PCAStd')` creates a parallel coordinates plot using the standardized principal component scores.

`parallelcoords(X,...,'Quantile',alpha)` plots only the median and the  $\alpha$  and  $1-\alpha$  quantiles of  $f(t)$  at each value of  $t$ . This is useful if  $X$  contains many observations.

`parallelcoords(X,...,'Group',group)` plots the data in different groups with different colors. Groups are defined by `group`, a numeric

array containing a group index for each observation. `group` can also be a categorical variable, character matrix, or cell array of strings, containing a group name for each observation.

`parallelcoords(X,...,'Labels',labels)` labels the coordinate tick marks along the horizontal axis using `labels`, a character array or cell array of strings.

`parallelcoords(X,...,PropertyName,PropertyValue,...)` sets properties to the specified property values for all line graphics objects created by `parallelcoords`.

`h = parallelcoords(X,...)` returns a column vector of handles to the line objects created by `parallelcoords`, one handle per row of `X`. If you use the `'Quantile'` input argument, `h` contains one handle for each of the three lines objects created. If you use both the `'Quantile'` and the `'Group'` input arguments, `h` contains three handles for each group.

`parallelcoords(axes,...)` plots into the axes with handle axes.

## Examples

```
% Make a grouped plot of the raw data
load fisheriris
labels = {'Sepal Length','Sepal Width',...
         'Petal Length','Petal Width'};
parallelcoords(meas,'group',species,'labels',labels);
```

```
% Plot only the median and quartiles of each group
parallelcoords(meas,'group',species,'labels',labels,...
              'quantile',.25);
```

## See Also

`andrewsplot` | `glyphplot`

## How To

- “Grouping Variables” on page 2-51

# ProbDistUnivParam.paramci

---

**Purpose** Return parameter confidence intervals of ProbDistUnivParam object

**Syntax** CI = paramci(PD)  
CI = paramci(PD, Alpha)

**Description** CI = paramci(PD) returns CI, a 2-by-N array containing 95% confidence intervals for the parameters of the ProbDistUnivParam object PD. N is the number of parameters in the distribution. When you create PD by specifying parameters (such as using the ProbDistUnivParam.ProbDistUnivParam constructor or using the fitdist function and specifying a 'binomial' or 'generalized pareto' distribution) rather than by fitting to data, the confidence intervals have a width of 0 because the parameters are viewed as estimates of an unknown parameter.

CI = paramci(PD, Alpha) returns 100\*(1 - Alpha)% confidence intervals. Default Alpha is 0.05, which specifies 95% confidence intervals.

---

**Note** If you create PD with a distribution that does not support confidence intervals, then CI contains NaN values.

---

**Input Arguments**

PD	An object of the class ProbDistUnivParam.
Alpha	A value between 0 and 1 that specifies a confidence interval. Default is 0.05, which specifies 95% confidence intervals.

**Output Arguments**

CI	A 2-by-N array containing 100*(1 - Alpha)% confidence intervals for the parameters of the ProbDistUnivParam object PD. N is the number of parameters in the distribution.
----	---

**See Also**     `fitdist`

# prob.ToolboxFittableParametricDistribution.paramci

---

**Purpose** Confidence intervals for probability distribution parameters

**Syntax**  
`ci = paramci(pd)`  
`ci = paramci(pd,Name,Value)`

**Description**  
`ci = paramci(pd)` returns the array `ci` containing the lower and upper boundaries of the 95% confidence interval for each parameter in probability distribution `pd`.  
`ci = paramci(pd,Name,Value)` returns confidence intervals with additional options specified by one or more name-value pair arguments. For example, you can specify a different percentage for the confidence interval, or compute confidence intervals only for selected parameters.

## Input Arguments

**pd - Probability distribution**  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'Alpha',0.01` specifies a 99% confidence interval.

## 'Alpha' - Alpha level

0.05 (default) | scalar value in the range (0,1)

Alpha level for the confidence interval, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value



in the range (0,1). The default value 0.05 corresponds to a 95% confidence interval.

**Example:** 'Alpha',0.01

## Data Types

single | double

## 'Parameter' - Parameter list

vector | cell array of strings

Parameter list for which to compute confidence intervals, specified as the comma-separated pair consisting of 'Parameter' and a vector or a cell array of strings containing the parameter names. By default, paramci computes confidence intervals for all distribution parameters.

**Example:** 'Parameter', 'mu'

## Data Types

char

## 'Type' - Computation method

'exact' | 'Wald' | 'lr'

Computation method for the confidence intervals, specified as the comma-separated pair consisting of 'Type' and 'exact', 'Wald', or 'lr'.

'exact' computes the confidence intervals using an exact method, and is available for the following distributions.

Binomial	Compute using the Clopper-Pearson method based on exact probability calculations. This method does not provide exact coverage probabilities.
Exponential	Compute using a method based on a chi-square distribution. This method provides exact coverage for complete and Type 2 censored samples.

# prob.ToolboxFittableParametricDistribution.paramci

---

Normal	Computation method based on $t$ and chi-square distributions for uncensored samples provides exact coverage for uncensored samples. For censored samples, <code>paramci</code> uses the Wald method if <code>Type</code> is <code>exact</code> .
Lognormal	Computation method based on $t$ and chi-square distributions for uncensored samples provides exact coverage. For censored samples, <code>paramci</code> uses the Wald method if <code>Type</code> is <code>exact</code> .
Poisson	Computation method based on a chi-square distribution provides exact coverage. For large degrees of freedom, the chi-square is approximated by a normal distribution for numerical efficiency.
Rayleigh	Computation method based on a chi-square distribution provides exact coverage probabilities.

'exact' is the default when it is available. Alternatively, you can specify 'Wald' to compute the confidence intervals using the Wald method, or 'lr' to compute the confidence intervals using the likelihood ratio method.

**Example:** 'Type', 'Wald'

## 'LogFlag' - Boolean flag for log scale

vector

Boolean flag for the log scale, specified as the comma-separated pair consisting of 'LogFlag' and a vector containing Boolean values corresponding to each distribution parameter. The flag specifies which Wald intervals to compute on a log scale. The default values depend on the distribution.

**Example:** 'LogFlag', [0, 1]

## Data Types

logical

## Output Arguments

### ci - Confidence interval

array

Confidence interval, returned as a  $p$ -by-2 array containing the lower and upper bounds of the  $(1 - \text{Alpha})\%$  confidence interval for each distribution parameter.  $p$  is the number of distribution parameters.

## Examples

### Parameter Confidence Intervals

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x,'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083 [73.4321, 76.5846]  
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the 95% confidence interval for the distribution parameters.

```
ci = paramci(pd)
```

```
ci =
```

```
73.4321 7.7391  
76.5846 9.9884
```

# prob.ToolboxFittableParametricDistribution.paramci

---

Column 1 of `ci` contains the lower and upper 95% confidence interval boundaries for the `mu` parameter, and column 2 contains the boundaries for the `sigma` parameter.

## Change Parameter Confidence Intervals

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x,'Normal')  
  
pd =  
  
NormalDistribution  
  
Normal distribution  
    mu = 75.0083    [73.4321, 76.5846]  
    sigma =  8.7202    [7.7391, 9.98843]
```

Compute the 99% confidence interval for the distribution parameters.

```
ci = paramci(pd,'Alpha',.01)  
  
ci =  
  
    72.9245    7.4627  
    77.0922   10.4403
```

Column 1 of `ci` contains the lower and upper 99% confidence interval boundaries for the `mu` parameter, and column 2 contains the boundaries for the `sigma` parameter.

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

# ProbDistParametric.ParamCov property

---

<b>Purpose</b>	Read-only covariance matrix of parameter estimates of ProbDistParametric object
<b>Description</b>	ParamCov is a read-only property of the ProbDistParametric class. ParamCov is a covariance matrix containing the parameter estimates of a distribution represented by a ProbDistParametric object. ParamCov has a size of NumParams-by-NumParams.
<b>Values</b>	This covariance matrix includes estimates for both the specified parameters and parameters that are fit to the data. For specified parameters, the covariance is 0, indicating the parameter is known exactly. Use this information to view and compare the descriptions of parameters supplied to create distributions.

# ProbDistParametric.ParamDescription property

---

<b>Purpose</b>	Read-only cell array specifying descriptions of parameters of ProbDistParametric object
<b>Description</b>	ParamDescription is a read-only property of the ProbDistParametric class. ParamDescription is a cell array of strings specifying the descriptions or meanings of the parameters of a distribution represented by a ProbDistParametric object. ParamDescription has a length of NumParams.
<b>Values</b>	This cell array includes a brief description of the meaning of both the specified parameters and parameters that are fit to the data. The description is the same as the parameter name when no further description information is available. Use this information to view and compare the descriptions of parameters used to create distributions.

# ProbDistParametric.ParamIsFixed property

---

<b>Purpose</b>	Read-only logical array specifying fixed parameters of ProbDistParametric object
<b>Description</b>	ParamIsFixed is a read-only property of the ProbDistParametric class. ParamIsFixed is a logical array specifying the fixed parameters of a distribution represented by a ProbDistParametric object. ParamIsFixed has a length of NumParams.
<b>Values</b>	This array specifies a 1 (true) for fixed parameters, and a 0 (false) for parameters that are estimated from the input data. Use this information to view and compare the fixed parameters used to create distributions.

# ProbDistParametric.ParamNames property

---

<b>Purpose</b>	Read-only cell array specifying names of parameters of ProbDistParametric object
<b>Description</b>	ParamNames is a read-only property of the ProbDistParametric class. ParamNames is a cell array of strings specifying the names of the parameters of a distribution represented by a ProbDistParametric object. ParamNames has a length of NumParams.
<b>Values</b>	This cell array includes the names of both the specified parameters and parameters that are fit to the data. Use this information to view and compare the names of parameters used to create distributions.



## Purpose

Parameter estimates

## Description

The Params property is an NClasses-by-NDims cell array containing the parameter estimates, excluding the class priors. Params(i, j) contains the parameter estimates for the jth feature in the ith class. Params(i, j) is an empty cell if the ith class is empty.

The entry in Params(i, j) depends on the distribution type used for the jth feature, as follows:

'normal'	A vector of length two. The first element is the mean, and the second element is standard deviation.
'kernel'	A ProbDistUnivKernel object
'mvmn'	A vector containing the probability for each possible value of the jth feature in the ith class. The order of the probabilities is decided by the sorted order of all the unique values of the jth feature.
'mn'	A scalar representing the probability the jth token appearing in the ith class, Prob(token j   class i). It is estimated as $(1 + \text{the number of occurrence of token J in class I}) / (\text{NDims} + \text{the total number of token occurrence in class I})$ .

# ProbDistParametric.Params property

---

<b>Purpose</b>	Read-only array specifying values of parameters of ProbDistParametric object
<b>Description</b>	Params is a read-only property of the ProbDistParametric class. Params is an array of values specifying the values of the parameters of a distribution represented by a ProbDistParametric object. Params has a length of NumParams.
<b>Values</b>	This array includes the values of both the specified parameters and parameters that are fit to the data. Use this information to view and compare the values of parameters used to create distributions.

# prob.ParametricTruncatableDistribution

---

<b>Superclasses</b>	TruncatableDistribution	
<b>Purpose</b>	Parametric truncatable probability distribution object	
<b>Description</b>	Create a probability distribution object with specified parameter values using <code>makedist</code> .	
<b>Methods</b>	<code>mean</code>	Mean of probability distribution object
	<code>std</code>	Standard deviation of probability distribution object
	<code>var</code>	Variance of probability distribution object
	<b>Inherited Methods</b>	
	<code>cdf</code>	Cumulative distribution function of probability distribution object
	<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
	<code>iqr</code>	Interquartile range of probability distribution object
	<code>median</code>	Median of probability distribution object
	<code>pdf</code>	Probability density function of probability distribution object
	<code>random</code>	Generate random numbers from probability distribution object
	<code>truncate</code>	Truncate probability distribution object

# prob.ParametricTruncatableDistribution

---

**See Also** `makedist`

**Concepts**

- Class Attributes
- Property Attributes

## Purpose

Parent node

## Syntax

```
p = parent(t)
p = parent(t,nodes)
```

## Description

`p = parent(t)` returns an  $n$ -element vector `p` containing the number of the parent node for each node in the tree `t`, where  $n$  is the number of nodes. The parent of the root node is 0.

`p = parent(t,nodes)` takes a vector `nodes` of node numbers and returns the parent nodes for the specified nodes.

## Examples

Create a classification tree for Fisher's iris data:

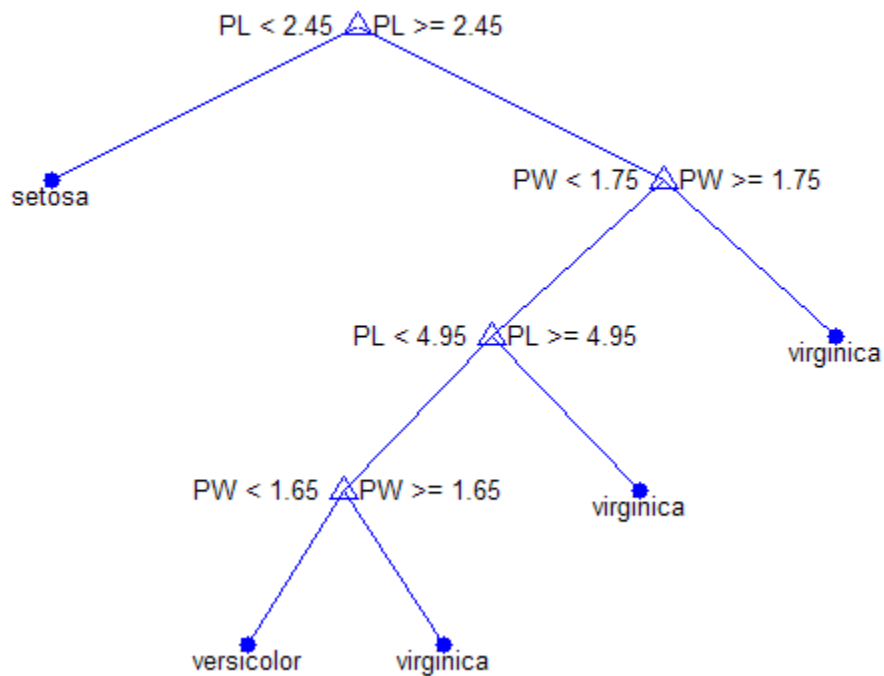
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 else node 3
2  class = setosa
3  if PW<1.75 then node 4 else node 5
4  if PL<4.95 then node 6 else node 7
5  class = virginica
6  if PW<1.65 then node 8 else node 9
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

# classregtree.parent

Click to display:  Magnification:  Pruning level:



p = parent(t)

p =  
0  
1  
1  
3  
3  
4  
4  
6

6

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `children` | `numnodes`

# paretotails

---

<b>Superclasses</b>	piecewisedistribution	
<b>Purpose</b>	Empirical distributions with Pareto tails	
<b>Construction</b>	paretotails	Construct Pareto tails object
<b>Methods</b>	lowerparams upperparams	Lower Pareto tails parameters Upper Pareto tails parameters

## Inherited Methods

Methods in the following table are inherited from `piecewisedistribution`.

boundary	Piecewise distribution boundaries
cdf	Cumulative distribution function for piecewise distribution
disp	Display <code>piecewisedistribution</code> object
display	Display <code>piecewisedistribution</code> object
icdf	Inverse cumulative distribution function for piecewise distribution
nsegments	Number of segments
pdf	Probability density function for piecewise distribution
random	Random numbers from piecewise distribution
segment	Segments containing values



## **Properties**

Objects of the `paretotails` class have no properties accessible by dot indexing, `get` methods, or `set` methods. To obtain information about a `paretotails` object, use the appropriate method.

## **Copy Semantics**

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## **How To**

- “Generalized Pareto Distribution” on page B-53

# paretotails

---

**Purpose** Construct Pareto tails object

**Syntax**  
`obj = paretotails(x,p1,pu)`  
`obj = paretotails(x,p1,pu,cdf_fun)`

**Description** `obj = paretotails(x,p1,pu)` creates an object `obj` defining a distribution consisting of the empirical distribution of `x` in the center and Pareto distributions in the tails. `x` is a real-valued vector of data values whose extreme observations are fit to generalized Pareto distributions (GPDs). `p1` and `pu` identify the lower- and upper-tail cumulative probabilities such that  $100 \cdot p1$  and  $100 \cdot (1 - pu)$  percent of the observations in `x` are, respectively, fit to a GPD by maximum likelihood. If `p1` is 0, or if there are not at least two distinct observations in the lower tail, then no lower Pareto tail is fit. If `pu` is 1, or if there are not at least two distinct observations in the upper tail, then no upper Pareto tail is fit.

`obj = paretotails(x,p1,pu,cdf_fun)` uses `cdf_fun` to estimate the cdf of `x` between the lower and upper tail probabilities. `cdf_fun` may be any of the following:

- 'ecdf' — Uses an interpolated empirical cdf, with data values as the midpoints in the vertical steps in the empirical cdf, and computed by linear interpolation between data values. This is the default.
- 'kernel' — Uses a kernel-smoothing estimate of the cdf.
- @fun — Uses a handle to a function of the form `[p,xi] = fun(x)` that accepts the input data vector `x` and returns a vector `p` of cdf values and a vector `xi` of evaluation points. Values in `xi` must be sorted and distinct but need not equal the values in `x`.

`cdf_fun` is used to compute the quantiles corresponding to `p1` and `pu` by inverse interpolation, and to define the fitted distribution between these quantiles.

The output object `obj` is a Pareto tails object with methods to evaluate the cdf, inverse cdf, and other functions of the fitted distribution. These methods are well-suited to copula and other Monte Carlo simulations.

The pdf method in the tails is the GPD density, but in the center it is computed as the slope of the interpolated cdf.

The `paretotails` class is a subclass of the `piecewisedistribution` class, and many of its methods are derived from that class.

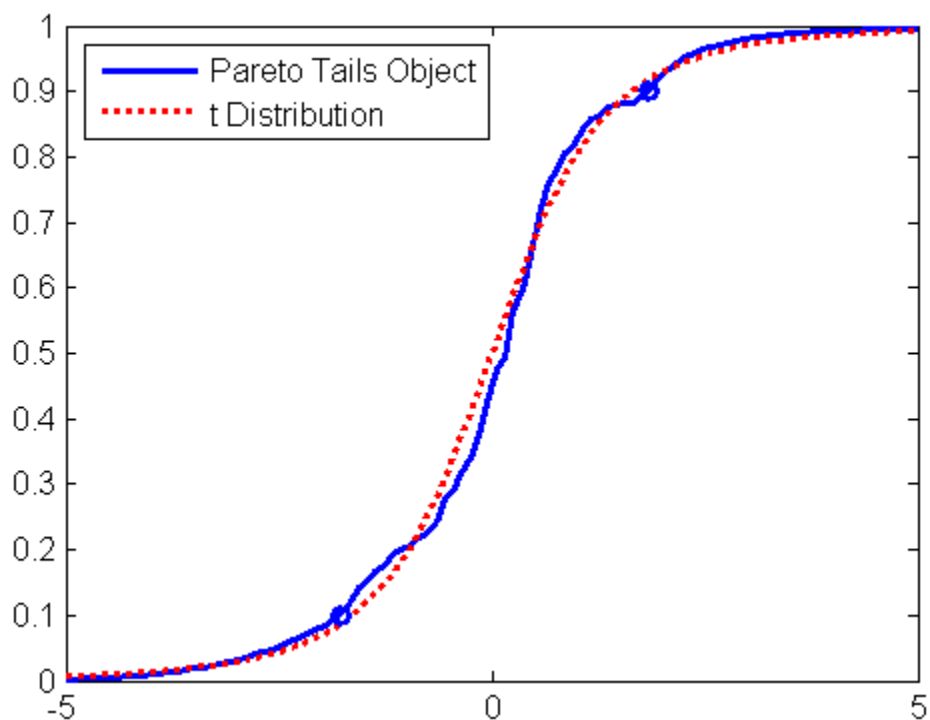
## Examples

Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj);

x = linspace(-5,5);
plot(x,cdf(obj,x),'b-','LineWidth',2)
hold on
plot(x,tcdf(x,3),'r:','LineWidth',2)
plot(q,p,'bo','LineWidth',2,'MarkerSize',5)
legend('Pareto Tails Object','t Distribution',...
       'Location','NW')
```

# paretotails



## See Also

[cdf](#) | [ecdf](#) | [gpfit](#) | [icdf](#) | [ksdensity](#)

**Purpose** Linear or rank partial correlation coefficients

**Syntax**

```
rho = partialcorr(x)
rho = partialcorr(x,z)
rho = partialcorr(x,y,z)
rho = partialcorr( __ ,Name,Value)
[rho,pval] = partialcorr( __ )
```

**Description** `rho = partialcorr(x)` returns the sample linear partial correlation coefficients between pairs of variables in `x`, controlling for the remaining variables in `x`.

`rho = partialcorr(x,z)` returns the sample linear partial correlation coefficients between pairs of variables in `x`, controlling for the variables in `z`.

`rho = partialcorr(x,y,z)` returns the sample linear partial correlation coefficients between pairs of variables in `x` and `y`, controlling for the variables in `z`.

`rho = partialcorr( __ ,Name,Value)` returns the sample linear partial correlation coefficients with additional options specified by one or more name-value pair arguments, using input arguments from any of the previous syntaxes. For example, you can specify whether to use Pearson or Spearman partial correlations, or specify how to treat missing values.

`[rho,pval] = partialcorr( __ )` also returns a matrix `pval` of  $p$ -values for testing the hypothesis of no partial correlation against the one- or two-sided alternative that there is a nonzero partial correlation.

# partialcorr

---

## Input Arguments

### **x - Data matrix**

matrix

Data matrix, specified as an  $n$ -by- $p_x$  matrix. The rows of  $x$  correspond to observations, and the columns correspond to variables.

### **Data Types**

single | double

### **y - Data matrix**

matrix

Data matrix, specified as an  $n$ -by- $p_y$  matrix. The rows of  $y$  correspond to observations, and the columns correspond to variables.

### **Data Types**

single | double

### **z - Data matrix**

matrix

Data matrix, specified as an  $n$ -by- $p_z$  matrix. The rows of  $z$  correspond to observations, and columns correspond to variables.

### **Data Types**

single | double

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

**Example:** 'Type', 'Spearman', 'Rows', 'complete' computes Spearman partial correlations using only the data in rows that contain no missing values.

### **'Type' - Type of partial correlations**

'Pearson' (default) | 'Spearman'

Type of partial correlations to compute, specified as the comma-separated pair consisting of 'Type' and one of the following.

'Pearson'	Compute Pearson (linear) partial correlations.
'Spearman'	Compute Spearman (rank) partial correlations.

**Example:** 'Type', 'Spearman'

**'Rows' - Rows to use in computation**

'all' (default) | 'complete' | 'pairwise'

Rows to use in computation, specified as the comma-separated pair consisting of 'Rows' and one of the following.

'all'	Use all rows regardless of missing (NaN) values.
'complete'	Use only rows with no missing values.
'pairwise'	Compute $\rho(i, j)$ using rows with no missing values in column $i$ or $j$ .

Using 'pairwise' can produce a  $\rho$  that is not positive definite. 'complete' always produces a positive definite  $\rho$ , but when data is missing, the estimates are generally based on fewer observations.

**Example:** 'Rows', 'complete'

**'Tail' - Alternative hypothesis**

'both' (default) | 'right' | 'left'

Alternative hypothesis to test against, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'	Test the alternative hypothesis that the correlation is not 0.
'right'	Test the alternative hypothesis that the correlation is greater than 0.
'left'	Test the alternative hypothesis that the correlation is less than 0.

Example: 'Tail', 'right'

## Output Arguments

### **rho** - Sample linear partial correlation coefficients

matrix

Sample linear partial correlation coefficients, returned as a matrix.

- If you input only an  $\mathbf{x}$  matrix,  $\mathbf{rho}$  is a symmetric  $p_x$ -by- $p_x$  matrix. The  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ -th and  $j$ -th columns in  $\mathbf{x}$ .
- If you input  $\mathbf{x}$  and  $\mathbf{z}$  matrices,  $\mathbf{rho}$  is a symmetric  $p_x$ -by- $p_x$  matrix. The  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th and  $j$ th columns in  $\mathbf{x}$ , controlled for the variables in  $\mathbf{z}$ .
- If you input  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mathbf{z}$  matrices,  $\mathbf{rho}$  is a  $p_x$ -by- $p_y$  matrix, where the  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th column in  $\mathbf{x}$  and the  $j$ th column in  $\mathbf{y}$ , controlled for the variables in  $\mathbf{z}$ .

If the covariance matrix of  $[\mathbf{x}, \mathbf{z}]$  is

$$S = \begin{pmatrix} S_{xx} & S_{xz} \\ S_{xz}^T & S_{zz} \end{pmatrix},$$

then the partial correlation matrix of  $\mathbf{x}$ , controlling for  $\mathbf{z}$ , can be defined formally as a normalized version of the covariance matrix  $S_{xy} = S_{xx} - (S_{xz} S_{zz}^{-1} S_{xz}^T)$

### **pval** - p-values

matrix



$p$ -values, returned as a matrix. Each element of `pval` is the  $p$ -value for the corresponding element of `rho`.

If `pval(i, j)` is small, then the corresponding partial correlation `rho(i, j)` is statistically significantly different from 0.

`partialcorr` computes  $p$ -values for linear and rank partial correlations using a Student's  $t$  distribution for a transformation of the correlation. This is exact for linear partial correlation when  $x$  and  $z$  are normal, but is a large-sample approximation otherwise.

## Examples

### Compute Partial Correlation Coefficients

Compute partial correlation coefficients between pairs of variables in the input matrix.

Load the sample data. Convert the strings in `hospital.Sex` to numeric group identifiers.

```
load hospital;
hospital.SexID = grp2idx(hospital.Sex);
```

Create an input matrix containing the sample data.

```
x = [hospital.SexID hospital.Age hospital.Smoker hospital.Weight];
```

Each row in `x` contains a patient's gender, age, smoking status, and weight.

Compute partial correlation coefficients between pairs of variables in `x`, while controlling for the effects of the remaining variables in `x`.

```
rho = partialcorr(x)
```

```
rho =
```

```

    1.0000    -0.0105    0.0273    0.9421
   -0.0105     1.0000    0.0419    0.0369
    0.0273     0.0419     1.0000    0.0451
    0.9421     0.0369     0.0451     1.0000
```

The matrix `rho` indicates, for example, a correlation of 0.9421 between gender and weight after controlling for all other variables in `x`. You can return the  $p$ -values as a second output, and examine them to confirm whether these correlations are statistically significant.

For a clearer display, create a table with appropriate variable and row labels.

```
rho = array2table(rho, ...  
    'VariableNames', {'SexID', 'Age', 'Smoker', 'Weight'}, ...  
    'RowNames', {'SexID', 'Age', 'Smoker', 'Weight'});
```

```
disp('Partial Correlation Coefficients')  
disp(rho)
```

```
Partial Correlation Coefficients  
                SexID      Age      Smoker      Weight  
-----  
SexID           1      -0.01052    0.027324    0.9421  
Age      -0.01052           1      0.041945    0.036873  
Smoker    0.027324    0.041945           1      0.045106  
Weight    0.9421      0.036873    0.045106           1
```

## Test for Partial Correlations with Controlled Variables

Test for partial correlation between pairs of variables in the input matrix, while controlling for the effects of a second set of variables.

Load the sample data. Convert the strings in `hospital.Sex` to numeric group identifiers.

```
load hospital;  
hospital.SexID = grp2idx(hospital.Sex);
```

Create two matrices containing the sample data.

```
x = [hospital.Age hospital.BloodPressure];  
z = [hospital.SexID hospital.Smoker hospital.Weight];
```

The `x` matrix contains the variables to test for partial correlation. The `z` matrix contains the variables to control for. The measurements for `BloodPressure` are contained in two columns: The first column contains the upper (systolic) number, and the second column contains the lower (diastolic) number. `partialcorr` treats each column as a separate variable.

Test for partial correlation between pairs of variables in `x`, while controlling for the effects of the variables in `z`. Compute the correlation coefficients.

```
[rho,pval] = partialcorr(x,z)
```

```
rho =
```

```
    1.0000    0.1300    0.0462
    0.1300    1.0000    0.0012
    0.0462    0.0012    1.0000
```

```
pval =
```

```
    0    0.2044    0.6532
    0.2044    0    0.9903
    0.6532    0.9903    0
```

The large values in `pval` indicate that there is no significant correlation between age and either blood pressure measurement after controlling for gender, smoking status, and weight.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames',{'Age','BPTop','BPBottom'},...
    'RowNames',{'SexID','Smoker','Weight'});
```

```
pval = array2table(pval, ...
```

```
'VariableNames',{ 'Age', 'BPTop', 'BPBottom'}, ...  
'RowNames',{ 'SexID', 'Smoker', 'Weight'});
```

```
disp('Partial Correlation Coefficients')  
disp(rho)  
disp('p-values')  
disp(pval)
```

```
Partial Correlation Coefficients
```

	Age	BPTop	BPBottom
SexID	1	0.13	0.046202
Smoker	0.13	1	0.0012475
Weight	0.046202	0.0012475	1

```
p-values
```

	Age	BPTop	BPBottom
SexID	0	0.20438	0.65316
Smoker	0.20438	0	0.99032
Weight	0.65316	0.99032	0

## Test for Paired Partial Correlation Coefficients

Test for partial correlation between pairs of variables in the x and y input matrices, while controlling for the effects of a third set of variables.

Load the sample data. Convert the strings in `hospital.Sex` to numeric group identifiers.

```
load hospital;  
hospital.SexID = grp2idx(hospital.Sex);
```

Create three matrices containing the sample data.

```
x = [hospital.BloodPressure];  
y = [hospital.Weight hospital.Age];  
z = [hospital.SexID hospital.Smoker];
```

`partialcorr` can test for partial correlation between the pairs of variables in `x` (the systolic and diastolic blood pressure measurements) and `y` (weight and age), while controlling for the variables in `z` (gender and smoking status). The measurements for `BloodPressure` are contained in two columns: The first column contains the upper (systolic) number, and the second column contains the lower (diastolic) number. `partialcorr` treats each column as a separate variable.

Test for partial correlation between pairs of variables in `x` and `y`, while controlling for the effects of the variables in `z`. Compute the correlation coefficients.

```
[rho,pval] = partialcorr(x,y,z)
```

```
rho =
```

```
   -0.0257    0.1289  
    0.0292    0.0472
```

```
pval =
```

```
    0.8018    0.2058  
    0.7756    0.6442
```

The results in `pval` indicate that, after controlling for gender and smoking status, there is no significant correlation between either of a patient's blood pressure measurements and that patient's weight or age.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...  
    'VariableNames',{'BPTop','BPBottom'},...  
    'RowNames',{'Weight','Age'});
```

```
pval = array2table(pval, ...  
    'VariableNames',{'BPTop','BPBottom'},...
```

```
'RowNames',{'Weight','Age'});  
  
disp('Partial Correlation Coefficients')  
disp(rho)  
disp('p-values')  
disp(pval)
```

```
Partial Correlation Coefficients  
                BPTop      BPBottom  
-----  
Weight          -0.02568    0.12893  
Age             0.029168    0.047226
```

```
p-values  
                BPTop      BPBottom  
-----  
Weight          0.80182     0.2058  
Age             0.77556     0.64424
```

## One-Tailed Partial Correlation Test

Test the hypothesis that pairs of variables have no correlation, against the alternative hypothesis that the correlation is greater than 0.

Load the sample data. Convert the strings in `hospital.Sex` to numeric group identifiers.

```
load hospital;  
hospital.SexID = grp2idx(hospital.Sex);
```

Create three matrices containing the sample data.

```
x = [hospital.BloodPressure];  
y = [hospital.Weight hospital.Age];  
z = [hospital.SexID hospital.Smoker];
```

`partialcorr` can test for partial correlation between the pairs of variables in `x` (the systolic and diastolic blood pressure measurements) and `y` (weight and age), while controlling for the variables in `z` (gender

and smoking status). The measurements for BloodPressure are contained in two columns: The first column contains the upper (systolic) number, and the second column contains the lower (diastolic) number. `partialcorr` treats each column as a separate variable.

Compute the correlation coefficients using a right-tailed test.

```
[rho,pval] = partialcorr(x,y,z,'Tail','right')
```

```
rho =
```

```
   -0.0257    0.1289  
    0.0292    0.0472
```

```
pval =
```

```
   0.5991    0.1029  
   0.3878    0.3221
```

The results in `pval` indicate that `partialcorr` does not reject the null hypothesis of nonzero correlations between the variables in `x` and `y`, after controlling for the variables in `z`, when the alternative hypothesis is that the correlations are greater than 0.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...  
    'VariableNames',{'BPTop','BPBottom'},...  
    'RowNames',{'Weight','Age'});
```

```
pval = array2table(pval, ...  
    'VariableNames',{'BPTop','BPBottom'},...  
    'RowNames',{'Weight','Age'});
```

```
disp('Partial Correlation Coefficients')  
disp(rho)
```

# partialcorr

---

```
disp('p-values')  
disp(pval)
```

```
Partial Correlation Coefficients  
          BPSys      BPDia  
Weight  -0.02568    0.12893  
Age      0.029168   0.047226
```

```
p-values  
          BPSys      BPDia  
Weight  0.59909    0.1029  
Age     0.38778    0.32212
```

## See Also

`corr` | `tiedrank` | `corrcoef` | `partialcorri`



## Purpose

Partial correlation coefficients adjusted for internal variables

## Syntax

```
rho = partialcorri(y,x)
rho = partialcorri(y,x,z)
rho = partialcorri( ___,Name,Value)
[rho,pval] = partialcorri( ___ )
```

## Description

`rho = partialcorri(y,x)` returns the sample linear partial correlation coefficients between pairs of variables in `y` and `x`, adjusting for the remaining variables in `x`.

`rho = partialcorri(y,x,z)` returns the sample linear partial correlation coefficients between pairs of variables in `y` and `x`, adjusting for the remaining variables in `x`, after first controlling both `x` and `y` for the variables in `z`.

`rho = partialcorri( ___,Name,Value)` returns the sample linear partial correlation coefficients with additional options specified by one or more name-value pair arguments, using input arguments from any of the previous syntaxes. For example, you can specify whether to use Pearson or Spearman partial correlations, or specify how to treat missing values.

`[rho,pval] = partialcorri( ___ )` also returns a matrix `pval` of  $p$ -values for testing the hypothesis of no partial correlation against the one- or two-sided alternative that there is a nonzero partial correlation.

## Input Arguments

### **x** - Data matrix

matrix

Data matrix, specified as an  $n$ -by- $p_x$  matrix. The rows of `x` correspond to observations, and the columns correspond to variables.

### Data Types

single | double

## **y - Data matrix**

matrix

Data matrix, specified as an  $n$ -by- $p_y$  matrix. The rows of  $y$  correspond to observations, and the columns correspond to variables.

### **Data Types**

single | double

## **z - Data matrix**

matrix

Data matrix, specified as an  $n$ -by- $p_z$  matrix. The rows of  $z$  correspond to observations, and the columns correspond to variables.

### **Data Types**

single | double

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

**Example:** 'Type', 'Spearman', 'Rows', 'complete' computes Spearman partial correlations using only the data in rows that contain no missing values.

## **'Type' - Type of partial correlations**

'Pearson' (default) | 'Spearman'

Type of partial correlations to compute, specified as the comma-separated pair consisting of 'Type' and either 'Pearson' or 'Spearman'. Pearson computes the Pearson (linear) partial correlations. Spearman computes the Spearman (rank) partial correlations.

**Example:** 'Type', 'Spearman'

**'Rows' - Rows to use in computation**`'all' (default) | 'complete' | 'pairwise'`

Rows to use in computation, specified as the comma-separated pair consisting of 'Rows' and one of the following.

<code>'all'</code>	Use all rows regardless of missing (NaN) values.
<code>'complete'</code>	Use only rows with no missing values.
<code>'pairwise'</code>	Use all available values in each column of $y$ when computing the partial correlation coefficients and $p$ -values corresponding to that column. For each column of $y$ , rows will be dropped corresponding to missing values in $x$ (and/or $z$ , if supplied). However, remaining rows with valid values in that column of $y$ are used, even if there are missing values in other columns of $y$ .

**Example:** `'Rows', 'complete'`

**'Tail' - Alternative hypothesis**`'both' (default) | 'right' | 'left'`

Alternative hypothesis to test against, specified as the comma-separated pair consisting of 'Tail' and one of the following.

<code>'both'</code>	Test the alternative hypothesis that the correlation is not zero.
<code>'right'</code>	Test the alternative hypothesis that the correlation is greater than 0.
<code>'left'</code>	Test the alternative hypothesis that the correlation is less than 0.

**Example:** `'Tail', 'right'`

## Output Arguments

### **rho - Sample linear partial correlation coefficients**

matrix

Sample linear partial correlation coefficients, returned as a  $p_y$ -by- $p_x$  matrix.

- If you input  $x$  and  $y$  matrices, the  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th column in  $y$  and the  $j$ th column in  $x$ , controlled for all the columns of  $x$  except column  $j$ .
- If you input  $x$ ,  $y$ , and  $z$  matrices, the  $(i,j)$ th entry is the sample linear partial correlation between the  $i$ th column in  $y$  and the  $j$ th column in  $x$ , adjusted for all the columns of  $x$  except column  $j$ , after first controlling both  $x$  and  $y$  for the variables in  $z$ .

### **pval - p-values**

matrix

$p$ -values, returned as a matrix. Each element of `pval` is the  $p$ -value for the corresponding element of `rho`. If `pval(i, j)` is small, then the corresponding partial correlation `rho(i, j)` is statistically significantly different from zero.

`partialcorri` computes  $p$ -values for linear and rank partial correlations using a Student's  $t$  distribution for a transformation of the correlation. This is exact for linear partial correlation when  $x$  and  $z$  are normal, but is a large-sample approximation otherwise.

## Examples

### **Compute Partial Correlation Coefficients**

Compute partial correlation coefficients for each pair of variables in the  $x$  and  $y$  input matrices, while controlling for the effects of the remaining variables in  $x$ .

Load the sample data.

```
load carsmall;
```

The data contains measurements from cars manufactured in 1970, 1976, and 1982. It includes MPG and Acceleration as performance measures, and Displacement, Horsepower, and Weight as design

variables. Acceleration is the time required to accelerate from 0 to 60 miles per hour, so a high value for Acceleration corresponds to a vehicle with low acceleration.

Define the input matrices. The y matrix includes the performance measures, and the x matrix includes the design variables.

```
y = [MPG,Acceleration];
x = [Displacement,Horsepower,Weight];
```

Compute the correlation coefficients. Include only rows with no missing values in the computation.

```
rho = partialcorri(y,x,'Rows','complete')
```

```
rho =
```

```
    -0.0537    -0.1520    -0.4856
    -0.3994    -0.4008     0.4912
```

The results suggest, for example, a 0.4912 correlation between weight and acceleration after controlling for the effects of displacement and horsepower. You can return the *p*-values as a second output, and examine them to confirm whether these correlations are statistically significant.

For a clearer display, create a table with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames',{'Displacement','Horsepower','Weight'}, ...
    'RowNames',{'MPG','Acceleration'});
```

```
disp('Partial Correlation Coefficients')
disp(rho)
```

```
Partial Correlation Coefficients
                Displacement    Horsepower    Weight
                -----
```

MPG	-0.053684	-0.15199	-0.48563
Acceleration	-0.39941	-0.40075	0.49123

## Test Partial Correlations While Controlling for Additional Variables

Test for partial correlation between pairs of variables in the x and y input matrices, while controlling for the effects of the remaining variables in x plus additional variables in matrix z.

Load the sample data.

```
load carsmall;
```

The data contains measurements from cars manufactured in 1970, 1976, and 1982. It includes MPG and Acceleration as performance measures, and Displacement, Horsepower, and Weight as design variables. Acceleration is the time required to accelerate from 0 to 60 miles per hour, so a high value for Acceleration corresponds to a vehicle with low acceleration.

Create a new variable Headwind, and randomly generate data to represent the notion of an average headwind along the performance measurement route.

```
rng('default'); % For reproducibility  
Headwind = (10:-0.2:-9.8)' + 5*randn(100,1);
```

Since headwind can affect the performance measures, control for its effects when testing for partial correlation between the remaining variables.

Define the input matrices. The y matrix includes the performance measures, and the x matrix includes the design variables. The z matrix contains additional variables to control for when computing the partial correlations, such as headwind.

```
y = [MPG,Acceleration];  
x = [Displacement,Horsepower,Weight];
```

```
z = Headwind;
```

Compute the partial correlation coefficients. Include only rows with no missing values in the computation.

```
[rho,pval] = partialcorri(y,x,z,'Rows','complete')
```

```
rho =
```

```
    0.0572    -0.1055    -0.5736
   -0.3845   -0.3966     0.4674
```

```
pval =
```

```
    0.5923    0.3221    0.0000
    0.0002    0.0001    0.0000
```

The small returned *p*-value of 0.001 in *pval* indicates, for example, a significant negative correlation between horsepower and acceleration, after controlling for displacement, weight, and headwind.

For a clearer display, create tables with appropriate variable and row labels.

```
rho = array2table(rho, ...
    'VariableNames',{'Displacement','Horsepower','Weight'}, ...
    'RowNames',{'MPG','Acceleration'});
```

```
pval = array2table(pval, ...
    'VariableNames',{'Displacement','Horsepower','Weight'}, ...
    'RowNames',{'MPG','Acceleration'});
```

```
disp('Partial Correlation Coefficients, Accounting for Headwind')
disp(rho)
disp('p-values, Accounting for Headwind')
disp(pval)
```

```
Partial Correlation Coefficients, Accounting for Headwind
```

# partialcorri

---

	Displacement	Horsepower	Weight
	-----	-----	-----
MPG	0.057197	-0.10555	-0.57358
Acceleration	-0.38452	-0.39658	0.4674
P-values, Accounting for Headwind			
	Displacement	Horsepower	Weight
	-----	-----	-----
MPG	0.59233	0.32212	3.4401e-09
Acceleration	0.00018272	0.00010902	3.4091e-06

## See Also

`partialcorr` | `corr`



**Purpose**

Principal component analysis of raw data

**Syntax**

```
coeff = pca(X)
coeff = pca(X,Name,Value)
[coeff,score,latent] = pca( ___ )
[coeff,score,latent,tsquared] = pca( ___ )
[coeff,score,latent,tsquared,explained,mu] = pca( ___ )
```

**Description**

`coeff = pca(X)` returns the principal component coefficients for the  $n$ -by- $p$  data matrix  $X$ . Rows of  $X$  correspond to observations and columns correspond to variables. The coefficient matrix is  $p$ -by- $p$ . Each column of `coeff` contains coefficients for one principal component, and the columns are in descending order of component variance. By default, `pca` centers the data and uses the singular value decomposition (SVD) algorithm.

`coeff = pca(X,Name,Value)` returns any of the output arguments in the previous syntaxes using additional options for computation and handling of special data types, specified by one or more `Name,Value` pair arguments.

For example, you can specify the number of principal components `pca` returns or an algorithm other than SVD to use.

`[coeff,score,latent] = pca( ___ )` also returns the principal component scores in `score` and the principal component variances in `latent`. You can use any of the input arguments in the previous syntaxes.

Principal component scores are the representations of  $X$  in the principal component space. Rows of `score` correspond to observations, and columns correspond to components.

The principal component variances are the eigenvalues of the covariance matrix of  $X$ .

`[coeff,score,latent,tsquared] = pca( ___ )` also returns the Hotelling's T-squared statistic for each observation in  $X$ .

`[coeff,score,latent,tsquared,explained,mu] = pca( ___ )` also returns `explained`, the percentage of the total variance explained by each principal component and `mu`, the estimated mean of each variable in  $X$ .

## Input Arguments

### **X - Input data**

matrix

Input data for which to compute the principal components, specified as an  $n$ -by- $p$  matrix. Rows of  $X$  correspond to observations and columns to variables.

### **Data Types**

single | double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

### **Example:**

`'Algorithm','eig','Centered',false,'Rows','all','NumComponents',3` specifies that `pca` uses eigenvalue decomposition algorithm, not center the data, use all of the observations, and return only the first three principal components.

### **'Algorithm' - Principal component algorithm**

`'svd' (default) | 'eig' | 'als'`

Principal component algorithm that `pca` uses to perform the principal component analysis, specified as the comma-separated pair consisting of `'Algorithm'` and one of the following.

'svd'	Default. Singular value decomposition (SVD) of $X$ .
'eig'	Eigenvalue decomposition (EIG) of the covariance matrix. The EIG algorithm is faster than SVD when the number of observations, $n$ , exceeds the number of variables, $p$ , but is less accurate because the condition number of the covariance is the square of the condition number of $X$ .
'als'	Alternating least squares (ALS) algorithm. This algorithm finds the best rank- $k$ approximation by factoring $X$ into a $n$ -by- $k$ left factor matrix, $L$ , and a $p$ -by- $k$ right factor matrix, $R$ , where $k$ is the number of principal components. The factorization uses an iterative method starting with random initial values. ALS is designed to better handle missing values. It is preferable to pairwise deletion ('Rows', 'pairwise') and deals with missing values without listwise deletion ('Rows', 'complete'). It can work well for data sets with a small percentage of missing data at random, but might not perform well on sparse data sets.

Example: 'Algorithm', 'eig'

### Data Types

char

### 'Centered' - Indicator for centering columns

true (default) | false

Indicator for centering the columns, specified as the comma-separated pair consisting of 'Centered' and one of these logical expressions.

true	Default. <code>pca</code> centers $X$ by subtracting column means before computing singular value decomposition or eigenvalue decomposition. If $X$ contains NaN missing values, <code>nanmean</code> is used to find the mean with any available data. You can reconstruct the centered data using <code>score*coeff</code> .
false	In this case <code>pca</code> does not center the data. You can reconstruct the original data using <code>score*coeff</code> .

**Example:** 'Centered', false

### Data Types

logical

### 'Economy' - Indicator for economy size output

true (default) | false

Indicator for the economy size output when the degrees of freedom,  $d$ , is smaller than the number of variables,  $p$ , specified as the comma-separated pair consisting of 'Economy' and one of these logical expressions.

true	Default. <code>pca</code> returns only the first $d$ elements of latent and the corresponding columns of <code>coeff</code> and <code>score</code> .  This option can be significantly faster when the number of variables $p$ is much larger than $d$ .
false	<code>pca</code> returns all elements of latent. The columns of <code>coeff</code> and <code>score</code> corresponding to zero elements in latent are zeros.

Note that when  $d < p$ , `score(:, d+1:p)` and `latent(d+1:p)` are necessarily zero, and the columns of `coeff(:, d+1:p)` define directions that are orthogonal to  $X$ .

**Example:** 'Economy', false

### Data Types

logical

### 'NumComponents' - Number of components requested

number of variables (default) | scalar integer

Number of components requested, specified as the comma-separated pair consisting of 'NumComponents' and a scalar integer  $k$  satisfying  $0 < k \leq p$ , where  $p$  is the number of original variables in  $X$ . When specified, `pca` returns the first  $k$  columns of `coeff` and `score`.

**Example:** 'NumComponents', 3

**Data Types**

single | double

**'Rows' - Action to take for NaN values**

'complete' (default) | 'pairwise' | 'all'

Action to take for NaN values in the data matrix  $X$ , specified as the comma-separated pair consisting of 'Rows' and one of the following.

'complete'	Default. Observations with NaN values are removed before calculation. Rows of NaNs are reinserted into score and tsquared at the corresponding locations.
'pairwise'	<p>This option only applies when the algorithm is 'eig'. If you don't specify the algorithm along with 'pairwise', then <code>pca</code> sets it to 'eig'. If you specify 'svd' as the algorithm, along with the option 'Rows', 'pairwise', then <code>pca</code> returns a warning message, sets the algorithm to 'eig' and continues.</p> <p>When you specify the 'Rows', 'pairwise' option, <code>pca</code> computes the <math>(i,j)</math> element of the covariance matrix using the rows with no NaN values in the columns <math>i</math> or <math>j</math> of <math>X</math>.</p> <p>Note that the resulting covariance matrix might not be positive definite. In that case, <code>pca</code> terminates with an error message.</p>
'all'	$X$ is expected to have no missing values. <code>pca</code> uses all of the data and terminates if any NaN value is found.

**Example:** 'Rows', 'pairwise'

**Data Types**

char

**'Weights' - Observation weights**

ones (default) | row vector

Observation weights, specified as the comma-separated pair consisting of 'Weights' and a vector of length  $n$  containing all positive elements.

**Data Types**

single | double

**'VariableWeights' - Variable weights**

row vector | 'variance'

Variable weights, specified as the comma-separated pair consisting of 'VariableWeights' and one of the following.

---

Vector of length  $p$  containing all positive elements.

The string 'variance'. The variable weights are the inverse of sample variance. If you also assign weights to observations using 'Weights', then the variable weights become the inverse of weighted sample variance.

If 'Centered' is set to true at the same time, the data matrix  $X$  is centered and standardized. In this case, pca returns the principal components based on the correlation matrix.

---

**Example:** 'VariableWeights', 'variance'

**Data Types**

single | double | char

**'Coeff0' - Initial value for coefficients**

matrix of random values (default) |  $p$ -by- $k$  matrix

Initial value for the coefficient matrix `coeff`, specified as the comma-separated pair consisting of 'Coeff0' and a  $p$ -by- $k$  matrix, where  $p$  is the number of variables, and  $k$  is the number of principal components requested.

---

**Note** You can use this name-value pair only when 'algorithm' is 'als'.

---

**Data Types**

single | double

**'Score0' - Initial value for scores**matrix of random values (default) |  $k$ -by- $m$  matrix

Initial value for scores matrix `score`, specified as a comma-separated pair consisting of 'Score0' and an  $n$ -by- $k$  matrix, where  $n$  is the number of observations and  $k$  is the number of principal components requested.

---

**Note** You can use this name-value pair only when 'algorithm' is 'als'.

---

**Data Types**

single | double

**'Options' - Options for iterations**

structure

Options for the iterations, specified as a comma-separated pair consisting of 'Options' and a structure created by the `statset` function. `pca` uses the following fields in the options structure.

'Display'	Level of display output. Choices are 'off', 'final', and 'iter'.
'MaxIter'	Maximum number steps allowed. The default is 100. Unlike in optimization settings, reaching the MaxIter value is regarded as convergence.
'TolFun'	Positive number giving the termination tolerance for the cost function. The default is 1e-6.
'TolX'	Positive number giving the convergence threshold for the relative change in the elements of the left and right factor matrices, L and R, in the ALS algorithm. The default is 1e-6.

---

**Note** You can use this name-value pair only when 'algorithm' is 'als'.

---

You can change the values of these fields and specify the new structure in `pca` using the 'Options' name-value pair argument.

**Example:** `opt = statset('pca'); opt.MaxIter = 2000; coeff = pca(X, 'Options', opt);`

### Data Types

struct

## Output Arguments

### **coeff** - Principal component coefficients

matrix

Principal component coefficients, returned as a  $p$ -by- $p$  matrix. Each column of `coeff` contains coefficients for one principal component. The columns are in the order of descending component variance, `latent`.

### **score** - Principal component scores

matrix

Principal component scores, returned as a matrix. Rows of `score` correspond to observations, and columns to components.

### **latent** - Principal component variances

column vector

Principal component variances, that is the eigenvalues of the covariance matrix of `X`, returned as a column vector.

### **tsquared** - Hotelling's T-squared statistic

column vector

"Hotelling's T-Squared Statistic" on page 21-2406, which is the sum of squares of the standardized scores for each observation, returned as a column vector.



**explained - Percentage of total variance explained**

column vector

Percentage of the total variance explained by each principal component, returned as a column vector.

**mu - Estimated means**

row vector

Estimated means of the variables in X, returned as a row vector. When 'algorithm' is 'als', this is estimated by ALS algorithm. When 'algorithm' is not 'als', mu is equal to the sample mean of X.

**Examples****Principal Components of a Data Set**

Load the sample data set.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Find the principal components for the ingredients data.

```
coeff = pca(ingredients)
```

```
coeff =
```

```

-0.0678   -0.6460    0.5673    0.5062
-0.6785   -0.0200   -0.5440    0.4933
 0.0290    0.7553    0.4036    0.5156
 0.7309   -0.1085   -0.4684    0.4844
```

The rows of `coeff` contain the coefficients for the four ingredient variables, and its columns correspond to four principal components.

**PCA in the Presence of Missing Data**

Find the principal component coefficients when there are missing values in a data set.

Load the sample data set.

```
load imports-85
```

Data matrix  $X$  has 13 continuous variables in columns 3 to 15: wheel-base, length, width, height, curb-weight, engine-size, bore, stroke, compression-ratio, horsepower, peak-rpm, city-mpg, and highway-mpg. The variables bore and stroke are missing four values in rows 56 to 59, and the variables horsepower and peak-rpm are missing two values in rows 131 and 132.

Perform principal component analysis.

```
coeff = pca(X(:,3:15));
```

By default, `pca` performs the action specified by the 'Rows', 'complete' name-value pair argument. This option removes the observations with NaN values before calculation. Rows of NaNs are reinserted into `score` and `tsquared` at the corresponding locations, namely rows 56 to 59, 131, and 132.

Use 'pairwise' to perform the principal component analysis.

```
coeff = pca(X(:,3:15), 'Rows', 'pairwise');
```

In this case, `pca` computes the  $(i,j)$  element of the covariance matrix using the rows with no NaN values in the columns  $i$  or  $j$  of  $X$ . Note that the resulting covariance matrix might not be positive definite. This option applies when the algorithm `pca` uses is eigenvalue decomposition. When you don't specify the algorithm, as in this example, `pca` sets it to 'eig'. If you require 'svd' as the algorithm, with the 'pairwise' option, then `pca` returns a warning message, sets the algorithm to 'eig' and continues.

If you use the 'Rows', 'all' name-value pair argument, `pca` terminates because this option assumes there are no missing values in the data set.

```
coeff = pca(X(:,3:15), 'Rows', 'all');
```

Error using pca (line 180)

Raw data contains NaN missing value while 'Rows' option is set to 'all'

## Weighted PCA

Use the inverse variable variances as weights while performing the principal components analysis.

Load the sample data set.

```
load hald
```

Perform the principal component analysis using the inverse of variances of the ingredients as variable weights.

```
[wcoeff,~,latent,~,explained] = pca(ingredients,...  
'VariableWeights','variance')
```

```
wcoeff =
```

-2.7998	2.9940	-3.9736	1.4180
-8.7743	-6.4411	4.8927	9.9863
2.5240	-3.8749	-4.0845	1.7196
9.1714	7.5529	3.2710	11.3273

```
latent =
```

2.2357
1.5761
0.1866
0.0016

```
explained =
```

55.8926
39.4017
4.6652
0.0406

Note that the coefficient matrix, `wcoeff`, is not orthonormal.

Calculate the orthonormal coefficient matrix.

```
coefforth = inv(diag(std(ingredients))) * wcoeff
```

```
coefforth =
```

```
-0.4760    0.5090   -0.6755    0.2411  
-0.5639   -0.4139    0.3144    0.6418  
 0.3941   -0.6050   -0.6377    0.2685  
 0.5479    0.4512    0.1954    0.6767
```

Check orthonormality of the new coefficient matrix, `coefforth`.

```
coefforth*coefforth'
```

```
ans =
```

```
 1.0000    0.0000   -0.0000   -0.0000  
 0.0000    1.0000   -0.0000   -0.0000  
-0.0000   -0.0000    1.0000         0  
-0.0000   -0.0000         0    1.0000
```

### PCA Using ALS for Missing Data

Find the principal components using the alternating least squares (ALS) algorithm when there are missing values in the data.

Load the sample data.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Perform principal component analysis using the ALS algorithm and display the component coefficients.

```
[coeff,score,latent,tsquared,explained] = pca(ingredients);  
coeff
```

```
coeff =
    -0.0678    -0.6460     0.5673     0.5062
    -0.6785    -0.0200    -0.5440     0.4933
     0.0290     0.7553     0.4036     0.5156
     0.7309    -0.1085    -0.4684     0.4844
```

Introduce missing values randomly.

```
y = ingredients;
rng('default'); % for reproducibility
ix = random('unif',0,1,size(y))<0.30;
y(ix) = NaN
```

```
y =
     7     26     6  NaN
     1     29    15   52
    NaN  NaN     8   20
    11    31  NaN   47
     7    52     6   33
    NaN   55  NaN  NaN
    NaN   71  NaN    6
     1    31  NaN   44
     2  NaN  NaN   22
    21   47     4   26
    NaN   40   23   34
    11   66     9  NaN
    10   68     8   12
```

Approximately 30% of the data has missing values now, indicated by NaN.

Perform principal component analysis using the ALS algorithm and display the component coefficients.

```
[coeff1,score1,latent,tsquared,explained,mu1] = pca(y,...
'algorithm','als');
```

coeff1

coeff1 =

-0.0362	0.8215	-0.5252	0.2190
-0.6831	-0.0998	0.1828	0.6999
0.0169	0.5575	0.8215	-0.1185
0.7292	-0.0657	0.1261	0.6694

Display the estimated mean.

mu1

mu1 =

8.9956	47.9088	9.0451	28.5515
--------	---------	--------	---------

Reconstruct the observed data.

t = score1\*coeff1' + repmat(mu1,13,1)

t =

7.0000	26.0000	6.0000	51.5250
1.0000	29.0000	15.0000	52.0000
10.7819	53.0230	8.0000	20.0000
11.0000	31.0000	13.5500	47.0000
7.0000	52.0000	6.0000	33.0000
10.4818	55.0000	7.8328	17.9362
3.0982	71.0000	11.9491	6.0000
1.0000	31.0000	-0.5161	44.0000
2.0000	53.7914	5.7710	22.0000
21.0000	47.0000	4.0000	26.0000
21.5809	40.0000	23.0000	34.0000
11.0000	66.0000	9.0000	5.7078
10.0000	68.0000	8.0000	12.0000

The ALS algorithm estimates the missing values in the data.

Another way to compare the results is to find the angle between the two spaces spanned by the coefficient vectors. Find the angle between the coefficients found for complete data and data with missing values using ALS.

```
subspace(coeff,coeff1)
```

```
ans =
```

```
2.2925e-16
```

This is a small value. It indicates that the results if you use `pca` with `'Rows','complete'` name-value pair argument when there is no missing data and if you use `pca` with `'algorithm','als'` name-value pair argument when there is missing data are close to each other.

Perform the principal component analysis using `'Rows','complete'` name-value pair argument and display the component coefficients.

```
[coeff2,score2,latent,tsquared,explained,mu2] = pca(y,...
'Rows','complete');
coeff2
```

```
coeff2 =
```

```
-0.2054    0.8587    0.0492
-0.6694   -0.3720    0.5510
 0.1474   -0.3513   -0.5187
 0.6986   -0.0298    0.6518
```

In this case, `pca` removes the rows with missing values, and `y` has only four rows with no missing values. `pca` returns only three principal components. You cannot use the `'Rows','pairwise'` option because the covariance matrix is not positive semidefinite and `pca` returns an error message.

Find the angle between the coefficients found for complete data and data with missing values using listwise deletion (when `'Rows','complete'`).

```
subspace(coeff(:,1:3),coeff2)
```

```
ans =
```

```
0.3576
```

The angle between the two spaces is substantially larger. This indicates that these two results are differ.

Display the estimated mean.

```
mu2
```

```
mu2 =
```

```
7.8889 46.9091 9.8750 29.6000
```

In this case, the mean is just the sample mean of y.

Reconstruct the observed data.

```
score2*coeff2'
```

```
ans =
```

```
NaN NaN NaN NaN
-7.5162 -18.3545 4.0968 22.0056
NaN NaN NaN NaN
NaN NaN NaN NaN
-0.5644 5.3213 -3.3432 3.6040
NaN NaN NaN NaN
NaN NaN NaN NaN
NaN NaN NaN NaN
NaN NaN NaN NaN
12.8315 -0.1076 -6.3333 -3.7758
NaN NaN NaN NaN
NaN NaN NaN NaN
1.4680 20.6342 -2.9292 -18.0043
```



This shows that deleting rows containing NaN values does not work as well as the ALS algorithm. Using ALS is better when the data has too many missing values.

### Principal Component Coefficients, Scores, and Variances

Find the coefficients, scores, and variances of the principal components.

Load the sample data set.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Find the principal component coefficients, scores, and variances of the components for the ingredients data.

```
[coeff,score,latent] = pca(ingredients)
```

```
coeff =
```

-0.0678	-0.6460	0.5673	0.5062
-0.6785	-0.0200	-0.5440	0.4933
0.0290	0.7553	0.4036	0.5156
0.7309	-0.1085	-0.4684	0.4844

```
score =
```

36.8218	-6.8709	-4.5909	0.3967
29.6073	4.6109	-2.2476	-0.3958
-12.9818	-4.2049	0.9022	-1.1261
23.7147	-6.6341	1.8547	-0.3786
-0.5532	-4.4617	-6.0874	0.1424
-10.8125	-3.6466	0.9130	-0.1350
-32.5882	8.9798	-1.6063	0.0818
22.6064	10.7259	3.2365	0.3243
-9.2626	8.9854	-0.0169	-0.5437
-3.2840	-14.1573	7.0465	0.3405

```
    9.2200    12.3861    3.4283    0.4352
   -25.5849   -2.7817   -0.3867    0.4468
   -26.9032   -2.9310   -2.4455    0.4116
```

latent =

```
517.7969
 67.4964
 12.4054
  0.2372
```

Each column of `score` corresponds to one principal component. The vector, `latent`, stores the variances of the four principal components.

Reconstruct the centered ingredients data.

```
Xcentered = score*coeff'
```

Xcentered =

```
-0.4615  -22.1538  -5.7692  30.0000
-6.4615  -19.1538   3.2308  22.0000
 3.5385   7.8462  -3.7692 -10.0000
 3.5385  -17.1538  -3.7692  17.0000
-0.4615   3.8462  -5.7692   3.0000
 3.5385   6.8462  -2.7692  -8.0000
-4.4615  22.8462   5.2308 -24.0000
-6.4615  -17.1538  10.2308  14.0000
-5.4615   5.8462   6.2308  -8.0000
13.5385  -1.1538  -7.7692  -4.0000
-6.4615  -8.1538  11.2308   4.0000
 3.5385  17.8462  -2.7692 -18.0000
 2.5385  19.8462  -3.7692 -18.0000
```

The new data in `Xcentered` is the original ingredients data centered by subtracting the column means from corresponding columns.

## T-Squared Statistic

Find the Hotelling's T-squared statistic values.

Load the sample data set.

```
load hald
```

The ingredients data has 13 observations for 4 variables.

Perform the principal component analysis and request the T-squared values.

```
[coeff,score,latent,tsquared] = pca(ingredients);  
tsquared
```

```
tsquared =
```

```
    5.6803  
    3.0758  
    6.0002  
    2.6198  
    3.3681  
    0.5668  
    3.4818  
    3.9794  
    2.6086  
    7.4818  
    4.1830  
    2.2327  
    2.7216
```

Request only the first two principal components and compute the T-squared values in the reduced space of requested principal components.

```
[coeff,score,latent,tsquared] = pca(ingredients,'NumComponents',2);  
tsquared
```

```
tsquared =  
  
    5.6803  
    3.0758  
    6.0002  
    2.6198  
    3.3681  
    0.5668  
    3.4818  
    3.9794  
    2.6086  
    7.4818  
    4.1830  
    2.2327  
    2.7216
```

Note that even when you specify a reduced component space, `pca` computes the T-squared values in the full space, using all four components.

The T-squared value in the reduced space corresponds to the Mahalanobis distance in the reduced space.

```
tsqreduced = mahal(score, score)
```

```
tsqreduced =  
  
    3.3179  
    2.0079  
    0.5874  
    1.7382  
    0.2955  
    0.4228  
    3.2457  
    2.6914  
    1.3619  
    2.9903
```

```

2.4371
1.3788
1.5251

```

Calculate the T-squared values in the discarded space by taking the difference of the T-squared values in the full space and Mahalanobis distance in the reduced space.

```
tsqdiscarded = tsquared - tsqreduced
```

```
tsqdiscarded =
```

```

2.3624
1.0679
5.4128
0.8816
3.0726
0.1440
0.2362
1.2880
1.2467
4.4915
1.7459
0.8539
1.1965

```

### **Percent Variability Explained by Principal Components**

Find the percent variability explained by the principal components.

Load the sample data set.

```
load imports-85
```

Data matrix  $X$  has 13 continuous variables in columns 3 to 15: wheel-base, length, width, height, curb-weight, engine-size, bore, stroke, compression-ratio, horsepower, peak-rpm, city-mpg, and highway-mpg.

Find the percent variability explained by principal components of these variables.

```
[coeff,score,latent,tsquared,explained] = pca(X(:,3:15));
```

```
explained
```

```
explained =
```

```
64.3429  
35.4484  
0.1550  
0.0379  
0.0078  
0.0048  
0.0013  
0.0011  
0.0005  
0.0002  
0.0002  
0.0000  
0.0000
```

The first two components explain 99.79% of all variability.

To skip any of the outputs, you can use ~ instead in the corresponding element. For example, if you don't want to get the T-squared values, specify

```
[coeff,score,latent,~,explained] = pca(X(:,3:15));
```

## **Definitions**

### **Hotelling's T-Squared Statistic**

Hotelling's T-squared statistic is a statistical measure of the multivariate distance of each observation from the center of the data set.

Even when you request fewer components than the number of variables, `pca` uses all principal components to compute the T-squared statistic

(computes it in the full space). If you want the T-squared statistic in the reduced or the discarded space, do one of the following:

- For the T-squared statistic in the reduced space, use `mahal(score, score)`.
- For the T-squared statistic in the discarded space, first compute the T-squared statistic using `[coeff, score, latent, tsquared] = pca(X, 'NumComponents', k, ...)`, compute the T-squared statistic in the reduced space using `tsqreduced = mahal(score, score)`, and then take the difference: `tsquared - tsqreduced`.

### Degrees of Freedom

The degrees of freedom,  $d$ , is equal to  $n - 1$ , if data is centered and  $n$  otherwise, where:

- $n$  is the number of rows without any NaNs if you use `'Rows', 'complete'`.
- $n$  is the number of rows without any NaNs in the column pair that has the maximum number of rows without NaNs if you use `'Rows', 'pairwise'`.

### Variable Weights

Note that when variable weights are used, the coefficient matrix is not orthonormal. Suppose the variable weights vector you used is called `varwei`, and the principal component coefficients vector `pca` returned is `wcoeff`. You can then calculate the orthonormal coefficients using the transformation `diag(sqrt(varwei))*wcoeff`.

## References

- [1] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., Springer, 2002.
- [2] Krzanowski, W. J. *Principles of Multivariate Analysis*. Oxford University Press, 1988.

[3] Seber, G. A. F. *Multivariate Observations*. Wiley, 1984.

[4] Jackson, J. E. A. *User's Guide to Principal Components*. Wiley, 1988.

[5] Roweis, S. "EM Algorithms for PCA and SPCA." *In Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems*. Vol.10 (NIPS 1997), Cambridge, MA, USA: MIT Press, 1998, pp. 626–632.

[6] Ilin, A., and T. Raiko. "Practical Approaches to Principal Component Analysis in the Presence of Missing Values." *J. Mach. Learn. Res.*. Vol. 11, August 2010, pp. 1957–2000.

### See Also

`barttest` | `biplot` | `canoncorr` | `factoran` | `pcacov` | `pcares`  
| `rotatefactors` | `ppca`

### Related Examples

- "Quality of Life in U.S. Cities" on page 13-81

### Concepts

- "Principal Component Analysis (PCA)" on page 13-80



<b>Purpose</b>	Principal component analysis on covariance matrix
<b>Syntax</b>	<pre>COEFF = pcacov(V) [COEFF,latent] = pcacov(V) [COEFF,latent,explained] = pcacov(V)</pre>
<b>Description</b>	<p>COEFF = pcacov(V) performs principal components analysis on the p-by-p covariance matrix V and returns the principal component coefficients, also known as loadings. COEFF is a p-by-p matrix, with each column containing coefficients for one principal component. The columns are in order of decreasing component variance.</p> <p>pcacov does not standardize V to have unit variances. To perform principal components analysis on standardized variables, use the correlation matrix <math>R = V ./ (SD * SD')</math>, where <math>SD = \text{sqrt}(\text{diag}(V))</math>, in place of V. To perform principal components analysis directly on the data matrix, use <code>pca</code>.</p> <p>[COEFF,latent] = pcacov(V) returns latent, a vector containing the principal component variances, that is, the eigenvalues of V.</p> <p>[COEFF,latent,explained] = pcacov(V) returns explained, a vector containing the percentage of the total variance explained by each principal component.</p>
<b>Examples</b>	<pre>load hald covx = cov(ingredients); [COEFF,latent,explained] = pcacov(covx) COEFF =     0.0678 -0.6460  0.5673 -0.5062     0.6785 -0.0200 -0.5440 -0.4933    -0.0290  0.7553  0.4036 -0.5156    -0.7309 -0.1085 -0.4684 -0.4844  latent =     517.7969      67.4964      12.4054</pre>

0.2372

explained =  
86.5974  
11.2882  
2.0747  
0.0397

## References

- [1] Jackson, J. E. *A User's Guide to Principal Components*. Hoboken, NJ: John Wiley and Sons, 1991.
- [2] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., New York: Springer-Verlag, 2002.
- [3] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

## See Also

barttest | biplot | factoran | pcares | pca | rotatefactors

**Purpose**

Residuals from principal component analysis

**Syntax**

```
residuals = pcares(X,ndim)
[residuals,reconstructed] = pcares(X,ndim)
```

**Description**

`residuals = pcares(X,ndim)` returns the residuals obtained by retaining `ndim` principal components of the `n`-by-`p` matrix `X`. Rows of `X` correspond to observations, columns to variables. `ndim` is a scalar and must be less than or equal to `p`. `residuals` is a matrix of the same size as `X`. Use the data matrix, *not* the covariance matrix, with this function.

`pcares` does not normalize the columns of `X`. To perform the principal components analysis based on standardized variables, that is, based on correlations, use `pcares(zscore(X), ndim)`. You can perform principal components analysis directly on a covariance or correlation matrix, but without constructing residuals, by using `pcacov`.

`[residuals,reconstructed] = pcares(X,ndim)` returns the reconstructed observations; that is, the approximation to `X` obtained by retaining its first `ndim` principal components.

**Examples**

This example shows the drop in the residuals from the first row of the Hald data as the number of component dimensions increases from one to three.

```
load hald
r1 = pcares(ingredients,1);
r2 = pcares(ingredients,2);
r3 = pcares(ingredients,3);

r11 = r1(1,:)
r11 =
    2.0350    2.8304   -6.8378    3.0879

r21 = r2(1,:)
r21 =
   -2.4037    2.6930   -1.6482    2.3425
```

```
r31 = r3(1,:)
r31 =
    0.2008    0.1957    0.2045    0.1921
```

## References

- [1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, 1991.
- [2] Jolliffe, I. T., *Principal Component Analysis*, 2nd Edition, Springer, 2002.
- [3] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [4] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

factoran | pcacov | pca

**Purpose**

Probabilistic principal component analysis

**Syntax**

```
[coeff,score,pcvar] = ppca(Y,K)
[coeff,score,pcvar] = ppca(Y,K,Name,Value)
[coeff,score,pcvar,mu] = ppca(____)
[coeff,score,pcvar,mu,v,S] = ppca(____)
```

**Description**

`[coeff,score,pcvar] = ppca(Y,K)` returns the principal component coefficients for the  $n$ -by- $p$  data matrix  $Y$  based on a probabilistic principal component analysis (PPCA). It also returns the principal component scores, which are the representations of  $Y$  in the principal component space, and the principal component variances, which are the eigenvalues of the covariance matrix of  $Y$ , in `pcvar`.

Each column of `coeff` contains coefficients for one principal component, and the columns are in descending order of component variance. Rows of `score` correspond to observations, and columns correspond to components. Rows of  $Y$  correspond to observations and columns correspond to variables.

Probabilistic principle component analysis might be preferable to other algorithms that handle missing data, such as the alternating least squares algorithm when any data vector has one or more missing values. It assumes that the values are missing at random through the data set. An expectation-maximization algorithm is used for both complete and missing data.

`[coeff,score,pcvar] = ppca(Y,K,Name,Value)` returns the principal component coefficients, scores, and variances using additional options for computation and handling of special data types, specified by one or more `Name,Value` pair arguments.

For example, you can introduce initial values for the residual variance, `v`, or change the termination criteria.

[coeff, score, pcvar, mu] = ppca( \_\_\_ ) also returns the estimated mean of each variable in  $Y$ . You can use any of the input arguments in the previous syntaxes.

[coeff, score, pcvar, mu, v, S] = ppca( \_\_\_ ) also returns the isotropic residual variance in  $v$  and the final results at convergence in structure  $S$ .

## Input Arguments

### **Y - Input data**

*n*-by-*p* matrix

Input data for which to compute the principal components, specified as an *n*-by-*p* matrix. Rows of  $Y$  correspond to observations and columns correspond to variables.

### **Data Types**

single | double

### **K - Number of principal components**

positive integer value less than rank

Number of principal components to return, specified as an integer value less than the rank of data. The maximum possible rank is  $\min(n,p)$ , where  $n$  is the number of observations and  $p$  is the number of variables. However, if the data is correlated, the rank might be smaller than  $\min(n,p)$ .

ppca orders the components based on their variance.

If  $K$  is  $\min(n,p)$ , ppca sets  $K$  equal to  $\min(n,p) - 1$ , and 'W0' is truncated to  $\min(p,n) - 1$  columns if you specify a *p*-by-*p*  $W0$  matrix.

For example, you can request only the first three components, based on the component variance as follows.

**Example:** coeff = ppca(Y,3)

### **Data Types**

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'W0', init, 'Options', opt` specifies that the initial values for `'W0'` are in matrix `init` and `ppca` uses the options defined by `opt`.

### 'W0' - Initial value of $W$

matrix of random values (default) |  $p$ -by- $k$  matrix

Initial value of  $W$  in the probabilistic principal component analysis algorithm, specified as a comma-separated pair consisting of `'W0'` and a  $p$ -by- $k$  matrix.

#### Data Types

single | double

### 'v0' - Initial value of residual variance

random number (default) | positive scalar value

Initial value of residual variance, specified as the comma-separated pair consisting of `'v0'` and a positive scalar value.

#### Data Types

single | double

### 'Options' - Options for iterations

structure

Options for the iterations, specified as a comma-separated pair `'Options'` and a structure created by the `statset` function. `ppca` uses the following fields in the options structure.

'Display'	Level of display output. Choices are 'off', 'final', and 'iter'.
'MaxIter'	Maximum number of steps allowed. The default is 1000. Unlike in optimization settings, reaching the MaxIter value is regarded as convergence.
'TolFun'	Positive integer stating the termination tolerance for the cost function. The default is 1e-6.
'TolX'	Positive integer stating the convergence threshold for the relative change in the elements of $W$ . The default is 1e-6.

You can change the values of these fields and specify the new structure in ppca using the 'Options' name-value pair argument.

**Example:** `opt = statset('ppca'); opt.MaxIter = 2000; coeff = ppca(Y,3,'Options',opt);`

**Data Types**

struct

**Output Arguments**

**coeff - Principal component coefficients**

*p*-by-*k* matrix

Principal component coefficients, returned as a *p*-by-*k* matrix. Each column of `coeff` contains coefficients for one principal component. The columns are in the order of descending component variance, `pcvar`.

**score - Principal component scores**

*n*-by-*k* matrix

Principal component scores, returned as an *n*-by-*k* matrix. Rows of `score` correspond to observations, and columns correspond to components.

**pcvar - Principal component variances**

column vector



Principal component variances, which are the eigenvalues of the covariance matrix of  $Y$ , returned as a column vector.

### **$\mu$ - Estimated mean**

row vector

Estimated mean of each variable in  $Y$ , returned as a row vector.

### **$v$ - Isotropic residual variance**

scalar value

Isotropic residual variance, returned as a scalar value.

### **$S$ - Final results at convergence**

structure

Final results at convergence, returned as a structure containing the following fields.

$W$	$W$ at convergence.
Xexp	Conditional expectation of the estimated latent variable $x$ .
Recon	Reconstructed observations using $k$ principle components. This is a low dimension approximation of the input data $Y$ , and is equal to $\mu + \text{score} * \text{coeff}'$ .
$v$	Residual variance.
RMSResid	Root mean square of residuals.
NumIter	Number of iteration counts.
nloglk	Negative loglikelihood function value.

## Examples

### Perform Probabilistic Principal Component Analysis

Load the sample data.

```
load fisheriris
```

The double matrix `meas` consists of four types of measurements on the flowers, which, respectively, are the length and width of sepals and petals.

Introduce missing values randomly.

```
y = meas;
rng('default'); % for reproducibility
ix = random('unif',0,1,size(y))<0.20;
y(ix) = NaN;
```

Now, approximately 20% of the data is missing, indicated by NaN.

Perform probabilistic principal component analysis and request the component coefficients and variances.

```
[coeff,score,pcvar,mu] = ppca(y,3);
coeff
```

```
coeff =
```

```
    0.3562    0.6709   -0.5518
   -0.0765    0.7121    0.6332
    0.8592   -0.1596    0.0596
    0.3592   -0.1318    0.5395
```

```
pcvar
```

```
pcvar =
```

```
    4.0912
    0.2126
    0.0617
```

Perform principal component analysis using the alternating least squares algorithm and request the component coefficients and variances.

```
[coeff2,score2,pcvar2,mu2] = pca(y,'algorithm','als',...
```

```
'NumComponents',3);
coeff2

coeff2 =

    0.3376    0.4955    0.7404
   -0.0731    0.8607   -0.4479
    0.8657   -0.1169   -0.1231
    0.3623   -0.0087   -0.4859
```

```
pcvar2
```

```
pcvar2 =

    4.0734
    0.2651
    0.1221
```

The coefficients and the variances of the first two principal components are similar.

Another way to compare the results is to find the angle between the two spaces spanned by the coefficient vectors.

```
subspace(coeff,coeff2)

ans =

    0.0881
```

The angle between the two spaces is pretty small. This indicates that these two results are close to each other.

### **Change the Termination Criteria for Probabilistic Principal Component Analysis**

Load the sample data set.

```
load imports-85
```

Data matrix  $X$  has 13 continuous variables in columns 3 to 15: wheel-base, length, width, height, curb-weight, engine-size, bore, stroke, compression-ratio, horsepower, peak-rpm, city-mpg, and highway-mpg. The variables bore and stroke are missing four values in rows 56 to 59, and the variables horsepower and peak-rpm are missing two values in rows 131 and 132.

Perform probabilistic principal component analysis and display the first three principal components.

```
[coeff,score,pcvar] = ppca(X(:,3:15),3);
```

```
Warning: Maximum number of iterations 1000 reached'.  
> In ppca at 249
```

Change the termination tolerance for the cost function to 0.01.

```
opt = statset('ppca');  
opt.TolFun = 0.01;
```

Perform probabilistic principal component analysis.

```
[coeff,score,pcvar] = ppca(X(:,3:15),3,'Options',opt);
```

ppca now terminates before the maximum number of iterations is reached because it meets the tolerance for the cost function.

## Reconstruct Observations

Load the sample data.

```
load hald  
y = ingredients;
```

The ingredients data has 13 observations for 4 variables.

Introduce missing values to the data.

```
y(2:16:end) = NaN;
```

Every 16th value is NaN. This corresponds to 7.69% of the data.

Find the first three principal components of data using PPCA and display the reconstructed observations.

```
[coeff,score,pcvar,mu,v,S] = ppca(y,3);
S.Recon
```

```
ans =
```

```

    6.8533    25.8675    5.8388    59.8755
    1.0431    28.9690   14.9652    51.9758
   11.5770   56.5080    8.6352   20.5062
   11.0833   31.0707    8.0920   47.0764
    7.0684   52.2539    6.0753   33.0597
   11.0486   55.0442    9.0534   22.0410
    2.8494   70.8719   16.8338    5.8624
    1.0331   31.0267   19.6906   44.0321
    2.0401   54.0364   18.0440   22.0337
   20.7823   46.8096    3.7603   25.8075
    0.9540   39.9590   22.9495   31.1540
   10.8251   65.8498    8.8072   11.8420
    9.9174   67.9309    7.9087   11.9230
```

You can also reconstruct the observations using the principle components and the estimated mean.

```
t = score*coeff' + repmat(mu,13,1);
```

## Results at Convergence

Load the sample data.

```
load hald
```

Here, `ingredients` is a real-valued matrix of predictor variables.

Perform the probabilistic principal components analysis and display coefficients.

```
[coeff,score,pcvariance,mu,v,S] = ppca(ingredients,3);  
coeff
```

```
coeff =
```

```
   -0.0693   -0.6459    0.5673  
   -0.6786   -0.0184   -0.5440  
    0.0308    0.7552    0.4036  
    0.7306   -0.1102   -0.4684
```

Display the algorithm results at convergence of the PPCA.

```
S
```

```
S =
```

```
           W: [4x3 double]  
          Xexp: [13x3 double]  
         Recon: [13x4 double]  
           v: 0.2372  
    NumIter: 1000  
   RMSResid: 0.2340  
     nloglk: 149.3388
```

Display the matrix W.

```
S.W
```

```
ans =
```

```
    0.5624    2.0279    5.4075  
    4.8320   -10.3894    5.9202  
   -3.7521   -3.0555   -4.1552  
   -1.5144   11.7122   -7.2564
```

Orthogonalizing W recovers the coefficients.

orth(S.W)

ans =

-0.0693	0.6459	0.5673
-0.6786	0.0184	-0.5440
0.0308	-0.7552	0.4036
0.7306	0.1102	-0.4684

## Definitions

### Probabilistic Principal Component Analysis

Probabilistic principle component analysis (PPCA) is a method to estimate the principal axes when any data vector has one or more missing values.

PPCA is based on an isotropic error model. It seeks to relate a  $p$ -dimensional observation vector  $y$  to a corresponding  $k$ -dimensional vector of latent (or unobserved) variable  $x$ , which is normal with mean zero and covariance  $I(k)$ . The relationship is

$$y^T = W * x^T + \mu + \varepsilon,$$

where  $y$  is the row vector of observed variable,  $x$  is the row vector of latent variables, and  $\varepsilon$  is the isotropic error term.  $\varepsilon$  is Gaussian with mean zero and covariance of  $v * I(k)$ , where  $v$  is the residual variance. Here,  $k$  needs to be smaller than the rank for the residual variance to be greater than 0 ( $v > 0$ ). Standard principal component analysis, where the residual variance is zero, is the limiting case of PPCA. The observed variables,  $y$ , are conditionally independent given the values of the latent variables,  $x$ . So, the latent variables explain the correlations between the observation variables and the error explains the variability unique to a particular  $y_i$ . The  $p$ -by- $k$  matrix  $W$  relates the latent and observation variables, and the vector  $\mu$  permits the model to have a nonzero mean. PPCA assumes that the values are missing at random through the data set. This means that whether a data value is missing or not does not depend on the latent variable given the observed data values.

Under this model,

$$y \sim N\left(\mu, W * W^T + v * I(k)\right).$$

There is no closed-form analytical solution for  $W$  and  $v$ , so their estimates are determined by iterative maximization of the corresponding loglikelihood using an expectation-maximization (EM) algorithm. This EM algorithm handles missing values by treating them as additional latent variables. At convergence, the columns of  $W$  spans the subspace, but they are not orthonormal. `ppca` obtains the orthonormal coefficients, `coeff`, for the components by orthogonalization of  $W$ .

## References

- [1] Tipping, M. E., and C. M. Bishop. Probabilistic Principal Component Analysis. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, Vol. 61, No.3, 1999, pp. 611–622.
- [2] Roweis, S. “EM Algorithms for PCA and SPCA.” *In Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems*. Vol.10 (NIPS 1997), Cambridge, MA, USA: MIT Press, 1998, pp. 626–632.
- [3] Ilin, A., and T. Raiko. “Practical Approaches to Principal Component Analysis in the Presence of Missing Values.” *J. Mach. Learn. Res.* Vol. 11, August, 2010, pp. 1957–2000.

## See Also

`pca` | `pcacov` | `pcares` | `biplot` | `barttest` | `canoncorr` | `factoran` | `rotatefactors`



# gmdistribution.PComponents property

---

**Purpose** Input vector of mixing proportions

**Description** Optional input vector of mixing proportions  $p$ , or its default value.

**Purpose** Probability density functions

**Syntax**  
 $Y = \text{pdf}(\text{name}, X, A)$   
 $Y = \text{pdf}(\text{name}, X, A, B)$   
 $Y = \text{pdf}(\text{name}, X, A, B, C)$

**Description**  $Y = \text{pdf}(\text{name}, X, A)$  computes the probability density function for the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`. Densities are evaluated at the values in `X` and returned in `Y`.

If `X` and `A` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`. If `A` is a scalar, it is expanded to a constant matrix the same size as `X`.

`Y` is the common size of `X` and `A` after any necessary scalar expansion.

$Y = \text{pdf}(\text{name}, X, A, B)$  computes the probability density function for two-parameter families of distributions, where parameter values are given in `A` and `B`.

If `X`, `A`, and `B` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A` and `B`. If either `A` or `B` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, and `B` after any necessary scalar expansion.

$Y = \text{pdf}(\text{name}, X, A, B, C)$  computes the probability density function for three-parameter families of distributions, where parameter values are given in `A`, `B`, and `C`.

If `X`, `A`, `B`, and `C` are arrays, they must be the same size. If `X` is a scalar, it is expanded to a constant matrix the same size as `A`, `B`, and `C`. If any of `A`, `B` or `C` are scalars, they are expanded to constant matrices the same size as `X`.

`Y` is the common size of `X`, `A`, `B` and `C` after any necessary scalar expansion.

Acceptable strings for `name` are:

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'beta' or 'Beta'	“Beta Distribution” on page B-5	a	b	—
'bino' or 'Binomial'	“Binomial Distribution” on page B-9	n: number of trials	p: probability of success for each trial	—
'birnbaumsaunders'	“Birnbaum-Saunders Distribution” on page B-13	$\theta$	$\gamma$	—
'burr' or 'Burr':	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale parameter	c: shape parameter	k: shape parameter
'chi2' or 'Chisquare'	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—
'exp' or 'Exponential'	“Exponential Distribution” on page B-32	$\mu$ : mean	—	—
'ev' or 'Extreme Value'	“Extreme Value Distribution” on page B-35	$\mu$ : location parameter	$\sigma$ : scale parameter	—
'f' or 'F'	“F Distribution” on page B-41	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	—
'gam' or 'Gamma'	“Gamma Distribution” on page B-43	a: shape parameter	b: scale parameter	—

<b>name</b>	<b>Distribution</b>	<b>Input Parameter A</b>	<b>Input Parameter B</b>	<b>Input Parameter C</b>
'gev' or 'Generalized Extreme Value'	“Generalized Extreme Value Distribution” on page B-48	k: shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter
'gp' or 'Generalized Pareto'	“Generalized Pareto Distribution” on page B-53	k: tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter
'geo' or 'Geometric'	“Geometric Distribution” on page B-57	p: probability parameter	—	—
'hyge' or 'Hypergeometric'	“Hypergeometric Distribution” on page B-60	M: size of the population	K: number of items with the desired characteristic in the population	n: number of samples drawn
'inversegaussian'	“Inverse Gaussian Distribution” on page B-63	$\mu$	$\lambda$	—
'logistic'	“Logistic Distribution” on page B-76	$\mu$	$\sigma$	—
'loglogistic'	“Loglogistic Distribution” on page B-78	$\mu$	$\sigma$	—
'logn' or 'Lognormal'	“Lognormal Distribution” on page B-80	$\mu$	$\sigma$	—

<b>name</b>	<b>Distribution</b>	<b>Input Parameter A</b>	<b>Input Parameter B</b>	<b>Input Parameter C</b>
'nakagami'	“Nakagami Distribution” on page B-97	$\mu$	$\omega$	—
'nbin' or 'Negative Binomial'	“Negative Binomial Distribution” on page B-98	r: number of successes	p: probability of success in a single trial	—
'ncf' or 'Noncentral F'	“Noncentral F Distribution” on page B-106	v1: numerator degrees of freedom	v2: denominator degrees of freedom	$\delta$ : noncentrality parameter
'nct' or 'Noncentral t'	“Noncentral t Distribution” on page B-108	v: degrees of freedom	$\delta$ : noncentrality parameter	—
'ncx2' or 'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-104	v: degrees of freedom	$\delta$ : noncentrality parameter	—
'norm' or 'Normal'	“Normal Distribution” on page B-110	$\mu$ : mean	$\sigma$ : standard deviation	—
'poiss' or 'Poisson'	“Poisson Distribution” on page B-118	$\lambda$ : mean	—	—
'rayl' or 'Rayleigh'	“Rayleigh Distribution” on page B-121	b: scale parameter	—	—
'rician'	“Rician Distribution” on page B-123	s: noncentrality parameter	$\sigma$ : scale parameter	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
't' or 'T'	“Student’s t Distribution” on page B-124	$\nu$ : degrees of freedom	—	—
'tlocationscale'	“t Location-Scale Distribution” on page B-126	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter
'unif' or 'Uniform'	“Uniform Distribution (Continuous)” on page B-133	a: lower endpoint (minimum)	b: upper endpoint (maximum)	—
'unid' or 'Discrete Uniform'	“Uniform Distribution (Discrete)” on page B-139	N: maximum observable value	—	—
'wbl' or 'Weibull'	“Weibull Distribution” on page B-141	a: scale parameter	b: shape parameter	—

## Examples

Compute the pdf of the normal distribution with mean 0 and standard deviation 1 at inputs  $-2, -1, 0, 1, 2$ :

```
p1 = pdf('Normal', -2:2, 0, 1)
p1 =
    0.0540    0.2420    0.3989    0.2420    0.0540
```

The order of the parameters is the same as for `normpdf`.

Compute the pdfs of Poisson distributions with rate parameters 0, 1, ..., 4 at inputs 1, 2, ..., 5, respectively:

```
p2 = pdf('Poisson', 0:4, 1:5)
```

---

```
p2 =  
  0.3679  0.2707  0.2240  0.1954  0.1755
```

The order of the parameters is the same as for `poisspdf`.

**See Also**

`cdf` | `icdf` | `mle` | `random`

# gmdistribution.pdf

---

**Purpose** Probability density function for Gaussian mixture distribution

**Syntax** `y = pdf(obj,X)`

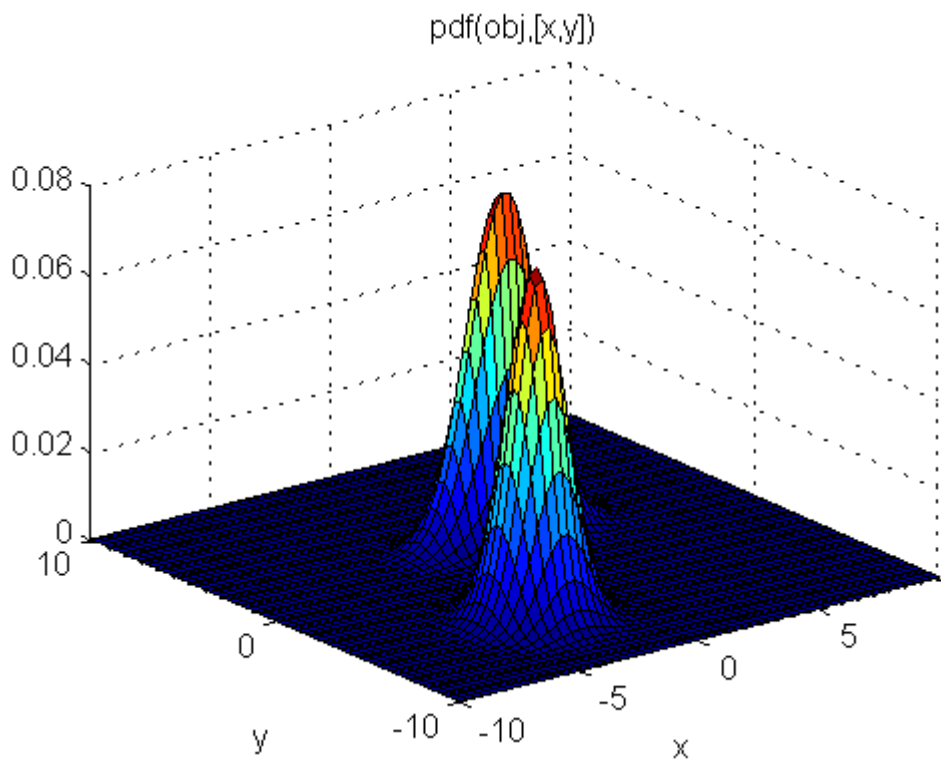
**Description** `y = pdf(obj,X)` returns a vector `y` of length  $n$  containing the values of the probability density function (`pdf`) for the `gmdistribution` object `obj`, evaluated at the  $n$ -by- $d$  data matrix `X`, where  $n$  is the number of observations and  $d$  is the dimension of the data. `obj` is an object created by `gmdistribution` or `fit`. `y(I)` is the pdf of observation `I`.

**Examples** Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2;-3 -5];
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);
p = ones(1,2)/2;
obj = gmdistribution(MU,SIGMA,p);

ezsurf(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
```





**See Also**

`gmdistribution` | `fit` | `cdf` | `mvnpdf`

# piecewisedistribution.pdf

---

**Purpose** Probability density function for piecewise distribution

**Syntax** `P = pdf(obj,X)`

**Description** `P = pdf(obj,X)` returns an array `P` of values of the probability density function for the piecewise distribution object `obj`, evaluated at the values in the array `X`.

---

**Note** For a Pareto tails object, the pdf is computed using the generalized Pareto distribution in the tails. In the center, the pdf is computed using the slopes of the cdf, which are interpolated between a set of discrete values. Therefore the pdf in the center is piecewise constant. It is noisy for a *cdf* specified in *paretotails* via the 'ecdf' option, and somewhat smoother for the 'kernel' option, but generally not a good estimate of the underlying density of the original data.

---

## Examples

Fit Pareto tails to a *t* distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);
[p,q] = boundary(obj)
p =
    0.1000
    0.9000
q =
   -1.7766
    1.8432

pdf(obj,q)
ans =
    0.2367
    0.1960
```

**See Also**     [paretotails](#) | [cdf](#)

# ProbDist.pdf

---

<b>Purpose</b>	Return probability density function (PDF) for ProbDist object	
<b>Syntax</b>	$Y = \text{pdf}(PD, X)$	
<b>Description</b>	$Y = \text{pdf}(PD, X)$ returns $Y$ , an array containing the probability density function (PDF) for the ProbDist object $PD$ , evaluated at values in $X$ .	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivParam or ProbDistUnivKernel.
	$X$	A numeric array of values where you want to evaluate the PDF.
<b>Output Arguments</b>	$Y$	An array containing the probability density function (PDF) for the ProbDist object $PD$ .
<b>See Also</b>	pdf	

<b>Purpose</b>	Probability density function of probability distribution object
<b>Syntax</b>	<code>y = pdf(pd,x)</code>
<b>Description</b>	<code>y = pdf(pd,x)</code> returns the probability density function (pdf) of the continuous probability distribution <code>pd</code> at the values in <code>x</code> . For discrete distributions, <code>pdf</code> returns the probability mass function.
<b>Input Arguments</b>	<p><b>pd - Probability distribution</b> probability distribution object</p> <p>Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code>. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using <code>fitdist</code> or the Distribution Fitting app.</p> <p><b>x - Values at which to calculate pdf</b> array</p> <p>Values at which to calculate pdf, specified as an array.</p> <p><b>Data Types</b> single   double</p>
<b>Output Arguments</b>	<p><b>y - Probability density function</b> array</p> <p>Probability density function of <code>pd</code>, evaluated at the values in data vector <code>x</code>, returned as a array. <code>y</code> has the same dimensions as input <code>x</code>.</p>
<b>Examples</b>	<p><b>Plot the pdf of a Standard Normal Distribution</b></p> <p>Create a standard normal distribution object.</p> <pre>pd = makedist('Normal')</pre> <pre>pd =</pre>

# prob.TruncatableDistribution.pdf

---

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

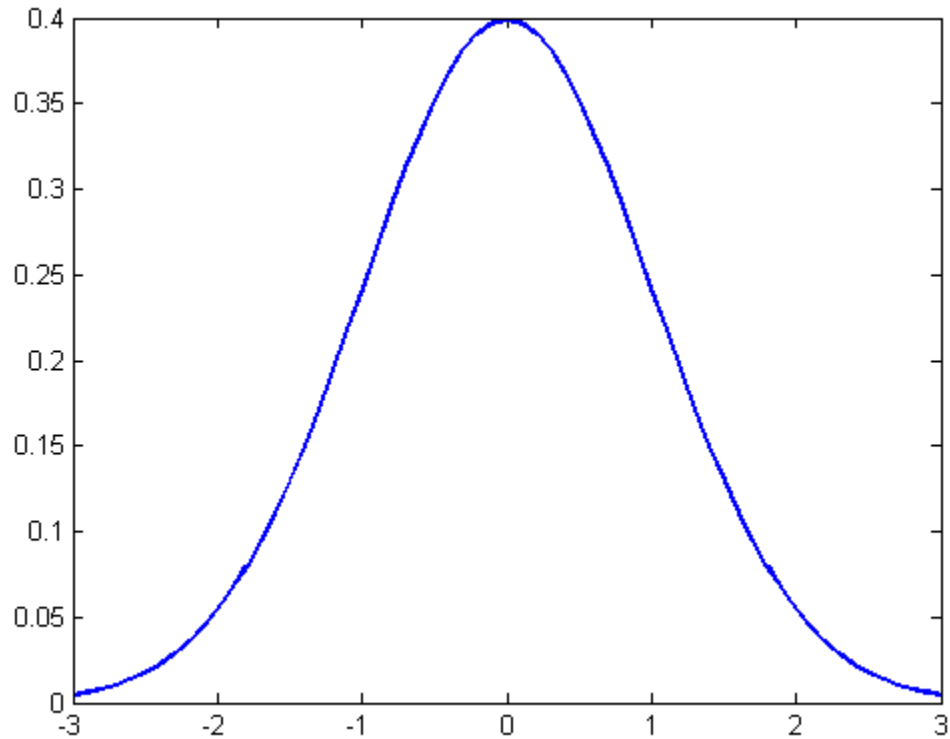
```
sigma = 1
```

Specify the x values and compute the pdf.

```
x = -3:.1:3;  
pdf_normal = pdf(pd,x);
```

Plot the pdf.

```
plot(x,pdf_normal,'LineWidth',2)
```



## Plot the pdf of a Weibull Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull','a',5,'b',2)
```

```
pd =
```

```
WeibullDistribution
```

# prob.TruncatableDistribution.pdf

---

Weibull distribution

A = 5

B = 2

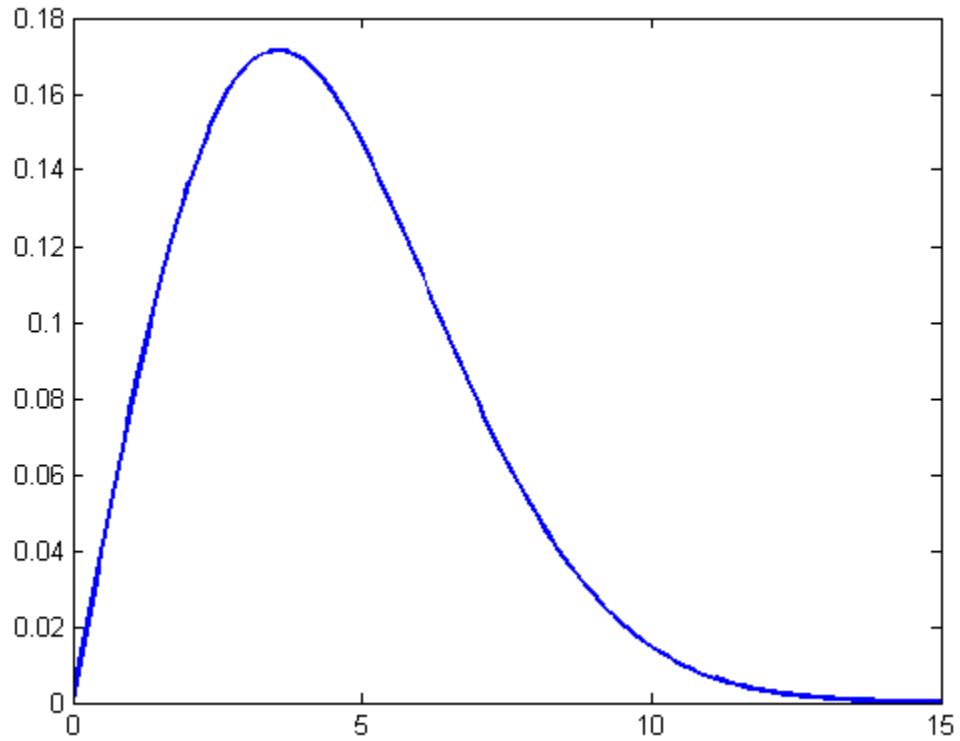
Specify the x values and compute the pdf.

```
x = 0:.1:15;  
pdf = pdf(pd,x);
```

Plot the pdf.

```
plot(x,pdf, 'LineWidth',2)
```





**See Also**

`makedist` | `fitdist` | `pdf` | `cdf` | `icdf` | `dfittool`

# pdist

---

**Purpose** Pairwise distance between pairs of objects

**Syntax**  
`D = pdist(X)`  
`D = pdist(X,distance)`

**Description** `D = pdist(X)` computes the Euclidean distance between pairs of objects in  $m$ -by- $n$  data matrix  $X$ . Rows of  $X$  correspond to observations, and columns correspond to variables.  $D$  is a row vector of length  $m(m-1)/2$ , corresponding to pairs of observations in  $X$ . The distances are arranged in the order  $(2,1)$ ,  $(3,1)$ , ...,  $(m,1)$ ,  $(3,2)$ , ...,  $(m,2)$ , ...,  $(m,m-1)$ .  $D$  is commonly used as a dissimilarity matrix in clustering or multidimensional scaling.

To save space and computation time,  $D$  is formatted as a vector. However, you can convert this vector into a square matrix using the `squareform` function so that element  $i, j$  in the matrix, where  $i < j$ , corresponds to the distance between objects  $i$  and  $j$  in the original data set.

`D = pdist(X,distance)` computes the distance between objects in the data matrix,  $X$ , using the method specified by `distance`, which can be any of the following character strings.

Metric	Description
'euclidean'	Euclidean distance (default).
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between rows in $X$ is scaled by dividing by the corresponding element of the standard deviation <code>S=nanstd(X)</code> . To specify another value for $S$ , use <code>D=pdist(X, 'seuclidean', S)</code> .
'cityblock'	City block metric.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use <code>D = pdist(X, 'minkowski', P)</code> , where $P$ is a scalar positive value of the exponent.

<b>Metric</b>	<b>Description</b>
'chebychev'	Chebychev distance (maximum coordinate difference).
'mahalanobis'	Mahalanobis distance, using the sample covariance of X as computed by nancov. To compute the distance with a different covariance, use <code>D = pdist(X, 'mahalanobis', C)</code> , where the matrix C is symmetric and positive definite.
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
'hamming'	Hamming distance, which is the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, which is the percentage of nonzero coordinates that differ.
custom distance function	<p>A distance function specified using @:  <code>D = pdist(X,@distfun)</code></p> <p>A distance function must be of form</p> $d2 = \text{distfun}(XI, XJ)$ <p>taking as arguments a 1-by-<i>n</i> vector XI, corresponding to a single row of X, and an <i>m2</i>-by-<i>n</i> matrix XJ, corresponding to multiple rows of X. <code>distfun</code> must accept a matrix XJ with an arbitrary number of rows. <code>distfun</code> must return an <i>m2</i>-by-1 vector of distances d2, whose <i>k</i>th element is the distance between XI and XJ(<i>k</i>, :).</p>

The output  $D$  is arranged in the order of  $((2,1),(3,1),\dots,(m,1),(3,2),\dots(m,2),\dots(m,m-1))$ , i.e. the lower left triangle of the full  $m$ -by- $m$  distance matrix in column order. To get the distance between the  $i$ th and  $j$ th observations ( $i < j$ ), either use the formula  $D((i-1)*(m-i/2)+j-i)$ , or use the helper function  $Z = \text{squareform}(D)$ , which returns an  $m$ -by- $m$  square symmetric matrix, with the  $(i,j)$  entry equal to distance between observation  $i$  and observation  $j$ .

### Metrics

Given an  $m$ -by- $n$  data matrix  $X$ , which is treated as  $m$  (1-by- $n$ ) row vectors  $x_1, x_2, \dots, x_m$ , the various distances between the vector  $x_s$  and  $x_t$  are defined as follows:

- Euclidean distance

$$d_{st}^2 = (x_s - x_t)(x_s - x_t)'$$

Notice that the Euclidean distance is a special case of the Minkowski metric, where  $p = 2$ .

- Standardized Euclidean distance

$$d_{st}^2 = (x_s - x_t)V^{-1}(x_s - x_t)'$$

where  $V$  is the  $n$ -by- $n$  diagonal matrix whose  $j$ th diagonal element is  $S(j)^2$ , where  $S$  is the vector of standard deviations.

- Mahalanobis distance

$$d_{st}^2 = (x_s - x_t)C^{-1}(x_s - x_t)'$$

where  $C$  is the covariance matrix.

- City block metric

$$d_{st} = \sum_{j=1}^n |x_{sj} - x_{tj}|$$

Notice that the city block distance is a special case of the Minkowski metric, where  $p=1$ .

- Minkowski metric

$$d_{st} = \sqrt[p]{\sum_{j=1}^n |x_{sj} - x_{tj}|^p}$$

Notice that for the special case of  $p = 1$ , the Minkowski metric gives the city block metric, for the special case of  $p = 2$ , the Minkowski metric gives the Euclidean distance, and for the special case of  $p = \infty$ , the Minkowski metric gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j \{|x_{sj} - x_{tj}|\}$$

Notice that the Chebychev distance is a special case of the Minkowski metric, where  $p = \infty$ .

- Cosine distance

$$d_{st} = 1 - \frac{x_s x_t'}{\sqrt{(x_s x_s')(x_t x_t')}}}$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \bar{x}_s)(x_t - \bar{x}_t)'}{\sqrt{(x_s - \bar{x}_s)(x_s - \bar{x}_s)' (x_t - \bar{x}_t)(x_t - \bar{x}_t)'}}$$

where

$$\bar{x}_s = \frac{1}{n} \sum_j x_{sj} \quad \text{and} \quad \bar{x}_t = \frac{1}{n} \sum_j x_{tj}$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq x_{tj}) / n)$$

- Jaccard distance

$$d_{st} = \frac{\#[(x_{sj} \neq x_{tj}) \cap ((x_{sj} \neq 0) \cup (x_{tj} \neq 0))]}{\#[(x_{sj} \neq 0) \cup (x_{tj} \neq 0)]}$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \bar{r}_s)(r_t - \bar{r}_t)'}{\sqrt{(r_s - \bar{r}_s)(r_s - \bar{r}_s)'} \sqrt{(r_t - \bar{r}_t)(r_t - \bar{r}_t)'}}$$

where

- $r_{sj}$  is the rank of  $x_{sj}$  taken over  $x_{1j}, x_{2j}, \dots, x_{mj}$ , as computed by `tiedrank`
- $r_s$  and  $r_t$  are the coordinate-wise rank vectors of  $x_s$  and  $x_t$ , i.e.,  
 $r_s = (r_{s1}, r_{s2}, \dots, r_{sn})$

$$\bar{r}_s = \frac{1}{n} \sum_j r_{sj} = \frac{(n+1)}{2}$$

$$\bar{r}_t = \frac{1}{n} \sum_j r_{tj} = \frac{(n+1)}{2}$$

## Examples

Generate random data and find the unweighted Euclidean distance and then find the weighted distance using two different methods:

```
% Compute the ordinary Euclidean distance.
X = randn(100, 5);
D = pdist(X, 'euclidean'); % euclidean distance
```

```
% Compute the Euclidean distance with each coordinate
% difference scaled by the standard deviation.
```

```
Dstd = pdist(X,'seuclidean');

% Use a function handle to compute a distance that weights
% each coordinate contribution differently.
Wgts = [.1 .3 .3 .2 .1];      % coordinate weights
weuc = @(XI,XJ,W)(sqrt(bsxfun(@minus,XI,XJ).^2 * W'));
Dwgt = pdist(X, @(Xi,Xj) weuc(Xi,Xj,Wgts));
```

**See Also**

[cluster](#) | [clusterdata](#) | [cmdscales](#) | [cophenet](#) | [dendrogram](#) | [inconsistent](#) | [linkage](#) | [pdist2](#) | [silhouette](#) | [squareform](#)

# pdist2

---

## Purpose

Pairwise distance between two sets of observations

## Syntax

```
D = pdist2(X,Y)
D = pdist2(X,Y,distance)
D = pdist2(X,Y,'minkowski',P)
D = pdist2(X,Y,'mahalanobis',C)
D = pdist2(X,Y,distance,'Smallest',K)
D = pdist2(X,Y,distance,'Largest',K)
[D,I] = pdist2(X,Y,distance,'Smallest',K)
[D,I] = pdist2(X,Y,distance,'Largest',K)
```

## Description

`D = pdist2(X,Y)` returns a matrix `D` containing the Euclidean distances between each pair of observations in the  $mx$ -by- $n$  data matrix `X` and  $my$ -by- $n$  data matrix `Y`. Rows of `X` and `Y` correspond to observations, columns correspond to variables. `D` is an  $mx$ -by- $my$  matrix, with the  $(i,j)$  entry equal to distance between observation  $i$  in `X` and observation  $j$  in `Y`. The  $(i,j)$  entry will be NaN if observation  $i$  in `X` or observation  $j$  in `Y` contain NaNs.

`D = pdist2(X,Y,distance)` computes `D` using `distance`. Choices are:

Metric	Description
'euclidean'	Euclidean distance (default).
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between rows in <code>X</code> and <code>Y</code> is scaled by dividing by the corresponding element of the standard deviation computed from <code>X</code> , <code>S=nanstd(X)</code> . To specify another value for <code>S</code> , use <code>D = PDIST2(X,Y,'seuclidean',S)</code> .
'cityblock'	City block metric.
'minkowski'	Minkowski distance. The default exponent is 2. To compute the distance with a different exponent, use <code>D = pdist2(X,Y,'minkowski',P)</code> , where the exponent <code>P</code> is a scalar positive value.



<b>Metric</b>	<b>Description</b>
'chebychev'	Chebychev distance (maximum coordinate difference).
'mahalanobis'	Mahalanobis distance, using the sample covariance of X as computed by nancov. To compute the distance with a different covariance, use $D = \text{pdist2}(X,Y, \text{'mahalanobis'}, C)$ where the matrix C is symmetric and positive definite.
'cosine'	One minus the cosine of the included angle between points (treated as vectors).
'correlation'	One minus the sample correlation between points (treated as sequences of values).
'spearman'	One minus the sample Spearman's rank correlation between observations, treated as sequences of values.
'hamming'	Hamming distance, the percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
function	<p>A distance function specified using @:  <math>D = \text{pdist2}(X,Y,@\text{distfun})</math>.</p> <p>A distance function must be of the form</p> <pre>function D2 = distfun(ZI, ZJ)</pre> <p>taking as arguments a 1-by-n vector ZI containing a single observation from X or Y, an m2-by-n matrix ZJ containing multiple observations from X or Y, and returning an m2-by-1 vector of distances D2, whose Jth element is the distance between the observations ZI and ZJ(J, :).</p>

Metric	Description
	If your data is not sparse, generally it is faster to use a built-in distance than to use a function handle.

$D = \text{pdist2}(X,Y,\text{distance},\text{'Smallest'},K)$  returns a  $K$ -by- $m_y$  matrix  $D$  containing the  $K$  smallest pairwise distances to observations in  $X$  for each observation in  $Y$ .  $\text{pdist2}$  sorts the distances in each column of  $D$  in ascending order.  $D = \text{pdist2}(X,Y,\text{distance},\text{'Largest'},K)$  returns the  $K$  largest pairwise distances sorted in descending order. If  $K$  is greater than  $m_x$ ,  $\text{pdist2}$  returns an  $m_x$ -by- $m_y$  distance matrix. For each observation in  $Y$ ,  $\text{pdist2}$  finds the  $K$  smallest or largest distances by computing and comparing the distance values to all the observations in  $X$ .

$[D,I] = \text{pdist2}(X,Y,\text{distance},\text{'Smallest'},K)$  returns a  $K$ -by- $m_y$  matrix  $I$  containing indices of the observations in  $X$  corresponding to the  $K$  smallest pairwise distances in  $D$ .  $[D,I] = \text{pdist2}(X,Y,\text{distance},\text{'Largest'},K)$  returns indices corresponding to the  $K$  largest pairwise distances.

## Metrics

Given an  $m_x$ -by- $n$  data matrix  $X$ , which is treated as  $m_x$  (1-by- $n$ ) row vectors  $x_1, x_2, \dots, x_{m_x}$ , and  $m_y$ -by- $n$  data matrix  $Y$ , which is treated as  $m_y$  (1-by- $n$ ) row vectors  $y_1, y_2, \dots, y_{m_y}$ , the various distances between the vector  $x_s$  and  $y_t$  are defined as follows:

- Euclidean distance

$$d_{st}^2 = (x_s - y_t)(x_s - y_t)'$$

Notice that the Euclidean distance is a special case of the Minkowski metric, where  $p=2$ .

- Standardized Euclidean distance

$$d_{st}^2 = (x_s - y_t)V^{-1}(x_s - y_t)'$$

where  $V$  is the  $n$ -by- $n$  diagonal matrix whose  $j$ th diagonal element is  $S(j)^2$ , where  $S$  is the vector of standard deviations.

- Mahalanobis distance

$$d_{st}^2 = (x_s - y_t)C^{-1}(x_s - y_t)'$$

where  $C$  is the covariance matrix.

- City block metric

$$d_{st} = \sum_{j=1}^n |x_{sj} - y_{tj}|$$

Notice that the city block distance is a special case of the Minkowski metric, where  $p=1$ .

- Minkowski metric

$$d_{st} = \sqrt[p]{\sum_{j=1}^n |x_{sj} - y_{tj}|^p}$$

Notice that for the special case of  $p = 1$ , the Minkowski metric gives the City Block metric, for the special case of  $p = 2$ , the Minkowski metric gives the Euclidean distance, and for the special case of  $p=\infty$ , the Minkowski metric gives the Chebychev distance.

- Chebychev distance

$$d_{st} = \max_j \{|x_{sj} - y_{tj}|\}$$

Notice that the Chebychev distance is a special case of the Minkowski metric, where  $p=\infty$ .

- Cosine distance

$$d_{st} = \left( 1 - \frac{x_s y_t'}{\sqrt{(x_s x_s')(y_t y_t')}} \right)$$

- Correlation distance

$$d_{st} = 1 - \frac{(x_s - \bar{x}_s)(y_t - \bar{y}_t)'}{\sqrt{(x_s - \bar{x}_s)(x_s - \bar{x}_s)'(y_t - \bar{y}_t)(y_t - \bar{y}_t)'}}$$

where

$$\bar{x}_s = \frac{1}{n} \sum_j x_{sj} \quad \text{and}$$
$$\bar{y}_t = \frac{1}{n} \sum_j y_{tj}$$

- Hamming distance

$$d_{st} = (\#(x_{sj} \neq y_{tj}) / n)$$

- Jaccard distance

$$d_{st} = \frac{\#[(x_{sj} \neq y_{tj}) \cap ((x_{sj} \neq 0) \cup (y_{tj} \neq 0))]}{\#[(x_{sj} \neq 0) \cup (y_{tj} \neq 0)]}$$

- Spearman distance

$$d_{st} = 1 - \frac{(r_s - \bar{r}_s)(r_t - \bar{r}_t)'}{\sqrt{(r_s - \bar{r}_s)(r_s - \bar{r}_s)'(r_t - \bar{r}_t)(r_t - \bar{r}_t)'}}$$

where

- $r_{sj}$  is the rank of  $x_{sj}$  taken over  $x_{1j}, x_{2j}, \dots, x_{mx,j}$ , as computed by `tiedrank`
- $r_{tj}$  is the rank of  $y_{tj}$  taken over  $y_{1j}, y_{2j}, \dots, y_{my,j}$ , as computed by `tiedrank`
- $r_s$  and  $r_t$  are the coordinate-wise rank vectors of  $x_s$  and  $y_t$ , i.e.  $r_s = (r_{s1}, r_{s2}, \dots, r_{sn})$  and  $r_t = (r_{t1}, r_{t2}, \dots, r_{tn})$
- $\bar{r}_s = \frac{1}{n} \sum_j r_{sj} = \frac{(n+1)}{2}$
- $\bar{r}_t = \frac{1}{n} \sum_j r_{tj} = \frac{(n+1)}{2}$

## Examples

Generate random data and find the unweighted Euclidean distance, then find the weighted distance using two different methods:

```
% Compute the ordinary Euclidean distance
X = randn(100, 5);
Y = randn(25, 5);
D = pdist2(X,Y,'euclidean'); % euclidean distance

% Compute the Euclidean distance with each coordinate
% difference scaled by the standard deviation
Dstd = pdist2(X,Y,'seuclidean');

% Use a function handle to compute a distance that weights
% each coordinate contribution differently.
Wgts = [.1 .3 .3 .2 .1];
weuc = @(XI,XJ,W)(sqrt(bsxfun(@minus,XI,XJ).^2 * W'));
Dwgt = pdist2(X,Y, @(Xi,Xj) weuc(Xi,Xj,Wgts));
```

## See Also

`pdist` | `createns` | `knnsearch` | `KDTreeSearcher` | `ExhaustiveSearcher`

# pearsrnd

---

## Purpose

Pearson system random numbers

## Syntax

```
r = pearsrnd(mu,sigma,skew,kurt,m,n)
r = pearsrnd(mu,sigma,skew,kurt)
r = pearsrnd(mu,sigma,skew,kurt,m,n,...)
r = pearsrnd(mu,sigma,skew,kurt,[m,n,...])
[r,type] = pearsrnd(...)
[r,type,coefs] = pearsrnd(...)
```

## Description

`r = pearsrnd(mu,sigma,skew,kurt,m,n)` returns an  $m$ -by- $n$  matrix of random numbers drawn from the distribution in the Pearson system with mean `mu`, standard deviation `sigma`, skewness `skew`, and kurtosis `kurt`. The parameters `mu`, `sigma`, `skew`, and `kurt` must be scalars.

---

**Note** Because `r` is a random sample, its sample moments, especially the skewness and kurtosis, typically differ somewhat from the specified distribution moments.

`pearsrnd` uses the definition of kurtosis for which a normal distribution has a kurtosis of 3. Some definitions of kurtosis subtract 3, so that a normal distribution has a kurtosis of 0. The `pearsrnd` function does not use this convention.

---

Some combinations of moments are not valid; in particular, the kurtosis must be greater than the square of the skewness plus 1. The kurtosis of the normal distribution is defined to be 3.

`r = pearsrnd(mu,sigma,skew,kurt)` returns a scalar value.

`r = pearsrnd(mu,sigma,skew,kurt,m,n,...)` or `r = pearsrnd(mu,sigma,skew,kurt,[m,n,...])` returns an  $m$ -by- $n$ -by-... array.

`[r,type] = pearsrnd(...)` returns the type of the specified distribution within the Pearson system. `type` is a scalar integer from 0

to 7. Set `m` and `n` to 0 to identify the distribution type without generating any random values.

The seven distribution types in the Pearson system correspond to the following distributions:

- 0 — Normal distribution
- 1 — Four-parameter beta distribution
- 2 — Symmetric four-parameter beta distribution
- 3 — Three-parameter gamma distribution
- 4 — Not related to any standard distribution. The density is proportional to:

$$(1 + ((x - a)/b)^2)^{-c} \exp(-d \arctan((x - a)/b)).$$

- 5 — Inverse gamma location-scale distribution
- 6 —  $F$  location-scale distribution
- 7 — Student's  $t$  location-scale distribution

`[r,type,coefs] = pearsrnd(...)` returns the coefficients `coefs` of the quadratic polynomial that defines the distribution via the differential equation

$$\frac{d}{dx} \log(p(x)) = \frac{-(a+x)}{c(0) + c(1)x + c(2)x^2}.$$

## Examples

Generate random values from the standard normal distribution:

```
r = pearsrnd(0,1,0,3,100,1); % Equivalent to randn(100,1)
```

Determine the distribution type:

```
[r,type] = pearsrnd(0,1,1,4,0,0);
r =
    []
```

# pearsrnd

---

type =  
1

## References

[1] Johnson, N.L., S. Kotz, and N. Balakrishnan (1994) Continuous Univariate Distributions, Volume 1, Wiley-Interscience, Pg 15, Eqn 12.33.

## See Also

random | johnsrnd



## Purpose

Compute Receiver Operating Characteristic (ROC) curve or other performance curve for classifier output

## Syntax

```
[X,Y] = perfcurve(labels,scores,posclass)
[X,Y] = perfcurve(labels,scores,posclass,'Name',value)
[X,Y,T,AUC,OPTROCPT,SUBY,SUBYNAMES] = perfcurve(labels,scores,
    posclass)
[X,Y,T,AUC] = perfcurve(labels,scores,posclass)
```

## Description

`[X,Y] = perfcurve(labels,scores,posclass)` computes a ROC curve for a vector of classifier predictions `scores` given true class labels, `labels`. `labels` can be a numeric vector, logical vector, character matrix, cell array of strings or categorical vector. `scores` is a numeric vector of scores returned by a classifier for some data. `posclass` is the positive class label (scalar), either numeric (for numeric labels), logical (for logical labels), or char. The returned values `X` and `Y` are coordinates for the performance curve and can be visualized with `plot(X,Y)`. For more information on `labels`, `scores`, and `posclass`, see “Input Arguments” on page 21-2459 . For more information on `X` and `Y`, see “Output Arguments” on page 21-2463.

`[X,Y] = perfcurve(labels,scores,posclass,'Name',value)` specifies one or more optional parameter name/value pairs, with `Name` in single quotes. See “Input Arguments” on page 21-2459 for a list of inputs, parameter name/value pairs, and respective explanations.

See “Grouping Variables” on page 2-51 for more information on grouping variables.

`[X,Y,T,AUC,OPTROCPT,SUBY,SUBYNAMES] = perfcurve(labels,scores, posclass)` returns:

- An array of thresholds on classifier scores for the computed values of `X` and `Y` (`T`).
- The area under curve (AUC) for the computed values of `X` and `Y`.
- The optimal operating point of the ROC curve (`OPTROCPT`).
- An array of `Y` values for negative subclasses (`SUBY`).

- A cell array of negative class names (SUBYNAMES).

For more information on each output, see “Output Arguments” on page 21-2463.

`[X,Y,T,AUC] = perfcurve(labels,scores,posclass)` also returns pointwise confidence bounds for the computed values X, Y, T, and AUC if you supply cell arrays for `labels` and `scores` or set `NBoot` (see “Input Arguments” on page 21-2459 ) to a positive integer. To compute the confidence bounds, `perfcurve` uses either vertical averaging (VA) or threshold averaging (TA). The returned values Y are an  $m$ -by-3 array in which the 1st element in every row gives the mean value, the 2nd element gives the lower bound and the 3rd element gives the upper bound. The returned AUC is a row-vector with 3 elements following the same convention. For VA, the returned values T are an  $m$ -by-3 array and X is a column-vector. For TA, the returned values X are an  $m$ -by-3 matrix and T is a column-vector.

`perfcurve` computes confidence bounds using either cross validation or bootstrap. If you supply cell arrays for `labels` and `scores`, `perfcurve` uses cross validation and treats elements in the cell arrays as cross validation folds. `labels` can be a cell array of numeric vectors, logical vectors, character matrices, cell arrays of strings or categorical vectors. All elements in `labels` must have the same type. `scores` is a cell array of numeric vectors. The cell arrays for `labels` and `scores` must have the same number of elements, and the number of labels in cell  $k$  must be equal to the number of scores in cell  $k$  for any  $k$  in the range from 1 to the number of elements in `scores`.

If you set `NBoot` to a positive integer, `perfcurve` generates `nboot` bootstrap replicas to compute pointwise confidence bounds. You cannot supply cell arrays for `labels` and `scores` and set `NBoot` to a positive integer at the same time.

`perfcurve` returns pointwise confidence bounds. It does not return a simultaneous confidence band for the entire curve.

If you use 'XCrit' or 'YCrit' options described below to set the criterion for X or Y to an anonymous function, `perfcurve` can only compute confidence bounds by bootstrap.

## Input Arguments

labels	labels can be a numeric vector, logical vector, character matrix, cell array of strings or categorical vector.
scores	scores is a numeric vector of scores returned by a classifier for some data. This vector must have as many elements as labels does.
posclass	posclass is the positive class label (scalar), either numeric (for numeric labels) or char. The specified positive class must be in the array of input labels.

## Name-Value Pair Arguments

### Name

### Value and Description

negClass

List of negative classes. Can be either a numeric array or an array of chars or a cell array of strings. By default, negClass is set to 'all' and all classes found in the input array of labels that are not the positive class are considered negative. If negClass is a subset of the classes found in the input array of labels, instances with labels that do not belong to either positive or negative classes are discarded.

xCrit

Criterion to compute for X. This criterion must be a monotone function of the positive class score. perfcurve supports the following criteria:

- TP — Number of true positive instances.
- FN — Number of false negative instances.
- FP — Number of false positive instances.
- TN — Number of true negative instances.
- TP+FP — Sum of TP and FP.

## Name

## Value and Description

- RPP — Rate of positive predictions.  
 $RPP = (TP + FP) / (TP + FN + FP + TN)$
- RNP — Rate of negative predictions.  
 $RNP = (TN + FN) / (TP + FN + FP + TN)$
- accu — Accuracy.  $accu = (TP + TN) / (TP + FN + FP + TN)$
- TPR, sens, reca — True positive rate, sensitivity, recall.  $TPR, sens, reca = TP / (TP + FN)$
- FNR, miss — False negative rate, miss.  $FNR, miss = FN / (TP + FN)$
- FPR, fall — False positive rate, fallout.  $FPR, fall = FP / (TN + FP)$
- TNR, spec — True negative rate, specificity.  $TNR, spec = TN / (TN + FP)$
- PPV, prec — Positive predictive value, precision.  $PPV, prec = TP / (TP + FP)$
- NPV — Negative predictive value.  $NPV = TN / (TN + FN)$
- ecost — Expected cost.  
 $ecost = (TP * COST(P|P) + FN * COST(N|P) + FP * COST(P|N) + TN * COST(N|N)) / (TP + FN + FP + TN)$

In addition, you can define an arbitrary criterion by supplying an anonymous function of three arguments, `(C, scale, cost)`, where `C` is a 2-by-2 confusion matrix, `scale` is a 2-by-1 array of class scales, and `cost` is a 2-by-2 misclassification cost matrix.

---

**Caution** Some of these criteria return NaN values at one of the two special thresholds, 'reject all' and 'accept all'.

---

<b>Name</b>	<b>Value and Description</b>
yCrit	Criterion to compute for Y. perfcurve supports the same criteria as for X. This criterion does not have to be a monotone function of the positive class score.
XVals	Values for the X criterion. The default value for xVals is 'all' and perfcurve computes X and Y values for all scores. If the value for xVals is not 'all', it must be a numeric array. In this case, perfcurve computes X and Y only for the specified xVals.
TVals	Thresholds for the positive class score. By default, TVals is unset and perfcurve computes X, Y, and T values for all scores. You can set TVals to either 'all' or a numeric array. If TVals is set to 'all' or unset and XVals is unset, perfcurve returns X, Y, and T values for all scores and computes pointwise confidence bounds for Y and X using threshold averaging. If TVals is set to a numeric array, perfcurve returns X, Y, and T values for the specified thresholds and computes pointwise confidence bounds for Y and X at these thresholds using threshold averaging. You cannot set XVals and TVals at the same time.
UseNearest	'on' to use nearest values found in the data instead of the specified numeric XVals or TVals and 'off' otherwise. If you specify numeric XVals and set UseNearest to 'on', perfcurve returns nearest unique values X found in the data, as well as corresponding values of Y and T. If you specify numeric XVals and set UseNearest to 'off', perfcurve returns these XVals sorted. By default this parameter is set to 'on'. If you compute confidence bounds by cross validation or bootstrap, this parameter is always 'off'.

Name	Value and Description
ProcessNaN	Specifies how perfcurve processes NaN scores. The default value is 'ignore' and perfcurve removes observations with NaN scores from the data. If you set the parameter to 'addtofalse', perfcurve adds instances with NaN scores to false classification counts in the respective class. That is, perfcurve always counts instances from the positive class as false negative (FN), and always counts instances from the negative class as false positive (FP).
Prior	Either string or array with two elements. It represents prior probabilities for the positive and negative class, respectively. Default is 'empirical', that is, perfcurve derives prior probabilities from class frequencies. If set to 'uniform', perfcurve sets all prior probabilities equal.
Cost	A 2-by-2 matrix of misclassification costs $[C(P P) \ C(N P); C(P N) \ C(N N)]$ where $C(I J)$ is the cost of misclassifying class $J$ as class $I$ . By default set to $[0 \ 0.5; \ 0.5 \ 0]$ .
Alpha	A numeric value between 0 and 1. perfcurve returns $100*(1-Alpha)$ percent pointwise confidence bounds for X, Y, T and AUC. By default set to 0.05 for 95% confidence interval.
Weights	A numeric vector of nonnegative observation weights. This vector must have as many elements as scores or labels do. If you supply cell arrays for scores and labels and you need to supply weights, you must supply them as a cell array too. In this case, every element in weights must be a numeric vector with as many elements as the corresponding element in scores: <code>numel(weights{1})==numel(scores{1})</code> etc. To compute X, Y and T or to compute confidence bounds by cross validation, perfcurve uses these observation weights instead of observation counts. To compute confidence bounds by bootstrap, perfcurve samples $N$ out of $N$ with replacement using these weights as multinomial sampling probabilities.

Name	Value and Description	
NBoot	Number of bootstrap replicas for computation of confidence bounds. Must be a positive integer. By default this parameter is set to zero, and bootstrap confidence bounds are not computed. If you supply cell arrays for <code>labels</code> and <code>scores</code> , this parameter must be set to zero because <code>perfcurve</code> cannot use both cross validation and bootstrap to compute confidence bounds.	
BootType	Confidence interval type <code>bootci</code> uses to compute confidence bounds. You can specify any type supported by <code>bootci</code> . By default set to <code>'bca'</code> .	
BootArg	Optional input arguments for <code>bootci</code> used to compute confidence bounds. You can specify all arguments <code>bootci</code> supports. Empty by default.	
<b>Output Arguments</b>	X	<i>x</i> -coordinates for the performance curve. By default, X is false positive rate, FPR, (equivalently, fallout, or 1-specificity). To change this output, use the <code>'xCrit'</code> name/value input. For accepted criterion, see <code>'xCrit'</code> in “Input Arguments” on page 21-2459 for more information.
	Y	<i>y</i> -coordinates for the performance curve. By default, Y is true positive rate, TPR, (equivalently, recall, or sensitivity). To change this output, use the <code>'yCrit'</code> input. For accepted criterion, see <code>'xCrit'</code> in “Input Arguments” on page 21-2459 for more information.

**T** An array of thresholds on classifier scores for the computed values of X and Y. It has the same number of rows as X and Y. For each threshold, TP is the count of true positive observations with scores greater or equal to this threshold, and FP is the count of false positive observations with scores greater or equal to this threshold. `perfcurve` defines negative counts, TN and FN, in a similar way then sorts the thresholds in the descending order which corresponds to the ascending order of positive counts.

For the M distinct thresholds found in the array of scores, `perfcurve` returns the X, Y and T arrays with M+1 rows. `perfcurve` sets elements T(2:M+1) to the distinct thresholds, and T(1) replicates T(2). By convention, T(1) represents the highest 'reject all' threshold and `perfcurve` computes the corresponding values of X and Y for TP=0 and FP=0. T(end) is the lowest 'accept all' threshold for which TN=0 and FN=0.

**AUC** The area under curve (AUC) for the computed values of X and Y. If you set `xVals` to 'all' (the default), `perfcurve` computes AUC using the returned X and Y values. If `xVals` is a numeric array, `perfcurve` computes AUC using X and Y values found from all distinct scores in the interval specified by the smallest and largest elements of `xVals`. More precisely, `perfcurve` finds X values for all distinct thresholds as if `xVals` were set to 'all', then uses a subset of these (with corresponding Y values) between `min(xVals)` and `max(xVals)` to compute AUC. The function uses trapezoidal approximation to estimate the area. If the first or last value of X or Y are NaNs, `perfcurve` removes them to allow calculation of AUC. This takes care of criteria that produce NaNs for the special 'reject all' or 'accept all' thresholds, for example, positive predictive value (PPV) or negative predictive value (NPV).



**OPTROCPPT** The optimal operating point of the ROC curve as an array of size 1-by-2 with FPR and TPR values for the optimal ROC operating point. `perfcurve` computes `optrocppt` only for the standard ROC curve and sets to NaNs otherwise. To obtain the optimal operating point for the ROC curve, `perfcurve` first finds the slope,  $S$ , using

$$S = \frac{\text{cost}(P | N) - \text{cost}(N | N) * \frac{N}{P}}{\text{cost}(N | P) - \text{cost}(P | P)}$$

where  $\text{cost}(I | J)$  is the cost of assigning an instance of class  $J$  to class  $I$ , and  $P=TP+FN$  and  $N=TN+FP$  are the total instance counts in the positive and negative class, respectively. `perfcurve` then finds the optimal operating point by moving the straight line with slope  $S$  from the upper left corner of the ROC plot (FPR=0, TPR=1) down and to the right until it intersects the ROC curve.

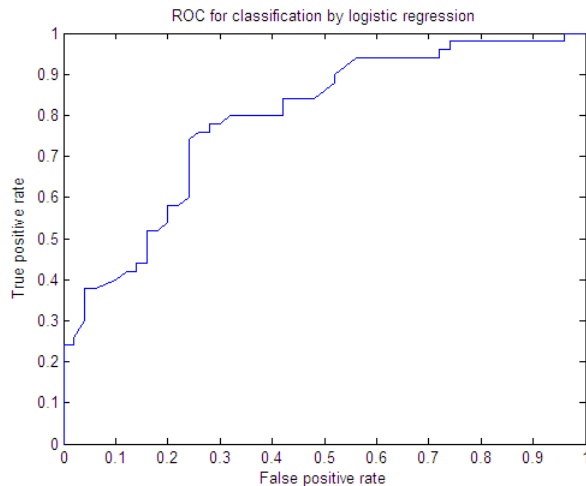
**SUBY** An array of  $Y$  values for negative subclasses. If you only specify one negative class, `SUBY` is identical to  $Y$ . Otherwise `SUBY` is a matrix of size  $M$ -by- $K$ , where  $M$  is the number of returned values for  $X$  and  $Y$ , and  $K$  is the number of negative classes. `perfcurve` computes  $Y$  values by summing counts over all negative classes. `SUBY` gives values of the  $Y$  criterion for each negative class separately. For each negative class, `perfcurve` places a new column in `SUBY` and fills it with  $Y$  values for  $TN$  and  $FP$  counted just for this class.

**SUBYNAMES** A cell array of negative class names. If you provide an input array, `negClass`, of negative class names, `perfcurve` copies it into `SUBYNAMES`. If you do not provide `negClass`, `perfcurve` extracts `SUBYNAMES` from input labels. The order of `SUBYNAMES` is the same as the order of columns in `SUBY`, that is, `SUBY(:, 1)` is for negative class `SUBYNAMES{1}` etc.

## Examples

Plot the ROC curve for classification by logistic regression:

```
load fisheriris
x = meas(51:end,1:2);
% iris data, 2 classes and 2 features
y = (1:100) '>50';
% versicolor=0, virginica=1
b = glmfit(x,y,'binomial');
% logistic regression
p = glmval(b,x,'logit');
% fit probabilities for scores
[X,Y,T,AUC] = perfcurve(species(51:end,:),p,'virginica');
plot(X,Y)
xlabel('False positive rate'); ylabel('True positive rate')
title('ROC for classification by logistic regression')
```



Obtain errors on TPR by vertical averaging

```
[X,Y] = perfcurve(species(51:end,:),p,'virginica',...
    'nboot',1000,'xvals','all');
% plot errors
```

```
errorbar(X,Y(:,1),Y(:,2)-Y(:,1),Y(:,3)-Y(:,1));
```

## References

- [1] T. Fawcett, ROC Graphs: Notes and Practical Considerations for Researchers, 2004.
- [2] M. Zweig and G. Campbell, Receiver-Operating Characteristic (ROC) Plots: A Fundamental Evaluation Tool in Clinical Medicine, *Clin. Chem.* 39/4, 561-577, 1993.
- [3] J. Davis and M. Goadrich, The relationship between precision-recall and ROC curves, in Proceedings of ICML '06, 233-240, 2006.
- [4] C. Moskowitz and M. Pepe, Quantifying and comparing the predictive accuracy of continuous prognostic factors for binary outcomes, *Biostatistics* 5, 113-127, 2004.
- [5] Y. Huang, M. Pepe and Z. Feng, Evaluating the Predictiveness of a Continuous Marker, U. Washington Biostatistics Paper Series, 282, 2006.
- [6] W. Briggs and R. Zaretzki, The Skill Plot: A Graphical Technique for Evaluating Continuous Diagnostic Tests, *Biometrics* 63, 250-261, 2008.
- [7] [http://www2.cs.uregina.ca/~hamilton/courses/831/notes/lift\\_chart/lift\\_chart.htm](http://www2.cs.uregina.ca/~hamilton/courses/831/notes/lift_chart/lift_chart.htm)  
<http://www.dmreview.com/news/5329-1.html>.
- [8] R. Bettinger, Cost-Sensitive Classifier Selection Using the ROC Convex Hull Method, SAS Institute.
- [9] <http://www.stata.com/statalist/archive/2003-02/msg00060.html>

## See Also

`bootci` | `glmfit` | `mnrfit` | `classify` | `NaiveBayes` | `classregtree`

## How To

- “Grouping Variables” on page 2-51
- “Performance Curves” on page 15-41
- “Plotting a Classification Performance Curve” on page 16-143

# perms

---

**Purpose** Enumeration of permutations

**Syntax** `P = perms(v)`

**Description** `P = perms(v)`, where `v` is a row vector of length `n`, creates a matrix whose rows consist of all possible permutations of the `n` elements of `v`. The matrix `P` contains `n!` rows and `n` columns.

`perms` is only practical when `n` is less than about 11 (for `n = 11`, the output takes over 3 gigabytes).

**Examples** `perms([2 4 6])`

`ans =`

```
6 4 2
6 2 4
4 6 2
4 2 6
2 4 6
2 6 4
```

**See Also**

`combnk`

**Purpose** Permute dimensions of categorical array

**Syntax** `B = permute(A,order)`

**Description** `B = permute(A,order)` rearranges the dimensions of the categorical array `A` so that they are in the order specified by the vector `order`. The array produced has the same values as `A` but the order of the subscripts needed to access any particular element are rearranged as specified by `order`. The elements of `order` must be a rearrangement of the numbers from 1 to `n`.

**See Also** `circshift` | `ipermute`

# piecewisedistribution

---

<b>Purpose</b>	Piecewise-defined distributions	
<b>Construction</b>	piecewisedistribution is an abstract class. To construct a piecewisedistribution object, use the subclass constructor, paretotails.	
<b>Methods</b>	boundary	Piecewise distribution boundaries
	cdf	Cumulative distribution function for piecewise distribution
	disp	Display piecewisedistribution object
	display	Display piecewisedistribution object
	icdf	Inverse cumulative distribution function for piecewise distribution
	nsegments	Number of segments
	pdf	Probability density function for piecewise distribution
	random	Random numbers from piecewise distribution
	segment	Segments containing values
<b>Properties</b>	Objects of the piecewisedistribution class have no properties accessible by dot indexing, get methods, or set methods. To obtain information about a piecewisedistribution object, use the appropriate method.	
<b>Copy Semantics</b>	Value. To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.	

**Purpose** Create piecewise distribution object

**Description** `piecewisedistribution` is an abstract class, and you cannot create instances of it directly. You can create `paretotails` objects that are derived from this class.

**See Also** `paretotails`

# prob.PiecewiseLinearDistribution

---

**Superclasses** ParametricTruncatableDistribution

**Purpose** Piecewise linear probability distribution object

**Description** `prob.PiecewiseLinearDistribution` is an object consisting of a model description for a piecewise linear probability distribution. Create a probability distribution object with specified parameters using `makedist`.

**Construction** `pd = makedist('PiecewiseLinear')` creates a piecewise linear probability distribution object using the default parameter values.

`pd = makedist('PiecewiseLinear', 'x', x, 'Fx', Fx)` creates a piecewise linear probability distribution object using the specified values.

## Input Arguments

### **x - Data values**

1 (default) | vector of scalar values

Data values at which the cumulative distribution function (cdf) changes slope, specified as a vector of scalar values.

### **Data Types**

single | double

### **Fx - cdf value**

1 (default) | vector of scalar values

cdf value at each value in `x`, specified as a vector of scalar values. `x` and `Fx` must be the same size. The first value in the vector `Fx` must be 0, and the last element must be 1. `Fx` increases linearly between `x(j)` and `x(j+1)`, for all `j`.

### **Data Types**

single | double



## Properties

**x**

Data values at which the cumulative distribution function (cdf) changes slope, stored as a vector of scalar values.

### Data Types

single | double

**Fx**

cdf value at each value in x, stored as a vector of scalar values.

### Data Types

single | double

**DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### Data Types

char

**IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

### Data Types

logical

**NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

### Data Types

single | double

**ParameterDescription**

# prob.PiecewiseLinearDistribution

---

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object

iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Piecewise Linear Distribution

The piecewise linear distribution is a nonparametric probability distribution created using a piecewise linear representation of the cumulative distribution function (cdf). The options specified for the piecewise linear distribution specify the form of the cdf. The probability density function (pdf) is a step function.

## Examples

### Create a Piecewise Linear Distribution Object Using Default Parameters

Create a piecewise linear distribution object using the default parameter values.

```
pd = makedist('PiecewiseLinear')
```

# prob.PiecewiseLinearDistribution

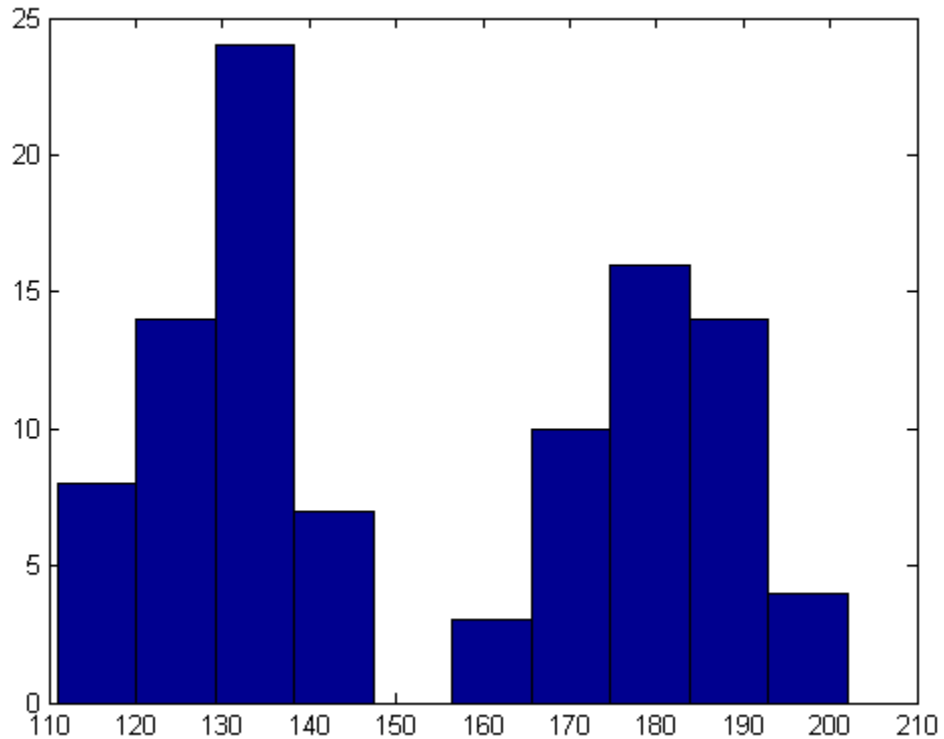
---

```
pd =  
  
    PiecewiseLinearDistribution  
  
F(0) = 0  
F(1) = 1
```

## **Create a Piecewise Linear Distribution Object Using Specified Parameters**

Load the sample data. Visualize the patient weight data using a histogram.

```
load hospital;  
hist(hospital.Weight)
```



The histogram shows that the data has two modes, one for female patients and one for male patients.

Compute the empirical cumulative distribution function (ecdf) for the data.

```
[f,x] = ecdf(hospital.Weight);
```

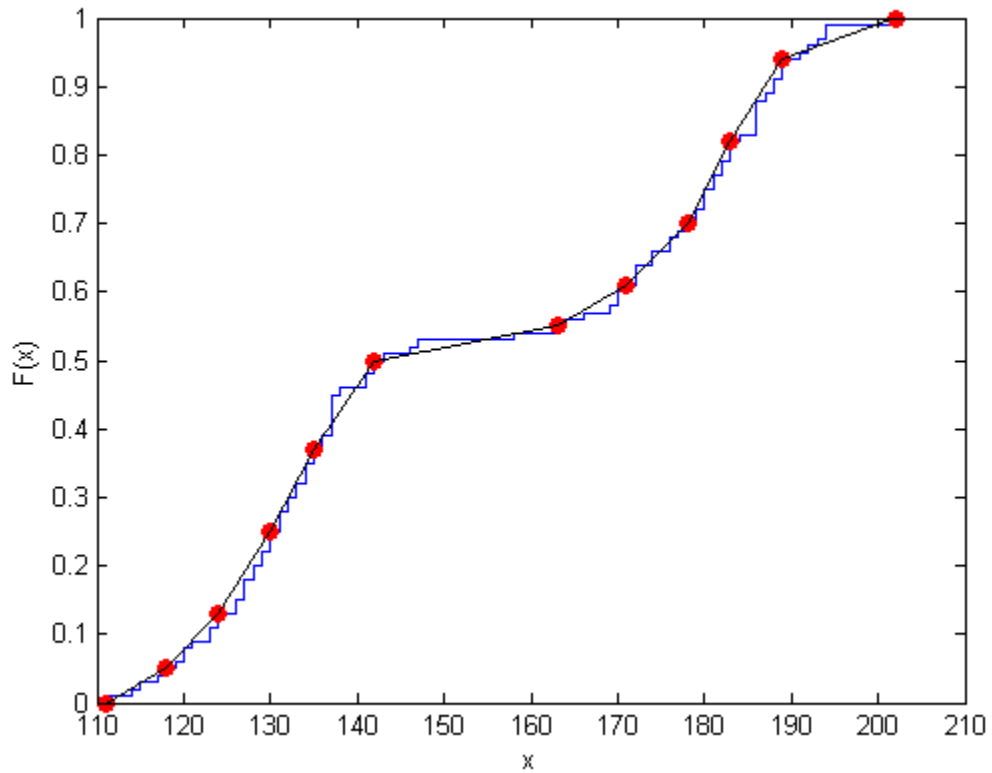
Construct a piecewise linear approximation to the ecdf and plot both functions.

# prob.PiecewiseLinearDistribution

---

```
f = f(1:5:end); % keep a less dense grid of points
x = x(1:5:end);

figure;
ecdf(hospital.Weight)
hold on
plot(x,f,'ro','MarkerFace','r') % overlay grid
plot(x,f,'k') % show interpolation
```



# prob.PiecewiseLinearDistribution

---

Create a piecewise linear probability distribution object using the piecewise approximation of the ecdf.

```
pd = makedist('PiecewiseLinear','x',x,'Fx',f)
```

```
pd =
```

```
    PiecewiseLinearDistribution
```

```
F(111) = 0
F(118) = 0.05
F(124) = 0.13
F(130) = 0.25
F(135) = 0.37
F(142) = 0.5
F(163) = 0.55
F(171) = 0.61
F(178) = 0.7
F(183) = 0.82
F(189) = 0.94
F(202) = 1
```

Generate 100 random numbers from the distribution.

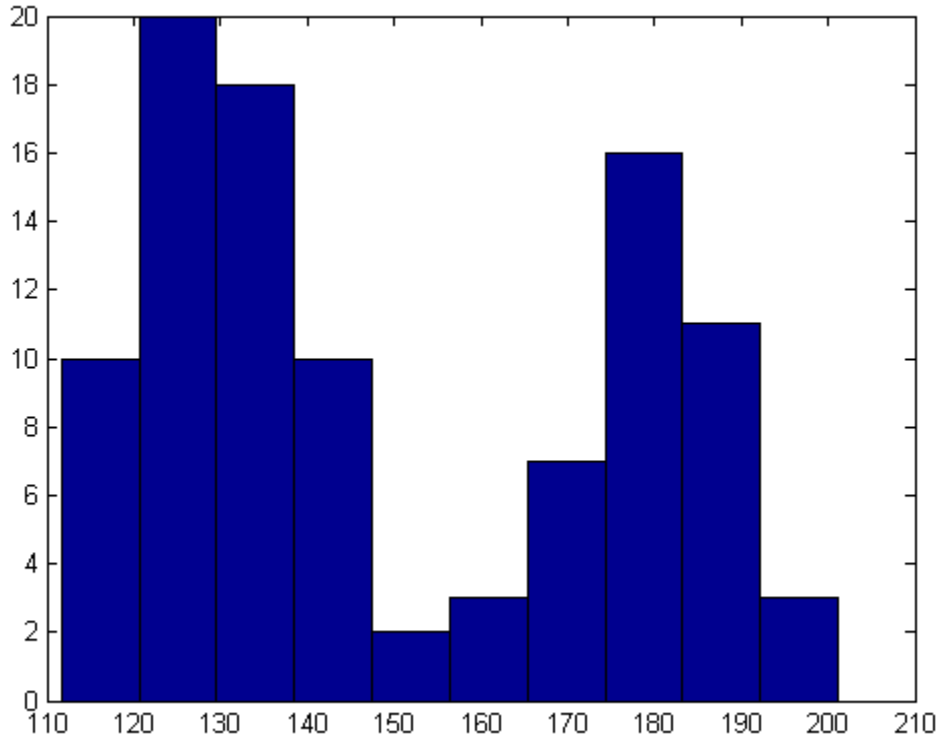
```
rw = random(pd,100,1);
```

Plot the random numbers to visually compare their distribution to the original data.

```
figure;
hist(rw)
```

# prob.PiecewiseLinearDistribution

---



The random numbers generated from the piecewise linear distribution have the same bimodal distribution as the original data.

**See Also** `makedist`

## Concepts

- “Piecewise Linear Distribution” on page B-115
- Class Attributes
- Property Attributes



<b>Purpose</b>	Plot clustering evaluation object criterion values
<b>Syntax</b>	<pre>plot(eva) h = plot(eva)</pre>
<b>Description</b>	<p><code>plot(eva)</code> displays a plot of the criterion values versus the number of clusters, based on the values stored in the clustering evaluation object <code>eva</code>.</p> <p><code>h = plot(eva)</code> returns a handle to the plot line.</p>
<b>Input Arguments</b>	<p><b>eva - Clustering evaluation data</b> clustering evaluation object</p> <p>Clustering evaluation data, specified as a clustering evaluation object. Create a clustering evaluation object using <code>evalclusters</code>.</p>
<b>Output Arguments</b>	<p><b>h - Handle to plot line</b> scalar value</p> <p>Handle to the plot line, returned as a scalar value.</p>
<b>Examples</b>	<p><b>Plot the Clustering Evaluation Criterion Values</b></p> <p>Plot the criterion values versus the number of clusters for each clustering solution stored in a clustering evaluation object.</p> <p>Load the sample data.</p> <pre>load fisheriris;</pre> <p>The data contains length and width measurements from the sepals and petals of three species of iris flowers.</p> <p>Create a clustering evaluation object. Cluster the data using <code>kmeans</code>, and evaluate the optimal number of clusters using the Calinski-Harabasz criterion.</p> <pre>rng('default'); % For reproducibility</pre>

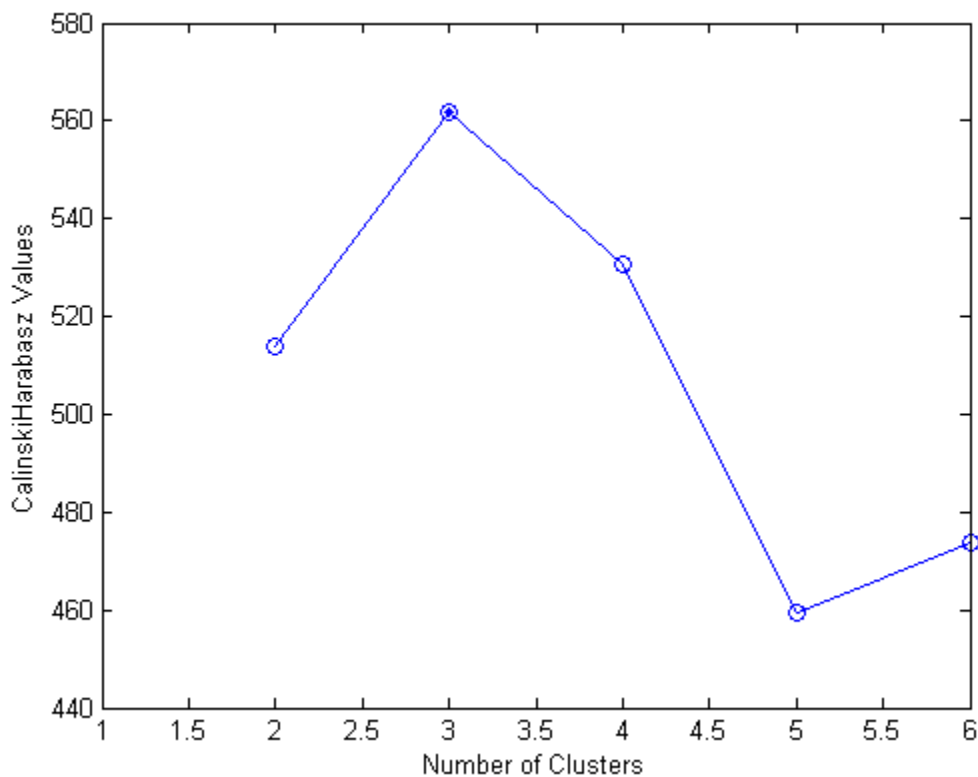
# clustering.evaluation.ClusterCriterion.plot

---

```
eva = evalclusters(meas,'kmeans','CalinskiHarabasz','KList',[1:6]);
```

Plot the Calinski-Harabasz criterion values for each number of clusters tested.

```
figure;  
plot(eva);
```



The plot shows that the highest Calinski-Harabasz value occurs at three clusters, suggesting that the optimal number of clusters is three.

**See Also** `evalclusters`

# LinearModel.plot

---

**Purpose** Scatter plot or added variable plot of linear model

**Syntax**  
`plot mdl)`  
`h = plot(mdl)`

**Description** `plot(mdl)` creates a plot of the fitted linear model. The plot type depends on the number of predictor variables.

- If there is just one predictor variable, `plot` creates a scatter plot of the data along with a fitted curve and confidence bounds.
- If there are multiple predictor variables, `plot` creates an added variable plot.
- If there are no predictors, `plot` creates a histogram of the residuals.

`h = plot(mdl)` returns handles to the lines in the plot.

**Input Arguments** **mdl**  
Linear model, as constructed by `fitlm` or `stepwiselm`.

**Output Arguments** **h**  
Vector of handles to lines or patches in the plot.

**Definitions** **Added Variable Plot, Adjusted Response**

An added variable plot illustrates the incremental effect on the response of specified terms by removing the effects of all other terms. The slope of the fitted line is the coefficient of the linear combination of the specified terms projected onto the best-fitting direction. The adjusted response includes the constant (intercept) terms, and averages out all other terms.

**Examples** **Create an Added Variable Plot**

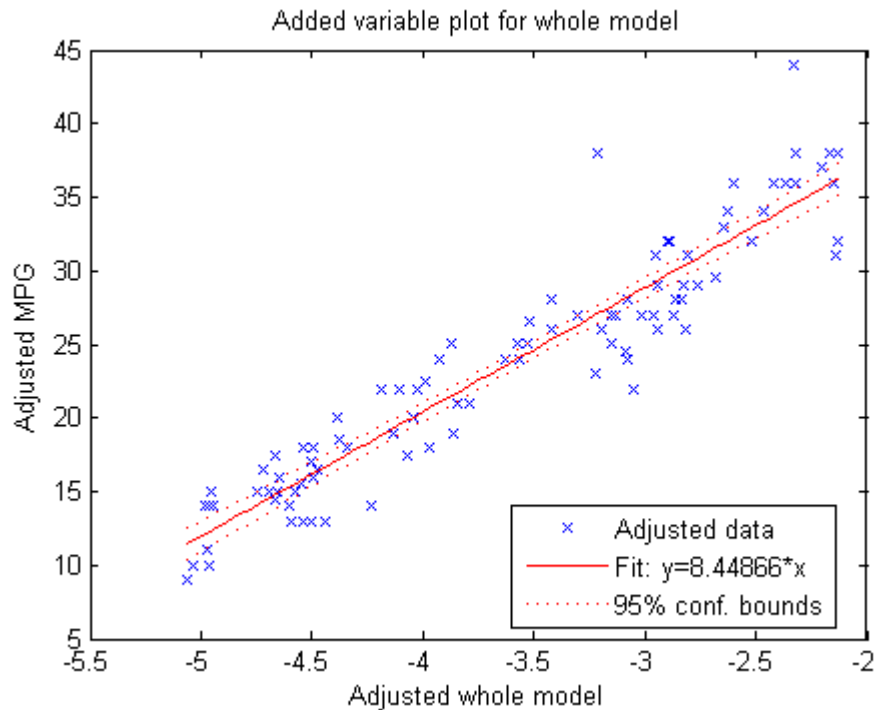
Create a model of car mileage as a function of weight and model year. Then create a plot to see the significance of the model.

Create a linear model of mileage from the carsmall data.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Create an added variable plot.

```
plot(mdl)
```



The plot illustrates that the model is significant—a horizontal line does not fit between the confidence bounds.

# LinearModel.plot

---

**Alternatives**    Use `plotAdded` to select particular predictors for an added variable plot.

**See Also**        `LinearModel` | `plotAdded`

**How To**            • “Linear Regression” on page 9-12

## Purpose

Added variable plot or leverage plot for linear model

## Syntax

```
plotAdded mdl
plotAdded mdl,coef
h = plotAdded mdl,...
h = plotAdded mdl,coef,Name,Value
```

## Description

`plotAdded(mdl)` produces a generalized added variable plot for all terms in `mdl` except the constant term.

`plotAdded(mdl,coef)` produces an added variable plot for the `coef` terms in `mdl`, after adjusting for all other terms.

`h = plotAdded(mdl,...)` returns handles to the lines in the plot.

`h = plotAdded(mdl,coef,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the *x* and *y* values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **coef**

Coefficients in `mdl`. Represent as:

- String giving a single coefficient name
- Vector of coefficient numbers in the `mdl.CoefficientNames` property

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can

# LinearModel.plotAdded

---

specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

## **'Color'**

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

## **'LineStyle'**

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

## **'LineWidth'**

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

## **'MarkerEdgeColor'**

Color of the marker or edge color for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

## **'MarkerFaceColor'**

Color of the marker face for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

## **'MarkerSize'**

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.



## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Definitions

### Added Variable Plot, Adjusted Response

An added variable plot illustrates the incremental effect on the response of specified terms by removing the effects of all other terms. The slope of the fitted line is the coefficient of the linear combination of the specified terms projected onto the best-fitting direction. The adjusted response includes the constant (intercept) terms, and averages out all other terms.

## Examples

### Create an Added Variable Plot

Create a model of car mileage as a function of weight and model year. Then create a plot to see the significance of the model.

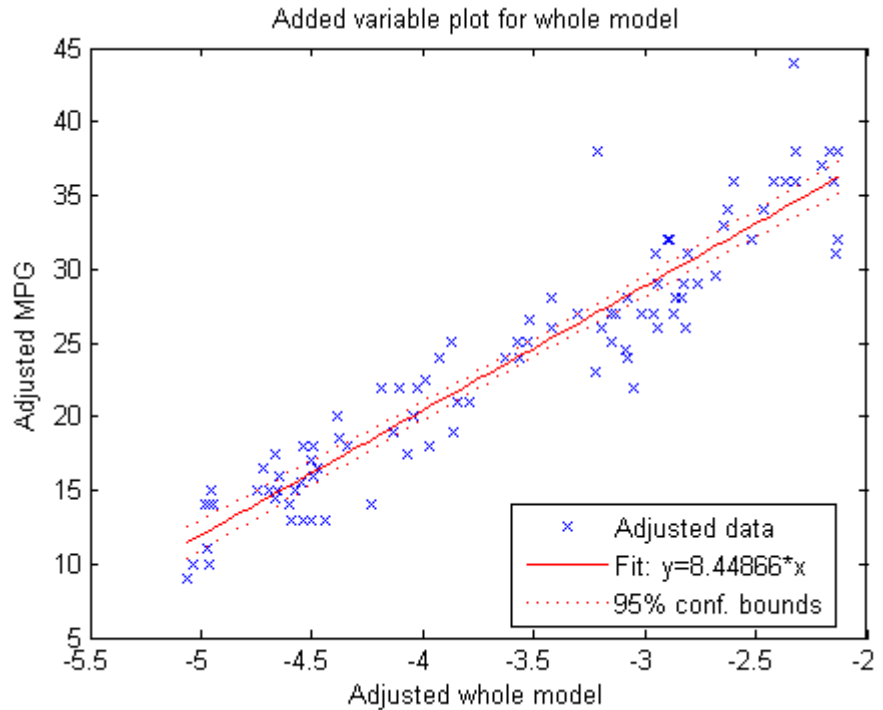
Create a linear model of mileage from the `carsmall` data.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Create an added variable plot.

```
plotAdded(mdl)
```

# LinearModel.plotAdded



The plot illustrates that the model is significant—a horizontal line does not fit between the confidence bounds.

## Create an Added Variable Plot for Particular Variables

Create a model of car mileage as a function of weight and model year. Then create a plot to see the effect of the weight terms (Weight and Weight<sup>2</sup>).

Create a linear model of mileage from the carsmall data.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Find the terms in the model corresponding to the Weight and Weight<sup>2</sup>.

```
mdl.CoefficientNames
```

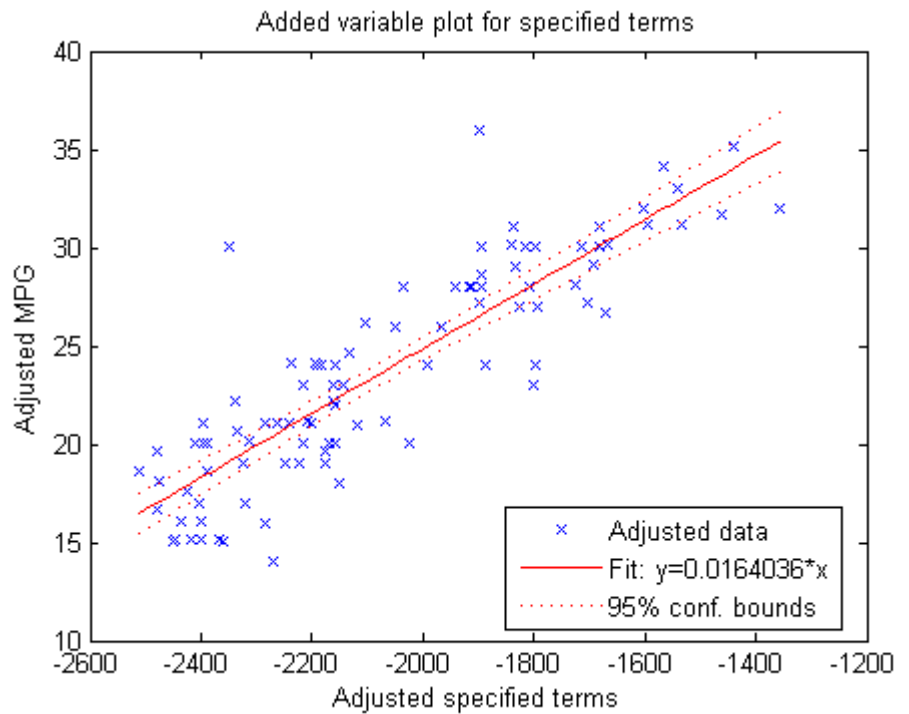
```
ans =
```

```
'(Intercept)' 'Weight' 'Year_76' 'Year_82' 'Weight^2'
```

The weight terms are 2 and 5.

Create an added variable plot with the weight terms.

```
coef = [2 5];  
plotAdded(mdl,coef)
```



# LinearModel.plotAdded

---

The plot illustrates that the weight terms are significant—a horizontal line does not fit between the confidence bounds.

## See Also

`LinearModel | plot`

## Tutorials

- “Plots to Understand Terms Effects” on page 9-35

## How To

- “Linear Regression” on page 9-12

# LinearModel.plotAdjustedResponse

---

## Purpose

Adjusted response plot for linear regression model

## Syntax

```
plotAdjustedResponse mdl, var)
h = plotAdjustedResponse mdl, var)
h = plotAdjustedResponse mdl, var, Name, Value)
```

## Description

`plotAdjustedResponse(mdl, var)` gives an adjusted response plot for the variable `var` in the `mdl` regression model.

`h = plotAdjustedResponse(mdl, var)` returns handles to the lines in the plot.

`h = plotAdjustedResponse(mdl, var, Name, Value)` plots with additional options specified by one or more `Name, Value` pair arguments.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **var**

Variable name, or scalar index of variable in `mdl.CoefficientNames`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

# LinearModel.plotAdjustedResponse

---

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

**'Color'**

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

**'LineStyle'**

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

**'LineWidth'**

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

**'MarkerEdgeColor'**

Color of the marker or edge color for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

**'MarkerFaceColor'**

Color of the marker face for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

**'MarkerSize'**

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

**Output Arguments**

**h**

Vector of handles to lines or patches in the plot.

## Definitions

### Adjusted Response Plot

The adjusted response plot shows the fitted response as a function of var, with the other predictors averaged out by averaging the fitted values over the data used in the fit. Adjusted data points are computed by adding the residual to the adjusted fitted value for each observation.

## Examples

### Plot Adjusted Responses

Plot the adjusted responses of a fitted linear model.

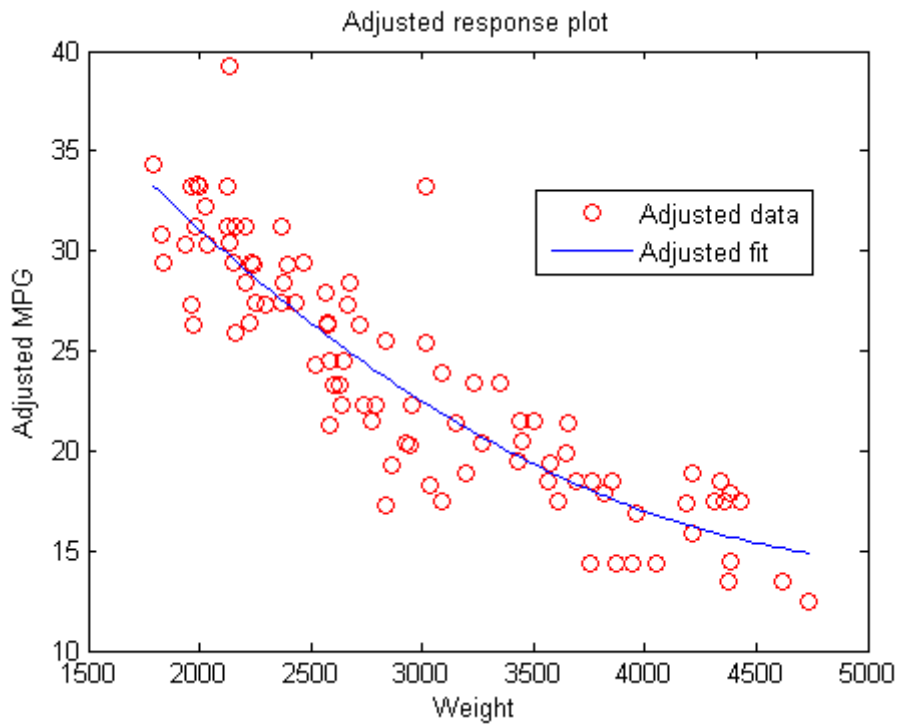
Load the carsmall data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Plot the effect of 'Weight' averaged over Year values.

```
plotAdjustedResponse(mdl,'Weight')
```

# LinearModel.plotAdjustedResponse

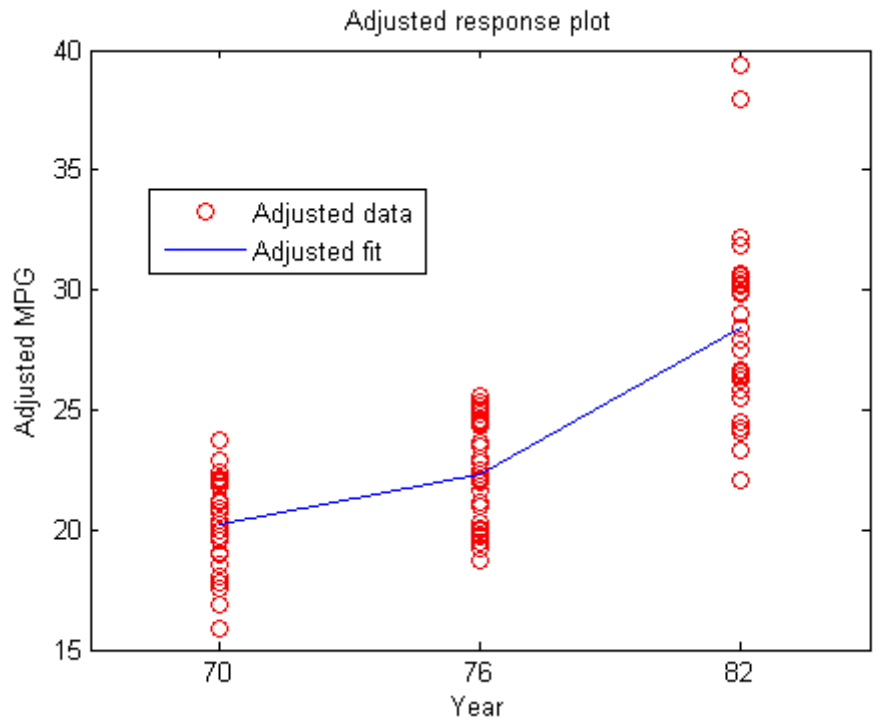


Plot the effect of Year averaged over 'Weight' values. Include the h output.

```
h = plotAdjustedResponse(md1, 'Year');
```



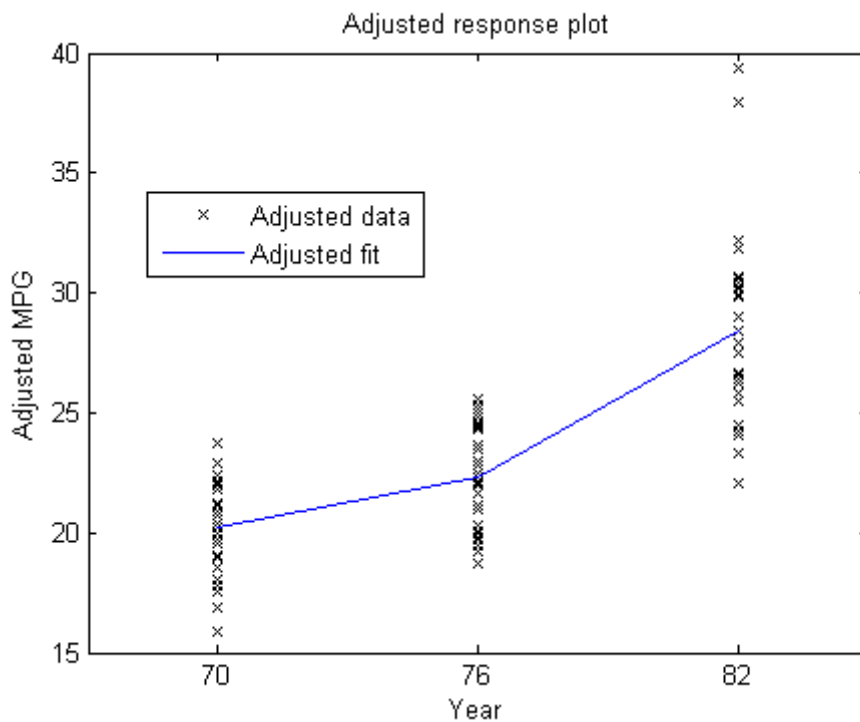
# LinearModel.plotAdjustedResponse



Change the adjusted data to black x instead of red o.

```
set(h(1), 'Marker', 'x', 'Color', 'k')
```

# LinearModel.plotAdjustedResponse



## See Also

[LinearModel](#) | [plotAdded](#) | [plotEffects](#) | [plotInteraction](#)

## How To

- “Linear Regression” on page 9-12

# GeneralizedLinearModel.plotDiagnostics

---

**Purpose** Plot diagnostics of generalized linear regression model

**Syntax**

```
plotDiagnostics mdl
plotDiagnostics mdl,plottype
h = plotDiagnostics(...)
h = plotDiagnostics mdl,plottype,Name,Value
```

**Description**

`plotDiagnostics(mdl)` plots diagnostics from the `mdl` linear model using leverage as the plot type.

`plotDiagnostics(mdl,plottype)` plots diagnostics from the `mdl` generalized linear model in a plot of type `plottype`.

`h = plotDiagnostics(...)` returns handles to the lines in the plot.

`h = plotDiagnostics(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

### **plottype**

String specifying the type of plot:

'contour'	Residual vs. leverage with overlaid Cook's contours
'cookd'	Cook's distance
'leverage'	Leverage (diagonal of Hat matrix)

**Default:** 'leverage'

# GeneralizedLinearModel.plotDiagnostics

---

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### **'Color'**

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

### **'LineStyle'**

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

### **'LineWidth'**

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### **'MarkerEdgeColor'**

Color of the marker or edge color for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

### **'MarkerFaceColor'**

Color of the marker face for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

### **'MarkerSize'**

# GeneralizedLinearModel.plotDiagnostics

---

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Definitions

### Hat Matrix

The *hat matrix*  $H$  is defined in terms of the data matrix  $X$  and a diagonal weight matrix  $W$ :

$$H = X(X^T W X)^{-1} X^T W^T.$$

$W$  has diagonal elements  $w_i$ :

$$w_i = \frac{g'(\mu_i)}{\sqrt{V(\mu_i)}},$$

where

- $g$  is the link function mapping  $y_i$  to  $x_i b$ .
- $g'$  is the derivative of the link function  $g$ .
- $V$  is the variance function.
- $\mu_i$  is the  $i$ th mean.

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

# GeneralizedLinearModel.plotDiagnostics

---

## Leverage

The *leverage* of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = w_i \frac{e_i^2}{p\hat{\phi}} \frac{h_{ii}}{(1-h_{ii})^2},$$

where

- $\hat{\phi}$  is the dispersion parameter (estimated or theoretical).
- $e_i$  is the linear predictor residual,  $g(y_i) - x_i\hat{\beta}$ , where
  - $g$  is the link function.
  - $y_i$  is the observed response.
  - $x_i$  is the observation.
  - $\hat{\beta}$  is the estimated coefficient vector.
- $p$  is the number of coefficients in the regression model.
- $h_{ii}$  is the  $i$ th diagonal element of the Hat Matrix  $H$ .

## Examples

### Diagnostic Plots for Generalized Linear Models

Create leverage and Cook's distance plots of a fitted generalized linear model.

Generate artificial data for the model, Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

# GeneralizedLinearModel.plotDiagnostics

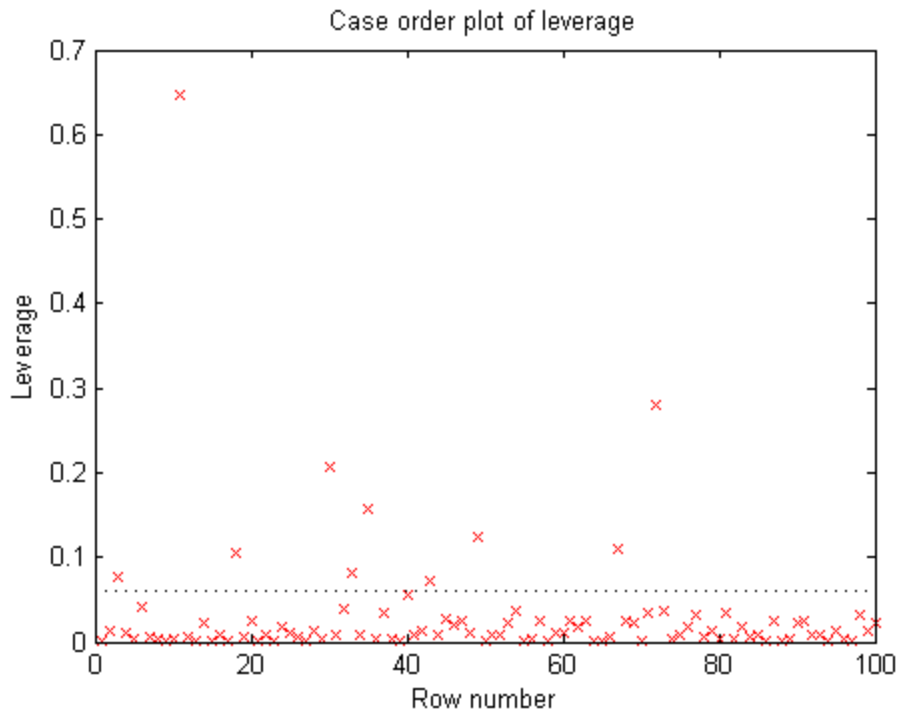
```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,...
    'y ~ x1 + x2','distr','poisson');
```

Create a leverage plot.

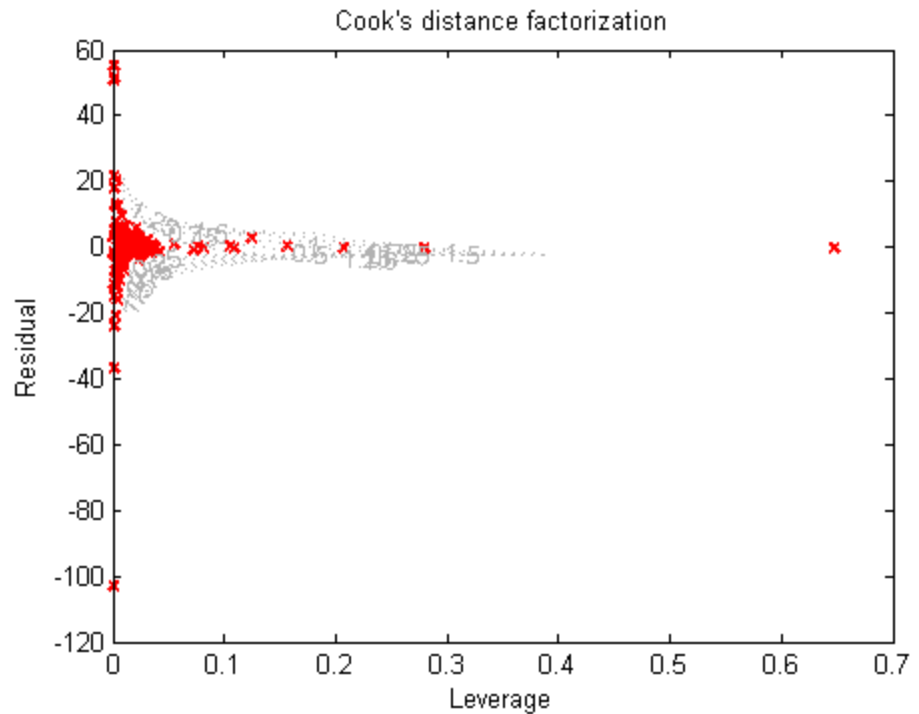
```
plotDiagnostics(mdl)
```



# GeneralizedLinearModel.plotDiagnostics

Create a contour plot with Cook's distance.

```
plotDiagnostics(md1, 'contour')
```



## References

[1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.

**See Also** [GeneralizedLinearModel |](#)

## Related Examples

- “Diagnostic Plots” on page 10-26

## Concepts

- “Generalized Linear Models” on page 10-12



**Purpose** Plot diagnostics of linear regression model

**Syntax**

```
plotDiagnostics mdl
plotDiagnostics mdl,plottype
h = plotDiagnostics(...)
h = plotDiagnostics mdl,plottype,Name,Value
```

**Description**

`plotDiagnostics(mdl)` plots diagnostics from the `mdl` linear model using scaled delete-1 fitted values.

`plotDiagnostics(mdl,plottype)` plots diagnostics in a plot of type `plottype`.

`h = plotDiagnostics(...)` returns handles to the lines in the plot.

`h = plotDiagnostics(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For many plots, the Data Cursor tool in the figure window displays the *x* and *y* values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **plottype**

String specifying the type of plot:

'contour'	Residual vs. leverage with overlaid Cook's contours
'cookd'	Cook's distance
'covratio'	Delete-1 ratio of determinant of covariance
'dfbetas'	Scaled delete-1 coefficient estimates

# LinearModel.plotDiagnostics

---

'dffits'	Scaled delete-1 fitted values
'leverage'	Leverage
's2_i'	Delete-1 variance estimate

Delete-1 means compute a new model without the current observation. If the delete-1 calculation differs significantly from the model using all observations, then the observation is influential.

**Default:** 'leverage'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### 'Color'

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

### 'LineStyle'

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

### 'LineWidth'

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

Default: 0.5

## 'MarkerEdgeColor'

Color of the marker or edge color for filled markers, a string or ColorSpec specification. For details, see `linespec`.

## 'MarkerFaceColor'

Color of the marker face for filled markers, a string or ColorSpec specification. For details, see `linespec`.

## 'MarkerSize'

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Definitions

### Hat Matrix

The *hat matrix*  $H$  is defined in terms of the data matrix  $X$ :

$$H = X(X^T X)^{-1} X^T.$$

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

### Leverage

The *leverage* of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the

# LinearModel.plotDiagnostics

---

number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

## Cook's Distance

Cook's distance is the scaled change in fitted values. Each element in `CooksDistance` is the normalized change in the vector of coefficients due to the deletion of an observation. The Cook's distance,  $D_i$ , of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $\text{MSE}$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p \text{MSE}} \left( \frac{h_{ii}}{(1 - h_{ii})^2} \right),$$

where  $r_i$  is the  $i$ th residual, and  $h_{ii}$  is the  $i$ th leverage value.

`CooksDistance` is an  $n$ -by-1 column vector in the `Diagnostics` dataset array of the `LinearModel` object.

## Examples

### Leverage Plot of Linear Model

Plot the leverage values of observations in a fitted model.

Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Plot the leverage values.

```
plotDiagnostics(mdl)
```

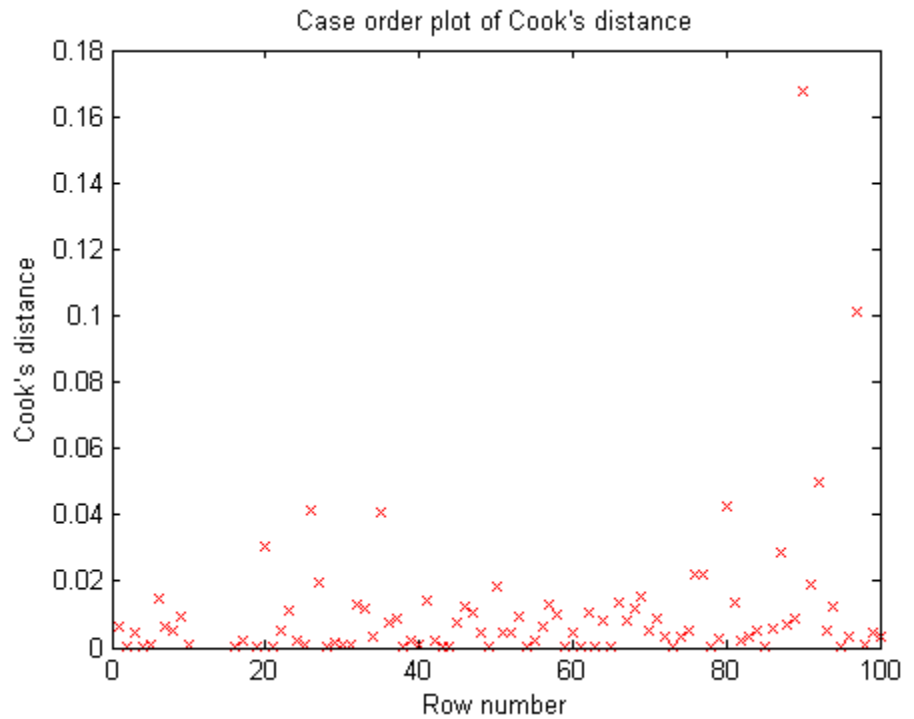
# LinearModel.plotDiagnostics

---



Plot the Cook's distance.

```
plotDiagnostics(md1, 'cookd')
```



The two diagnostic plots give different results.

## References

[1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.

## Alternatives

The `mdl.Diagnostics` property contains the information that `plotDiagnostics` uses to create plots.

## See Also

`LinearModel`

## Tutorials

- “Diagnostic Plots” on page 9-24

# LinearModel.plotDiagnostics

---

## How To

- “Linear Regression” on page 9-12



## Purpose

Plot diagnostics of nonlinear regression model

## Syntax

```
plotDiagnostics mdl
plotDiagnostics mdl,plottype
h = plotDiagnostics(...)
h = plotDiagnostics mdl,plottype,Name,Value
```

## Description

`plotDiagnostics(mdl)` plots diagnostics from the `mdl` linear model using leverage as the plot type.

`plotDiagnostics(mdl,plottype)` plots diagnostics in a plot of type `plottype`.

`h = plotDiagnostics(...)` returns handles to the lines in the plot.

`h = plotDiagnostics(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **plottype**

String specifying the type of plot:

'contour'	Residual vs. leverage with overlaid Cook's contours
'cookd'	Cook's distance
'leverage'	Leverage (diagonal of Hat matrix)

**Default:** 'leverage'

# NonLinearModel.plotDiagnostics

---

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### **'Color'**

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

### **'LineStyle'**

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

### **'LineWidth'**

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### **'MarkerEdgeColor'**

Color of the marker or edge color for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

### **'MarkerFaceColor'**

Color of the marker face for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

### **'MarkerSize'**

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Definitions

### Hat Matrix

The *hat matrix*  $H$  is defined in terms of the data matrix  $X$  and the Jacobian matrix  $J$ :

$$J_{i,j} = \left. \frac{\partial f}{\partial \beta_j} \right|_{x_i, \beta}$$

Here  $f$  is the nonlinear model function, and  $\beta$  is the vector of model coefficients.

The Hat Matrix  $H$  is

$$H = J(J^T J)^{-1} J^T.$$

The diagonal elements  $H_{ii}$  satisfy

$$0 \leq h_{ii} \leq 1$$

$$\sum_{i=1}^n h_{ii} = p,$$

where  $n$  is the number of observations (rows of  $X$ ), and  $p$  is the number of coefficients in the regression model.

### Leverage

The *leverage* of observation  $i$  is the value of the  $i$ th diagonal term,  $h_{ii}$ , of the hat matrix  $H$ . Because the sum of the leverage values is  $p$  (the number of coefficients in the regression model), an observation  $i$  can be considered to be an outlier if its leverage substantially exceeds  $p/n$ , where  $n$  is the number of observations.

# NonLinearModel.plotDiagnostics

---

## Cook's Distance

The Cook's distance  $D_i$  of observation  $i$  is

$$D_i = \frac{\sum_{j=1}^n (\hat{y}_j - \hat{y}_{j(i)})^2}{p \text{MSE}},$$

where

- $\hat{y}_j$  is the  $j$ th fitted response value.
- $\hat{y}_{j(i)}$  is the  $j$ th fitted response value, where the fit does not include observation  $i$ .
- $\text{MSE}$  is the mean squared error.
- $p$  is the number of coefficients in the regression model.

Cook's distance is algebraically equivalent to the following expression:

$$D_i = \frac{r_i^2}{p \text{MSE}} \left( \frac{h_{ii}}{(1 - h_{ii})^2} \right),$$

where  $e_i$  is the  $i$ th residual.

## Examples

### Nonlinear Model Leverage Plot

Create a leverage plot of a fitted nonlinear model, and find the points with high leverage.

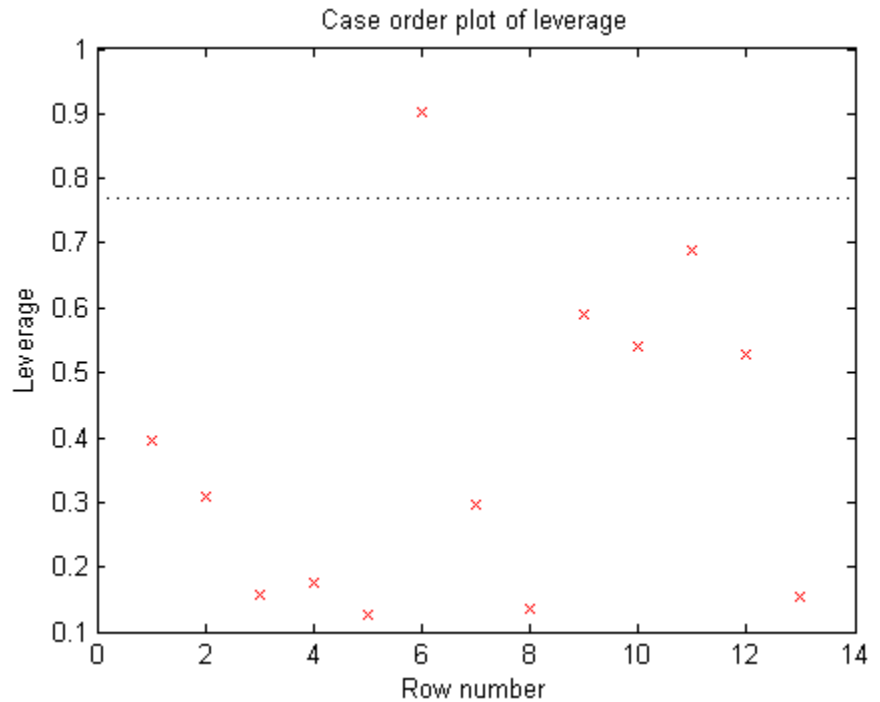
Load the reaction data and fit a model of the reaction rate as a function of reactants.

```
load reaction
mdl = fitnlm(reactants,rate,@hougen,[1 .05 .02 .1 2]);
```

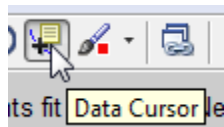
# NonLinearModel.plotDiagnostics

Create a leverage plot of the fitted model.

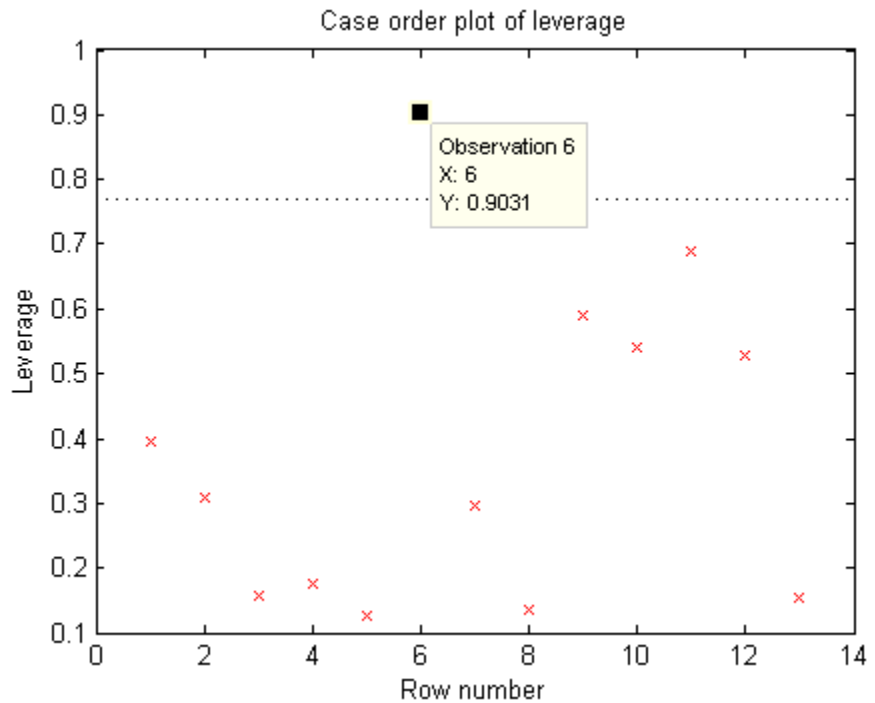
```
plotDiagnostics(md1)
```



To examine the observation with high leverage, activate the Data Cursor and click the observation.



# NonLinearModel.plotDiagnostics



Alternatively, find the high-leverage observation at the command line.

```
find(md1.Diagnostics.Leverage > 0.8)
```

```
ans =
```

```
6
```

## References

[1] Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, Fourth Edition. Irwin, Chicago, 1996.

## See Also

NonLinearModel | plotResiduals |

## **Related Examples**

- “Examine Quality and Adjust the Fitted Model” on page 11-8
- “Nonlinear Regression Workflow” on page 11-16

## **Concepts**

- “Nonlinear Regression” on page 11-2

# LinearModel.plotEffects

---

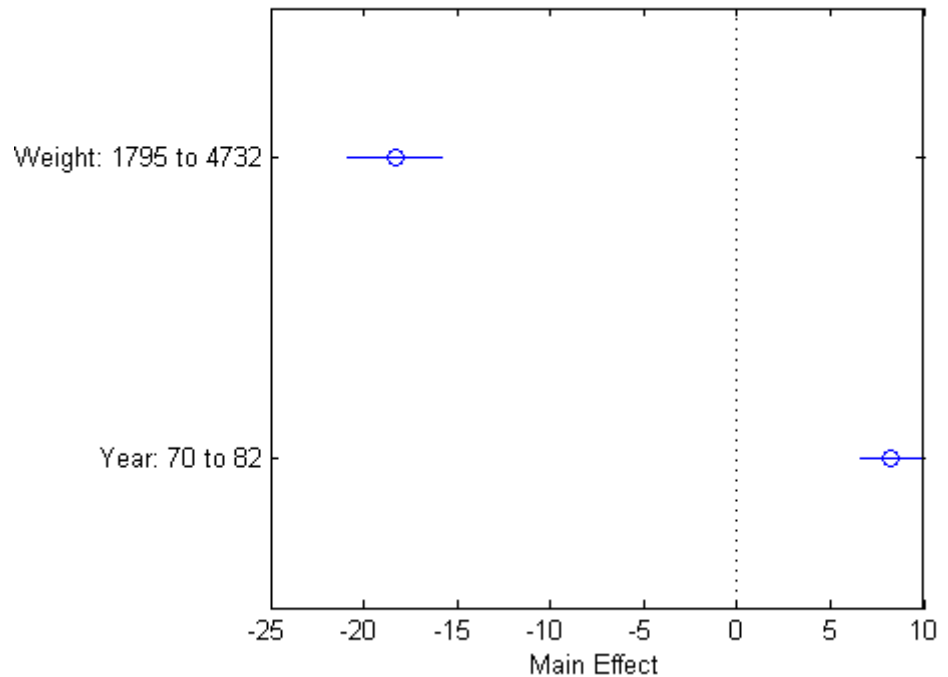
<b>Purpose</b>	Plot main effects of each predictor in linear regression model
<b>Syntax</b>	<pre>plotEffects mdl h = plotEffects mdl</pre>
<b>Description</b>	<p><code>plotEffects mdl</code> produces an effects plot for the predictors in the <code>mdl</code> regression model. The plot shows the estimated effect on the response from changing each predictor value, averaging out the effects of the other predictors. <code>plotEffects</code> chooses values to produce a relatively large effect on the response.</p> <p><code>h = plotEffects mdl</code> returns handles to the lines in the plot.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• For many plots, the Data Cursor tool in the figure window displays the <math>x</math> and <math>y</math> values for any data point, along with the observation name or number.</li></ul>
<b>Input Arguments</b>	<p><b>mdl</b></p> <p>Linear model, as constructed by <code>fitlm</code> or <code>stepwiselm</code>.</p>
<b>Output Arguments</b>	<p><b>h</b></p> <p>Vector of handles to lines or patches in plot. <code>h(1)</code> is a handle to the circles that represent the main effect estimates. <code>h(j+1)</code> is a handle to the line that defines the confidence interval for the effect of predictor <math>j</math>.</p>
<b>Examples</b>	<p><b>Effects Plot</b></p> <p>Plot the effects of two predictors in a fitted linear model.</p> <p>Load the <code>carsmall</code> data and fit a linear model of the mileage as a function of model year, weight, and weight squared.</p> <pre>load carsmall ds = dataset(MPG,Weight); ds.Year = ordinal(Model_Year);</pre>



```
mdl = fitlm(ds, 'MPG ~ Year + Weight^2');
```

Create an effects plot.

```
plotEffects(mdl)
```



The width of each horizontal line in the figure shows a confidence interval for the effect on the response of the listed change in each predictor. The estimated effect of changing Year from 70 to 82 is an increase of about 8, and is between 6 and 10 with 95% confidence.

## Alternatives

Use `plotInteraction` for an effects plot of the interactions of two specified variables.

# LinearModel.plotEffects

---

## **See Also**

LinearModel | plotAdjustedResponse | plotInteraction

## **Tutorials**

- “Plots to Understand Predictor Effects” on page 9-30

## **How To**

- “Linear Regression” on page 9-12

**Purpose** Plot interaction effects of two predictors in linear regression model

**Syntax**

```
plotInteraction(md1,var1,var2)
plotInteraction(md1,var1,var2,ptype)
h = plotInteraction(...)
```

**Description** `plotInteraction(md1,var1,var2)` creates a plot of the interaction effects of the predictors `var1` and `var2` in `md1`. The plot shows the estimated effect on the response from changing each predictor value, averaging out the effects of the other predictors. The plot also shows the estimated effect with the other predictor fixed at certain values. `plotInteraction` chooses values to produce a relatively large effect on the response. The plot lets you examine whether the effect of one predictor depends on the value of the other predictor.

`plotInteraction(md1,var1,var2,ptype)` returns a plot of the type specified in `ptype`.

`h = plotInteraction(...)` returns handles to the lines in the plot.

**Tips**

- For many plots, the Data Cursor tool in the figure window displays the *x* and *y* values for any data point, along with the observation name or number.

## Input Arguments

### **md1**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **var1**

String naming the variable for plot. `plotInteraction` chooses values of `var1` to create relatively large changes in the response. When you set `ptype = 'predictions'`, the plot shows curves as a function of `var2` with various fixed values of `var1`.

### **var2**

String naming the variable for plot. `plotInteraction` chooses values of `var2` to create relatively large changes in the response.

# LinearModel.plotInteraction

---

When you set `ptype = 'predictions'`, the plot shows curves as a function of `var2` various fixed values of `var1`.

## **ptype**

String naming the plot type.

- 'effects' — The plot shows each effect as a circle, with a horizontal bar showing the confidence interval for the estimated effect. `plotInteraction` computes the effect values from the adjusted response curve, as shown by the `plotAdjustedResponse` function.
- 'predictions' — The plot shows the adjusted response curve as a function of `var2`, with `var1` fixed at certain values.

**Default:** 'effects'

## **Output Arguments**

### **h**

Vector of handles to lines or patches in the plot.

## **Examples**

### **Interaction Plot**

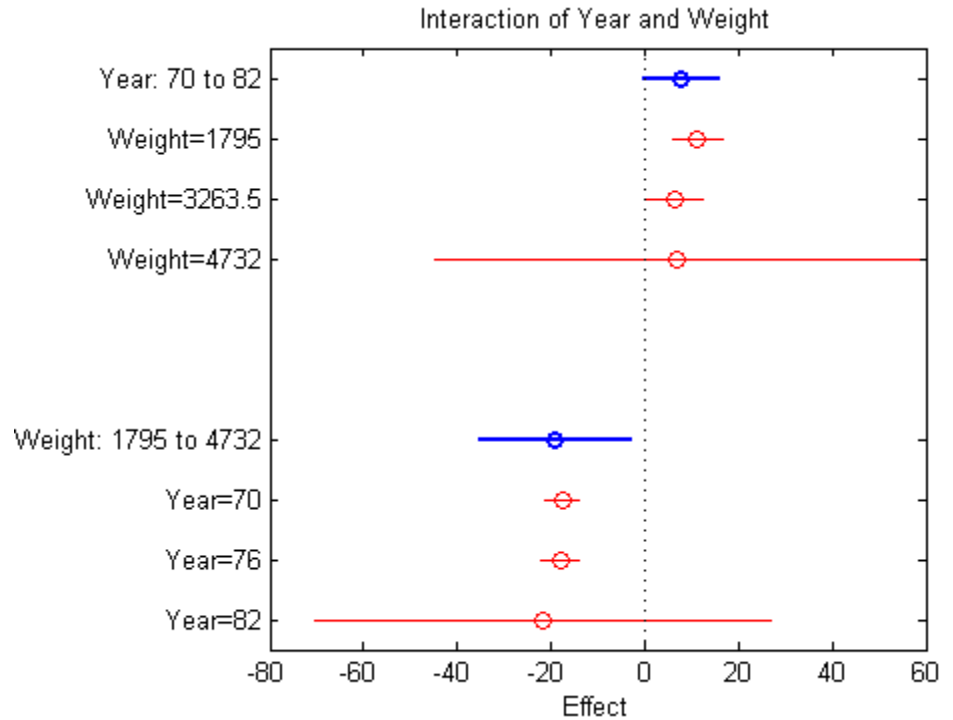
Create a model of car mileage as a function of weight and model year. Then create a plot to see if the predictors have interactions.

Create a linear model of mileage from the `carsmall` data.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
var1 = 'Year';
var2 = 'Weight';
mdl = fitlm(ds,'MPG ~ Year * Weight^2');
```

Create an interaction plot.

```
plotInteraction(mdl,var1,var2)
```



The plot might show an interaction, because the groups of points are not perfectly vertical. But the error bars seem large enough that a vertical line could pass within all of the confidence intervals for each group, possibly indicating no interaction.

## Prediction Curve Interaction Plot

Create a model of car mileage as a function of weight and model year. Then create an interaction curve plot to see if the predictors have interactions.

Create a linear model of mileage from the carsmall data.

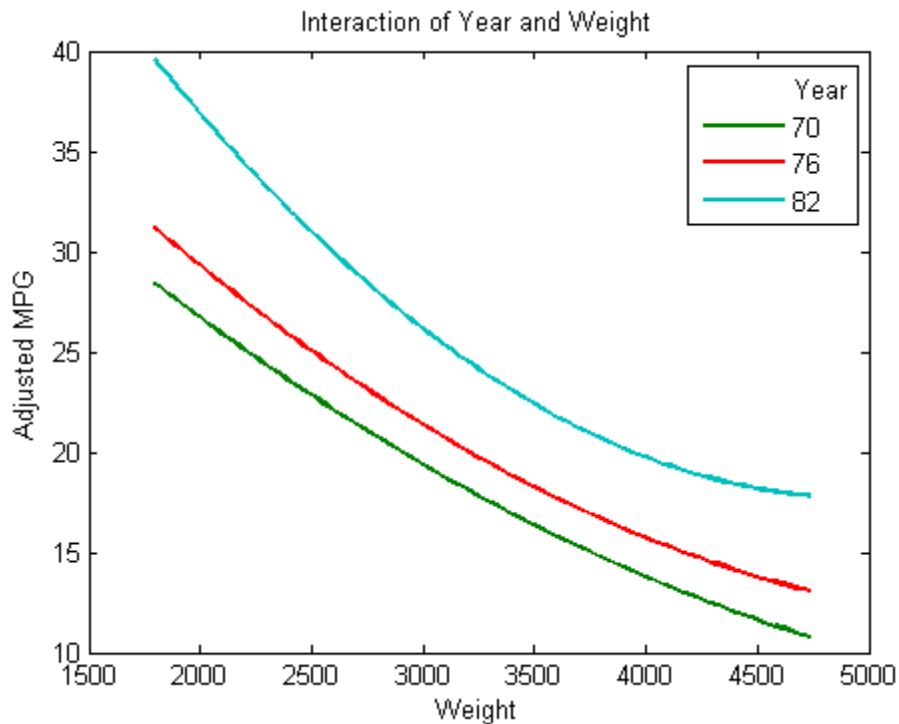
```
load carsmall
```

# LinearModel.plotInteraction

```
ds = dataset(MPG,Weight);  
ds.Year = ordinal(Model_Year);  
var1 = 'Year';  
var2 = 'Weight';  
mdl = fitlm(ds,'MPG ~ Year * Weight^2');
```

Create an interaction plot with type 'predictions'.

```
plotInteraction(mdl,var1,var2,'predictions')
```



The curves are not parallel. This indicates interactions between the predictors. The effect is subtle enough not to definitively indicate a interaction.

**Alternatives** Use `plotEffects` for an effects plot showing separate effects for all predictors.

**See Also** `LinearModel` | `plotAdjustedResponse` | `plotEffects`

**Tutorials** • “Plots to Understand Predictor Effects” on page 9-30

**How To** • “Linear Regression” on page 9-12

# GeneralizedLinearModel.plotResiduals

---

**Purpose** Plot residuals of generalized linear regression model

**Syntax**

```
plotResiduals mdl
plotResiduals mdl,plottype
h = plotResiduals(...)
h = plotResiduals mdl,plottype,Name,Value)
```

**Description**

`plotResiduals(mdl)` gives a histogram plot of the residuals of the `mdl` nonlinear model.

`plotResiduals(mdl,plottype)` plots residuals in a plot of type `plottype`.

`h = plotResiduals(...)` returns handles to the lines in the plot.

`h = plotResiduals(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

### **plottype**

String specifying the type of plot:

'caseorder'	Residuals vs. case (row) order
'fitted'	Residuals vs. fitted values
'histogram'	Histogram
'lagged'	Residuals vs. lagged residual ( $r(t)$ vs. $r(t-1)$ )



# GeneralizedLinearModel.plotResiduals

---

'probability'	Normal probability plot
'symmetry'	Symmetry plot

**Default:** 'histogram'

## Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### 'Color'

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

### 'LineStyle'

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

### 'LineWidth'

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### 'MarkerEdgeColor'

Color of the marker or edge color for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

# GeneralizedLinearModel.plotResiduals

---

## 'MarkerFaceColor'

Color of the marker face for filled markers, a string or ColorSpec specification. For details, see `linespec`.

## 'MarkerSize'

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## 'ResidualType'

String giving type of residual used in the plot.

'Raw'	Observed minus fitted values
'LinearPredictor'	Residuals on the linear predictor scale, equal to the adjusted response value minus the fitted linear combination of the predictors
'Pearson'	Raw residuals divided by RMSE
'Anscombe'	Residuals defined on transformed data with the transformation chosen to remove skewness
'Deviance'	Residuals based on the contribution of each observation to the deviance

**Default:** 'Raw'

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Definitions

### Deviance

*Deviance* is twice the log likelihood of the model. Because this overall log likelihood is a sum of log likelihoods for each observation, the residual plot of deviance type shows the log likelihood per observation.

## Examples

### Residual Plots for Generalized Linear Models

Create residual plots of a fitted generalized linear model.

Generate artificial data for the model, Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

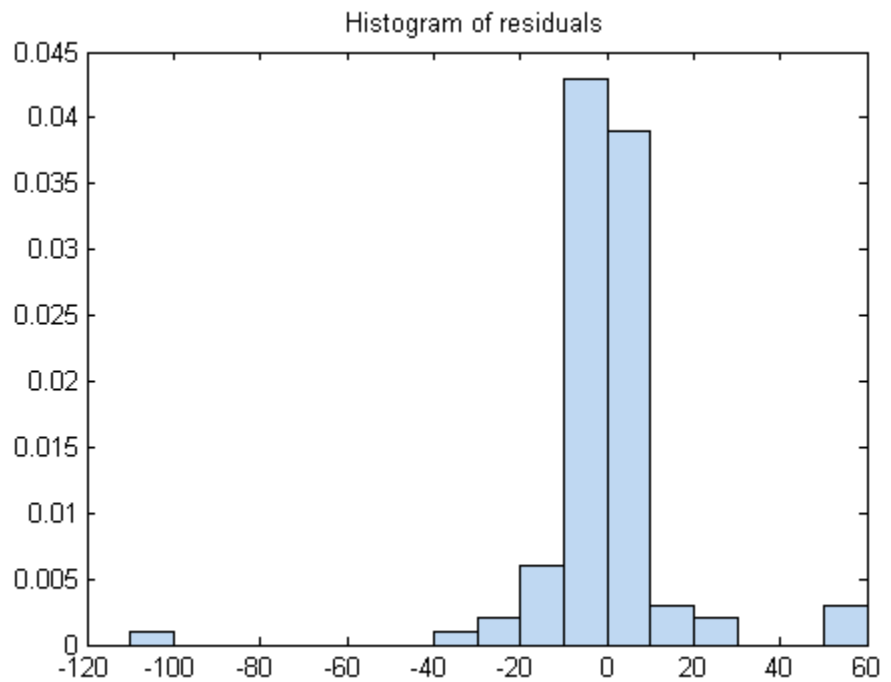
```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Create a default residuals plot.

```
plotResiduals(mdl)
```

# GeneralizedLinearModel.plotResiduals

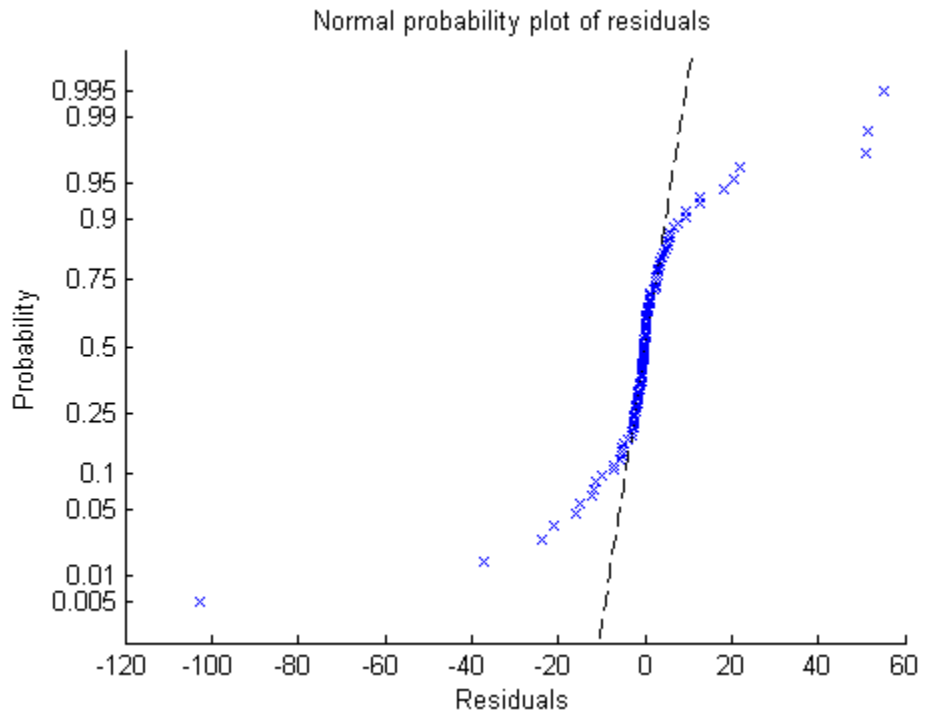
---



Create a probability plot.

```
plotResiduals mdl, 'probability'
```

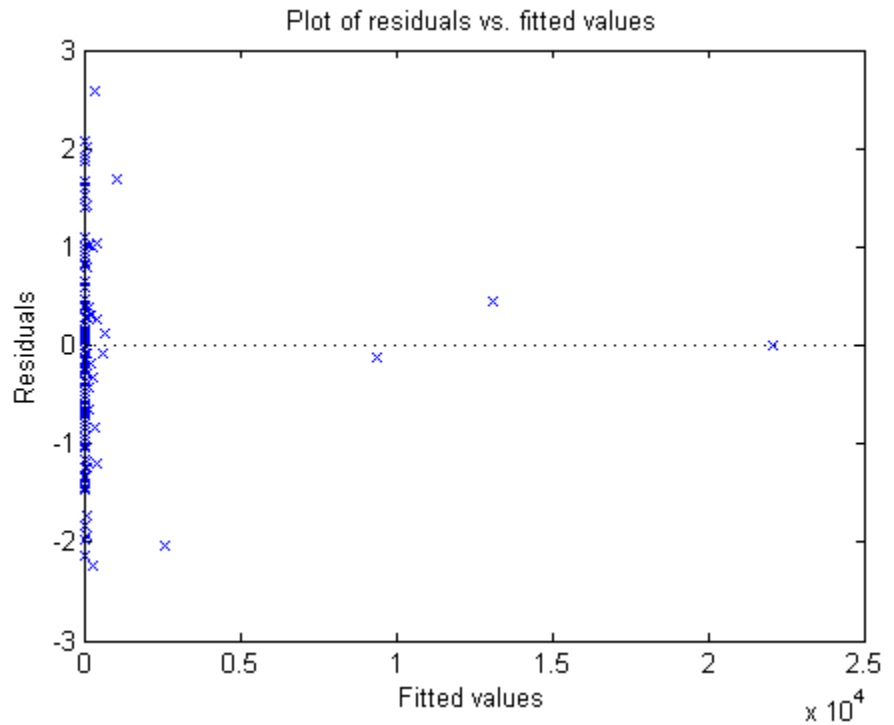
# GeneralizedLinearModel.plotResiduals



The residuals do not match a normal distribution in the tails—they are more spread out.

Create a plot of the fitted residuals of Anscombe type.

# GeneralizedLinearModel.plotResiduals



**See Also** [GeneralizedLinearModel |](#)

## Related Examples

- “Residuals — Model Quality for Training Data” on page 10-29

## Concepts

- “Generalized Linear Models” on page 10-12

## Purpose

Plot residuals of linear regression model

## Syntax

```
plotResiduals mdl
plotResiduals mdl,plottype
h = plotResiduals(...)
h = plotResiduals mdl,plottype,Name,Value
```

## Description

`plotResiduals(mdl)` gives a histogram plot of the residuals of the `mdl` linear model.

`plotResiduals(mdl,plottype)` plots residuals in a plot of type `plottype`.

`h = plotResiduals(...)` returns handles to the lines in the plot.

`h = plotResiduals(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

## Tips

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **plottype**

String specifying the type of plot:

'caseorder'	Residuals vs. case (row) order
'fitted'	Residuals vs. fitted values
'histogram'	Histogram
'lagged'	Residuals vs. lagged residual ( $r(t)$ vs. $r(t-1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot

# LinearModel.plotResiduals

---

**Default:** 'histogram'

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### **'Color'**

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

### **'LineStyle'**

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

### **'LineWidth'**

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### **'MarkerEdgeColor'**

Color of the marker or edge color for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

### **'MarkerFaceColor'**

Color of the marker face for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.



## 'MarkerSize'

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## 'ResidualType'

Type of residual used in the plot:

'Raw'	Observed minus fitted values
'Pearson'	Raw residuals divided by RMSE
'Standardized'	Raw residuals divided by their estimated standard deviation
'Studentized'	Raw residuals divided by an independent (delete-1) estimate of their standard deviation

**Default:** 'Raw'

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

### Linear Residuals Plot

Plot the residuals of a fitted linear model.

Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

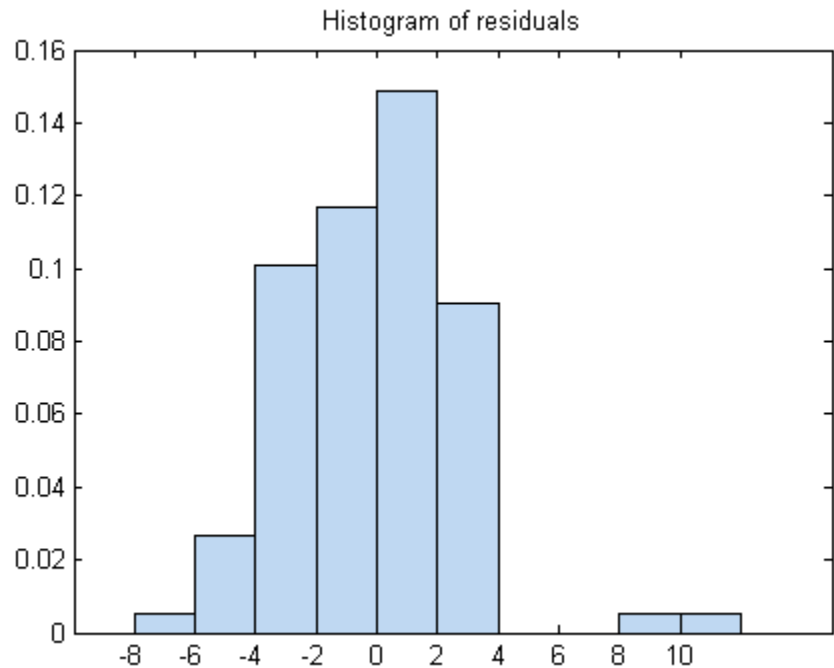
```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```

Plot the raw residuals.

```
plotResiduals(mdl)
```

# LinearModel.plotResiduals

---



## Residual Probability Plot

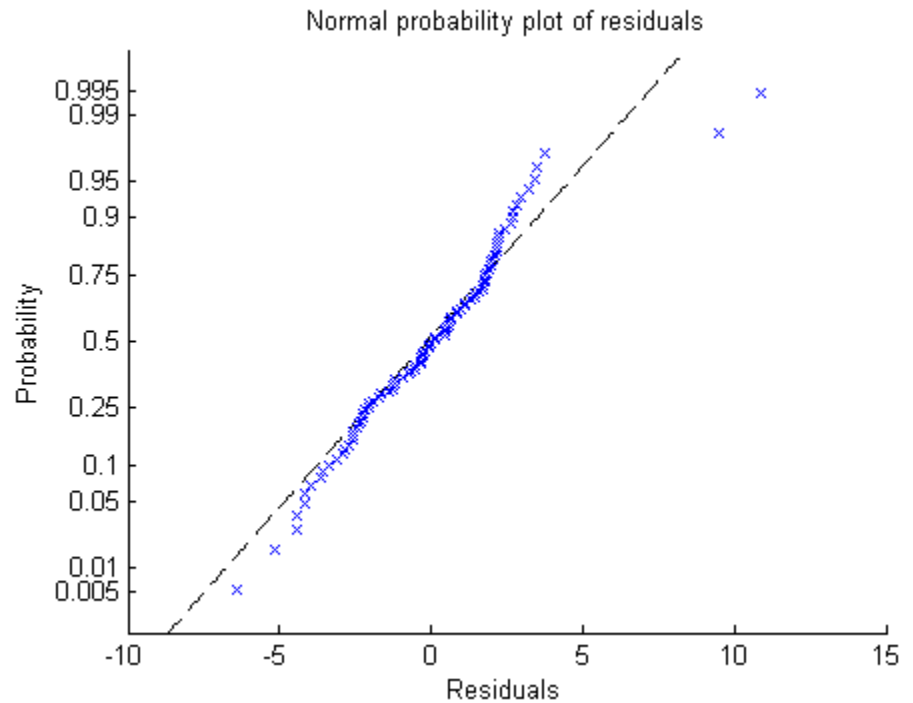
Create a normal probability plot of the residuals of a fitted linear model.

Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
X = [Weight,Model_Year];
mdl = fitlm(X,MPG,...
    'y ~ x2 + x1^2','Categorical',2);
```

Create a normal probability plot of the residuals of the fitted model.

```
plotResiduals(mdl,'probability')
```



## Alternatives

The `mdl.Residuals` dataset array contains the information in residual plots.

## See Also

`LinearModel | plotDiagnostics`

## Tutorials

- “Residuals — Model Quality for Training Data” on page 9-25
- “Linear Regression Workflow” on page 9-44
- “Compare large and small stepwise models” on page 9-119
- “Robust Regression versus Standard Least-Squares Fit” on page 9-125

# LinearModel.plotResiduals

---

## How To

- “Linear Regression” on page 9-12

# LinearMixedModel.plotResiduals

---

**Purpose** Plot residuals of linear mixed-effects model

**Syntax** `plotResiduals(lme,plottype)`  
`plotResiduals(lme,plottype,Name,Value)`

`h = plotResiduals( ___ )`

**Description** `plotResiduals(lme,plottype)` plots the raw conditional residuals of the linear mixed-effects model `lme` in a plot of the type specified by `plottype`.

`plotResiduals(lme,plottype,Name,Value)` also plots the residuals of the linear mixed-effects model `lme` with additional options specified by one or more name-value pair arguments. For example, you can specify the residual type to plot.

`plotResiduals` also accepts some other name-value pair arguments that specify the properties of the primary line in the plot. For those name-value pairs, see `plot`.

`h = plotResiduals( ___ )` returns a handle, `h`, to the lines or patches in the plot of residuals.

## Input Arguments

### **lme - Linear mixed-effects model**

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **plottype - Type of residual plot**

'histogram' (default) | 'caseorder' | 'fitted' | 'lagged' | 'probability' | 'symmetry'

Type of residual plot, specified as one of the following strings.

# LinearMixedModel.plotResiduals

---

'histogram'	Default. Histogram of residuals
'caseorder'	Residuals versus case (row) order
'fitted'	Residuals versus fitted values
'lagged'	Residuals versus lagged residual ( $r(t)$ versus $r(t - 1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot

**Example:** `plotResiduals(lme, 'lagged')`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## **'Conditional' - Indicator for conditional residuals**

True (default) | False

Indicator for conditional residuals, specified as the comma-separated pair consisting of 'Conditional' and one of the following.

True	Contribution from both fixed effects and random effects (conditional)
False	Contribution from only fixed effects (marginal)

**Example:** `'Conditional', 'False'`

## **'ResidualType' - Residual type**

# LinearMixedModel.plotResiduals

'Raw' (default) | 'Pearson' | 'Standardized'

Residual type, specified by the comma-separated pair consisting of ResidualType and one of the following.

Residual Type	Conditional	Marginal
'Raw'	$r_i^C = [y - X\hat{\beta} - Zb]_i$	$r_i^M = [y - X\hat{\beta}]_i$
'Pearson'	$pr_i^C = \frac{r_i^C}{\sqrt{[\text{Var}_{y,b}(y - X\hat{\beta} - Zb)]_{ii}}}$	$pr_i^M = \frac{r_i^M}{\sqrt{[\text{Var}_y(y - X\hat{\beta})]_{ii}}}$
'Standardized'	$st_i^C = \frac{r_i^C}{\sqrt{[\text{Var}_y(r^C)]_{ii}}}$	$st_i^M = \frac{r_i^M}{\sqrt{[\text{Var}_y(r^M)]_{ii}}}$

Example: 'ResidualType', 'Standardized'

## Output Arguments

### h - Handle to residual plot

handle

Handle to the residual plot, returned as a handle.

## Examples

### Examine Residuals

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

# LinearMixedModel.plotResiduals

---

weight contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define Subject and Program as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);  
ds.Subject = nominal(ds.Subject);  
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

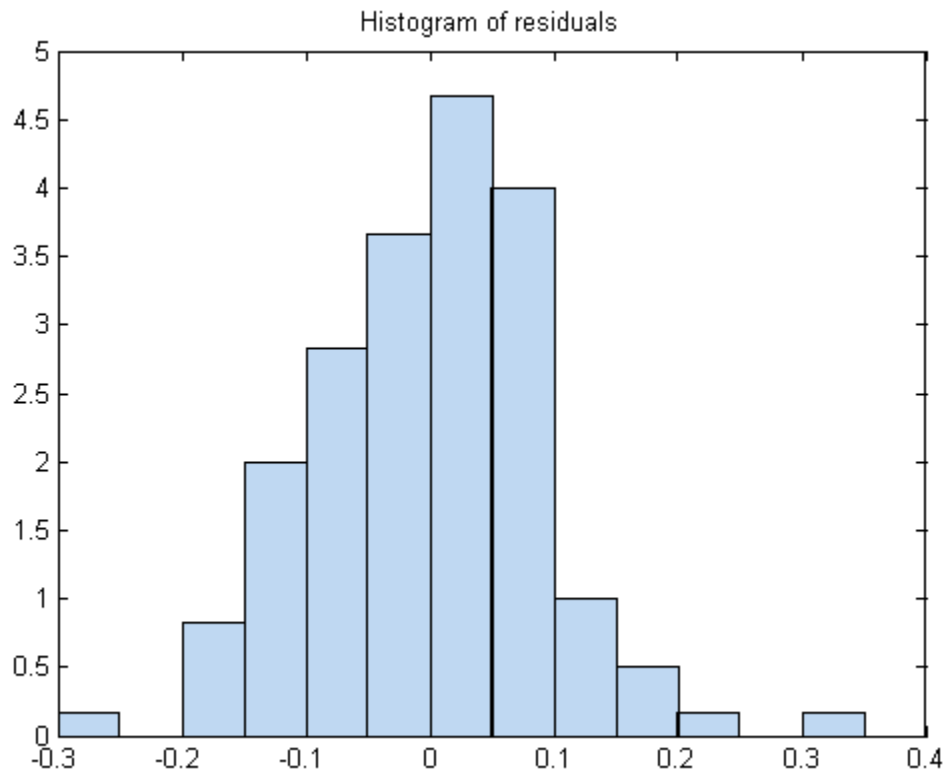
```
lme = fitlme(ds,'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Plot the histogram of the raw residuals.

```
plotResiduals(lme)
```



# LinearMixedModel.plotResiduals

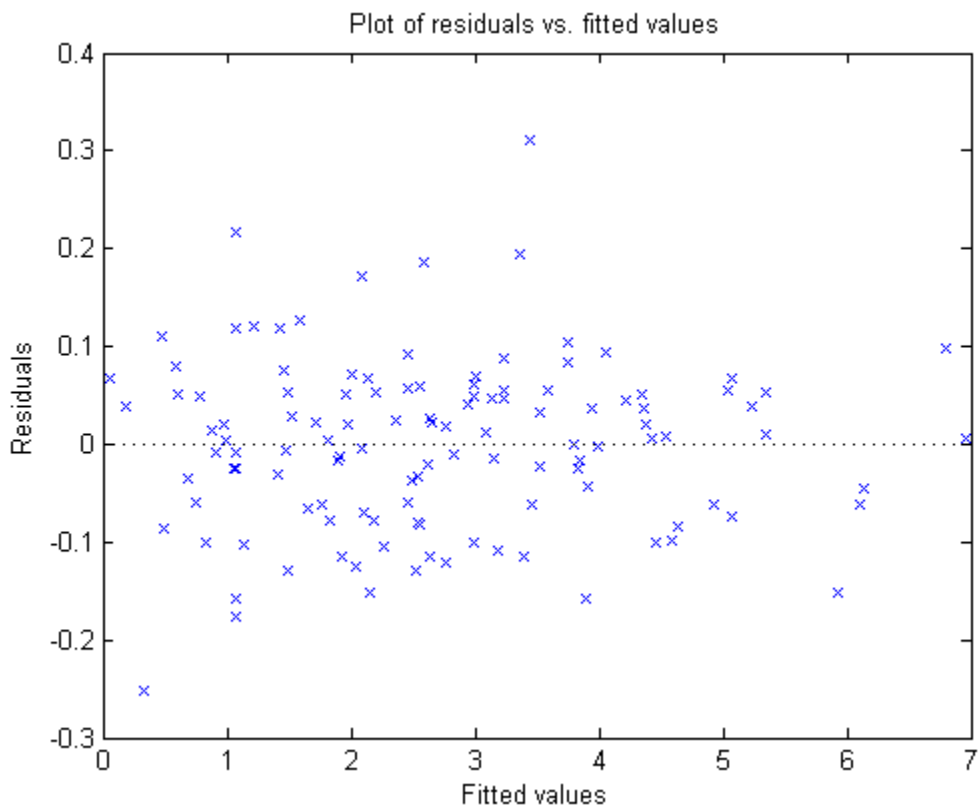


Plot the residuals versus the fitted values.

```
figure();  
plotResiduals(lme, 'fitted')
```

# LinearMixedModel.plotResiduals

---

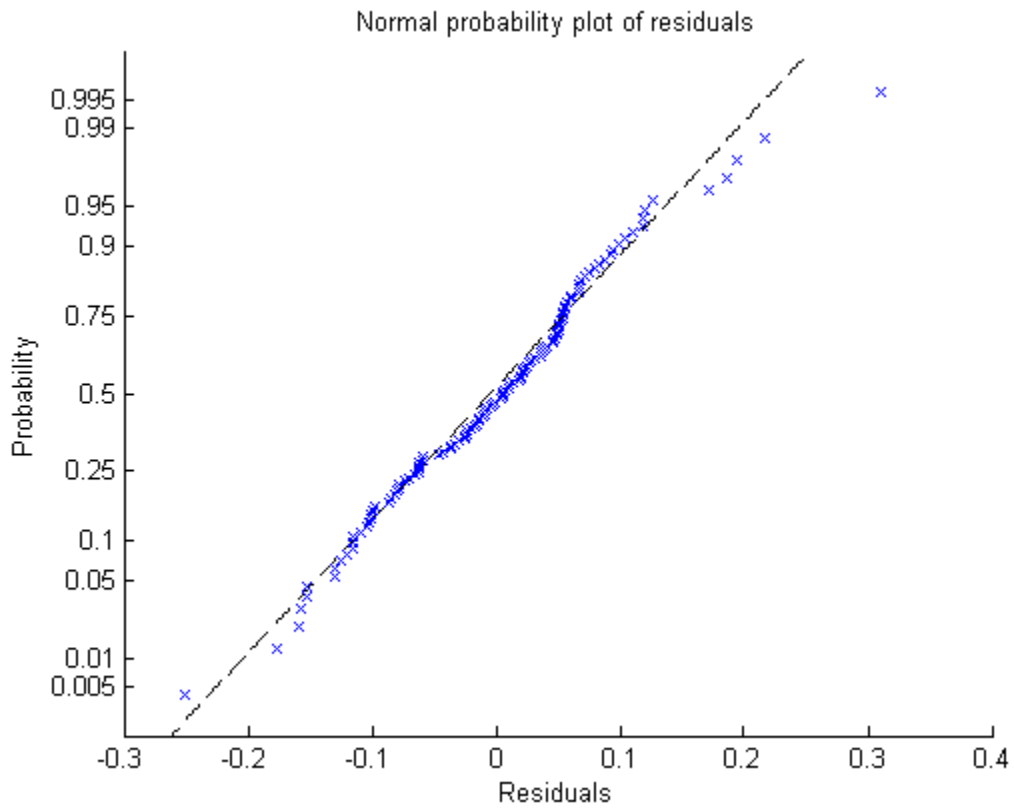


There is no obvious pattern, so there are no immediate signs of heteroscedasticity.

Create the normal probability plot of residuals.

```
figure();  
plotResiduals(lme, 'probability')
```

# LinearMixedModel.plotResiduals



Data appears to be normal.

Find the observation number for the data that appears to be an outlier to the right of the plot.

```
find(residuals(lme)>0.25)
```

```
ans =
```

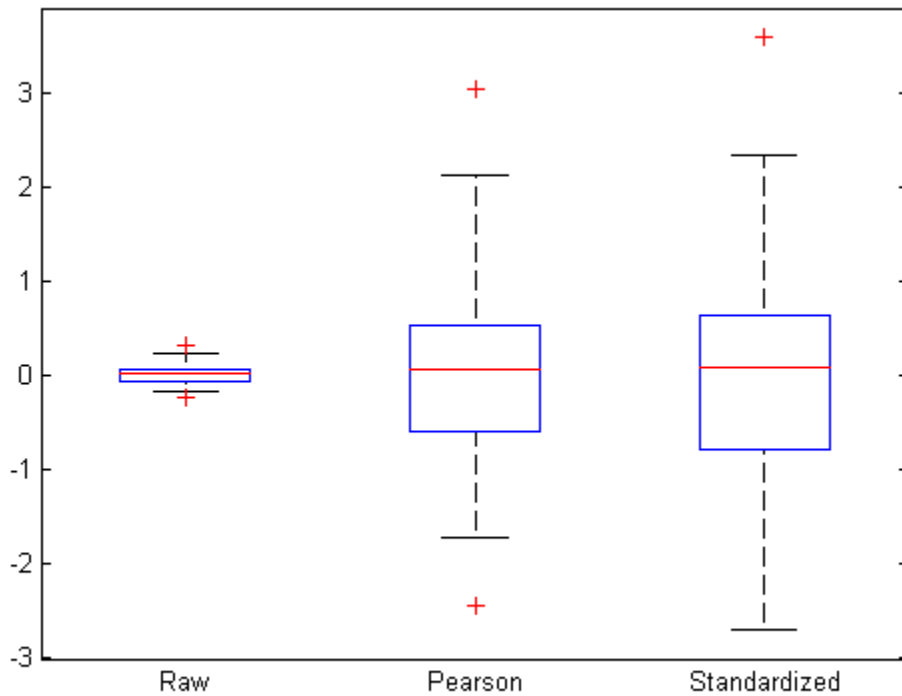
```
101
```

# LinearMixedModel.plotResiduals

---

Create a box plot of the raw, Pearson, and standardized residuals.

```
r = residuals(lme);  
pr = residuals(lme, 'ResidualType', 'Pearson');  
st = residuals(lme, 'ResidualType', 'Standardized');  
X = [r pr st];  
boxplot(X, 'labels', {'Raw', 'Pearson', 'Standardized'});
```



All three box plots point out the outlier on the right tail of the distribution. The box plots of raw and Pearson residuals also point

out a second possible outlier on the left tail. Find the corresponding observation number.

```
find(pr<-2)
```

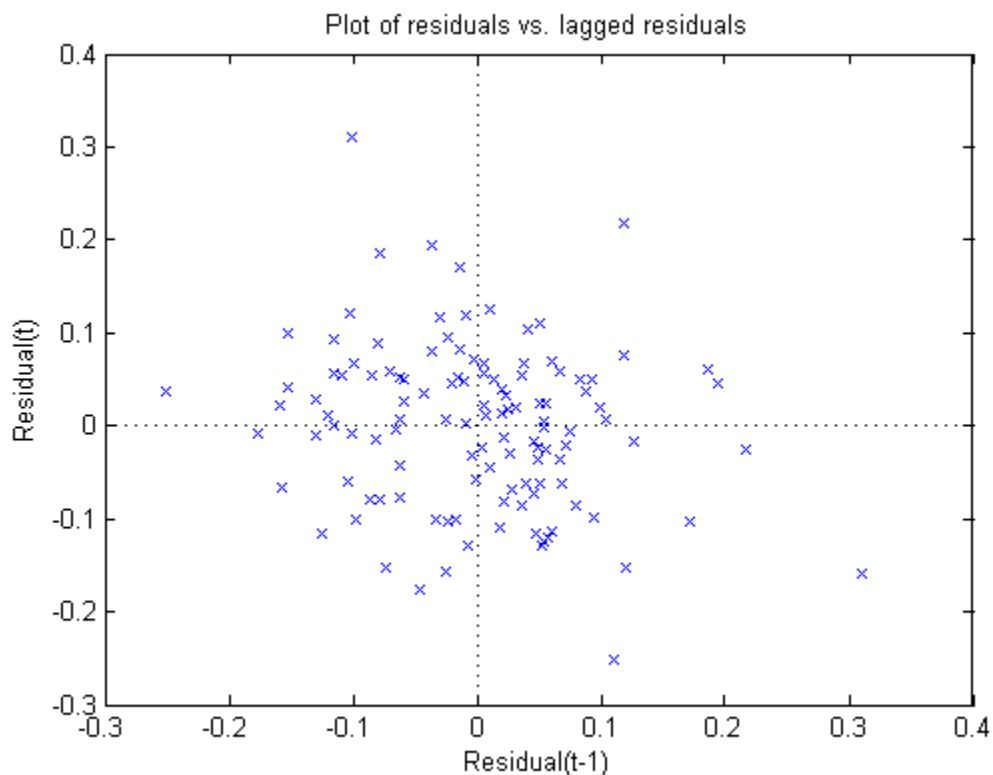
```
ans =
```

```
10
```

Plot the raw residuals versus lagged residuals.

```
plotResiduals(lme, 'lagged')
```

# LinearMixedModel.plotResiduals



There is no obvious pattern in the graph. The residuals do not appear to be correlated.

## Definitions

### Conditional and Marginal Residuals

Conditional residuals include contributions from both fixed and random effects, whereas marginal residuals include contribution from only fixed effects.

Suppose the linear mixed-effects model `lme` has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also,

suppose the  $p$ -by-1 estimated fixed-effects vector is  $\hat{\beta}$ , and the  $q$ -by-1 estimated best linear unbiased predictor (BLUP) vector of random effects is  $\hat{b}$ . The fitted conditional response is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b},$$

and the fitted marginal response is

$$\hat{y}_{Mar} = X\hat{\beta}.$$

`residuals` can return three types of residuals: raw, Pearson, and standardized. For any type, you can compute the conditional or the marginal residuals. For example, the conditional raw residual is

$$r_{Cond} = y - X\hat{\beta} - Z\hat{b},$$

and the marginal raw residual is

$$r_{Mar} = y - X\hat{\beta}.$$

For more information on other types of residuals, see the `ResidualType` name-value pair argument.

## See Also

`LinearMixedModel` | `residuals` | `fitted` |

# NonLinearModel.plotResiduals

---

**Purpose** Plot residuals of nonlinear regression model

**Syntax**

```
plotResiduals mdl
plotResiduals mdl,plottype
h = plotResiduals(...)
h = plotResiduals mdl,plottype,Name,Value)
```

**Description**

`plotResiduals(mdl)` gives a histogram plot of the residuals of the `mdl` nonlinear model.

`plotResiduals(mdl,plottype)` plots residuals in a plot of type `plottype`.

`h = plotResiduals(...)` returns handles to the lines in the plot.

`h = plotResiduals(mdl,plottype,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For many plots, the Data Cursor tool in the figure window displays the  $x$  and  $y$  values for any data point, along with the observation name or number.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **plottype**

String specifying the type of plot:

'caseorder'	Residuals vs. case (row) order
'fitted'	Residuals vs. fitted values
'histogram'	Histogram
'lagged'	Residuals vs. lagged residual ( $r(t)$ vs. $r(t-1)$ )
'probability'	Normal probability plot
'symmetry'	Symmetry plot



**Default:** 'histogram'

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

---

**Note** The plot property name-value pairs apply to the first returned handle `h(1)`.

---

### **'Color'**

Color of the line or marker, a string or `ColorSpec` specification. For details, see `linespec`.

### **'LineStyle'**

Type of line, a string or `lineseriesproperties` specification. For details, see `linespec`.

### **'LineWidth'**

Width of the line or edges of filled area, in points, a positive scalar. One point is 1/72 inch.

**Default:** 0.5

### **'MarkerEdgeColor'**

Color of the marker or edge color for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

### **'MarkerFaceColor'**

Color of the marker face for filled markers, a string or `ColorSpec` specification. For details, see `linespec`.

# NonLinearModel.plotResiduals

---

## 'MarkerSize'

Size of the marker in points, a strictly positive scalar. One point is 1/72 inch.

## 'ResidualType'

Type of residual used in the plot:

'Raw'	Observed minus fitted values
'Pearson'	Raw residuals divided by RMSE
'Standardized'	Raw residuals divided by their estimated standard deviation
'Studentized'	Raw residuals divided by an independent (delete-1) estimate of their standard deviation

**Default:** 'Raw'

## Output Arguments

**h**

Vector of handles to lines or patches in the plot.

## Examples

### Residual Plot

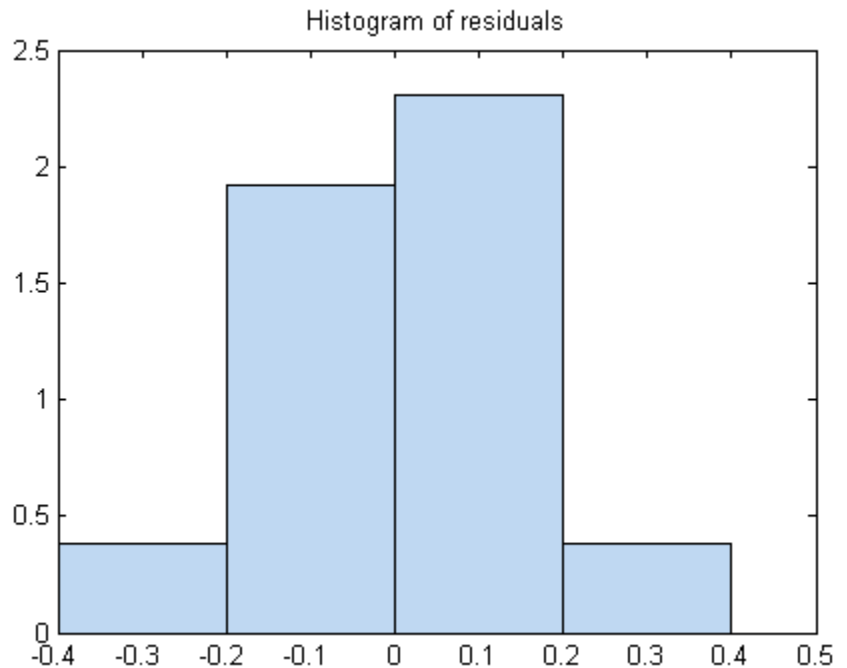
Plot the residuals of a fitted nonlinear model.

Load the reaction data and fit a model of the reaction rate as a function of reactants.

```
load reaction
mdl = fitnlm(reactants,rate,@hougen,[1 .05 .02 .1 2]);
```

Plot the residuals of the fitted model.

```
plotResiduals(mdl)
```



## Residual Probability Plot

Create a normal probability plot of the residuals of a fitted nonlinear model.

Load the reaction data and fit a model of the reaction rate as a function of reactants.

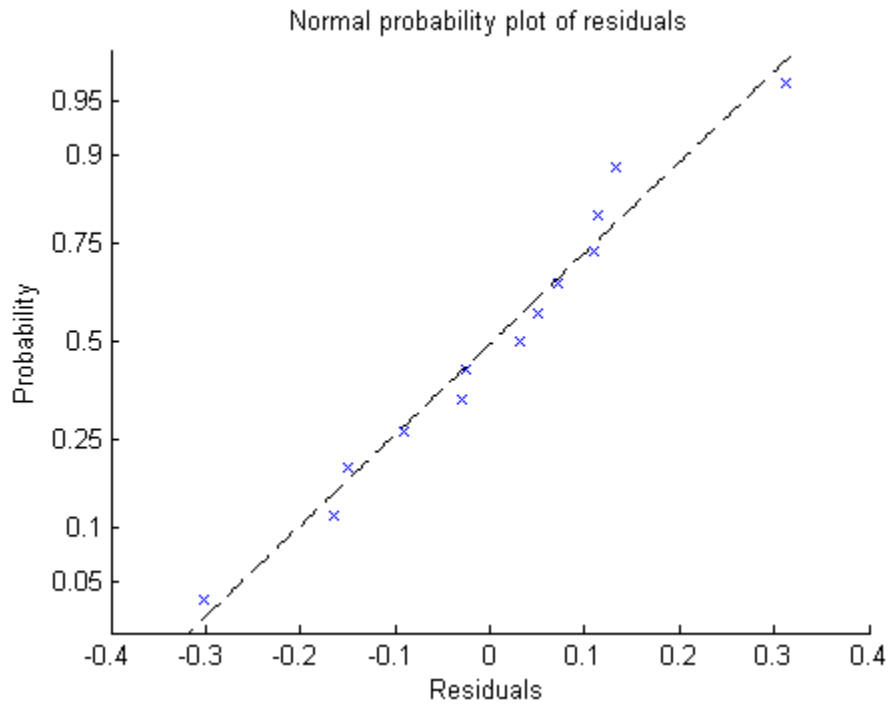
```
load reaction
mdl = fitnlm(reactants,rate,@hougen,[1 .05 .02 .1 2]);
```

Create a normal probability plot of the residuals of the fitted model.

```
plotResiduals(mdl,'probability')
```

# NonLinearModel.plotResiduals

---



## See Also

`NonLinearModel | plotDiagnostics |`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-8
- “Nonlinear Regression Workflow” on page 11-16

## Concepts

- “Nonlinear Regression” on page 11-2

<b>Purpose</b>	Plot of slices through fitted generalized linear regression surface
<b>Syntax</b>	<pre>plotSlice mdl h = plotSlice mdl</pre>
<b>Description</b>	<p><code>plotSlice(mdl)</code> creates a new figure containing a series of plots, each representing a slice through the regression surface predicted by <code>mdl</code>. For each plot, the surface slice is shown as a function of a single predictor variable, with the other predictor variables held constant.</p> <p><code>h = plotSlice(mdl)</code> returns handles to the lines in the plot.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• If there are more than eight predictors, <code>plotSlice</code> selects the first five for plotting. Use the <b>Predictors</b> menu to control which predictors are plotted.</li><li>• The <b>Bounds</b> menu lets you choose between simultaneous or non-simultaneous bounds, and between bounds on the function or bounds on a new observation.</li></ul>
<b>Input Arguments</b>	<p><b>mdl</b></p> <p>Generalized linear model, as constructed by <code>fitglm</code> or <code>stepwiseglm</code>.</p>
<b>Output Arguments</b>	<p><b>h</b></p> <p>Vector of handles to lines or patches in the plot.</p>
<b>Examples</b>	<p><b>Slice Plot of Generalized Linear Regression Model</b></p> <p>Create a slice plot of a Poisson generalized linear model.</p> <p>Generate artificial data for the model using Poisson random numbers with two underlying predictors <math>X(1)</math> and <math>X(2)</math>.</p> <pre>rng('default') % for reproducibility rndvars = randn(100,2); X = [2+rndvars(:,1),rndvars(:,2)];</pre>

# GeneralizedLinearModel.plotSlice

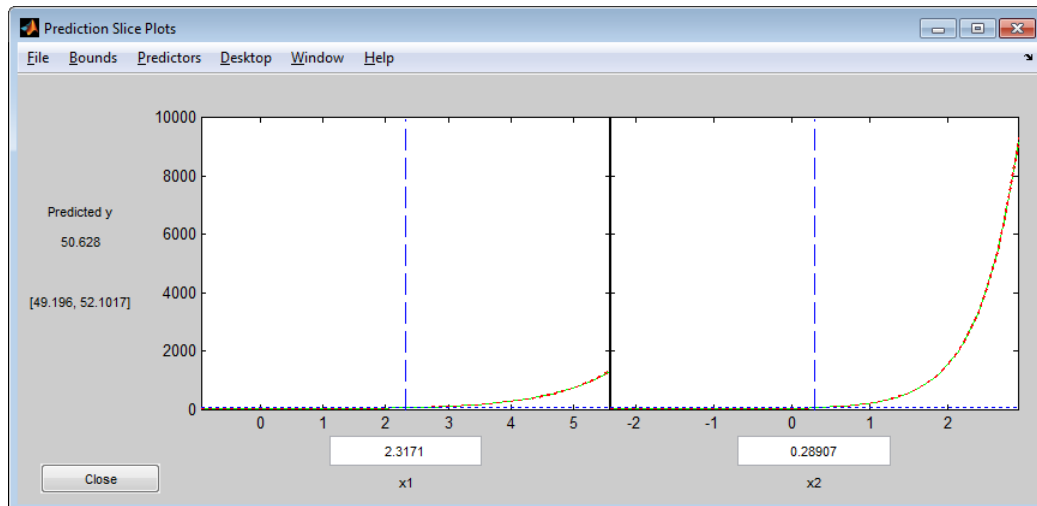
```
mu = exp(1 + X*[1;2]);  
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

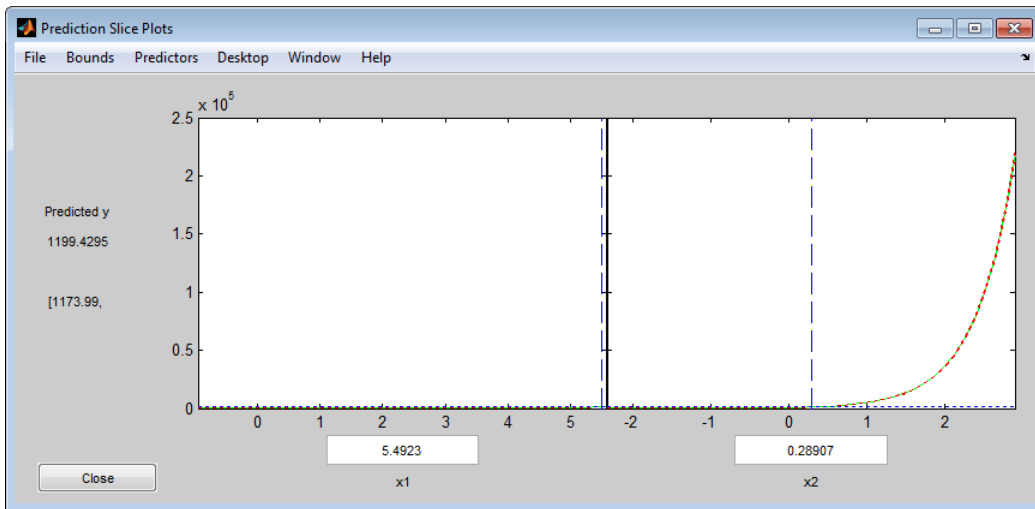
```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Create the slice plot.

```
plotSlice(mdl)
```



Drag the x1 prediction line to the right and view the changes in the prediction and the response curve for the x2 predictor.



## See Also

`GeneralizedLinearModel | predict |`

## Related Examples

- “Diagnostic Plots” on page 10-26
- “Plots to Understand Predictor Effects and How to Modify a Model” on page 10-32

## Concepts

- “Generalized Linear Models” on page 10-12

# LinearModel.plotSlice

---

**Purpose** Plot of slices through fitted linear regression surface

**Syntax** `plotSlice mdl`  
`h = plotSlice(mdl)`

**Description** `plotSlice(mdl)` creates a new figure containing a series of plots, each representing a slice through the regression surface predicted by `mdl`. For each plot, the surface slice is shown as a function of a single predictor variable, with the other predictor variables held constant.

`h = plotSlice(mdl)` returns handles to the lines in the plot.

- Tips**
- If there are more than eight predictors, `plotSlice` selects the first five for plotting. Use the **Predictors** menu to control which predictors are plotted.
  - The **Bounds** menu lets you choose between simultaneous or non-simultaneous bounds, and between bounds on the function or bounds on a new observation.

**Input Arguments** **mdl**  
Linear model, as constructed by `fitlm` or `stepwiselm`.

**Output Arguments** **h**  
Vector of handles to lines or patches in the plot.

## Examples

### Slice Plot

Plot the slices through a fitted linear model.

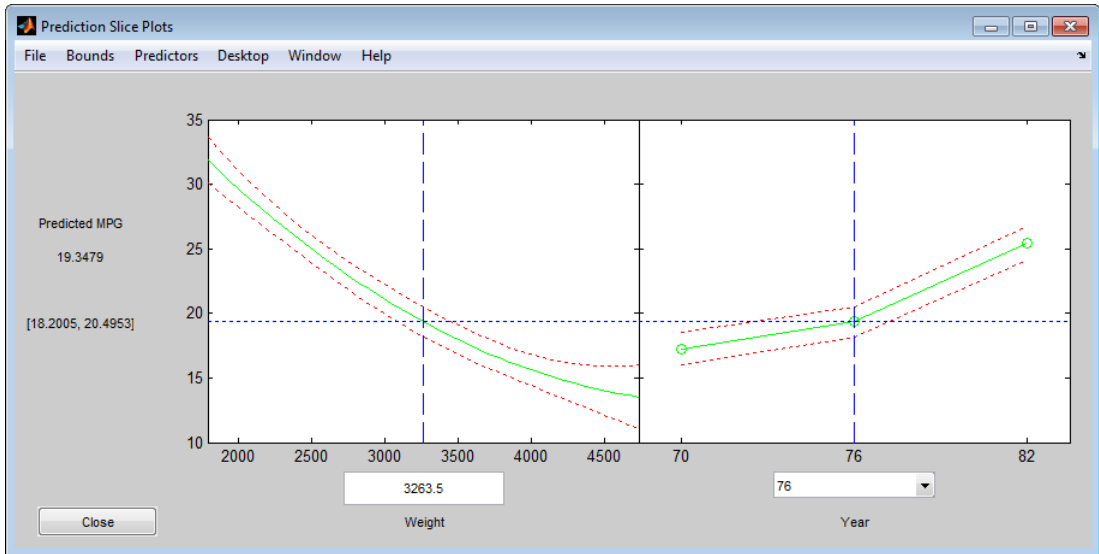
Load the `carsmall` data and fit a linear model of the mileage as a function of model year, weight, and weight squared.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
mdl = fitlm(ds,'MPG ~ Year + Weight^2');
```



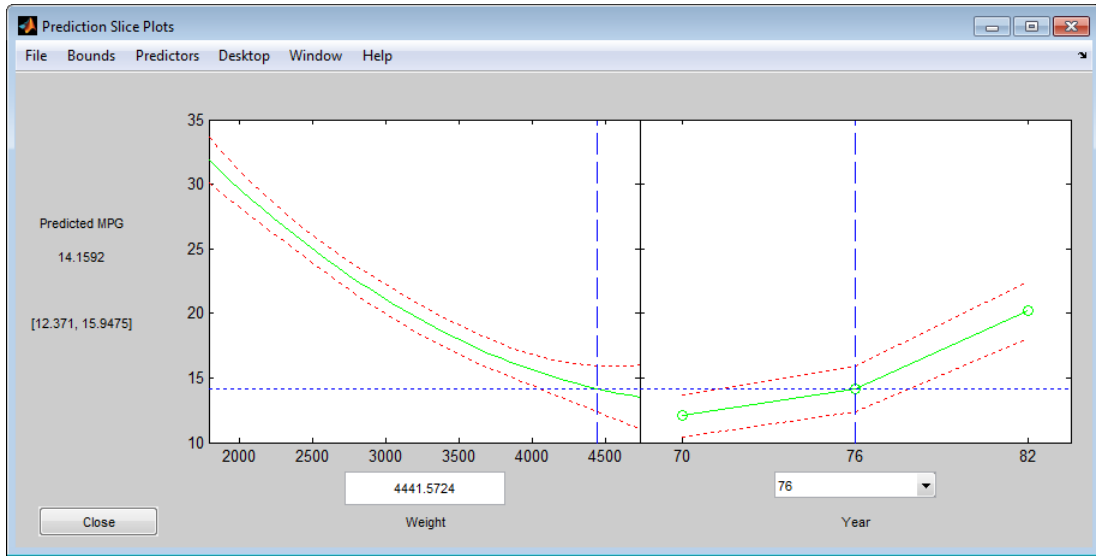
Create a slice plot.

```
plotSlice mdl
```



Drag the Weight prediction line to the right and observe the change in the predicted MPG and the response curve for Year.

# LinearModel.plotSlice



## See Also

`LinearModel | predict`

## Tutorials

- “Plots to Understand Predictor Effects” on page 9-30

## How To

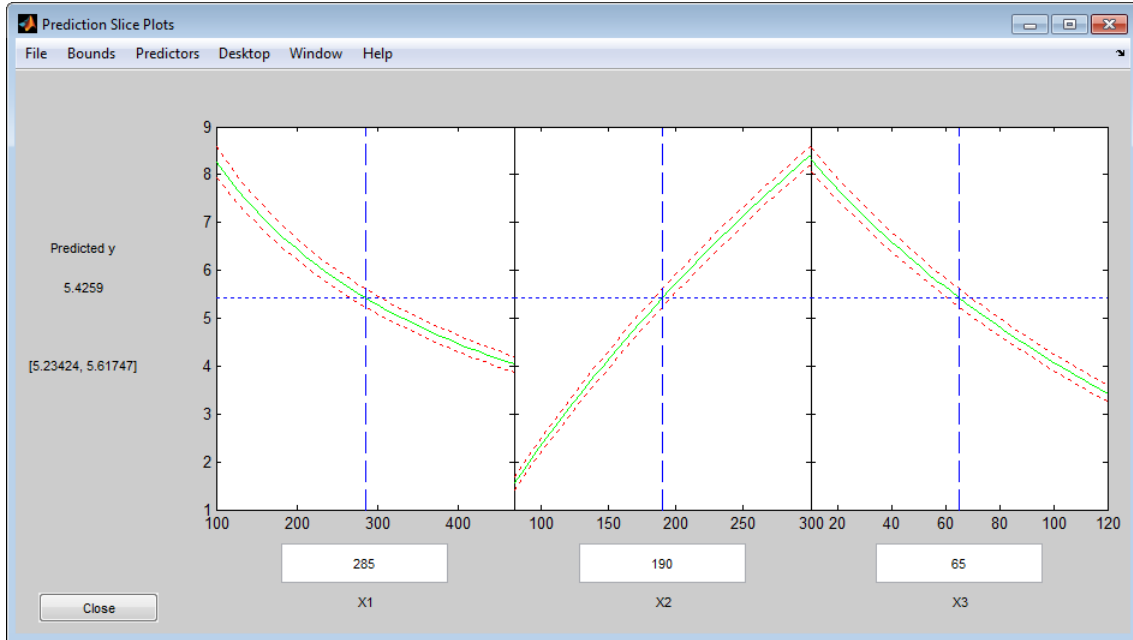
- “Linear Regression” on page 9-12

<b>Purpose</b>	Plot of slices through fitted nonlinear regression surface
<b>Syntax</b>	<pre>plotSlice mdl h = plotSlice mdl</pre>
<b>Description</b>	<p><code>plotSlice(mdl)</code> creates a new figure containing a series of plots, each representing a slice through the regression surface predicted by <code>mdl</code>. For each plot, the surface slice is shown as a function of a single predictor variable, with the other predictor variables held constant.</p> <p><code>h = plotSlice(mdl)</code> returns handles to the lines in the plot.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• If there are more than eight predictors, <code>plotSlice</code> selects the first five for plotting. Use the <b>Predictors</b> menu to control which predictors are plotted.</li><li>• The <b>Bounds</b> menu lets you choose between simultaneous or non-simultaneous bounds, and between bounds on the function or bounds on a new observation.</li></ul>
<b>Input Arguments</b>	<p><b>mdl</b></p> <p>Nonlinear regression model, constructed by <code>fitnlm</code>.</p>
<b>Output Arguments</b>	<p><b>h</b></p> <p>Vector of handles to lines or patches in the plot.</p>
<b>Examples</b>	<p><b>Slice Plot</b></p> <p>Plot slices of a fitted nonlinear model.</p> <p>Load the reaction data and fit a model of the reaction rate as a function of reactants.</p> <pre>load reaction mdl = fitnlm(reactants,...     rate,@hougen,[1 .05 .02 .1 2]);</pre>

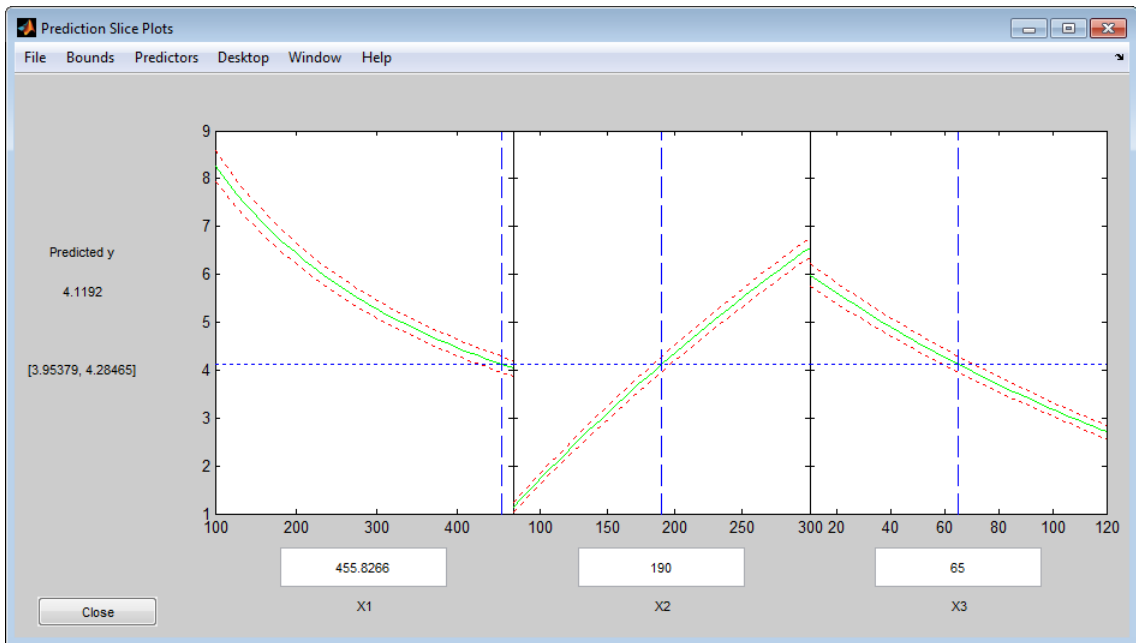
# NonLinearModel.plotSlice

Create a slice plot

```
plotSlice mdl
```



Drag the X1 prediction line to the right, and observe the change in the predicted response  $y$  and in the predicted response curves to X2 and X3.



## See Also

`NonLinearModel | predict |`

## Related Examples

- “Examine Quality and Adjust the Fitted Model” on page 11-8
- “Predict or Simulate Responses to New Data” on page 11-12
- “Nonlinear Regression Workflow” on page 11-16

## Concepts

- “Nonlinear Regression” on page 11-2

## Purpose

Partial least-squares regression

## Syntax

```
[XL,YL] = plsregress(X,Y,ncomp)
[XL,YL,XS] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA] = PLSREGRESS(X,Y,ncomp,...)
[XL,YL,XS,YS,BETA,PCTVAR] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA,PCTVAR,MSE] = plsregress(X,Y,ncomp)
[XL,YL,XS,YS,BETA,PCTVAR,MSE] = plsregress(...,param1,val1,
    param2,val2,...)
[XL,YL,XS,YS,BETA,PCTVAR,MSE,stats] = PLSREGRESS(X,Y,ncomp,
    ...)
```

## Description

`[XL,YL] = plsregress(X,Y,ncomp)` computes a partial least-squares (PLS) regression of  $Y$  on  $X$ , using  $ncomp$  PLS components, and returns the predictor and response loadings in  $XL$  and  $YL$ , respectively.  $X$  is an  $n$ -by- $p$  matrix of predictor variables, with rows corresponding to observations and columns to variables.  $Y$  is an  $n$ -by- $m$  response matrix.  $XL$  is a  $p$ -by- $ncomp$  matrix of predictor loadings, where each row contains coefficients that define a linear combination of PLS components that approximate the original predictor variables.  $YL$  is an  $m$ -by- $ncomp$  matrix of response loadings, where each row contains coefficients that define a linear combination of PLS components that approximate the original response variables.

`[XL,YL,XS] = plsregress(X,Y,ncomp)` returns the predictor scores  $XS$ , that is, the PLS components that are linear combinations of the variables in  $X$ .  $XS$  is an  $n$ -by- $ncomp$  orthonormal matrix with rows corresponding to observations and columns to components.

`[XL,YL,XS,YS] = plsregress(X,Y,ncomp)` returns the response scores  $YS$ , that is, the linear combinations of the responses with which the PLS components  $XS$  have maximum covariance.  $YS$  is an  $n$ -by- $ncomp$  matrix with rows corresponding to observations and columns to components.  $YS$  is neither orthogonal nor normalized.

`plsregress` uses the SIMPLS algorithm, first centering  $X$  and  $Y$  by subtracting off column means to get centered variables  $X0$  and  $Y0$ .

However, it does not rescale the columns. To perform PLS with standardized variables, use `zscore` to normalize `X` and `Y`.

If `ncomp` is omitted, its default value is `min(size(X,1)-1,size(X,2))`.

The relationships between the scores, loadings, and centered variables `X0` and `Y0` are:

$$XL = (XS \setminus X0)' = X0' * XS,$$

$$YL = (XS \setminus Y0)' = Y0' * XS,$$

`XL` and `YL` are the coefficients from regressing `X0` and `Y0` on `XS`, and `XS*XL'` and `XS*YL'` are the PLS approximations to `X0` and `Y0`.

`plsregress` initially computes `YS` as:

$$YS = Y0 * YL = Y0 * Y0' * XS,$$

By convention, however, `plsregress` then orthogonalizes each column of `YS` with respect to preceding columns of `XS`, so that `XS' * YS` is lower triangular.

`[XL, YL, XS, YS, BETA] = PLSREGRESS(X, Y, ncomp, ...)` returns the PLS regression coefficients `BETA`. `BETA` is a  $(p+1)$ -by- $m$  matrix, containing intercept terms in the first row:

$$Y = [\text{ones}(n, 1), X] * BETA + Y_{\text{residuals}},$$

`Y0 = X0 * BETA(2:end, :) + Y_{\text{residuals}}`. Here `Y_{\text{residuals}}` is the vector of response residuals.

`[XL, YL, XS, YS, BETA, PCTVAR] = plsregress(X, Y, ncomp)` returns a 2-by-`ncomp` matrix `PCTVAR` containing the percentage of variance explained by the model. The first row of `PCTVAR` contains the percentage of variance explained in `X` by each PLS component, and the second row contains the percentage of variance explained in `Y`.

`[XL, YL, XS, YS, BETA, PCTVAR, MSE] = plsregress(X, Y, ncomp)` returns a 2-by- $(ncomp+1)$  matrix `MSE` containing estimated mean-squared errors for PLS models with 0:`ncomp` components. The first row of `MSE` contains mean-squared errors for the predictor variables in `X`, and the second row contains mean-squared errors for the response variable(s) in `Y`.

# plsregress

`[XL,YL,XS,YS,BETA,PCTVAR,MSE] = plsregress(...,param1,va11, param2,va12,...)` specifies optional parameter name/value pairs from the following table to control the calculation of MSE.

Parameter	Value
'cv'	<p>The method used to compute MSE.</p> <ul style="list-style-type: none"> <li>• When the value is a positive integer <i>k</i>, <code>plsregress</code> uses <i>k</i>-fold cross-validation.</li> <li>• When the value is an object of the <code>cvpartition</code> class, other forms of cross-validation can be specified.</li> <li>• When the value is 'resubstitution', <code>plsregress</code> uses <i>X</i> and <i>Y</i> both to fit the model and to estimate the mean-squared errors, without cross-validation.</li> </ul> <p>The default is 'resubstitution'.</p>
'mcreps'	<p>A positive integer indicating the number of Monte-Carlo repetitions for cross-validation. The default value is 1. The value must be 1 if the value of 'cv' is 'resubstitution'.</p>
options	<p>A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the <code>options</code> structure with <code>statset</code>. Option fields:</p> <ul style="list-style-type: none"> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: 'mlfg6331_64' or 'mrg32k3a'.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array consisting of one such object. If you do not specify <code>Streams</code>, <code>plsregress</code> uses the default stream.</li> </ul>

`[XL,YL,XS,YS,BETA,PCTVAR,MSE,stats] = PLSREGRESS(X,Y,ncomp,...)` returns a structure `stats` with the following fields:



- $W$  — A  $p$ -by- $n_{\text{comp}}$  matrix of PLS weights so that  $XS = X0*W$ .
- $T^2$  — The  $T^2$  statistic for each point in  $XS$ .
- $X_{\text{residuals}}$  — The predictor residuals, that is,  $X0 - XS*XL'$ .
- $Y_{\text{residuals}}$  — The response residuals, that is,  $Y0 - XS*YL'$ .

## Examples

Load data on near infrared (NIR) spectral intensities of 60 samples of gasoline at 401 wavelengths, and their octane ratings:

```
load spectra
X = NIR;
y = octane;
```

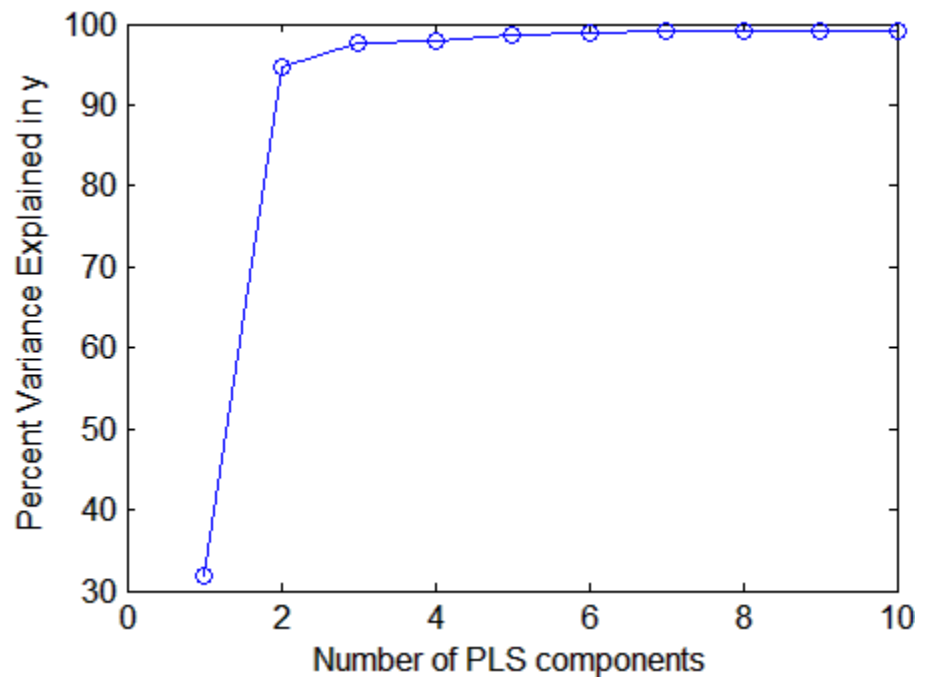
Perform PLS regression with ten components:

```
[XL,y1,XS,YS,beta,PCTVAR] = plsregress(X,y,10);
```

Plot the percent of variance explained in the response variable as a function of the number of components:

```
plot(1:10,cumsum(100*PCTVAR(2,:)),'-bo');
xlabel('Number of PLS components');
ylabel('Percent Variance Explained in y');
```

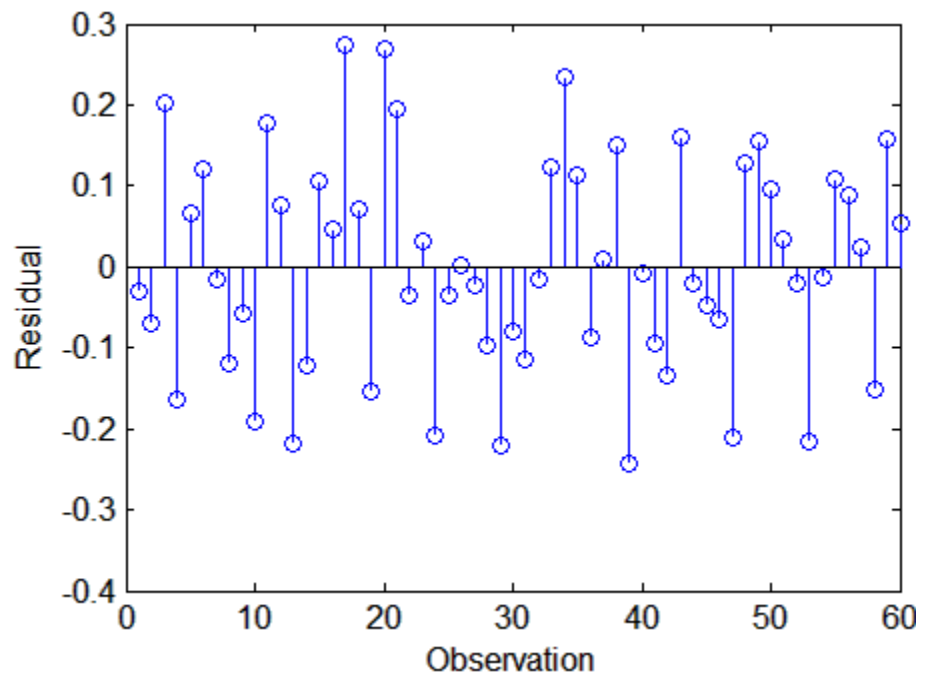
# plsregress



Compute the fitted response and display the residuals:

```
yfit = [ones(size(X,1),1) X]*beta;  
residuals = y-yfit;
```

```
stem(residuals)  
xlabel('Observation');  
ylabel('Residual');
```



## References

- [1] de Jong, S. “SIMPLS: An Alternative Approach to Partial Least Squares Regression.” *Chemometrics and Intelligent Laboratory Systems*. Vol. 18, 1993, pp. 251–263.
- [2] Rosipal, R., and N. Kramer. “Overview and Recent Advances in Partial Least Squares.” *Subspace, Latent Structure and Feature Selection: Statistical and Optimization Perspectives Workshop (SLSFS 2005), Revised Selected Papers (Lecture Notes in Computer Science 3940)*. Berlin, Germany: Springer-Verlag, 2006, pp. 34–51.

## See Also

regress | sequentialfs

# sobolset.PointOrder property

---

**Purpose** Point generation method

**Description** The `PointOrder` property contains a string that specifies the order in which the Sobol sequence points are produced. The property value must be one of 'standard' or 'graycode'. When set to 'standard' the points produced match the original Sobol sequence implementation. When set to 'graycode', the sequence is generated using an implementation that uses the Gray code of the index instead of the index itself.

**Purpose** Point set from which stream is drawn

**Description** The PointSet property contains a copy of the point set from which the stream is providing points. The point set is specified during construction of a quasi-random stream and cannot subsequently be altered.

## Examples

```
Q = grandstream('sobol', 5, 'Skip', 8);  
% Create a new stream based on the same sequence as that in Q  
Q2 = grandstream(Q.PointSet);  
u1 = rand(Q, 10)  
u2 = rand(Q2, 10) % contains exactly the same values as u1
```

# poisscdf

---

**Purpose** Poisson cumulative distribution function

**Syntax**  
`p = poisscdf(x,lambda)`  
`p = poisscdf(x,lambda,'upper')`

**Description** `p = poisscdf(x,lambda)` returns the Poisson cdf at each value in `x` using the corresponding mean parameters in `lambda`. `x` and `lambda` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `lambda` must be positive.

`p = poisscdf(x,lambda,'upper')` returns the complement of the Poisson cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The Poisson cdf is

$$p = F(x | \lambda) = e^{-\lambda} \sum_{i=0}^{\text{floor}(x)} \frac{\lambda^i}{i!}$$

## Examples **Compute Poisson Distribution cdf**

For example, consider a Quality Assurance department that performs random tests of individual hard disks. Their policy is to shut down the manufacturing process if an inspector finds more than four bad sectors on a disk. What is the probability of shutting down the process if the mean number of bad sectors ( $\lambda$ ) is two?

```
probability = 1-poisscdf(4,2)
```

```
probability =  
    0.0527
```

About 5% of the time, a normally functioning manufacturing process produces more than four flaws on a hard disk.

Suppose the average number of flaws ( $\lambda$ ) increases to four. What is the probability of finding fewer than five flaws on a hard drive?

```
probability = poisscdf(4,4)
```

```
probability =  
    0.6288
```

This means that this faulty manufacturing process continues to operate after this first inspection almost 63% of the time.

**See Also**

[cdf](#) | [poisspdf](#) | [poissinv](#) | [poisstat](#) | [poissfit](#) | [poissrnd](#)

**How To**

- “Poisson Distribution” on page B-118

# poissfit

---

**Purpose** Poisson parameter estimates

**Syntax**  
`lambdahat = poissfit(data)`  
`[lambdahat,lambdaci] = poissfit(data)`  
`[lambdahat,lambdaci] = poissfit(data,alpha)`

**Description** `lambdahat = poissfit(data)` returns the maximum likelihood estimate (MLE) of the parameter of the Poisson distribution,  $\lambda$ , given the data `data`.

`[lambdahat,lambdaci] = poissfit(data)` also gives 95% confidence intervals in `lambdaci`.

`[lambdahat,lambdaci] = poissfit(data,alpha)` gives  $100(1 - \alpha)\%$  confidence intervals. For example `alpha = 0.001` yields 99.9% confidence intervals.

The sample mean is the MLE of  $\lambda$ .

$$\hat{\lambda} = \frac{1}{n} \sum_{i=1}^n x_i$$

**Examples**

```
r = poissrnd(5,10,2);  
[l,lci] = poissfit(r)  
l =  
    7.4000    6.3000  
lci =  
    5.8000    4.8000  
    9.1000    7.9000
```

**See Also** `mle` | `poisspdf` | `poisscdf` | `poissinv` | `poisstat` | `poissrnd`

**How To** • “Poisson Distribution” on page B-118



**Purpose** Poisson inverse cumulative distribution function

**Syntax** `X = poissinv(P,lambda)`

**Description** `X = poissinv(P,lambda)` returns the smallest value  $X$  such that the Poisson cdf evaluated at  $X$  equals or exceeds  $P$ , using mean parameters in `lambda`. `P` and `lambda` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input.

**Examples** If the average number of defects ( $\lambda$ ) is two, what is the 95th percentile of the number of defects?

```
poissinv(0.95,2)
ans =
     5
```

What is the median number of defects?

```
median_defects = poissinv(0.50,2)
median_defects =
     2
```

**See Also** `icdf` | `poisscdf` | `poisspdf` | `poisstat` | `poissfit` | `poissrnd`

**How To** • “Poisson Distribution” on page B-118

# prob.PoissonDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Poisson probability distribution object

**Description** `prob.PoissonDistribution` is an object consisting of parameters, a model description, and sample data for a Poisson probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Poisson')` creates a Poisson probability distribution object using the default parameter values.

`pd = makedist('Poisson','lambda',lambda)` creates a Poisson distribution object using the specified parameter value.

## Input Arguments

### lambda - Mean

1 (default) | nonnegative scalar value

Mean of the Poisson distribution, specified as a nonnegative scalar value.

### Data Types

single | double

## Properties

### lambda

Mean of the Poisson distribution, stored as a nonnegative scalar value.

### Data Types

single | double

### DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

## InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the

# prob.PoissonDistribution

---

estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters

# prob.PoissonDistribution

---

proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions **Poisson Distribution**

The Poisson distribution is appropriate for applications that involve counting the number of times a random event occurs in a given amount of time, distance, area, etc. If the number of counts follows the Poisson distribution, then the interval between individual counts follows the exponential distribution.

The Poisson distribution uses the following parameters.

Parameter	Description	Support
lambda	Mean	$\lambda \geq 0$

The probability density function of the Poisson distribution is

$$f(x | \lambda) = \frac{\lambda^x}{x!} e^{-\lambda} \quad ; \quad x = 0, 1, 2, \dots, \infty .$$

## Examples

### Create a Poisson Distribution Object Using Default Parameters

Create a Poisson distribution object using the default parameter values.

```
pd = makedist('Poisson')
```

```
pd =
```

```
PoissonDistribution
```

```
Poisson distribution  
lambda = 1
```

## Create a Poisson Distribution Object Using Specified Parameters

Create a Poisson distribution object by specifying the parameter values.

```
pd = makedist('Poisson','lambda',5)
```

```
pd =
```

```
PoissonDistribution
```

```
Poisson distribution  
lambda = 5
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =
```

```
5
```

For the Poisson distribution, both the mean and variance are equal to the parameter lambda.

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Poisson Distribution” on page B-118
- Class Attributes
- Property Attributes

# poisspdf

---

**Purpose** Poisson probability density function

**Syntax** `Y = poisspdf(X,lambda)`

**Description** `Y = poisspdf(X,lambda)` computes the Poisson pdf at each of the values in `X` using mean parameters in `lambda`. `X` and `lambda` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other input. The parameters in `lambda` must all be positive.

The Poisson pdf is

$$f(x | \lambda) = \frac{\lambda^x}{x!} e^{-\lambda}; x = 0, 1, 2, \dots, \infty.$$

The density function is zero unless  $x$  is an integer.

## Examples

A computer hard disk manufacturer has observed that flaws occur randomly in the manufacturing process at the average rate of two flaws in a 4 GB hard disk and has found this rate to be acceptable. What is the probability that a disk will be manufactured with no defects?

In this problem,  $\lambda = 2$  and  $x = 0$ .

```
p = poisspdf(0,2)
p =
    0.1353
```

## See Also

`pdf` | `poisscdf` | `poissinv` | `poisstat` | `poissfit` | `poissrnd`

## How To

- “Poisson Distribution” on page B-118



<b>Purpose</b>	Poisson random numbers
<b>Syntax</b>	<pre>R = poissrnd(lambda) R = poissrnd(lambda,m,n,...) R = poissrnd(lambda,[m,n,...])</pre>
<b>Description</b>	<p><code>R = poissrnd(lambda)</code> generates random numbers from the Poisson distribution with mean parameter <code>lambda</code>. <code>lambda</code> can be a vector, a matrix, or a multidimensional array. The size of <code>R</code> is the size of <code>lambda</code>.</p> <p><code>R = poissrnd(lambda,m,n,...)</code> or <code>R = poissrnd(lambda,[m,n,...])</code> generates an <code>m</code>-by-<code>n</code>-by-... array. The <code>lambda</code> parameter can be a scalar or an array of the same size as <code>R</code>.</p>
<b>Examples</b>	<p>Generate a random sample of 10 pseudo-observations from a Poisson distribution with <math>\lambda = 2</math>.</p> <pre>lambda = 2;  random_sample1 = poissrnd(lambda,1,10) random_sample1 =     1    0    1    2    1    3    4    2    0    0  random_sample2 = poissrnd(lambda,[1 10]) random_sample2 =     1    1    1    5    0    3    2    2    3    4  random_sample3 = poissrnd(lambda(ones(1,10))) random_sample3 =     3    2    1    1    0    0    4    0    2    0</pre>
<b>See Also</b>	<code>random</code>   <code>poisspdf</code>   <code>poisscdf</code>   <code>poissinv</code>   <code>poisstat</code>   <code>poissfit</code>
<b>How To</b>	• “Poisson Distribution” on page B-118

# poisstat

---

**Purpose** Poisson mean and variance

**Syntax** `M = poisstat(lambda)`  
`[M,V] = poisstat(lambda)`

**Description** `M = poisstat(lambda)` returns the mean of the Poisson distribution using mean parameters in `lambda`. The size of `M` is the size of `lambda`.  
`[M,V] = poisstat(lambda)` also returns the variance `V` of the Poisson distribution.  
For the Poisson distribution with parameter  $\lambda$ , both the mean and variance are equal to  $\lambda$ .

**Examples** Find the mean and variance for the Poisson distribution with  $\lambda = 2$ .

```
[m,v] = poisstat([1 2; 3 4])  
m =  
    1    2  
    3    4  
v =  
    1    2  
    3    4
```

**See Also** `poisspdf` | `poisscdf` | `poissinv` | `poissfit` | `poissrnd`

**How To** • “Poisson Distribution” on page B-118

**Purpose** Polynomial confidence intervals

**Syntax**

```
Y = polyconf(p,X)
[Y,DELTA] = polyconf(p,X,S)
[Y,DELTA] = polyconf(p,X,S,param1,val1,param2,val2,...)
```

**Description**

`Y = polyconf(p,X)` evaluates the polynomial `p` at the values in `X`. `p` is a vector of coefficients in descending powers.

`[Y,DELTA] = polyconf(p,X,S)` takes outputs `p` and `S` from `polyfit` and generates 95% prediction intervals `Y - DELTA` for new observations at the values in `X`.

`[Y,DELTA] = polyconf(p,X,S,param1,val1,param2,val2,...)` specifies optional parameter name/value pairs chosen from the following list.

Parameter	Value
'alpha'	A value between 0 and 1 specifying a confidence level of $100 * (1 - \text{alpha})\%$ . The default is 0.05.
'mu'	A two-element vector containing centering and scaling parameters. With this option, <code>polyconf</code> uses $(X - \text{mu}(1)) / \text{mu}(2)$ in place of <code>X</code> .
'predopt'	Either 'observation' (the default) to compute prediction intervals for new observations at the values in <code>X</code> , or 'curve' to compute confidence intervals for the fit evaluated at the values in <code>X</code> . See below.
'simopt'	Either 'off' (the default) for nonsimultaneous bounds, or 'on' for simultaneous bounds. See below.

The 'predopt' and 'simopt' parameters can be understood in terms of the following functions:

- $p(x)$  — the unknown mean function estimated by the fit
- $l(x)$  — the lower confidence bound

- $u(x)$  — the upper confidence bound

Suppose you make a new observation  $y_{n+1}$  at  $x_{n+1}$ , so that

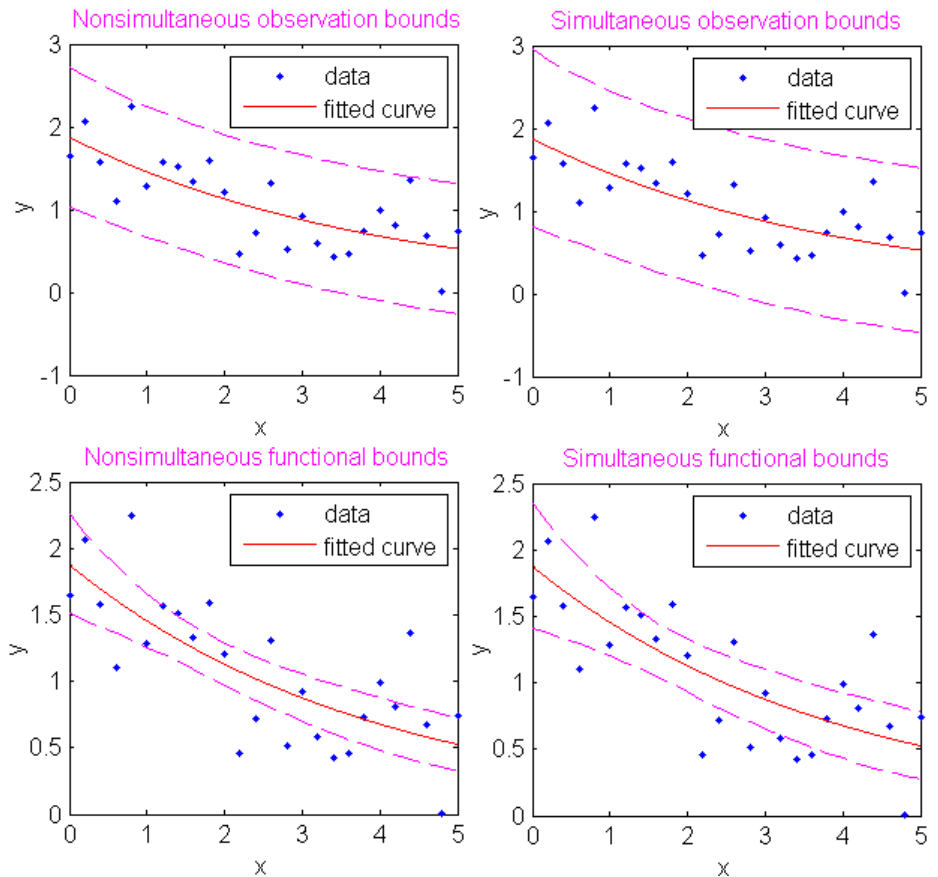
$$y_{n+1}(x_{n+1}) = p(x_{n+1}) + \varepsilon_{n+1}$$

By default, the interval  $[l_{n+1}(x_{n+1}), u_{n+1}(x_{n+1})]$  is a 95% confidence bound on  $y_{n+1}(x_{n+1})$ .

The following combinations of the 'predopt' and 'simopt' parameters allow you to specify other bounds.

'simopt'	'predopt'	Bounded Quantity
'off'	'observation'	$y_{n+1}(x_{n+1})$ (default)
'off'	'curve'	$p(x_{n+1})$
'on'	'observation'	$y_{n+1}(x)$ , for all $x$
'on'	'curve'	$p(x)$ , for all $x$

In general, 'observation' intervals are wider than 'curve' intervals, because of the additional uncertainty of predicting a new response value (the curve plus random errors). Likewise, simultaneous intervals are wider than nonsimultaneous intervals, because of the additional uncertainty of bounding values for all predictors  $x$ .



## Examples

This example uses code from the documentation example function `polydemo`, and calls the documentation example function `polystr` to convert the coefficient vector `p` into a string for the polynomial expression displayed in the figure title. It combines the functions `polyfit`, `polyval`, `roots`, and `polyconf` to produce a formatted display of data with a polynomial fit.

---

**Note** Statistics Toolbox documentation example files are located in the `\help\toolbox\stats\examples` subdirectory of your MATLAB root folder (`matlabroot`). This subdirectory is not on the MATLAB path at installation. To use the files in this subdirectory, either add the subdirectory to the MATLAB path (`addpath`) or make the subdirectory your current working folder (`cd`).

---

Display simulated data with a quadratic trend, a fitted quadratic polynomial, and 95% prediction intervals for new observations:

```
xdata = -5:5;
ydata = x.^2 - 5*x - 3 + 5*randn(size(x));

degree = 2; % Degree of the fit
alpha = 0.05; % Significance level

% Compute the fit and return the structure used by
% POLYCONF.
[p,S] = polyfit(xdata,ydata,degree);

% Compute the real roots and determine the extent of the
% data.
r = roots(p)'; % Roots as a row vector.
real_r = r(imag(r) == 0); % Real roots.

% Assure that the data are row vectors.
xdata = reshape(xdata,1,length(xdata));
ydata = reshape(ydata,1,length(ydata));

% Extent of the data.
mx = min([real_r,xdata]);
Mx = max([real_r,xdata]);
my = min([ydata,0]);
My = max([ydata,0]);
```

```

% Scale factors for plotting.
sx = 0.05*(Mx-mx);
sy = 0.05*(My-my);

% Plot the data, the fit, and the roots.
hdata = plot(xdata,ydata,'md','MarkerSize',5,...
    'LineWidth',2);
hold on
xfit = mx-sx:0.01:Mx+sx;
yfit = polyval(p,xfit);
hfit = plot(xfit,yfit,'b-','LineWidth',2);
hroots = plot(real_r,zeros(size(real_r)),...
    'bo','MarkerSize',5,...
    'LineWidth',2,...
    'MarkerFaceColor','b');

grid on
plot(xfit,zeros(size(xfit)),'k-','LineWidth',2)
axis([mx-sx Mx+sx my-sy My+sy])

% Add prediction intervals to the plot.
[Y,DELTA] = polyconf(p,xfit,S,'alpha',alpha);
hconf = plot(xfit,Y+DELTA,'b--');
plot(xfit,Y-DELTA,'b--')

% Display the polynomial fit and the real roots.
approx_p = round(100*p)/100; % Round for display.
htitle = title(['{\bf Fit:   }',...
    texlabel(polystr(approx_p))]);
set(htitle,'Color','b')
approx_real_r = round(100*real_r)/100; % Round for display.
hxlabel = xlabel(['{\bf Real Roots:   }',...
    num2str(approx_real_r)]);
set(hxlabel,'Color','b')

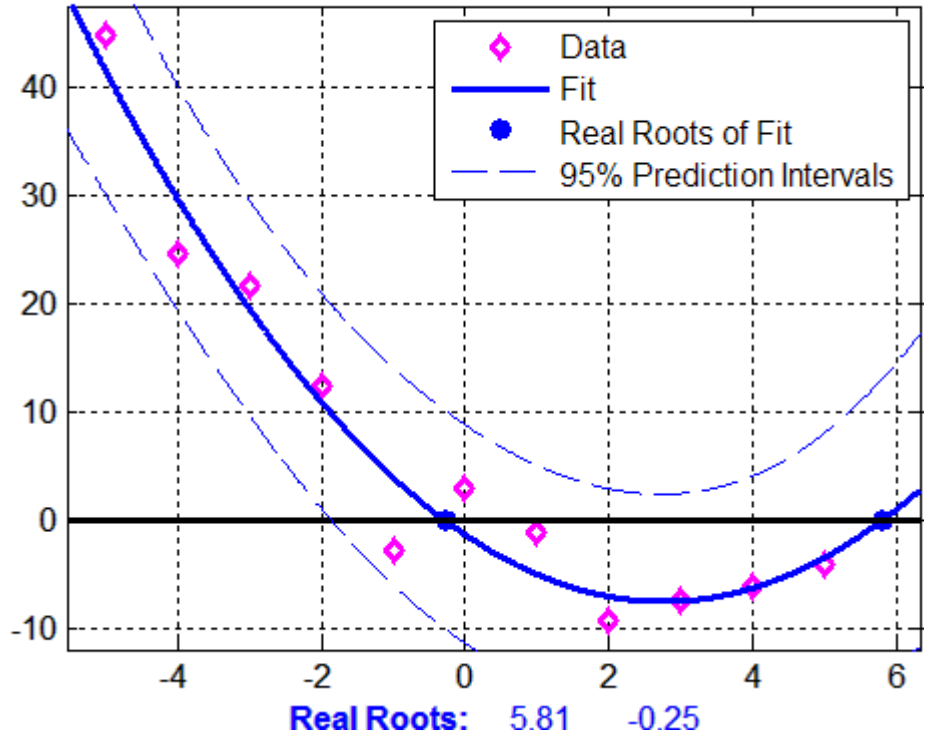
% Add a legend.
legend([hdata,hfit,hroots,hconf],...
    'Data','Fit','Real Roots of Fit',...

```

# polyconf

'95% Prediction Intervals')

**Fit:**  $0.81x^2 - 4.51x - 1.19$



## See Also

[polyfit](#) | [polyval](#) | [polytool](#)



**Purpose** Interactive polynomial fitting

**Syntax**

```
polytool
polytool(x,y)
polytool(x,y,n)
polytool(x,y,n,alpha)
polytool(x,y,n,alpha,xname,yname)
h = polytool(...)
```

**Description**

`polytool`

`polytool(x,y)` fits a line to the vectors `x` and `y` and displays an interactive plot of the result in a graphical interface. You can use the interface to explore the effects of changing the parameters of the fit and to export fit results to the workspace.

`polytool(x,y,n)` initially fits a polynomial of degree `n`. The default is 1, which produces a linear fit.

`polytool(x,y,n,alpha)` initially plots  $100(1 - \alpha)\%$  confidence intervals on the predicted values. The default is 0.05 which results in 95% confidence intervals.

`polytool(x,y,n,alpha,xname,yname)` labels the `x` and `y` values on the graphical interface using the strings `xname` and `yname`. Specify `n` and `alpha` as `[]` to use their default values.

`h = polytool(...)` outputs a vector of handles, `h`, to the line objects in the plot. The handles are returned in the degree: data, fit, lower bounds, upper bounds.

## Examples **Interactive polynomial fitting**

This example shows how to start an interactive fitting session with `polytool`.

Generate data from a quadratic curve with added noise.

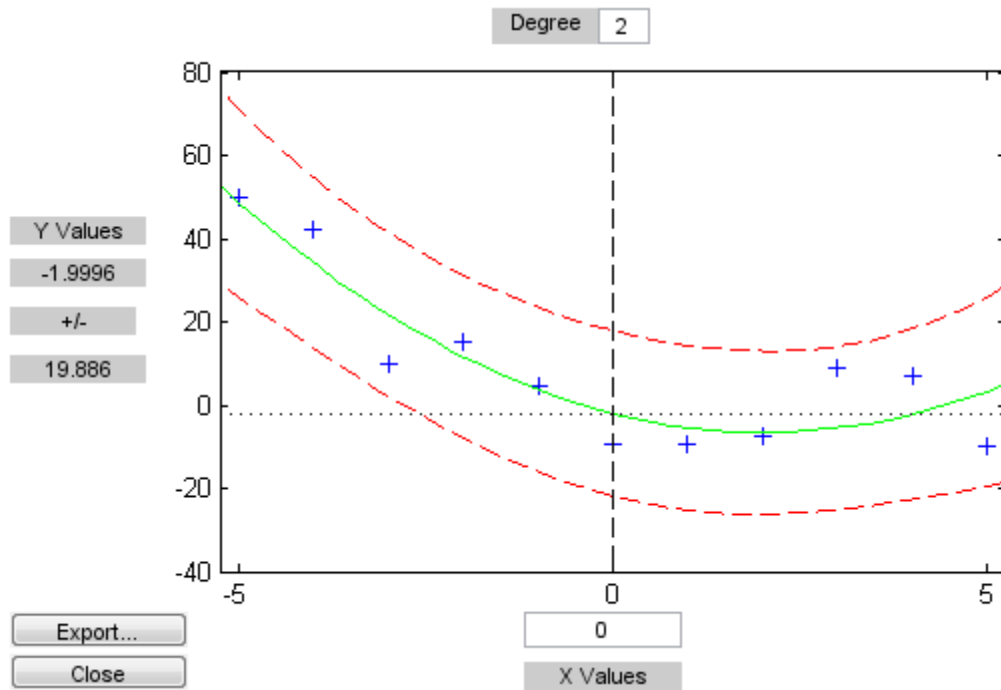
```
rng('default') % for reproducibility
x = -5:5;
```

# polytool

```
y = x.^2 - 5*x - 3 + 5*randn(size(x));
```

Fit a quadratic (degree-2) model with 0.90 confidence intervals.

```
n = 2;  
alpha = 0.1;  
polytool(x,y,n,alpha)
```



**See Also** [polyfit](#) | [polyval](#) | [polyconf](#) | [invpred](#)

## Purpose

Posterior probabilities of components

## Syntax

```
P = posterior(obj,X)
[P,nlogl] = posterior(obj,X)
```

## Description

`P = posterior(obj,X)` returns the posterior probabilities of each of the  $k$  components in the Gaussian mixture distribution defined by `obj` for each observation in the data matrix  $X$ .  $X$  is  $n$ -by- $d$ , where  $n$  is the number of observations and  $d$  is the dimension of the data. `obj` is an object created by `gmdistribution` or `fit`.  $P$  is  $n$ -by- $k$ , with  $P(I,J)$  the probability of component  $J$  given observation  $I$ .

`posterior` treats NaN values as missing data. Rows of  $X$  with NaN values are excluded from the computation.

`[P,nlogl] = posterior(obj,X)` also returns `nlogl`, the negative log-likelihood of the data.

## Examples

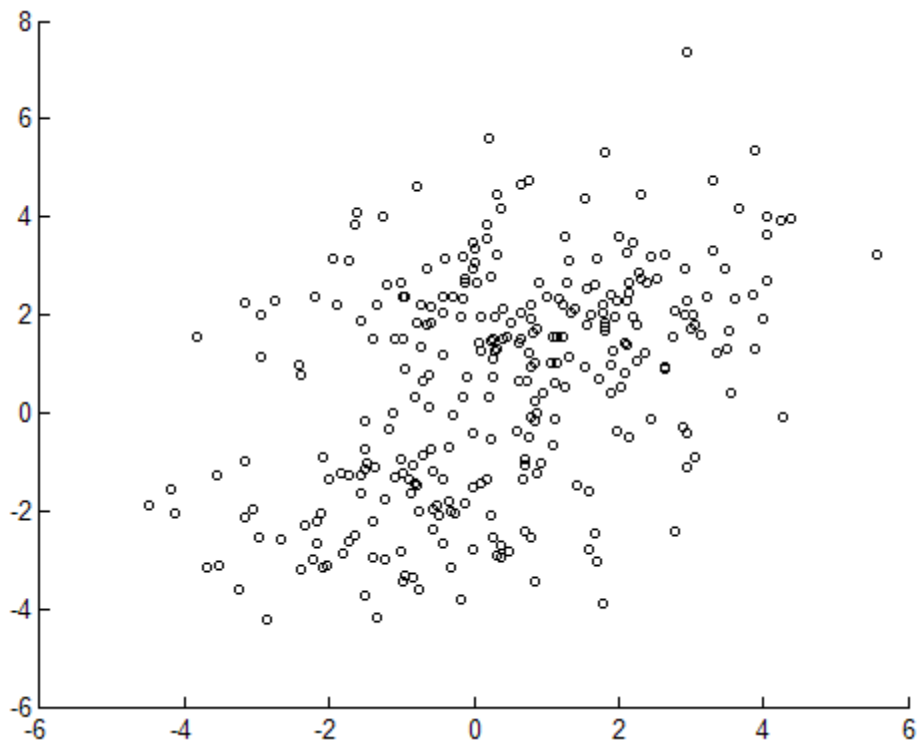
Generate data from a mixture of two bivariate Gaussian distributions using the `mvnrnd` function:

```
MU1 = [1 2];
SIGMA1 = [2 0; 0 .5];
MU2 = [-3 -5];
SIGMA2 = [1 0; 0 1];
X = [mvnrnd(MU1,SIGMA1,1000);mvnrnd(MU2,SIGMA2,1000)];

scatter(X(:,1),X(:,2),10,'.')
hold on
```

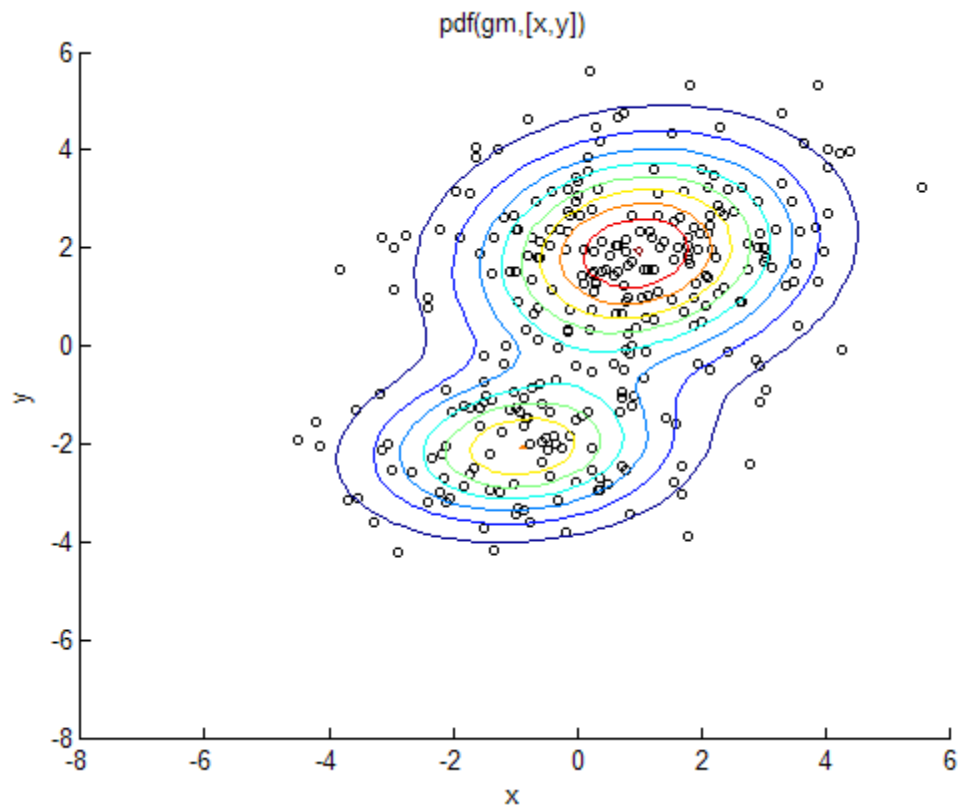
# gmdistribution.posterior

---



Fit a two-component Gaussian mixture model:

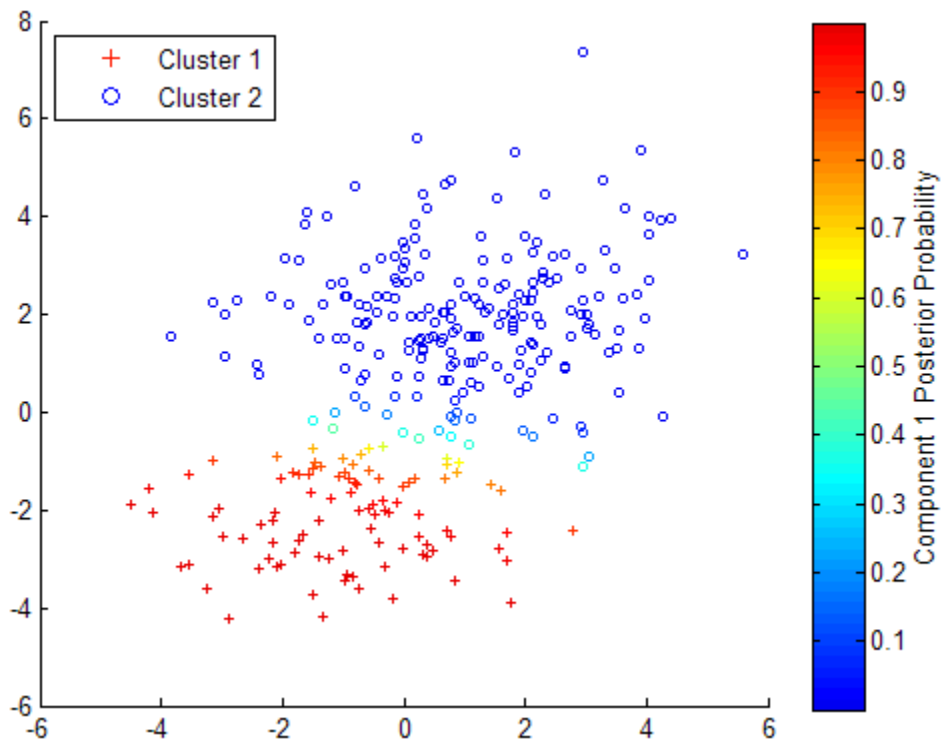
```
obj = gmdistribution.fit(X,2);  
h = ezcontour(@(x,y)pdf(obj,[x y]),[-8 6],[-8 6]);
```



Compute posterior probabilities of the components:

```
P = posterior(obj,X);  
  
delete(h)  
scatter(X(:,1),X(:,2),10,P(:,1),'o')  
hb = colorbar;  
ylabel(hb,'Component 1 Probability')
```

# gmdistribution.posterior



## See Also

`gmdistribution` | `fit` | `cluster` | `mahal`

## Purpose

Compute posterior probability of each class for test data

## Syntax

```
post = posterior(nb,test)
[post,cpre] = posterior(nb,test)
[post,cpre,logp] = posterior(nb,test)
[...] = posterior(..., 'HandleMissing',val)
```

## Description

`post = posterior(nb,test)` returns the posterior probability of the observations in `test` according to the `NaiveBayes` object `nb`. `test` is a `N-by-nb.ndims` matrix, where `N` is the number of observations in the test data. Rows of `test` correspond to points, columns of `test` correspond to features. `post` is a `N-by-nb.nclasses` matrix containing the posterior probability of each observation for each class. `post(i,j)` is the posterior probability of point `I` belonging to class `j`. Classes are ordered the same as `nb.clevels`, i.e., column `j` of `post` corresponds to the `j`th class in `nb.clevels`. The posterior probabilities corresponding to any empty classes are `NaN`.

`[post,cpre] = posterior(nb,test)` returns `cpre`, an `N-by-1` vector, containing the class to which each row of `test` has been assigned. `cpre` has the same type as `nb.CLevels`.

`[post,cpre,logp] = posterior(nb,test)` returns `logp`, an `N-by-1` vector containing estimates of the log of the probability density function (PDF). `logp(i)` is the log of the PDF of point `i`. The PDF value of point `i` is the sum of  $\text{Prob}(\text{point } I \mid \text{class } J) * \text{Pr}\{\text{class } J\}$  taken over all classes.

`[...] = posterior(..., 'HandleMissing',val)` specifies how `posterior` treats `NaN` (missing values). `val` can be one of the following:

# NaiveBayes.posterior

---

'off' (default)	Observations with NaN in any of the columns are not classified into any class. The corresponding rows in <code>post</code> and <code>logp</code> are NaN. The corresponding rows in <code>cpre</code> are NaN (if <code>obj.clevels</code> is numeric or logical), empty strings (if <code>obj.clevels</code> is char or cell array of strings) or (if <code>obj.clevels</code> is categorical).
'on'	For observations having NaN in some (but not all) columns, <code>post</code> and <code>cpre</code> are computed using the columns with non-NaN values. Corresponding <code>logp</code> values are NaN.

## See Also

NaiveBayes | fit | predict



---

<b>Purpose</b>	Percentiles of a data set
<b>Syntax</b>	$Y = \text{prctile}(X,p)$ $Y = \text{prctile}(X,p,\text{dim})$
<b>Description</b>	<p><math>Y = \text{prctile}(X,p)</math> returns percentiles of the values in a data vector or matrix <math>X</math> for the percentages <math>p</math> in the interval <math>[0,100]</math>.</p> <ul style="list-style-type: none"><li>• If <math>X</math> is a vector, then <math>Y</math> is a scalar or a vector with the same length as the number of percentiles required (<math>\text{length}(p)</math>). <math>Y(i)</math> contains the <math>p(i)</math> percentile.</li><li>• If <math>X</math> is a matrix, then <math>Y</math> is a row vector or a matrix, where the number of rows of <math>Y</math> is equal to the number of percentiles required (<math>\text{length}(p)</math>). The <math>i</math>th row of <math>Y</math> contains the <math>p(i)</math> percentiles of each column of <math>X</math>.</li><li>• For multidimensional arrays, <math>\text{prctile}</math> operates along the first nonsingleton dimension of <math>X</math>.</li></ul> <p><math>Y = \text{prctile}(X,p,\text{dim})</math> returns percentiles along dimension <math>\text{dim}</math>.</p>
<b>Input Arguments</b>	<p><b>X - Input data</b> vector   array</p> <p>Input data, specified as a vector or array.</p> <p><b>Data Types</b> double   single</p> <p><b>p - Percentages</b> scalar   vector</p> <p>Percentages for which to compute percentiles, returned as a scalar or vector of scalars from 0 to 100.</p> <p><b>Example:</b> 25</p> <p><b>Example:</b> [25, 50, 75]</p>

## Data Types

double | single

## dim - Dimension

1 (default) | positive integer

Dimension along which the percentiles of  $X$  are required, specified as a positive integer. For example, for a matrix  $X$ , when  $\text{dim} = 1$ , `prctile` returns the quantile(s) of the columns of  $X$  and when  $\text{dim} = 2$ , `prctile` returns the quantile(s) of the rows of  $X$ . For a multidimensional array  $X$ , the length of the  $\text{dim}$ th dimension of  $Y$  is equal to the length of  $p$ .

## Data Types

double

## Output Arguments

### Y - Percentiles

scalar | array

Percentiles of a data vector or array, specified as a scalar or array for one or more percentage values.

- If  $X$  is a vector, then  $Y$  is a scalar or a vector with the same length as the number of percentiles required (`length(p)`).  $Y(i)$  contains the  $p(i)$ th percentile.
- If  $X$  is a matrix, then  $Y$  is a vector or a matrix with the length of the  $\text{dim}$ th dimension equal to the number percentiles required (`length(p)`). When  $\text{dim} = 1$ , for example, the  $i$ th row of  $Y$  contains the  $p(i)$ th percentiles of columns of  $X$ .
- If  $X$  is an array of dimension  $d$ , then  $Y$  is an array with the length of the  $\text{dim}$ th dimension equal to the number of percentiles required (`length(p)`).

## Definitions

### Multidimensional Array

A *multidimensional array* is an array with more than two dimensions. For example, if  $X$  is a 1-by-3-by-4 array, then  $X$  is a 3-D array.

## Nonsingleton Dimension

A *first nonsingleton dimension* is the first dimension of an array whose size is not equal to 1. For example, if  $X$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $X$ .

## Linear Interpolation

Linear interpolation uses linear polynomials to find  $y_i = f(x_i)$ , the values of the underlying function  $Y = f(X)$  at the points in the vector or array  $x$ . Given the data points  $(x_1, y_1)$  and  $(x_2, y_2)$ , where  $y_1 = f(x_1)$  and  $y_2 = f(x_2)$ , linear interpolation finds  $y = f(x)$  for a given  $x$  between  $x_1$  and  $x_2$  as follows:

$$y = f(x) = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1).$$

Similarly, if the  $100(1.5/n)$ th percentile is  $y_{1.5/n}$  and the  $100(2.5/n)$ th percentile is  $y_{2.5/n}$ , then linear interpolation finds the  $100(2.3/n)$ th percentile,  $y_{2.3/n}$  as:

$$\frac{y_{2.3}}{n} = \frac{y_{1.5}}{n} + \frac{\left(\frac{2.3}{n} - \frac{1.5}{n}\right)}{\left(\frac{2.5}{n} - \frac{1.5}{n}\right)} \left(\frac{y_{2.5}}{n} - \frac{y_{1.5}}{n}\right).$$

## Algorithms

For an  $n$ -element vector  $X$ , `prctile` returns percentiles as follows:

- 1 The sorted values in  $X$  are taken as the  $100(0.5/n)$ th,  $100(1.5/n)$ th, ...,  $100([n - 0.5]/n)$ th percentiles. For example:
  - For a data vector of five elements such as  $\{6, 3, 2, 10, 1\}$ , the sorted elements  $\{1, 2, 3, 6, 10\}$  respectively correspond to the 10th, 30th, 50th, 70th, and 90th percentiles.
  - For a data vector of six elements such as  $\{6, 3, 2, 10, 8, 1\}$ , the sorted elements  $\{1, 2, 3, 6, 8, 10\}$  respectively correspond to the

(50/6)th, (150/6)th, (250/6)th, (350/6)th, (450/6)th, and (550/6)th percentiles.

- 2** `prctile` uses linear interpolation to compute percentiles for percentages between  $100(0.5/n)$  and  $100([n - 0.5]/n)$ .
- 3** `prctile` assigns the minimum or maximum values in `X` to the percentiles corresponding to the percentages outside that range.

`prctile` treats NaNs as missing values and removes them.

## Examples

### Percentiles of a Data Vector

Generate a data set of size 10.

```
rng('default'); % for reproducibility
x = normrnd(5,2,1,10)
```

```
x =
    6.0753    8.6678    0.4823    6.7243    5.6375    2.3846    4.1328
```

Calculate the 42nd percentile.

```
Y = prctile(X,42)
```

```
Y =
    5.6709
```

### Percentiles of a Data Matrix

Calculate the percentiles along the columns and rows of a data matrix for specified percentages.

Generate a 5-by-5 data matrix.

```
X = (1:5)'*(2:6)
```

```
X =
```

2	3	4	5	6
4	6	8	10	12
6	9	12	15	18
8	12	16	20	24
10	15	20	25	30

Calculate the 25th, 50th, and 75th percentiles along the columns of X.

```
Y = prctile(X,[25 50 75],1)
```

```
Y =
    3.5000    5.2500    7.0000    8.7500   10.5000
    6.0000    9.0000   12.0000   15.0000   18.0000
    8.5000   12.7500   17.0000   21.2500   25.5000
```

The rows of Y correspond to the percentiles of columns of X. For example, the 25th, 50th, and 75th percentiles of the third column of X with elements (4, 8, 12, 16, 20) are 7, 12, and 17, respectively. `Y = prctile(X,[25 50 75])` returns the same percentile matrix.

Calculate the 25th, 50th, and 75th percentiles along the rows of X.

```
Y = prctile(X,[25 50 75],2)
```

```
Y =
    2.7500    4.0000    5.2500
    5.5000    8.0000   10.5000
    8.2500   12.0000   15.7500
   11.0000   16.0000   21.0000
   13.7500   20.0000   26.2500
```

The rows of Y correspond to the percentiles of rows of X. For example, the 25th, 50th, and 75th percentiles of the first row of X with elements (2, 3, 4, 5, 6) are 2.75, 4, and 5.25, respectively.

## See Also

[quantile](#) | [median](#) | [iqr](#)

# prctile

---

## **Concepts**

- “Quantiles and Percentiles” on page 3-7

<b>Purpose</b>	Predict $k$ -nearest neighbor classification
<b>Syntax</b>	<pre>label = predict(md1,Xnew) [label,score] = predict(md1,Xnew) [label,score,cost] = predict(md1,Xnew)</pre>
<b>Description</b>	<p><code>label = predict(md1,Xnew)</code> returns a vector of predicted class labels for a matrix <code>Xnew</code>, based on <code>md1</code>, a <code>ClassificationKNN</code> model.</p> <p><code>[label,score] = predict(md1,Xnew)</code> returns a matrix of scores, indicating the likelihood that a label comes from a particular class.</p> <p><code>[label,score,cost] = predict(md1,Xnew)</code> returns a matrix of costs; <code>label</code> is the vector of minimal costs for each row of <code>cost</code></p>
<b>Input Arguments</b>	<p><b>md1</b></p> <p><math>k</math>-nearest neighbor classifier, created by <code>ClassificationKNN.fit</code>.</p> <p><b>Xnew</b></p> <p>Points at which <code>md1</code> predicts classifications. Each row of <code>Xnew</code> is one point. The number of columns in <code>Xnew</code> must equal the number of predictors in <code>md1</code>.</p>
<b>Output Arguments</b>	<p><b>label</b></p> <p>Predicted class labels for the points in <code>Xnew</code>, a vector with length equal to the number of rows of <code>Xnew</code>. The label is the class with minimal expected cost. See “Predicted Class Label” on page 21-2608.</p> <p><b>score</b></p> <p>Numeric matrix of size N-by-K, where N is the number of observations (rows) in <code>Xnew</code>, and K is the number of classes (in <code>md1.ClassNames</code>). <code>score(i,j)</code> is the posterior probability that row <code>i</code> of <code>Xnew</code> is of class <code>j</code>. See “Posterior Probability” on page 21-2608.</p> <p><b>cost</b></p>

# ClassificationKNN.predict

---

Matrix of expected costs of size N-by-K, where N is the number of observations (rows) in  $X_{\text{new}}$ , and K is the number of classes (in  $\text{mdl.ClassNames}$ ).  $\text{cost}(i, j)$  is the cost of classifying row  $i$  of  $X$  as class  $j$ . See “Expected Cost” on page 21-2609.

## Definitions

### Predicted Class Label

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1, \dots, K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

### Posterior Probability

For a vector (single query point)  $X_{\text{new}}$  and model  $\text{mdl}$ , let:

- $K$  be the number of nearest neighbors used in prediction,  $\text{mdl.NumNeighbors}$
- $\text{nbd}(\text{mdl}, X_{\text{new}})$  be the  $K$  nearest neighbors to  $X_{\text{new}}$  in  $\text{mdl.X}$
- $Y(\text{nbd})$  be the classifications of the points in  $\text{nbd}(\text{mdl}, X_{\text{new}})$ , namely  $\text{mdl.Y}(\text{nbd})$
- $W(\text{nbd})$  be the weights of the points in  $\text{nbd}(\text{mdl}, X_{\text{new}})$
- $\text{prior}$  be the priors of the classes in  $\text{mdl.Y}$



If there is a vector of prior probabilities, then the observation weights  $W$  are normalized by class to sum to the priors. This might involve a calculation for the point  $X_{\text{new}}$ , because weights can depend on the distance from  $X_{\text{new}}$  to the points in  $\text{mdl.X}$ .

The posterior probability  $p(j | X_{\text{new}})$  is

$$p(j | X_{\text{new}}) = \frac{\sum_{i \in \text{nd}} W(i) 1_{Y(X(i)=j)}}{\sum_{i \in \text{nd}} W(i)}.$$

Here,  $1_{Y(X(i)=j)}$  means 1 when  $\text{mdl.Y}(i) = j$ , and 0 otherwise.

## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `ClassificationKNN.fit`. `Cost(i,j)` is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default, `Cost(i,j)=1` if  $i \neq j$ , and `Cost(i,j)=0` if  $i=j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have `K` classes. You place the observations into a matrix `Xnew` with one observation per row. The command

```
[label,score,cost] = predict(mdl,Xnew)
```

# ClassificationKNN.predict

---

returns, among other outputs, a `cost` matrix of size `Nobs-by-K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{new}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new}(n))$  is the posterior probability of class  $i$  for observation  $X_{new}(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## Examples

### KNN Classification Predictions

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ . Evaluate some model predictions on new data.

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',5);
```

Predict the classifications for flowers with minimum, mean, and maximum characteristics.

```
Xnew = [min(X);mean(X);max(X)];
[label,score,cost] = predict(mdl,Xnew)
```

```
label =
  'setosa'
  'versicolor'
  'virginica'

score =
  1    0    0
  0    1    0
  0    0    1

cost =
  0    1    1
  1    0    1
  1    1    0
```

The classifications have binary values for the score and cost matrices, meaning all five nearest neighbors of each of the three points have identical classifications.

**See Also** [ClassificationKNN |](#)

## Related Examples

- “Predict Classification Based on a KNN Classifier” on page 16-28

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9

# CompactClassificationDiscriminant.predict

---

<b>Purpose</b>	Predict classification
<b>Syntax</b>	<pre>label = predict(obj,X) [label,score] = predict(obj,X) [label,score,cost] = predict(obj,X)</pre>
<b>Description</b>	<p><code>label = predict(obj,X)</code> returns a vector of predicted class labels for a matrix <code>X</code>, based on <code>obj</code>, a trained full or compact classifier.</p> <p><code>[label,score] = predict(obj,X)</code> returns a matrix of scores (posterior probabilities).</p> <p><code>[label,score,cost] = predict(obj,X)</code> returns a matrix of costs; <code>label</code> is the vector of minimal costs for each row of <code>cost</code>.</p>
<b>Input Arguments</b>	<p><b>obj</b></p> <p>Discriminant analysis classifier of class <code>ClassificationDiscriminant</code> or <code>CompactClassificationDiscriminant</code>, typically constructed with <code>ClassificationDiscriminant.fit</code>.</p> <p><b>X</b></p> <p>Matrix where each row represents an observation, and each column represents a predictor. The number of columns in <code>X</code> must equal the number of predictors in <code>obj</code>.</p>
<b>Output Arguments</b>	<p><b>label</b></p> <p>Vector of class labels of the same type as the response data used in training <code>obj</code>. Each entry of <code>labels</code> corresponds to a predicted class label for the corresponding row of <code>X</code>; see “Predicted Class Label” on page 21-2614.</p> <p><b>score</b></p> <p>Numeric matrix of size N-by-K, where N is the number of observations (rows) in <code>X</code>, and K is the number of classes (in</p>

obj.ClassNames). score(i, j) is the posterior probability that row i of X is of class j; see "Posterior Probability" on page 21-2613.

## cost

Matrix of expected costs of size N-by-K. cost(i, j) is the cost of classifying row i of X as class j. See "Cost" on page 21-2614.

## Definitions

### Posterior Probability

The posterior probability that a point  $z$  belongs to class  $j$  is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_j$  and covariance  $\Sigma_j$  at a point  $z$  is

$$P(x | k) = \frac{1}{(2\pi |\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k | x) = \frac{P(x | k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, the sum over  $k$  of  $P(x | k)P(k)$ .

### Prior Probability

The prior probability is one of three choices:

- 'uniform' — The prior probability of class  $k$  is one over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.

# CompactClassificationDiscriminant.predict

---

- Custom — The prior probability of class  $k$  is the  $k$ th element of the prior vector. See `ClassificationDiscriminant.fit`.

After creating a classifier `obj`, you can set the prior by dot addressing:

```
obj.Prior = v;
```

where  $v$  is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

## Cost

The matrix of expected costs per observation is defined in “Cost” on page 15-8.

## Predicted Class Label

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1,\dots,K} \sum_{k=1}^K \hat{P}(k|x) C(y|k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k|x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y|k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

## Examples

Examine predictions for a few rows in the Fisher iris data:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
X = meas(99:102,:); % take four rows
```

# CompactClassificationDiscriminant.predict

---

```
[label score cost] = predict(obj,X)
```

```
label =
```

```
    'versicolor'  
    'versicolor'  
    'virginica'  
    'virginica'
```

```
score =
```

```
    0.0000    1.0000    0.0000  
    0.0000    0.9999    0.0001  
    0.0000    0.0000    1.0000  
    0.0000    0.0011    0.9989
```

```
cost =
```

```
    1.0000    0.0000    1.0000  
    1.0000    0.0001    0.9999  
    1.0000    1.0000    0.0000  
    1.0000    0.9989    0.0011
```

## See Also

[ClassificationDiscriminant](#) | [ClassificationDiscriminant.fit](#) | [edge](#) | [loss](#) | [margin](#)

## How To

- “Discriminant Analysis” on page 15-3

# CompactClassificationEnsemble.predict

---

**Purpose** Predict classification

**Syntax**  
`labels = predict(ens,X)`  
`[labels,score] = predict(ens,X)`  
`[labels,...] = predict(ens,X,Name,Value)`

**Description** `labels = predict(ens,X)` returns a vector of predicted class labels for a matrix `X`, based on `ens`, a trained full or compact classification ensemble.  
`[labels,score] = predict(ens,X)` also returns scores for all classes.  
`[labels,...] = predict(ens,X,Name,Value)` predicts classifications with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A classification ensemble created by `fitensemble`, or a compact classification ensemble created by `compact`.

### **X**

A matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `ens`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'learners'**

Indices of weak learners `predict` uses for computation of responses, a numeric vector.



**Default:** 1:T, where T is the number of weak learners in ens

## 'UseObsForLearner'

A logical matrix of size N-by-T, where:

- N is the number of rows of X.
- T is the number of weak learners in ens.

When UseObsForLearner(i, j) is true, learner j is used in predicting the class of row i of X.

**Default:** true(N,T)

## Output Arguments

### labels

Vector of classification labels. labels has the same data type as the labels used in training ens.

### score

A matrix with one row per observation and one column per class. For each observation and each class, the score generated by each tree is the probability of this observation originating from this class computed as the fraction of observations of this class in a tree leaf. predict averages these scores over all trees in the ensemble.

## Definitions

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

# CompactClassificationEnsemble.predict

---

## Examples

Train a boosting ensemble for the ionosphere data, and predict the classification of the mean of the data:

```
load ionosphere;
ada = fitensemble(X,Y,'AdaBoostM1',100,'tree');
Xbar = mean(X);
[ypredict score] = predict(ada,Xbar)
```

```
ypredict =
    'g'
```

```
score =
    -2.9460    2.9460
```

## See Also

[margin](#) | [edge](#) | [loss](#)

## Purpose

Predict classification

## Syntax

```
label = predict(tree,X)
[label,score] = predict(tree,X)
[label,score,node] = predict(tree,X)
[label,score,node,cnum] = predict(tree,X)
[label,...] = predict(tree,X,Name,Value)
```

## Description

`label = predict(tree,X)` returns a vector of predicted class labels for a matrix `X`, based on `tree`, a trained full or compact classification tree.

`[label,score] = predict(tree,X)` returns a matrix of scores, indicating the likelihood that a label comes from a particular class.

`[label,score,node] = predict(tree,X)` returns a vector of predicted node numbers for the classification, based on `tree`.

`[label,score,node,cnum] = predict(tree,X)` returns a vector of predicted class number for the classification, based on `tree`.

`[label,...] = predict(tree,X,Name,Value)` returns labels with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **tree**

A classification tree created by `ClassificationTree.fit`, or a compact classification tree created by `compact`.

### **X**

A matrix where each row represents an observation, and each column represents a predictor. The number of columns in `X` must equal the number of predictors in `tree`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can

# CompactClassificationTree.predict

---

specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

## 'subtrees'

Numeric vector of pruning levels, with 0 representing the full, unpruned tree. To use the `subtrees` name-value pair, `tree` must include a pruning sequence as created by the `ClassificationTree.fit` or `prune` methods. If `subtrees` has `T` elements, and `X` has `N` rows, then `labels` is an `N-by-T` matrix. The `i`th column of `labels` contains the fitted values produced by the `subtrees(I)` subtree. Similarly, `score` is an `N-by-K-by-T` array, and `node` and `cnum` are `N-by-T` matrices. `subtrees` must be sorted in ascending order. (To compute fitted values for a tree that is not part of the optimal pruning sequence, first use `prune` to prune the tree.)

**Default:** 0

## Output Arguments

### label

Vector of class labels of the same type as the response data used in training `tree`. Each entry of `label` corresponds to the class with minimal expected cost for the corresponding row of `X`. See “Predicted Class Label” on page 21-2621.

### score

Numeric matrix of size `N-by-K`, where `N` is the number of observations (rows) in `X`, and `K` is the number of classes (in `tree.ClassNames`). `score(i, j)` is the posterior probability that row `i` of `X` is of class `j`.

### node

Numeric vector of node numbers for the predicted classes. Each entry corresponds to the predicted node in `tree` for the corresponding row of `X`.

### cnum

Numeric vector of class numbers corresponding to the predicted labels. Each entry of `cnum` corresponds to a predicted class number for the corresponding row of `X`.

## Definitions

### Predicted Class Label

`predict` classifies so as to minimize the expected classification cost:

$$\hat{y} = \arg \min_{y=1,\dots,K} \sum_{k=1}^K \hat{P}(k | x) C(y | k),$$

where

- $\hat{y}$  is the predicted classification.
- $K$  is the number of classes.
- $\hat{P}(k | x)$  is the posterior probability of class  $k$  for observation  $x$ .
- $C(y | k)$  is the cost of classifying an observation as  $y$  when its true class is  $k$ .

### Score (tree)

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

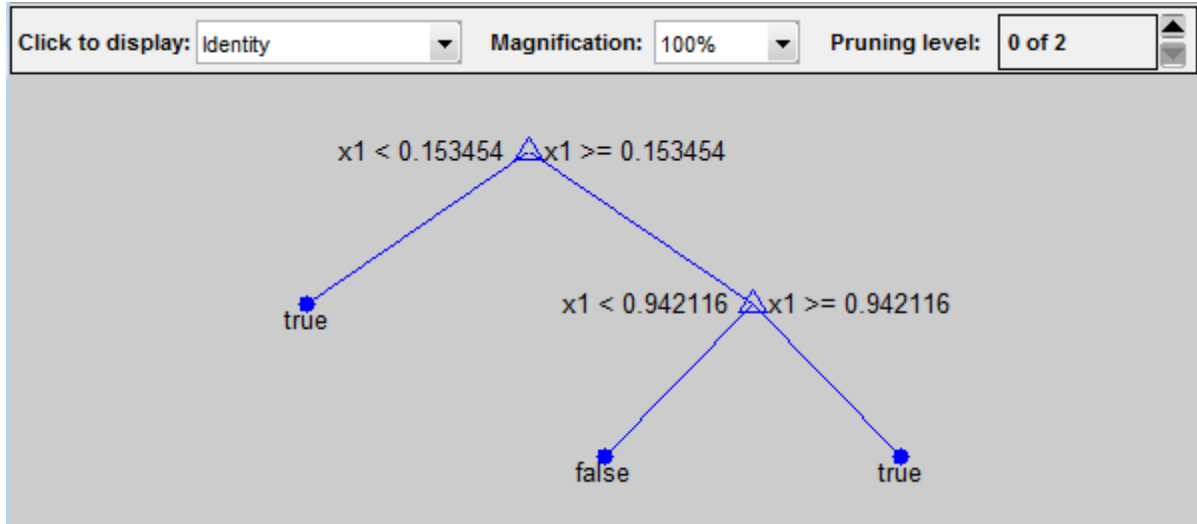
1

Generate 100 random points and classify them:

```
rng(0, 'twister') % for reproducibility
X = rand(100,1);
```

# CompactClassificationTree.predict

```
Y = (abs(X - .55) > .4);  
tree = ClassificationTree.fit(X,Y);  
view(tree,'mode','graph')
```

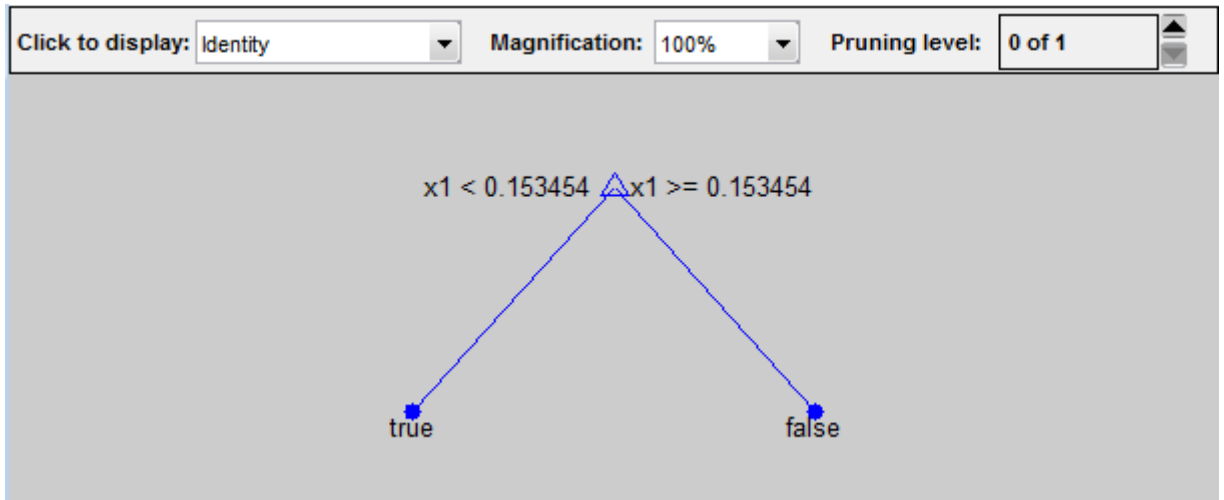


2

Prune the tree:

```
tree1 = prune(tree,'level',1);  
view(tree1,'mode','graph')
```

# CompactClassificationTree.predict



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
    0.9059    0.0941    0.8147  
    0.9059    0.0941    0.9058  
         0    1.0000    0.1270  
    0.9059    0.0941    0.9134  
    0.9059    0.0941    0.6324  
         0    1.0000    0.0975  
    0.9059    0.0941    0.2785
```

# CompactClassificationTree.predict

---

0.9059	0.0941	0.5469
0.9059	0.0941	0.9575
0.9059	0.0941	0.9649

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## True Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you create the classifier using the `ClassificationTree.fit` method.  $\text{Cost}(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $\text{Cost}(i, j)=1$  if  $i \neq j$ , and  $\text{Cost}(i, j)=0$  if  $i=j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

Suppose you have  $N_{\text{obs}}$  observations that you want to classify with a trained classifier. Suppose you have  $K$  classes. You place the observations into a matrix  $X_{\text{new}}$  with one observation per row.

The expected cost matrix  $CE$  has size  $N_{\text{obs}}$ -by- $K$ . Each row of  $CE$  contains the expected (average) cost of classifying the observation into each of the  $K$  classes.  $CE(n, k)$  is



$$\sum_{i=1}^K \hat{P}(i | X_{new}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new}(n))$  is the posterior probability of class  $i$  for observation  $X_{new}(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## Predictive Measure of Association

The predictive measure of association between the optimal split on variable  $i$  and a surrogate split on variable  $j$  is:

$$\lambda_{i,j} = \frac{\min(P_L, P_R) - (1 - P_{L_i L_j} - P_{R_i R_j})}{\min(P_L, P_R)}.$$

Here

- $P_L$  and  $P_R$  are the node probabilities for the optimal split of node  $i$  into Left and Right nodes respectively.
- $P_{L_i L_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Left.
- $P_{R_i R_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Right.

Clearly,  $\lambda_{i,j}$  lies from  $-\infty$  to 1. Variable  $j$  is a worthwhile surrogate split for variable  $i$  if  $\lambda_{i,j} > 0$ .

# CompactClassificationTree.predict

---

## Examples

Examine predictions for a few rows in the Fisher iris data:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
X = meas(99:102,:); % take four rows
[label score node cnum] = predict(tree,X)
```

```
label =
    'versicolor'
    'versicolor'
    'virginica'
    'virginica'
```

```
score =
    0    1.0000         0
    0    1.0000         0
    0    0.0217    0.9783
    0    0.0217    0.9783
```

```
node =
    8
    8
    5
    5
```

```
cnum =
    2
    2
    3
    3
```

---

Examine predictions from pruned trees for the Fisher iris model:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
X = meas(99:102,:); % taking four rows
```

# CompactClassificationTree.predict

```
[label score node cnum] = predict(tree,X,'subtrees',[2 3 4])
```

```
label =
```

```
    'versicolor'    'versicolor'    'setosa'  
    'versicolor'    'versicolor'    'setosa'  
    'virginica'     'versicolor'    'setosa'  
    'virginica'     'versicolor'    'setosa'
```

```
score(:, :, 1) =
```

```
    0    0.9074    0.0926  
    0    0.9074    0.0926  
    0    0.0217    0.9783  
    0    0.0217    0.9783
```

```
score(:, :, 2) =
```

```
    0    0.5000    0.5000  
    0    0.5000    0.5000  
    0    0.5000    0.5000  
    0    0.5000    0.5000
```

```
score(:, :, 3) =
```

```
    0.3333    0.3333    0.3333  
    0.3333    0.3333    0.3333  
    0.3333    0.3333    0.3333  
    0.3333    0.3333    0.3333
```

```
node =
```

```
    4    3    1  
    4    3    1  
    5    3    1  
    5    3    1
```

```
cnum =
```

```
    2    2    1  
    2    2    1  
    3    2    1  
    3    2    1
```

# CompactClassificationTree.predict

---

## Algorithms

`predict` generates predictions by following the branches of `tree` until it reaches a leaf node or a missing value. If `predict` reaches a leaf node, it returns the classification of that node.

If `predict` reaches a node with a missing value for a predictor, its behavior depends on the setting of the `Surrogate` name-value pair when `ClassificationTree.fit` constructs `tree`.

- **Surrogate = 'off'** (default) — `predict` returns the label with the largest number of training samples that reach the node.
- **Surrogate = 'on'** — `predict` uses the best surrogate split at the node. If all surrogate split variables with positive *predictive measure of association* are missing, `predict` returns the label with the largest number of training samples that reach the node. For a definition, see “Predictive Measure of Association” on page 21-2625.

## See Also

`ClassificationTree.fit` | `compact` | `prune` | `loss` | `edge` | `margin`

**Purpose**

Predict response of ensemble

**Syntax**

```
Yfit = predict(ens,Xdata)
Yfit = predict(ens,Xdata,Name,Value)
```

**Description**

`Yfit = predict(ens,Xdata)` returns predicted responses to the data in `Xdata`, based on the `ens` regression ensemble model.

`Yfit = predict(ens,Xdata,Name,Value)` predicts with additional options specified by one or more `Name,Value` pair arguments.

**Input Arguments****ens**

Regression ensemble created by `fitensemble`, or by the compact method.

**Xdata**

Numeric array with the same number of columns as the array used for creating `ens`. Each row of `Xdata` corresponds to one data point, and each column corresponds to one predictor.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'learners'**

Indices of weak learners in the ensemble ranging from 1 to `ens.NTrained`. `oobEdge` uses only these learners for calculating loss.

**Default:** `1:NTrained`

**'UseObsForLearner'**

# CompactRegressionEnsemble.predict

---

A logical matrix of size N-by-NTrained, where N is the number of observations in `ens.X`, and NTrained is the number of weak learners. When `UseObsForLearner(I,J)` is true, `predict` uses learner J in predicting observation I.

**Default:** `true(N,NTrained)`

## Output Arguments

### Yfit

A numeric column vector with the same number of rows as `Xdata`. Each row of `Yfit` gives the predicted response to the corresponding row of `Xdata`, based on the `ens` regression model.

## Examples

Find the predicted mileage for a four-cylinder car, with 200 cubic inch engine displacement, 150 horsepower, weighing 3000 lbs, based on the `carsmall` data:

```
load carsmall
X = [Cylinders Displacement Horsepower Weight];
rens = fitensemble(X,MPG,'LSBoost',100,'Tree');
Mileage = predict(rens,[4 200 150 3000])
```

```
Mileage =
    20.4982
```

## See Also

`loss` | `fitensemble`

## Purpose

Predict response of regression tree

## Syntax

```
Yfit = predict(tree,Xdata)
[Yfit,node] = predict(tree,Xdata)
[Yfit,node] = predict(tree,Xdata,Name,Value)
```

## Description

`Yfit = predict(tree,Xdata)` returns predicted responses to the data in `Xdata`, based on the tree regression tree.

`[Yfit,node] = predict(tree,Xdata)` returns the predicted node numbers of tree in response to `Xdata`.

`[Yfit,node] = predict(tree,Xdata,Name,Value)` predicts response with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **tree**

Regression tree created by `RegressionTree.fit`, or by the `compact` method.

### **Xdata**

Numeric array with the same number of columns as the array used for creating `tree`. Each row of `Xdata` corresponds to one data point, and each column corresponds to one predictor.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'subtrees'**

Numeric vector of pruning levels, with 0 representing the full, unpruned tree. To use the `subtrees` name-value pair, `tree` must include a pruning sequence as created by the

# CompactRegressionTree.predict

---

RegressionTree.fit or prune methods. If subtrees has T elements, and X has N rows, then Yfit is an N-by-T matrix. The ith column of Yfit contains the fitted values produced by the subtrees(I) subtree. Similarly, node is an N-by-T matrix. subtrees must be sorted in ascending order. (To compute fitted values for a tree that is not part of the optimal pruning sequence, first use prune to prune the tree.)

**Default:** 0

## Output Arguments

### Yfit

A numeric column vector with the same number of rows as Xdata. Each row of Yfit gives the predicted response to the corresponding row of Xdata, based on the tree regression model.

### node

Numeric vector of node numbers for the predictions. Each entry corresponds to the predicted leaf node in tree for the corresponding row of Xdata.

## Examples

Find the predicted mileage for a car with 200 cubic inch engine displacement, 150 horsepower, weighing 3000 lbs, based on the carsmall data:

```
load carsmall
X = [Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
Mileage = predict(tree,[200 150 3000])
```

```
Mileage =
    21.9375
```

## See Also

RegressionTree.fit | compact | loss



## Purpose

Predict response

## Syntax

```
YFIT = predict(B,X)
[YFIT,stdevs] = predict(B,X)
[YFIT,scores] = predict(B,X)
[YFIT,scores,stdevs] = predict(B,X)
Y = predict(B,X,'param1',val1,'param2',val2,...)
```

## Description

`YFIT = predict(B,X)` computes the predicted response of the trained ensemble `B` for predictors `X`. By default, `predict` takes a democratic (nonweighted) average vote from all trees in the ensemble. In `X`, rows represent observations and columns represent variables. `YFIT` is a cell array of strings for classification and a numeric array for regression.

For regression, `[YFIT,stdevs] = predict(B,X)` also returns standard deviations of the computed responses over the ensemble of the grown trees.

For classification, `[YFIT,scores] = predict(B,X)` returns scores for all classes. `scores` is a matrix with one row per observation and one column per class. For each observation and each class, the score generated by each tree is the probability of this observation originating from this class computed as the fraction of observations of this class in a tree leaf. `predict` averages these scores over all trees in the ensemble.

`[YFIT,scores,stdevs] = predict(B,X)` also returns standard deviations of the computed scores for classification. `stdevs` is a matrix with one row per observation and one column per class, with standard deviations taken over the ensemble of the grown trees.

`Y = predict(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

# CompactTreeBagger.predict

---

'trees'	Array of tree indices to use for computation of responses. Default is 'all'.
'treeweights'	Array of NTrees weights for weighting votes from the specified trees.
'useifort'	Logical matrix of size Nobs-by-NTrees indicating which trees to use to make predictions for each observation. By default all trees are used for all observations.

## See Also

[ClassificationTree.predict](#) | [RegressionTree.predict](#) | [TreeBagger.predict](#)

## Purpose

Predict response of generalized linear regression model

## Syntax

```
ypred = predict mdl, Xnew
[ypred, yci] = predict mdl, Xnew
[ypred, yci] = predict mdl, Xnew, Name, Value
```

## Description

`ypred = predict mdl, Xnew` returns the predicted response of the `mdl` generalized linear regression model to the points in `Xnew`.

`[ypred, yci] = predict mdl, Xnew` returns confidence intervals for the true mean responses.

`[ypred, yci] = predict mdl, Xnew, Name, Value` predicts responses with additional options specified by one or more `Name, Value` pair arguments.

## Tips

- For predictions with added noise, use `random`.
- For a syntax that can be easier to use with models created from dataset arrays, try `feval`.

## Input Arguments

### `mdl`

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

### `Xnew`

Points at which `mdl` predicts responses.

- If `Xnew` is a dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding

# GeneralizedLinearModel.predict

---

value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

## 'alpha'

Positive scalar from 0 to 1. Confidence level of  $y_{ci}$  is  $100(1 - \alpha)\%$ .

**Default:** 0.05, meaning a 95% confidence interval.

## 'BinomialSize'

Value of the binomial  $n$  parameter for each row in the training data. BinomialSize can be a vector the same length as  $X_{new}$ , or a scalar that applies to each row. The default value 1 produces  $y_{pred}$  values that are predicted proportions. Use BinomialSize only if `mdl` is fit to a binomial distribution.

**Default:** 1

## 'Offset'

Value of the offset for each row in  $X_{new}$ . Offset can be a vector the same length as  $X_{new}$ , or a scalar that applies to each row. The offset is used as an additional predictor with a coefficient value fixed at 1. In other words, if  $\mathbf{b}$  is the fitted coefficient vector, and  $\text{link}$  is the link function,

$$\text{link}(y_{\text{pred}}) = \text{Offset} + X_{\text{new}} * \mathbf{b}.$$

**Default:** `zeros(size(Xnew,1))`

## 'Simultaneous'

Logical value specifying whether the confidence bounds are for all predictor values simultaneously (`true`), or hold for each individual predictor value (`false`). Simultaneous bounds are wider than separate bounds, because it is more stringent to require that the

entire curve be within the bounds than to require that the curve at a single predictor value be within the bounds.

For details, see `polyconf`.

**Default:** `false`

## Output Arguments

### **ypred**

Vector of predicted mean values at `Xnew`.

### **yci**

Confidence intervals, a two-column matrix with each row providing one interval. The meaning of the confidence interval depends on the settings of the name-value pairs.

## Examples

### **Generalized Linear Model Predictions**

Create a generalized linear model, and predict its response to new data.

Generate artificial data for the model using Poisson random numbers with two underlying predictors `X(1)` and `X(2)`.

```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson');
```

Create points for prediction.

```
[Xtest1 Xtest2] = meshgrid(-1:.5:3,-2:.5:2);
Xnew = [Xtest1(:),Xtest2(:)];
```

Predict responses at the new points.

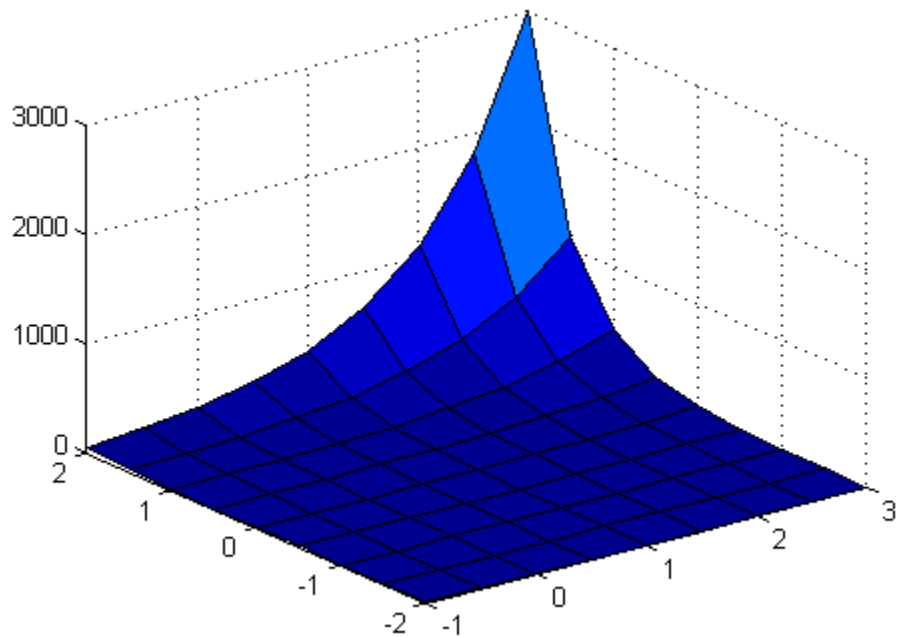
# GeneralizedLinearModel.predict

---

```
ypred = predict mdl, Xnew;
```

Plot the predictions.

```
surf(Xtest1, Xtest2, reshape(ypred, 9, 9))
```



Create confidence intervals on the predictions.

```
[ypred yci] = predict(mdl, Xnew);
```

## Alternatives

`feval` gives the same predictions, but uses separate input arrays for each predictor, instead of one input array containing all predictors.

`random` predicts with added noise.

**See Also**

GeneralizedLinearModel | random |

**Related Examples**

- “predict” on page 10-37
- “Generalized Linear Model Workflow” on page 10-42

**Concepts**

- “Generalized Linear Models” on page 10-12

# LinearModel.predict

---

**Purpose** Predict response of linear regression model

**Syntax**

```
ypred = predict mdl,Xnew
[ypred,yci] = predict mdl,Xnew
[ypred,yci] = predict mdl,Xnew,Name,Value)
```

**Description** `ypred = predict mdl,Xnew` returns the predicted response of the `mdl` linear regression model to the points in `Xnew`.

`[ypred,yci] = predict mdl,Xnew` returns confidence intervals for the true mean responses.

`[ypred,yci] = predict mdl,Xnew,Name,Value)` predicts responses with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For predictions with added noise, use `random`.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.



## 'alpha'

Positive scalar from 0 to 1. Confidence level of  $y_{ci}$  is  $100(1 - \alpha)\%$ .

**Default:** 0.05, meaning a 95% confidence interval.

## 'Prediction'

String specifying the type of prediction:

- 'curve' — predict predicts confidence bounds for the fitted mean values.
- 'observation' — predict predicts confidence bounds for the new observations. This results in wider bounds because the error in a new observation is equal to the error in the estimated mean value, plus the variability in the observation from the true mean.

For details, see `polyconf`.

**Default:** 'curve'

## 'Simultaneous'

Logical value specifying whether the confidence bounds are for all predictor values simultaneously (`true`), or hold for each individual predictor value (`false`). Simultaneous bounds are wider than separate bounds, because it is more stringent to require that the entire curve be within the bounds than to require that the curve at a single predictor value be within the bounds.

For details, see `polyconf`.

**Default:** `false`

## Output Arguments

### `ypred`

Vector of predicted mean values at `Xnew`.

# LinearModel.predict

---

**yci**

Confidence intervals, a two-column matrix with each row providing one interval. The meaning of the confidence interval depends on the settings of the name-value pairs.

## Examples

### Predict Response to Data

Create a model of car mileage as a function of weight, and predict the response.

Create a quadratic model of car mileage as a function of weight from the `carsmall` data.

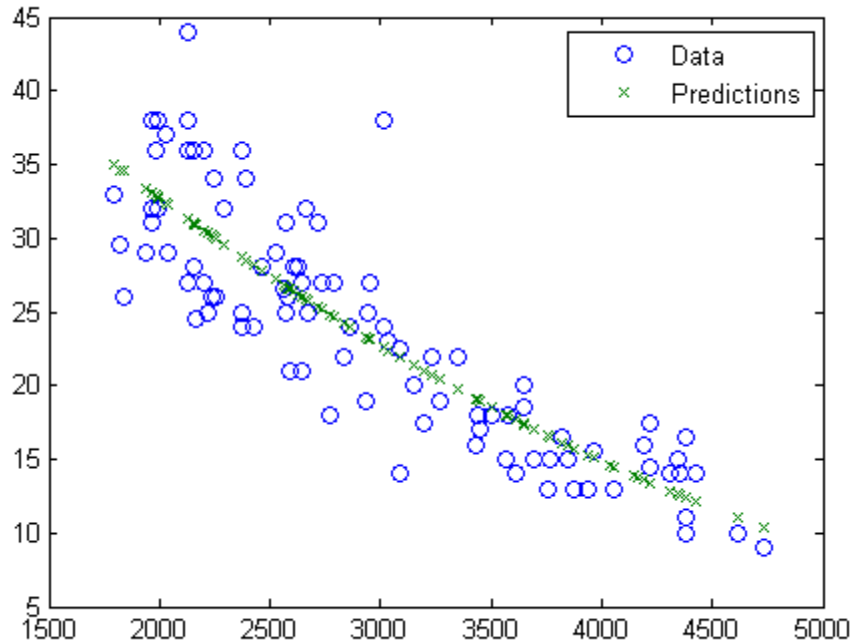
```
load carsmall
X = Weight;
y = MPG;
mdl = fitlm(X,y,'quadratic');
```

Create predicted responses to the data.

```
Xnew = X;
ypred = predict(mdl,Xnew);
```

Plot the original responses and the predicted responses to see how they differ.

```
plot(X,y,'o',Xnew,ypred,'x')
legend('Data','Predictions')
```



## Alternatives

`feval` gives the same predictions, but uses multiple input arrays with one component in each input argument. `feval` can be simpler to use with a model created from a dataset array `ds`. `feval` does not give confidence intervals on its predictions.

`random` predicts with added noise.

## See Also

`feval` | `LinearModel` | `fitlm` | `random` | `stepwiselm`

## Tutorials

- “predict” on page 9-40
- “Linear Regression Workflow” on page 9-44

# LinearModel.predict

---

## How To

- “Linear Regression” on page 9-12

**Purpose** Predict response of linear mixed-effects model

**Syntax**

```
ypred = predict(lme)
ypred = predict(lme,dsnew)
ypred = predict(lme,Xnew,Znew)
ypred = predict(lme,Xnew,Znew,Gnew)
ypred = predict(__,Name,Value)
[ypred,ypredCI] = predict(__)
[ypred,ypredCI,DF] = predict(__)
```

**Description** `ypred = predict(lme)` returns a vector of conditional predicted responses `ypred` at the original predictors used to fit the linear mixed-effects model `lme`.

`ypred = predict(lme,dsnew)` returns a vector of conditional predicted responses `ypred` from the fitted linear mixed-effects model `lme` at the values in the new dataset array `dsnew`. Use a dataset array for `predict` if you use a dataset array for fitting the model `lme`.

If a particular grouping variable in `dsnew` has levels that are not in the original data, then the random effects for that grouping variable do not contribute to the 'Conditional' prediction at observations where the grouping variable has new levels.

`ypred = predict(lme,Xnew,Znew)` returns a vector of conditional predicted responses `ypred` from the fitted linear mixed-effects model `lme` at the values in the new fixed- and random-effects design matrices, `Xnew` and `Znew`, respectively. `Znew` can also be a cell array of matrices. In this case, the grouping variable `G` is `ones(n,1)`, where  $n$  is the number of observations used in the fit.

Use the matrix format for `predict` if using design matrices for fitting the model `lme`.

`ypred = predict(lme,Xnew,Znew,Gnew)` returns a vector of conditional predicted responses `ypred` from the fitted linear

# LinearMixedModel.predict

---

mixed-effects model `lme` at the values in the new fixed- and random-effects design matrices, `Xnew` and `Znew`, respectively, and the grouping variable `Gnew`.

`Znew` and `Gnew` can also be cell arrays of matrices and grouping variables, respectively.

`ypred = predict( ___, Name, Value)` returns a vector of predicted responses `ypred` from the fitted linear mixed-effects model `lme` with additional options specified by one or more `Name, Value` pair arguments.

For example, you can specify the confidence level, simultaneous confidence bounds, or contributions from only fixed effects.

`[ypred,ypredCI] = predict( ___)` also returns confidence intervals `ypredCI` for the predictions `ypred` for any of the input arguments in the previous syntaxes.

`[ypred,ypredCI,DF] = predict( ___)` also returns the degrees of freedom `DF` used in computing the confidence intervals for any of the input arguments in the previous syntaxes.

## Input Arguments

### **lme** - Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **dsnew** - New input data

dataset array

New input data, which includes the response variable, predictor variables, and grouping variables, specified as a dataset array. The predictor variables can be continuous or grouping variables.

`dsnew` must have the same variables as in the original dataset array used to fit the linear mixed-effects model `lme`.

## Data Types

single | double | logical | char

## **Xnew - New fixed-effects design matrix**

*n*-by-*p* matrix

New fixed-effects design matrix, specified as an *n*-by-*p* matrix, where *n* is the number of observations and *p* is the number of fixed predictor variables. Each row of *X* corresponds to one observation and each column of *X* corresponds to one variable.

## Data Types

single | double

## **Znew - New random-effects design**

*n*-by-*q* matrix | cell array of length *R*

New random-effects design, specified as an *n*-by-*q* matrix or a cell array of *R* design matrices  $Z\{r\}$ , where  $r = 1, 2, \dots, R$ . If *Znew* is a cell array, then each  $Z\{r\}$  is an *n*-by- $q(r)$  matrix, where *n* is the number of observations, and  $q(r)$  is the number of random predictor variables.

## Data Types

single | double | logical | char | cell

## **Gnew - New grouping variable or variables**

vector | cell array of grouping variables of length *R*

New grouping variable or variables, specified as a vector or a cell array, of length *R*, of grouping variables with the same levels or groups as the original grouping variables used to fit the linear mixed-effects model `lme`.

## Data Types

single | double | logical | char | cell

# LinearMixedModel.predict

---

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## **'Alpha' - Confidence level**

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

**Example:** `'Alpha', 0.01`

## **Data Types**

single | double

## **'Conditional' - Indicator for conditional predictions**

True (default) | False

Indicator for conditional predictions, specified as the comma-separated pair consisting of `'Conditional'` and one of the following.

True	Contributions from both fixed effects and random effects (conditional)
False	Contribution from only fixed effects (marginal)

**Example:** `'Conditional', 'False'`

## **'DFMethod' - Method for computing approximate degrees of freedom**



'Residual' (default) | 'Satterthwaite' | 'None'

Method for computing approximate degrees of freedom to use in the confidence interval computation, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'Residual'	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'Satterthwaite'	Satterthwaite approximation.
'None'	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

**Example:** 'DFMethod', 'Satterthwaite'

## 'Simultaneous' - Type of confidence bounds

false (default) | true

Type of confidence bounds, specified as the comma-separated pair consisting of 'Simultaneous' and one of the following.

false	Default. Nonsimultaneous bounds.
true	Simultaneous bounds.

**Example:** 'Simultaneous', true

## 'Prediction' - Type of prediction

'curve' (default) | 'observation'

Type of prediction, specified as the comma-separated pair consisting of 'Prediction' and one of the following.

# LinearMixedModel.predict

---

'curve'	Default. Confidence bounds apply to the fitted function and do not include any variability associated with a new observation.
'observation'	Variability due to observation error for the new observations is included in the confidence bound calculations.

**Example:** 'Prediction', 'observation'

## Output Arguments

### **ypred - Predicted responses**

vector

Predicted responses, returned as a vector. `ypred` can contain the conditional or marginal responses, depending on the value choice of the 'Conditional' name-value pair argument. Conditional predictions include contributions from both fixed and random effects.

### **ypredCI - Point-wise confidence intervals**

two-column matrix

Point-wise confidence intervals for the predicted values, returned as a two-column matrix. The first column of `yCI` contains the lower bounds, and the second column contains the upper bound. By default, `yCI` contains the 95% confidence intervals for the predictions. You can change the confidence level using the `Alpha` name-value pair argument, make them simultaneous using the `Simultaneous` name-value pair argument, and also make them for a new observation rather than for the curve using the `Prediction` name-value pair argument.

### **DF - Degrees of freedom**

vector | scalar value

Degrees of freedom used in computing the confidence intervals, returned as a vector or a scalar value.

- If the 'Simultaneous' name-value pair argument is false, then DF is a vector.
- If the 'Simultaneous' name-value pair argument is true, then DF is a scalar value.

## Examples

### Predict Responses at the Original Design Values

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;
ds.Tomato = nominal(ds.Tomato);
ds.Soil = nominal(ds.Soil);
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)')
```

# LinearMixedModel.predict

---

Predict the response values at the original design values. Display the first five predictions with the observed response values.

```
yhat = predict(lme);  
[yhat(1:5) ds.Yield(1:5)]
```

```
ans =
```

```
115.4788 104.0000  
135.1455 136.0000  
152.8121 158.0000  
160.4788 174.0000  
58.0839 57.0000
```

## Plot Predictions vs. Observed Responses

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, with a fixed effect for Weight, and a random intercept grouped by Model\_Year. First, store the data in a dataset array.

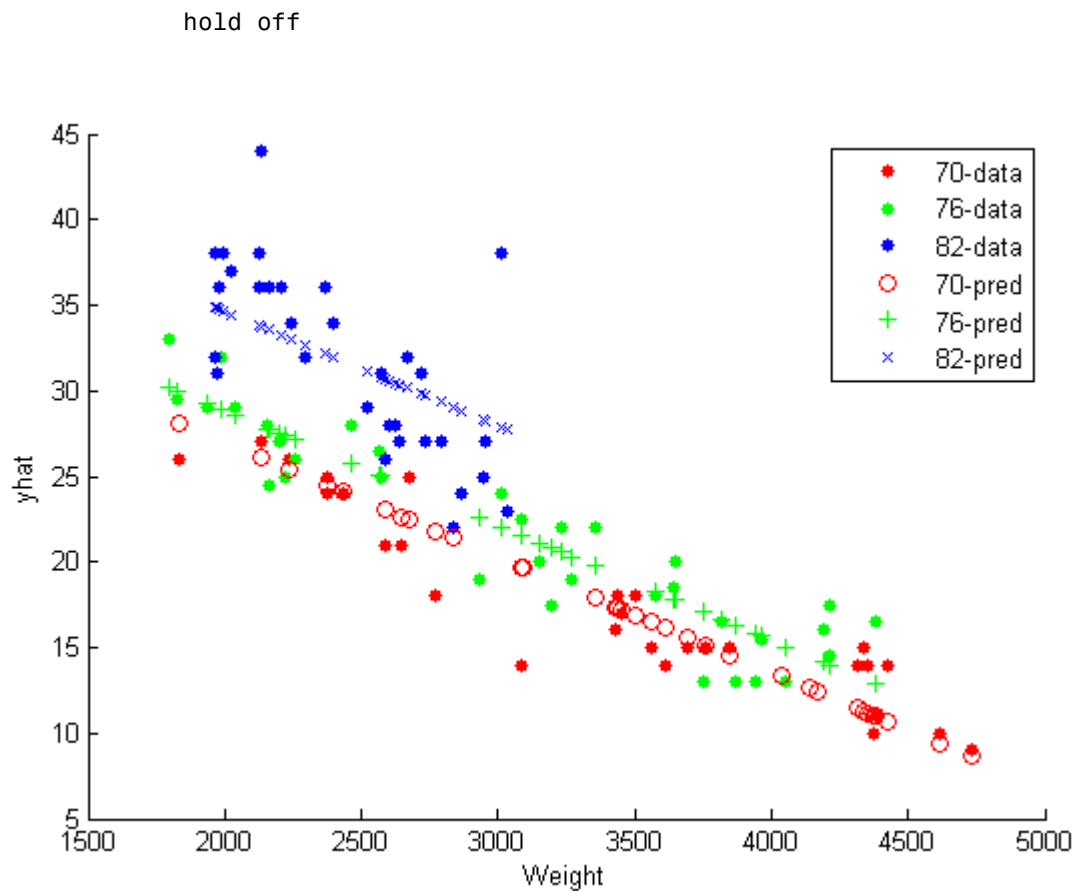
```
ds = dataset(MPG,Weight,Model_Year);  
lme = fitlme(ds,'MPG ~ Weight + (1|Model_Year)');
```

Create predicted responses to the data.

```
yhat = predict(lme,ds);
```

Plot the original responses and the predicted responses to see how they differ. Group them by model year.

```
figure()  
gscatter(Weight,MPG,Model_Year)  
hold on  
gscatter(Weight,yhat,Model_Year,[],'o+x')  
legend('70-data','76-data','82-data','70-pred','76-pred','82-pred')
```



## Predict Responses at Values in a New Dataset Array

Navigate to a folder containing sample data.

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

Load the sample data.

# LinearMixedModel.predict

---

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Create a new dataset array with design values. The new dataset array must have the same variables as the original dataset array you use for fitting the model `lme`.

```
dsnew = dataset();  
dsnew.Soil = nominal({'Sandy'; 'Silty'});  
dsnew.Tomato = nominal({'Cherry'; 'Vine'});  
dsnew.Fertilizer = nominal([2;2]);
```

Predict the conditional and marginal responses at the original design points.

```
yhatC = predict(lme, dsnew);  
yhatM = predict(lme, dsnew, 'Conditional', false);
```

```
[yhatC yhatM]
```

```
ans =
```

```
    92.7505    111.6667  
    87.5891    82.6667
```

## Predict Responses at the Values in New Design Matrices

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];  
Z = [ones(406,1) Acceleration];  
Model_Year = nominal(Model_Year);  
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...  
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...  
{{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'});
```

Create the design matrices that contain the data at which to predict the response values. `Xnew` must have three columns as in `X`. The first column must be a column of 1s. And the values in the last two columns must correspond to `Acceleration` and `Horsepower`, respectively. The first column of `Znew` must be a column of 1s, and the second column must contain the same `Acceleration` values as in `Xnew`. The original

# LinearMixedModel.predict

---

grouping variable in G is the model year. So, Gnew must contain values for the model year. Note that Gnew must contain nominal values.

```
Xnew = [1,13.5,185; 1,17,205; 1,21.2,193];  
Znew = [1,13.5; 1,17; 1,21.2];  
Gnew = nominal([73 77 82]);
```

Predict the responses for the data in the new design matrices.

```
yhat = predict(lme,Xnew,Znew,Gnew)
```

```
yhat =
```

```
    8.7063  
    5.4423  
   12.5384
```

Now, repeat the same for a linear mixed-effects model with uncorrelated random-effects terms for intercept and acceleration. First, change the original random effects design and the random effects grouping variables. Then, refit the model.

```
Z = {ones(406,1),Acceleration};  
G = {Model_Year,Model_Year};
```

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',....  
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...  
{{'Intercept'},{'Acceleration'}},'RandomEffectGroups',{'Model_Year','Model_Year'});
```

Now, recreate the new random effects design, Znew, and the grouping variable design, Gnew, using which to predict the response values.

```
Znew = {[1;1;1],[13.5;17;21.2]};  
MY = nominal([73 77 82]);  
Gnew = {MY,MY};
```

Predict the responses using the new design matrices.

```
yhat = predict(lme,Xnew,Znew,Gnew)
```



```
yhat =  
  
    8.6365  
    5.9199  
   12.1247
```

## Compute Confidence Intervals for Predictions

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year. First, store the variables in a dataset array.

```
ds = dataset(MPG,Acceleration,Horsepower,Model_Year);
```

Now, fit the model using `fitlme` with the defined design matrices and grouping variables.

```
lme = fitlme(ds,'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Year)');
```

Create the new data and store it in a new dataset array.

```
dsnew = dataset();  
dsnew.Acceleration = linspace(8,25)';  
dsnew.Horsepower = linspace(nanmin(Horsepower),nanmax(Horsepower))';  
dsnew.Model_Year = repmat(70,100,1);
```

`linspace` creates 100 equally distanced values between the lower and the upper input limits. `Model_Year` is fixed at 70. You can repeat this for any model year.

Compute and plot the predicted values and 95% confidence limits (nonsimultaneous).

```
[ypred,yCI,DF] = predict(lme,dsnew);  
figure();
```

# LinearMixedModel.predict

---

```
h1 = line(dsnew.Acceleration,ypred);
hold on;
h2 = plot(dsnew.Acceleration,yCI,'g-');
```

Display the degrees of freedom.

```
DF(1)
```

```
ans =
```

```
389
```

Compute and plot the simultaneous confidence bounds.

```
[ypred,yCI,DF] = predict(lme,dsnew,'Simultaneous',true);
h3 = plot(dsnew.Acceleration,yCI,'r--');
```

Display the degrees of freedom.

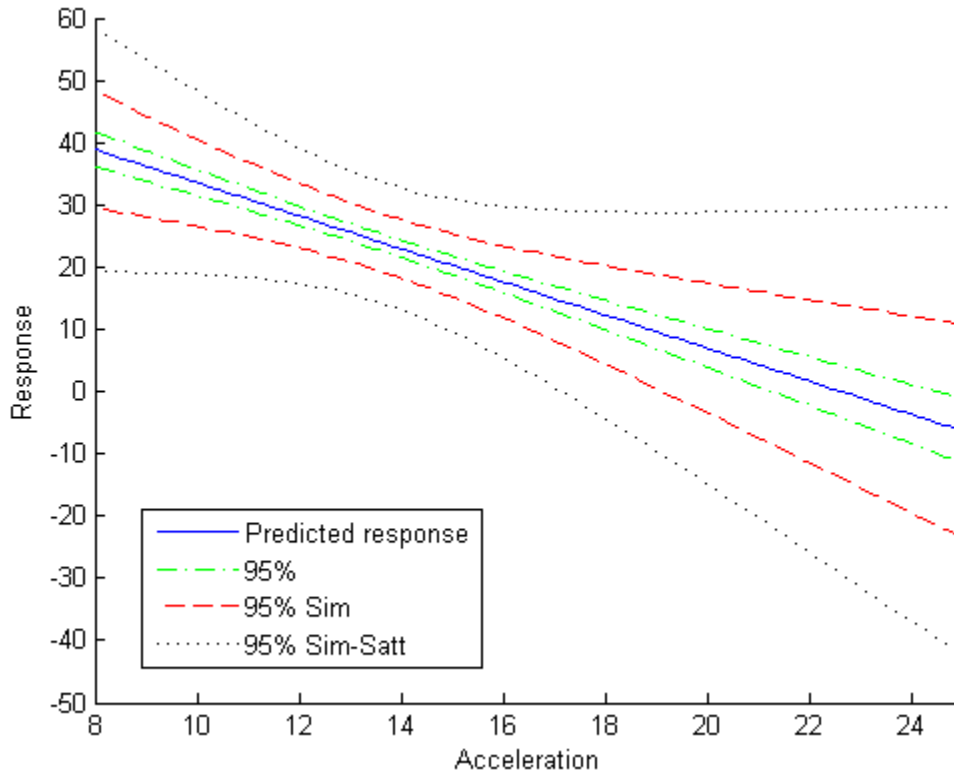
```
DF
```

```
DF =
```

```
389
```

Compute the simultaneous confidence bounds using the Satterthwaite method to compute the degrees of freedom.

```
[ypred,yCI,DF] = predict(lme,dsnew,'Simultaneous',true,'DFMethod','Satterthwaite');
y4 = plot(dsnew.Acceleration,yCI,'k:');
hold off
xlabel('Acceleration')
ylabel('Response')
ylim([-50,60])
xlim([8,25])
legend([h1,h2(1),h3(1),h4(1)],'Predicted response','95%','95% Sim',...
'95% Sim-Satt','Location','Best')
```



Display the degrees of freedom.

DF

DF =

3.6001

## Definitions

### Conditional and Marginal Predictions

A conditional prediction includes contributions from both fixed and random effects, whereas a marginal model includes contribution from only fixed effects.

Suppose the linear mixed-effects model `lme` has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also, suppose the estimated  $p$ -by-1 fixed-effects vector is  $\hat{\beta}$ , and the  $q$ -by-1 estimated best linear unbiased predictor (BLUP) vector of random effects is  $\hat{b}$ . The predicted conditional response is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b},$$

which corresponds to the 'Conditional', 'true' and 'Prediction', 'curve' name-value pair arguments. The predicted conditional response that also includes observation error is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b} + \varepsilon,$$

which corresponds to the 'Conditional', 'true' and 'Prediction', 'observation' name-value pair arguments.

The predicted marginal response is

$$\hat{y}_{Mar} = X\hat{\beta}.$$

This corresponds to the 'Conditional', 'false' and 'Prediction', 'curve' name-value pair arguments. The marginal conditional response that also includes observation error is

$$\hat{y}_{Mar} = X\hat{\beta} + \varepsilon,$$

which corresponds to the 'Conditional', 'false' and 'Prediction', 'observation' name-value pair arguments.

When making predictions, if a particular grouping variable has new levels (1s that were not in the original data), then the random effects

for the grouping variable do not contribute to the 'Conditional' prediction at observations where the grouping variable has new levels.

## See Also

`LinearMixedModel` | `fitted` | `random` |

# NaiveBayes.predict

---

**Purpose** Predict class label for test data

**Syntax**  
`cpre = predict(nb,test)`  
`cpre = predict(...,'HandleMissing',val)`

**Description** `cpre = predict(nb,test)` classifies each row of data in `test` into one of the classes according to the `NaiveBayes` classifier `nb`, and returns the predicted class level `cpre`. `test` is an `N`-by-`nb.ndims` matrix, where `N` is the number of observations in the test data. Rows of `test` correspond to points, columns of `test` correspond to features. `cpre` is an `N`-by-1 vector of the same type as `nb.CLevels`, and it indicates the class to which each row of `test` has been assigned.

`cpre = predict(...,'HandleMissing',val)` specifies how `predict` treats NaN (missing values). `val` can be one of the following:

'off'  
(default) Observations with NaN in any of the columns are not classified into any class. The corresponding rows in `post` and `logp` are NaN. The corresponding rows in `cpre` are NaN (if `obj.clevels` is numeric or logical), empty strings (if `obj.clevels` is char or cell array of strings) or `<undefined>` (if `obj.clevels` is categorical).

'on' For observations having NaN in some (but not all) columns, `post` and `predict` computes `cpre` using the columns with non-NaN values. Corresponding `logp` values are NaN.

**See Also** `NaiveBayes` | `fit` | `posterior`

## Purpose

Predict response of nonlinear regression model

## Syntax

```
ypred = predict mdl, Xnew
[ypred, yci] = predict mdl, Xnew
[ypred, yci] = predict mdl, Xnew, Name, Value
```

## Description

`ypred = predict(mdl, Xnew)` returns the predicted response of the `mdl` nonlinear regression model to the points in `Xnew`.

`[ypred, yci] = predict(mdl, Xnew)` returns confidence intervals for the true mean responses.

`[ypred, yci] = predict(mdl, Xnew, Name, Value)` predicts responses with additional options specified by one or more `Name, Value` pair arguments.

## Tips

- For predictions with added noise, use `random`.
- For a syntax that can be easier to use with models created from dataset arrays, try `feval`.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments.

`Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can

# NonLinearModel.predict

---

specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## 'alpha'

Positive scalar from 0 to 1. Confidence level of `yci` is  $100(1 - \text{alpha})\%$ .

**Default:** 0.05, meaning a 95% confidence interval.

## 'Prediction'

String specifying the type of prediction:

- 'curve' — `predict` predicts confidence bounds for the fitted mean values.
- 'observation' — `predict` predicts confidence bounds for the new observations. This results in wider bounds because the error in a new observation is equal to the error in the estimated mean value, plus the variability in the observation from the true mean.

For details, see `polyconf`.

**Default:** 'curve'

## 'Simultaneous'

Logical value specifying whether the confidence bounds are for all predictor values simultaneously (`true`), or hold for each individual predictor value (`false`). Simultaneous bounds are wider than separate bounds, because it is more stringent to require that the entire curve be within the bounds than to require that the curve at a single predictor value be within the bounds.

For details, see `polyconf`.

**Default:** `false`



## 'Weights'

Vector of real, positive value weights or a function handle.

- If you specify a vector, then it must have the same number of elements as the number of observations (or rows) in  $X_{new}$ .
- If you specify a function handle, then the function must accept a vector of predicted response values as input, and return a vector of real positive weights as output.

Given weights,  $W$ , `predict` estimates the error variance at observation  $i$  by  $MSE * (1/W(i))$ , where  $MSE$  is the mean squared error.

**Default:** No weights

## Output Arguments

### `ypred`

Vector of predicted mean values at  $X_{new}$ .

### `yci`

Confidence intervals, a two-column matrix with each row providing one interval. The meaning of the confidence interval depends on the settings of the name-value pairs.

## Examples

### Predict Responses

Create a nonlinear model of car mileage as a function of weight, and predict the response.

Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
y = MPG;
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';
```

# NonLinearModel.predict

---

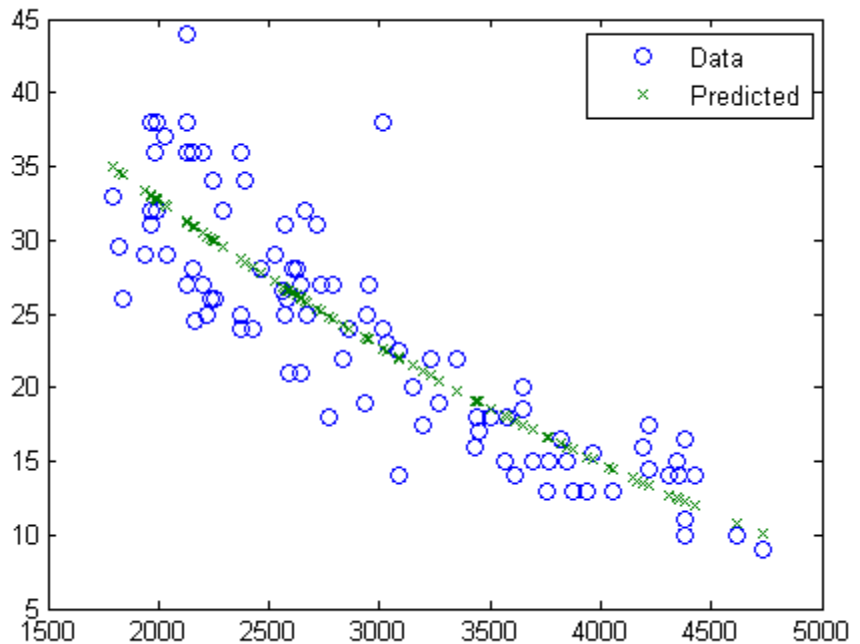
```
beta0 = [1 1 1];  
mdl = fitnlm(X,y,modelfun,beta0);
```

Create predicted responses to the data.

```
Xnew = X;  
ypred = predict(mdl,Xnew);
```

Plot the original responses and the predicted responses to see how they differ.

```
plot(X,y,'o',X,ypred,'x')  
legend('Data','Predicted')
```



## Confidence Intervals for Predictions

Create a nonlinear model of car mileage as a function of weight, and examine confidence intervals of some responses.

Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
y = MPG;
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';
beta0 = [1 1 1];
mdl = fitnlm(X,y,modelfun,beta0);
```

Create predicted responses to the smallest, mean, and largest data points.

```
Xnew = [min(X);mean(X);max(X)];
[ypred,yci] = predict(mdl,Xnew)
```

```
ypred =
    34.9469
    22.6868
    10.0617
```

```
yci =
    32.5212    37.3726
    21.4061    23.9674
    7.0148    13.1086
```

## Simultaneous Confidence Intervals for Robust Fit Curve

Generate sample data from the nonlinear regression model

$$y = b_1 + b_2 \exp\{-b_3 x\} + \varepsilon,$$

# NonLinearModel.predict

---

where  $b_1$ ,  $b_2$ , and  $b_3$  are coefficients, and the error term is normally distributed with mean 0 and standard deviation 0.5.

```
modelfun = @(b,x)(b(1)+b(2)*exp(-b(3)*x));
```

```
rng('default') % for reproducibility  
b = [1;3;2];  
x = exprnd(2,100,1);  
y = modelfun(b,x) + normrnd(0,0.5,100,1);
```

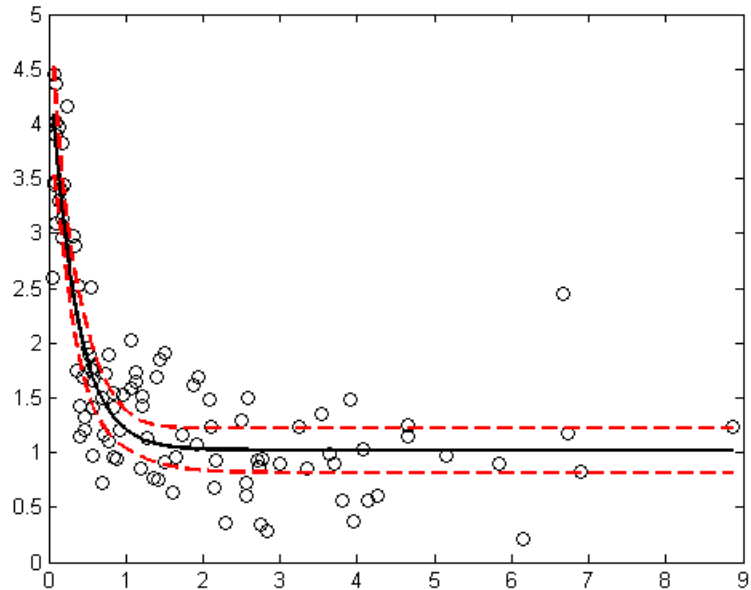
Fit the nonlinear model using robust fitting options.

```
opts = statset('nlinfit');  
opts.RobustWgtFun = 'bisquare';  
b0 = [2;2;2];  
mdl = fitnlm(x,y,modelfun,b0,'Options',opts);
```

Plot the fitted regression model and simultaneous 95% confidence bounds.

```
xrange = [min(x):.01:max(x)]';  
[ypred,yci] = predict(mdl,xrange,'Simultaneous',true);
```

```
figure()  
plot(x,y,'ko') % observed data  
hold on  
plot(xrange,ypred,'k','LineWidth',2)  
plot(xrange,yci,'r--','LineWidth',1.5)
```



## Confidence Interval Using Observation Weights

Load sample data.

```
S = load('reaction');  
X = S.reactants;  
y = S.rate;  
beta0 = S.beta;
```

Specify a function handle for observation weights, then fit the Hougen-Watson model to the rate data using the specified observation weights function.

```
a = 1; b = 1;  
weights = @(yhat) 1./((a + b*abs(yhat)).^2);  
mdl = fitnlm(X,y,@hougen,beta0,'Weights',weights);
```

# NonLinearModel.predict

---

Compute the 95% prediction interval for a new observation with reactant levels [100,100,100] using the observation weight function.

```
[ypred,yci] = predict mdl,[100,100,100], 'Prediction', 'observation', ...  
                'Weights', weights)
```

```
ypred =
```

```
    1.8149
```

```
yci =
```

```
    1.5264    2.1033
```

## References

[1] Lane, T. P. and W. H. DuMouchel. “Simultaneous Confidence Intervals in Multiple Regression.” *The American Statistician*. Vol. 48, No. 4, 1994, pp. 315–321.

[2] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.

## See Also

NonLinearModel | random |

## Related Examples

- “Predict or Simulate Responses to New Data” on page 11-12
- “Nonlinear Regression Workflow” on page 11-16

## Concepts

- “Nonlinear Regression” on page 11-2

**Purpose** Predict response

**Syntax**

```
Y = predict(B,X)
[Y,stdevs] = predict(B,X)
[Y,scores] = predict(B,X)
[Y,scores,stdevs] = predict(B,X)
Y = predict(B,X,'param1',val1,'param2',val2,...)
```

**Description** `Y = predict(B,X)` computes predicted response of the trained ensemble `B` for data `X`. The output has one prediction for each row of `X`. The returned `Y` is a cell array of strings for classification and a numeric array for regression.

For regression, `[Y,stdevs] = predict(B,X)` also returns standard deviations of the computed responses over the ensemble of the grown trees.

For classification, `[Y,scores] = predict(B,X)` returns scores for all classes. `scores` is a matrix with one row per observation and one column per class. For each observation and each class, the score generated by each tree is the probability of this observation originating from this class computed as the fraction of observations of this class in a tree leaf. `predict` averages these scores over all trees in the ensemble.

`[Y,scores,stdevs] = predict(B,X)` also returns standard deviations of the computed scores for classification. `stdevs` is a matrix with one row per observation and one column per class, with standard deviations taken over the ensemble of the grown trees.

`Y = predict(B,X,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

# TreeBagger.predict

---

'trees'	Array of tree indices to use for computation of responses. Default is 'all'.
'treeweights'	Array of NTrees weights for weighting votes from the specified trees.
'useifort'	Logical matrix of size Nobs-by-NTrees indicating which trees to use to make predictions for each observation. By default all trees are used for all observations.

## See Also

`CompactTreeBagger.predict`



# CompactClassificationEnsemble.predictorImportance

---

<b>Purpose</b>	Estimates of predictor importance
<b>Syntax</b>	<code>imp = predictorImportance(ens)</code> <code>[imp,ma] = predictorImportance(ens)</code>
<b>Description</b>	<p><code>imp = predictorImportance(ens)</code> computes estimates of predictor importance for <code>ens</code> by summing these estimates over all weak learners in the ensemble. <code>imp</code> has one element for each input predictor in the data used to train this ensemble. A high value indicates that this predictor is important for <code>ens</code>.</p> <p><code>[imp,ma] = predictorImportance(ens)</code> returns a P-by-P matrix with predictive measures of association for P predictors, when the learners in <code>ens</code> contain surrogate splits. See “Definitions” on page 21-2674.</p>
<b>Input Arguments</b>	<p><b>ens</b></p> <p>A classification ensemble created by <code>fitensemble</code>, or by the <code>compact</code> method.</p>
<b>Output Arguments</b>	<p><b>imp</b></p> <p>A row vector with the same number of elements as the number of predictors (columns) in <code>ens.X</code>. The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.</p> <p><b>ma</b></p> <p>A P-by-P matrix of predictive measures of association for P predictors. Element <code>ma(I,J)</code> is the predictive measure of association averaged over surrogate splits on predictor J for which predictor I is the optimal split predictor. <code>predictorImportance</code> averages this predictive measure of association over all trees in the ensemble.</p>

## Definitions

### Predictor Importance

`predictorImportance` computes estimates of predictor importance for `ens` by summing changes in the *risk* due to splits on every predictor and dividing the sum by the number of branch nodes. If `ens` is grown without surrogate splits, this sum is taken over best splits found at each branch node. If `ens` is grown with surrogate splits, this sum is taken over all splits at each branch node including surrogate splits. `imp` has one element for each input predictor in the data used to train `ens`. Predictor importance associated with this split is computed as the difference between the risk for the parent node and the total risk for the two children.

### Impurity and Node Error

`ClassificationTree` splits nodes based on either *impurity* or *node error*. Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a *pure* node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('`deviance`') — With  $p(i)$  defined as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('`twoing`') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a

# CompactClassificationEnsemble.predictorImportance

split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R)\left(\sum_i |L(i) - R(i)|\right)^2,$$

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Predictive Measure of Association

The predictive measure of association between the optimal split on variable  $i$  and a surrogate split on variable  $j$  is:

$$\lambda_{i,j} = \frac{\min(P_L, P_R) - (1 - P_{L_i L_j} - P_{R_i R_j})}{\min(P_L, P_R)}.$$

Here

- $P_L$  and  $P_R$  are the node probabilities for the optimal split of node  $i$  into Left and Right nodes respectively.
- $P_{L_i L_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Left.
- $P_{R_i R_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Right.

# CompactClassificationEnsemble.predictorImportance

---

Clearly,  $\lambda_{ij}$  lies from  $-\infty$  to 1. Variable  $j$  is a worthwhile surrogate split for variable  $i$  if  $\lambda_{ij} > 0$ .

Element  $ma(i, j)$  is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of optimal splits on predictor  $i$ , including splits for which the predictive measure of association between predictors  $i$  and  $j$  is negative.

## Examples

Estimate the predictor importance for all variables in the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
imp = predictorImportance(ens)
```

```
imp =
    0.0001    0.0005    0.0384    0.0146
```

The first two predictors are not very important in ens.

---

Estimate the predictor importance for all variables in the Fisher iris data for an ensemble where the trees contain surrogate splits:

```
load fisheriris
surrtree = ClassificationTree.template('Surrogate','on');
ens2 = fitensemble(meas,species,'AdaBoostM2',100,surrtree);
[imp2,ma] = predictorImportance(ens2)
```

```
imp2 =
    0.0224    0.0142    0.0525    0.0508
```

```
ma =
    1.0000         0    0.0001    0.0001
    0.0115    1.0000    0.0023    0.0054
```

# CompactClassificationEnsemble.predictorImportance

---

0.2810	0.1747	1.0000	0.5372
0.0789	0.0463	0.2339	1.0000

The first two predictors show much more importance than in the previous example.

**See Also** [predictorImportance](#)

# CompactClassificationTree.predictorImportance

---

<b>Purpose</b>	Estimates of predictor importance
<b>Syntax</b>	<code>imp = predictorImportance(tree)</code>
<b>Description</b>	<code>imp = predictorImportance(tree)</code> computes estimates of predictor importance for <code>tree</code> by summing changes in the risk due to splits on every predictor and dividing the sum by the number of branch nodes.
<b>Input Arguments</b>	<b>tree</b> A classification tree created by <code>ClassificationTree.fit</code> , or by the <code>compact</code> method.
<b>Output Arguments</b>	<b>imp</b> A row vector with the same number of elements as the number of predictors (columns) in <code>tree.X</code> . The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.

## Definitions **Predictor Importance**

`predictorImportance` computes estimates of predictor importance for `tree` by summing changes in the *risk* due to splits on every predictor and dividing the sum by the number of branch nodes. If `tree` is grown without surrogate splits, this sum is taken over best splits found at each branch node. If `tree` is grown with surrogate splits, this sum is taken over all splits at each branch node including surrogate splits. `imp` has one element for each input predictor in the data used to train `tree`. Predictor importance associated with this split is computed as the difference between the risk for the parent node and the total risk for the two children.

Estimates of predictor importance do not depend on the order of predictors if you use surrogate splits, but do depend on the order if you do not use surrogate splits.

If you use surrogate splits, `predictorImportance` computes estimates before the tree is reduced by pruning or merging leaves. If you do not

use surrogate splits, `predictorImportance` computes estimates after the tree is reduced by pruning or merging leaves. Therefore, reducing the tree by pruning affects the predictor importance for a tree grown without surrogate splits, and does not affect the predictor importance for a tree grown with surrogate splits.

## Impurity and Node Error

`ClassificationTree` splits nodes based on either *impurity* or *node error*. Impurity means one of several things, depending on your choice of the `SplitCriterion` name-value pair:

- Gini's Diversity Index (`gdi`) — The Gini index of a node is

$$1 - \sum_i p^2(i),$$

where the sum is over the classes  $i$  at the node, and  $p(i)$  is the observed fraction of classes with class  $i$  that reach the node. A node with just one class (a *pure* node) has Gini index 0; otherwise the Gini index is positive. So the Gini index is a measure of node impurity.

- Deviance ('`deviance`') — With  $p(i)$  defined as for the Gini index, the deviance of a node is

$$-\sum_i p(i) \log p(i).$$

A pure node has deviance 0; otherwise, the deviance is positive.

- Twoing rule ('`twoing`') — Twoing is not a purity measure of a node, but is a different measure for deciding how to split a node. Let  $L(i)$  denote the fraction of members of class  $i$  in the left child node after a split, and  $R(i)$  denote the fraction of members of class  $i$  in the right child node after a split. Choose the split criterion to maximize

$$P(L)P(R) \left( \sum_i |L(i) - R(i)| \right)^2,$$

# CompactClassificationTree.predictorImportance

---

where  $P(L)$  and  $P(R)$  are the fractions of observations that split to the left and right respectively. If the expression is large, the split made each child node purer. Similarly, if the expression is small, the split made each child node similar to each other, and hence similar to the parent node, and so the split did not increase node purity.

- Node error — The node error is the fraction of misclassified classes at a node. If  $j$  is the class with largest number of training samples at a node, the node error is

$$1 - p(j).$$

## Examples

Estimate the predictor importance for all variables in the Fisher iris data:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
imp = predictorImportance(tree)
```

```
imp =
      0      0  0.0403  0.0303
```

The first two elements of `imp` are zero. Therefore, the first two predictors do not enter into tree calculations for classifying irises.

---

Estimate the predictor importance for all variables in the Fisher iris data for a tree grown with surrogate splits:

```
tree2 = ClassificationTree.fit(meas,species,...
    'Surrogate','on');
imp2 = predictorImportance(tree2)
```

```
imp2 =
  0.0287  0.0136  0.0560  0.0556
```

In this case, all predictors have some importance. As you expect by comparing to the first example, the first two predictors are less important than the final two.



# CompactClassificationTree.predictorImportance

---

Estimates of predictor importance do not depend on the order of predictors if you use surrogate splits, but do depend on the order if you do not use surrogate splits. For example, permute the order of the data columns in the previous example:

```
load fisheriris
meas3 = meas(:, [4 1 3 2]);
tree3 = ClassificationTree.fit(meas3, species);
imp3 = predictorImportance(tree2)
```

```
imp3 =
    0.0674         0    0.0033         0
```

The estimates of predictor importance are not a permutation of `imp` from the first example.

Estimate the predictor importance using surrogate splits.

```
tree4 = ClassificationTree.fit(meas3, species, ...
    'Surrogate', 'on');
imp4 = predictorImportance(tree4)
```

```
imp4 =
    0.0556    0.0287    0.0560    0.0136
```

`imp4` is a permutation of `imp2`, demonstrating that estimates of predictor importance do not depend on the order of predictors with surrogate splits.

## See Also

`predictorImportance`

# CompactRegressionEnsemble.predictorImportance

---

<b>Purpose</b>	Estimates of predictor importance
<b>Syntax</b>	<code>imp = predictorImportance(ens)</code> <code>[imp,ma] = predictorImportance(ens)</code>
<b>Description</b>	<p><code>imp = predictorImportance(ens)</code> computes estimates of predictor importance for <code>ens</code> by summing these estimates over all weak learners in the ensemble. <code>imp</code> has one element for each input predictor in the data used to train this ensemble. A high value indicates that this predictor is important for <code>ens</code>.</p> <p><code>[imp,ma] = predictorImportance(ens)</code> returns a P-by-P matrix with predictive measures of association for P predictors.</p>
<b>Input Arguments</b>	<p><b>ens</b></p> <p>A regression ensemble created by <code>fitensemble</code>, or by the compact method.</p>
<b>Output Arguments</b>	<p><b>imp</b></p> <p>A row vector with the same number of elements as the number of predictors (columns) in <code>ens.X</code>. The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.</p> <p><b>ma</b></p> <p>A P-by-P matrix of predictive measures of association for P predictors. Element <code>ma(I,J)</code> is the predictive measure of association averaged over surrogate splits on predictor J for which predictor I is the optimal split predictor. <code>predictorImportance</code> averages this predictive measure of association over all trees in the ensemble.</p>
<b>Definitions</b>	<p><b>Predictor Importance</b></p> <p><code>predictorImportance</code> computes estimates of predictor importance for tree by summing changes in the mean squared error (MSE) due to</p>

splits on every predictor and dividing the sum by the number of branch nodes. If the tree is grown without surrogate splits, this sum is taken over best splits found at each branch node. If the tree is grown with surrogate splits, this sum is taken over all splits at each branch node including surrogate splits. `imp` has one element for each input predictor in the data used to train this tree. At each node, MSE is estimated as node error weighted by the node probability. Variable importance associated with this split is computed as the difference between MSE for the parent node and the total MSE for the two children.

## Predictive Measure of Association

The predictive measure of association between the optimal split on variable  $i$  and a surrogate split on variable  $j$  is:

$$\lambda_{i,j} = \frac{\min(P_L, P_R) - (1 - P_{L_i L_j} - P_{R_i R_j})}{\min(P_L, P_R)}.$$

Here

- $P_L$  and  $P_R$  are the node probabilities for the optimal split of node  $i$  into Left and Right nodes respectively.
- $P_{L_i L_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Left.
- $P_{R_i R_j}$  is the probability that both (optimal) node  $i$  and (surrogate) node  $j$  send an observation to the Right.

Clearly,  $\lambda_{i,j}$  lies from  $-\infty$  to 1. Variable  $j$  is a worthwhile surrogate split for variable  $i$  if  $\lambda_{i,j} > 0$ .

Element `ma(i, j)` is the predictive measure of association averaged over surrogate splits on predictor  $j$  for which predictor  $i$  is the optimal split predictor. This average is computed by summing positive values of the predictive measure of association over optimal splits on predictor  $i$  and surrogate splits on predictor  $j$  and dividing by the total number of

# CompactRegressionEnsemble.predictorImportance

---

optimal splits on predictor *i*, including splits for which the predictive measure of association between predictors *i* and *j* is negative.

## Examples

Estimate the predictor importance for all numeric variables in the `carsmall` data:

```
load carsmall
X = [Acceleration Cylinders Displacement ...
     Horsepower Model_Year Weight];
ens = fitensemble(X,MPG,'LSBoost',100,'Tree');
imp = predictorImportance(ens)

imp =
    0.0082         0    0.0049    0.0133    0.0142    0.1737
```

The weight (last predictor) has the most impact on mileage (MPG). The second predictor has importance 0; this means the number of cylinders has no impact on predictions made with `ens`.

---

Estimate the predictor importance for all variables in the `carsmall` data for an ensemble where the trees contain surrogate splits:

```
load carsmall
surrtree = RegressionTree.template('Surrogate','on');
X = [Acceleration Cylinders Displacement ...
     Horsepower Model_Year Weight];
ens2 = fitensemble(X,MPG,'LSBoost',100,surrtree);
[imp2,ma] = predictorImportance(ens2)

imp2 =
    0.0725    0.1342    0.1425    0.1397    0.1380    0.1855

ma =
    1.0000    0.0414    0.0607    0.0782    0.0102    0.0322
         0    1.0000         0         0         0         0
    0.0441    0.0704    1.0000    0.0883    0.0175    0.0913
    0.0944    0.1166    0.1400    1.0000    0.0390    0.1308
```

# CompactRegressionEnsemble.predictorImportance

---

0.0121	0.0139	0.0127	0.0127	1.0000	0.0113
0.0818	0.1317	0.2072	0.1878	0.0340	1.0000

While weight (last predictor) still has the most impact on mileage (MPG), this estimate has the second predictor (number of cylinders) is essentially tied for third most important predictor.

## See Also

`predictorImportance`

# CompactRegressionTree.predictorImportance

---

<b>Purpose</b>	Estimates of predictor importance
<b>Syntax</b>	<code>imp = predictorImportance(tree)</code>
<b>Description</b>	<code>imp = predictorImportance(tree)</code> computes estimates of predictor importance for <code>tree</code> by summing changes in the mean squared error due to splits on every predictor and dividing the sum by the number of branch nodes.
<b>Input Arguments</b>	<b>tree</b> A regression tree created by <code>RegressionTree.fit</code> , or by the <code>compact</code> method.
<b>Output Arguments</b>	<b>imp</b> A row vector with the same number of elements as the number of predictors (columns) in <code>tree.X</code> . The entries are the estimates of predictor importance, with 0 representing the smallest possible importance.
<b>Definitions</b>	<b>Predictor Importance</b> <p><code>predictorImportance</code> computes estimates of predictor importance for <code>tree</code> by summing changes in the mean squared error (MSE) due to splits on every predictor and dividing the sum by the number of branch nodes. If the tree is grown without surrogate splits, this sum is taken over best splits found at each branch node. If the tree is grown with surrogate splits, this sum is taken over all splits at each branch node including surrogate splits. <code>imp</code> has one element for each input predictor in the data used to train this tree. At each node, MSE is estimated as node error weighted by the node probability. Variable importance associated with this split is computed as the difference between MSE for the parent node and the total MSE for the two children.</p> <p>Estimates of predictor importance do not depend on the order of predictors if you use surrogate splits, but do depend on the order if you do not use surrogate splits.</p>

# CompactRegressionTree.predictorImportance

---

If you use surrogate splits, `predictorImportance` computes estimates before the tree is reduced by pruning or merging leaves. If you do not use surrogate splits, `predictorImportance` computes estimates after the tree is reduced by pruning or merging leaves. Therefore, reducing the tree by pruning affects the predictor importance for a tree grown without surrogate splits, and does not affect the predictor importance for a tree grown with surrogate splits.

## Examples

Find predictor importance for the `carsmall` data. Use just the numeric predictors:

```
load carsmall
X = [Acceleration Cylinders Displacement ...
     Horsepower Model_Year Weight];
tree = RegressionTree.fit(X,MPG);
imp = predictorImportance(tree)

imp =
    0.0315         0    0.1082    0.0686    0.1629    1.2924
```

The weight (last predictor) has the most impact on mileage (MPG). The second predictor has importance 0; this means the number of cylinders has no impact on predictions made with `tree`.

---

Estimate the predictor importance for all variables in the `carsmall` data for a tree grown with surrogate splits:

```
load carsmall
X = [Acceleration Cylinders Displacement ...
     Horsepower Model_Year Weight];
tree2 = RegressionTree.fit(X,MPG,...
    'Surrogate','on');
imp2 = predictorImportance(tree2)

imp2 =
    0.5287    1.1977    1.2400    0.7059    1.0677    1.4106
```

# CompactRegressionTree.predictorImportance

---

While weight (last predictor) still has the most impact on mileage (MPG), this estimate has the second predictor (number of cylinders) as the third most important predictor.

## See Also

`predictorImportance`



**Purpose** Principal component analysis (PCA) on data

**Syntax**

```
[COEFF,SCORE] = princomp(X)
[COEFF,SCORE,latent] = princomp(X)
[COEFF,SCORE,latent,tsquare] = princomp(X)
[...] = princomp(X,'econ')
```

**Description** `COEFF = princomp(X)` performs principal components analysis (PCA) on the  $n$ -by- $p$  data matrix  $X$ , and returns the principal component coefficients, also known as loadings. Rows of  $X$  correspond to observations, columns to variables. `COEFF` is a  $p$ -by- $p$  matrix, each column containing coefficients for one principal component. The columns are in order of decreasing component variance.

`princomp` centers  $X$  by subtracting off column means, but does not rescale the columns of  $X$ . To perform principal components analysis with standardized variables, that is, based on correlations, use `princomp(zscore(X))`. To perform principal components analysis directly on a covariance or correlation matrix, use `pcacov`.

`[COEFF,SCORE] = princomp(X)` returns `SCORE`, the principal component scores; that is, the representation of  $X$  in the principal component space. Rows of `SCORE` correspond to observations, columns to components.

`[COEFF,SCORE,latent] = princomp(X)` returns `latent`, a vector containing the eigenvalues of the covariance matrix of  $X$ .

`[COEFF,SCORE,latent,tsquare] = princomp(X)` returns `tsquare`, which contains Hotelling's  $T^2$  statistic for each data point.

The scores are the data formed by transforming the original data into the space of the principal components. The values of the vector `latent` are the variance of the columns of `SCORE`. Hotelling's  $T^2$  is a measure of the multivariate distance of each observation from the center of the data set.

When  $n \leq p$ , `SCORE(:,n:p)` and `latent(n:p)` are necessarily zero, and the columns of `COEFF(:,n:p)` define directions that are orthogonal to  $X$ .

[...] = princomp(X, 'econ') returns only the elements of latent that are not necessarily zero, and the corresponding columns of COEFF and SCORE, that is, when  $n \leq p$ , only the first  $n-1$ . This can be significantly faster when  $p$  is much larger than  $n$ .

## Examples

Compute principal components for the ingredients data in the Hald data set, and the variance accounted for by each component.

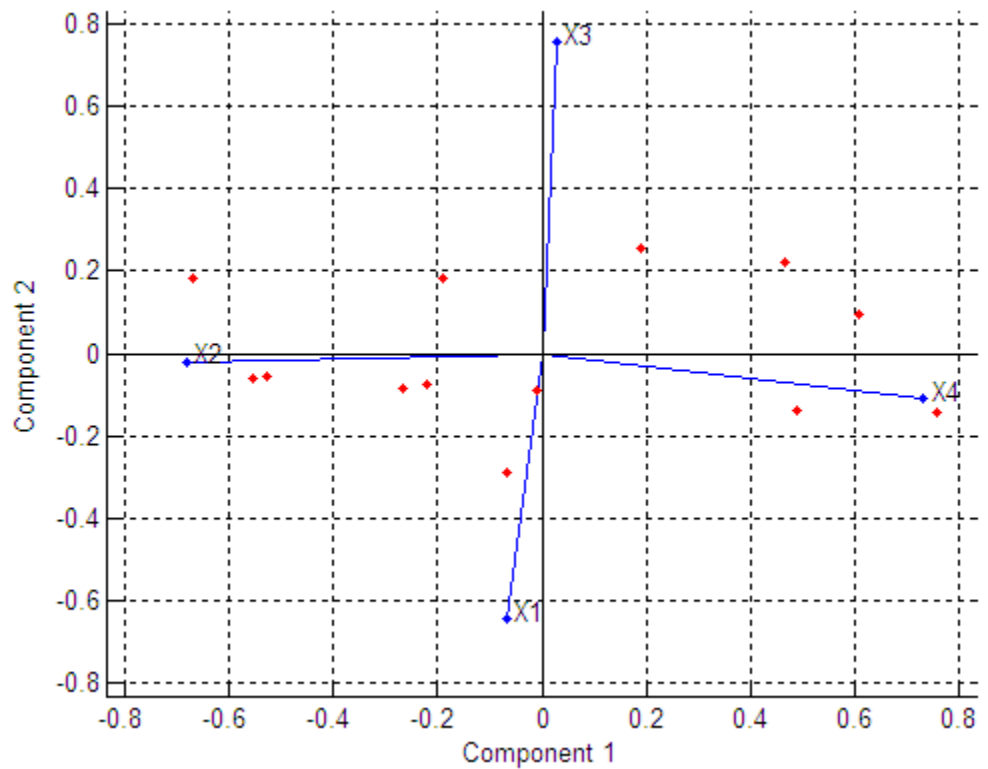
```
load hald;
[pc,score,latent,tsquare] = princomp(ingredients);
pc,latent
```

```
pc =
    0.0678 -0.6460  0.5673 -0.5062
    0.6785 -0.0200 -0.5440 -0.4933
   -0.0290  0.7553  0.4036 -0.5156
   -0.7309 -0.1085 -0.4684 -0.4844
```

```
latent =
517.7969
 67.4964
 12.4054
  0.2372
```

The following command and plot show that two components account for 98% of the variance:

```
cumsum(latent)./sum(latent)
ans =
    0.86597
    0.97886
    0.9996
     1
biplot(pc(:,1:2), 'Scores', score(:,1:2), 'VarLabels', ...
        {'X1' 'X2' 'X3' 'X4'})
```



For a more detailed example and explanation of this analysis method, see “Principal Component Analysis (PCA)” on page 13-80.

## References

- [1] Jackson, J. E., *A User's Guide to Principal Components*, John Wiley and Sons, 1991, p. 592.
- [2] Jolliffe, I. T., *Principal Component Analysis*, 2nd edition, Springer, 2002.
- [3] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.

# princomp

---

[4] Seber, G. A. F., *Multivariate Observations*, Wiley, 1984.

## See Also

barttest | biplot | canoncorr | factoran | pca | pcacov | pcares  
| rotatefactors

## How To

• “Principal Component Analysis (PCA)” on page 13-80

<b>Purpose</b>	Prior class probabilities
<b>Description</b>	The Prior property is a vector with prior probabilities for classes. This property is empty for ensembles of regression trees.
<b>See Also</b>	<code>classregtree</code>

# ProbDist

---

<b>Purpose</b>	Object representing probability distribution	
<b>Description</b>	ProbDist is an abstract class representing a probability distribution.	
<b>Construction</b>	ProbDist is an abstract class. You cannot create instances of this class directly. You can construct an object in a subclass, such as ProbDistUnivParam or ProbDistUnivKernel, either by calling the subclass constructors (ProbDistUnivParam.ProbDistUnivParam or ProbDistUnivKernel.ProbDistUnivKernel) or by using the fitdist function.	
<b>Methods</b>	cdf	Return cumulative distribution function (CDF) for ProbDist object
	pdf	Return probability density function (PDF) for ProbDist object
	random	Generate random number drawn from ProbDist object
<b>Properties</b>	DistName	Read-only string containing probability distribution name of ProbDist object
	InputData	Read-only structure containing information about input data to ProbDist object
	Support	Read-only structure containing information about support of ProbDist object
<b>Copy Semantics</b>	Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.	

## See Also

`fitdist` | `ProbDistParametric` | `ProbDistKernel`  
| `ProbDistUnivParam` | `ProbDistUnivKernel`  
| `ProbDistUnivParam.ProbDistUnivParam` |  
`ProbDistUnivKernel.ProbDistUnivKernel`

# ProbDistKernel

---

**Superclasses** ProbDist

**Purpose** Object representing nonparametric probability distribution defined by kernel smoothing

**Description** ProbDistKernel is an abstract class defining the properties and methods of a nonparametric distribution defined by a kernel smoothing function.

**Construction** ProbDistKernel is an abstract class. You cannot create instances of this class directly. You can construct an object in a subclass, ProbDistUnivKernel, either by calling the subclass constructor, ProbDistUnivKernel.ProbDistUnivKernel, or by using the fitdist function.

<b>Methods</b>	cdf	Return cumulative distribution function (CDF) for ProbDist object
	pdf	Return probability density function (PDF) for ProbDist object
	random	Generate random number drawn from ProbDist object

---

**Note** The above methods are inherited from the ProbDist class.

---

<b>Properties</b>	BandWidth	Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object
	DistName	Read-only string containing probability distribution name of ProbDist object



InputData	Read-only structure containing information about input data to ProbDist object
Kernel	Read-only string specifying name of kernel smoothing function for ProbDistKernel object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** Some of the above properties are inherited from the ProbDist class.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## See Also

`fitdist` | `ProbDist` | `ProbDistUnivKernel` | `ProbDistUnivKernel.ProbDistUnivKernel`

# ProbDistParametric

---

<b>Superclasses</b>	ProbDist
<b>Purpose</b>	Object representing parametric probability distribution
<b>Description</b>	ProbDistParametric is an abstract class defining the properties and methods of a parametric probability distribution.
<b>Construction</b>	ProbDistParametric is an abstract class. You cannot create instances of this class directly. You can construct an object in its subclass, ProbDistUnivParam, either by calling the subclass constructor, ProbDistUnivParam.ProbDistUnivParam, or by using the fitdist function.

<b>Methods</b>	cdf	Return cumulative distribution function (CDF) for ProbDist object
	pdf	Return probability density function (PDF) for ProbDist object
	random	Generate random number drawn from ProbDist object

---

**Note** The above methods are inherited from the ProbDist class.

---

<b>Properties</b>	DistName	Read-only string containing probability distribution name of ProbDist object
	InputData	Read-only structure containing information about input data to ProbDist object

NLogL	Read-only value specifying negative log likelihood for input data to ProbDistParametric object
NumParams	Read-only value specifying number of parameters of ProbDistParametric object
ParamCov	Read-only covariance matrix of parameter estimates of ProbDistParametric object
ParamDescription	Read-only cell array specifying descriptions of parameters of ProbDistParametric object
ParamIsFixed	Read-only logical array specifying fixed parameters of ProbDistParametric object
ParamNames	Read-only cell array specifying names of parameters of ProbDistParametric object
Params	Read-only array specifying values of parameters of ProbDistParametric object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** Some of the above properties are inherited from the ProbDist class.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

# ProbDistParametric

---

## See Also

[fitdist](#) | [ProbDist](#) | [ProbDistUnivParam](#) |  
[ProbDistUnivParam.ProbDistUnivParam](#)

**Superclasses** ProbDistKernel

**Purpose** Object representing univariate kernel probability distribution

**Description** A ProbDistUnivKernel object represents a univariate nonparametric probability distribution defined by kernel smoothing. You create this object using the `fitdist` function to fit the distribution to data.

## Construction

<b>Methods</b>	<code>cdf</code>	Return cumulative distribution function (CDF) for ProbDist object
	<code>icdf</code>	Return inverse cumulative distribution function (ICDF) for ProbDistUnivKernel object
	<code>iqr</code>	Return interquartile range (IQR) for ProbDistUnivKernel object
	<code>median</code>	Return median of ProbDistUnivKernel object
	<code>pdf</code>	Return probability density function (PDF) for ProbDist object
	<code>random</code>	Generate random number drawn from ProbDist object

---

**Note** Some of the above methods are inherited from the ProbDistKernel class.

---

# ProbDistUnivKernel

---

## Properties

BandWidth	Read-only value specifying bandwidth of kernel smoothing function for ProbDistKernel object
DistName	Read-only string containing probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
Kernel	Read-only string specifying name of kernel smoothing function for ProbDistKernel object
NLogL	Read-only value specifying negative log likelihood for input data to ProbDistUnivKernel object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** Some of the above properties are inherited from the ProbDistKernel class.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## References

[1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

**See Also**

`fitdist` | `ksdensity` | `ProbDist` | `ProbDistKernel` |  
`ProbDistUnivKernel.ProbDistUnivKernel`

# ProbDistUnivKernel

---

**Purpose** Construct ProbDistUnivKernel object

**Syntax**  $PD = \text{ProbDistUnivKernel}(X)$   
 $PD = \text{ProbDistUnivKernel}(X, \text{param1}, \text{val1}, \text{param2}, \text{val2}, \dots)$

## Description

---

**Tip** Although you can use this constructor function to create a ProbDistUnivKernel object, using the `fitdist` function is an easier way to create the ProbDistUnivKernel object.

---

$PD = \text{ProbDistUnivKernel}(X)$  creates  $PD$ , a ProbDistUnivKernel object, which represents a nonparametric probability distribution, based on a normal kernel smoothing function.

$PD = \text{ProbDistUnivKernel}(X, \text{param1}, \text{val1}, \text{param2}, \text{val2}, \dots)$  specifies optional parameter name/value pairs, as described in the Parameter/Values table. Parameter and value names are case insensitive.

**Input Arguments**  $X$  A column vector of data.

---

**Note** Any NaN values in  $X$  are ignored by the fitting calculations.

---



Parameter	Values
'censoring'	<p>A Boolean vector the same size as <math>X</math>, containing 1s when the corresponding elements in <math>X</math> are right-censored observations and 0s when the corresponding elements are exact observations. Default is a vector of 0s.</p> <hr/> <p><b>Note</b> Any NaN values in this censoring vector are ignored by the fitting calculations.</p> <hr/>
'frequency'	<p>A vector the same size as <math>X</math>, containing nonnegative integers specifying the frequencies for the corresponding elements in <math>X</math>. Default is a vector of 1s.</p> <hr/> <p><b>Note</b> Any NaN values in this frequency vector are ignored by the fitting calculations.</p> <hr/>
'kernel'	<p>A string specifying the type of kernel smoother to use. Choices are:</p> <ul style="list-style-type: none"><li>• 'normal' (default)</li><li>• 'box'</li><li>• 'triangle'</li><li>• 'epanechnikov'</li></ul>

# ProbDistUnivKernel

---

Parameter	Values
'support'	Any of the following to specify the support: <ul style="list-style-type: none"><li>• 'unbounded' — Default. If the density can extend over the whole real line.</li><li>• 'positive' — To restrict it to positive values.</li><li>• A two-element vector giving finite lower and upper limits for the support of the density.</li></ul>
'width'	A value specifying the bandwidth of the kernel smoothing window. The default is optimal for estimating normal densities, but you may want to choose a smaller value to reveal features such as multiple modes.

## Output Arguments

*PD*

An object in the ProbDistUnivKernel class, which is derived from the ProbDist class. It represents a nonparametric probability distribution.

## References

[1] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.

## See Also

fitdist | ksdensity

<b>Superclasses</b>	ProbDistParametric	
<b>Purpose</b>	Object representing univariate parametric probability distribution	
<b>Description</b>	A ProbDistUnivParam object represents a univariate parametric probability distribution. You create this object by using the constructor ( <code>ProbDistUnivParam.ProbDistUnivParam</code> ) and supplying parameter values, or by using the <code>fitdist</code> function to fit the distribution to data.	
<b>Construction</b>	ProbDistUnivParam	Construct ProbDistUnivParam object
<b>Methods</b>	<code>cdf</code>	Return cumulative distribution function (CDF) for ProbDist object
	<code>icdf</code>	Return inverse cumulative distribution function (ICDF) for ProbDistUnivParam object
	<code>iqr</code>	Return interquartile range (IQR) for ProbDistUnivParam object
	<code>mean</code>	Return mean of ProbDistUnivParam object
	<code>median</code>	Return median of ProbDistUnivParam object
	<code>paramci</code>	Return parameter confidence intervals of ProbDistUnivParam object
	<code>pdf</code>	Return probability density function (PDF) for ProbDist object
	<code>random</code>	Generate random number drawn from ProbDist object

# ProbDistUnivParam

---

std	Return standard deviation of ProbDistUnivParam object
var	Return variance of ProbDistUnivParam object

---

**Note** Some of the above methods are inherited from the ProbDistParametric class.

---

## Properties

DistName	Read-only string containing probability distribution name of ProbDist object
InputData	Read-only structure containing information about input data to ProbDist object
NLogL	Read-only value specifying negative log likelihood for input data to ProbDistParametric object
NumParams	Read-only value specifying number of parameters of ProbDistParametric object
ParamCov	Read-only covariance matrix of parameter estimates of ProbDistParametric object
ParamDescription	Read-only cell array specifying descriptions of parameters of ProbDistParametric object
ParamIsFixed	Read-only logical array specifying fixed parameters of ProbDistParametric object

ParamNames	Read-only cell array specifying names of parameters of ProbDistParametric object
Params	Read-only array specifying values of parameters of ProbDistParametric object
Support	Read-only structure containing information about support of ProbDist object

---

**Note** The above properties are inherited from the ProbDistParametric class.

---

---

**Note** Parameter values are also properties. For example, if you create PD, a univariate parametric probability distribution object that represents a normal distribution, then PD.mu and PD.sigma are properties that give the values of the mu and sigma parameters.

---

## Copy Semantics

Value. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

## References

[1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.

[2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

## See Also

fitdist | ProbDist | ProbDistParametric | ProbDistUnivParam.ProbDistUnivParam

# ProbDistUnivParam

---

## How To

- Appendix B, “Distribution Reference”

**Purpose** Construct ProbDistUnivParam object

**Syntax** `PD = ProbDistUnivParam(DistName, Params)`

**Description** `PD = ProbDistUnivParam(DistName, Params)` creates `PD`, a ProbDistUnivParam object, which represents a probability distribution. This distribution is defined by the parametric distribution specified by `DistName`, with parameters specified by the numeric vector `Params`.

**Input Arguments**

*DistName* A string specifying a distribution. Choices are:

- 'beta'
- 'binomial'
- 'birnbaumsaunders'
- 'exponential'
- 'extreme value' or 'ev'
- 'gamma'
- 'generalized extreme value' or 'gev'
- 'generalized pareto' or 'gp'
- 'inversegaussian'
- 'logistic'
- 'loglogistic'
- 'lognormal'
- 'nakagami'
- 'negative binomial' or 'nbin'
- 'normal'
- 'poisson'

# ProbDistUnivParam

---

- 'rayleigh'
- 'rician'
- 'tlocationscale'
- 'weibull' or 'wbl'

For more information on these parametric distributions, see Appendix B, “Distribution Reference”.

*Params*

Numeric vector of distribution parameters. The number and type of parameters depends on the distribution you specify with *DistName*. For information on parameters for each distribution type, see Appendix B, “Distribution Reference”.

## Output Arguments

*PD*

An object in the ProbDistUnivParam class, which is derived from the ProbDist class. It represents a parametric probability distribution.

## Examples

- 1 Create an object representing a normal distribution with a mean of 100 and a standard deviation of 10.

```
pd = ProbDistUnivParam('normal',[100 10])
```

```
pd =
```

```
normal distribution
```

```
mu = 100  
sigma = 10
```



**2** Generate a 4-by-5 matrix of random values from this distribution.

```
random(pd,4,5)
```

```
ans =
```

```
105.3767 103.1877 135.7840 107.2540 98.7586  
118.3389 86.9231 127.6944 99.3695 114.8970  
77.4115 95.6641 86.5011 107.1474 114.0903  
108.6217 103.4262 130.3492 97.9503 114.1719
```

## References

- [1] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.
- [2] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.

## See Also

fitdist

## How To

- Appendix B, “Distribution Reference”

# probplot

---

**Purpose** Probability plots

**Syntax**

```
probplot(Y)
probplot(distribution,Y)
probplot(Y,cens,freq)
probplot(ax,Y)
probplot(...,'noref')
probplot(ax,PD)
probplot(ax,fun,params)
h = probplot(...)
```

**Description** `probplot(Y)` produces a normal probability plot comparing the distribution of the data `Y` to the normal distribution. `Y` can be a single vector, or a matrix with a separate sample in each column. The plot includes a reference line useful for judging whether the data follow a normal distribution.

`probplot` uses midpoint probability plotting positions. The  $i^{\text{th}}$  sorted value from a sample of size  $N$  is plotted against the midpoint in the jump of the empirical CDF on the  $y$  axis. With uncensored data, that midpoint is  $(i-0.5)/N$ . With censored data (see below), the  $y$  value is more complicated to compute.

`probplot(distribution,Y)` creates a probability plot for the distribution specified by *distribution*. Acceptable strings for *distribution* are:

- 'exponential' — Exponential probability plot (nonnegative values)
- 'extreme value' — Extreme value probability plot (all values)
- 'lognormal' — Lognormal probability plot (positive values)
- 'normal' — Normal probability plot (all values)
- 'rayleigh' — Rayleigh probability plot (positive values)
- 'weibull' — Weibull probability plot (positive values)

The  $y$  axis scale is based on the selected distribution. The  $x$  axis has a log scale for the Weibull and lognormal distributions, and a linear scale for the others.

Not all distributions are appropriate for all data sets, and `probplot` will error when asked to create a plot with a data set that is inappropriate for a specified distribution. Appropriate data ranges for each distribution are given parenthetically in the list above.

`probplot(Y,cens,freq)` or `probplot(distname,Y,cens,freq)` requires a vector  $Y$ . `cens` is a vector of the same size as  $Y$  and contains 1 for observations that are right-censored and 0 for observations that are observed exactly. `freq` is a vector of the same size as  $Y$ , containing integer frequencies for the corresponding elements in  $Y$ .

`probplot(ax,Y)` takes a handle `ax` to an existing probability plot, and adds additional lines for the samples in  $Y$ . `ax` is a handle for a set of axes.

`probplot(..., 'noref')` omits the reference line.

`probplot(ax,PD)` takes a probability distribution object, `PD`, and adds a fitted line to the axes specified by `ax` to represent the probability distribution specified by `PD`. `PD` is a `ProbDist` object of the `ProbDistUnivParam` class or `ProbDistUnivKernel` class.

`probplot(ax,fun,params)` takes a function `fun` and a set of parameters, `params`, and adds fitted lines to the axes of an existing probability plot specified by `ax`. `fun` is a function handle to a cdf function, specified with `@` (for example, `@wblcdf`). `params` is the set of parameters required to evaluate `fun`, and is specified as a cell array or vector. The function must accept a vector of  $X$  values as its first argument, then the optional parameters, and must return a vector of cdf values evaluated at  $X$ .

`h = probplot(...)` returns handles to the plotted lines.

## Examples

### Test Data for Weibull Distribution Using `probplot`

This example shows how to use a probability plot to test whether sample data comes from a Weibull distribution.

# probplot

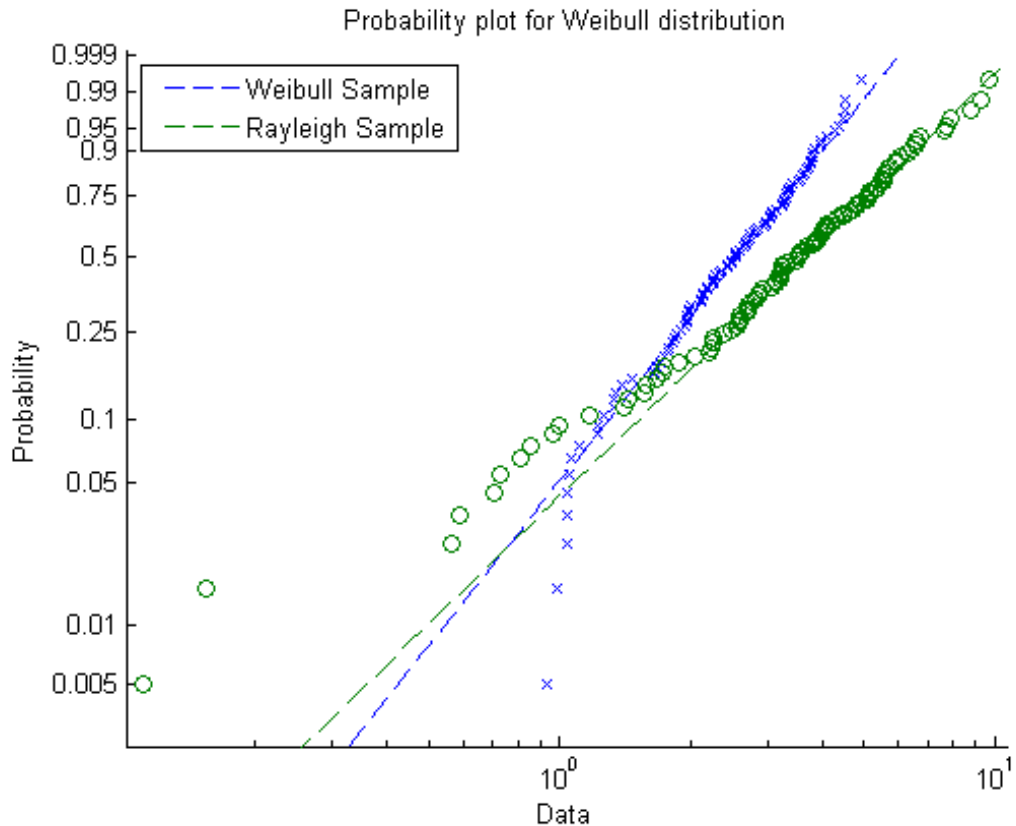
---

Generate sample data. The sample `x1` contains 100 random numbers from a Weibull distribution with scale parameter `A = 3` and shape parameter `B = 3`. The sample `x2` contains 100 random numbers from a Rayleigh distribution with scale parameter `B = 3`.

```
rng('default'); % For reproducibility
x1 = wblrnd(3,3,100,1);
x2 = raylrnd(3,100,1);
```

Create a probability plot to assess whether the data in `x1` and `x2` comes from a Weibull distribution.

```
probplot('weibull',[x1 x2])
legend('Weibull Sample','Rayleigh Sample','Location','NW')
```



The probability plot shows that the data in x1 comes from a Weibull distribution, while the data in x2 does not.

### Test Data for Normal Distribution Using probplot

This example shows how to use a probability plot to test whether sample data comes from a normal distribution.

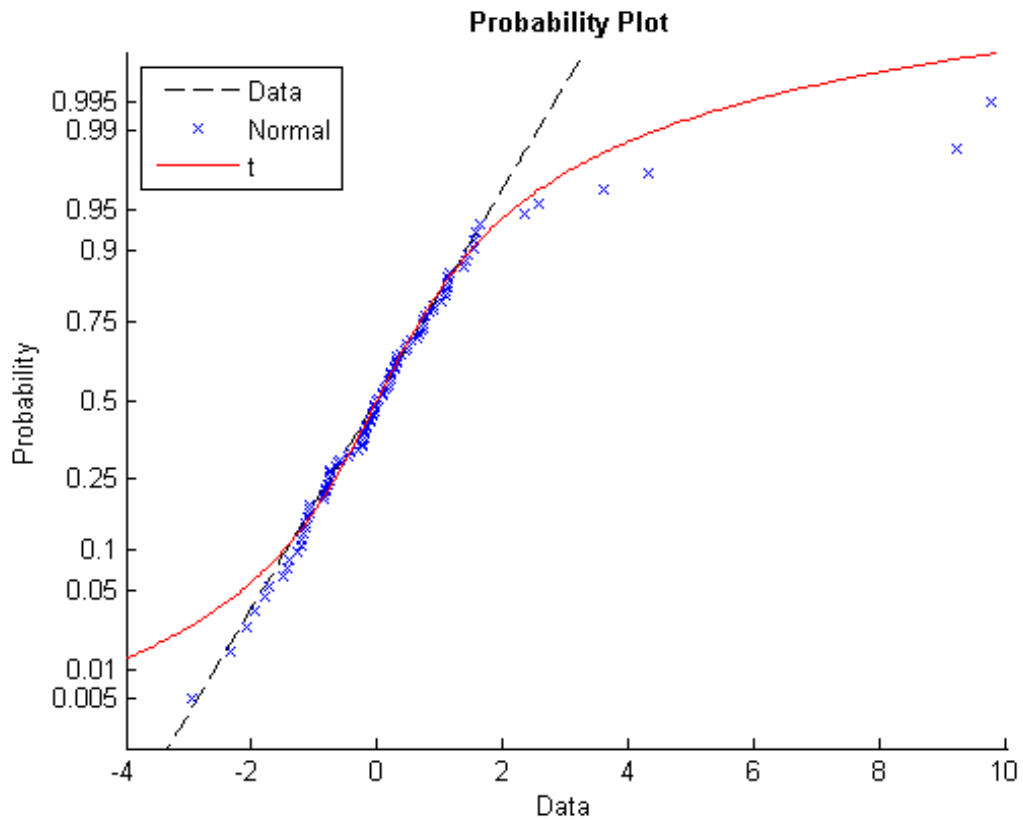
Generate sample data containing about 20% outliers in the tails. The left tail of the sample data contains 10 values randomly generated from an exponential distribution with parameter  $\mu = 1$ . The right tail

contains 10 values randomly generated from an exponential distribution with parameter  $\mu = 5$ . The center of the sample data contains 80 values randomly generated from a standard normal distribution.

```
rng('default'); % For reproducibility
left_tail = -exprnd(1,10,1);
right_tail = exprnd(5,10,1);
center = randn(80,1);
data = [left_tail;center;right_tail];
```

Create a probability plot to assess whether the data in `data` comes from a normal distribution. Plot a  $t$  location-scale curve on the same figure to compare with data.

```
probplot(data);
p = mle(data,'dist','tlo');
t = @(data,mu,sig,df)cdf('tlocationscale',data,mu,sig,df);
h = probplot(gca,t,p);
set(h,'color','r','linestyle','-')
title('\bf Probability Plot')
legend('Data','Normal','t','Location','NW')
```



The plot shows that neither the normal line nor the  $t$  location-scale curve fit the tails very well because of the outliers.

**See Also**

`normplot` | `ecdf` | `wblplot`

**Purpose** Procrustes analysis

**Syntax**

```
d = procrustes(X,Y)
[d,Z] = procrustes(X,Y)
[d,Z,transform] = procrustes(X,Y)
[...] = procrustes(...,'scaling',flag)
[...] = procrustes(...,'reflection',flag)
```

**Description** `d = procrustes(X,Y)` determines a linear transformation (translation, reflection, orthogonal rotation, and scaling) of the points in matrix `Y` to best conform them to the points in matrix `X`. The goodness-of-fit criterion is the sum of squared errors. `procrustes` returns the minimized value of this dissimilarity measure in `d`. `d` is standardized by a measure of the scale of `X`, given by:

```
sum(sum((X-repmat(mean(X,1),size(X,1),1)).^2,1))
```

That is, the sum of squared elements of a centered version of `X`. However, if `X` comprises repetitions of the same point, the sum of squared errors is not standardized.

`X` and `Y` must have the same number of points (rows), and `procrustes` matches `Y(i)` to `X(i)`. Points in `Y` can have smaller dimension (number of columns) than those in `X`. In this case, `procrustes` adds columns of zeros to `Y` as necessary.

`[d,Z] = procrustes(X,Y)` also returns the transformed `Y` values.

`[d,Z,transform] = procrustes(X,Y)` also returns the transformation that maps `Y` to `Z`. `transform` is a structure array with fields:

- `c` — Translation component
- `T` — Orthogonal rotation and reflection component
- `b` — Scale component

That is:

```
c = transform.c;
T = transform.T;
```



```
b = transform.b;
```

```
Z = b*Y*T + c;
```

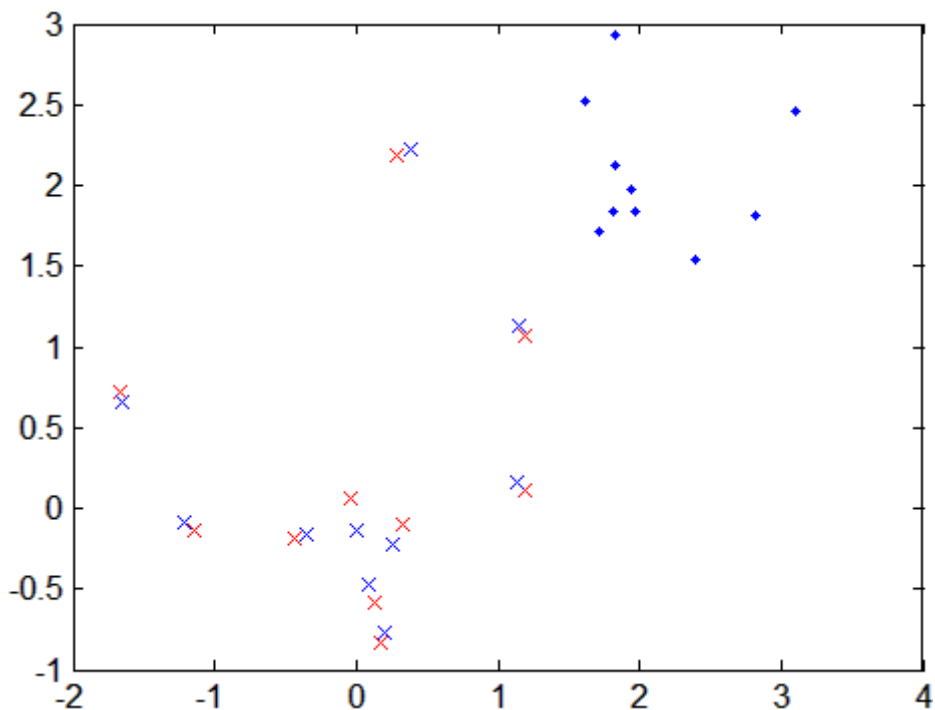
[...] = `procrustes(..., 'scaling', flag)`, when `flag` is `false`, allows you to compute the transformation without a scale component (that is, with `b` equal to 1). The default `flag` is `true`.

[...] = `procrustes(..., 'reflection', flag)`, when `flag` is `false`, allows you to compute the transformation without a reflection component (that is, with `det(T)` equal to 1). The default `flag` is `'best'`, which computes the best-fitting transformation, whether or not it includes a reflection component. A `flag` of `true` forces the transformation to be computed with a reflection component (that is, with `det(T)` equal to -1)

## Examples

This example creates some random points in two dimensions, then rotates, scales, translates, and adds some noise to those points. It uses `procrustes` to conform `Y` to `X`, then plots the original `X` and `Y` with the transformed `Y`.

```
n = 10;
X = normrnd(0,1,[n 2]);
S = [0.5 -sqrt(3)/2; sqrt(3)/2 0.5];
Y = normrnd(0.5*X*S+2,0.05,n,2);
[d,Z,tr] = procrustes(X,Y);
plot(X(:,1),X(:,2),'rx',...
      Y(:,1),Y(:,2),'b.',...
      Z(:,1),Z(:,2),'bx');
```



## References

[1] Kendall, David G. "A Survey of the Statistical Theory of Shape." *Statistical Science*. Vol. 4, No. 2, 1989, pp. 87–99.

[2] Bookstein, Fred L. *Morphometric Tools for Landmark Data*. Cambridge, UK: Cambridge University Press, 1991.

[3] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.

## See Also

`cmdscales` | `factoran`

## Purpose

Profile likelihood function for probability distribution object

## Syntax

```
[ll,param] = proflik(pd,pnum)
[ll,param] = proflik(pd,pnum,'Display',display)
[ll,param] = proflik(pd,pnum,setparam)
[ll,param] = proflik(pd,pnum,setparam,'Display',display)
[ll,param,other] = proflik( ___ )
```

## Description

[ll,param] = proflik(pd,pnum) returns a vector ll of loglikelihood values and a vector param of corresponding parameter values for the parameter in the position indicated by pnum.

[ll,param] = proflik(pd,pnum,'Display',display) returns the loglikelihood values and corresponding parameter values, and plots the profile likelihood overlaid on an approximation of the loglikelihood.

[ll,param] = proflik(pd,pnum,setparam) returns the loglikelihood values and corresponding parameter values as specified by setparam.

[ll,param] = proflik(pd,pnum,setparam,'Display',display) returns the loglikelihood values and corresponding parameter values as specified by setparam, and plots the profile likelihood overlaid on an approximation of the loglikelihood.

[ll,param,other] = proflik( \_\_\_ ) also returns a matrix other containing the values of the other parameters that maximize the likelihood, using any of the input arguments from the previous syntaxes.

## Input Arguments

### pd - Probability distribution

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

### pnum - Parameter number

positive integer value

# prob.ToolboxFittableParametricDistribution.proflik

---

Parameter number for which to compute the profile likelihood, specified as a positive integer value corresponding to the position of the desired parameter in the parameter name vector. For example, a Weibull distribution has a parameter name vector {'A', 'B'}, so specify `pnum` as 2 to compute the profile likelihood for B.

## Data Types

single | double

## setparam - Parameter value restriction

scalar value | vector of scalar values

Parameter value restriction, specified as a scalar value or a vector of such values. If you do not specify `setparam`, `proflik` chooses the values for output vector `param` based on the default confidence interval method for the probability distribution `pd`. If the parameter can take only restricted values, and if the confidence interval violates that restriction, you can use `setparam` to specify valid values.

**Example:** [3,3.5,4]

## display - Display toggle

'off' (default) | 'on'

Display toggle, specified as either 'on' or 'off'. Specify 'on' to display a plot of the profile loglikelihood overlaid on an approximation of the loglikelihood. Specify 'off' to omit the display. The approximation is based on a Taylor series expansion around the estimated parameter value, as a function of the parameter in position `pnum` or its logarithm. The intersection of the curves with the horizontal dotted line marks the endpoints of 95% confidence intervals.

## Output Arguments

### ll - Loglikelihood values

vector

Loglikelihood values, returned as a vector. The loglikelihood is the value of the likelihood with the parameter in position `pnum` set to the values in `param`, maximized over the remaining parameters.

### param - Parameter values

vector

Parameter values corresponding to the loglikelihood values in `ll`, returned as a vector. If you specify parameter values using `setparam`, then `param` is equal to `setparam`.

### other - Other parameter values

matrix

Other parameter values that maximize the likelihood, returned as a matrix. Each row of `other` contains the values for all parameters except the parameter in position `pnum`.

## Examples

### Profile Likelihood of a Distribution Parameter

Load the sample data. Create a probability distribution object by fitting a Weibull distribution to the miles per gallon (MPG) data.

```
load carsmall;
pd = fitdist(MPG,'Weibull')

pd =

WeibullDistribution

Weibull distribution
A = 26.5079    [24.8333, 28.2954]
B = 3.27193   [2.79441, 3.83104]
```

View the parameter names for the distribution.

```
pd.ParameterNames
```

# prob.ToolboxFittableParametricDistribution.proflik

---

```
ans =
```

```
    'A'    'B'
```

For the Weibull distribution, A is in position 1, and B is in position 2.

Compute the profile likelihood for B, which is in position `pnum = 2`.

```
[ll,param] = proflik(pd,2);
```

Display the loglikelihood values for the estimated values of B.

```
[ll',param']
```

```
ans =
```

```
-329.9688    2.7132  
-329.4312    2.7748  
-328.9645    2.8365  
-328.5661    2.8981  
-328.2340    2.9597  
-327.9658    3.0213  
-327.7596    3.0830  
-327.6135    3.1446  
-327.5256    3.2062  
-327.4943    3.2678  
-327.5178    3.3295  
-327.5946    3.3911  
-327.7233    3.4527  
-327.9023    3.5143  
-328.1303    3.5760  
-328.4060    3.6376  
-328.7281    3.6992  
-329.0956    3.7608  
-329.5071    3.8224  
-329.9617    3.8841  
-330.4583    3.9457
```

These results show that the profile loglikelihood is maximized between the estimated B values of 3.2678 and 3.3295, which correspond to loglikelihood values -327.4943 and -327.5178. From the earlier fit, the MLE of B is 3.27193, which is in this interval as expected.

## Profile Likelihood With Restricted Parameter Values

Load the sample data. Create a probability distribution object by fitting a generalized extreme value distribution to the miles per gallon (MPG) data.

```
load carsmall;
pd = fitdist(MPG,'GeneralizedExtremeValue')

pd =

GeneralizedExtremeValueDistribution

Generalized Extreme Value distribution
    k = -0.207765    [-0.381674, -0.0338564]
   sigma =  7.49674    [6.31755, 8.89603]
    mu =  20.6233    [18.8859, 22.3606]
```

View the parameter names for the distribution.

```
pd.ParameterNames

ans =

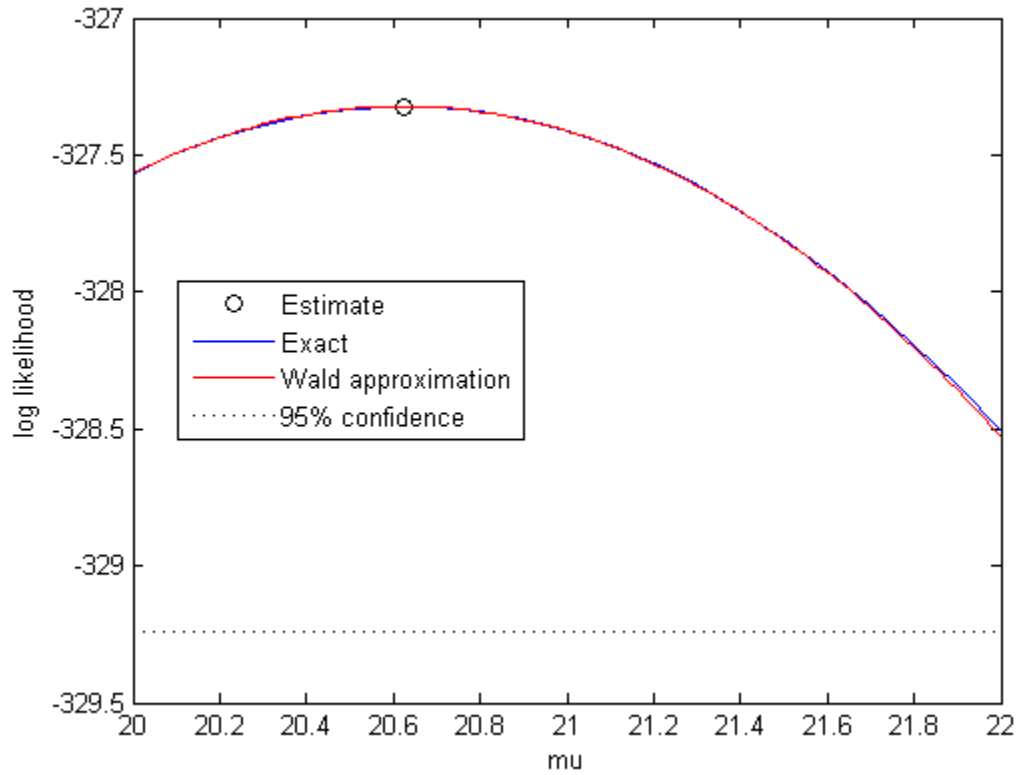
    'k'    'sigma'    'mu'
```

For the generalized extreme value distribution, k is in position 1, sigma is in position 2, and mu is in position 3.

Compute the profile likelihood for mu, which is in position `pnum = 3`. Restrict the computation to parameter values from 20 to 22, and display the plot.

```
[ll,param,other] = proflik(pd,3,20:.1:22,'display','on');
```

# prob.ToolboxFittableParametricDistribution.proflik



The plot shows the estimated value for the parameter  $\mu$  that maximizes the loglikelihood.

Display the loglikelihood values for the estimated values of  $\mu$ , and the values of the other distribution parameters that maximize the corresponding loglikelihood.

```
[ll',param',other]
```

```
ans =
```



-327.5706	20.0000	-0.1803	7.4087
-327.4971	20.1000	-0.1846	7.4218
-327.4364	20.2000	-0.1890	7.4354
-327.3887	20.3000	-0.1934	7.4493
-327.3538	20.4000	-0.1978	7.4636
-327.3317	20.5000	-0.2023	7.4783
-327.3223	20.6000	-0.2067	7.4932
-327.3257	20.7000	-0.2112	7.5084
-327.3418	20.8000	-0.2156	7.5240
-327.3706	20.9000	-0.2201	7.5399
-327.4119	21.0000	-0.2245	7.5560
-327.4659	21.1000	-0.2289	7.5723
-327.5324	21.2000	-0.2333	7.5889
-327.6113	21.3000	-0.2378	7.6057
-327.7027	21.4000	-0.2422	7.6228
-327.8065	21.5000	-0.2465	7.6400
-327.9227	21.6000	-0.2509	7.6575
-328.0511	21.7000	-0.2553	7.6751
-328.1917	21.8000	-0.2596	7.6930
-328.3446	21.9000	-0.2639	7.7111
-328.5095	22.0000	-0.2682	7.7293

The first column contains the loglikelihood value that corresponds to the estimate of  $\mu$  in the second column. The loglikelihood is maximized between the parameter values 20.6000 and 20.7000, corresponding to loglikelihood values -327.3223 and -327.3257. The third column contains the value of  $k$  that maximizes the corresponding loglikelihood for  $\mu$ . The fourth column contains the value of  $\sigma$  that maximizes the corresponding loglikelihood for  $\mu$ .

## See Also

`makedist` | `fitdist` | `dfittool`

# CompactTreeBagger.proximity

---

**Purpose** Proximity matrix for data

**Syntax** `prox = proximity(B,X)`

**Description** `prox = proximity(B,X)` computes a numeric matrix of size Nobs-by-Nobs of proximities for data X, where Nobs is the number of observations (rows) in X. Proximity between any two observations in the input data is defined as a fraction of trees in the ensemble B for which these two observations land on the same leaf. This is a symmetric matrix with ones on the diagonal and off-diagonal elements ranging from 0 to 1.

# TreeBagger.Proximity property

---

**Purpose**

Proximity matrix for observations

**Description**

The Proximity property is a numeric matrix of size Nobs-by-Nobs, where Nobs is the number of observations in the training data, containing measures of the proximity between observations. For any two observations, their proximity is defined as the fraction of trees for which these observations land on the same leaf. This is a symmetric matrix with 1s on the diagonal and off-diagonal elements ranging from 0 to 1.

**See Also**

`CompactTreeBagger.proximity` | `classregtree.varimportance`

# ClassificationTree.prune

---

**Purpose** Produce sequence of subtrees by pruning

**Syntax**  
`tree1 = prune(tree)`  
`tree1 = prune(tree,Name,Value)`

**Description** `tree1 = prune(tree)` creates a copy of the classification tree `tree` with its optimal pruning sequence filled in.

`tree1 = prune(tree,Name,Value)` creates a pruned tree with additional options specified by one `Name,Value` pair argument. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

**Tips**

- `tree1 = prune(tree)` returns the decision tree `tree1` that is the full, unpruned tree, but with optimal pruning information added. This is useful only if you created `tree` by pruning another tree, or by using the `ClassificationTree.fit` method with pruning set 'off'. If you plan to prune a tree multiple times along the optimal pruning sequence, it is more efficient to create the optimal pruning sequence first.

**Input Arguments**

**tree**  
A classification tree created with `ClassificationTree.fit`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### **'alpha'**

A numeric scalar. `prune` prunes `tree` to the specified value of the pruning cost.

### **'level'**

A numeric scalar from 0 (no pruning) to the largest pruning level of this tree `max(tree.PruneList)`. `prune` returns the tree pruned to this level.

## 'nodes'

A numeric vector with elements from 1 to `tree.NumNodes`. Any `tree` branch nodes listed in `nodes` become leaf nodes in `tree1`, unless their parent nodes are also pruned.

## Output Arguments

### `tree1`

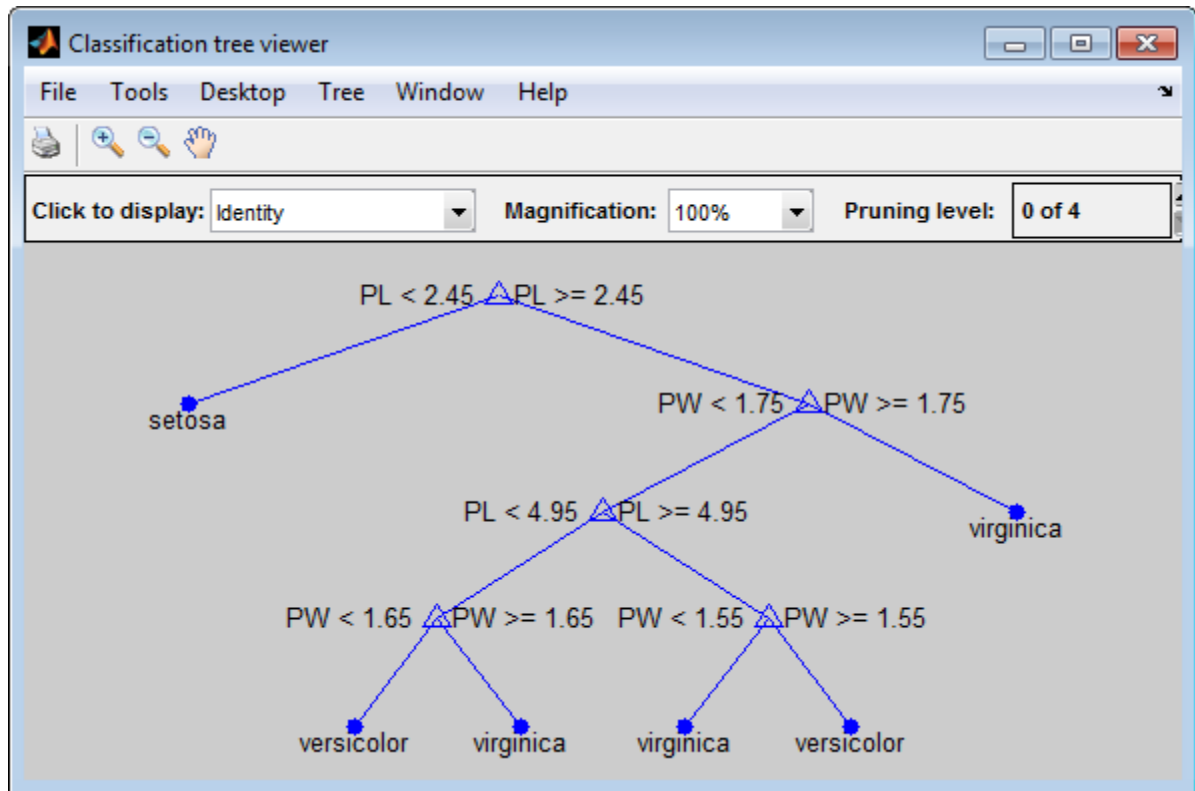
A classification tree.

## Examples

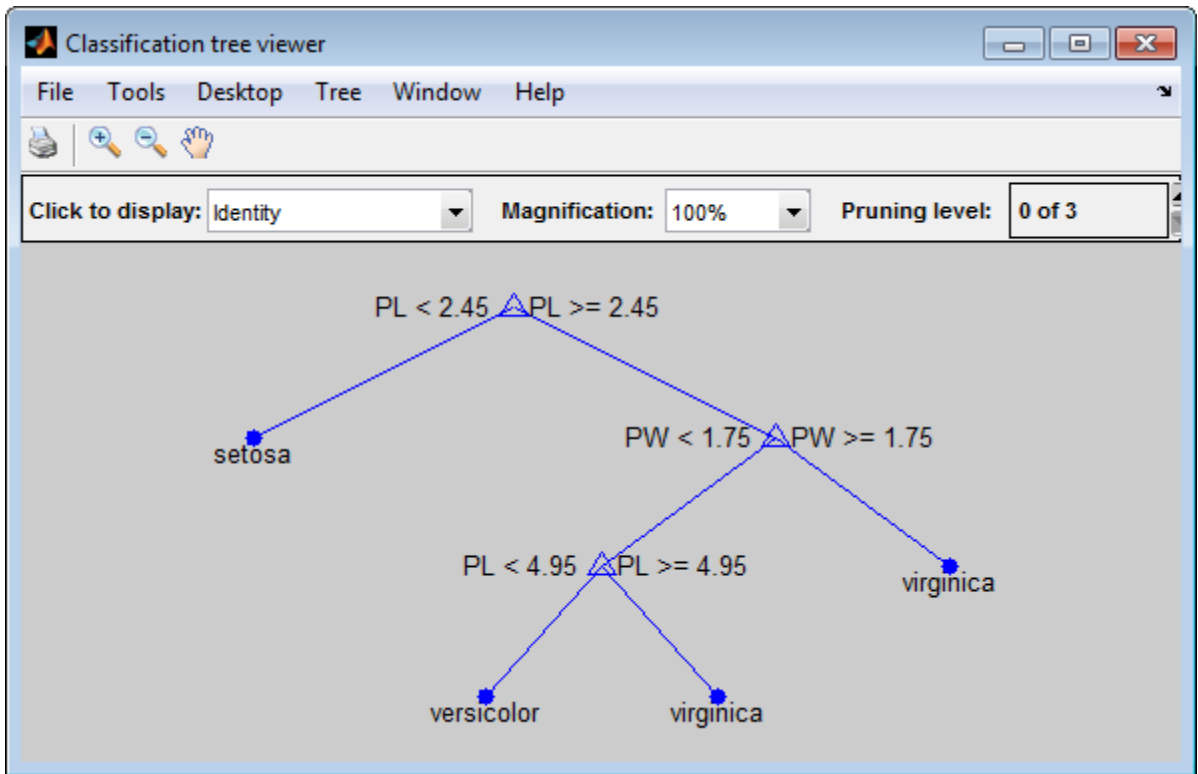
Display a full tree for Fisher's iris data, as well as the next largest tree from the optimal pruning sequence:

```
load fisheriris;
varnames = {'SL' 'SW' 'PL' 'PW'};
t1 = ClassificationTree.fit(meas,species,...
    'minparent',5,'predictorNames',varnames);
view(t1,'mode','graph');
```

# ClassificationTree.prune



```
t2 = prune(t1,'level',1);  
view(t2,'mode','graph');
```



**See Also**      ClassificationTree.fit

# classregtree.prune

---

**Purpose** Prune tree

**Syntax**

```
t2 = prune(t1, 'level', level)
t2 = prune(t1, 'nodes', nodes)
t2 = prune(t1)
```

**Description** `t2 = prune(t1, 'level', level)` takes a decision tree `t1` and a pruning level `level`, and returns the decision tree `t2` pruned to that level. If `level` is 0, there is no pruning. Trees are pruned based on an optimal pruning scheme that first prunes branches giving less improvement in error cost.

`t2 = prune(t1, 'nodes', nodes)` prunes the nodes listed in the `nodes` vector from the tree. Any `t1` branch nodes listed in `nodes` become leaf nodes in `t2`, unless their parent nodes are also pruned. Use `view` to display the node numbers for any node you select.

`t2 = prune(t1)` returns the decision tree `t2` that is the full, unpruned `t1`, but with optimal pruning information added. This is useful only if `t1` is created by pruning another tree, or by using the `classregtree` function with the `'prune'` parameter set to `'off'`. If you plan to prune a tree multiple times along the optimal pruning sequence, it is more efficient to create the optimal pruning sequence first.

Pruning is the process of reducing a tree by turning some branch nodes into leaf nodes and removing the leaf nodes under the original branch.

**Examples** Display the full tree for Fisher's iris data:

```
load fisheriris;

t1 = classregtree(meas, species, ...
                 'names', {'SL' 'SW' 'PL' 'PW'}, ...
                 'splitmin', 5)

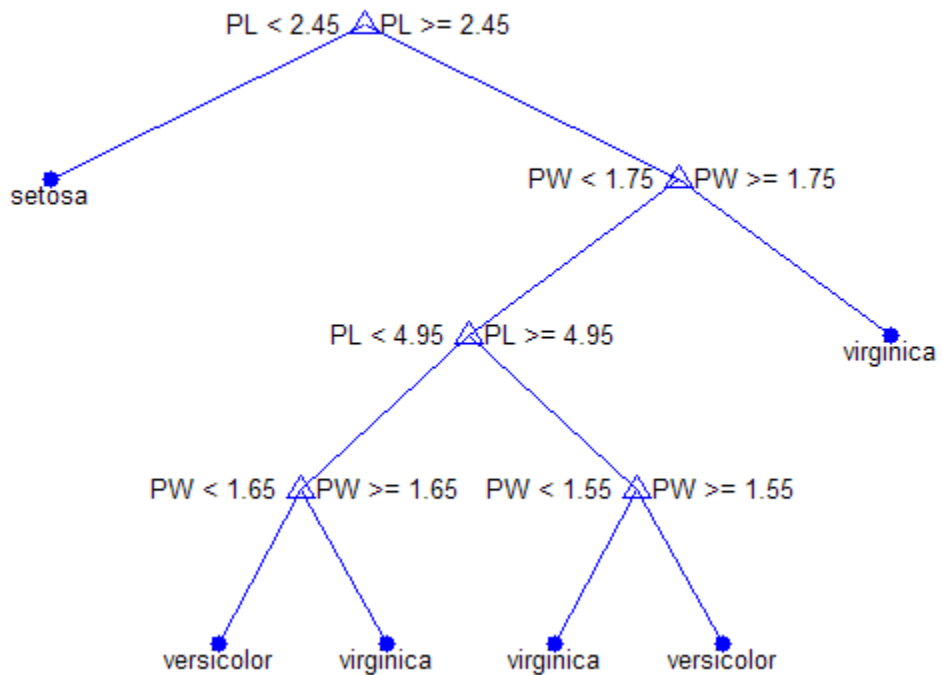
t1 =
Decision tree for classification
1 if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2 class = setosa
```



```
3 if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4 if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5 class = virginica
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica
```

```
view(t1)
```

Click to display:  Magnification:  Pruning level:



# classregtree.prune

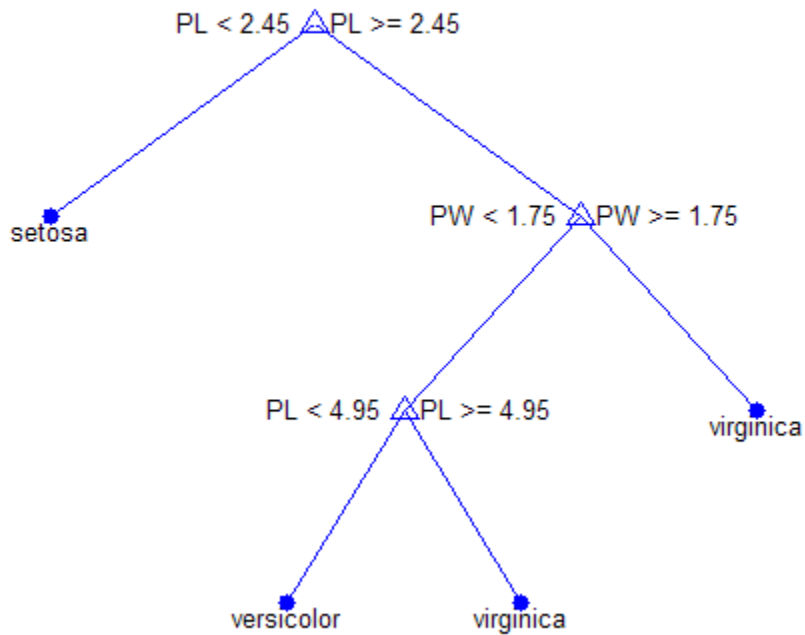
---

Display the next largest tree from the optimal pruning sequence:

```
t2 = prune(t1,'level',1)
t2 =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  class = versicolor
7  class = virginica

view(t2)
```

Click to display:  Magnification:  Pruning level:



## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[classregtree](#) | [test](#) | [view](#)

# RegressionTree.prune

---

**Purpose** Produce sequence of subtrees by pruning

**Syntax**  
`tree1 = prune(tree)`  
`tree1 = prune(tree,Name,Value)`

**Description** `tree1 = prune(tree)` creates a copy of the regression tree `tree` with its optimal pruning sequence filled in.

`tree1 = prune(tree,Name,Value)` creates a pruned tree with additional options specified by one `Name,Value` pair argument. You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

**Tips**

- `tree1 = prune(tree)` returns the decision tree `tree1` that is the full, unpruned tree, but with optimal pruning information added. This is useful only if you created `tree` by pruning another tree, or by using the `RegressionTree.fit` with pruning set 'off'. If you plan to prune a tree multiple times along the optimal pruning sequence, it is more efficient to create the optimal pruning sequence first.

**Input Arguments**

**tree**  
A regression tree created with `RegressionTree.fit`.

## **Name-Value Pair Arguments**

Optional comma-separated pair of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify only one name-value pair argument.

### **'alpha'**

A numeric scalar from 0 (no pruning) to 1 (prune to one node). Prunes to minimize the sum of (`alpha` times the number of leaf nodes) and a cost (mean squared error).

### **'level'**

A numeric scalar from 0 (no pruning) to the largest pruning level of this tree `max(tree.PruneList)`. `prune` returns the tree pruned to this level.

## 'nodes'

A numeric vector with elements from 1 to `tree.NumNodes`. Any `tree` branch nodes listed in `nodes` become leaf nodes in `tree1`, unless their parent nodes are also pruned.

## Output Arguments

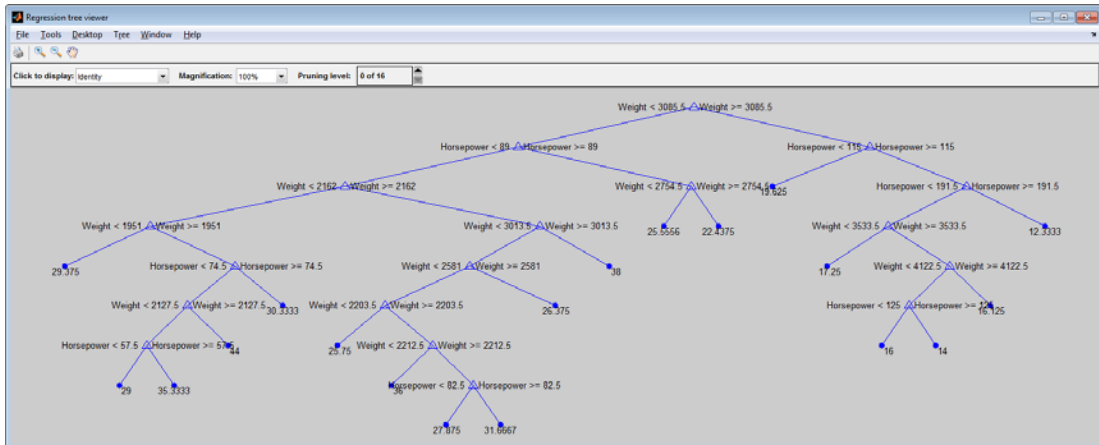
**tree1**

A regression tree.

## Examples

Display a full tree for the `carsmall` data, as well as the tree pruned to level 10:

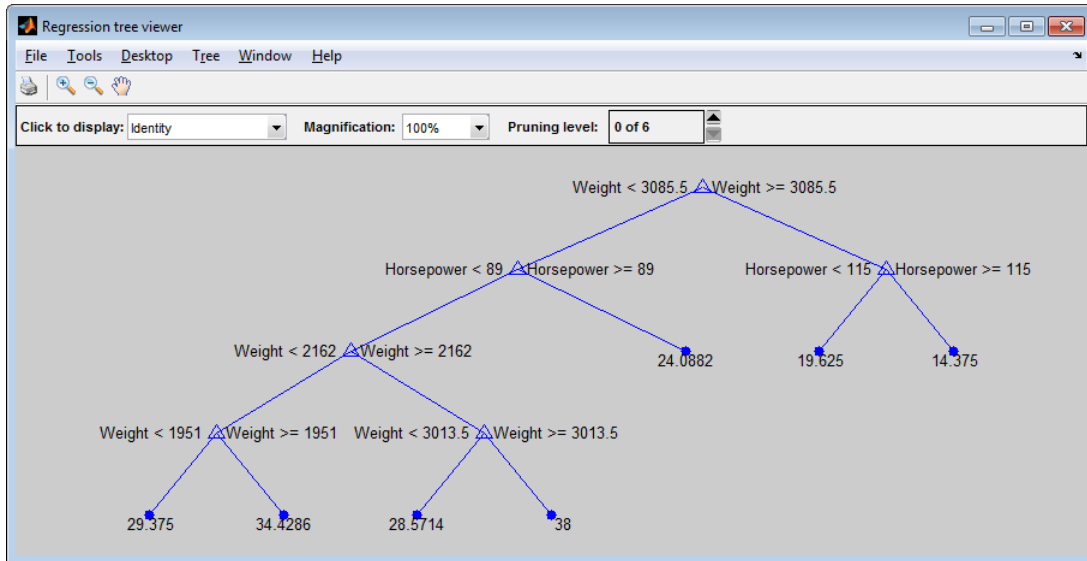
```
load carsmall;
varnames = {'Weight' 'Horsepower'};
t1 = RegressionTree.fit([Weight Horsepower],MPG,...
    'predictornames',varnames)
view(t1,'mode','graph');
```



```
t2 = prune(t1,'level',10);
```

# RegressionTree.prune

```
view(t2, 'mode', 'graph');
```



**See Also**

`RegressionTree.fit`

**Purpose** Flag to prune trees

**Description** The Prune property is true if decision trees are pruned and false if they are not. Pruning decision trees is not recommended for ensembles. The default value is false.

**See Also** `classregtree.prune`

# classregtree.prunelist

---

**Purpose** Pruning levels for decision tree nodes

**Syntax** P = prunelist(T)  
P = prunelist(T,J)

**Description** P = prunelist(T) returns an  $n$ -element numeric vector with the pruning levels in each node of the tree T, where  $n$  is the number of nodes. When you call `prune(T, 'level', level)`, nodes with the pruning levels below *level* are pruned, and nodes with the pruning levels greater or equal to *level* are not pruned.

P = prunelist(T,J) takes an array J of node numbers and returns the pruning levels for the specified nodes.

**See Also** classregtree | numnodes



**Purpose** Generate quasi-random points from stream

**Syntax**  
`x = grand(q)`  
`X = grand(q,n)`

**Description** `x = grand(q)` returns the next value `x` in the quasi-random number stream `q` of the `grandstream` class. `x` is a 1-by- $d$  vector, where  $d$  is the dimension of the stream. The command sets `q.State` to the index in the underlying point set of the next value to be returned.

`X = grand(q,n)` returns the next  $n$  values `X` in an  $n$ -by- $d$  matrix.

Objects `q` of the `grandstream` class encapsulate properties of a specified quasi-random number stream. Values of the stream are not generated and stored in memory until `q` is accessed using `grand`.

**Examples** Use `grandstream` to construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = grandstream('halton',3,'Skip',1e3,'Leap',1e2)
q =
    Halton quasi-random stream in 3 dimensions
    Point set properties:
        Skip : 1000
        Leap : 100
        ScrambleMethod : none
```

```
nextIdx = q.State
nextIdx =
     1
```

Use `grand` to generate two samples of size four:

```
X1 = grand(q,4)
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
```

# grandstream.grand

---

```
    0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
    5
```

```
X2 = grand(q,4)
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
    9
```

Use reset to reset the stream, then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
    1

X = grand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

## See Also

[grandstream](#) | [reset](#)

**Purpose** Quasi-random point sets

**Description** grandset is a base class that encapsulates a sequence of multi-dimensional quasi-random numbers. This base class is abstract and cannot be instantiated directly. Concrete subclasses include sobolset and haltonset.

**Construction** grandset Abstract quasi-random point set class

**Methods**

disp	Display grandset object
end	Last index in indexing expression for point set
length	Length of point set
ndims	Number of dimensions in matrix
net	Generate quasi-random point set
scramble	Scramble quasi-random point set
size	Number of dimensions in matrix
subsref	Subscripted reference for grandset

**Properties**

Dimensions	Number of dimensions
Leap	Interval between points
ScrambleMethod	Settings that control scrambling

# grandset

---

Skip	Number of initial points to omit from sequence
Type	Name of sequence on which point set P is based

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## See Also

[haltonset](#) | [sobolset](#)

## How To

- “Quasi-Random Point Sets” on page 6-16

**Purpose** Abstract quasi-random point set class

**Description** `grandset` is an abstract class, and you cannot create instances of it directly. You must use `haltonset` or `sobolset` to create a `grandset` object.

**See Also** `haltonset` | `sobolset`

# grandstream

---

<b>Purpose</b>	Quasi-random number streams	
<b>Construction</b>	grandstream	Construct quasi-random number stream
<b>Methods</b>	addlistener	Add listener for event
	delete	Delete handle object
	disp	Display grandstream object
	eq	Test handle equality
	findobj	Find objects matching specified conditions
	findprop	Find property of MATLAB handle object
	ge	Greater than or equal relation for handles
	gt	Greater than relation for handles
	invalid	Test handle validity
	le	Less than or equal relation for handles
	lt	Less than relation for handles
	ne	Not equal relation for handles
	notify	Notify listeners of event
	grand	Generate quasi-random points from stream
	rand	Generate quasi-random points from stream
	reset	Reset state

## Properties

PointSet

Point set from which stream is drawn

State

Current state of the stream

## Copy Semantics

Handle. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

# grandstream

---

**Purpose** Construct quasi-random number stream

**Syntax**

```
q = grandstream(type,d)
q = grandstream(type,d,prop1,val1,prop2,val2,...)
q = grandstream(p)
```

**Description**

`q = grandstream(type,d)` constructs a *d*-dimensional quasi-random number stream *q* of the `grandstream` class, of type specified by the string *type*. *type* is either 'halton' or 'sobol', and *q* is based on a point set from either the `haltonset` class or `sobolset` class, respectively, with default property settings.

`q = grandstream(type,d,prop1,val1,prop2,val2,...)` specifies property name/value pairs for the point set on which the stream is based. Applicable properties depend on *type*.

`q = grandstream(p)` constructs a stream based on the specified point set *p*. *p* must be a point set from either the `haltonset` class or `sobolset` class.

**Examples**

Construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = grandstream('halton',3,'Skip',1e3,'Leap',1e2)
q =
  Halton quasi-random stream in 3 dimensions
  Point set properties:
      Skip : 1000
      Leap : 100
      ScrambleMethod : none

nextIdx = q.State
nextIdx =
     1
```

Use `grand` to generate two samples of size four:

```
X1 = grand(q,4)
```



```
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
nextIdx = q.State
nextIdx =
     5

X2 = grand(q,4)
X2 =
    0.2446    0.0238    0.8102
    0.5298    0.7540    0.0438
    0.3843    0.5112    0.2758
    0.8335    0.2245    0.4694
nextIdx = q.State
nextIdx =
     9
```

Use `reset` to reset the stream, and then generate another sample:

```
reset(q)
nextIdx = q.State
nextIdx =
     1

X = grand(q,4)
X =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166
```

## See Also

`haltonset` | `grand` | `reset` | `sobolset`

# qqplot

---

**Purpose** Quantile-quantile plot

**Syntax**

```
qqplot(X)
qqplot(X,Y)
qqplot(X,PD)
qqplot(X,Y,pvec)
h = qqplot(X,Y,pvec)
```

**Description** `qqplot(X)` displays a quantile-quantile plot of the sample quantiles of  $X$  versus theoretical quantiles from a normal distribution. If the distribution of  $X$  is normal, the plot will be close to linear.

`qqplot(X,Y)` displays a quantile-quantile plot of two samples. If the samples do come from the same distribution, the plot will be linear.

`qqplot(X,PD)` makes an empirical quantile-quantile plot of the quantiles of the data in the vector  $X$  versus the quantiles of the distribution specified by  $PD$ , a `ProbDist` object of the `ProbDistUnivParam` class or `ProbDistUnivKernel` class.

For matrix  $X$  and  $Y$ , `qqplot` displays a separate line for each pair of columns. The plotted quantiles are the quantiles of the smaller data set.

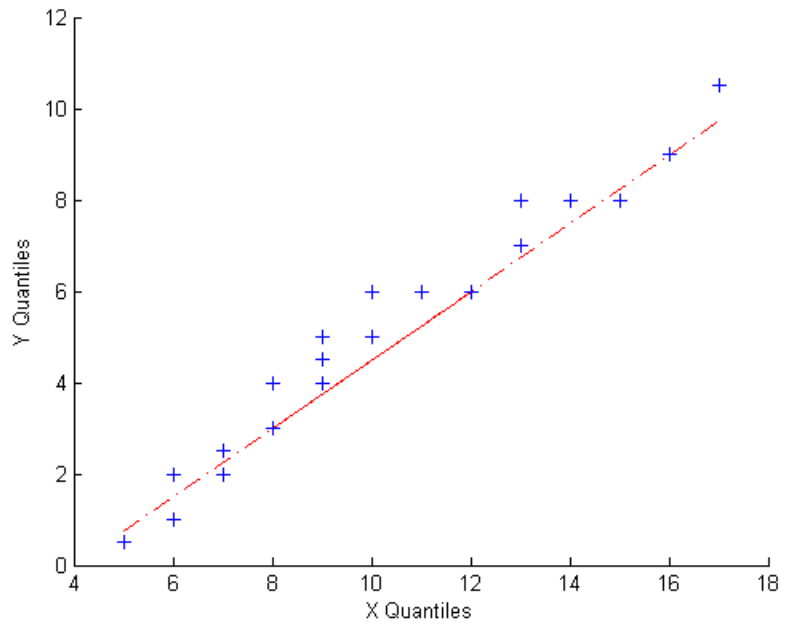
The plot has the sample data displayed with the plot symbol '+' . Superimposed on the plot is a line joining the first and third quartiles of each distribution (this is a robust linear fit of the order statistics of the two samples). This line is extrapolated out to the ends of the sample to help evaluate the linearity of the data.

Use `qqplot(X,Y,pvec)` to specify the quantiles in the vector `pvec`.

`h = qqplot(X,Y,pvec)` returns handles to the lines in `h`.

**Examples** The following example shows a quantile-quantile plot of two samples from Poisson distributions. The solid line joins the first and third quartiles. The dashed line extrapolates the solid line.

```
x = poissrnd(10,50,1);
y = poissrnd(5,100,1);
qqplot(x,y);
```



**How To**

- normplot

# quantile

---

## Purpose

Quantiles of a data set

## Syntax

`Y = quantile(X,p)`  
`Y = quantile(X,p,dim)`

`Y = quantile(X,N)`  
`Y = quantile(X,N,dim)`

## Description

`Y = quantile(X,p)` returns quantiles of the values in data vector or matrix `X` for the cumulative probability or probabilities `p` in the interval `[0,1]`.

- If `X` is a vector, then `Y` is a scalar or a vector having the same length as `p`.
- If `X` is a matrix, then `Y` is a row vector or a matrix where the number of rows of `Y` is equal to the length of `p`.
- For multidimensional arrays, `quantile` operates along the first nonsingleton dimension of `X`.

`Y = quantile(X,p,dim)` returns quantiles along dimension `dim`.

`Y = quantile(X,N)` returns quantiles for `N` evenly spaced cumulative probabilities  $(1/(N+1), 2/(N+1), \dots, N/(N+1))$  for integer  $N > 1$ .

- If `X` is a vector, then `Y` is a scalar or a vector with length `N`.
- If `X` is a matrix, then `Y` is a matrix where the number of rows of `Y` is equal to `N`.
- For multidimensional arrays, `quantile` operates along the first nonsingleton dimension of `X`.

`Y = quantile(X,N,dim)` returns quantiles at the `N` evenly-spaced cumulative probabilities  $(1/(N+1), 2/(N+1), \dots, N/(N+1))$  for integer  $N > 1$  along dimension `dim`.

## Input Arguments

### **X - Input data**

vector | array

Input data, specified as a vector or array.

### **Data Types**

double | single

### **p - Cumulative probabilities**

scalar | vector

Cumulative probabilities, for which to compute the quantiles, specified as a scalar or vector of scalars from 0 to 1.

**Example:** 0.3

**Example:** [0.25, 0.5, 0.75]

**Example:** (0:0.25:1)

### **Data Types**

double | single

### **N - Number of quantiles**

positive integer

Number of quantiles to compute, specified as a positive integer. `quantile` returns N quantiles that divide the data set into evenly distributed N+1 segments.

### **Data Types**

double | single

### **dim - Dimension**

1 (default) | positive integer

Dimension along which the quantiles of a matrix X are required, specified as a positive integer. For example, for a matrix X, when `dim = 1`, `quantile` returns the quantile(s) of the columns of X and when `dim = 2`, `quantile` returns the quantile(s) of the rows of X. For a multidimensional array X, the length of the `dim`th dimension of Y is same as length of p.

# quantile

---

## Output Arguments

### Y - Quantiles

scalar | array

Quantiles of a data vector or matrix, returned as a scalar or array for one or multiple values of cumulative probabilities.

- If  $X$  is a vector, then  $Y$  is a scalar or a vector with the same length as the number of quantiles required ( $N$  or  $\text{length}(p)$ ).  $Y(i)$  contains the  $p(i)$  quantile.
- If  $X$  is a matrix, then  $Y$  is a vector or a matrix with the length of  $\text{dimth}$  dimension equal to the number of quantiles required ( $N$  or  $\text{length}(p)$ ). When  $\text{dim} = 1$ , for example, the  $i$ th row of  $Y$  contains the  $p(i)$  quantiles of columns of  $X$ .
- If  $X$  is an array of dimension  $d$ , then  $Y$  is an array with the length of  $\text{dimth}$  dimension equal to the number of quantiles required ( $N$  or  $\text{length}(p)$ ).

## Definitions

### Multidimensional Array

A *multidimensional array* is an array with more than two dimensions. For example, if  $X$  is a 1-by-3-by-4 array, then  $X$  is a 3-D array.

### First Nonsingleton Dimension

A *first nonsingleton dimension* is the first dimension of an array whose size is not equal to 1. For example, if  $X$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $X$ .

### Linear Interpolation

Linear interpolation uses linear polynomials to find  $y_i = f(x_i)$ , the values of the underlying function  $Y = f(X)$  at the points in the vector or array  $x$ . Given the data points  $(x_1, y_1)$  and  $(x_2, y_2)$ , where  $y_1 = f(x_1)$  and  $y_2 = f(x_2)$ , linear interpolation finds  $y = f(x)$  for a given  $x$  between  $x_1$  and  $x_2$  as follows:

$$y = f(x) = y_1 + \frac{(x - x_1)}{(x_2 - x_1)}(y_2 - y_1).$$

Similarly, if the  $1.5/n$  quantile is  $y_{1.5/n}$  and the  $2.5/n$  quantile is  $y_{2.5/n}$ , then linear interpolation finds the  $2.3/n$  quantile  $y_{2.3/n}$  as

$$y_{\frac{2.3}{n}} = y_{\frac{1.5}{n}} + \frac{\left(\frac{2.3}{n} - \frac{1.5}{n}\right)}{\left(\frac{2.5}{n} - \frac{1.5}{n}\right)} \left(y_{\frac{2.5}{n}} - y_{\frac{1.5}{n}}\right).$$

## Algorithms

For an  $n$ -element vector  $X$ , `quantile` computes quantiles as follows:

- 1** The sorted values in  $X$  are taken as the  $(0.5/n)$ ,  $(1.5/n)$ , ...,  $([n - 0.5]/n)$  quantiles. For example:
  - For a data vector of five elements such as  $\{6, 3, 2, 10, 1\}$ , the sorted elements  $\{1, 2, 3, 6, 10\}$  respectively correspond to the 0.1, 0.3, 0.5, 0.7, 0.9 quantiles.
  - For a data vector of six elements such as  $\{6, 3, 2, 10, 8, 1\}$ , the sorted elements  $\{1, 2, 3, 6, 8, 10\}$  respectively correspond to the  $(0.5/6)$ ,  $(1.5/6)$ ,  $(2.5/6)$ ,  $(3.5/6)$ ,  $(4.5/6)$ ,  $(5.5/6)$  quantiles.
- 2** `quantile` uses Linear interpolation to compute quantiles for probabilities between  $(0.5/n)$  and  $([n - 0.5]/n)$ .
- 3** For the quantiles corresponding to the probabilities outside that range, `quantile` assigns the minimum or maximum values in  $X$ .

`quantile` treats NaNs as missing values and removes them.

## Examples

### Quantiles for Given Probabilities

Calculate the quantiles of a data set for specified probabilities.

Generate a data set of size 10.

```
rng('default'); % for reproducibility
x = normrnd(0,1,1,10)
```

# quantile

---

```
x =  
    0.5377    1.8339   -2.2588    0.8622    0.3188   -1.3077   -0.4336
```

Calculate the 0.3 quantile.

```
y = quantile(x,0.30)
```

```
y =  
   -0.0574
```

Calculate the quantiles for the cumulative probabilities 0.025, 0.25, 0.5, 0.75, and 0.975.

```
y = quantile(x,[0.025 0.25 0.50 0.75 0.975])
```

```
y =  
   -2.2588   -0.4336    0.4401    1.8339    3.5784
```

## Quantiles of a Matrix for Given Probabilities

Calculate the quantiles along the columns and rows of a data matrix for specified probabilities.

Generate a 4-by-6 data matrix.

```
rng('default'); % for reproducibility  
X = normrnd(0,1,4,6)
```

```
X =  
    0.5377    0.3188    3.5784    0.7254   -0.1241    0.6715  
    1.8339   -1.3077    2.7694   -0.0631    1.4897   -1.2075  
   -2.2588   -0.4336   -1.3499    0.7147    1.4090    0.7172  
    0.8622    0.3426    3.0349   -0.2050    1.4172    1.6302
```

Calculate the 0.3 quantile for each column of X (dim = 1).

```
y = quantile(x,0.3,1)
```



```
y =
    -0.3013    -0.6958     1.5336    -0.1056     0.9491     0.1078
```

`quantile` returns a row vector `y` when calculating one quantile for each column of a matrix. For example, `-0.3013` is the 0.3 quantile of the first column of `X` with elements `(0.5377, 1.8339, -2.2588, 0.8622)`. `y = quantile(X,0.3)` returns the same answer because the default value of `dim` is 1.

Calculate the 0.3 quantile for each row of `X` (`dim = 2`).

```
y = quantile(x,0.3,2)
```

```
y =
    0.3844
   -0.8642
   -1.0750
    0.4985
```

`quantile` returns a column vector `y` when calculating one quantile for each row of a matrix. For example `0.3844` is the 0.3 quantile of the first row of `X` with elements `(0.5377, 0.3188, 3.5784, 0.7254, -0.1241, 0.6715)`.

## Quantiles for *N* Evenly Spaced Cumulative Probabilities

Calculate the quantiles of a data set for a given number of quantiles.

Generate a data set of size 10.

```
rng('default'); % for reproducibility
x = normrnd(0,1,1,10)
```

```
x =
    0.5377     1.8339    -2.2588     0.8622     0.3188    -1.3077    -0.4330
```

Calculate four evenly spaced quantiles.

```
y = quantile(x,4)
```

# quantile

---

```
y =  
    -0.8706    0.3307    0.6999    2.3017
```

Using `y = quantile(x,[0.2,0.4,0.6,0.8])` is another way to return the four evenly spaced quantiles.

## Quantiles of a Matrix for Given Number of Quantiles

Calculate the  $N$  evenly spaced quantiles along the columns and rows of a data matrix.

Generate a 6-by-10 data matrix.

```
rng('default'); % for reproducibility  
X = unidrnd(10,6,7)
```

```
X =  
  
     9     3    10     8     7     8     7  
    10     6     5    10     8     1     4  
     2    10     9     7     8     3    10  
    10    10     2     1     4     1     1  
     7     2     5     9     7     1     5  
     1    10    10    10     2     9     4
```

Calculate three evenly spaced quantiles for each column of  $X$  (`dim = 1`).

```
y = quantile(X,3,1)
```

```
y =  
  
    2.0000    3.0000    5.0000    7.0000    4.0000    1.0000    4.0000  
    8.0000    8.0000    7.0000    8.5000    7.0000    2.0000    4.5000  
   10.0000   10.0000   10.0000   10.0000    8.0000    8.0000    7.0000
```

Each column of matrix  $y$  corresponds to the three evenly spaced quantiles of each column of matrix  $X$ . For example, the first column of  $y$  with elements (2, 8, 10) has the quantiles for the first column of  $X$

with elements (9, 10, 2, 10, 7, 1). `y = quantile(X,3)` returns the same answer because the default value of `dim` is 1.

Calculate three evenly spaced quantiles for each row of `X` (`dim = 2`).

```
y = quantile(X,3,2)
```

```
y =
```

```

7.0000    8.0000    8.7500
4.2500    6.0000    9.5000
4.0000    8.0000    9.7500
1.0000    2.0000    8.5000
2.7500    5.0000    7.0000
2.5000    9.0000   10.0000

```

Each row of matrix `y` corresponds to the three evenly spaced quantiles of each row of matrix `X`. For example, the first row of `y` with elements (7, 8, 8.75) has the quantiles for the first column of `X` with elements (9, 3, 10, 8, 7, 8, 7).

## Median and Quartiles for Even Number of Data Elements

Find median and quartiles of a vector, `x`, with even number of elements.

Enter the data.

```
x = [2 5 6 10 11 13]
```

```
x =
```

```

2      5      6      10     11     13

```

Calculate the median of `x`.

```
y = quantile(x,0.50)
```

```
y =
```

```

8

```

# quantile

---

Calculate the quartiles of  $x$ .

```
y = quantile(x,[0.25, 0.5, 0.75])
```

```
y =  
    5    8   11
```

Using `y = quantile(x,3)` is another way to compute the quartiles of  $x$ .

These results might be different than the textbook definitions because `quantile` uses linear interpolation to find the median and quartiles.

## **Median and Quartiles for Odd Number of Data Elements**

Find median and quartiles of a vector,  $x$ , with odd number of elements.

Enter the data.

```
x = [2 4 6 8 10 12 14]
```

```
x =  
    2    4    6    8   10   12   14
```

Find the median of  $x$ .

```
y = quantile(x,0.50)
```

```
y =  
    8
```

Find the quartiles of  $x$ .

```
y = quantile(x,[0.25, 0.5, 0.75])
```

```
y =  
    4.5000    8.0000   11.5000
```

Using `y = quantile(x,3)` is another way to compute the quartiles of  $x$ .

These results might be different than the textbook definitions because `quantile` uses linear interpolation to find the median and quartiles.

**See Also**

`iqr` | `median` | `prctile`

**Concepts**

- “Quantiles and Percentiles” on page 3-7

# grandstream.rand

---

**Purpose** Generate quasi-random points from stream

**Syntax**

```
rand
rand(q,n)
rand(q)
rand(q,m,n)
rand(q,[m,n])
rand(q,m,n,p,...)
rand(q,[m,n,p,...])
```

**Description** rand returns a matrix of quasi-random values and is intended to allow objects of the grandstream class to be used in code that contains calls to the rand method of the MATLAB pseudo-random randstream class. Due to the multidimensional nature of quasi-random numbers, only some syntaxes of rand are supported by the grandstream class.

rand(q,n) returns an n-by-n matrix only when n is equal to the number of dimensions. Any other value of n produces an error.

rand(q) returns a scalar only when the stream is in one dimension. Having more than one dimension in q produces an error.

rand(q,m,n) or rand(q,[m,n]) returns an m-by-n matrix only when n is equal to the number of dimensions in the stream. Any other value of n produces an error.

rand(q,m,n,p,...) or rand(q,[m,n,p,...]) produces an error unless p and all following dimensions sizes are equal to one.

**Examples** Generate the first 256 points from a 5-D Sobol sequence:

```
q = grandstream('sobol',5);
X = rand(q,256,5);
```

**See Also** grandstream | grand | rand

**Purpose**

Gamma random numbers with unit scale

**Syntax**

```
Y = randg
Y = randg(A)
Y = randg(A,m)
Y = randg(A,m,n,p,...)
Y = randg(A,[m,n,p,...])
```

**Description**

`Y = randg` returns a scalar random value chosen from a gamma distribution with unit scale and shape.

`Y = randg(A)` returns a matrix of random values chosen from gamma distributions with unit scale. `Y` is the same size as `A`, and `randg` generates each element of `Y` using a shape parameter equal to the corresponding element of `A`.

`Y = randg(A,m)` returns an `m`-by-`m` matrix of random values chosen from gamma distributions with shape parameters `A`. `A` is either an `m`-by-`m` matrix or a scalar. If `A` is a scalar, `randg` uses that single shape parameter value to generate all elements of `Y`.

`Y = randg(A,m,n,p,...)` or `Y = randg(A,[m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array of random values chosen from gamma distributions with shape parameters `A`. `A` is either an `m`-by-`n`-by-`p`-by-... array or a scalar.

`randg` produces pseudo-random numbers using the MATLAB functions `rand` and `randn`. The sequence of numbers generated is determined by the settings of the uniform random number generator that underlies `rand` and `randn`. Control that shared random number generator using `rng`. See the `rng` documentation for more information.

---

**Note** To generate gamma random numbers and specify both the scale and shape parameters, you should call `gamrnd`.

---

## Examples

### Example 1

Generate a 100-by-1 array of values drawn from a gamma distribution with shape parameter 3.

```
r = randg(3,100,1);
```

### Example 2

Generate a 100-by-2 array of values drawn from gamma distributions with shape parameters 3 and 2.

```
A = [ones(100,1)*3,ones(100,1)*2];  
r = randg(A,[100,2]);
```

### Example 3

To create reproducible output from `randg`, reset the random number generator used by `rand` and `randn` to its default startup settings. This way `randg` produces the same random numbers as if you restarted MATLAB.

```
rng('default')  
randg(3,1,5)
```

```
ans =
```

```
6.9223    4.3369    1.0505    3.2662   11.3269
```

### Example 4

Save the settings for the random number generator used by `rand` and `randn`, generate 5 values from `randg`, restore the settings, and repeat those values.

```
s = rng; % Obtain the current state of the random stream  
r1 = randg(10,1,5)
```

```
r1 =
```

```
9.4719    9.0433   15.0774   14.7763    6.3775
```



```
rng(s); % Reset the stream to the previous state
r2 = randg(10,1,5)

r2 =

    9.4719    9.0433   15.0774   14.7763    6.3775
```

r2 contains exactly the same values as r1.

### Example 5

Reinitialize the random number generator used by `rand` and `randn` with a seed based on the current time. `randg` returns different values each time you do this. Note that it is usually not necessary to do this more than once per MATLAB session.

```
rng('shuffle');
randg(2,1,5);
```

### References

[1] Marsaglia, G., and W. W. Tsang. "A Simple Method for Generating Gamma Variables." *ACM Transactions on Mathematical Software*. Vol. 26, 2000, pp. 363–372.

### See Also

`gamrnd`

# random

---

## Purpose

Random numbers

## Syntax

```
Y = random(name,A)
Y = random(name,A,B)
Y = random(name,A,B,C)
Y = random(name,A,m,n,...)
Y = random(name,A,[m,n,...])
Y = random(name,A,B,m,n,...)
Y = random(name,A,B,[m,n,...])
Y = random(name,A,B,C,m,n,...)
Y = random(name,A,B,C,[m,n,...])
```

## Description

`Y = random(name,A)` where `name` is the name of a distribution that takes a single parameter, returns random numbers `Y` from the one-parameter family of distributions specified by `name`. Parameter values for the distribution are given in `A`.

`Y` is the same size as `A`.

`Y = random(name,A,B)` returns random numbers `Y` from a two-parameter family of distributions. Parameter values for the distribution are given in `A` and `B`.

If `A` and `B` are arrays, they must be the same size. If either `A` or `B` are scalars, they are expanded to constant matrices of the same size.

`Y = random(name,A,B,C)` returns random numbers `Y` from a three-parameter family of distributions. Parameter values for the distribution are given in `A`, `B`, and `C`.

If `A`, `B`, and `C` are arrays, they must be the same size. If any of `A`, `B`, or `C` are scalars, they are expanded to constant matrices of the same size.

`Y = random(name,A,m,n,...)` or `Y = random(name,A,[m,n,...])` returns an `m`-by-`n`-by... matrix of random numbers.

Similarly, `Y = random(name,A,B,m,n,...)` or `Y = random(name,A,B,[m,n,...])` returns an `m`-by-`n`-by... matrix of random numbers for distributions that require two parameters. `Y = random(name,A,B,C,m,n,...)` or `Y =`

`random(name, A, B, C, [m, n, ...])` returns an m-by-n-by... matrix of random numbers for distributions that require three parameters.

If any of A, B, or C are arrays, then the specified dimensions must match the common dimensions of A, B, and C after any necessary scalar expansion.

The following table denotes the acceptable strings for name, as well as the parameters for that distribution:

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'beta' or 'Beta'	“Beta Distribution” on page B-5	a	b	—
'bino' or 'Binomial'	“Binomial Distribution” on page B-9	n: number of trials	p: probability of success for each trial	—
'birnbaumsaunders' or 'Birnbaum-Saunders'	“Birnbaum-Saunders Distribution” on page B-13	$\alpha$	$\gamma$	—
'burr' or 'Burr'	“Burr Type XII Distribution” on page B-15	$\alpha$ : scale parameter	c: shape parameter	k: shape parameter
'chi2' or 'Chisquare'	“Chi-Square Distribution” on page B-28	$\nu$ : degrees of freedom	—	—
'exp' or 'Exponential'	“Exponential Distribution” on page B-32	$\mu$ : mean	—	—
'ev' or 'Extreme Value'	“Extreme Value Distribution” on page B-35	$\mu$ : location parameter	$\sigma$ : scale parameter	—

# random

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'f' or 'F'	“F Distribution” on page B-41	v1: numerator degrees of freedom	v2: denominator degrees of freedom	—
'gam' or 'Gamma'	“Gamma Distribution” on page B-43	a: shape parameter	b: scale parameter	—
'gev' or 'Generalized Extreme Value'	“Generalized Extreme Value Distribution” on page B-48	k: shape parameter	$\sigma$ : scale parameter	$\mu$ : location parameter
'gp' or 'Generalized Pareto'	“Generalized Pareto Distribution” on page B-53	k: tail index (shape) parameter	$\sigma$ : scale parameter	$\mu$ : threshold (location) parameter
'geo' or 'Geometric'	“Geometric Distribution” on page B-57	p: probability parameter	—	—
'hyge' or 'Hypergeometric'	“Hypergeometric Distribution” on page B-60	M: size of the population	K: number of items with the desired characteristic in the population	n: number of samples drawn
'inversegaussian'	“Inverse Gaussian Distribution” on page B-63	$\mu$	$\lambda$	—
'logistic'	“Logistic Distribution” on page B-76	$\mu$	$\sigma$	—

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'loglogistic'	“Loglogistic Distribution” on page B-78	$\mu$	$\sigma$	—
'logn' or 'Lognormal'	“Lognormal Distribution” on page B-80	$\mu$	$\sigma$	—
'nakagami'	“Nakagami Distribution” on page B-97	$\mu$	$\omega$	—
'nbin' or 'Negative Binomial'	“Negative Binomial Distribution” on page B-98	r: number of successes	p: probability of success in a single trial	—
'ncf' or 'Noncentral F'	“Noncentral F Distribution” on page B-106	$\nu_1$ : numerator degrees of freedom	$\nu_2$ : denominator degrees of freedom	$\delta$ : noncentrality parameter
'nct' or 'Noncentral t'	“Noncentral t Distribution” on page B-108	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—
'ncx2' or 'Noncentral Chi-square'	“Noncentral Chi-Square Distribution” on page B-104	$\nu$ : degrees of freedom	$\delta$ : noncentrality parameter	—
'norm' or 'Normal'	“Normal Distribution” on page B-110	$\mu$ : mean	$\sigma$ : standard deviation	—
'poiss' or 'Poisson'	“Poisson Distribution” on page B-118	$\lambda$ : mean	—	—

# random

name	Distribution	Input Parameter A	Input Parameter B	Input Parameter C
'rayl' or 'Rayleigh'	“Rayleigh Distribution” on page B-121	b: scale parameter	—	—
'rician'	“Rician Distribution” on page B-123	s: noncentrality parameter	$\sigma$ : scale parameter	—
't' or 'T'	“Student’s t Distribution” on page B-124	$\nu$ : degrees of freedom	—	—
'tlocationscale'	“t Location-Scale Distribution” on page B-126	$\mu$ : location parameter	$\sigma$ : scale parameter	$\nu$ : shape parameter
'unif' or 'Uniform'	“Uniform Distribution (Continuous)” on page B-133	a: lower endpoint (minimum)	b: upper endpoint (maximum)	—
'unid' or 'Discrete Uniform'	“Uniform Distribution (Discrete)” on page B-139	N: maximum observable value	—	—
'wbl' or 'Weibull'	“Weibull Distribution” on page B-141	a: scale parameter	b: shape parameter	—

## Examples

Generate a 2-by-4 array of random values from the normal distribution with mean 0 and standard deviation 1:

```
x1 = random('Normal',0,1,2,4)
x1 =
```

```
1.1650  0.0751  -0.6965  0.0591
0.6268  0.3516  1.6961  1.7971
```

The order of the parameters is the same as for `normrnd`.

Generate a single random value from Poisson distributions with rate parameters 1, 2, ..., 6, respectively:

```
x2 = random('Poisson',1:6,1,6)
x2 =
    0    0    1    2    5    7
```

## See Also

[cdf](#) | [pdf](#) | [icdf](#) | [mle](#)

# GeneralizedLinearModel.random

---

**Purpose** Simulate responses for generalized linear regression model

**Syntax**  
`ysim = random mdl, Xnew`  
`ysim = random mdl, Xnew, Name, Value`

**Description**  
`ysim = random mdl, Xnew` simulates responses from the `mdl` generalized linear model to the data in `Xnew`.  
`ysim = random mdl, Xnew, Name, Value` simulates responses with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

### **Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### **'BinomialSize'**

The value of the binomial  $n$  parameter for each row in the training data. `BinomialSize` can be a vector the same length as `Xnew`, or



a scalar that applies to each row. The default value 1 produces `ysim` values that are predicted proportions. Use `BinomialSize` only if `mdl` is fit to a binomial distribution.

**Default:** 1

## 'Offset'

Value of the offset for each row in `Xnew`. `Offset` can be a vector the same length as `Xnew`, or a scalar that applies to each row. The offset is used as an additional predictor with a coefficient value fixed at 1. In other words, if `b` is the fitted coefficient vector, and `link` is the link function,

$$\text{link}(\text{ysim}) = \text{Offset} + \text{Xnew} * \text{b}.$$

**Default:** `zeros(size(Xnew,1))`

## Output Arguments

### `ysim`

Vector of simulated values at `Xnew`.

`random` generates `ysim` using random values with mean given by the fitted model, and with the distribution used in `mdl`. The values in `ysim` are independent conditional on the predictors. For binomial and Poisson fits, `random` generates `ysim` with the specified distribution with no adjustment for any estimated dispersion.

## Examples

### Generalized Linear Model Simulation

Create a generalized linear model, and simulate its response to new data.

Generate artificial data for the model, Poisson random numbers with one underlying predictors `X`.

```
rng('default') % reproducible
X = rand(20,1);
```

# GeneralizedLinearModel.random

---

```
mu = exp(1 + 2*X);  
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1','distr','poisson');
```

Create points for prediction.

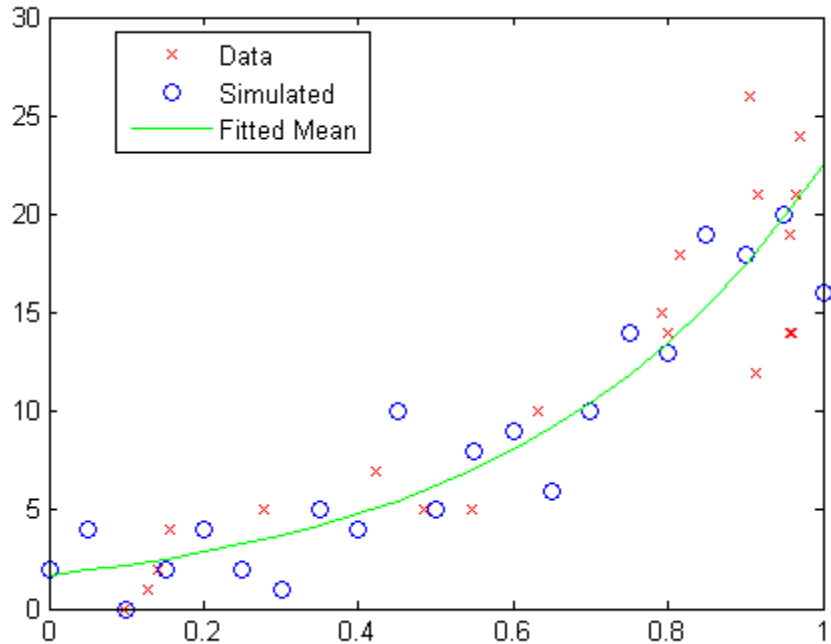
```
Xnew = (0:.05:1)';
```

Simulate responses at the new points.

```
ysim = random(mdl,Xnew);
```

Plot the simulated values along with the original values.

```
plot(X,y,'rx',Xnew,ysim,'bo',...  
      Xnew,feval(mdl,Xnew),'g-')  
legend('Data','Simulated','Fitted Mean',...  
       'Location','best')
```



## Alternatives

For predictions without random noise, use `predict` or `feval`.

## See Also

`GeneralizedLinearModel` | `predict` |

## Related Examples

- “random” on page 10-39

## Concepts

- “Generalized Linear Models” on page 10-12

# gmdistribution.random

---

**Purpose** Random numbers from Gaussian mixture distribution

**Syntax**

```
y = random(obj)
Y = random(obj,n)
[Y,idx] = random(obj,n)
```

**Description** `y = random(obj)` generates a 1-by- $d$  vector  $y$  drawn at random from the  $d$ -dimensional Gaussian mixture distribution defined by `obj`. `obj` is an object created by `gmdistribution` or `fit`.

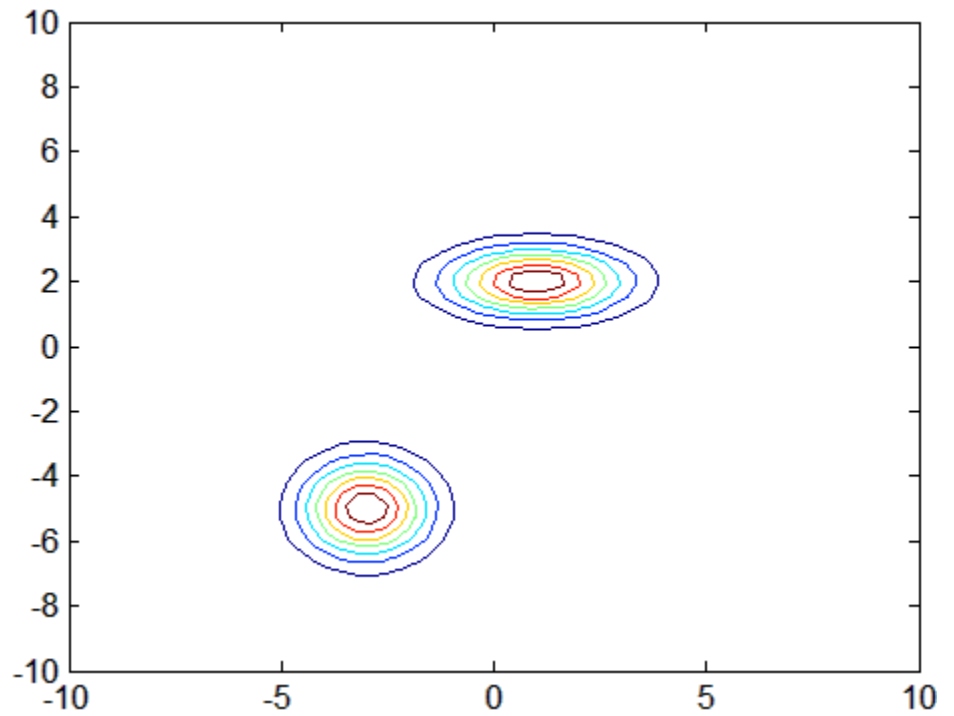
`Y = random(obj,n)` generates an  $n$ -by- $d$  matrix  $Y$  of  $n$   $d$ -dimensional random samples.

`[Y,idx] = random(obj,n)` also returns an  $n$ -by-1 vector `idx`, where `idx(I)` is the index of the component used to generate  $Y(I,:)$ .

**Examples** Create a `gmdistribution` object defining a two-component mixture of bivariate Gaussian distributions:

```
MU = [1 2; -3 -5];
SIGMA = cat(3,[2 0;0 .5],[1 0;0 1]);
p = ones(1,2)/2;
obj = gmdistribution(MU,SIGMA,p);

ezcontour(@(x,y)pdf(obj,[x y]),[-10 10],[-10 10])
hold on
```



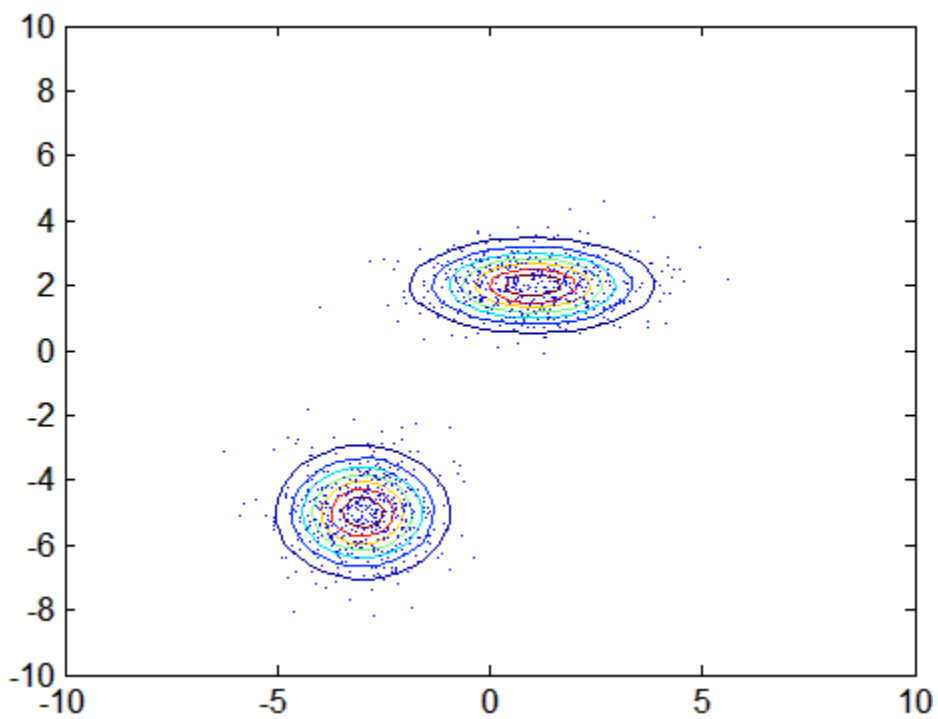
Generate 1000 random values:

```
Y = random(obj,1000);
```

```
scatter(Y(:,1),Y(:,2),10,'.')
```

# gmdistribution.random

---



## See Also

[gmdistribution](#) | [fit](#) | [mvnrnd](#)

**Purpose**

Simulate responses for linear regression model

**Syntax**

```
ysim = random mdl  
ysim = random mdl, Xnew
```

**Description**

`ysim = random(mdl)` simulates responses from the fitted linear model `mdl` at the original design points.

`ysim = random(mdl, Xnew)` simulates responses from the `mdl` linear model to the data in `Xnew`, adding random noise.

**Input Arguments****mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

**Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

**Output Arguments****ysim**

Vector of predicted mean values at `Xnew`, perturbed by random noise. The noise is independent, normally distributed, with mean zero, and variance equal to the estimated error variance of the model.

**Examples****Simulate Response Data**

Create a model of car mileage as a function of weight, and simulate the response.

Create a quadratic model of car mileage as a function of weight from the `carsmall` data.

# LinearModel.random

---

```
load carsmall
X = Weight;
y = MPG;
mdl = fitlm(X,y,'quadratic');
```

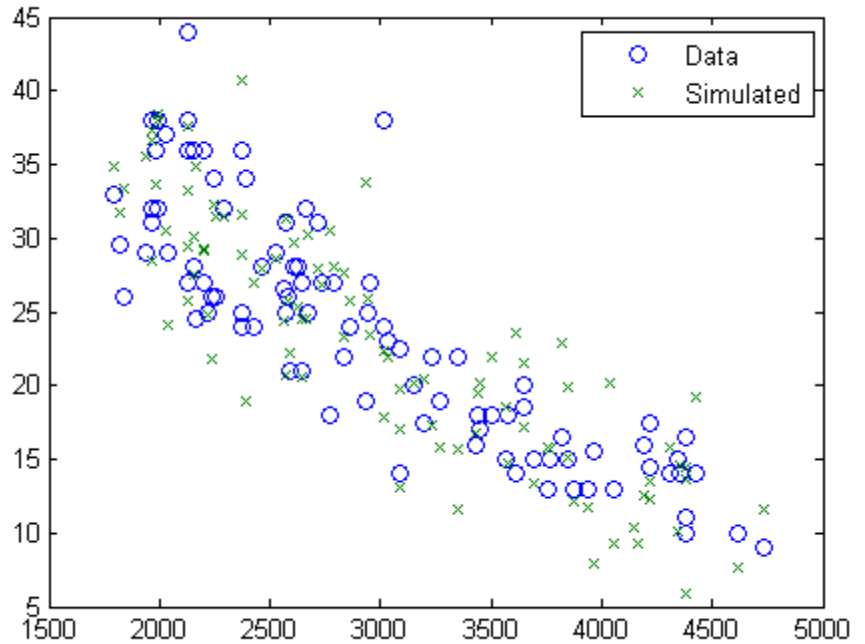
Create simulated responses to the data.

```
Xnew = X;
ysim = random(mdl,Xnew);
```

Plot the original responses and the simulated responses to see how they differ.

```
plot(X,y,'o',X,ysim,'x')
legend('Data','Simulated')
```





**Alternatives** For predictions without random noise, use `predict` or `feval`.

**See Also** `feval` | `LinearModel` | `predict`

**Tutorials** • “random” on page 9-42

**How To** • “Linear Regression” on page 9-12

# LinearMixedModel.random

---

**Purpose** Generate random responses from fitted linear mixed-effects model

**Syntax**

```
ysim = random(lme)
ysim = random(lme, dsnew)
ysim = random(lme, Xnew, Znew)
ysim = random(lme, Xnew, Znew, Gnew)
```

**Description** `ysim = random(lme)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the original fixed- and random-effects design points, used to fit `lme`.

`random` simulates new random-effects vector and new observation errors. So, the simulated response is

$$y_{sim} = X\hat{\beta} + Z\hat{b} + \varepsilon,$$

where  $\hat{\beta}$  is the estimated fixed-effects coefficients,  $\hat{b}$  is the new random effects, and  $\varepsilon$  is the new observation error.

`random` also accounts for the effect of observation weights, if you use any when fitting the model.

`ysim = random(lme, dsnew)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the values in the new dataset array `dsnew`. Use a dataset array for `random` if you use a dataset array for fitting the model `lme`.

`ysim = random(lme, Xnew, Znew)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the values in the new fixed- and random-effects design matrices, `Xnew` and `Znew`, respectively. `Znew` can also be a cell array of matrices. Use the matrix format for `random` if you use design matrices for fitting the model `lme`.

`ysim = random(lme, Xnew, Znew, Gnew)` returns a vector of simulated responses `ysim` from the fitted linear mixed-effects model `lme` at the values in the new fixed- and random-effects design matrices, `Xnew` and `Znew`, respectively, and the grouping variable `Gnew`.

`Znew` and `Gnew` can also be cell arrays of matrices and grouping variables, respectively.

## Input Arguments

### **lme** - Linear mixed-effects model

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **dsnew** - New input data

dataset array

New input data, which includes the response variable, predictor variables, and grouping variables, specified as a dataset array. The predictor variables can be continuous or grouping variables. `dsnew` must have the same variables as in the original dataset array used to fit the linear mixed-effects model `lme`.

### Data Types

`single` | `double` | `logical` | `char`

### **Xnew** - New fixed-effects design matrix

$n$ -by- $p$  matrix

New fixed-effects design matrix, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of fixed predictor variables. Each row of `X` corresponds to one observation and each column of `X` corresponds to one variable.

### Data Types

`single` | `double`

### **Znew** - New random-effects design

$n$ -by- $q$  matrix | cell array of length  $R$

# LinearMixedModel.random

---

New random-effects design, specified as an  $n$ -by- $q$  matrix or a cell array of  $R$  design matrices  $Z\{r\}$ , where  $r = 1, 2, \dots, R$ . If  $Z_{\text{new}}$  is a cell array, then each  $Z\{r\}$  is an  $n$ -by- $q(r)$  matrix, where  $n$  is the number of observations, and  $q(r)$  is the number of random predictor variables.

## Data Types

single | double | logical | char | cell

## **Gnew** - New grouping variable or variables

vector | cell array of grouping variables of length  $R$

New grouping variable or variables, specified as a vector or a cell array, of length  $R$ , of grouping variables used to fit the linear mixed-effects model, `lme`.

random treats all levels of each grouping variable as new levels. It draws an independent random effects vector for each level of each grouping variable.

## Data Types

single | double | logical | char | cell

## Output Arguments

### **ysim** - Simulated response values

$n$ -by-1 vector

Simulated response values, returned as an  $n$ -by-1 vector, where  $n$  is the number of observations.

## Examples

### **Generate Random Responses at the Original Design Values**

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Generate random response values at the original design points. Display the first five values.

```
rng(123, 'twister') % For reproducibility  
ysim = random(lme);  
ysim(1:5)
```

```
ans =
```

```
114.8785  
134.2018  
154.2818  
169.7554  
84.6089
```

# LinearMixedModel.random

---

## Plot Randomly Generated vs. Observed Response Values

Load the sample data.

```
load carsmall
```

Fit a linear mixed-effects model, with a fixed-effects for Weight, and a random intercept grouped by Model\_Year. First, store the data in a dataset array.

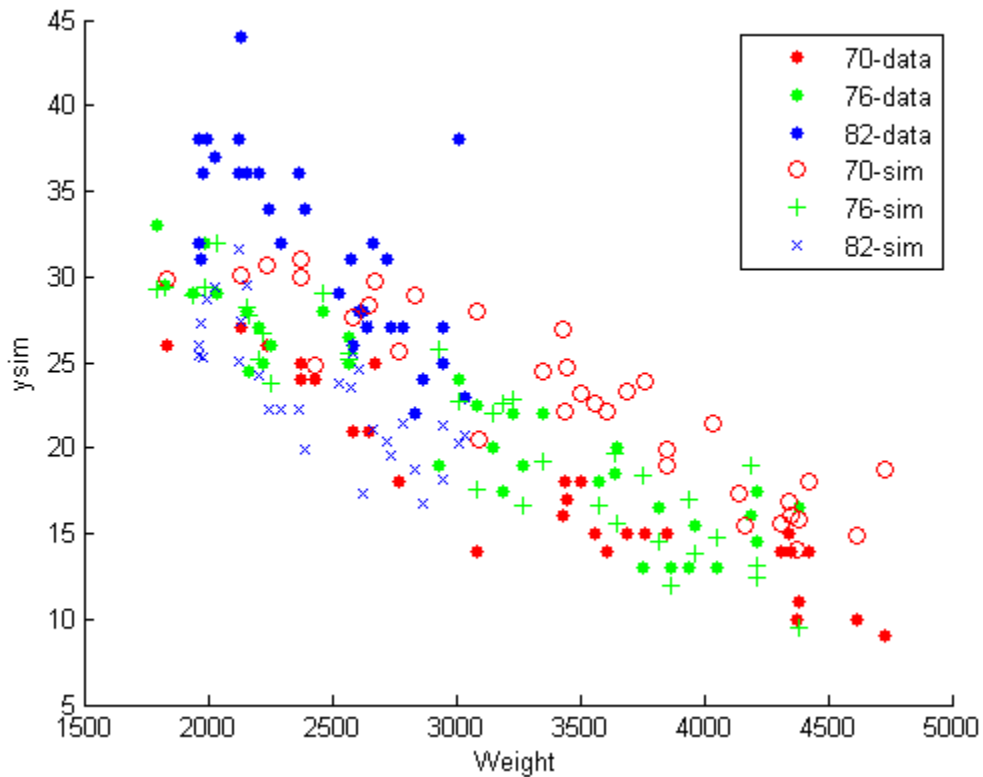
```
ds = dataset(MPG,Weight,Model_Year);  
lme = fitlme(ds,'MPG ~ Weight + (1|Model_Year)');
```

Randomly generate responses using the original data.

```
rng(123,'twister') % For reproducibility  
ysim = random(lme,ds);
```

Plot the original and the randomly generated responses to see how they differ. Group them by model year.

```
figure()  
gscatter(Weight,MPG,Model_Year)  
hold on  
gscatter(Weight,ysim,Model_Year,[],'o+x')  
legend('70-data','76-data','82-data','70-sim','76-sim','82-sim')  
hold off
```



Note that the simulated random response values for year 82 are lower than the original data for that year. This might be due to a lower simulated random effect for year 82 than the estimated random effect in the original data.

## Generate Responses Using a New Dataset Array

Navigate to a folder containing sample data.

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

# LinearMixedModel.random

---

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)')
```

Create a new dataset array with design values. The new dataset array must have the same variables as the original dataset array you use for fitting the model `lme`.

```
dsnew = dataset();  
dsnew.Soil = nominal({'Sandy'; 'Silty'; 'Silty'});  
dsnew.Tomato = nominal({'Cherry'; 'Vine'; 'Plum'});  
dsnew.Fertilizer = nominal([2;2;4]);
```

Generate random responses at the new points.

```
rng(123, 'twister') % For reproducibility
```



```
ysim = random(lme,dsnew)
```

```
ysim =  
    99.6006  
    101.9911  
    161.4026
```

## Generate Random Responses Using New Design Matrices

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and cylinders, and potentially correlated random effect for intercept and acceleration grouped by model year.

First, prepare the design matrices for fitting the linear mixed-effects model.

```
X = [ones(406,1) Acceleration Horsepower];  
Z = [ones(406,1) Acceleration];  
Model_Year = nominal(Model_Year);  
G = Model_Year;
```

Now, fit the model using `fitlmematrix` with the defined design matrices and grouping variables.

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...  
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...  
{{'Intercept','Acceleration'}},'RandomEffectGroups',{'Model_Year'});
```

Create the design matrices that contain the data at which to predict the response values. `Xnew` must have three columns as in `X`. The first column must be a column of 1s. And the values in the last two columns must correspond to `Acceleration` and `Horsepower`, respectively. The first column of `Znew` must be a column of 1s, and the second column

# LinearMixedModel.random

---

must contain the same Acceleration values as in Xnew. The original grouping variable in G is the model year. So, Gnew must contain values for the model year. Note that Gnew must contain nominal values.

```
Xnew = [1,13.5,185; 1,17,205; 1,21.2,193];  
Znew = [1,13.5; 1,17; 1,21.2];  
Gnew = nominal([73 77 82]);
```

Generate random responses for the data in the new design matrices.

```
rng(123,'twister') % For reproducibility  
ysim = random(lme,Xnew,Znew,Gnew)
```

```
ysim =
```

```
15.7416  
10.6085  
6.8796
```

Now, repeat the same for a linear mixed-effects model with uncorrelated random-effects terms for intercept and acceleration. First, change the original random effects design and the random effects grouping variables. Then, fit the model.

```
Z = {ones(406,1),Acceleration};  
G = {Model_Year,Model_Year};
```

```
lme = fitlmematrix(X,MPG,Z,G,'FixedEffectPredictors',...  
{'Intercept','Acceleration','Horsepower'},'RandomEffectPredictors',...  
{{'Intercept'},{'Acceleration'}},'RandomEffectGroups',{'Model_Year','Model_Year'});
```

Now, recreate the new random effects design, Znew, and the grouping variable design, Gnew, using which to predict the response values.

```
Znew = {[1;1;1],[13.5;17;21.2]};  
MY = nominal([73 77 82]);  
Gnew = {MY,MY};
```

Generate random responses using the new design matrices.

```
rng(123,'twister') % For reproducibility  
ysim = random(lme,Xnew,Znew,Gnew)
```

```
ysim =  
  
    16.8280  
    10.4375  
     4.1027
```

## See Also

[predict](#) | [fitlme](#) | [fitlmematrixLinearMixedModel](#) |

# NonLinearModel.random

---

**Purpose** Simulate responses for nonlinear regression model

**Syntax**

```
ysim = random mdl
ysim = random mdl,Xnew
ysim = random mdl,Xnew,'Weights',W
```

**Description**

`ysim = random(mdl)` simulates responses from the fitted nonlinear model `mdl` at the original design points.

`ysim = random(mdl,Xnew)` simulates responses from the fitted nonlinear model `mdl` to the data in `Xnew`, adding random noise.

`ysim = random(mdl,Xnew,'Weights',W)` simulates responses using the observation weights, `W`.

## Input Arguments

### **mdl**

Nonlinear regression model, constructed by `fitnlm`.

### **Xnew**

Points at which `mdl` predicts responses.

- If `Xnew` is a dataset array, it must contain the predictor names in `mdl`.
- If `Xnew` is a numeric matrix, it must have the same number of variables (columns) as was used to create `mdl`. Furthermore, all variables used in creating `mdl` must be numeric.

### **W**

Vector of real, positive value weights or a function handle.

- If you specify a vector, then it must have the same number of elements as the number of observations (or rows) in `Xnew`.
- If you specify a function handle, the function must accept a vector of predicted response values as input, and returns a vector of real positive weights as output.

Given weights,  $W$ , random estimates the error variance at observation  $i$  by  $MSE * (1/W(i))$ , where  $MSE$  is the mean squared error.

**Default:** No weights

## Output Arguments

### **ysim**

Vector of predicted mean values at  $X_{new}$ , perturbed by random noise. The noise is independent, normally distributed, with mean zero, and variance equal to the estimated error variance of the model.

## Examples

### **Simulate Responses**

Create a nonlinear model of car mileage as a function of weight, and simulate the response.

Create an exponential model of car mileage as a function of weight from the `carsmall` data. Scale the weight by a factor of 1000 so all the variables are roughly equal in size.

```
load carsmall
X = Weight;
y = MPG;
modelfun = 'y ~ b1 + b2*exp(-b3*x/1000)';
beta0 = [1 1 1];
mdl = fitnlm(X,y,modelfun,beta0);
```

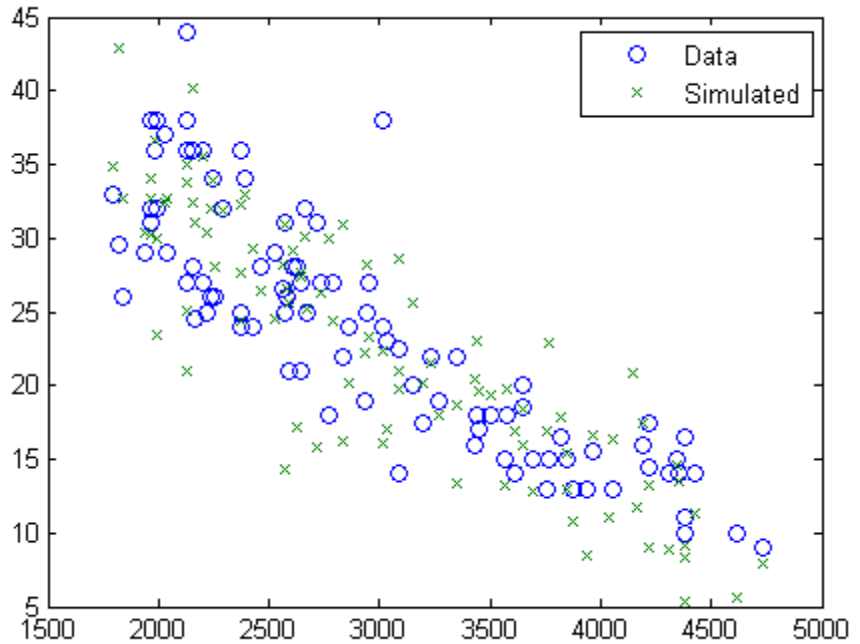
Create simulated responses to the data.

```
Xnew = X;
ysim = random(mdl,Xnew);
```

Plot the original responses and the simulated responses to see how they differ.

```
plot(X,y,'o',X,ysim,'x')
legend('Data','Simulated')
```

# NonLinearModel.random



## Alternatives

For predictions without added noise, use `predict`.

## See Also

`feval` | `NonLinearModel` | `predict` |

## Related Examples

- “Predict or Simulate Responses to New Data” on page 11-12

## Concepts

- “Nonlinear Regression” on page 11-2

<b>Purpose</b>	Random numbers from piecewise distribution
<b>Syntax</b>	<pre>r = random(obj) R = random(obj,n) R = random(obj,m,n) R = random(obj,[m,n]) R = random(obj,m,n,p,...) R = random(obj,[m,n,p,...])</pre>
<b>Description</b>	<p><code>r = random(obj)</code> generates a pseudo-random number <code>r</code> drawn from the piecewise distribution object <code>obj</code>.</p> <p><code>R = random(obj,n)</code> generates an <math>n</math>-by-<math>n</math> matrix of pseudo-random numbers <code>R</code>.</p> <p><code>R = random(obj,m,n)</code> or <code>R = random(obj,[m,n])</code> generates an <math>m</math>-by-<math>n</math> matrix of pseudo-random numbers <code>R</code>.</p> <p><code>R = random(obj,m,n,p,...)</code> or <code>R = random(obj,[m,n,p,...])</code> generates an <math>m</math>-by-<math>n</math>-by-<math>p</math>-by-... array of pseudo-random numbers <code>R</code>.</p>
<b>Examples</b>	<p>Fit Pareto tails to a <math>t</math> distribution at cumulative probabilities 0.1 and 0.9:</p> <pre>t = trnd(3,100,1); obj = paretotails(t,0.1,0.9);  r = random(obj) r =     0.8285</pre>
<b>See Also</b>	<code>paretotails</code>   <code>cdf</code>   <code>icdf</code>

# ProbDist.random

---

**Purpose** Generate random number drawn from ProbDist object

**Syntax**  
 $Y = \text{random}(PD)$   
 $Y = \text{random}(PD, N)$   
 $Y = \text{random}(PD, N, M, \dots)$

**Description**  
 $Y = \text{random}(PD)$  generates a random number drawn from the distribution specified by  $PD$ , a ProbDist object.  
 $Y = \text{random}(PD, N)$  generates an  $N$ -by- $N$  array of random numbers drawn from the distribution specified by  $PD$ , a ProbDist object.  
 $Y = \text{random}(PD, N, M, \dots)$  generates an  $N$ -by- $M$ -by- $\dots$  array of random numbers drawn from the distribution specified by  $PD$ , a ProbDist object.

**Input Arguments**

$PD$	An object of the class ProbDistUnivParam or ProbDistUnivKernel.
$N$	A positive integer.
$M$	A positive integer.

**Output Arguments**

$Y$	A random number drawn from the distribution specified by $PD$ .
-----	---

**See Also** random



# prob.TruncatableDistribution.random

## Purpose

Generate random numbers from probability distribution object

## Syntax

```
r = random(pd)
r = random(pd, sz1, ..., szN)
r = random(pd, [sz1, ..., szN])
```

## Description

`r = random(pd)` generates a random number `r` from the probability distribution `pd`.

`r = random(pd, sz1, ..., szN)` generates a `sz1`-by-...-by-`szN` array of random numbers from the probability distribution `pd`.

`r = random(pd, [sz1, ..., szN])` generates a `sz1`-by-...-by-`szN` array of random numbers from the probability distribution `pd`.

## Input Arguments

### **pd - Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

### **sz1, ..., szN - Size of each dimension**

two or more integer values | vector of integer values

Size of each dimension, specified as two or more integer values, or a vector of such values. For example, specifying `5,3,2` or `[5,3,2]` generates a 5-by-3-by-2 array of random numbers from the probability distribution `pd`.

### **Data Types**

single | double

# prob.TruncatableDistribution.random

---

## Output Arguments

### **r - Random number**

scalar value | array of values

Random number generated from the probability distribution, returned as a scalar value or an array of scalar values with the dimensions specified by `sz1`, ..., `szN`.

## Examples

### **Generate One Random Number**

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Generate one random number from the distribution.

```
r = random(pd)
```

```
r =
```

```
0.5377
```

### **Generate Multiple Random Numbers**

Create a Weibull probability distribution object using the default parameter values.

```
pd = makedist('Weibull')
```

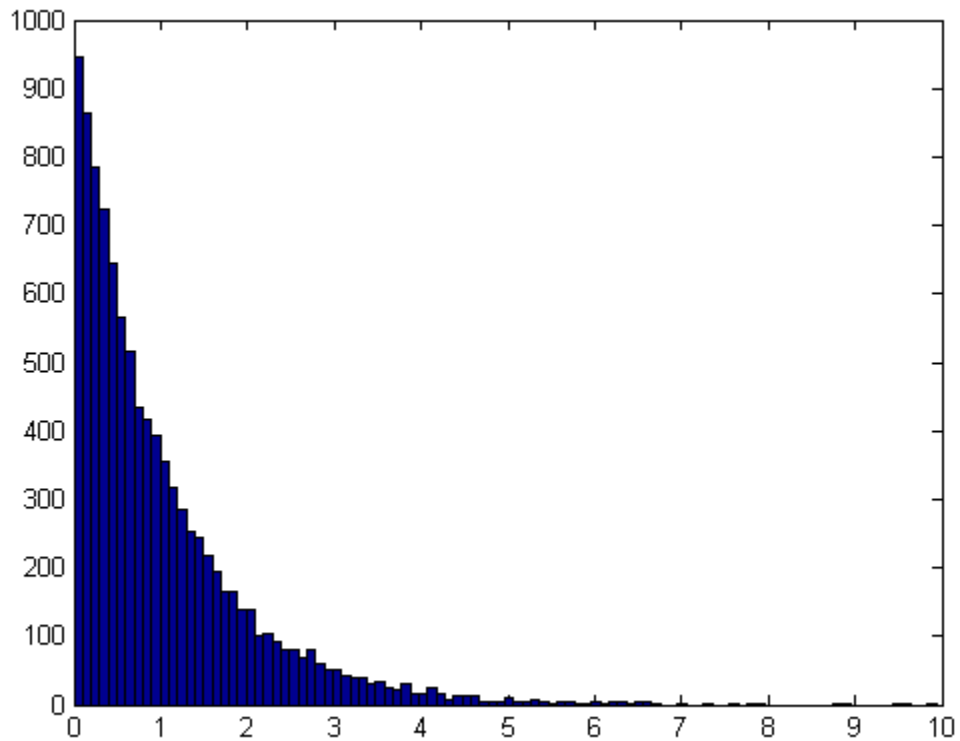
```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution  
A = 1  
B = 1
```

Generate random numbers from distribution and visualize with a histogram.

```
r = random(pd,10000,1);  
hist(r,100)
```



# prob.TruncatableDistribution.random

---

## Generate a Multidimensional Array of Random Numbers

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Generate a 2-by-3-by-2 array of random numbers from the distribution.

```
r = random(pd,[2,3,2])
```

```
random_make_array(:,:,1) =
```

```
-1.0689  -2.9443  0.3252  
-0.8095   1.4384 -0.7549
```

```
random_make_array(:,:,2) =
```

```
1.3703  -0.1022  0.3192  
-1.7115  -0.2414  0.3129
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Purpose

Estimates of random effects and related statistics

## Syntax

```
B = randomEffects(lme)
[B,Bnames] = randomEffects(lme)
[B,Bnames,stats] = randomEffects(lme)
[B,Bnames,stats] = randomEffects(lme,Name,Value)
```

## Description

`B = randomEffects(lme)` returns the estimates of the best linear unbiased predictors (BLUPs) of random effects in the linear mixed-effects model `lme`.

`[B,Bnames] = randomEffects(lme)` also returns the names of the coefficients in `Bnames`. Each name corresponds to a coefficient in `B`.

`[B,Bnames,stats] = randomEffects(lme)` also returns the estimated BLUPs of random effects in the linear mixed-effects model `lme` and related statistics.

`[B,Bnames,stats] = randomEffects(lme,Name,Value)` also returns the BLUPs of random effects in the linear mixed-effects model `lme` and related statistics with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **lme - Linear mixed-effects model**

`LinearMixedModel` object

Linear mixed-effects model, returned as a `LinearMixedModel` object.

For properties and methods of this object, see `LinearMixedModel`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can

# LinearMixedModel.randomEffects

---

specify several name and value pair arguments in any order as  
Name1, Value1, . . . , NameN, ValueN.

## 'Alpha' - Confidence level

0.05 (default) | scalar value in the range 0 to 1

Confidence level, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range 0 to 1. For a value  $\alpha$ , the confidence level is  $100*(1-\alpha)\%$ .

For example, for 99% confidence intervals, you can specify the confidence level as follows.

**Example:** 'Alpha', 0.01

## Data Types

single | double

## 'DFMethod' - Method for computing approximate degrees of freedom

'Residual' (default) | 'Satterthwaite' | 'None'

Method for computing approximate degrees of freedom for the  $t$ -statistics that test the random-effects coefficients against 0, specified as the comma-separated pair consisting of 'DFMethod' and one of the following.

'Residual'	Default. The degrees of freedom are assumed to be constant and equal to $n - p$ , where $n$ is the number of observations and $p$ is the number of fixed effects.
'Satterthwaite'	Satterthwaite approximation.
'None'	All degrees of freedom are set to infinity.

For example, you can specify the Satterthwaite approximation as follows.

Example: 'DFMethod', 'Satterthwaite'

## Output Arguments

### **B - Estimated best linear unbiased predictors of random effects**

column vector

Estimated best linear unbiased predictors of random effects of linear mixed-effects model `lme`, returned as a column vector.

Suppose `lme` has  $R$  grouping variables  $g_1, g_2, \dots, g_R$ , with levels  $m_1, m_2, \dots, m_R$ , respectively. Also suppose  $q_1, q_2, \dots, q_R$  are the lengths of the random-effects vectors that are associated with  $g_1, g_2, \dots, g_R$ , respectively. Then, **B** is a column vector of length  $q_1 * m_1 + q_2 * m_2 + \dots + q_R * m_R$ .

`randomEffects` creates **B** by concatenating the best linear unbiased predictors of random-effects vectors corresponding to each level of each grouping variable as `[g1level1; g1level2; ...; g1levelm1; g2level1; g2level2; ...; g2levelm2; ...; gRlevel1; gRlevel2; ...; gRlevelmR]`'.

### **Bnames - Names of random-effects coefficients**

dataset array

Names of random-effects coefficients in **B**, returned as a dataset array.

### **stats - Estimates of random effects BLUPs and related statistics**

dataset array

Estimates of random effects BLUPs and related statistics, returned as a dataset array that has one row for each of the fixed effects and one column for each of the following statistics.

Group	Grouping variable associated with the random effect
-------	---

Level	Level within the grouping variable corresponding to the random effect
-------	---

# LinearMixedModel.randomEffects

---

Name	Name of the random-effect coefficient
Estimate	Best linear unbiased predictor (BLUP) of random effect
SEPred	Standard error of the estimate (BLUP minus random effect)
tStat	$t$ -statistic for a test that the random effect is zero
DF	Estimated degrees of freedom for the $t$ -statistic
pValue	$p$ -value for the $t$ -statistic
Lower	Lower limit of a 95% confidence interval for the random effect
Upper	Upper limit of a 95% confidence interval for the random effect

## Examples

### Display Random-Effects Estimates and Coefficient Names

Load the sample data.

```
load carbig
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration and horsepower, and potentially correlated random effects for intercept and acceleration, grouped by the model year. First, store the data in a dataset array.

```
ds = dataset(Acceleration,Horsepower,Model_Year,MPG);
```

Fit the model.

```
lme = fitlme(ds, 'MPG ~ Acceleration + Horsepower + (Acceleration|Model_Y
```



Compute the BLUPs of the random-effects coefficients and display the names of the corresponding random effects.

```
[B,Bnames] = randomEffects(lme)
```

```
B =
```

```
 3.1270  
-0.2426  
-1.6532  
-0.0086  
 1.2075  
-0.2179  
 4.4107  
-0.4887  
-1.3103  
-0.0208  
 2.8029  
-0.3790  
 0.0865  
-0.1280  
 0.4216  
-0.0259  
-2.3889  
 0.1634  
 0.9618  
 0.0117  
-2.2345  
 0.5020  
-2.1332  
 0.3254  
-3.2979  
 0.5090
```

```
Bnames =
```

# LinearMixedModel.randomEffects

---

Group	Level	Name
'Model_Year'	'70'	'(Intercept)'
'Model_Year'	'70'	'Acceleration'
'Model_Year'	'71'	'(Intercept)'
'Model_Year'	'71'	'Acceleration'
'Model_Year'	'72'	'(Intercept)'
'Model_Year'	'72'	'Acceleration'
'Model_Year'	'73'	'(Intercept)'
'Model_Year'	'73'	'Acceleration'
'Model_Year'	'74'	'(Intercept)'
'Model_Year'	'74'	'Acceleration'
'Model_Year'	'75'	'(Intercept)'
'Model_Year'	'75'	'Acceleration'
'Model_Year'	'76'	'(Intercept)'
'Model_Year'	'76'	'Acceleration'
'Model_Year'	'77'	'(Intercept)'
'Model_Year'	'77'	'Acceleration'
'Model_Year'	'78'	'(Intercept)'
'Model_Year'	'78'	'Acceleration'
'Model_Year'	'79'	'(Intercept)'
'Model_Year'	'79'	'Acceleration'
'Model_Year'	'80'	'(Intercept)'
'Model_Year'	'80'	'Acceleration'
'Model_Year'	'81'	'(Intercept)'
'Model_Year'	'81'	'Acceleration'
'Model_Year'	'82'	'(Intercept)'
'Model_Year'	'82'	'Acceleration'

Since intercept and acceleration have potentially correlated random effects, grouped by model year of the cars, `randomEffects` creates a separate row for intercept and acceleration at each level of the grouping variable.

Compute the covariance parameters of the random effects.

```
[~,~,stats] = covarianceParameters(lme)
stats{1}
```

# LinearMixedModel.randomEffects

---

```
ans =
```

```
Covariance Type: FullCholesky
```

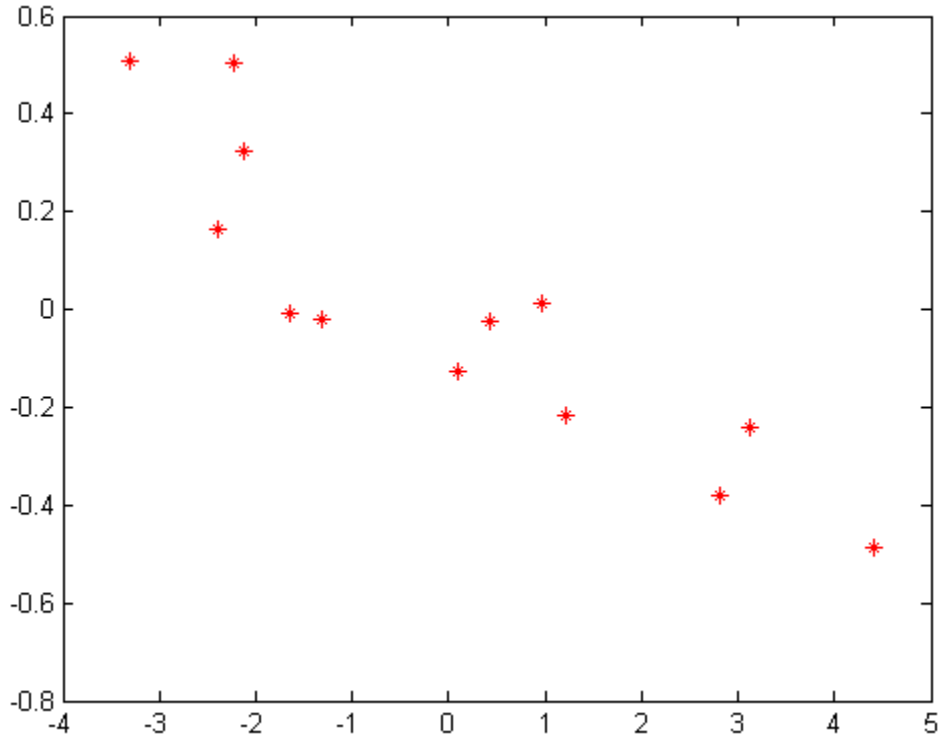
Group	Name1	Name2	Type
Model_Year	'(Intercept)'	'(Intercept)'	'std'
Model_Year	'Weight'	'(Intercept)'	'corr'
Model_Year	'Weight'	'Weight'	'std'

The correlation value suggests that random effects seem negatively correlated. Plot the random effects for intercept versus acceleration to confirm this.

```
plot(B(1:2:end),B(2:2:end),'r*')
```

# LinearMixedModel.randomEffects

---



## Compute Random-Effects Estimates and Related Statistics

Navigate to a folder containing sample data.

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

Load the sample data.

```
load fertilizer
```

The dataset array includes data from a split-plot experiment, where soil is divided into three blocks based on the soil type: sandy, silty, and loamy. Each block is divided into five plots, where five different types of tomato plants (cherry, heirloom, grape, vine, and plum) are randomly assigned to these plots. The tomato plants in the plots are then divided into subplots, where each subplot is treated by one of four fertilizers. This is simulated data.

Store the data in a dataset array called `ds`, for practical purposes, and define `Tomato`, `Soil`, and `Fertilizer` as categorical variables.

```
ds = fertilizer;  
ds.Tomato = nominal(ds.Tomato);  
ds.Soil = nominal(ds.Soil);  
ds.Fertilizer = nominal(ds.Fertilizer);
```

Fit a linear mixed-effects model, where `Fertilizer` and `Tomato` are the fixed-effects variables, and the mean yield varies by the block (soil type), and the plots within blocks (tomato types within soil types) independently.

```
lme = fitlme(ds, 'Yield ~ Fertilizer * Tomato + (1|Soil) + (1|Soil:Tomato)');
```

Compute the BLUPs and related statistics for random effects.

```
[~,~,stats] = randomEffects(lme)
```

```
stats =
```

```
Random effect coefficients: DFMethod = 'Residual', Alpha = 0.05
```

Group	Level	Name
'Soil'	'Loamy'	'(Intercept)'
'Soil'	'Sandy'	'(Intercept)'
'Soil'	'Silty'	'(Intercept)'
'Soil:Tomato'	'Loamy Cherry'	'(Intercept)'
'Soil:Tomato'	'Loamy Grape'	'(Intercept)'

# LinearMixedModel.randomEffects

---

'Soil:Tomato'	'Loamy Heirloom'	'(Intercept)'	16
'Soil:Tomato'	'Loamy Plum'	'(Intercept)'	-5.3
'Soil:Tomato'	'Loamy Vine'	'(Intercept)'	-4.4
'Soil:Tomato'	'Sandy Cherry'	'(Intercept)'	-17.3
'Soil:Tomato'	'Sandy Grape'	'(Intercept)'	-7.3
'Soil:Tomato'	'Sandy Heirloom'	'(Intercept)'	-8.3
'Soil:Tomato'	'Sandy Plum'	'(Intercept)'	7.3
'Soil:Tomato'	'Sandy Vine'	'(Intercept)'	0.2
'Soil:Tomato'	'Silty Cherry'	'(Intercept)'	4.3
'Soil:Tomato'	'Silty Grape'	'(Intercept)'	10.3
'Soil:Tomato'	'Silty Heirloom'	'(Intercept)'	-8.3
'Soil:Tomato'	'Silty Plum'	'(Intercept)'	-2.3
'Soil:Tomato'	'Silty Vine'	'(Intercept)'	4.3

The first three rows contain the random-effects estimates and the statistics for the three levels, Loamy, Sandy, and Silty of the grouping variable `Soil`. The corresponding  $p$ -values 0.66918, 0.51825, and 0.82593 indicate that these random-effects are not significantly different from 0. The following 15 rows include the BLUPS of random-effects estimates for the intercept, grouped by the variable `Tomato` nested in `Soil`, i.e. interaction of `Tomato` and `Soil`.

## Compute Confidence Intervals with Specified Options

Load the sample data.

```
load carsmall
```

`Shift` and `Operator` are nominal variables.

```
shift.Shift = nominal(shift.Shift);  
shift.Operator = nominal(shift.Operator);
```

Fit a linear mixed-effects model with a random intercept grouped by `operator`, to assess if there is a significant difference in the performance according to the time of the shift. Use the restricted maximum likelihood method.

```
lme = fitlme(shift, 'QCDev ~ Shift + (1|Operator)');
```

# LinearMixedModel.randomEffects

Compute the 99% confidence intervals for random effects using the residuals option to compute the degrees of freedom. This is the default method.

```
[~,~,stats] = randomEffects(lme, 'alpha', 0.01)
```

```
stats =
```

```
Random effect coefficients: DFMethod = 'Residual', Alpha = 0.01
```

Group	Level	Name	Estimate	SEPr
'Operator'	'1'	'(Intercept)'	0.57753	0.90
'Operator'	'2'	'(Intercept)'	1.1757	0.90
'Operator'	'3'	'(Intercept)'	-2.1715	0.90
'Operator'	'4'	'(Intercept)'	2.3655	0.90
'Operator'	'5'	'(Intercept)'	-1.9472	0.90

Compute the 99% confidence intervals for random effects using the Satterthwaite approximation to compute the degrees of freedom.

```
[~,~,stats] = randomEffects(lme, 'DFMethod', 'Satterthwaite', 'alpha', 0.01)
```

```
stats =
```

```
Random effect coefficients: DFMethod = 'Satterthwaite', Alpha = 0.01
```

Group	Level	Name	Estimate	SEPr
'Operator'	'1'	'(Intercept)'	0.57753	0.90
'Operator'	'2'	'(Intercept)'	1.1757	0.90
'Operator'	'3'	'(Intercept)'	-2.1715	0.90
'Operator'	'4'	'(Intercept)'	2.3655	0.90
'Operator'	'5'	'(Intercept)'	-1.9472	0.90

The Satterthwaite method usually produces smaller values for the degrees of freedom (DF), which results in larger  $p$ -values (pValue) and

# LinearMixedModel.randomEffects

---

larger confidence intervals (Lower and Upper) for the random-effects estimates.

## See Also

[LinearMixedModel](#) | [fitlmecoefCI](#) | [coefTest](#) | [fixedEffects](#) |



## Purpose

Random sample

## Syntax

```
y = randsample(n,k)
y = randsample(population,k)
y = randsample(n,k,replacement)
y = randsample(population,k,replacement)
y = randsample(n,k,true,w)
y = randsample(population,k,true,w)
y = randsample(s,...)
```

## Description

`y = randsample(n,k)` returns a  $k$ -by-1 vector  $y$  of values sampled uniformly at random, without replacement, from the integers 1 to  $n$ .

`y = randsample(population,k)` returns a vector of  $k$  values sampled uniformly at random, without replacement, from the values in the vector `population`. The orientation of  $y$  (row or column) is the same as `population`.

`y = randsample(n,k,replacement)` or `y = randsample(population,k,replacement)` returns a sample taken with replacement if `replacement` is `true`, or without replacement if `replacement` is `false`. The default is `false`.

`y = randsample(n,k,true,w)` or `y = randsample(population,k,true,w)` returns a weighted sample taken with replacement, using a vector of positive weights  $w$ , whose length is  $n$ . The probability that the integer  $i$  is selected for an entry of  $y$  is  $w(i) / \text{sum}(w)$ . Usually,  $w$  is a vector of probabilities. `randsample` does not support weighted sampling without replacement.

`y = randsample(s,...)` uses the stream `s` for random number generation. `s` is a member of the `RandStream` class. Default is the MATLAB default random number stream.

## Examples

Draw a single value from the integers 1 through 10:

```
n = 10;
x = randsample(n,1);
```

# randsample

---

Draw a single value from the population 1 through n, where  $n > 1$ :

```
y = randsample(1:n,1);
```

---

**Note** If population is a numeric vector containing only nonnegative integer values, and population can have length 1, use

```
y = population(randsample(length(population),k))
```

instead of `y = randsample(population,k)`.

---

Generate a random sequence of the characters A, C, G, and T, with replacement, according to the specified probabilities.

```
R = randsample('ACGT',48,true,[0.15 0.35 0.35 0.15])
```

- To randomly sample data, with or without replacement, use `datasample`.

## See Also

`rand` | `randperm` | `datasample` | `RandStream`

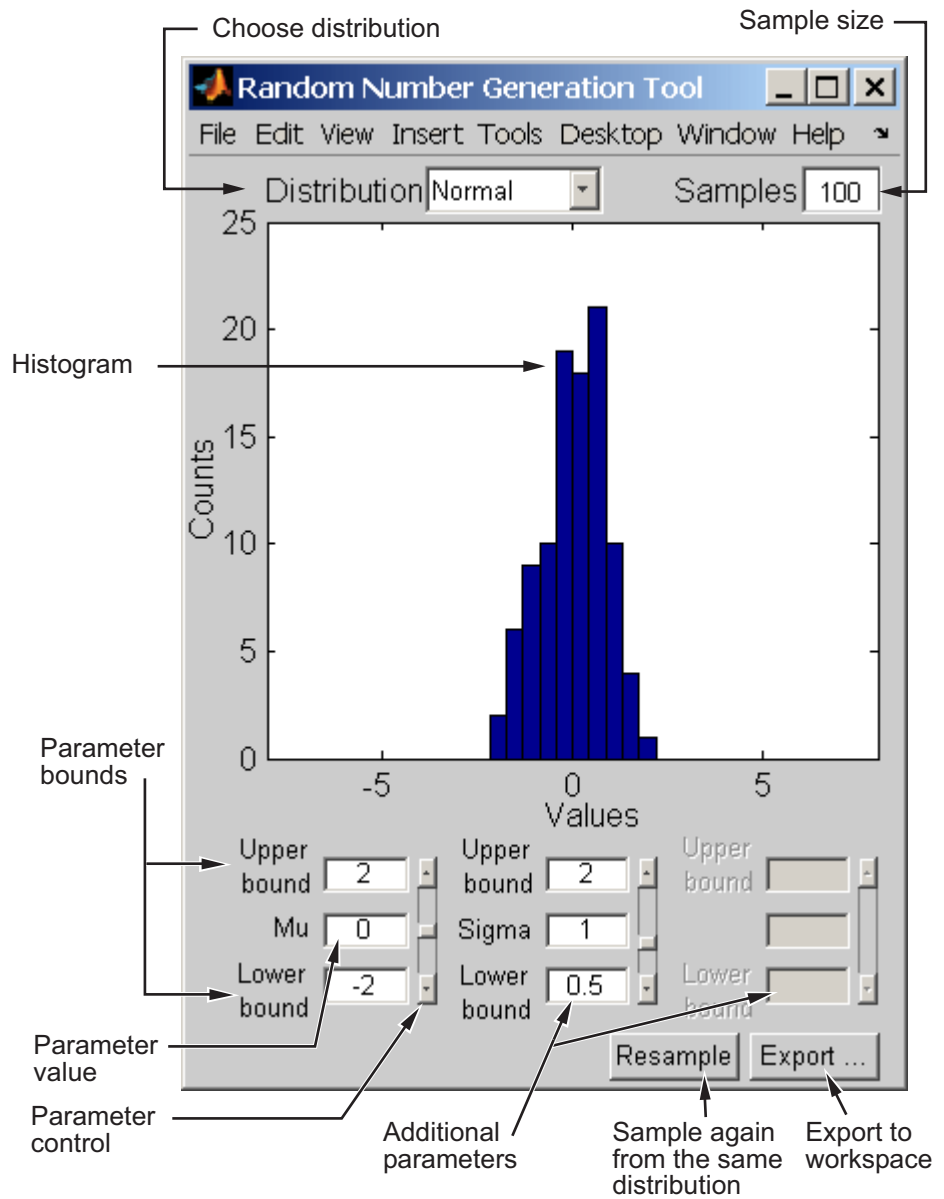
**Purpose** Interactive random number generation

**Syntax** randtool

**Description** randtool opens the Random Number Generation Tool.

The Random Number Generation Tool is a graphical user interface that generates random samples from specified probability distributions and displays the samples as histograms. Use the tool to explore the effects of changing parameters and sample size on the distributions.

# randtool



Start by selecting a distribution, then enter the desired sample size.

You can also

- Use the controls at the bottom of the window to set parameter values for the distribution and to change their upper and lower bounds.
- Draw another sample from the same distribution, with the same size and parameters.
- Export the current sample to your workspace. A dialog box enables you to provide a name for the sample.

**See Also**

`disttool` | `dfittool`

# range

---

**Purpose** Range of values

**Syntax** `range(X)`  
`y = range(X,dim)`

**Description** `range(X)` returns the difference between the maximum and the minimum of a sample. For vectors, `range(x)` is the range of the elements. For matrices, `range(X)` is a row vector containing the range of each column of `X`. For N-dimensional arrays, `range` operates along the first nonsingleton dimension of `X`.

`y = range(X,dim)` operates along the dimension `dim` of `X`.

`range` treats NaNs as missing values and ignores them.

The range is an easily-calculated estimate of the spread of a sample. Outliers have an undue influence on this statistic, which makes it an unreliable estimator.

**Examples** The range of a large sample of standard normal random numbers is approximately six. This is the motivation for the process capability indices  $C_p$  and  $C_{pk}$  in statistical quality control applications.

```
rv = normrnd(0,1,1000,5);
near6 = range(rv)
near6 =
    6.1451    6.4986    6.2909    5.8894    7.0002
```

**See Also** `std` | `iqr` | `mad`

**Purpose**

Find all neighbors within specified distance

**Syntax**

```
idx = rangesearch(X,Y,r)
[idx,D]= rangesearch(X,Y,r)
[idx,D]= rangesearch(X,Y,r,Name,Value)
```

**Description**

`idx = rangesearch(X,Y,r)` finds all the  $X$  points that are within distance  $r$  of the  $Y$  points. Rows of  $X$  and  $Y$  correspond to observations, and columns correspond to variables.

`[idx,D]= rangesearch(X,Y,r)` returns the distances between each row of  $Y$  and the rows of  $X$  that are  $r$  or less distant.

`[idx,D]= rangesearch(X,Y,r,Name,Value)` finds nearby points with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For a fixed positive integer  $K$ , `knnsearch` finds  $K$  points in  $X$  that are nearest each  $Y$  point. In contrast, for a fixed positive real value  $r$ , `rangesearch` finds all the  $X$  points that are within a distance  $r$  of each  $Y$  point.

**Input Arguments****X**

$m \times n$ -by- $n$  numeric matrix, where each row represents one  $n$ -dimensional point. The number of columns  $n$  must equal as the number of columns in  $Y$ .

**Y**

$m \times n$ -by- $n$  numeric matrix, where each row represents one  $n$ -dimensional point. The number of columns  $n$  must equal as the number of columns in  $X$ .

**r**

Search radius, a scalar. `rangesearch` finds all  $X$  points (rows) that are within distance  $r$  of each  $Y$  point. The meaning of distance depends on the `Distance` name-value pair.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### 'BucketSize'

Maximum number of data points in the leaf node of the *kd*-tree. This argument is only meaningful when *kd*-tree is used for finding nearest neighbors.

**Default:** 50

### 'Cov'

Positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This argument is only valid when the `Distance` name-value pair is 'mahalanobis'.

**Default:** `nancov(X)`

### 'Distance'

String or function handle specifying the distance metric.

Value	Description
'euclidean'	Euclidean distance.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between <code>X</code> and a query point is scaled, meaning divided by a scale value <code>S</code> . The default value of <code>S</code> is the standard deviation computed from <code>X</code> , <code>S=nansd(X)</code> . To specify another value for <code>S</code> , use the <code>Scale</code> name-value pair.



Value	Description
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair.
'cityblock'	City block distance.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' name-value pair.
'chebychev'	Chebychev distance (maximum coordinate difference).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.

# rangesearch

---

Value	Description
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	Distance function handle. <i>distfun</i> has the form  function D2 = DISTFUN(ZI,ZJ) ...  where <ul style="list-style-type: none"><li>• ZI is a 1-by-N vector containing one row of X or Y.</li><li>• ZJ is an M2-by-N matrix containing multiple rows of X or Y.</li><li>• D2 is an M2-by-1 vector of distances, D2(k) is the distance between the observations ZI and ZJ(J,:).</li></ul>

For definitions, see “Distance Metrics” on page 16-9.

**Default:** 'euclidean'

**'NSMethod'**

Nearest neighbors search method.

Value	Meaning
'kdtree'	Creates and uses a <i>kd</i> -tree to find nearest neighbors. 'kdtree' is only valid when the distance metric is one of: <ul style="list-style-type: none"> <li>• 'chebyshev'</li> <li>• 'cityblock'</li> <li>• 'euclidean'</li> <li>• 'minkowski'</li> </ul>
'exhaustive'	Uses the exhaustive search algorithm. The distances from all X points to each Y point are computed to find nearest neighbors.

**Default:** 'kdtree' when the number of columns of X is not greater than 10, X is not sparse, and the distance metric is one of the valid 'kdtree' metrics. Otherwise, the default is 'exhaustive'.

**'P'**

Positive scalar indicating the exponent of Minkowski distance. This argument is only valid when the Distance name-value pair is 'minkowski'.

**Default:** 2

**'Scale'**

Vector S containing nonnegative values, with length equal to the number of columns in X. Each coordinate difference between X and a query point is scaled by the corresponding element of S. This argument is only valid when the Distance name-value pair is 'seuclidean'.

**Default:** nanstd(X)

# rangesearch

---

## Output Arguments

### **idx**

*my*-by-1 cell array, where *my* is the number of rows in *Y*. `idx{I}` contains the indices of points (rows) in *X* whose distances to  $Y(I, :)$  are not greater than *r*. The entries in `idx{I}` are in ascending order of distance.

### **D**

*my*-by-1 cell array, where *my* is the number of rows in *Y*. `D{I}` contains the distance values between  $Y(I, :)$  and the corresponding points in `idx{I}`.

## Definitions

### **Distance Metrics**

For definitions, see “Distance Metrics” on page 16-9.

## Examples

Find the *X* points that are within a Euclidean distance 1.5 of each *Y* point. Both *X* and *Y* are samples of 5-D normally distributed variables.

```
rng('default') % for reproducibility
X = randn(100,5);
Y = randn(10,5);
[idx, dist] = rangesearch(X,Y,1.5)
```

```
idx =
    [1x7 double]
    [1x2 double]
    [1x11 double]
    [1x2 double]
    [1x12 double]
    [1x9 double]
    [      89]
    [1x0 double]
    [1x0 double]
    [1x0 double]
```

```
dist =
```

```

[1x7 double]
[1x2 double]
[1x11 double]
[1x2 double]
[1x12 double]
[1x9 double]
[    1.1739]
[1x0 double]
[1x0 double]
[1x0 double]

```

In this case, the last three Y points are more than 1.5 distant from any X point. X(89, :) is 1.1739 distant from Y(7, :), and there is no other X point that is within distance 1.5 of Y(7, :). There are 12 points in X within distance 1.5 of Y(5, :).

## Algorithms

For an overview of the *kd*-tree algorithm, see “*k*-Nearest Neighbor Search Using a *kd*-Tree” on page 16-14.

The exhaustive search algorithm finds the distance of each point in X to each point in Y.

## Alternatives

`rangesearch` is the `ExhaustiveSearcher` method for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to 'exhaustive'.

`rangesearch` is the `KDTreeSearcher` method for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to 'kdtree'.

## See Also

`createns` | `ExhaustiveSearcher` | `KDTreeSearcher` | `knnsearch` | `pdist2`

## How To

- “*k*-Nearest Neighbor Search and Radius Search” on page 16-12

# ExhaustiveSearcher.rangesearch

---

**Purpose** Find all neighbors within specified distance using ExhaustiveSearcher object

**Syntax**  
`idx = rangesearch(NS,Y,r)`  
`[idx,D]= rangesearch(NS,Y,r)`  
`[idx,D]= rangesearch(NS,Y,r,Name,Value)`

**Description** `idx = rangesearch(NS,Y,r)` finds all points in `NS.X` that are within distance `r` of the `Y` points. Rows of `NS.X` and `Y` correspond to observations, and columns correspond to variables.

`[idx,D]= rangesearch(NS,Y,r)` returns the distances between each row of `Y` and the rows of `NS.X` that are `r` or less distant.

`[idx,D]= rangesearch(NS,Y,r,Name,Value)` finds nearby points with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For a fixed positive integer `K`, `knnsearch` finds the `K` points in `NS.X` that are nearest each `Y` point. In contrast, for a fixed positive real value `r`, `rangesearch` finds all the points in `NS.X` that are within a distance `r` of each `Y` point.

## Input Arguments

### **NS**

ExhaustiveSearcher object, constructed using ExhaustiveSearcher or createns.

### **Y**

*m*-by-*n* numeric matrix, where each row represents one *n*-dimensional point. The number of columns *n* must equal the number of columns in `NS.X`.

### **r**

Search radius, a scalar. `rangesearch` finds all `NS.X` points (rows) that are within distance `r` of each `Y` point. The meaning of distance depends on the `Distance` name-value pair.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### 'Cov'

Positive definite matrix indicating the covariance matrix when computing the Mahalanobis distance. This argument is only valid when the `Distance` name-value pair is 'mahalanobis'.

**Default:** `nancov(X)`

### 'Distance'

String or function handle specifying the distance metric.

Value	Description
'euclidean'	Euclidean distance.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between <code>X</code> and a query point is scaled, meaning divided by a scale value <code>S</code> . The default value of <code>S</code> is the standard deviation computed from <code>X</code> , <code>S=nanstd(X)</code> . To specify another value for <code>S</code> , use the <code>Scale</code> name-value pair.
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix <code>C</code> . The default value of <code>C</code> is the sample covariance matrix of <code>X</code> , as computed by <code>nancov(X)</code> . To specify a different value for <code>C</code> , use the 'Cov' name-value pair.
'cityblock'	City block distance.

# ExhaustiveSearcher.rangearch

---

Value	Description
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' name-value pair.
'chebychev'	Chebychev distance (maximum coordinate difference).
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'correlation'	One minus the sample linear correlation between observations (treated as sequences of values).
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	Distance function handle. <i>distfun</i> has the form  function D2 = DISTFUN(ZI,ZJ) ...  where <ul style="list-style-type: none"><li>• ZI is a 1-by-N vector containing one row of X or Y.</li><li>• ZJ is an M2-by-N matrix containing multiple rows of X or Y.</li></ul>



Value	Description
	<ul style="list-style-type: none"><li>D2 is an M2-by-1 vector of distances, D2(k) is the distance between the observations ZI and ZJ(J, :).</li></ul>

For definitions, see “Distance Metrics” on page 16-9.

**Default:** 'euclidean'

**'P'**

Positive scalar indicating the exponent of Minkowski distance. This argument is only valid when the Distance name-value pair is 'minkowski'.

**Default:** 2

**'Scale'**

Vector S containing nonnegative values, with length equal to the number of columns in X. Each coordinate difference between X and a query point is scaled by the corresponding element of S. This argument is only valid when the Distance name-value pair is 'seuclidean'.

**Default:** nanstd(X)

## Output Arguments

**idx**

*m<sub>y</sub>*-by-1 cell array, where *m<sub>y</sub>* is the number of rows in Y. idx{I} contains the indices of points (rows) in NS.X whose distances to Y(I, :) are not greater than r. The entries in idx{I} are in ascending order of distance.

**D**

# ExhaustiveSearcher.rangesearch

---

$m_y$ -by-1 cell array, where  $m_y$  is the number of rows in  $Y$ .  
 $D\{I\}$  contains the distance values between  $Y(I,:)$  and the corresponding points in  $idx\{I\}$ .

## Definitions

### Distance Metrics

For definitions, see “Distance Metrics” on page 16-9.

## Examples

Create  $X$  and  $Y$  as samples of 5-D normally distributed variables. Create an ExhaustiveSearcher object  $NS$  from  $X$ . Find the points in  $NS.X$  that are within a Euclidean distance 1.5 of each point in  $Y$ .

```
rng('default') % for reproducibility
X = randn(100,5);
Y = randn(10,5);
NS = ExhaustiveSearcher(X);
[idx, dist] = rangesearch(NS,Y,1.5)
```

```
idx =
    [1x7 double]
    [1x2 double]
    [1x11 double]
    [1x2 double]
    [1x12 double]
    [1x9 double]
    [          89]
    [1x0 double]
    [1x0 double]
    [1x0 double]
```

```
dist =
    [1x7 double]
    [1x2 double]
    [1x11 double]
    [1x2 double]
    [1x12 double]
    [1x9 double]
```

```
[      1.1739]
[1x0  double]
[1x0  double]
[1x0  double]
```

In this case, the last three points in  $Y$  are more than 1.5 distant from any point in  $NS.X$ .  $NS.X(89, :)$  is 1.1739 distant from  $Y(7, :)$ , and there is no other point in  $NS.X$  that is within distance 1.5 of  $Y(7, :)$ . There are 12 points in  $NS.X$  within distance 1.5 of  $Y(5, :)$ .

## Algorithms

The exhaustive search algorithm finds the distance of each point in  $X$  to each point in  $Y$ .

## Alternatives

`rangesearch` is the `ExhaustiveSearcher` method for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to `'exhaustive'`.

`rangesearch` is the `KDTreeSearcher` method for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to `'kdtree'`.

## See Also

`createns` | `ExhaustiveSearcher` | `knnsearch` | `pdist2` | `rangesearch`

# KDTreeSearcher.rangesearch

---

**Purpose** Find all neighbors within specified distance using KDTreeSearcher object

**Syntax**  
`idx = rangesearch(NS,Y,r)`  
`[idx,D]= rangesearch(NS,Y,r)`  
`[idx,D]= rangesearch(NS,Y,r,Name,Value)`

**Description** `idx = rangesearch(NS,Y,r)` finds all points in `NS.X` that are within distance `r` of the `Y` points. Rows of `NS.X` and `Y` correspond to observations, and columns correspond to variables.

`[idx,D]= rangesearch(NS,Y,r)` returns the distances between each row of `Y` and the rows of `NS.X` that are `r` or less distant.

`[idx,D]= rangesearch(NS,Y,r,Name,Value)` finds nearby points with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- For a fixed positive integer `K`, `knnsearch` finds the `K` points in `NS.X` that are nearest each `Y` point. In contrast, for a fixed positive real value `r`, `rangesearch` finds all the points in `NS.X` that are within a distance `r` of each `Y` point.

## Input Arguments

### NS

KDTreeSearcher object, constructed using KDTreeSearcher or createns.

### Y

$m$ -by- $n$  numeric matrix, where each row represents one  $n$ -dimensional point. The number of columns  $n$  must equal the number of columns in `NS.X`.

### r

Search radius, a scalar. `rangesearch` finds all `NS.X` points (rows) that are within distance `r` of each `Y` point. The meaning of distance depends on the `Distance` name-value pair.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### 'Distance'

String or function handle specifying the distance metric.

Value	Description
'euclidean'	Euclidean distance.
'cityblock'	City block distance.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' name-value pair.
'chebychev'	Chebychev distance (maximum coordinate difference).

For definitions, see “Distance Metrics” on page 16-9.

**Default:** `NS.Distance`

### 'P'

Positive scalar indicating the exponent of Minkowski distance. This argument is only valid when the `Distance` name-value pair is `'minkowski'`.

**Default:** 2

## Output Arguments

### `idx`

$m_y$ -by-1 cell array, where  $m_y$  is the number of rows in `Y`. `idx{I}` contains the indices of points (rows) in `NS.X` whose distances

# KDTreeSearcher.rangesearch

---

to  $Y(I, :)$  are not greater than  $r$ . The entries in  $idx\{I\}$  are in ascending order of distance.

## **D**

$m_y$ -by-1 cell array, where  $m_y$  is the number of rows in  $Y$ .  
 $D\{I\}$  contains the distance values between  $Y(I, :)$  and the corresponding points in  $idx\{I\}$ .

## **Definitions**

### **Distance Metrics**

For definitions, see “Distance Metrics” on page 16-9.

## **Examples**

Create  $X$  and  $Y$  as samples of 5-D normally distributed variables. Create a KDTreeSearcher object  $NS$  from  $X$ . Find the points in  $NS.X$  that are within a Euclidean distance 1.5 of each point in  $Y$ .

```
rng('default') % for reproducibility
X = randn(100,5);
Y = randn(10,5);
NS = KDTreeSearcher(X);
[idx, dist] = rangesearch(NS,Y,1.5)
```

```
idx =
    [1x7 double]
    [1x2 double]
    [1x11 double]
    [1x2 double]
    [1x12 double]
    [1x9 double]
    [      89]
    [1x0 double]
    [1x0 double]
    [1x0 double]
```

```
dist =
    [1x7 double]
    [1x2 double]
```

```
[1x11 double]
[1x2  double]
[1x12 double]
[1x9  double]
[      1.1739]
[1x0  double]
[1x0  double]
[1x0  double]
```

In this case, the last three points in  $Y$  are more than 1.5 distant from any point in  $NS.X$ .  $NS.X(89, :)$  is 1.1739 distant from  $Y(7, :)$ , and there is no other point in  $NS.X$  that is within distance 1.5 of  $Y(7, :)$ . There are 12 points in  $NS.X$  within distance 1.5 of  $Y(5, :)$ .

## Algorithms

For an overview of the *kd*-tree algorithm, see “*k*-Nearest Neighbor Search Using a *kd*-Tree” on page 16-14.

## Alternatives

`rangesearch` is the `KDTreeSearcher` method for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to `'kdtree'`.

`rangesearch` is the `ExhaustiveSearcher` method for distance search. It is equivalent to the `rangesearch` function with the `NSMethod` name-value pair set to `'exhaustive'`.

## See Also

`createns` | `KDTreeSearcher` | `knnsearch` | `pdist2` | `rangesearch`

# ranksum

---

**Purpose** Wilcoxon rank sum test

**Syntax**

```
p = ranksum(x,y)
[p,h] = ranksum(x,y)
[p,h,stats] = ranksum(x,y)
[ ___ ] = ranksum(x,y,Name,Value)
```

**Description** `p = ranksum(x,y)` returns the  $p$ -value of a two-sided Wilcoxon rank sum test. `ranksum` tests the null hypothesis that data in `x` and `y` are samples from continuous distributions with equal medians, against the alternative that they are not. The test assumes that the two samples are independent. `x` and `y` can have different lengths.

This test is equivalent to a Mann-Whitney U-test.

`[p,h] = ranksum(x,y)` also returns a logical value indicating the test decision. The result `h = 1` indicates a rejection of the null hypothesis, and `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level.

`[p,h,stats] = ranksum(x,y)` also returns the structure `stats` with information about the test statistic.

`[ ___ ] = ranksum(x,y,Name,Value)` returns any of the output arguments in the previous syntaxes, for a rank sum test with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **x - Sample data**

vector

Sample data, specified as a vector.

### **Data Types**

single | double

### **y - Sample data**



vector

Sample data, specified as a vector. The length of *y* does not have to be the same as the length of *x*.

### Data Types

single | double

### Name-Value Pair Arguments

Specify optional comma-separated pairs of *Name*, *Value* arguments. *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as *Name1*, *Value1*, ..., *NameN*, *ValueN*.

**Example:** 'alpha',0.01,'method','approximate','tail','right' specifies a right-tailed rank sum test with 1% significance level, which returns the approximate *p*-value.

### 'alpha' - Significance level

0.05 (default) | scalar value in the range 0 to 1

Significance level of the decision of a hypothesis test, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range 0 to 1. The significance level of *h* is 100 \* alpha%.

**Example:** 'alpha',0.01

### Data Types

double | single

### 'method' - Computation method of the *p*-value

'exact' | 'approximate'

Computation method of the *p*-value, *p*, specified as the comma-separated pair consisting of 'method' and one of the following:

'exact'	Exact computation of the <i>p</i> -value, <i>p</i> .
'approximate'	Normal approximation while computing the <i>p</i> -value, <i>p</i> .

# ranksum

---

When 'method' is unspecified, the default is:

- 'exact' if  $\min(n_x, n_y) < 10$  and  $n_x + n_y < 20$
- 'approximate' otherwise

$n_x$  and  $n_y$  are the sizes of the samples in x and y, respectively.

**Example:** 'method', 'exact'

## Data Types

char

## 'tail' - Type of test

'both' (default) | 'right' | 'left'

Type of test, specified as the comma-separated pair consisting of 'tail' and one of the following:

'both'	Two-sided hypothesis test, where the alternative hypothesis states that x and y have different medians. Default test type if 'tail' is not specified.
'right'	Right-tailed hypothesis test, where the alternative hypothesis states that the median of x is greater than the median of y.
'left'	Left-tailed hypothesis test, where the alternative hypothesis states that the median of x is less than the median of y.

**Example:** 'tail', 'left'

## Data Types

char

## Output Arguments

### p - p-value of the test

nonnegative scalar

$p$ -value of the test, returned as a positive scalar from 0 to 1.  $p$  is the probability of observing a test statistic as or more extreme than

the observed value under the null hypothesis. `ranksum` computes the two-sided  $p$ -value by doubling the most significant one-sided value.

### **h - Result of the hypothesis test**

1 | 0

Result of the hypothesis test, returned as a logical value.

- If  $h = 1$ , this indicates rejection of the null hypothesis at the  $100 * \alpha\%$  significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the  $100 * \alpha\%$  significance level.

### **stats - Test statistics**

structure

Test statistics, returned as a structure. The test statistics stored in `stats` are:

- `ranksum` : Value of the rank sum test statistic
- `zval`: Value of the z-statistic (computed when 'method' is 'approximate')

## **Examples**

### **Test for Equal Median of Two Populations**

Test the hypothesis of equal medians for two independent unequal-sized samples.

Generate sample data.

```
rng('default') % for reproducibility
x = unifrnd(0,1,10,1);
y = unifrnd(0.25,1.25,15,1);
```

These samples come from populations with identical distributions except for a shift of 0.25 in the location.

Test the equality of medians of  $x$  and  $y$ .

# ranksum

---

```
p = ranksum(x,y)
```

```
p =  
0.0375
```

The  $p$ -value of 0.0375 indicates that `ranksum` rejects the null hypothesis of equal medians at the default 5% significance level.

## Statistics of the Test for Two Population Medians

Obtain the statistics of the test for the equality of two population medians.

Load the sample data.

```
load mileage
```

Test if the mileage per gallon is the same for the first and second type of cars.

```
[p,h,stats] = ranksum(mileage(:,1),mileage(:,2))
```

```
p =  
0.0043
```

```
h =  
1
```

```
stats =  
ranksum: 21.5000
```

Both the  $p$ -value, 0.043, and  $h = 1$  indicate the rejection of the null hypothesis of equal medians at the default 5% significance level. Because the sample sizes are small (six each), `ranksum` calculates the  $p$ -value using the exact method. The structure `stats` includes only the value of the rank sum test statistic.

## Increase in the Median

Test the hypothesis of an increase in the population median.

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weather
```

The weather data shows the daily high temperatures taken in the same month in two consecutive years.

Perform a left-sided test to assess the increase in the median at the 1% significance level.

```
[p,h,stats] = ranksum(year1,year2,'alpha',0.01,...
'tail','left')
```

```
p =
```

```
    0.1271
```

```
h =
```

```
    0
```

```
stats =
```

```
    zval: -1.1403
    ranksum: 837.5000
```

Both the  $p$ -value of 0.1271 and  $h = 0$  indicate that there is not enough evidence to reject the null hypothesis and conclude that there is a positive shift in the median of observed high temperatures in the same month from year 1 to year 2 at the 1% significance level. Notice that

ranksum uses the approximate method to calculate the  $p$ -value due to the large sample sizes.

Use the exact method to calculate the  $p$ -value.

```
[p,h,stats] = ranksum(year1,year2,'alpha',0.01,...  
'tail','left','method','exact')
```

```
p =
```

```
    0.1273
```

```
h =
```

```
    0
```

```
stats =
```

```
ranksum: 837.5000
```

The results of the approximate and exact methods are consistent with each other.

## Definitions

### Wilcoxon Rank Sum Test

The Wilcoxon rank sum test is a nonparametric test for two populations when samples are independent. If  $X$  and  $Y$  are independent samples with different sample sizes, the test statistic which ranksum returns is the rank sum of the first sample.

The Wilcoxon rank sum test is equivalent to the Mann-Whitney U-test. The Mann-Whitney U-test is a nonparametric test for equality of population medians of two independent samples  $X$  and  $Y$ .

The Mann-Whitney U-test statistic,  $U$ , is the number of times a  $y$  precedes an  $x$  in an ordered arrangement of the elements in the two

independent samples  $X$  and  $Y$ . It is related to the Wilcoxon rank sum statistic in the following way: If  $X$  is a sample of size  $n_X$ , then

$$U = W - \frac{n_X(n_X + 1)}{2}.$$

### **z-Statistic**

For large samples, ranksum uses a  $z$ -statistic to compute the approximate  $p$ -value of the test.

If  $X$  and  $Y$  are two independent samples of size  $n_X$  and  $n_Y$ , where  $n_X < n_Y$  the  $z$ -statistic is

$$z = \frac{W - E(W)}{\sqrt{V(W)}} = \frac{W - \left[ \frac{n_X n_Y + n_X(n_X + 1)}{2} \right] - 0.5 * \text{sign}(W - E(W))}{\sqrt{\frac{n_X n_Y (n_X + n_Y + 1) - \text{tiescor}}{12}}},$$

with continuity correction and tie adjustment. Here *tiescor* is given by

$$\text{tiescor} = \frac{2 * \text{tieadj}}{(n_X + n_Y)(n_X + n_Y - 1)},$$

where ranksum uses `[ranks, tieadj] = tiedrank(x, y)` to obtain tie adjustments. The standard normal distribution gives the  $p$ -value for this  $z$ -statistic.

## **Algorithms**

ranksum treats NaNs in  $x$  and  $y$  as missing values and ignores them.

For a two-sided test of medians with unequal sample sizes, the test statistic that ranksum returns is the rank sum of the first sample.

## References

[1] Gibbons, J. D., and S. Chakraborti. *Nonparametric Statistical Inference*, 5th Ed., Boca Raton, FL: Chapman & Hall/CRC Press, Taylor & Francis Group, 2011.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

## See Also

`kruskalwallis` | `signrank` | `signtest` | `ttest2`



**Purpose** Rayleigh cumulative distribution function

**Syntax**  
`p = raylcdf(x,b)`  
`p = raylcdf(x,b,'upper')`

**Description** `p = raylcdf(x,b)` returns the Rayleigh cdf at each value in `x` using the corresponding scale parameter, `b`. `x` and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for `x` or `b` is expanded to a constant array with the same dimensions as the other input.

`p = raylcdf(x,b,'upper')` returns the complement of the Rayleigh cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The Rayleigh cdf is

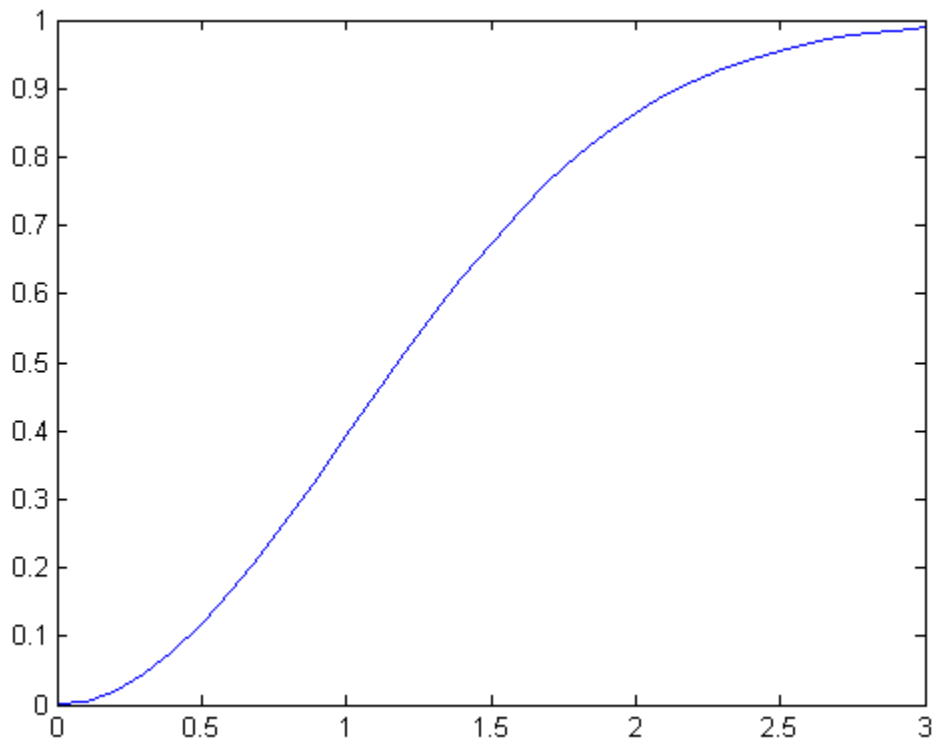
$$y = F(x | b) = \int_0^x \frac{t}{b^2} e^{\left(\frac{-t^2}{2b^2}\right)} dt$$

**Examples**      **Compute Rayleigh Distribution cdf**

```
x = 0:0.1:3;  
p = raylcdf(x,1);  
plot(x,p)
```

# raylcdf

---



## References

[1] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. Hoboken, NJ: Wiley-Interscience, 2000. pp. 134–136.

## See Also

[cdf](#) | [raylpdf](#) | [raylinv](#) | [raylstat](#) | [raylfit](#) | [raylrnd](#)

## How To

- “Rayleigh Distribution” on page B-121

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Rayleigh probability distribution object

**Description** `prob.RayleighDistribution` is an object consisting of parameters, a model description, and sample data for a normal probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Rayleigh')` creates a Rayleigh probability distribution object using the default parameter values.

`pd = makedist('Rayleigh','b',b)` creates a Rayleigh probability distribution object using the specified parameter value.

### Input Arguments

#### **b** - Defining parameter

1 (default) | positive scalar value

Defining parameter for the Rayleigh distribution, specified as a positive scalar value.

#### Data Types

single | double

**Properties** **b**

Defining parameter for the Rayleigh distribution, stored as a positive scalar value.

#### Data Types

single | double

#### DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

# prob.RayleighDistribution

---

## Data Types

char

## InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value.

If IsTruncated equals 0, the distribution is not truncated.

If IsTruncated equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution.

The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the

estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

**Data Types**

single | double

**ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

**Data Types**

char

**ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

**Data Types**

logical

**ParameterNames**

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

**Data Types**

char

**ParameterValues**

Values of distribution parameters, stored as a vector. This property is read-only.

**Data Types**

single | double

**Truncation**

# prob.RayleighDistribution

---

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

<code>cdf</code>	Cumulative distribution function of probability distribution object
<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
<code>iqr</code>	Interquartile range of probability distribution object
<code>median</code>	Median of probability distribution object
<code>pdf</code>	Probability density function of probability distribution object
<code>random</code>	Generate random numbers from probability distribution object
<code>truncate</code>	Truncate probability distribution object
<code>mean</code>	Mean of probability distribution object
<code>negloglik</code>	Negative loglikelihood of probability distribution object
<code>paramci</code>	Confidence intervals for probability distribution parameters

proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions **Rayleigh Distribution**

The Rayleigh distribution is a special case of the Weibull distribution. It is often used in communication theory to model scattered signals that reach a receiver by multiple paths.

The Rayleigh distribution uses the following parameter.

Parameter	Description	Support
b	Defining parameter	$b > 0$

The probability density function (pdf) is

$$f(x|b) = \frac{x}{b^2} \exp\left\{\frac{-x^2}{2b^2}\right\} ; \quad x \geq 0.$$

## Examples **Create a Rayleigh Distribution Object Using Default Parameters**

Create a Rayleigh distribution object using the default parameter values.

```
pd = makedist('Rayleigh')
```

```
pd =
```

```
RayleighDistribution
```

# prob.RayleighDistribution

---

```
Rayleigh distribution
B = 1
```

## Create a Rayleigh Distribution Object Using Specified Parameters

Create a Rayleigh distribution object by specifying the parameter values.

```
pd = makedist('Rayleigh','b',3)
```

```
pd =
```

```
RayleighDistribution
```

```
Rayleigh distribution
B = 3
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
3.7599
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “Rayleigh Distribution” on page B-121
- Class Attributes
- Property Attributes



---

<b>Purpose</b>	Rayleigh parameter estimates
<b>Syntax</b>	<code>raylfit(data,alpha)</code> <code>[phat,pci] = raylfit(data,alpha)</code>
<b>Description</b>	<code>raylfit(data,alpha)</code> returns the maximum likelihood estimates of the parameter of the Rayleigh distribution given the data in the vector <code>data</code> . <code>[phat,pci] = raylfit(data,alpha)</code> returns the maximum likelihood estimate and $100(1 - \alpha)\%$ confidence interval given the data. The default value of the optional parameter <code>alpha</code> is 0.05, corresponding to 95% confidence intervals.
<b>See Also</b>	<code>mle</code>   <code>raylpdf</code>   <code>raylcdf</code>   <code>raylinv</code>   <code>raylstat</code>   <code>raylrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Rayleigh Distribution” on page B-121</li></ul>

# raylinv

---

**Purpose** Rayleigh inverse cumulative distribution function

**Syntax** `X = raylinv(P,B)`

**Description** `X = raylinv(P,B)` returns the inverse of the Rayleigh cumulative distribution function using the corresponding scale parameter, **B** at the corresponding probabilities in **P**. **P** and **B** can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input for **P** or **B** is expanded to a constant array with the same dimensions as the other input.

**Examples**

```
x = raylinv(0.9,1)
x =
    2.1460
```

**See Also** `raylcdf` | `raylpdf` | `raylrnd` | `raylstat`

**How To**

- `icdf`
- “Rayleigh Distribution” on page B-121

**Purpose** Rayleigh probability density function

**Syntax** `Y = raylpdf(X,B)`

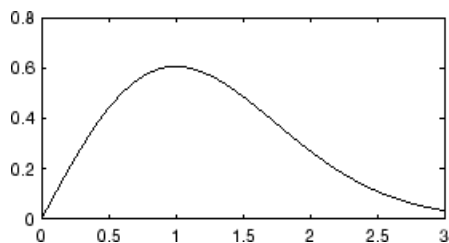
**Description** `Y = raylpdf(X,B)` computes the Rayleigh pdf at each of the values in `X` using the corresponding scale parameter, `B`. `X` and `B` can be vectors, matrices, or multidimensional arrays that all have the same size, which is also the size of `Y`. A scalar input for `X` or `B` is expanded to a constant array with the same dimensions as the other input.

The Rayleigh pdf is

$$y = f(x | b) = \frac{x}{b^2} e^{\left(\frac{-x^2}{2b^2}\right)}$$

### Examples

```
x = 0:0.1:3;  
p = raylpdf(x,1);  
plot(x,p)
```



**See Also** `pdf` | `raylcdf` | `raylinv` | `raylstat` | `raylfit` | `raylrnd`

**How To**

- “Rayleigh Distribution” on page B-121

# raylrnd

---

**Purpose** Rayleigh random numbers

**Syntax**  
`R = raylrnd(B)`  
`R = raylrnd(B,v)`  
`R = raylrnd(B,m,n)`

**Description** `R = raylrnd(B)` returns a matrix of random numbers chosen from the Rayleigh distribution with scale parameter, `B`. `B` can be a vector, a matrix, or a multidimensional array. The size of `R` is the size of `B`.

`R = raylrnd(B,v)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where `v` is a row vector. If `v` is a 1-by-2 vector, `R` is a matrix with `v(1)` rows and `v(2)` columns. If `v` is 1-by-`n`, `R` is an `n`-dimensional array.

`R = raylrnd(B,m,n)` returns a matrix of random numbers chosen from the Rayleigh distribution with parameter `B`, where scalars `m` and `n` are the row and column dimensions of `R`.

**Examples**

```
r = raylrnd(1:5)
r =
    1.7986    0.8795    3.3473    8.9159    3.5182
```

**See Also** `random` | `raylpdf` | `raylcdf` | `raylinv` | `raylstat` | `raylfit`

**How To**

- “Rayleigh Distribution” on page B-121

**Purpose** Rayleigh mean and variance

**Syntax** [M,V] = raylstat(B)

**Description** [M,V] = raylstat(B) returns the mean of and variance for the Rayleigh distribution with scale parameter B.

The mean of the Rayleigh distribution with parameter  $b$  is  $b\sqrt{\pi/2}$  and the variance is

$$\frac{4-\pi}{2}b^2$$

**Examples** [mn,v] = raylstat(1)

```
mn =  
    1.2533  
v =  
    0.4292
```

**See Also** raylpdf | raylcdf | raylinv | raylfit | raylrnd

**How To** • “Rayleigh Distribution” on page B-121

# rcoplot

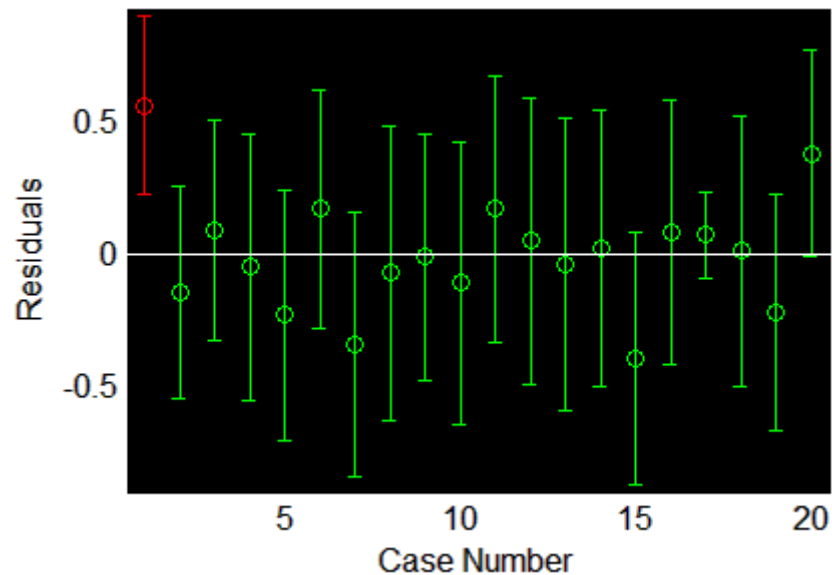
**Purpose** Residual case order plot

**Syntax** `rcoplot(r,rint)`

**Description** `rcoplot(r,rint)` displays an errorbar plot of the confidence intervals on the residuals from a regression. The residuals appear in the plot in case order. Inputs `r` and `rint` are outputs from the `regress` function.

**Examples** The following plots residuals and prediction intervals from a regression of a linearly additive model to the data in `moore.mat`:

```
load moore
X = [ones(size(moore,1),1) moore(:,1:5)];
y = moore(:,6);
alpha = 0.05;
[betahat,Ibeta,res,Ires,stats] = regress(y,X,alpha);
rcoplot(res,Ires)
```



The interval around the first residual, shown in red, does not contain zero. This indicates that the residual is larger than expected in 95% of new observations, and suggests the data point is an outlier.

**See Also**

`regress`

# refcurve

---

**Purpose** Add reference curve to plot

**Syntax** `refcurve(p)`  
`refcurve`  
`hcurve = refcurve(...)`

**Description** `refcurve(p)` adds a polynomial reference curve with coefficients `p` to the current axes. If `p` is a vector with `n+1` elements, the curve is:

$$y = p(1)*x^n + p(2)*x^{(n-1)} + \dots + p(n)*x + p(n+1)$$

`refcurve` with no input arguments adds a line along the  $x$  axis.

`hcurve = refcurve(...)` returns the handle `hcurve` to the curve.

## **Examples**      **Add Population and Fitted Mean Functions**

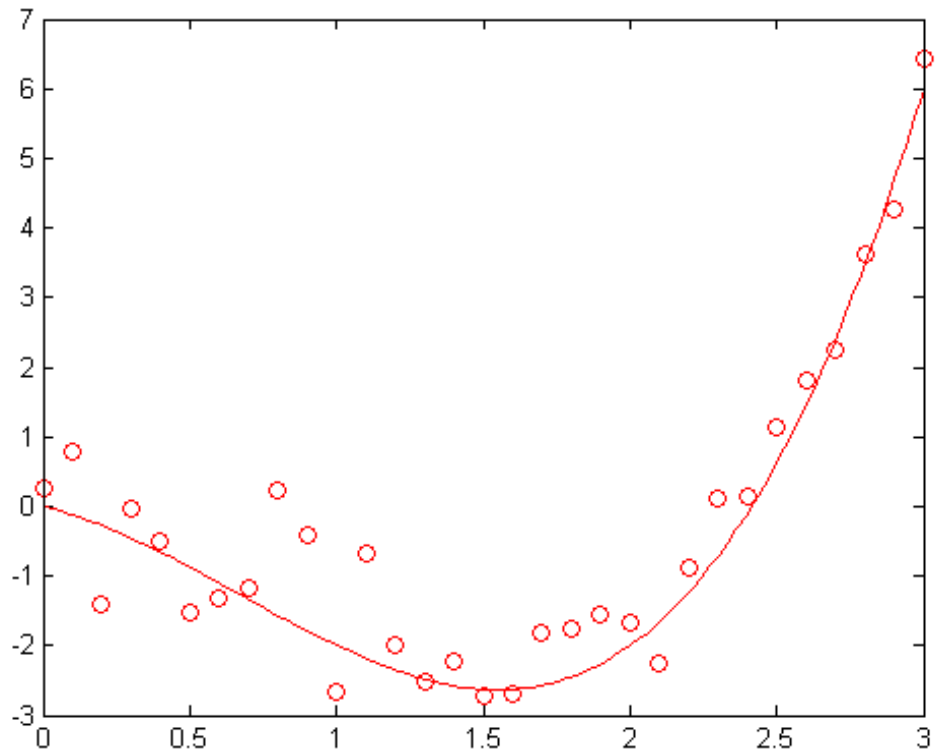
Generate data with a polynomial trend.

```
p = [1 -2 -1 0];  
t = 0:0.1:3;  
rng('default') % for reproducibility  
y = polyval(p,t) + 0.5*randn(size(t));
```

Plot data and add the population mean function using `refcurve`.

```
plot(t,y,'ro')  
h = refcurve(p);  
set(h,'Color','r')
```

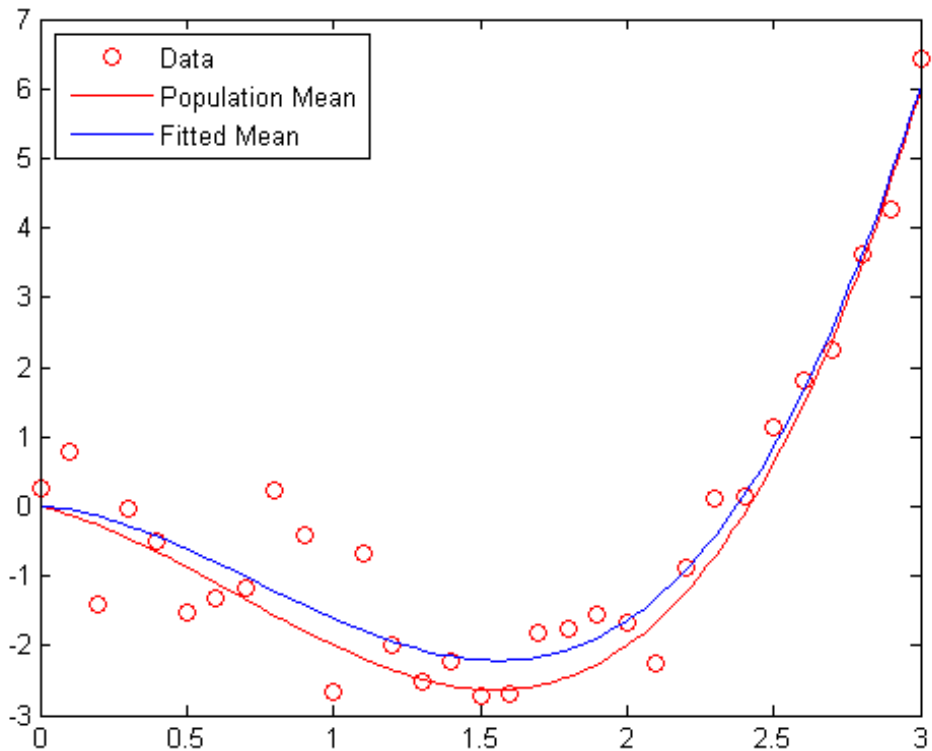




Also add the fitted mean function.

```
q = polyfit(t,y,3);  
refcurve(q)  
legend('Data','Population Mean','Fitted Mean',...  
      'Location','NW')
```

# refcurve



## Plot Trajectories of a Batted Baseball Using refcurve

Introduce the relevant physical constants.

```
M = 0.145;      % Mass (kg)
R = 0.0366;     % Radius (m)
A = pi*R^2;     % Area (m^2)
rho = 1.2;      % Density of air (kg/m^3)
C = 0.5;        % Drag coefficient
D = rho*C*A/2; % Drag proportional to the square of the speed
```

```

g = 9.8;           % Acceleration due to gravity (m/s^2)

Simulate the trajectory with drag proportional to the square of the
speed, assuming constant acceleration in each time interval.

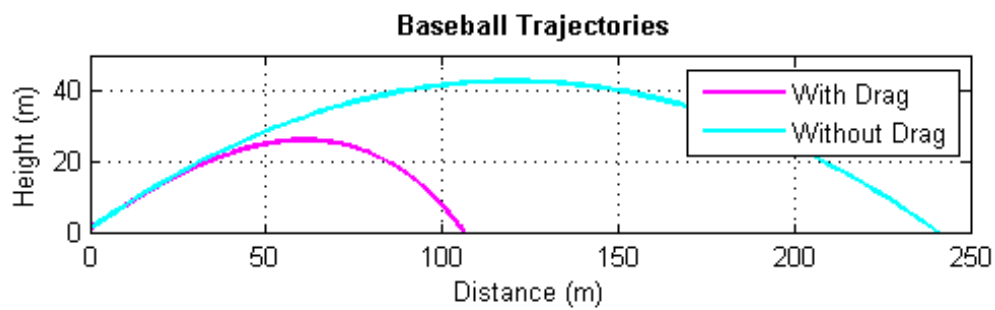
dt = 1e-2;        % Simulation time interval (s)
r0 = [0 1];       % Initial position (m)
s0 = 50;          % Initial speed (m/s)
alpha0 = 35;      % Initial angle (deg)
v0 = s0*[cosd(alpha0) sind(alpha0)]; % Initial velocity (m/s)

r = r0;
v = v0;
trajectory = r0;
while r(2) > 0
    a = [0 -g] - (D/M)*norm(v)*v;
    v = v + a*dt;
    r = r + v*dt + (1/2)*a*(dt^2);
    trajectory = [trajectory;r];
end

Plot trajectory and use refcurve to add the drag-free parabolic trajectory
(found analytically) to the plot of trajectory.

plot(trajectory(:,1),trajectory(:,2),'m','LineWidth',2)
xlim([0,250])
h = refcurve([-g/(2*v0(1)^2),...
    (g*r0(1)/v0(1)^2) + (v0(2)/v0(1)),...
    (-g*r0(1)^2/(2*v0(1)^2)) - (v0(2)*r0(1)/v0(1)) + r0(2)]);
set(h,'Color','c','LineWidth',2)
axis equal
ylim([0,50])
grid on
xlabel('Distance (m)')
ylabel('Height (m)')
title('{\bf Baseball Trajectories}')
legend('With Drag','Without Drag')

```



## See Also

`refline` | `lsline` | `gline` | `polyfit`

**Purpose**

Add reference line to plot

**Syntax**

```
refline(m,b)
refline(coeffs)
refline
hline = refline(...)
```

**Description**

`refline(m,b)` adds a reference line with slope `m` and intercept `b` to the current axes.

`refline(coeffs)`, where `coeffs` is a two-element coefficient vector, adds the line

$$y = \text{coeffs}(1)*x + \text{coeffs}(2)$$

to the figure.

`refline` with no input arguments is equivalent to `lsline`.

`hline = refline(...)` returns the handle `hline` to the line.

**Examples****Add a Reference Line at the Mean**

Generate sample data for independent variable `x` and a dependent variable `y`.

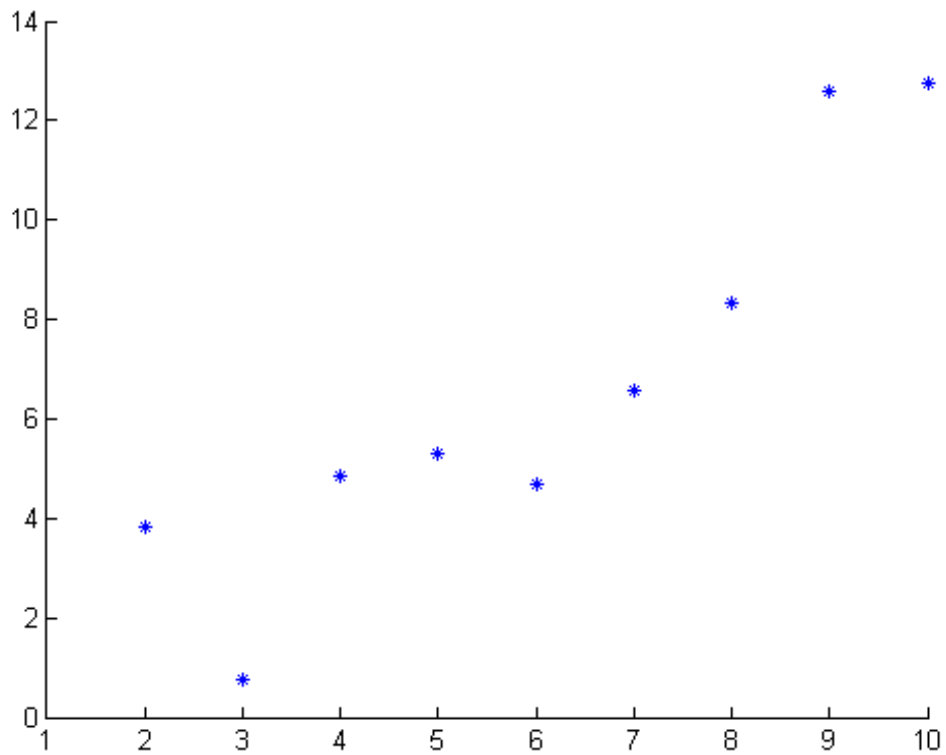
```
x = 1:10;
y = x + randn(1,10);
```

Create a scatter plot of `x` and `y`.

```
scatter(x,y,25,'b','*')
```

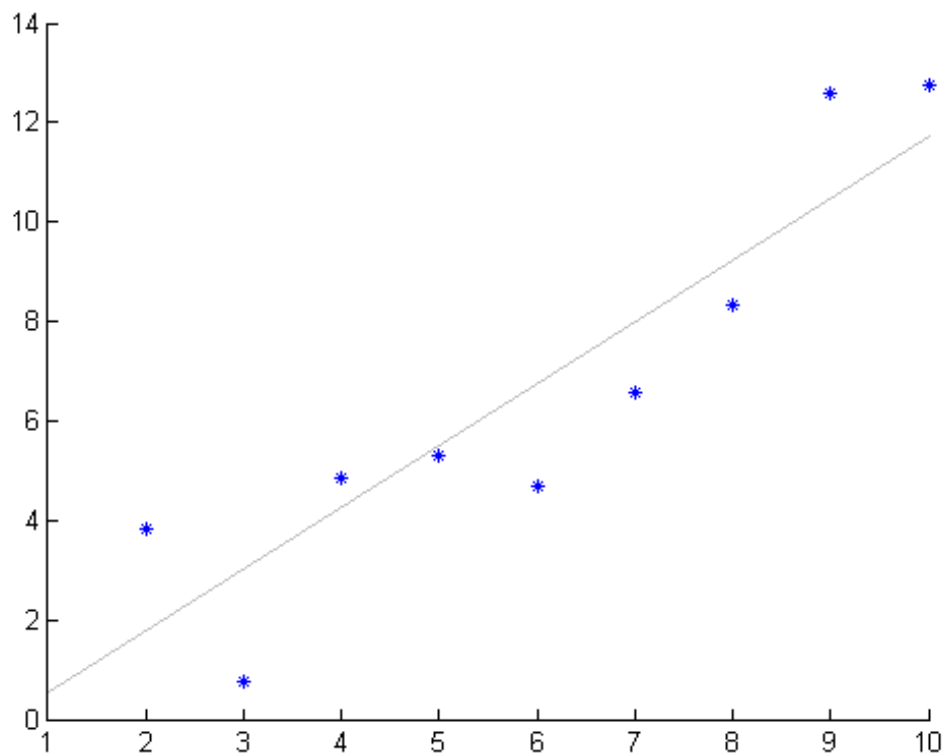
# refline

---



Superimpose a least-squares line on the scatter plot.

```
lsline
```

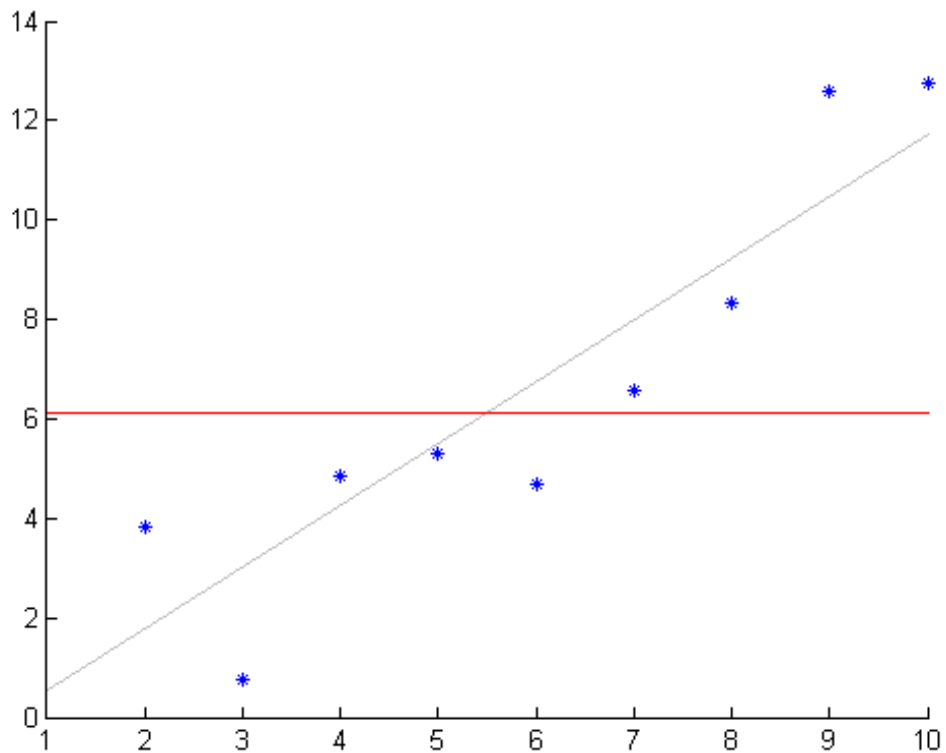


Add a reference line at the mean of the scatter and its least-squares line.

```
mu = mean(y);  
hline = refline([0 mu]);  
set(hline, 'Color', 'r')
```

# refline

---



The red line shows the reference line at the mean of data.

## See Also

`refcurve` | `lsline` | `gline`



**Purpose**

Multiple linear regression

**Syntax**

```
b = regress(y,X)
[b,bint] = regress(y,X)
[b,bint,r] = regress(y,X)
[b,bint,r,rint] = regress(y,X)
[b,bint,r,rint,stats] = regress(y,X)
[...] = regress(y,X,alpha)
```

**Description**

`b = regress(y,X)` returns a  $p$ -by-1 vector `b` of coefficient estimates for a multilinear regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses.

`regress` treats NaNs in `X` or `y` as missing values, and ignores them.

If the columns of `X` are linearly dependent, `regress` obtains a basic solution by setting the maximum number of elements of `b` to zero.

`[b,bint] = regress(y,X)` returns a  $p$ -by-2 matrix `bint` of 95% confidence intervals for the coefficient estimates. The first column of `bint` contains lower confidence bounds for each of the  $p$  coefficient estimates; the second column contains upper confidence bounds.

If the columns of `X` are linearly dependent, `regress` returns zeros in elements of `bint` corresponding to the zero elements of `b`.

`[b,bint,r] = regress(y,X)` returns an  $n$ -by-1 vector `r` of residuals.

`[b,bint,r,rint] = regress(y,X)` returns an  $n$ -by-2 matrix `rint` of intervals that can be used to diagnose outliers. If the interval `rint(i,:)` for observation `i` does not contain zero, the corresponding residual is larger than expected in 95% of new observations, suggesting an outlier.

In a linear model, observed values of `y` are random variables, and so are their residuals. Residuals have normal distributions with zero mean but with different variances at different values of the predictors. To put residuals on a comparable scale, they are “Studentized,” that is, they are divided by an estimate of their standard deviation that is independent of their value. Studentized residuals have  $t$  distributions with known

# regress

---

degrees of freedom. The intervals returned in `rint` are shifts of the 95% confidence intervals of these  $t$  distributions, centered at the residuals.

`[b,bint,r,rint,stats] = regress(y,X)` returns a 1-by-4 vector `stats` that contains, in order, the  $R^2$  statistic, the  $F$  statistic and its  $p$  value, and an estimate of the error variance.

---

**Note** When computing statistics,  $X$  should include a column of 1s so that the model contains a constant term. The  $F$  statistic and its  $p$  value are computed under this assumption, and they are not correct for models without a constant.

The  $F$  statistic is the test statistic of the F-test on the regression model, for a significant linear regression relationship between the response variable and the predictor variables.

The  $R^2$  statistic can be negative for models without a constant, indicating that the model is not appropriate for the data.

---

`[...] = regress(y,X,alpha)` uses a  $100*(1-\text{alpha})\%$  confidence level to compute `bint` and `rint`.

## Examples

### Estimate Multiple Linear Regression Coefficients

This example shows how to estimate the coefficients of a multiple linear regression.

Load the sample data. Identify weight and horsepower as predictors, and mileage as the response.

```
load carsmall
x1 = Weight;
x2 = Horsepower;    % Contains NaN data
y = MPG;
```

Compute the regression coefficients for a linear model with an interaction term.

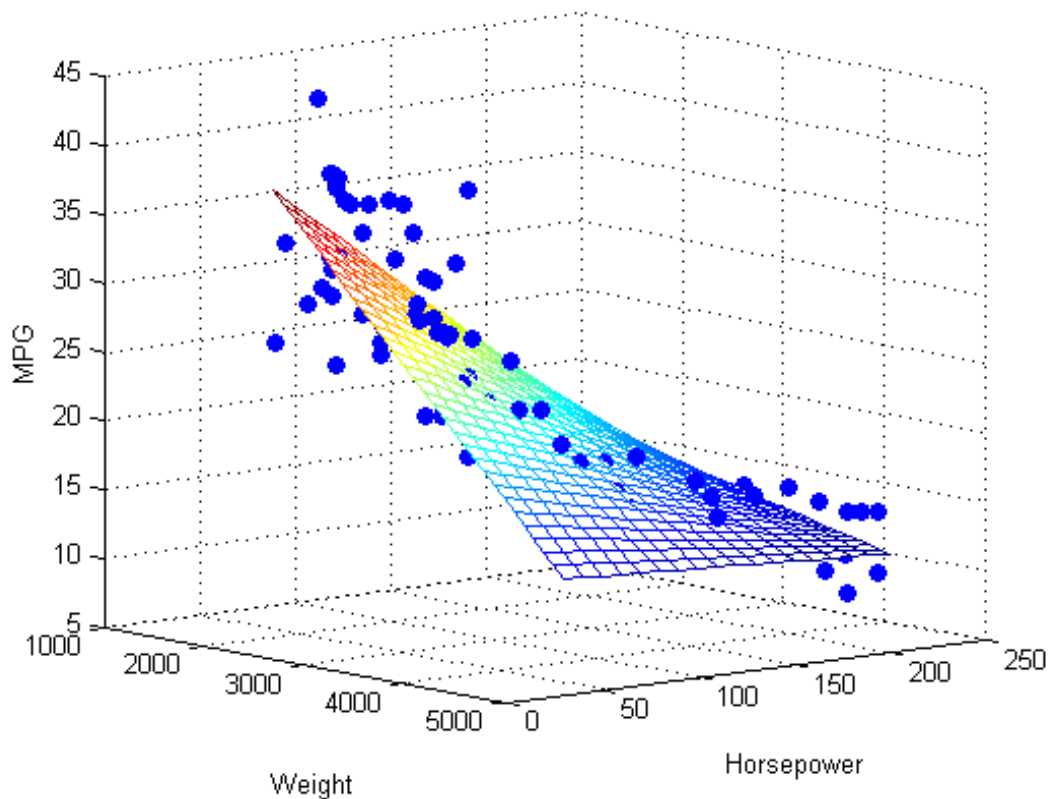
```
X = [ones(size(x1)) x1 x2 x1.*x2];  
b = regress(y,X)    % Removes NaN data
```

```
b =
```

```
    60.7104  
   -0.0102  
   -0.1882  
    0.0000
```

Plot the data and the model.

```
scatter3(x1,x2,y,'filled')  
hold on  
x1fit = min(x1):100:max(x1);  
x2fit = min(x2):10:max(x2);  
[X1FIT,X2FIT] = meshgrid(x1fit,x2fit);  
YFIT = b(1) + b(2)*X1FIT + b(3)*X2FIT + b(4)*X1FIT.*X2FIT;  
mesh(X1FIT,X2FIT,YFIT)  
xlabel('Weight')  
ylabel('Horsepower')  
zlabel('MPG')  
view(50,10)
```



## References

[1] Chatterjee, S., and A. S. Hadi. “Influential Observations, High Leverage Points, and Outliers in Linear Regression.” *Statistical Science*. Vol. 1, 1986, pp. 379–416.

## See Also

`LinearModel` | `fitlm` | `stepwiselm` | `mvregress` | `rcoplot`

## Related Examples

- “Interpret Linear Regression Results” on page 9-64
- “Linear Regression Workflow” on page 9-44

<b>Superclasses</b>	RegressionEnsemble
<b>Purpose</b>	Regression ensemble grown by resampling
<b>Description</b>	RegressionBaggedEnsemble combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners.
<b>Construction</b>	<code>ens = fitensemble(X,Y,'bag',nlearn,learners,'type','regression')</code> creates a bagged regression ensemble. For more information on the syntax, see the <code>fitensemble</code> function reference page.
<b>Properties</b>	<p><b>CategoricalPredictors</b></p> <p>List of categorical predictors. <code>CategoricalPredictors</code> is a numeric vector with indices from 1 to <code>p</code>, where <code>p</code> is the number of columns of <code>X</code>.</p> <p><b>CombineWeights</b></p> <p>A string describing how the ensemble combines learner predictions.</p> <p><b>FitInfo</b></p> <p>A numeric array of fit information. The <code>FitInfoDescription</code> property describes the content of this array.</p> <p><b>FitInfoDescription</b></p> <p>String describing the meaning of the <code>FitInfo</code> array.</p> <p><b>FResample</b></p> <p>A numeric scalar between 0 and 1. <code>FResample</code> is the fraction of training data <code>fitensemble</code> resampled at random for every weak learner when constructing the ensemble.</p> <p><b>LearnerNames</b></p>

# RegressionBaggedEnsemble

---

Cell array of strings with names of the weak learners in the ensemble. The name of each learner appears just once. For example, if you have an ensemble of 100 trees, `LearnerNames` is `{'Tree'}`.

## **Method**

A string with the name of the algorithm `fitensemble` used for training the ensemble.

## **ModelParams**

Parameters used in training `ens`.

## **NObservations**

Numeric scalar containing the number of observations in the training data.

## **NTrained**

Number of trained learners in the ensemble, a positive scalar.

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

## **ReasonForTermination**

A string describing the reason `fitensemble` stopped adding weak learners to the ensemble.

## **Regularization**

A structure containing the result of the `regularize` method. Use `Regularization` with `shrink` to lower resubstitution error and shrink the ensemble.

## **Replace**

Boolean flag indicating if training data for weak learners in this ensemble were sampled with replacement. `Replace` is `true` for sampling with replacement, `false` otherwise.

## ResponseName

A string with the name of the response variable  $Y$ .

## ResponseTransform

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ .

Add or change a ResponseTransform function by dot addressing:

```
ens.ResponseTransform = @function
```

## Trained

The trained learners, a cell array of compact regression models.

## TrainedWeights

A numeric vector of weights the ensemble assigns to its learners. The ensemble computes predicted response by aggregating weighted predictions from its learners.

## UseObsForLearner

A logical matrix of size  $N$ -by- $N_{\text{Trained}}$ , where  $N$  is the number of rows (observations) in the training data  $X$ , and  $N_{\text{Trained}}$  is the number of trained weak learners. `UseObsForLearner(I,J)` is true if observation  $I$  was used for training learner  $J$ , and is false otherwise.

## W

The scaled `weights`, a vector with length  $n$ , the number of rows in  $X$ . The sum of the elements of  $W$  is 1.

## X

The matrix of predictor values that trained the ensemble. Each column of  $X$  represents one variable, and each row represents one observation.

## Y

# RegressionBaggedEnsemble

---

The numeric column vector with the same number of rows as X that trained the ensemble. Each entry in Y is the response to the data in the corresponding row of X.

## Methods

<code>oobLoss</code>	Out-of-bag regression error
<code>oobPredict</code>	Predict out-of-bag response of ensemble

## Inherited Methods

<code>compact</code>	Create compact regression ensemble
<code>crossval</code>	Cross validate ensemble
<code>cvshrink</code>	Cross validate shrinking (pruning) ensemble
<code>regularize</code>	Find weights to minimize resubstitution error plus penalty term
<code>resubLoss</code>	Regression error by resubstitution
<code>resubPredict</code>	Predict response of ensemble by resubstitution
<code>resume</code>	Resume training ensemble
<code>shrink</code>	Prune ensemble
<code>loss</code>	Regression error
<code>predict</code>	Predict response of ensemble
<code>predictorImportance</code>	Estimates of predictor importance
<code>removeLearners</code>	Remove members of compact regression ensemble



## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Create a bagged regression ensemble to predict the mileage of cars in the `carsmall` data set based on their engine displacement, horsepower, and weight:

```
load carsmall
X = [Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'bag',100,'Tree',...
    'type','regression')

ens =

classreg.learning.regr.RegressionBaggedEnsemble:
    PredictorNames: {'x1' 'x2' 'x3'}
    CategoricalPredictors: []
    ResponseName: 'Y'
    ResponseTransform: 'none'
    NObservations: 94
    NTrained: 100
    Method: 'Bag'
    LearnerNames: {'Tree'}
    ReasonForTermination: [1x77 char]
    FitInfo: []
    FitInfoDescription: 'None'
    Regularization: []
    FResample: 1
    Replace: 1
    UseObsForLearner: [94x100 logical]
```

Predict the mileage of a car whose characteristics are the average of those of the first 10 cars:

```
car10 = mean(X(1:10,:));
predict(ens,car10)
```

# RegressionBaggedEnsemble

---

```
ans =  
    14.6569
```

**See Also**      RegressionEnsemble | fitensemble

<b>Superclasses</b>	CompactRegressionEnsemble
<b>Purpose</b>	Ensemble regression
<b>Description</b>	RegressionEnsemble combines a set of trained weak learner models and data on which these learners were trained. It can predict ensemble response for new data by aggregating predictions from its weak learners.
<b>Construction</b>	<p><code>ens = fitensemble(X,Y,method,nlearn,learners)</code> returns an ensemble model that can predict responses to data. The ensemble consists of models listed in <code>learners</code>. For more information on the syntax, see the <code>fitensemble</code> function reference page.</p> <p><code>ens = fitensemble(X,Y,method,nlearn,learners,Name,Value)</code> returns an ensemble model with additional options specified by one or more <code>Name,Value</code> pair arguments. For more information on the syntax, see the <code>fitensemble</code> function reference page.</p>
<b>Properties</b>	<p><b>CategoricalPredictors</b></p> <p>List of categorical predictors. <code>CategoricalPredictors</code> is a numeric vector with indices from 1 to <code>p</code>, where <code>p</code> is the number of columns of <code>X</code>.</p> <p><b>CombineWeights</b></p> <p>A string describing how the ensemble combines learner predictions.</p> <p><b>FitInfo</b></p> <p>A numeric array of fit information. The <code>FitInfoDescription</code> property describes the content of this array.</p> <p><b>FitInfoDescription</b></p> <p>String describing the meaning of the <code>FitInfo</code> array.</p> <p><b>LearnerNames</b></p>

# RegressionEnsemble

---

Cell array of strings with names of the weak learners in the ensemble. The name of each learner appears just once. For example, if you have an ensemble of 100 trees, `LearnerNames` is `{'Tree'}`.

## **Method**

A string with the name of the algorithm `fitensemble` used for training the ensemble.

## **ModelParams**

Parameters used in training `ens`.

## **NObservations**

Numeric scalar containing the number of observations in the training data.

## **NTrained**

Number of trained learners in the ensemble, a positive scalar.

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

## **ReasonForTermination**

A string describing the reason `fitensemble` stopped adding weak learners to the ensemble.

## **Regularization**

A structure containing the result of the `regularize` method. Use Regularization with `shrink` to lower resubstitution error and shrink the ensemble.

## **ResponseName**

A string with the name of the response variable `Y`.

## **ResponseTransform**

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ .

Add or change a ResponseTransform function by dot addressing:

```
ens.ResponseTransform = @function
```

## Trained

The trained learners, a cell array of compact regression models.

## TrainedWeights

A numeric vector of weights the ensemble assigns to its learners. The ensemble computes predicted response by aggregating weighted predictions from its learners.

## W

The scaled **weights**, a vector with length  $n$ , the number of rows in  $X$ . The sum of the elements of  $W$  is 1.

## X

The matrix of predictor values that trained the ensemble. Each column of  $X$  represents one variable, and each row represents one observation.

## Y

The numeric column vector with the same number of rows as  $X$  that trained the ensemble. Each entry in  $Y$  is the response to the data in the corresponding row of  $X$ .

## Methods

compact	Create compact regression ensemble
crossval	Cross validate ensemble
cvshrink	Cross validate shrinking (pruning) ensemble

# RegressionEnsemble

---

regularize	Find weights to minimize resubstitution error plus penalty term
resubLoss	Regression error by resubstitution
resubPredict	Predict response of ensemble by resubstitution
resume	Resume training ensemble
shrink	Prune ensemble

## Inherited Methods

loss	Regression error
predict	Predict response of ensemble
predictorImportance	Estimates of predictor importance
removeLearners	Remove members of compact regression ensemble

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Create a boosted regression ensemble to predict the mileage of cars in the carsmall data set based on their weights and numbers of cylinders:

```
load carsmall
learner = RegressionTree.template('MinParent',20);
ens = fitensemble([Weight, Cylinders],MPG,...
    'LSBoost',100,learner,'PredictorNames',{'W','C'},...
    'categoricalpredictors',2)

ens =
classreg.learning.regr.RegistrationEnsemble:
    PredictorNames: {'W' 'C'}
    CategoricalPredictors: 2
```

```
    ResponseName: 'Response'  
    ResponseTransform: 'none'  
    NObservations: 94  
    NTrained: 100  
    Method: 'LSBoost'  
    LearnerNames: {'Tree'}  
    ReasonForTermination: [1x77 char]  
    FitInfo: [100x1 double]  
    FitInfoDescription: [2x83 char]  
    Regularization: []
```

Predict the mileage of 4,000-pound cars with 4, 6, and 8 cylinders:

```
mileage4K = predict(ens,[4000 4; 4000 6; 4000 8])
```

```
mileage4K =  
    20.0294  
    19.4206  
    15.5000
```

## See Also

[ClassificationEnsemble](#) | [fitensemble](#) |  
[CompactRegressionEnsemble](#)

# RegressionPartitionedEnsemble

---

**Superclasses** RegressionPartitionedModel

**Purpose** Cross-validated regression ensemble

**Description** RegressionPartitionedEnsemble is a set of regression ensembles trained on cross-validated folds. Estimate the quality of classification by cross validation using one or more “kfold” methods: `kfoldfun`, `kfoldLoss`, or `kfoldPredict`. Every “kfold” method uses models trained on in-fold observations to predict response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, every training fold contains roughly 4/5 of the data and every test fold contains roughly 1/5 of the data. The first model stored in `Trained{1}` was trained on  $X$  and  $Y$  with the first 1/5 excluded, the second model stored in `Trained{2}` was trained on  $X$  and  $Y$  with the second 1/5 excluded, and so on. When you call `kfoldPredict`, it computes predictions for the first 1/5 of the data using the first model, for the second 1/5 of data using the second model and so on. In short, response for every observation is computed by `kfoldPredict` using the model trained without this observation.

**Construction** `cvens = crossval(ens)` creates a cross-validated ensemble from `ens`, a regression ensemble. For syntax details, see the `crossval` method reference page.

`cvens = fitensemble(X,Y,method,nlearn,learners,name,value)` creates a cross-validated ensemble when `name` is one of 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'. For syntax details, see the `fitensemble` function reference page.

## Input Arguments

**ens**

A regression ensemble constructed with `fitensemble`.



## Properties

### **CategoricalPredictors**

List of categorical predictors. `CategoricalPredictors` is a numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .

### **CrossValidatedModel**

Name of the cross-validated model, a string.

### **Kfold**

Number of folds used in a cross-validated tree, a positive integer.

### **ModelParams**

Object holding parameters of tree.

### **NObservations**

Numeric scalar containing the number of observations in the training data.

### **NTrainedPerFold**

Vector of `Kfold` elements. Each entry contains the number of trained learners in this cross-validation fold.

### **Partition**

The partition of class `cvpartition` used in creating the cross-validated ensemble.

### **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in  $X$ .

### **ResponseName**

Name of the response variable  $Y$ , a string.

### **ResponseTransform**

Function handle for transforming scores, or string representing a built-in transformation function. 'none' means no transformation; equivalently, 'none' means  $@(x)x$ .

# RegressionPartitionedEnsemble

---

Add or change a ResponseTransform function by dot addressing:

```
ens.ResponseTransform = @function
```

## Trainable

Cell array of ensembles trained on cross-validation folds. Every ensemble is full, meaning it contains its training data and weights.

## Trained

Cell array of compact ensembles trained on cross-validation folds.

## W

The scaled weights, a vector with length  $n$ , the number of rows in  $X$ .

## X

A matrix of predictor values. Each column of  $X$  represents one variable, and each row represents one observation.

## Y

A numeric column vector with the same number of rows as  $X$ . Each entry in  $Y$  is the response to the data in the corresponding row of  $X$ .

## Methods

kfoldLoss	Cross-validation loss of partitioned regression ensemble
resume	Resume training ensemble

## Inherited Methods

kfoldfun	Cross validate function
kfoldLoss	Cross-validation loss of partitioned regression model
kfoldPredict	Predict response for observations not used for training.

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Construct a partitioned regression ensemble, and examine the cross-validation losses for the folds:

```
load carsmall
XX = [Cylinders Displacement Horsepower Weight];
YY = MPG;
rens = fitensemble(XX,YY,'LSBoost',100,'Tree');
cvrens = crossval(rens);
L = kfoldLoss(cvrens,'mode','individual')
```

```
L =
    42.4468
    12.3158
    65.9432
    39.0019
    30.5908
    16.6225
    17.3071
    46.1769
     8.0561
    12.9689
```

## See Also

[ClassificationPartitionedEnsemble](#) | [RegressionPartitionedModel](#) | [RegressionEnsemble](#)

# RegressionPartitionedModel

---

**Purpose** Cross-validated regression model

**Description** `RegressionPartitionedModel` is a set of regression models trained on cross-validated folds. Estimate the quality of regression by cross validation using one or more “kfold” methods: `kfoldPredict`, `kfoldLoss`, and `kfoldfun`. Every “kfold” method uses models trained on in-fold observations to predict response for out-of-fold observations. For example, suppose you cross validate using five folds. In this case, every training fold contains roughly 4/5 of the data and every test fold contains roughly 1/5 of the data. The first model stored in `Trained{1}` was trained on `X` and `Y` with the first 1/5 excluded, the second model stored in `Trained{2}` was trained on `X` and `Y` with the second 1/5 excluded, and so on. When you call `kfoldPredict`, it computes predictions for the first 1/5 of the data using the first model, for the second 1/5 of data using the second model and so on. In short, response for every observation is computed by `kfoldPredict` using the model trained without this observation.

**Construction** `cvmodel = crossval(tree)` creates a cross-validated classification model from a regression tree. For syntax details, see the `crossval` method reference page.

`cvmodel = RegressionTree.fit(X,Y,name,value)` creates a cross-validated model when `name` is one of `'crossval'`, `'kfold'`, `'holdout'`, `'leaveout'`, or `'cvpartition'`. For syntax details, see the `RegressionTree.fit` function reference page.

## Input Arguments

### **tree**

A regression tree constructed with `RegressionTree.fit`.

## Properties

### **CategoricalPredictors**

List of categorical predictors. `CategoricalPredictors` is a numeric vector with indices from 1 to `p`, where `p` is the number of columns of `X`.

**CrossValidatedModel**

Name of the cross-validated model, a string.

**Kfold**

Number of folds used in a cross-validated tree, a positive integer.

**ModelParams**

Object holding parameters of `tree`.

**Partition**

The partition of class `cvpartition` used in the cross-validated model.

**PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in  $X$ .

**ResponseName**

Name of the response variable  $Y$ , a string.

**ResponseTransform**

Function handle for transforming the raw response values (mean squared error). The function handle should accept a matrix of response values and return a matrix of the same size. The default string 'none' means  $@(x)x$ , or no transformation.

Add or change a `ResponseTransform` function by dot addressing:

```
ctree.ResponseTransform = @function
```

**Trained**

The trained learners, a cell array of compact regression models.

**W**

The scaled `weights`, a vector with length  $n$ , the number of rows in  $X$ .

# RegressionPartitionedModel

---

## **X**

A matrix of predictor values. Each column of  $X$  represents one variable, and each row represents one observation.

## **Y**

A numeric column vector with the same number of rows as  $X$ . Each entry in  $Y$  is the response to the data in the corresponding row of  $X$ .

## **Methods**

<code>kfoldfun</code>	Cross validate function
<code>kfoldLoss</code>	Cross-validation loss of partitioned regression model
<code>kfoldPredict</code>	Predict response for observations not used for training.

## **Copy Semantics**

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## **Examples**

Evaluate the cross-validation error of the `carsmall` data using Horsepower and Weight as predictor variables for mileage (MPG):

```
load carsmall
X = [Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
cvtree = crossval(tree);
L = kfoldLoss(cvtree)
```

```
L =
```

```
26.4414
```

## **See Also**

`RegressionPartitionedEnsemble` |  
`ClassificationPartitionedModel`

**Superclasses** CompactRegressionTree

**Purpose** Regression tree

**Description** A decision tree with binary splits for regression. An object of class `RegressionTree` can predict responses for new data with the `predict` method. The object contains the data used for training, so can compute resubstitution predictions.

**Construction** `tree = RegressionTree.fit(X,Y)` returns a regression tree based on the input variables (also known as predictors, features, or attributes) `X` and output (response) `Y`. `tree` is a binary tree where each branching node is split based on the values of a column of `X`.

`tree = RegressionTree.fit(X,Y,Name,Value)` fits a tree with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### X

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

`RegressionTree.fit` considers NaN values in `X` as missing values. `RegressionTree.fit` does not use observations with all missing values for `X` the fit. `RegressionTree.fit` uses observations with some missing values for `X` to find splits on variables for which these observations have valid values.

### Y

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

`RegressionTree.fit` considers NaN values in `Y` to be missing values. `RegressionTree.fit` does not use observations with missing values for `Y` in the fit.

# RegressionTree

---

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

## **CategoricalPredictors**

List of categorical predictors. Pass `CategoricalPredictors` as one of:

- A numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .
- A logical vector of length  $p$ , where a true entry means that the corresponding column of  $X$  is a categorical variable.
- 'all', meaning all predictors are categorical.
- A cell array of strings, where each element in the array is the name of a predictor variable. The names must match entries in the `PredictorNames` property.
- A character matrix, where each row of the matrix is a name of a predictor variable. Pad the names with extra blanks so each row of the character matrix has the same length.

**Default:** []

## **crossval**

Flag to grow a cross-validated decision tree. Possible values are 'on' or 'off'.

If 'on', `RegressionTree.fit` grows a cross-validated decision tree with 10 folds. You can override this cross-validation setting using one of the 'kfold', 'holdout', 'leaveout', or 'cvpartition' name-value pair arguments. Note that you can only use one of these four options ('kfold', 'holdout',



'leaveout', or 'cvpartition') at a time when creating a cross-validated tree.

Alternatively, cross-validate `tree` later using the `crossval` method.

**Default:** 'off'

## **cvpartition**

Partition created with `cvpartition` to use in cross-validated tree.

Note that if you use 'cvpartition', you cannot use any of the 'kfold', 'holdout', or 'leaveout' name-value pair arguments.

## **holdout**

Holdout validation tests the specified fraction of the data, and uses the rest of the data for training. Specify a numeric scalar from 0 to 1.

Note that if you use 'holdout', you cannot use any of the 'cvpartition', 'kfold', or 'leaveout' name-value pair arguments.

## **kfold**

Number of folds to use in a cross-validated tree, a positive integer.

Note that if you use 'kfold', you cannot use any of the 'cvpartition', 'holdout', or 'leaveout' name-value pair arguments.

**Default:** 10

## **leaveout**

Use leave-one-out cross validation by setting to 'on'.

Note that if you use 'leaveout', you cannot use any of the 'cvpartition', 'holdout', or 'kfold' name-value pair arguments.

# RegressionTree

---

## **MergeLeaves**

When 'on', RegressionTree merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', RegressionTree does not merge leaves.

**Default:** 'on'

## **MinLeaf**

Each leaf has at least MinLeaf observations per tree leaf. If you supply both MinParent and MinLeaf, RegressionTree uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** 1

## **MinParent**

Each branch node in the tree has at least MinParent observations. If you supply both MinParent and MinLeaf, RegressionTree uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** 10

## **NVarToSample**

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all'

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in X.

**Default:** {'x1', 'x2', ...}

## **Prune**

When 'on', RegressionTree computes the full tree and the optimal sequence of pruned subtrees. When 'off' RegressionTree computes the full tree without pruning.

**Default:** 'on'

## **PruneCriterion**

String with the pruning criterion, always 'error'.

**Default:** 'error'

## **QEToler**

Defines tolerance on quadratic error per node for regression trees. Splitting nodes stops when quadratic error per node drops below  $QEToler * QED$ , where QED is the quadratic error for the entire data computed before the decision tree is grown.

**Default:** 1e-6

## **ResponseName**

Name of the response variable Y, a string.

**Default:** 'Y'

## **ResponseTransform**

Function handle for transforming the raw response values. The function handle should accept a matrix of response values and return a matrix of the same size. The default string 'none' means  $@(x)x$ , or no transformation.

Add or change a ResponseTransform function by dot addressing:

# RegressionTree

---

`tree.ResponseTransform = @function`

**Default:** 'none'

## SplitCriterion

Criterion for choosing a split, always the string 'MSE', meaning mean squared error.

**Default:** 'MSE'

## Surrogate

String describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', `RegressionTree.fit` finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, `RegressionTree.fit` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `RegressionTree.fit` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## Weights

Vector of observation weights. The length of weights is the number of rows in X.

**Default:** `ones(size(X,1),1)`

## Properties

### CategoricalPredictors

List of categorical predictors, a numeric vector with indices from 1 to  $p$ , where  $p$  is the number of columns of  $X$ .

### CatSplit

An  $n$ -by-2 cell array, where  $n$  is the number of categorical splits in `tree`. Each row in `CatSplit` gives left and right values for a categorical split. For each branch node with categorical split  $j$  based on a categorical predictor variable  $z$ , the left child is chosen if  $z$  is in `CatSplit(j,1)` and the right child is chosen if  $z$  is in `CatSplit(j,2)`. The splits are in the same order as nodes of the tree. Nodes for these splits can be found by running `cuttype` and selecting 'categorical' cuts from top to bottom.

### Children

An  $n$ -by-2 array containing the numbers of the child nodes for each node in `tree`, where  $n$  is the number of nodes. Leaf nodes have child node 0.

### CutCategories

An  $n$ -by-2 cell array of the categories used at branches in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a categorical predictor variable  $x$ , the left child is chosen if  $x$  is among the categories listed in `CutCategories{i,1}`, and the right child is chosen if  $x$  is among those listed in `CutCategories{i,2}`. Both columns of `CutCategories` are empty for branch nodes based on continuous predictors and for leaf nodes.

`CutPoint` contains the cut points for 'continuous' cuts, and `CutCategories` contains the set of categories.

### CutPoint

An  $n$ -element vector of the values used as cut points in `tree`, where  $n$  is the number of nodes. For each branch node  $i$  based on a continuous predictor variable  $x$ , the left child is chosen if  $x < \text{CutPoint}(i)$  and the right child is chosen if  $x \geq \text{CutPoint}(i)$ .

# RegressionTree

---

CutPoint is NaN for branch nodes based on categorical predictors and for leaf nodes.

## CutType

An  $n$ -element cell array indicating the type of cut at each node in `tree`, where  $n$  is the number of nodes. For each node  $i$ , `CutType{i}` is:

- 'continuous' — If the cut is defined in the form  $x < v$  for a variable  $x$  and cut point  $v$ .
- 'categorical' — If the cut is defined by whether a variable  $x$  takes a value in a set of categories.
- '' — If  $i$  is a leaf node.

CutPoint contains the cut points for 'continuous' cuts, and CutCategories contains the set of categories.

## CutVar

An  $n$ -element cell array of the names of the variables used for branching in each node in `tree`, where  $n$  is the number of nodes. These variables are sometimes known as *cut variables*. For leaf nodes, CutVar contains an empty string.

CutPoint contains the cut points for 'continuous' cuts, and CutCategories contains the set of categories.

## IsBranch

An  $n$ -element logical vector `ib` that is true for each branch node and false for each leaf node of `tree`.

## ModelParams

Object holding parameters of `tree`.

## NObservations

Number of observations in the training data, a numeric scalar. `NObservations` can be less than the number of rows of input data `X` when there are missing values in `X` or response `Y`.

## **NodeErr**

An  $n$ -element vector `e` of the errors of the nodes in `tree`, where  $n$  is the number of nodes. `e(i)` is the misclassification probability for node `i`.

## **NodeMean**

An  $n$ -element numeric array with mean values in each node of `tree`, where  $n$  is the number of nodes in the tree. Every element in `NodeMean` is the average of the true `Y` values over all observations in the node.

## **NodeProb**

An  $n$ -element vector `p` of the probabilities of the nodes in `tree`, where  $n$  is the number of nodes. The probability of a node is computed as the proportion of observations from the original data that satisfy the conditions for the node. This proportion is adjusted for any prior probabilities assigned to each class.

## **NodeRisk**

An  $n$ -element vector of the risk of the nodes in the tree, where  $n$  is the number of nodes. The risk for each node is the node error weighted by the node probability.

## **NodeSize**

An  $n$ -element vector `sizes` of the sizes of the nodes in `tree`, where  $n$  is the number of nodes. The size of a node is defined as the number of observations from the data used to create the tree that satisfy the conditions for the node.

## **NumNodes**

The number of nodes  $n$  in `tree`.

## **Parent**

# RegressionTree

---

An  $n$ -element vector `p` containing the number of the parent node for each node in `tree`, where  $n$  is the number of nodes. The parent of the root node is 0.

## **PredictorNames**

A cell array of names for the predictor variables, in the order in which they appear in `X`.

## **PruneAlpha**

Numeric vector with one element per pruning level. If the pruning level ranges from 0 to  $M$ , then `PruneAlpha` has  $M + 1$  elements sorted in ascending order. `PruneAlpha(1)` is for pruning level 0 (no pruning), `PruneAlpha(2)` is for pruning level 1, and so on.

## **PruneList**

An  $n$ -element numeric vector with the pruning levels in each node of `tree`, where  $n$  is the number of nodes. The pruning levels range from 0 (no pruning) to  $M$ , where  $M$  is the distance between the deepest leaf and the root node.

## **ResponseName**

Name of the response variable `Y`, a string.

## **ResponseTransform**

Function handle for transforming the raw response values (mean squared error). The function handle should accept a matrix of response values and return a matrix of the same size. The default string 'none' means  $@(x)x$ , or no transformation.

Add or change a `ResponseTransform` function by dot addressing:

```
tree.ResponseTransform = @function
```

## **SurrCutCategories**

An  $n$ -element cell array of the categories used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutCategories{k}` is a cell array. The



length of `SurrCutCategories{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutCategories{k}` is either an empty string for a continuous surrogate predictor, or is a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split, and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutCategories` contains an empty cell.

## **SurrCutFlip**

An  $n$ -element cell array of the numeric cut assignments used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutFlip{k}` is a numeric vector. The length of `SurrCutFlip{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutFlip{k}` is either zero for a categorical surrogate predictor, or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutFlip` contains an empty array.

## **SurrCutPoint**

An  $n$ -element cell array of the numeric values used for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutPoint{k}` is a numeric vector. The length of

`SurrCutPoint{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrCutPoint{k}` is either NaN for a categorical surrogate predictor, or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and `SurrCutFlip` for this surrogate split is  $+1$ , or if  $Z < C$  and `SurrCutFlip` for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables returned by `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutPoint` contains an empty cell.

## **SurrCutType**

An  $n$ -element cell array indicating types of surrogate splits at each node in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrCutType{k}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutType` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable  $Z$  and cut point  $V$  or 'categorical' if the cut is defined by whether  $Z$  takes a value in a set of categories.

## **SurrCutVar**

An  $n$ -element cell array of the names of the variables used for surrogate splits in each node in `tree`, where  $n$  is the number of nodes in `tree`. Every element of `SurrCutVar` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only

variables with the positive predictive measure are included. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrCutVar` contains an empty cell.

## **SurrVarAssoc**

An  $n$ -element cell array of the predictive measures of association for surrogate splits in `tree`, where  $n$  is the number of nodes in `tree`. For each node  $k$ , `SurrVarAssoc{k}` is a numeric vector. The length of `SurrVarAssoc{k}` is equal to the number of surrogate predictors found at this node. Every element of `SurrVarAssoc{k}` gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is the order of variables in `SurrCutVar`. The optimal-split variable at this node does not appear. For nonbranch (leaf) nodes, `SurrVarAssoc` contains an empty cell.

## **W**

The scaled `weights`, a vector with length  $n$ , the number of rows in `X`.

## **X**

A matrix of predictor values. Each column of `X` represents one variable, and each row represents one observation.

## **Y**

A numeric column vector with the same number of rows as `X`. Each entry in `Y` is the response to the data in the corresponding row of `X`.

## **Methods**

<code>compact</code>	Compact regression tree
<code>crossval</code>	Cross-validated decision tree
<code>cvloss</code>	Regression error by cross validation
<code>fit</code>	Binary decision tree for regression

# RegressionTree

---

prune	Produce sequence of subtrees by pruning
resubLoss	Regression error by resubstitution
resubPredict	Predict resubstitution response of tree
template	Create regression template

## Inherited Methods

loss	Regression error
meanSurrVarAssoc	Mean predictive measure of association for surrogate splits in decision tree
predict	Predict response of regression tree
predictorImportance	Estimates of predictor importance
view	View tree

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

Load the data in `carsmall.mat`, and make a regression tree to predict the mileage of cars based on their weights and numbers of cylinders:

```
load carsmall
tree = RegressionTree.fit([Weight, Cylinders],MPG,...
    'categoricalpredictors',2,'MinParent',20,...
    'PredictorNames',{'W','C'})
```

```
tree =
```

```
RegressionTree
  PredictorNames: {'W' 'C'}
  ResponseName: 'Y'
  ResponseTransform: 'none'
  CategoricalPredictors: 2
  NObservations: 94
```

Properties, Methods

Predict the mileage of 4,000-pound cars with 4, 6, and 8 cylinders:

```
mileage4K = predict(tree,[4000 4; 4000 6; 4000 8])
```

```
mileage4K =
    19.2778
    19.2778
    14.3889
```

## See Also

[ClassificationTree](#) | [RegressionEnsemble](#) | [RegressionTree.fit](#) | [CompactRegressionTree](#) | [predict](#)

## Tutorials

- “Classification Trees and Regression Trees” on page 16-31

# regstats

---

**Purpose** Regression diagnostics

**Syntax**

```
regstats(y,X,model)
stats = regstats(...)
stats = regstats(y,X,model,whichstats)
```

**Description** `regstats(y,X,model)` performs a multilinear regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses.

---

**Note** By default, `regstats` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`.

---

The optional input `model` controls the regression model. By default, `regstats` uses a linear additive model with a constant term. `model` can be any one of the following strings:

- 'linear' — Constant and linear terms (the default)
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

Alternatively, `model` can be a matrix of model terms accepted by the `x2fx` function. See `x2fx` for a description of this matrix and for a description of the order in which terms appear. You can use this matrix to specify other models including ones without a constant term.

With this syntax, the function displays a graphical user interface (GUI) with a list of diagnostic statistics, as shown in the following figure.

**Regstats Export to Workspace** [ - ] [ □ ] [ × ]

<input type="checkbox"/> Q from QR Decomposition	Q
<input type="checkbox"/> R from QR Decomposition	R
<input type="checkbox"/> Coefficients	beta
<input type="checkbox"/> Coefficient Covariance	covb
<input type="checkbox"/> Fitted Values	yhat
<input type="checkbox"/> Residuals	r
<input type="checkbox"/> Mean Square Error	mse
<input type="checkbox"/> R-square Statistic	rsquare
<input type="checkbox"/> Adjusted R-square Statistic	adjrsquare
<input type="checkbox"/> Leverage	leverage
<input type="checkbox"/> Hat Matrix	hatmat
<input type="checkbox"/> Delete-1 Variance	s2_i
<input type="checkbox"/> Delete-1 Coefficients	beta_i
<input type="checkbox"/> Standardized Residuals	standres
<input type="checkbox"/> Studentized Residuals	studres
<input type="checkbox"/> Change in Beta	dfbetas
<input type="checkbox"/> Change in Fitted Value	dffit
<input type="checkbox"/> Scaled Change in Fit	dffits
<input type="checkbox"/> Change in Covariance	cowratio
<input type="checkbox"/> Cook's Distance	cookd
<input type="checkbox"/> t Statistics	tstat
<input type="checkbox"/> F Statistic	fstat
<input type="checkbox"/> DW Statistic	dwstat

**OK**   **Cancel**   **Help**

## regstats

---

When you select check boxes corresponding to the statistics you want to compute and click **OK**, `regstats` returns the selected statistics to the MATLAB workspace. The names of the workspace variables are displayed on the right-hand side of the interface. You can change the name of the workspace variable to any valid MATLAB variable name.

`stats = regstats(...)` creates the structure `stats`, whose fields contain all of the diagnostic statistics for the regression. This syntax does not open the GUI. The fields of `stats` are listed in the following table.

Field	Description
Q	$Q$ from the $QR$ decomposition of the design matrix
R	$R$ from the $QR$ decomposition of the design matrix
beta	Regression coefficients
covb	Covariance of regression coefficients
yhat	Fitted values of the response data
r	Residuals
mse	Mean squared error
rsquare	$R^2$ statistic
adjrsquare	Adjusted $R^2$ statistic
leverage	Leverage
hatmat	Hat matrix
s2_i	Delete-1 variance
beta_i	Delete-1 coefficients
standres	Standardized residuals
studres	Studentized residuals
dfbetas	Scaled change in regression coefficients
dffit	Change in fitted values



Field	Description
dffits	Scaled change in fitted values
covratio	Change in covariance
cookd	Cook's distance
tstat	$t$ statistics and $p$ -values for coefficients
fstat	$F$ statistic and $p$ -value
dwstat	Durbin-Watson statistic and $p$ -value

Note that the fields names of `stats` correspond to the names of the variables returned to the MATLAB workspace when you use the GUI. For example, `stats.beta` corresponds to the variable `beta` that is returned when you select **Coefficients** in the GUI and click **OK**.

`stats = regstats(y,X,model,whichstats)` returns only the statistics that you specify in `whichstats`. `whichstats` can be a single string such as `'leverage'` or a cell array of strings such as `{'leverage' 'standres' 'studres'}`. Set `whichstats` to `'all'` to return all of the statistics.

---

**Note** The  $F$  statistic is computed under the assumption that the model contains a constant term. It is not correct for models without a constant. The  $R^2$  statistic can be negative for models without a constant, which indicates that the model is not appropriate for the data.

---

## Examples

Open the `regstats` GUI using data from `hald.mat`:

```
load hald
regstats(heat,ingredients,'linear');
```

Select **Fitted Values** and **Residuals** in the GUI:



Click **OK** to export the fitted values and residuals to the MATLAB workspace in variables named `yhat` and `r`, respectively.

You can create the same variables using the `stats` output, without opening the GUI:

```
whichstats = {'yhat','r'};  
stats = regstats(heat,ingredients,'linear',whichstats);  
yhat = stats.yhat;  
r = stats.r;
```

## References

- [1] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [2] Chatterjee, S., and A. S. Hadi. “Influential Observations, High Leverage Points, and Outliers in Linear Regression.” *Statistical Science*. Vol. 1, 1986, pp. 379–416.
- [3] Cook, R. D., and S. Weisberg. *Residuals and Influence in Regression*. New York: Chapman & Hall/CRC Press, 1983.
- [4] Goodall, C. R. “Computation Using the QR Decomposition.” *Handbook in Statistics*. Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.

## See Also

LinearModel | fitlm | stepwiselm

## Related Examples

- “Interpret Linear Regression Results” on page 9-64
- “Linear Regression Workflow” on page 9-44

**Purpose** Find weights to minimize resubstitution error plus penalty term

**Syntax**  
`ens1 = regularize(ens)`  
`ens1 = regularize(ens,Name,Value)`

**Description**  
`ens1 = regularize(ens)` finds optimal weights for learners in `ens` by lasso regularization. `regularize` returns a regression ensemble identical to `ens`, but with a populated `Regularization` property.  
`ens1 = regularize(ens,Name,Value)` computes optimal weights with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

**Input Arguments**  
**`ens`**  
A regression ensemble, created by `fitensemble`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### **'lambda'**

Vector of nonnegative regularization parameter values for lasso. For the default setting of `lambda`, `regularize` calculates the smallest value `lambda_max` for which all optimal weights for learners are 0. The default value of `lambda` is a vector including 0 and nine exponentially-spaced numbers from `lambda_max/1000` to `lambda_max`.

**Default:** `[0  
logspace(log10(lambda_max/1000),log10(lambda_max),9)]`

### **'npass'**

# RegressionEnsemble.regularize

---

Maximal number of passes for lasso optimization, a positive integer.

**Default:** 10

## **'reltol'**

Relative tolerance on the regularized loss for lasso, a numeric positive scalar.

**Default:** 1e-3

## **'verbose'**

Verbosity level, either 0 or 1. When set to 1, `regularize` displays more information as it runs.

**Default:** 0

## **Output Arguments**

### **ens1**

A regression ensemble. Usually you set `ens1` to the same name as `ens`.

## **Definitions**

### **Lasso**

The lasso algorithm finds an optimal set of learner weights  $\alpha_t$  that minimize

$$\sum_{n=1}^N w_n g \left( \left( \sum_{t=1}^T \alpha_t h_t(x_n) \right), y_n \right) + \lambda \sum_{t=1}^T |\alpha_t|.$$

Here

- $\lambda \geq 0$  is a parameter you provide, called the lasso parameter.
- $h_t$  is a weak learner in the ensemble trained on  $N$  observations with predictors  $x_n$ , responses  $y_n$ , and weights  $w_n$ .

- $g(f,y) = (f - y)^2$  is the squared error.

## Examples

Regularize an ensemble of bagged trees:

```
X = rand(2000,20);
Y = repmat(-1,2000,1);
Y(sum(X(:,1:5),2)>2.5) = 1;
bag = fitensemble(X,Y,'Bag',300,'Tree',...
    'type','regression');
bag = regularize(bag,'lambda',[0.001 0.1],'verbose',1);
```

regularize reports on its progress.

To see the resulting regularization structure:

```
bag.Regularization
ans =
    Method: 'Lasso'
    TrainedWeights: [300x2 double]
    Lambda: [1.0000e-003 0.1000]
    ResubstitutionMSE: [0.0616 0.0812]
    CombineWeights: @classreg.learning.combiner.WeightedSum
```

See how many learners in the regularized ensemble have positive weights (so would be included in a shrunken ensemble):

```
sum(bag.Regularization.TrainedWeights > 0)
```

```
ans =
    116    91
```

To shrink the ensemble using the weights from Lambda = 0.1:

```
cmp = shrink(bag,'weightcolumn',2)
```

```
cmp =
```

```
classreg.learning.regr.CompactRegressionEnsemble:
    PredictorNames: {1x20 cell}
```

# RegressionEnsemble.regularize

---

```
CategoricalPredictors: []  
  ResponseName: 'Y'  
  ResponseTransform: 'none'  
    NTrained: 91
```

There are 91 members in the regularized ensemble, which is less than 1/3 of the original 300.

## See Also

[shrink](#) | [cvshrink](#) | [lasso](#)

## Tutorials

- “Ensemble Regularization” on page 16-103

## gmdistribution.RegV property

---

**Purpose** Value of 'Regularize' parameter

**Description** The value of the parameter 'Regularize'.

---

**Note** This property applies only to gmdistribution objects constructed with `fit`.

---

# relieff

---

<b>Purpose</b>	Importance of attributes (predictors) using ReliefF algorithm
<b>Syntax</b>	<pre>[RANKED,WEIGHT] = relieff(X,Y,K) [RANKED,WEIGHT] = relieff(X,Y,K, 'PARAM1',val1, 'PARAM2',val2,     ...)</pre>
<b>Description</b>	<p>[RANKED,WEIGHT] = relieff(X,Y,K) computes ranks and weights of attributes (predictors) for input data matrix <i>X</i> and response vector <i>Y</i> using the ReliefF algorithm for classification or RReliefF for regression with <i>K</i> nearest neighbors. For classification, relieff uses <i>K</i> nearest neighbors per class. RANKED are indices of columns in <i>X</i> ordered by attribute importance, meaning RANKED(1) is the index of the most important predictor. WEIGHT are attribute weights ranging from -1 to 1 with large positive weights assigned to important attributes.</p> <p>If <i>Y</i> is numeric, relieff by default performs RReliefF analysis for regression. If <i>Y</i> is categorical, logical, a character array, or a cell array of strings, relieff by default performs ReliefF analysis for classification.</p> <p>Attribute ranks and weights computed by relieff usually depend on <i>K</i>. If you set <i>K</i> to 1, the estimates computed by relieff can be unreliable for noisy data. If you set <i>K</i> to a value comparable with the number of observations (rows) in <i>X</i>, relieff can fail to find important attributes. You can start with <i>K</i> = 10 and investigate the stability and reliability of relieff ranks and weights for various values of <i>K</i>.</p> <p>[RANKED,WEIGHT] = relieff(X,Y,K, 'PARAM1',val1, 'PARAM2',val2, ...) specifies optional parameter name/value pairs.</p>
<b>Name-Value Pair Arguments</b>	<p>Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.</p> <p><b>method</b></p>



Either 'regression' (default if Y is numeric) or 'classification' (default if Y is not numeric).

### **prior**

Prior probabilities for each class, specified as a string ('empirical' or 'uniform') or as a vector (one value for each distinct group name) or as a structure S with two fields:

- S.group containing the group names as a categorical variable, character array, or cell array of strings
- S.prob containing a vector of corresponding probabilities

If the input value is 'empirical' (default), class probabilities are determined from class frequencies in Y. If the input value is 'uniform', all class probabilities are set equal.

### **updates**

Number of observations to select at random for computing the weight of every attribute. By default all observations are used.

### **categoricalx**

'on' or 'off', 'off' by default. If 'on', treat all predictors in X as categorical. If 'off', treat all predictors in X as numerical. You cannot mix numerical and categorical predictors.

### **sigma**

Distance scaling factor. For observation  $i$ , influence on the attribute weight from its nearest neighbor  $j$  is multiplied by  $\exp(-\text{rank}(i,j)/\text{sigma}^2)$ , where  $\text{rank}(i,j)$  is the position of  $j$  in the list of nearest neighbors of  $i$  sorted by distance in the ascending order. Default is Inf (all nearest neighbors have the same influence) for classification and 50 for regression.

## **Examples**

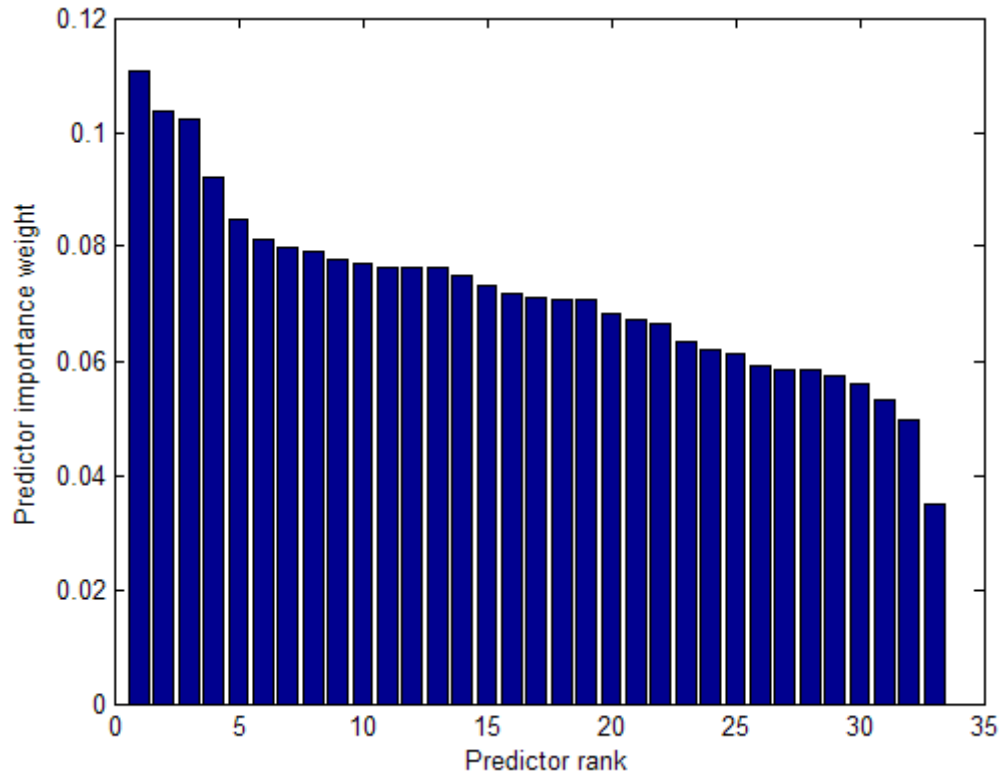
Identify important predictors in the ionosphere dataset:

```
load ionosphere;
```

# relieff

---

```
[ranked,weights] = relieff(X,Y,10);  
bar(weights(ranked));  
xlabel('Predictor rank');  
ylabel('Predictor importance weight');
```



Examine the Fisher iris data to find the important predictors:

```
load fisheriris  
[ranked,weight] = relieff(meas,species,10)
```

```
ranked =
```

	4	3	1	2
weight =	0.1399	0.1226	0.3590	0.3754

The fourth predictor is the most important, and the second predictor is the least important.

## References

- [1] Kononenko, I., Simec, E., & Robnik-Sikonja, M. (1997). *Overcoming the myopia of inductive learning algorithms with RELIEFF*. Retrieved from CiteSeerX: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4740>
- [2] Robnik-Sikonja, M., & Kononenko, I. (1997). *An adaptation of Relief for attribute estimation in regression*. Retrieved from CiteSeerX: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.8381>
- [3] Robnik-Sikonja, M., & Kononenko, I. (2003). *Theoretical and empirical analysis of ReliefF and RReliefF*. *Machine Learning*, 53, 23–69.

## See Also

knnsearch | pdist2

# CompactClassificationEnsemble.removeLearners

---

**Purpose** Remove members of compact classification ensemble

**Syntax** `cens1 = removeLearners(cens,idx)`

**Description** `cens1 = removeLearners(cens,idx)` creates a compact classification ensemble identical to `cens` only without the ensemble members in the `idx` vector.

- Tips**
- Typically, set `cens1` equal to `cens` to retain just one ensemble.
  - Removing learners reduces the memory used by the ensemble and speeds up its predictions.

**Input Arguments**

**cens**  
Compact classification ensemble, constructed with `compact`.

**idx**  
Vector of positive integers with entries from 1 to `cens.NTrained`, where `cens.NTrained` is the number of members in `cens`. `cens1` contains all members of `cens` except those with indices in `idx`.  
Typically, you set `idx = j:cens.NTrained` for some positive integer `j`.

**Output Arguments**

**cens1**  
Compact classification ensemble, identical to `cens` except `cens1` does not contain those members of `cens` with indices in `idx`.

## Examples Remove Learners from an Ensemble

Create a compact classification ensemble. Compact it further by removing members of the ensemble.

Create a compact classification ensemble for the ionosphere data.

```
load ionosphere
ens = fitensemble(X,Y,'AdaBoostM1',100,'Tree');
```

# CompactClassificationEnsemble.removeLearners

---

```
cens = compact(ens);
```

Remove the last 50 members of the ensemble.

```
idx = cens.NTrained-49:cens.NTrained;  
cens1 = removeLearners(cens,idx);
```

**See Also** CompactClassificationEnsemble |

## Related Examples

- “Classification with Imbalanced Data” on page 16-77

# CompactRegressionEnsemble.removeLearners

---

**Purpose** Remove members of compact regression ensemble

**Syntax** `cens1 = removeLearners(cens,idx)`

**Description** `cens1 = removeLearners(cens,idx)` creates a compact regression ensemble identical to `cens` only without the ensemble members in the `idx` vector.

- Tips**
- Typically, set `cens1` equal to `cens` to retain just one ensemble.
  - Removing learners reduces the memory used by the ensemble and speeds up its predictions.

## Input Arguments

**cens**

Compact regression ensemble, constructed with `compact`.

**idx**

Vector of positive integers with entries from 1 to `cens.NTrained`, where `cens.NTrained` is the number of members in `cens`. `cens1` contains the members of `cens` except those with indices in `idx`.

Typically, you set `idx = j:cens.NTrained` for some positive integer `j`.

## Output Arguments

**cens1**

Compact regression ensemble, identical to `cens` except `cens1` does not contain members of `cens` with indices in `idx`.

## Examples

### Remove Learners from an Ensemble

Create a compact regression ensemble. Compact it further by removing members of the ensemble.

Create a compact regression ensemble for the `carsmall` data.

```
load carsmall
X = [Weight Cylinders];
```

# CompactRegressionEnsemble.removeLearners

---

```
ens = fitensemble(X,MPG,'LSBoost',100,'Tree','categorical',2);
cens = compact(ens);
```

Remove the last 50 members of the ensemble.

```
idx = cens.NTrained-49:cens.NTrained;
cens1 = removeLearners(cens,idx);
```

**See Also** [CompactRegressionEnsemble](#) |

# GeneralizedLinearModel.removeTerms

---

**Purpose** Remove terms from generalized linear model

**Syntax** `mdl1 = removeTerms(mdl, terms)`

**Description** `mdl1 = removeTerms(mdl, terms)` returns a linear model the same as `mdl` but with fewer terms.

## Input Arguments

### **mdl**

Generalized linear model, as constructed by `fitglm` or `stepwiseglm`.

### **terms**

Terms to remove from the `mdl` regression model. Specify as either a:

- Text string representing one or more terms to remove. For details, see “Wilkinson Notation” on page 21-2928.
- Row or rows in the terms matrix (see `modelspec` in `fitglm`). For example, if there are three variables A, B, and C:

```
[0 0 0] represents a constant term or intercept  
[0 1 0] represents B; equivalently, A^0 * B^1 * C^0  
[1 0 1] represents A*C  
[2 0 0] represents A^2  
[0 1 2] represents B*(C^2)
```

## Output Arguments

### **mdl1**

Generalized linear model, the same as `mdl` but without the terms given in `terms`. You can set `mdl1` equal to `mdl` to overwrite `mdl`.

## Definitions

### **Wilkinson Notation**

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.



# GeneralizedLinearModel.removeTerms

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

For details, see Wilkinson and Rogers [1].

## Examples

### Remove a Term from a Generalized Linear Regression Model

This example makes a model using two predictors, then removes one.

Generate artificial data for the model, Poisson random numbers with two underlying predictors  $X(1)$  and  $X(2)$ .

```
rng('default') % for reproducibility
rndvars = randn(100,2);
X = [2+rndvars(:,1),rndvars(:,2)];
mu = exp(1 + X*[1;2]);
y = poissrnd(mu);
```

Create a generalized linear regression model of Poisson data.

```
mdl = fitglm(X,y,'y ~ x1 + x2','distr','poisson')
```

# GeneralizedLinearModel.removeTerms

---

```
mdl =
```

```
Generalized Linear regression model:
```

```
log(y) ~ 1 + x1 + x2
```

```
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	1.0405	0.022122	47.034	0
x1	0.9968	0.003362	296.49	0
x2	1.987	0.0063433	313.24	0

```
100 observations, 97 error degrees of freedom
```

```
Dispersion: 1
```

```
Chi^2-statistic vs. constant model: 2.95e+05, p-value = 0
```

```
Remove the second predictor from the model.
```

```
mdl1 = removeTerms(mdl, 'x2')
```

```
mdl1 =
```

```
Generalized Linear regression model:
```

```
log(y) ~ 1 + x1
```

```
Distribution = Poisson
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	2.7784	0.014043	197.85	0
x1	1.1732	0.0033653	348.6	0

```
100 observations, 98 error degrees of freedom
```

```
Dispersion: 1
```

# GeneralizedLinearModel.removeTerms

---

Chi<sup>2</sup>-statistic vs. constant model: 1.25e+05, p-value = 0

## References

[1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. J. Royal Statistics Society 22, pp. 392–399, 1973.

## Alternatives

step adds or removes terms from a model using a greedy one-step algorithm.

## See Also

addTerms | GeneralizedLinearModel | step | stepwiseglm

## Related Examples

- “Plots to Understand Predictor Effects and How to Modify a Model” on page 10-32
- “Generalized Linear Model Workflow” on page 10-42

## Concepts

- “Generalized Linear Models” on page 10-12

# LinearModel.removeTerms

---

<b>Purpose</b>	Remove terms from linear model
<b>Syntax</b>	<code>mdl1 = removeTerms(mdl, terms)</code>
<b>Description</b>	<code>mdl1 = removeTerms(mdl, terms)</code> returns a linear model the same as <code>mdl</code> but with terms removed.
<b>Input Arguments</b>	<b>mdl</b> Linear model, as constructed by <code>fitlm</code> or <code>stepwiselm</code> . <b>terms</b> Terms to remove from the <code>mdl</code> regression model. Specify as either a: <ul style="list-style-type: none"><li>• Text string representing one or more terms to remove. For details, see “Wilkinson Notation” on page 21-2932.</li><li>• Row or rows in the terms matrix (see <code>modelspec</code> in <code>fitlm</code>). For example, if there are three variables A, B, and C:  <code>[0 0 0]</code> represents a constant term or intercept <code>[0 1 0]</code> represents B; equivalently, <math>A^0 * B^1 * C^0</math> <code>[1 0 1]</code> represents <math>A * C</math> <code>[2 0 0]</code> represents <math>A^2</math> <code>[0 1 2]</code> represents <math>B * (C^2)</math></li></ul>
<b>Output Arguments</b>	<b>mdl1</b> Linear model, the same as <code>mdl</code> but with terms removed. You can set <code>mdl1</code> equal to <code>mdl</code> to overwrite <code>mdl</code> .
<b>Definitions</b>	<b>Wilkinson Notation</b> Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using `-1`.

For details, see Wilkinson and Rogers [1].

## Examples

### Remove Terms from Model

Construct a default linear model of the Hald data. Remove terms with high  $p$ -values.

Load the data.

```
load hald
X = ingredients; % predictor variables
y = heat; % response
```

Fit a default linear model to the data.

```
mdl = fitlm(X,y)
```

```
mdl =
```

# LinearModel.removeTerms

---

Linear regression model:  
 $y \sim 1 + x1 + x2 + x3 + x4$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	62.405	70.071	0.8906	0.39913
x1	1.5511	0.74477	2.0827	0.070822
x2	0.51017	0.72379	0.70486	0.5009
x3	0.10191	0.75471	0.13503	0.89592
x4	-0.14406	0.70905	-0.20317	0.84407

Number of observations: 13, Error degrees of freedom: 8  
Root Mean Squared Error: 2.45  
R-squared: 0.982, Adjusted R-Squared 0.974  
F-statistic vs. constant model: 111, p-value = 4.76e-07

Remove the x3 and x4 terms because their *p*-values are so high.

```
terms = 'x3 + x4'; % terms to remove  
mdl1 = removeTerms(mdl, terms)
```

```
mdl1 =
```

Linear regression model:  
 $y \sim 1 + x1 + x2$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	52.577	2.2862	22.998	5.4566e-10
x1	1.4683	0.1213	12.105	2.6922e-07
x2	0.66225	0.045855	14.442	5.029e-08

Number of observations: 13, Error degrees of freedom: 10  
Root Mean Squared Error: 2.41  
R-squared: 0.979, Adjusted R-Squared 0.974

F-statistic vs. constant model: 230, p-value = 4.41e-09

The new model has the same adjusted R-Squared value (0.974) as the previous model, meaning it is about as good a fit. All the terms in the new model have extremely low  $p$ -values.

## References

[1] Wilkinson, G. N., and C. E. Rogers. *Symbolic description of factorial models for analysis of variance*. J. Royal Statistics Society 22, pp. 392–399, 1973.

## Alternatives

Use `stepwiselm` to select a model from a starting model, continuing until no single step is beneficial.

Use `addTerms` to add particular terms.

Use `step` to optimally improve the model by adding or removing terms.

## See Also

`addTerms` | `LinearModel` | `step` | `stepwiselm`

## Tutorials

- “Change Models” on page 9-37

## How To

- “Linear Regression” on page 9-12

# categorical.reorderlevels

---

## **Purpose**

Reorder levels

## **Syntax**

```
B = reorderlevels(A,newlevels)
```

## **Description**

`B = reorderlevels(A,newlevels)` reorders the levels of the categorical array `A`. `newlevels` is a cell array of strings or a 2-D character matrix that specifies the new order. `newlevels` must be a reordering of `getlabels(A)`.

The order of the levels of an ordinal array has significance for relational operators, minimum and maximum, and for sorting.

## **Examples**

Reorder hockey standings:

```
standings = ordinal(1:3,{'Leafs','Canadiens','Bruins'});
getlabels(standings)
ans =
    'Leafs'    'Canadiens'    'Bruins'

standings = reorderlevels(standings,...
    {'Canadiens','Leafs','Bruins'});
getlabels(standings)
ans =
    'Canadiens'    'Leafs'    'Bruins'
```

## **See Also**

`addlevels` | `droplevels` | `getlabels` | `islevel` | `mergelevels`



**Purpose** Repartition data for cross-validation

**Syntax** `cnew = repartition(c)`

**Description** `cnew = repartition(c)` constructs an object `cnew` of the `cvpartition` class defining a random partition of the same type as `c`, where `c` is also an object of the `cvpartition` class.

Repartitioning is useful for Monte-Carlo repetitions of cross-validation analyses. `repartition` is called by `crossval` when the `'mcreps'` parameter is specified.

**Examples** Partition and repartition 100 observations for 3-fold cross-validation:

```
c = cvpartition(100,'kfold',3)
c =
K-fold cross validation partition
      N: 100
  NumTestSets: 3
   TrainSize: 67 66 67
   TestSize: 33 34 33

cnew = repartition(c)
cnew =
K-fold cross validation partition
      N: 100
  NumTestSets: 3
   TrainSize: 67 66 67
   TestSize: 33 34 33
```

Check for equality of the test data in the first fold:

```
isequal(test(c,1),test(cnew,1))
ans =
     0
```

**See Also** `cvpartition`

# dataset.replacedata

---

**Purpose** Replace dataset variables

**Syntax**

```
B = replacedata(A,X)
B = replacedata(A,X,vars)
B = replacedata(A,fun)
B = replacedata(A,fun,vars)
```

**Description** `B = replacedata(A,X)` creates a dataset array `B` with the same variables as the dataset array `A`, but with the data for those variables replaced by the data in the array `X`. `replacedata` creates each variable in `B` using one or more columns from `X`, in order. `X` must have as many columns as the total number of columns in all of the variables in `A`, and as many rows as `A` has observations.

`B = replacedata(A,X,vars)` creates a dataset array `B` with the same variables as the dataset array `A`, but with the data for the variables specified in `vars` replaced by the data in the array `X`. The remaining variables in `B` are copies of the corresponding variables in `A`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. Each variable in `B` has as many columns as the corresponding variable in `A`. `X` must have as many columns as the total number of columns in all the variables specified in `vars`.

`B = replacedata(A,fun)` or `B = replacedata(A,fun,vars)` creates a dataset array `B` by applying the function `fun` to the values in `A`'s variables. `replacedata` first horizontally concatenates `A`'s variables into a single array, then applies the function `fun`. The specified variables in `A` must have types and sizes compatible with the concatenation. `fun` is a function handle that accepts a single input array and returns an array with the same number of rows and columns as the input.

**Examples**

```
data = dataset({rand(3,3)}, 'Var1', 'Var2', 'Var3')
```

```
% Use ZSCORE to normalize each variable in a dataset array
% separately, by explicitly extracting and transforming the
% data, and then replacing it.
```

```
X = double(data);
X = zscore(X);
data = replacedata(data,X)

% Equivalently, provide a handle to ZSCORE.
data = replacedata(data,@zscore)

% Use ZSCORE to normalize each observation in a dataset
% array separately by creating an anonymous function.
data = replacedata(data,@(x) zscore(x,[],2))
```

### See Also

dataset

# dataset.replaceWithMissing

---

**Purpose** Insert missing data indicators into a dataset array

**Syntax** `ds2 = replaceWithMissing(ds,Name,Value)`

**Description** `ds2 = replaceWithMissing(ds,Name,Value)` replaces specified values in a dataset array with standard missing data indicators using options specified by one or more `Name,Value` pair arguments. Use `replaceWithMissing` to specify:

- Which numeric missing value indicators to replace with NaN.
- Which string missing value indicators to replace with an empty string.
- Which categorical levels to replace with `<undefined>`.

## Input Arguments

**ds**  
dataset array.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

#### 'NumericValues'

Vector of numeric values that `replaceWithMissing` replaces with NaN.

#### 'CategoricalLevels'

String or cell array of strings naming the categorical levels that `replaceWithMissing` replaces with `<undefined>`.

#### 'Strings'

String or cell array of strings containing the strings that `replaceWithMissing` replaces with the empty string, `''`.

## 'DataVars'

Specified set of variables in `ds` for which `replaceWithMissing` replaces values. You can specify a positive integer or vector of positive integers indicating the variable column numbers, a variable name or a cell array of variables names, or a logical vector indicating which variables to replace missing values in.

**Default:** All variables in `ds`.

## Output Arguments

### `ds2`

dataset array that has the specified missing value indicators, in the specified variables of `ds`, replaced with standard missing value indicators.

## Examples

### Replace Nonstandard Missing Value Indicators

Replace nonstandard missing value indicators with standard missing value indicators.

Replace numeric missing values coded 99 with NaN, and string missing values coded ' .' with the empty string.

```
ds = replaceWithMissing(ds,'NumericValues',99,'Strings','.');
```

## See Also

[dataset](#) | [ismissing](#) |

## Related Examples

- “Clean Messy and Missing Data” on page 2-117

## Concepts

- “Dataset Arrays” on page 2-135

# categorical repmat

---

**Purpose** Replicate and tile categorical array

**Syntax**  
B = repmat(A,m,n)  
B = repmat(A,[m n p ...])

**Description** B = repmat(A,m,n) creates a large array B consisting of an m-by-n tiling of copies of the categorical array A. The size of B is [size(A,1)\*M size(A,2)\*N, size(A,3), ...]. repmat(A,n) creates an n-by-n tiling.

B = repmat(A,[m n p ...]) tiles the categorical array A to produce a multidimensional array B composed of copies of A. The size of B is [size(A,1)\*M, size(A,2)\*N, size(A,3)\*P, ...].

**See Also** ndims | size

**Purpose** Reset state

**Syntax** reset(q)

**Description** reset(q) resets the state of the quasi-random number stream q of the grandstream class back to its initial state, 1. Subsequent points drawn from the stream will be the same as those drawn from a new stream. The command is equivalent to q.State = 1.

**Examples** Use grandstream to construct a 3-D Halton stream, based on a point set that skips the first 1000 values and then retains every 101st point:

```
q = grandstream('halton',3,'Skip',1e3,'Leap',1e2)
q =
  Halton quasi-random stream in 3 dimensions
  Point set properties:
      Skip : 1000
      Leap : 100
      ScrambleMethod : none

nextIdx = q.State
nextIdx =
     1
```

Use grand to generate two samples of size four:

```
X1 = grand(q,4)
X1 =
    0.0928    0.3475    0.0051
    0.6958    0.2035    0.2371
    0.3013    0.8496    0.4307
    0.9087    0.5629    0.6166

nextIdx = q.State
nextIdx =
     5

X2 = grand(q,4)
```

# grandstream.reset

---

```
X2 =  
    0.2446    0.0238    0.8102  
    0.5298    0.7540    0.0438  
    0.3843    0.5112    0.2758  
    0.8335    0.2245    0.4694  
nextIdx = q.State  
nextIdx =  
    9
```

Use reset to reset the stream, then generate another sample:

```
reset(q)  
nextIdx = q.State  
nextIdx =  
    1  
  
X = grand(q,4)  
X =  
    0.0928    0.3475    0.0051  
    0.6958    0.2035    0.2371  
    0.3013    0.8496    0.4307  
    0.9087    0.5629    0.6166
```

## See Also

[grandstream](#) | [grand](#)



## Purpose

Resize categorical array

## Syntax

```
B = reshape(A,M,N)
B = reshape(A,m,n,p,...)
reshape(A,[m n p ...])
B = reshape(A,...,[],...)
```

## Description

`B = reshape(A,M,N)` returns an  $m$ -by- $n$  categorical matrix whose elements are taken columnwise from the categorical array `A`. An error results if `A` does not have  $m*n$  elements.

`B = reshape(A,m,n,p,...)` or `reshape(A,[m n p ...])` returns an array with the same elements as `A` but reshaped to have the size  $m$ -by- $n$ -by- $p$ -by- $\dots$ .  $m*n*p*\dots$  must be the same as `numel(A)`.

`B = reshape(A,...,[],...)` calculates the length of the dimension represented by `[]`, such that the product of the dimensions equals `numel(A)`. `numel(A)` must be evenly divisible by the product of the known dimensions. You can use only one occurrence of `[]`.

In general, `reshape(A,siz)` returns an array with the same elements as `A` but reshaped to the size `siz`. `prod(siz)` must be the same as `numel(A)`.

## See Also

`shiftdim` | `squeeze`

# LinearMixedModel.residuals

---

**Purpose** Residuals of fitted linear mixed-effects model

**Syntax**  
R = residuals(lme)  
R = residuals(lme,Name,Value)

**Description** R = residuals(lme) returns the raw conditional residuals from a fitted linear mixed-effects model lme.

R = residuals(lme,Name,Value) returns the residuals from the linear mixed-effects model lme with additional options specified by one or more Name,Value pair arguments.

For example, you can specify Pearson or standardized residuals, or residuals with contributions from only fixed effects.

## Input Arguments

### **lme - Linear mixed-effects model**

LinearMixedModel object

Linear mixed-effects model, returned as a LinearMixedModel object.

For properties and methods of this object, see LinearMixedModel.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### **'Conditional' - Indicator for conditional residuals**

True (default) | False

Indicator for conditional residuals, specified as the comma-separated pair consisting of 'Conditional' and one of the following.

True	Contribution from both fixed effects and random effects (conditional)
False	Contribution from only fixed effects (marginal)

**Example:** 'Conditional', 'False'

## 'ResidualType' - Residual type

'Raw' (default) | 'Pearson' | 'Standardized'

Residual type, specified by the comma-separated pair consisting of ResidualType and one of the following.

Residual Type	Conditional	Marginal
'Raw'	$r_i^C = [y - X\hat{\beta} - Z\hat{b}]_i$	$r_i^M = [y - X\hat{\beta}]_i$
'Pearson'	$pr_i^C = \frac{r_i^C}{\sqrt{[\text{Var}_{y,b}(y - X\hat{\beta} - Z\hat{b})]_{ii}}}$	$pr_i^M = \frac{r_i^M}{\sqrt{[\text{Var}_y(y - X\hat{\beta})]_{ii}}}$
'Standardized'	$st_i^C = \frac{r_i^C}{\sqrt{[\text{Var}_y(r^C)]_{ii}}}$	$st_i^M = \frac{r_i^M}{\sqrt{[\text{Var}_y(r^M)]_{ii}}}$

**Example:** 'ResidualType', 'Standardized'

## Output Arguments

### R - Residuals

*n*-by-1 vector

Residuals of the fitted linear mixed-effects model `lmer` returned as an *n*-by-1 vector, where *n* is the number of observations.

# LinearMixedModel.residuals

---

## Examples

### Plot Residuals vs. Fitted Values

Navigate to a folder containing sample data.

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

Load the sample data.

```
load weight
```

`weight` contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over six 2-week time periods. This is simulated data.

Store the data in a dataset array. Define `Subject` and `Program` as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);
ds.Subject = nominal(ds.Subject);
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

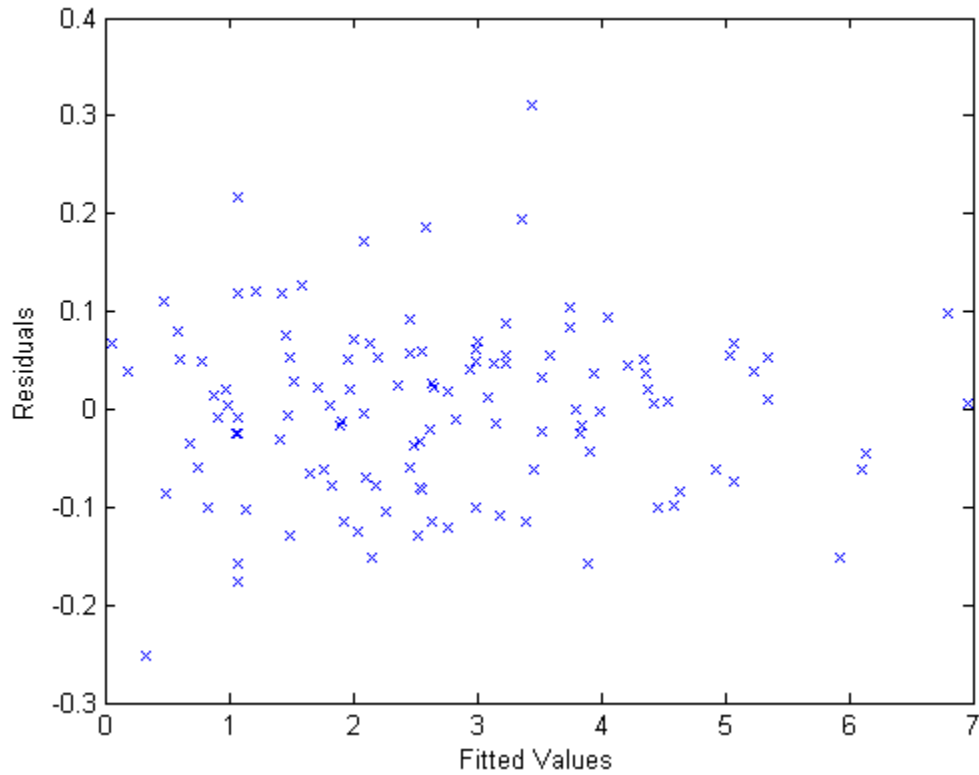
```
lme = fitlme(ds,'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Compute the fitted values and raw residuals.

```
F = fitted(lme);
R = residuals(lme);
```

Plot the residuals versus the fitted values.

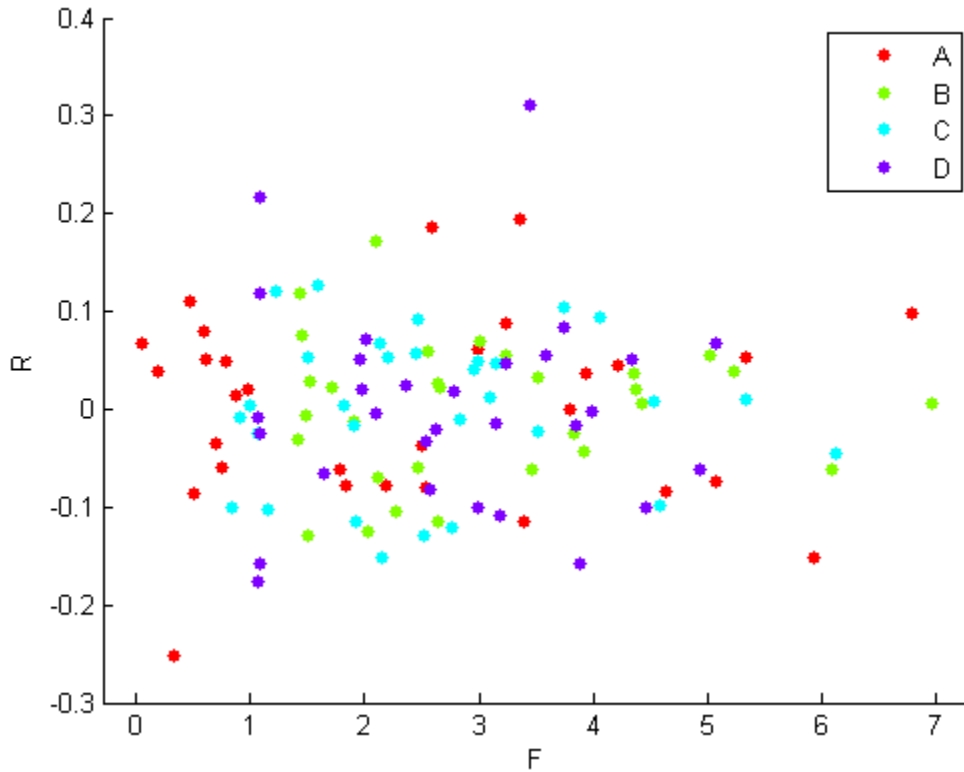
```
plot(F,R,'bx')
xlabel('Fitted Values')
ylabel('Residuals')
```



Now, plot the residuals versus the fitted values, grouped by program.

```
figure();  
gscatter(F,R,Program)
```

# LinearMixedModel.residuals



The residuals seem to behave similarly across levels of the program as expected.

## Compute Conditional and Marginal Pearson Residuals

Load the sample data.

```
load carbig
```

Store the variables for miles per gallon (MPG), acceleration, horsepower, cylinders, and model year in a dataset array.

```
ds = dataset(MPG,Acceleration,Horsepower,Cylinders,Model_Year);
```

Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and the cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

```
lme = fitlme(ds,'MPG ~ Acceleration + Horsepower + Cylinders + (Accel
```

Compute the conditional Pearson residuals and display the first five residuals.

```
PR = residuals(lme,'ResidualType','Pearson');  
PR(1:5)
```

```
ans =
```

```
-0.0533  
0.0652  
0.3655  
-0.0106  
-0.3340
```

Compute the marginal Pearson residuals and display the first five residuals.

```
PRM = residuals(lme,'ResidualType','Pearson','Conditional',false);  
PRM(1:5)
```

```
ans =
```

```
-0.1250  
0.0130  
0.3242  
-0.0861  
-0.3006
```

# LinearMixedModel.residuals

---

## Examine Residuals

Load the sample data.

```
load carbig
```

Store the variables for miles per gallon (MPG), acceleration, horsepower, cylinders, and model year in a dataset array.

```
ds = dataset(MPG,Acceleration,Horsepower,Cylinders,Model_Year);
```

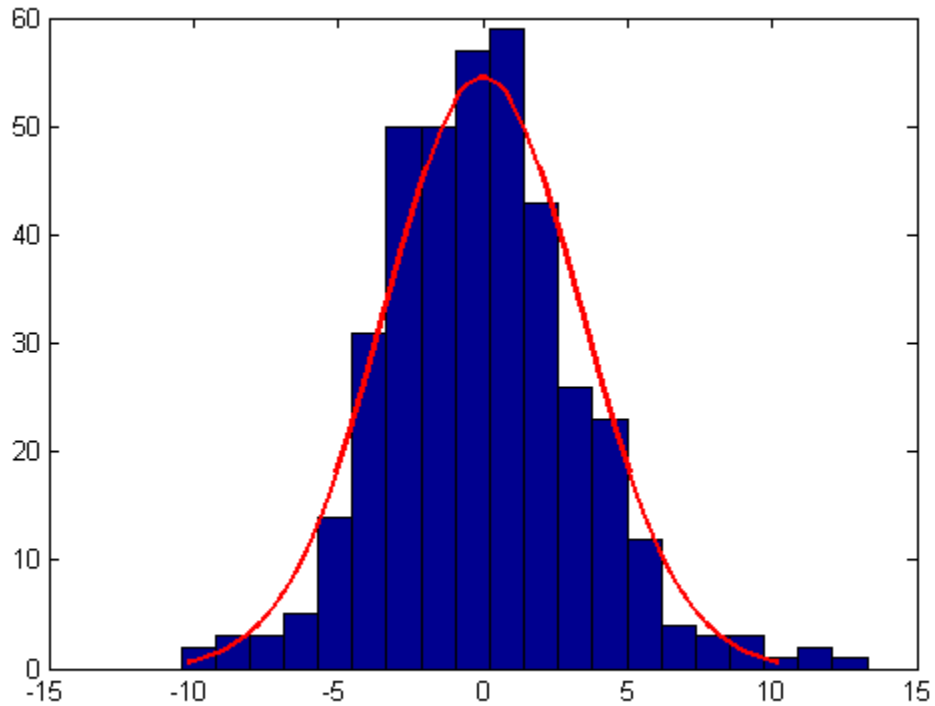
Fit a linear mixed-effects model for miles per gallon (MPG), with fixed effects for acceleration, horsepower, and the cylinders, and potentially correlated random effects for intercept and acceleration grouped by model year.

```
lme = fitlme(ds,'MPG ~ Acceleration + Horsepower + Cylinders + (Accelerat
```

Draw a histogram of the raw residuals with a normal fit.

```
r = residuals(lme);  
histfit(r)
```



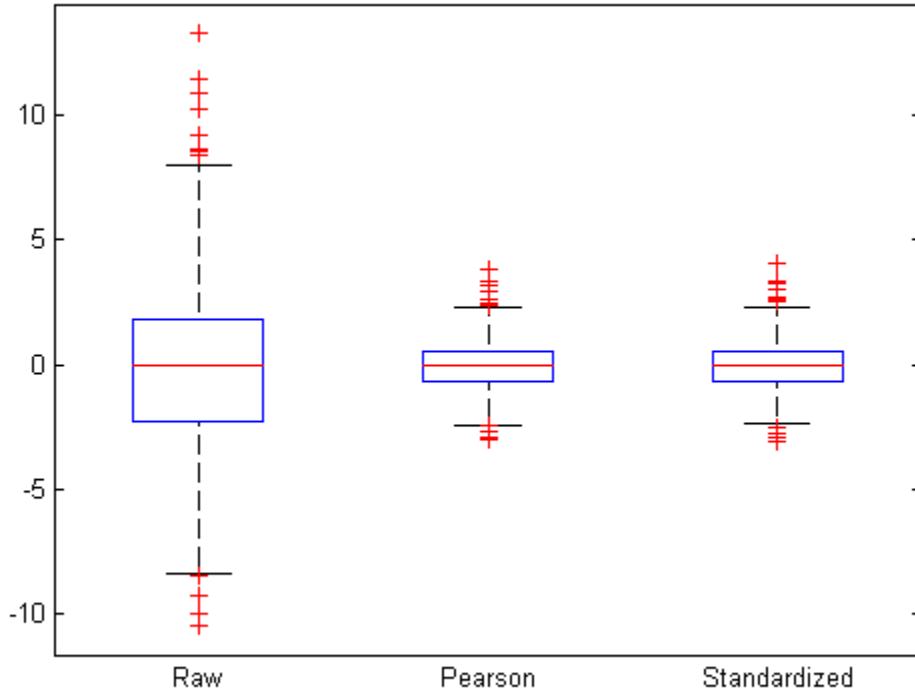


Normal distribution seems to be a good fit for the residuals.

Compute the conditional Pearson and standardized residuals and create box plots of all three types of residuals.

```
pr = residuals(lme, 'ResidualType', 'Pearson');  
st = residuals(lme, 'ResidualType', 'Standardized');  
X = [r pr st];  
figure();  
boxplot(X)
```

# LinearMixedModel.residuals



Red plus signs show the observations with residuals above or below  $q3 + 1.5(q3 - q1)$  and  $q1 - 1.5(q3 - q1)$ , where  $q1$  and  $q3$  are the 25th and 75th percentiles, respectively.

Find the observations with residuals that are 2.5 standard deviations above and below the mean.

```
find(r > nanmean(r) + 2.5*nanstd(r))
```

```
ans =
```

```
62  
252  
255  
330  
337  
341  
396
```

```
find(r < nanmean(r) - 2.5*nanstd(r))
```

```
ans =
```

```
119  
324  
375
```

## Definitions

### Conditional and Marginal Residuals

Conditional residuals include contributions from both fixed and random effects, whereas marginal residuals include contribution from only fixed effects.

Suppose the linear mixed-effects model `lme` has an  $n$ -by- $p$  fixed-effects design matrix  $X$  and an  $n$ -by- $q$  random-effects design matrix  $Z$ . Also, suppose the  $p$ -by-1 estimated fixed-effects vector is  $\hat{\beta}$ , and the  $q$ -by-1 estimated best linear unbiased predictor (BLUP) vector of random effects is  $\hat{b}$ . The fitted conditional response is

$$\hat{y}_{Cond} = X\hat{\beta} + Z\hat{b},$$

and the fitted marginal response is

$$\hat{y}_{Mar} = X\hat{\beta}.$$

# LinearMixedModel.residuals

---

`residuals` can return three types of residuals: raw, Pearson, and standardized. For any type, you can compute the conditional or the marginal residuals. For example, the conditional raw residual is

$$r_{Cond} = y - X\hat{\beta} - Z\hat{b},$$

and the marginal raw residual is

$$r_{Mar} = y - X\hat{\beta}.$$

For more information on other types of residuals, see the `ResidualType` name-value pair argument.

## See Also

`LinearMixedModel` | `fitted` | `plotResiduals` | `response` |

<b>Purpose</b>	Response vector of the linear mixed-effects model
<b>Syntax</b>	<code>y = response(lme)</code>
<b>Description</b>	<code>y = response(lme)</code> returns the response vector <code>y</code> used to fit the linear mixed-effects model <code>lme</code> .
<b>Input Arguments</b>	<b>lme - Linear mixed-effects model</b> <code>LinearMixedModel</code> object  Linear mixed-effects model, returned as a <code>LinearMixedModel</code> object.  For properties and methods of this object, see <code>LinearMixedModel</code> .
<b>Output Arguments</b>	<b>y - Response values</b> <code>n</code> -by-1 vector  Response values, specified as an <code>n</code> -by-1 vector, where <code>n</code> is the number of observations.  <b>Data Types</b> <code>single</code>   <code>double</code>
<b>Examples</b>	<b>Plot Response versus Fitted Values</b>  Navigate to a folder containing sample data.  <code>cd(matlabroot)</code> <code>cd('help/toolbox/stats/examples')</code>  Load the sample data.  <code>load weight</code>  <code>weight</code> contains data from a longitudinal study, where 20 subjects are randomly assigned to 4 exercise programs, and their weight loss is recorded over two-week time periods. This is simulated data.

# LinearMixedModel.response

---

Store the data in a dataset array. Define Subject and Program as categorical variables.

```
ds = dataset(InitialWeight,Program,Subject,Week,y);  
ds.Subject = nominal(ds.Subject);  
ds.Program = nominal(ds.Program);
```

Fit a linear mixed-effects model where the initial weight, type of program, week, and the interaction between the week and type of program are the fixed effects. The intercept and week vary by subject.

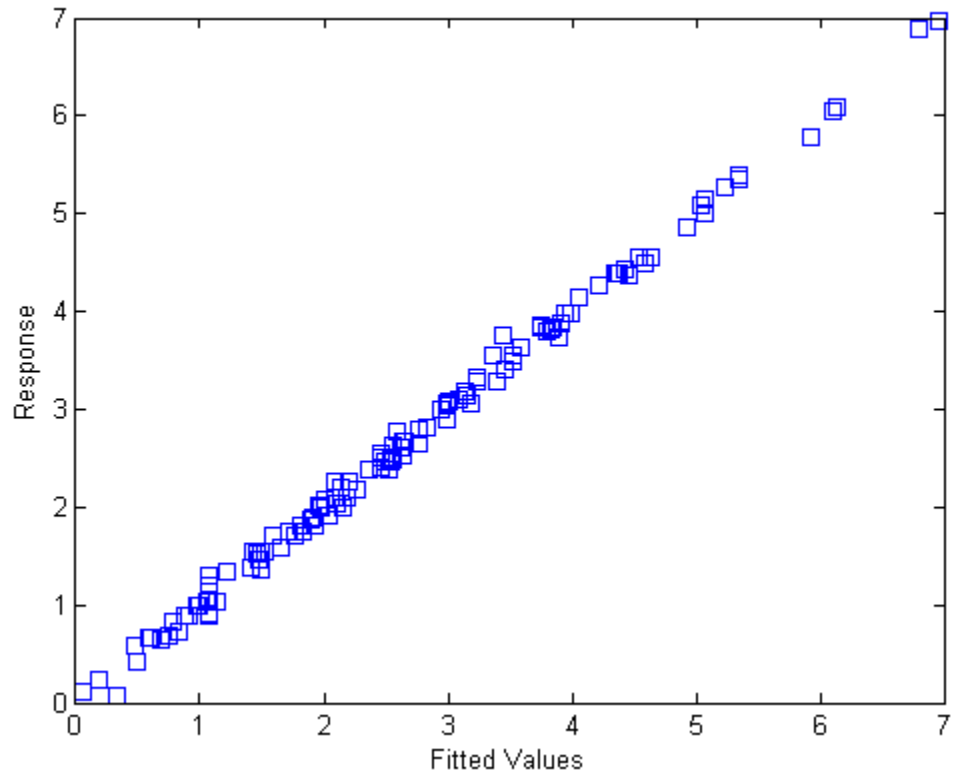
```
lme = fitlme(ds,'y ~ InitialWeight + Program*Week + (Week|Subject)');
```

Compute the fitted values and the response.

```
F = fitted(lme);  
y = response(lme);
```

Plot the response versus the fitted values.

```
plot(F,y,'bs')  
xlabel('Fitted Values')  
ylabel('Response')
```



## See Also

[LinearMixedModel](#) | [fitted](#) | [residuals](#) |

# ClassificationDiscriminant.resubEdge

---

<b>Purpose</b>	Classification edge by resubstitution
<b>Syntax</b>	<code>edge = resubEdge(obj)</code>
<b>Description</b>	<code>edge = resubEdge(obj)</code> returns the classification edge obtained by <code>obj</code> on its training data.
<b>Input Arguments</b>	<b>obj</b> Discriminant analysis classifier, produced using <code>ClassificationDiscriminant.fit</code> .
<b>Output Arguments</b>	<b>edge</b> Classification edge obtained by resubstituting the training data into the calculation of <code>edge</code> .
<b>Definitions</b>	<b>Edge</b> The <i>edge</i> is the weighted mean value of the classification <i>margin</i> . The weights are the class probabilities in <code>obj.Prior</code> . If you supply weights in the <code>weights</code> name-value pair, those weights are normalized to sum to the prior probabilities in the respective classes, and are then used to compute the weighted average. <b>Margin</b> The classification <i>margin</i> is the difference between the classification <i>score</i> for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix <code>obj.X</code> . A high value of margin indicates a more reliable prediction than a low value. <b>Score (discriminant analysis)</b> For discriminant analysis, the <i>score</i> of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 15-7.



## Examples

Estimate the quality of a discriminant analysis classifier for the Fisher iris data by resubstitution:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
redge = resubEdge(obj)

redge =
    0.9454
```

## See Also

[ClassificationDiscriminant](#) | [edge](#) | [resubMargin](#)

## How To

- “Discriminant Analysis” on page 15-3

# ClassificationEnsemble.resubEdge

---

**Purpose** Classification edge by resubstitution

**Syntax**  
edge = resubEdge(ens)  
edge = resubEdge(ens,Name,Value)

**Description**  
edge = resubEdge(ens) returns the classification edge obtained by ens on its training data.  
edge = resubEdge(ens,Name,Value) calculates edge with additional options specified by one or more Name,Value pair arguments. You can specify several name-value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

## Input Arguments

**ens**  
A classification ensemble created with fitensemble.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

#### 'learners'

Indices of weak learners in the ensemble ranging from 1 to NTrained. resubEdge uses only these learners for calculating edge.

**Default:** 1:NTrained

#### 'mode'

String representing the meaning of the output edge:

- 'ensemble' — edge is a scalar value, the loss for the entire ensemble.

- 'individual' — edge is a vector with one element per trained learner.
- 'cumulative' — edge is a vector in which element J is obtained by using learners 1:J from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

### edge

Classification edge obtained by `ens` by resubstituting the training data into the calculation of edge. Classification edge is classification margin averaged over the entire data. edge can be a scalar or vector, depending on the setting of the `mode` name-value pair.

## Definitions

### Edge

The *edge* is the weighted mean value of the classification margin. The weights are the class probabilities in `ens.Prior`.

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

# ClassificationEnsemble.resubEdge

---

## Examples

Find the resubstitution edge for an ensemble that classifies the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
edge = resubEdge(ens)
```

```
edge =
    3.2486
```

## See Also

[resubMargin](#) | [resubLoss](#) | [resubPredict](#) | [resubEdge](#)

<b>Purpose</b>	Edge of $k$ -nearest neighbor classifier by resubstitution
<b>Syntax</b>	$E = \text{resubEdge}(\text{mdl})$
<b>Description</b>	$E = \text{resubEdge}(\text{mdl})$ returns the classification edge for $\text{mdl}$ with the data used to train $\text{mdl}$ (see “Edge” on page 21-2965).
<b>Input Arguments</b>	<b>mdl</b> $k$ -nearest neighbor classifier, created by <code>ClassificationKNN.fit</code> .
<b>Output Arguments</b>	<b>E</b> Classification edge, a scalar that is the mean classification margin (see “Margin” on page 21-2965).
<b>Definitions</b>	<b>Edge</b> The <i>edge</i> is the mean value of the classification <i>margin</i> . <b>Margin</b> The classification <i>margin</i> is the difference between the classification <i>score</i> for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the training data. <b>Score</b> The <i>score</i> of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 21-2608.
<b>Examples</b>	<b>Resubstitution Edge Calculation</b> Construct a $k$ -nearest neighbor classifier for the Fisher iris data, where $k = 5$ .

# ClassificationKNN.resubEdge

---

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',5);
```

Examine the resubstitution edge of the classifier.

```
E = resubEdge(mdl)
```

```
E =
```

```
0.9253
```

## See Also

[ClassificationKNN](#) | [resubLoss](#) | [resubMargin](#) | [resubPredict](#) |

## Related Examples

- “Examine the Quality of a KNN Classifier” on page 16-27

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9

<b>Purpose</b>	Classification edge by resubstitution
<b>Syntax</b>	<code>edge = resubEdge(tree)</code>
<b>Description</b>	<code>edge = resubEdge(tree)</code> returns the classification edge obtained by <code>tree</code> on its training data.
<b>Input Arguments</b>	<b>tree</b> A classification tree created by <code>ClassificationTree.fit</code> .
<b>Output Arguments</b>	<b>edge</b> Classification edge obtained by resubstituting the training data into the calculation of <code>edge</code> .

## Definitions

### Edge

The *edge* is the weighted mean value of the classification margin. The weights are the class probabilities in `tree.Prior`.

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `X`.

### Score (tree)

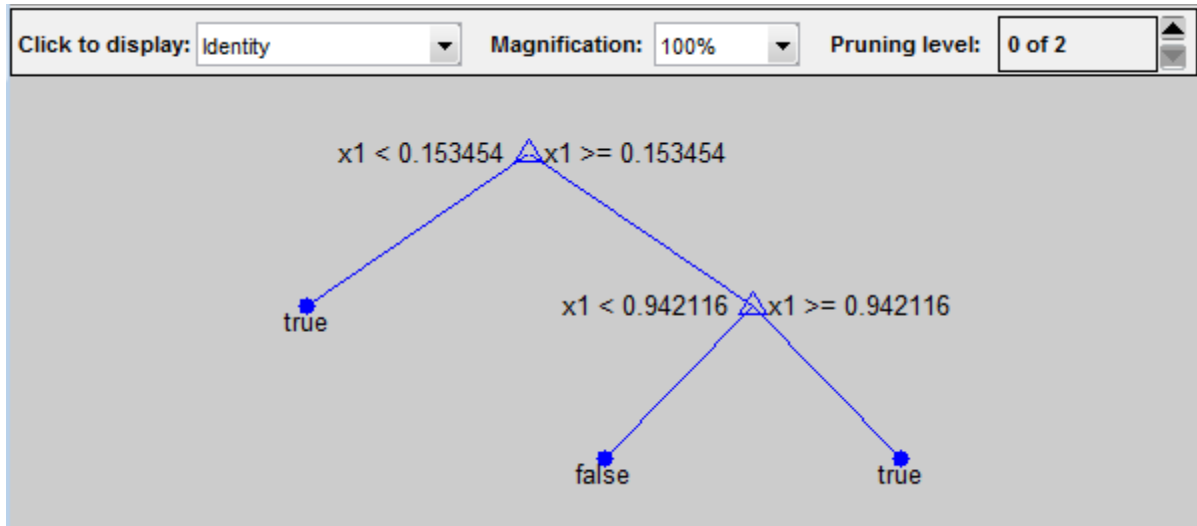
For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor `X` as true when `X < 0.15` or `X > 0.95`, and `X` is false otherwise.

# ClassificationTree.resubEdge

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```

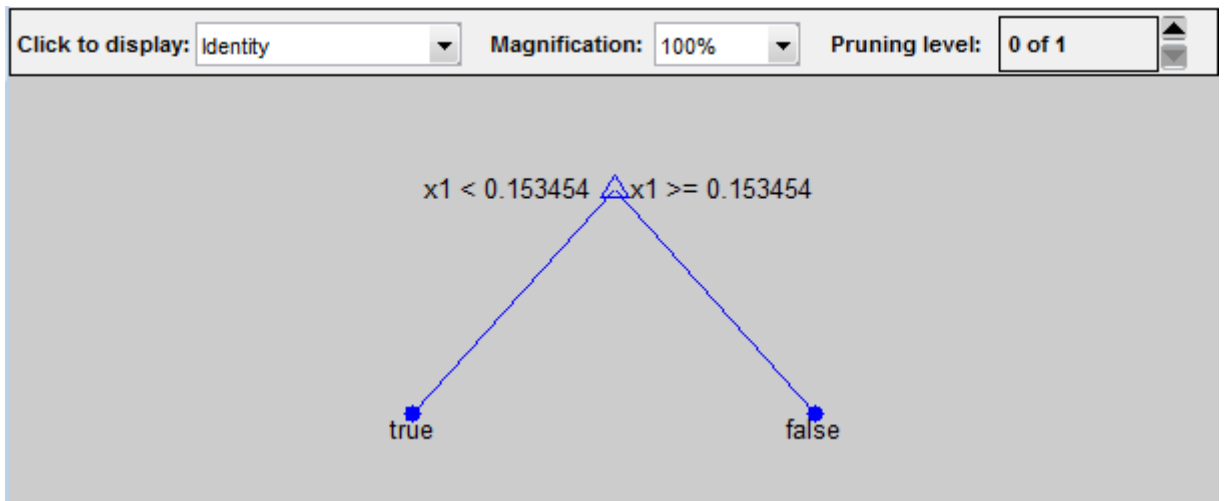


**2**

Prune the tree:

```
tree1 = prune(tree,'level',1);
view(tree1,'mode','graph')
```





The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
0.9059    0.0941    0.8147  
0.9059    0.0941    0.9058  
0        1.0000    0.1270  
0.9059    0.0941    0.9134  
0.9059    0.0941    0.6324  
0        1.0000    0.0975  
0.9059    0.0941    0.2785
```

# ClassificationTree.resubEdge

---

0.9059	0.0941	0.5469
0.9059	0.0941	0.9575
0.9059	0.0941	0.9649

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## Examples

Estimate the quality of a classification tree for the Fisher iris data by resubstitution:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
redge = resubEdge(tree)

redge =
    0.9384
```

## See Also

[edge](#) | [resubMargin](#) | [resubLoss](#) | [resubPredict](#)

## Purpose

Classification error by resubstitution

## Syntax

```
L = resubLoss(obj)
L = resubLoss(obj,Name,Value)
```

## Description

`L = resubLoss(obj)` returns the resubstitution loss, meaning the loss computed for the data that `ClassificationDiscriminant.fit` used to create `obj`.

`L = resubLoss(obj,Name,Value)` returns loss statistics with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **obj**

Discriminant analysis classifier, produced using `ClassificationDiscriminant.fit`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'lossfun'**

Function handle or string representing a loss function. Built-in loss functions are:

- 'binodeviance' — See “Loss Functions” on page 21-2972.
- 'classiferror' — Fraction of misclassified observations. See “Loss Functions” on page 21-2972.
- 'exponential' — See “Loss Functions” on page 21-2972.
- 'mincost' — Smallest misclassification cost as given by the `obj.Cost` matrix.

# ClassificationDiscriminant.resubLoss

---

You can write your own loss function using the syntax described in “Loss Functions” on page 21-2972.

**Default:** 'mincost'

## Output Arguments

**L**

Classification error, a scalar. The meaning of the error depends on the values in `weights` and `lossfun`. See “Classification Error” on page 21-2972.

## Definitions

### Classification Error

The default classification error is the fraction of the training data `X` that `obj` misclassifies.

Weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when `obj` misclassifies the  $i$ th row of `X`, divided by the sum of the weights.

### Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

- 'classiferror' — Predict the label with the largest posterior probability. The loss is then the fraction of misclassified observations.

- 'mincost' — Predict the label with the smallest expected misclassification cost, with expectation taken over the posterior probability, and cost as given by the `Cost` property of the classifier (a matrix). The loss is then the true misclassification cost averaged over the observations.

To write your own loss function, create a function file in this form:

```
function loss = lossfun(C,S,W,COST)
```

- `N` is the number of rows of `X`.
- `K` is the number of classes in the classifier, represented in the `ClassNames` property.
- `C` is an `N`-by-`K` logical matrix, with one true per row for the true class. The index for each class is its position in the `ClassNames` property.
- `S` is an `N`-by-`K` numeric matrix. `S` is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from `predict`.
- `W` is a numeric vector with `N` elements, the observation weights. If you pass `W`, the elements are normalized to sum to the prior probabilities in the respective classes.
- `COST` is a `K`-by-`K` numeric matrix of misclassification costs. For example, you can use `COST = ones(K) - eye(K)`, which means a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## Posterior Probability

The posterior probability that a point  $z$  belongs to class  $j$  is the product of the prior probability and the multivariate normal density. The density function of the multivariate normal with mean  $\mu_j$  and covariance  $\Sigma_j$  at a point  $z$  is

# ClassificationDiscriminant.resubLoss

---

$$P(x | k) = \frac{1}{(2\pi |\Sigma_k|)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k)\right),$$

where  $|\Sigma_k|$  is the determinant of  $\Sigma_k$ , and  $\Sigma_k^{-1}$  is the inverse matrix.

Let  $P(k)$  represent the prior probability of class  $k$ . Then the posterior probability that an observation  $x$  is of class  $k$  is

$$\hat{P}(k | x) = \frac{P(x | k)P(k)}{P(x)},$$

where  $P(x)$  is a normalization constant, the sum over  $k$  of  $P(x | k)P(k)$ .

## Prior Probability

The prior probability is one of three choices:

- 'uniform' — The prior probability of class  $k$  is one over the total number of classes.
- 'empirical' — The prior probability of class  $k$  is the number of training samples of class  $k$  divided by the total number of training samples.
- Custom — The prior probability of class  $k$  is the  $k$ th element of the prior vector. See `ClassificationDiscriminant.fit`.

After creating a classifier `obj`, you can set the prior by dot addressing:

```
obj.Prior = v;
```

where `v` is a vector of positive elements representing the frequency with which each element occurs. You do not need to retrain the classifier when you set a new prior.

## Cost

The matrix of expected costs per observation is defined in “Cost” on page 15-8.

## Examples

Compute the resubstituted classification error for the Fisher iris data:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
L = resubLoss(obj)
```

```
L =
    0.0200
```

## See Also

[ClassificationDiscriminant | loss](#)

## How To

- “Discriminant Analysis” on page 15-3

# ClassificationEnsemble.resubLoss

---

**Purpose** Classification error by resubstitution

**Syntax**  
L = resubLoss(ens)  
L = resubLoss(ens,Name,Value)

**Description** L = resubLoss(ens) returns the resubstitution loss, meaning the loss computed for the data that fitensemble used to create ens.

L = resubLoss(ens,Name,Value) calculates loss with additional options specified by one or more Name,Value pair arguments. You can specify several name-value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

## Input Arguments

**ens**  
A classification ensemble created with fitensemble.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, ...,NameN,ValueN.

### 'learners'

Indices of weak learners in the ensemble ranging from 1 to NTrained. resubLoss uses only these learners for calculating loss.

**Default:** 1:NTrained

### 'lossfun'

Function handle or string representing a loss function. Built-in loss functions:

- 'binodeviance' — See “Loss Functions” on page 21-2977
- 'classiferror' — Fraction of misclassified data



- 'exponential' — See “Loss Functions” on page 21-2977

You can write your own loss function in the syntax described in “Loss Functions” on page 21-2977.

**Default:** 'classiferror'

## 'mode'

String representing the meaning of the output L:

- 'ensemble' — L is a scalar value, the loss for the entire ensemble.
- 'individual' — L is a vector with one element per trained learner.
- 'cumulative' — L is a vector in which element J is obtained by using learners 1:J from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

**L**

Loss, by default the fraction of misclassified data. L can be a vector, and can mean different things, depending on the name-value pair settings.

## Definitions

### Classification Error

The default classification error is the fraction of the training data X that `ens` misclassifies.

Weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when `tree` misclassifies the  $i$ th row of X, divided by the sum of the weights.

### Loss Functions

The built-in loss functions are:

# ClassificationEnsemble.resubLoss

---

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'classiferror' — Fraction of misclassified data, weighted by  $w$ .
- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

To write your own loss function, create a function file of the form

```
function loss = lossfun(C,S,W,COST)
```

- $N$  is the number of rows of `ens.X`.
- $K$  is the number of classes in `ens`, represented in `ens.ClassNames`.
- $C$  is an  $N$ -by- $K$  logical matrix, with one `true` per row for the true class. The index for each class is its position in `tree.ClassNames`.
- $S$  is an  $N$ -by- $K$  numeric matrix.  $S$  is a matrix of posterior probabilities for classes with one row per observation, similar to the `score` output from `predict`.
- $W$  is a numeric vector with  $N$  elements, the observation weights.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. The default 'classiferror' gives a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## Examples

Compute the resubstitution loss for a classification ensemble for the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
loss = resubLoss(ens)
```

```
loss =
    0.0333
```

## See Also

[resubEdge](#) | [resubMargin](#) | [resubPredict](#) | [resubLoss](#)

# ClassificationKNN.resubLoss

---

**Purpose** Loss of  $k$ -nearest neighbor classifier by resubstitution

**Syntax** `L = resubLoss mdl`  
`L = resubLoss mdl, Name, Value`

**Description** `L = resubLoss mdl` returns the resubstitution loss, meaning the loss computed for the data that `ClassificationKNN.fit` used to create `mdl`.  
`L = resubLoss mdl, Name, Value` returns loss statistics with additional options specified by one or more `Name, Value` pair arguments.

**Input Arguments** **mdl**  
*k*-nearest neighbor classifier, created by `ClassificationKNN.fit`.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## 'lossfun'

Function handle or string representing a loss function. Built-in loss functions are:

- 'binodeviance' — See "Loss Functions" on page 21-2981.
- 'classiferror' — Fraction of misclassified observations. See "Loss Functions" on page 21-2981.
- 'exponential' — See "Loss Functions" on page 21-2981.
- 'mincost' — Smallest misclassification cost as given by the `mdl.Cost` matrix.

You can write your own loss function using the syntax described in “Loss Functions” on page 21-2981.

**Default:** 'mincost'

## Output Arguments

**L**

Classification error, a scalar. The meaning of the error depends on the values in `weights` and `lossfun`. See “Classification Error” on page 21-2981.

## Definitions

### Classification Error

The default classification error is the fraction of data  $X$  that `mdl` misclassifies, where  $Y$  represents the true classifications.

The weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when `mdl` misclassifies the  $i$ th row of  $X$ , divided by the sum of the weights.

### Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

- 'classiferror' — Predict the label with the largest posterior probability. The loss is then the fraction of misclassified observations.

# ClassificationKNN.resubLoss

---

- 'mincost' — Predict the label with the smallest expected misclassification cost, with expectation taken over the posterior probability, and cost as given by the `Cost` property of the classifier (a matrix). The loss is then the true misclassification cost averaged over the observations.

To write your own loss function, create a function file in this form:

```
function loss = lossfun(C,S,W,COST)
```

- `N` is the number of rows of `X`.
- `K` is the number of classes in the classifier, represented in the `ClassNames` property.
- `C` is an `N`-by-`K` logical matrix, with one true per row for the true class. The index for each class is its position in the `ClassNames` property.
- `S` is an `N`-by-`K` numeric matrix. `S` is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from `predict`.
- `W` is a numeric vector with `N` elements, the observation weights. If you pass `W`, the elements are normalized to sum to the prior probabilities in the respective classes.
- `COST` is a `K`-by-`K` numeric matrix of misclassification costs. For example, you can use `COST = ones(K) - eye(K)`, which means a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `ClassificationKNN.fit`. `Cost(i, j)` is the cost of classifying an observation into class `j` if its true class is `i`. By default, `Cost(i, j)=1` if `i~j`, and `Cost(i, j)=0` if `i=j`. In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have `K` classes. You place the observations into a matrix `Xnew` with one observation per row. The command

```
[label,score,cost] = predict(mdl,Xnew)
```

returns, among other outputs, a `cost` matrix of size `Nobs-by-K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n, k)` is

$$\sum_{i=1}^K \hat{P}(i | X_{new}(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X_{new}(n))$  is the posterior probability of class  $i$  for observation  $X_{new}(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

# ClassificationKNN.resubLoss

---

## Examples

### Loss Calculation

Construct a  $k$ -nearest neighbor classifier for the Fisher iris data, where  $k = 5$ .

Load the data.

```
load fisheriris
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(meas,species,'NumNeighbors',5);
```

Examine the resubstitution loss of the classifier.

```
L = resubLoss(mdl)
```

```
L =
```

```
0.0333
```

The classifier predicts incorrect classifications for 1/30 of its training data.

## See Also

[ClassificationKNN](#) | [resubEdge](#) | [resubMargin](#) | [resubPredict](#) |

## Related Examples

- “Examine the Quality of a KNN Classifier” on page 16-27
- “Predict Classification Based on a KNN Classifier” on page 16-28
- “Modify a KNN Classifier” on page 16-29

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9



## Purpose

Classification error by resubstitution

## Syntax

```
L = resubLoss(tree)
L = resubLoss(tree,Name,Value)
L = resubLoss(tree,'subtrees',subtreevector)
[L,se] = resubLoss(tree,'subtrees',subtreevector)
[L,se,NLeaf] = resubLoss(tree,'subtrees',subtreevector)
[L,se,NLeaf,bestlevel] = resubLoss(tree,'subtrees',
    subtreevector)
[L,...] =
resubLoss(tree,'subtrees',subtreevector,Name,Value)
```

## Description

`L = resubLoss(tree)` returns the resubstitution loss, meaning the loss computed for the data that `ClassificationTree.fit` used to create `tree`.

`L = resubLoss(tree,Name,Value)` returns the loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

`L = resubLoss(tree,'subtrees',subtreevector)` returns a vector of classification errors for the trees in the pruning sequence `subtreevector`.

`[L,se] = resubLoss(tree,'subtrees',subtreevector)` returns the vector of standard errors of the classification errors.

`[L,se,NLeaf] = resubLoss(tree,'subtrees',subtreevector)` returns the vector of numbers of leaf nodes in the trees of the pruning sequence.

`[L,se,NLeaf,bestlevel] = resubLoss(tree,'subtrees',subtreevector)` returns the best pruning level as defined in the `treesize` name-value pair. By default, `bestlevel` is the pruning level that gives loss within one standard deviation of minimal loss.

`[L,...] = resubLoss(tree,'subtrees',subtreevector,Name,Value)` returns loss statistics with additional options specified by one or more

# ClassificationTree.resubLoss

---

Name, Value pair arguments. You can specify several name-value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

## Input Arguments

### **tree**

A classification tree constructed by `ClassificationTree.fit`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### **'lossfun'**

Function handle or string representing a loss function. Built-in loss functions:

- 'binodeviance' — See “Loss Functions” on page 21-2988
- 'classiferror' — Fraction of misclassified observations. See “Loss Functions” on page 21-2988.
- 'exponential' — See “Loss Functions” on page 21-2988
- 'mincost' — Smallest misclassification cost as given by the `tree.Cost` matrix. See “Loss Functions” on page 21-2988.

You can write your own loss function in the syntax described in “Loss Functions” on page 21-2988.

**Default:** 'mincost'

Name, Value arguments associated with pruning subtrees:

### **'subtrees'**

A vector with integer values from 0 (full unpruned tree) to the maximal pruning level `max(tree.PruneList)`.

**Default:** 0

**'treesize'**

One of the following strings:

- 'se' — loss returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in subtrees).
- 'min' — loss returns the element of subtrees with smallest loss, usually the smallest element of subtrees.

## Output Arguments

**L**

Classification error, a vector the length of subtrees. The meaning of the error depends on the values in `weights` and `lossfun`; see “Classification Error” on page 21-2988.

**se**

Standard error of loss, a vector the length of subtrees.

**Nleaf**

Number of leaves (terminal nodes) in the pruned subtrees, a vector the length of subtrees.

**bestlevel**

A scalar whose value depends on `treesize`:

- `treesize = 'se'` — loss returns the highest pruning level with loss within one standard deviation of the minimum ( $L+se$ , where  $L$  and  $se$  relate to the smallest value in subtrees).
- `treesize = 'min'` — loss returns the element of subtrees with smallest loss, usually the smallest element of subtrees.

## Definitions

### Classification Error

The default classification error is the fraction of the training data  $X$  that `tree` misclassifies.

Weighted classification error is the sum of weight  $i$  times the Boolean value that is 1 when `tree` misclassifies the  $i$ th row of  $X$ , divided by the sum of the weights.

### Loss Functions

The built-in loss functions are:

- 'binodeviance' — For binary classification, assume the classes  $y_n$  are -1 and 1. With weight vector  $w$  normalized to have sum 1, and predictions of row  $n$  of data  $X$  as  $f(X_n)$ , the binomial deviance is

$$\sum w_n \log(1 + \exp(-2y_n f(X_n))).$$

- 'exponential' — With the same definitions as for 'binodeviance', the exponential loss is

$$\sum w_n \exp(-y_n f(X_n)).$$

- 'classiferror' — Predict the label with the largest posterior probability. The loss is then the fraction of misclassified observations.
- 'mincost' — Predict the label with the smallest expected misclassification cost, with expectation taken over the posterior probability, and cost as given by the `Cost` property of the classifier (a matrix). The loss is then the true misclassification cost averaged over the observations.

To write your own loss function, create a function file in this form:

```
function loss = lossfun(C,S,W,COST)
```

- $N$  is the number of rows of  $X$ .

- $K$  is the number of classes in the classifier, represented in the `ClassNames` property.
- $C$  is an  $N$ -by- $K$  logical matrix, with one true per row for the true class. The index for each class is its position in the `ClassNames` property.
- $S$  is an  $N$ -by- $K$  numeric matrix.  $S$  is a matrix of posterior probabilities for classes with one row per observation, similar to the posterior output from `predict`.
- $W$  is a numeric vector with  $N$  elements, the observation weights. If you pass  $W$ , the elements are normalized to sum to the prior probabilities in the respective classes.
- $COST$  is a  $K$ -by- $K$  numeric matrix of misclassification costs. For example, you can use  $COST = \text{ones}(K) - \text{eye}(K)$ , which means a cost of 0 for correct classification, and 1 for misclassification.
- The output `loss` should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## True Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you create the classifier using the `ClassificationTree.fit` method.  $Cost(i, j)$  is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default,  $Cost(i, j) = 1$  if  $i \neq j$ , and  $Cost(i, j) = 0$  if  $i = j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Expected Misclassification Cost

There are two costs associated with classification: the true misclassification cost per class, and the expected misclassification cost per observation.

# ClassificationTree.resubLoss

---

Suppose you have `Nobs` observations that you want to classify with a trained classifier. Suppose you have `K` classes. You place the observations into a matrix `Xnew` with one observation per row.

The expected cost matrix `CE` has size `Nobs`-by-`K`. Each row of `CE` contains the expected (average) cost of classifying the observation into each of the `K` classes. `CE(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | Xnew(n)) C(k | i),$$

where

- `K` is the number of classes.
- $\hat{P}(i | Xnew(n))$  is the posterior probability of class `i` for observation `Xnew(n)`.
- $C(k | i)$  is the true misclassification cost of classifying an observation as `k` when its true class is `i`.

## Examples

Compute the resubstituted classification error for the ionosphere data:

```
load ionosphere
tree = ClassificationTree.fit(X,Y);
L = resubLoss(tree)
```

```
L =
    0.0114
```

## See Also

`loss` | `resubEdge` | `resubMargin` | `resubPredict`

## Purpose

Regression error by resubstitution

## Syntax

```
L = resubLoss(ens)
L = resubLoss(ens,Name,Value)
```

## Description

`L = resubLoss(ens)` returns the resubstitution loss, meaning the mean squared error computed for the data that `fitensemble` used to create `ens`.

`L = resubLoss(ens,Name,Value)` calculates loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### **ens**

A regression ensemble created with `fitensemble`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' ' ). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### **'learners'**

Indices of weak learners in the ensemble ranging from 1 to `NTrained`. `resubLoss` uses only these learners for calculating loss.

**Default:** `1:NTrained`

### **'lossfun'**

Function handle for loss function, or the string `'mse'`, meaning mean squared error. If you pass a function handle `fun`, `resubLoss` calls it as

```
FUN(Y,Yfit,W)
```

# RegressionEnsemble.resubLoss

---

where  $Y$ ,  $Y_{fit}$ , and  $W$  are numeric vectors of the same length.  $Y$  is the observed response,  $Y_{fit}$  is the predicted response, and  $W$  is the observation weights.

**Default:** 'mse'

## 'mode'

String representing the meaning of the output  $L$ :

- 'ensemble' —  $L$  is a scalar value, the loss for the entire ensemble.
- 'individual' —  $L$  is a vector with one element per trained learner.
- 'cumulative' —  $L$  is a vector in which element  $J$  is obtained by using learners  $1:J$  from the input list of learners.

**Default:** 'ensemble'

## Output Arguments

### $L$

Loss, by default the mean squared error.  $L$  can be a vector, and can mean different things, depending on the name-value pair settings.

## Examples

Find the resubstitution predictions of mileage from the `carsmall` data based on horsepower and weight, and look at their mean square difference from the training data.

```
load carsmall
X = [Horsepower Weight];
ens = fitensemble(X,MPG,'LSBoost',100,'Tree');
MSE = resubLoss(ens)
```

```
MSE =
    6.4336
```

## See Also

[resubPredict](#) | [resubLoss](#) | [loss](#)



## Purpose

Regression error by resubstitution

## Syntax

```
L = resubLoss(tree)
L = resubLoss(tree,Name,Value)
L = resubLoss(tree,'subtrees',subtreevector)
[L,se] = resubLoss(tree,'subtrees',subtreevector)
[L,se,NLeaf] = resubLoss(tree,'subtrees',subtreevector)
[L,se,NLeaf,bestlevel] = resubLoss(tree,'subtrees',
    subtreevector)
[L,...] =
resubLoss(tree,'subtrees',subtreevector,Name,Value)
```

## Description

`L = resubLoss(tree)` returns the resubstitution loss, meaning the loss computed for the data that `RegressionTree.fit` used to create `tree`.

`L = resubLoss(tree,Name,Value)` returns the loss with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

`L = resubLoss(tree,'subtrees',subtreevector)` returns a vector of mean squared errors for the trees in the pruning sequence `subtreevector`.

`[L,se] = resubLoss(tree,'subtrees',subtreevector)` returns the vector of standard errors of the classification errors.

`[L,se,NLeaf] = resubLoss(tree,'subtrees',subtreevector)` returns the vector of numbers of leaf nodes in the trees of the pruning sequence.

`[L,se,NLeaf,bestlevel] = resubLoss(tree,'subtrees',subtreevector)` returns the best pruning level as defined in the `treesize` name-value pair. By default, `bestlevel` is the pruning level that gives loss within one standard deviation of minimal loss.

`[L,...] = resubLoss(tree,'subtrees',subtreevector,Name,Value)` returns loss statistics with additional options specified by one or more

# RegressionTree.resubLoss

---

Name, Value pair arguments. You can specify several name-value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

## Input Arguments

### **tree**

A regression tree constructed by RegressionTree.fit.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

### **'lossfun'**

Function handle, or the string 'mse' meaning mean squared error.

You can write your own loss function in the syntax described in “Loss Functions” on page 21-2995.

**Default:** 'mse'

Name, Value arguments associated with pruning subtrees:

### **'subtrees'**

A vector with integer values from 0 (full unpruned tree) to the maximal pruning level max(tree.PruneList).

**Default:** 0

### **'treesize'**

One of the following strings:

- 'se' — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L + se$ , where  $L$  and  $se$  relate to the smallest value in subtrees).
- 'min' — `loss` returns the element of subtrees with smallest loss, usually the smallest element of subtrees.

## Output Arguments

**L**

Mean squared error, a vector the length of subtrees. The meaning of the error depends on the values in `weights` and `lossfun`.

**se**

Standard error of loss, a vector the length of subtrees.

**Nleaf**

Number of leaves (terminal nodes) in the pruned subtrees, a vector the length of subtrees.

**bestlevel**

A scalar whose value depends on `treesize`:

- `treesize = 'se'` — `loss` returns the highest pruning level with loss within one standard deviation of the minimum ( $L + se$ , where  $L$  and  $se$  relate to the smallest value in subtrees).
- `treesize = 'min'` — `loss` returns the element of subtrees with smallest loss, usually the smallest element of subtrees.

## Definitions

### Loss Functions

The built-in loss function is 'mse', meaning mean squared error.

To write your own loss function, create a function file of the form

```
function loss = lossfun(Y,Yfit,W)
```

- $N$  is the number of rows of `tree.X`.
- $Y$  is an  $N$ -element vector representing the observed response.

# RegressionTree.resubLoss

---

- Yfit is an N-element vector representing the predicted responses.
- W is an N-element vector representing the observation weights.
- The output loss should be a scalar.

Pass the function handle `@lossfun` as the value of the `lossfun` name-value pair.

## Examples

Find the mean square error of a model of the `carsmall` data:

```
load carsmall
X = [Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
resubLoss(tree)
```

```
ans =
    4.8952
```

## See Also

`resubPredict` | `loss`

# ClassificationDiscriminant.resubMargin

<b>Purpose</b>	Classification margins by resubstitution
<b>Syntax</b>	<code>M = resubMargin(obj)</code>
<b>Description</b>	<code>M = resubMargin(obj)</code> returns resubstitution classification margins for <code>obj</code> .
<b>Input Arguments</b>	<b>obj</b> Discriminant analysis classifier, produced using <code>ClassificationDiscriminant.fit</code> .
<b>Output Arguments</b>	<b>M</b> Numeric column-vector of length <code>size(obj.X,1)</code> containing the classification margins.
<b>Definitions</b>	<b>Margin</b> The classification <i>margin</i> is the difference between the classification <i>score</i> for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix <code>obj.X</code> . A high value of margin indicates a more reliable prediction than a low value. <b>Score (discriminant analysis)</b> For discriminant analysis, the <i>score</i> of a classification is the posterior probability of the classification. For the definition of posterior probability in discriminant analysis, see “Posterior Probability” on page 15-7.
<b>Examples</b>	Find the margins for a classification tree for the Fisher iris data by resubstitution. Examine several entries: <pre>load fisheriris obj = ClassificationDiscriminant.fit(meas,species); M = resubMargin(obj); M(1:25:end)</pre>

# ClassificationDiscriminant.resubMargin

---

```
ans =  
  1.0000  
  1.0000  
  0.9998  
  0.9998  
  1.0000  
  0.9946
```

**See Also**      ClassificationDiscriminant | margin

**How To**        • “Discriminant Analysis” on page 15-3

<b>Purpose</b>	Classification margins by resubstitution
<b>Syntax</b>	<pre>margin = resubMargin(ens) margin = resubMargin(ens,Name,Value)</pre>
<b>Description</b>	<p><code>margin = resubMargin(ens)</code> returns the classification margin obtained by <code>ens</code> on its training data.</p> <p><code>margin = resubMargin(ens,Name,Value)</code> calculates margins with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Input Arguments</b>	<p><b>ens</b></p> <p>A classification ensemble created with <code>fitensemble</code>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>'learners'</b></p> <p>Indices of weak learners in the ensemble ranging from 1 to <code>NTrained</code>. <code>resubMargin</code> uses only these learners for calculating margin.</p> <p><b>Default:</b> <code>1:NTrained</code></p>
<b>Output Arguments</b>	<p><b>margin</b></p> <p>A numeric column-vector of length <code>size(ens.X,1)</code> containing the classification margins.</p>

# ClassificationEnsemble.resubMargin

---

## Definitions

### Margin

The classification *margin* is the difference between the classification *score* for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the matrix `ens.X`.

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## Examples

Find the resubstitution margins for an ensemble that classifies the Fisher iris data:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
margin = resubMargin(ens);
[min(margin) mean(margin) max(margin)]
```

```
ans =
    -0.5674     3.2486     4.6245
```

## See Also

[resubEdge](#) | [resubLoss](#) | [resubPredict](#) | [resubMargin](#)



<b>Purpose</b>	Margin of $k$ -nearest neighbor classifier by resubstitution
<b>Syntax</b>	<code>m = resubMargin mdl</code>
<b>Description</b>	<code>m = resubMargin mdl</code> returns the classification margins of the data used to train <code>mdl</code> . For the definition, see “Margin” on page 21-3001.
<b>Input Arguments</b>	<b>mdl</b> $k$ -nearest neighbor classifier, created by <code>ClassificationKNN.fit</code> .
<b>Output Arguments</b>	<b>m</b> A numeric column vector of length <code>size(mdl.X,1)</code> , where <code>mdl.X</code> is the training data for <code>mdl</code> . Each entry in <code>m</code> represents the margin for the corresponding row of <code>mdl.X</code> and (true class) <code>mdl.Y</code> .
<b>Definitions</b>	<b>Margin</b> The classification <i>margin</i> is the difference between the classification <i>score</i> for the true class and maximal classification score for the false classes. Margin is a column vector with the same number of rows as in the training data. <b>Score</b> The <i>score</i> of a classification is the posterior probability of the classification. The posterior probability is the number of neighbors that have that classification, divided by the number of neighbors. For a more detailed definition that includes weights and prior probabilities, see “Posterior Probability” on page 21-2608.
<b>Examples</b>	<b>Resubstitution Margin Calculation</b> Construct a $k$ -nearest neighbor classifier for the Fisher iris data, where $k = 5$ . Load the data.

# ClassificationKNN.resubMargin

---

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',5);
```

Examine some statistics of the resubstitution margin of the classifier.

```
m = resubMargin(mdl);
[max(m) min(m) mean(m)]
```

```
ans =
```

```
1.0000 -0.6000 0.9253
```

The mean margin is over 0.9, indicating fairly high classification accuracy for resubstitution. For more reliable assessment of model accuracy, consider cross validation, such as `kfoldLoss`.

## See Also

[ClassificationKNN](#) | [resubEdge](#) | [resubLoss](#) | [resubPredict](#) |

## Related Examples

- “Examine the Quality of a KNN Classifier” on page 16-27

## Concepts

- “Classification Using Nearest Neighbors” on page 16-9

<b>Purpose</b>	Classification margins by resubstitution
<b>Syntax</b>	<code>M = resubMargin(tree)</code>
<b>Description</b>	<code>M = resubMargin(tree)</code> returns resubstitution classification margins for <code>tree</code> .
<b>Input Arguments</b>	<b>tree</b> A classification tree created by <code>ClassificationTree.fit</code> .
<b>Output Arguments</b>	<b>M</b> A numeric column-vector of length <code>size(tree.X,1)</code> containing the classification margins.

## Definitions

### Margin

Classification *margin* is the difference between classification *score* for the true class and maximal classification score for the false classes. A high value of margin indicates a more reliable prediction than a low value.

### Score (tree)

For trees, the *score* of a classification of a leaf node is the posterior probability of the classification at that node. The posterior probability of the classification at a node is the number of training sequences that lead to that node with the classification, divided by the number of training sequences that lead to that node.

For example, consider classifying a predictor  $X$  as true when  $X < 0.15$  or  $X > 0.95$ , and  $X$  is false otherwise.

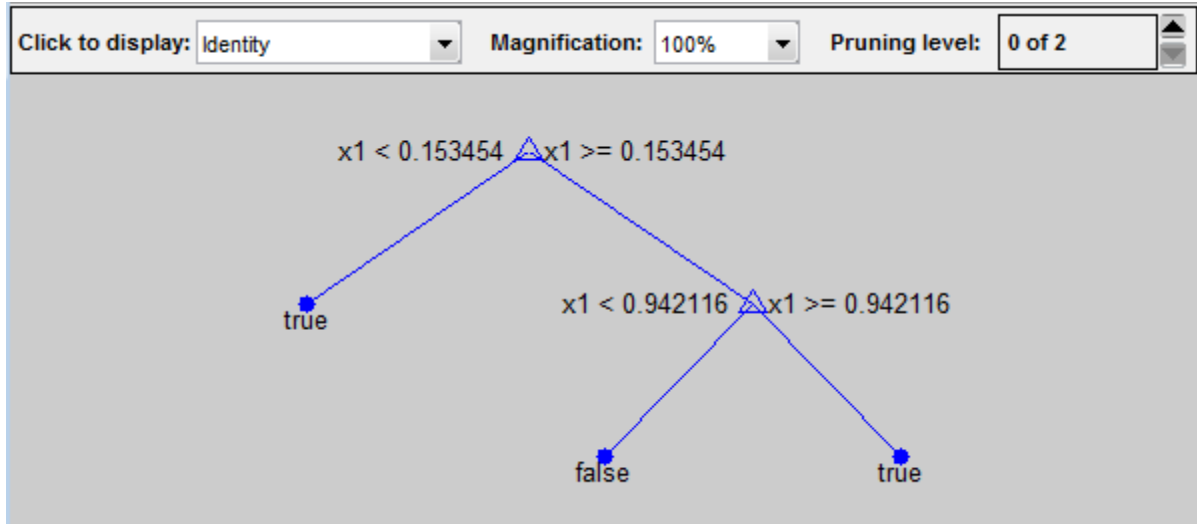
1

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
```

# ClassificationTree.resubMargin

```
Y = (abs(X - .55) > .4);  
tree = ClassificationTree.fit(X,Y);  
view(tree,'mode','graph')
```

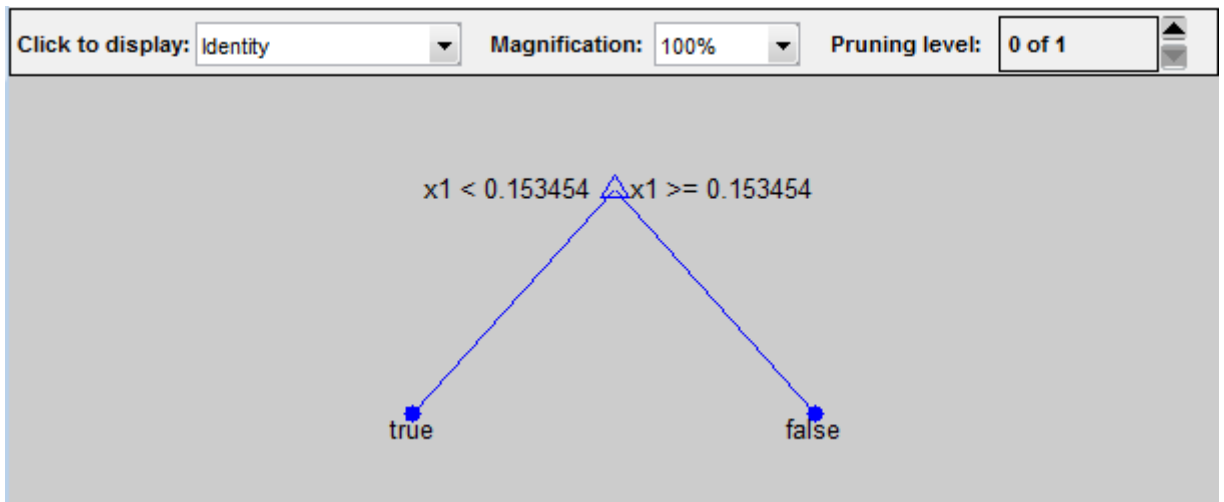


2

Prune the tree:

```
tree1 = prune(tree,'level',1);  
view(tree1,'mode','graph')
```

# ClassificationTree.resubMargin



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations from .15 to .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore, the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));
[score X(1:10,:)]
```

```
ans =
    0.9059    0.0941    0.8147
    0.9059    0.0941    0.9058
         0    1.0000    0.1270
    0.9059    0.0941    0.9134
    0.9059    0.0941    0.6324
         0    1.0000    0.0975
    0.9059    0.0941    0.2785
```

# ClassificationTree.resubMargin

---

```
0.9059    0.0941    0.5469
0.9059    0.0941    0.9575
0.9059    0.0941    0.9649
```

Indeed, every value of  $X$  (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of  $X$  have associated scores of 0.91 and 0.09. The difference (score 0.09 instead of the expected .06) is due to a statistical fluctuation: there are 8 observations in  $X$  in the range  $(.95, 1)$  instead of the expected 5 observations.

## Examples

Find the margins for a classification tree for the Fisher iris data by resubstitution. Examine several entries:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
M = resubMargin(tree);
M(1:25:end)

ans =
    1.0000
    1.0000
    1.0000
    1.0000
    0.9565
    0.9565
```

## See Also

[margin](#) | [resubLoss](#) | [resubPredict](#) | [resubEdge](#)

# ClassificationDiscriminant.resubPredict

---

<b>Purpose</b>	Predict resubstitution response of classifier
<b>Syntax</b>	<pre>label = resubPredict(obj) [label,posterior] = resubPredict(obj) [label,posterior,cost] = resubPredict(obj)</pre>
<b>Description</b>	<p>label = resubPredict(obj) returns the labels obj predicts for the data obj.X. label is the predictions of obj on the data that ClassificationDiscriminant.fit used to create obj.</p> <p>[label,posterior] = resubPredict(obj) returns the posterior class probabilities for the predictions.</p> <p>[label,posterior,cost] = resubPredict(obj) returns the predicted misclassification costs per class for the resubstituted data.</p>
<b>Input Arguments</b>	<p><b>obj</b></p> <p>Discriminant analysis classifier, produced using ClassificationDiscriminant.fit.</p>
<b>Output Arguments</b>	<p><b>label</b></p> <p>Response obj predicts for the training data. label is the same data type as the training response data obj.Y. The predicted class labels are those with minimal expected misclassification cost; see “How the predict Method Classifies” on page 15-6.</p> <p><b>posterior</b></p> <p>N-by-K matrix of posterior probabilities for classes obj predicts, where N is the number of observations and K is the number of classes.</p> <p><b>cost</b></p> <p>N-by-K matrix of predicted misclassification costs. Each cost is the average misclassification cost with respect to the posterior probability.</p>

# ClassificationDiscriminant.resubPredict

---

## Definitions

### Posterior Probability

`posterior(i,k)` is the posterior probability of class `k` for observation `i`. For the mathematical definition, see “Posterior Probability” on page 15-7.

## Examples

Find the total number of misclassifications of the Fisher iris data for a discriminant analysis classifier:

```
load fisheriris
obj = ClassificationDiscriminant.fit(meas,species);
Ypredict = resubPredict(obj); % the predictions
Ysame = strcmp(Ypredict,species); % true when ==
sum(~Ysame) % how many are different?

ans =
     3
```

## See Also

`ClassificationDiscriminant` | `predict`

## How To

- “Discriminant Analysis” on page 15-3



## Purpose

Predict ensemble response by resubstitution

## Syntax

```
label = resubPredict(ens)
[label,score] = resubPredict(ens)
[label,score] = resubPredict(ens,Name,Value)
```

## Description

`label = resubPredict(ens)` returns the labels `ens` predicts for the data `ens.X`. `label` is the predictions of `ens` on the data that `fitensemble` used to create `ens`.

`[label,score] = resubPredict(ens)` also returns scores for all classes.

`[label,score] = resubPredict(ens,Name,Value)` finds resubstitution predictions with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A classification ensemble created with `fitensemble`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'learners'**

Indices of weak learners in the ensemble ranging from 1 to `NTrained`. `oobLoss` uses only these learners for calculating loss.

**Default:** 1:NTrained

# ClassificationEnsemble.resubPredict

---

## Output Arguments

### label

The response `ens` predicts for the training data. `label` is the same data type as the training response data `ens.Y`, and has the same number of entries as the number of rows in `ens.X`.

### score

An N-by-K matrix, where N is the number of rows in `ens.X`, and K is the number of classes in `ens`. High score value indicates that an observation likely comes from this class.

## Definitions

### Score (ensemble)

For ensembles, a classification *score* represents the confidence of a classification into a class. The higher the score, the higher the confidence.

Different ensemble algorithms have different definitions for their scores. Furthermore, the range of scores depends on ensemble type. For example:

- AdaBoostM1 scores range from  $-\infty$  to  $\infty$ .
- Bag scores range from 0 to 1.

## Examples

Find the total number of misclassifications of the Fisher iris data for a classification ensemble:

```
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,'Tree');
Ypredict = resubPredict(ens); % the predictions
Ysame = strcmp(Ypredict,species); % true when ==
sum(~Ysame) % how many are different?

ans =
    5
```

## See Also

[resubEdge](#) | [resubMargin](#) | [resubLoss](#) | [resubPredict](#)

<b>Purpose</b>	Predict resubstitution response of $k$ -nearest neighbor classifier
<b>Syntax</b>	<pre>label = resubPredict mdl [label,score] = resubPredict mdl [label,score,cost] = resubPredict mdl</pre>
<b>Description</b>	<p>label = resubPredict(mdl) returns the labels mdl predicts for the data mdl.X. label is the predictions of mdl on the data that ClassificationKNN.fit used to create mdl.</p> <p>[label,score] = resubPredict(mdl) returns the posterior class probabilities for the predictions.</p> <p>[label,score,cost] = resubPredict(mdl) returns the misclassification costs.</p>
<b>Input Arguments</b>	<p><b>mdl</b></p> <p><math>k</math>-nearest neighbor classifier, created by ClassificationKNN.fit.</p>
<b>Output Arguments</b>	<p><b>label</b></p> <p>Predicted class labels for the points in the training data X, a vector with length equal to the number of rows in the training data X. The label is the class with minimal expected cost (see “Expected Cost” on page 21-3012).</p> <p><b>score</b></p> <p>Numeric matrix of size N-by-K, where N is the number of observations (rows) in the training data X, and K is the number of classes (in mdl.ClassNames). score(i,j) is the posterior probability that row i of X is of class j. See “Posterior Probability” on page 21-3012.</p> <p><b>cost</b></p> <p>Matrix of expected costs of size N-by-K, where N is the number of observations (rows) in the training data X, and K is the number of</p>

# ClassificationKNN.resubPredict

---

classes (in `mdl.ClassNames`). `cost(i, j)` is the cost of classifying row `i` of `X` as class `j`. See “Expected Cost” on page 21-3012.

## Definitions Posterior Probability

For a vector (single query point) `X` and model `mdl`, let

- `K` be the number of nearest neighbors used in prediction, `mdl.NumNeighbors`
- `nbd(mdl, X)` be the `K` nearest neighbors to `X` in `mdl.X`
- `Y(nbd)` be the classifications of the points in `nbd(mdl, X)`, namely `mdl.Y(nbd)`
- `W(nbd)` be the weights of the points in `nbd(mdl, X)`
- `prior` be the priors of the classes in `mdl.Y`

If there is a vector of prior probabilities, then the observation weights `W` are normalized by class to sum to the priors. This might involve a calculation for the point `X`, because weights can depend on the distance from `X` to the points in `mdl.X`.

The posterior probability  $p(j | X)$  is

$$p(j | X) = \frac{\sum_{i \in \text{nbd}} W(i) 1_{Y(X(i)=j)}}{\sum_{i \in \text{nbd}} W(i)}.$$

Here  $1_{Y(X(i)=j)}$  means 1 when `mdl.Y(i) = j`, and 0 otherwise.

## Expected Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation. The third output of `predict` is the expected misclassification cost per observation.

Suppose you have `Nobs` observations that you want to classify with a trained classifier `mdl`. Suppose you have `K` classes. You place the observations into a matrix `X` with one observation per row. The command

```
[label,score,cost] = predict(mdl,X)
```

returns, among other outputs, a `cost` matrix of size `Nobs`-by-`K`. Each row of the `cost` matrix contains the expected (average) cost of classifying the observation into each of the `K` classes. `cost(n,k)` is

$$\sum_{i=1}^K \hat{P}(i | X(n)) C(k | i),$$

where

- $K$  is the number of classes.
- $\hat{P}(i | X(n))$  is the posterior probability of class  $i$  for observation  $X(n)$ .
- $C(k | i)$  is the true misclassification cost of classifying an observation as  $k$  when its true class is  $i$ .

## True Misclassification Cost

There are two costs associated with KNN classification: the true misclassification cost per class, and the expected misclassification cost per observation.

You can set the true misclassification cost per class in the `Cost` name-value pair when you run `ClassificationKNN.fit`. `Cost(i,j)` is the cost of classifying an observation into class  $j$  if its true class is  $i$ . By default, `Cost(i,j)=1` if  $i \neq j$ , and `Cost(i,j)=0` if  $i=j$ . In other words, the cost is 0 for correct classification, and 1 for incorrect classification.

## Examples

### Resubstitution Prediction

Examine the quality of a classifier by its resubstitution predictions.

# ClassificationKNN.resubPredict

---

Load the data.

```
load fisheriris
X = meas;
Y = species;
```

Construct a classifier for 5-nearest neighbors.

```
mdl = ClassificationKNN.fit(X,Y,'NumNeighbors',5);
```

Generate the resubstitution predictions.

```
label = resubPredict(mdl);
```

Calculate the number of differences between the predictions label and the original data Y.

```
mydiff = not(strcmp(Y,label)); % mydiff(i) = 1 means they differ
sum(mydiff) % number of differences

ans =
     5
```

## See Also

[ClassificationKNN](#) | [predict](#) | [resubEdge](#) | [resubLoss](#) | [resubMargin](#) |

**Purpose** Predict resubstitution response of tree

**Syntax**

```
label = resubPredict(tree)
[label,posterior] = resubPredict(tree)
[label,posterior,node] = resubPredict(tree)
[label,posterior,node,cnum] = resubPredict(tree)
[label,...] = resubPredict(tree,Name,Value)
```

**Description**

label = resubPredict(tree) returns the labels tree predicts for the data tree.X. label is the predictions of tree on the data that ClassificationTree.fit used to create tree.

[label,posterior] = resubPredict(tree) returns the posterior class probabilities for the predictions.

[label,posterior,node] = resubPredict(tree) returns the node numbers of tree for the resubstituted data.

[label,posterior,node,cnum] = resubPredict(tree) returns the predicted class numbers for the predictions.

[label,...] = resubPredict(tree,Name,Value) returns resubstitution predictions with additional options specified by one or more Name,Value pair arguments.

## Input Arguments

### tree

A classification tree constructed by ClassificationTree.fit.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

### 'subtrees'

# ClassificationTree.resubPredict

---

A vector with integer values from 0 (full unpruned tree) to the maximal pruning level `max(tree.PruneList)`. `subtrees` must be in ascending order.

**Default:** 0

## Output Arguments

### label

The response `tree` predicts for the training data. `label` is the same data type as the training response data `tree.Y`.

If the `subtrees` name-value argument contains  $m > 1$  entries, `label` has  $m$  columns, each of which represents the predictions of the corresponding subtree. Otherwise, `label` is a vector.

### posterior

Matrix or array of posterior probabilities for classes `tree` predicts.

If the `subtrees` name-value argument is a scalar or is missing, `posterior` is an  $n$ -by- $k$  matrix, where  $n$  is the number of rows in the training data `tree.X`, and  $k$  is the number of classes.

If `subtrees` contains  $m > 1$  entries, `posterior` is an  $n$ -by- $k$ -by- $m$  array, where the matrix for each  $m$  gives posterior probabilities for the corresponding subtree.

### node

The node numbers of `tree` where each data row resolves.

If the `subtrees` name-value argument is a scalar or is missing, `node` is a numeric column vector with  $n$  rows, the same number of rows as `tree.X`.

If `subtrees` contains  $m > 1$  entries, `node` is a  $n$ -by- $m$  matrix. Each column represents the node predictions of the corresponding subtree.

### cnum

The class numbers that `tree` predicts for the resubstituted data.



If the `subtrees` name-value argument is a scalar or is missing, `cnum` is a numeric column vector with `n` rows, the same number of rows as `tree.X`.

If `subtrees` contains `m>1` entries, `cnum` is a `n-by-m` matrix. Each column represents the class predictions of the corresponding subtree.

## Definitions

### Posterior Probability

The posterior probability of the classification at a node is the number of training sequences that lead to that node with this classification, divided by the number of training sequences that lead to that node.

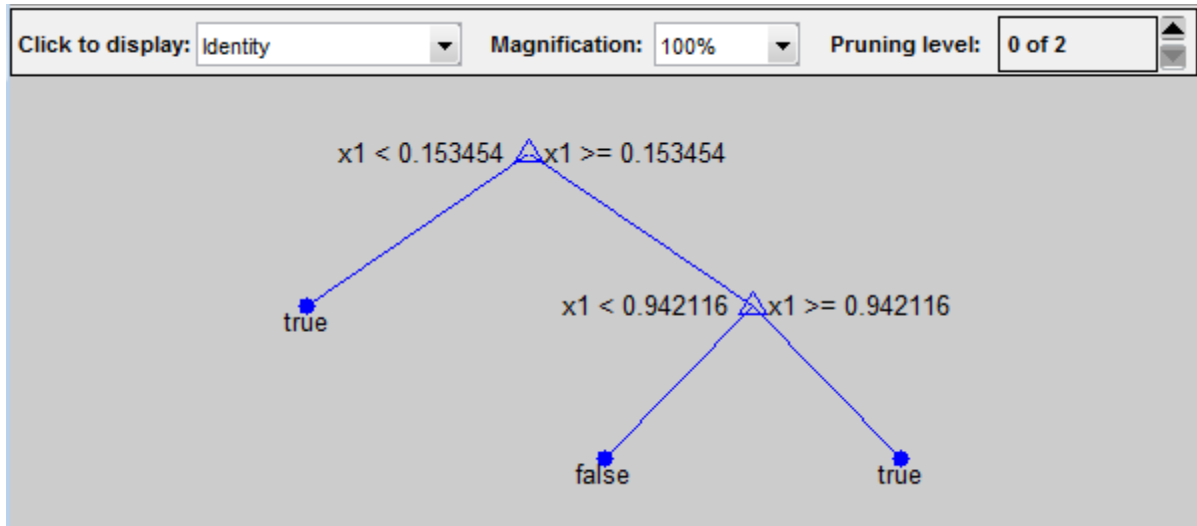
For example, consider classifying a predictor `X` as `true` when `X<0.15` or `X>0.95`, and `X` is `false` otherwise.

1

Generate 100 random points and classify them:

```
rng(0,'twister') % for reproducibility
X = rand(100,1);
Y = (abs(X - .55) > .4);
tree = ClassificationTree.fit(X,Y);
view(tree,'mode','graph')
```

# ClassificationTree.resubPredict

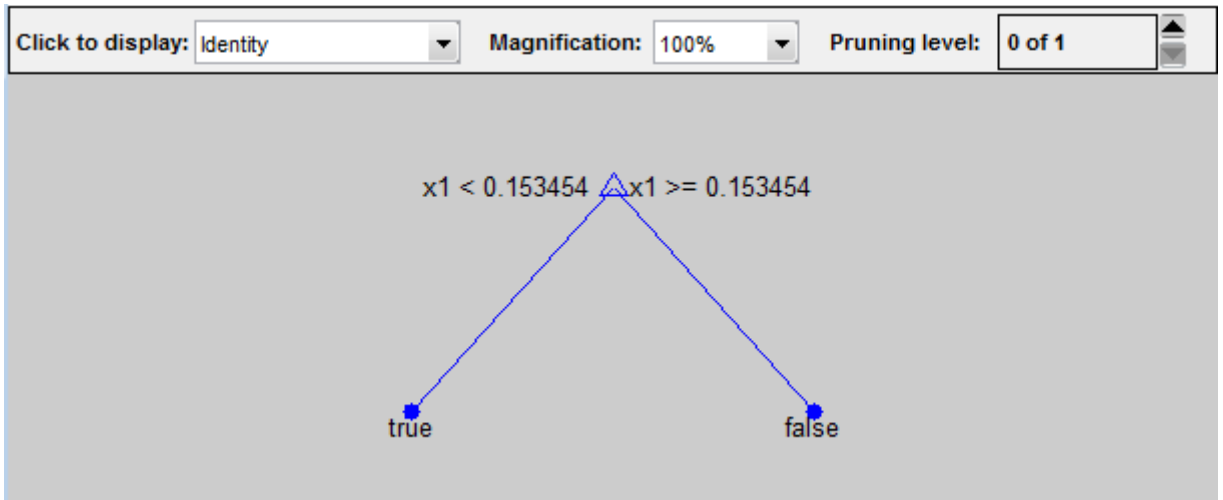


**2**

Prune the tree:

```
tree1 = prune(tree, 'level', 1);  
view(tree1, 'mode', 'graph')
```

# ClassificationTree.resubPredict



The pruned tree correctly classifies observations that are less than 0.15 as true. It also correctly classifies observations between .15 and .94 as false. However, it incorrectly classifies observations that are greater than .94 as false. Therefore the score for observations that are greater than .15 should be about  $.05/.85=.06$  for true, and about  $.8/.85=.94$  for false.

### 3

Compute the prediction scores for the first 10 rows of X:

```
[~,score] = predict(tree1,X(1:10));  
[score X(1:10,:)]
```

```
ans =  
    0.9405    0.0595    0.6555  
    0.9405    0.0595    0.1712  
    0.9405    0.0595    0.7060  
         0    1.0000    0.0318  
    0.9405    0.0595    0.2769  
         0    1.0000    0.0462
```

# ClassificationTree.resubPredict

---

0	1.0000	0.0971
0.9405	0.0595	0.8235
0.9405	0.0595	0.6948
0.9405	0.0595	0.3171

Indeed, every value of X (the rightmost column) that is less than 0.15 has associated scores (the left and center columns) of 0 and 1, while the other values of X have associated scores of 0.94 and 0.06.

## Examples

Find the total number of misclassifications of the Fisher iris data for a classification tree:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
Ypredict = resubPredict(tree); % the predictions
Ysame = strcmp(Ypredict,species); % true when ==
sum(~Ysame) % how many are different?
```

```
ans =
     3
```

## See Also

[resubEdge](#) | [resubMargin](#) | [resubLoss](#) | [predict](#)

<b>Purpose</b>	Predict response of ensemble by resubstitution
<b>Syntax</b>	<pre>Yfit = resubPredict(ens) Yfit = resubPredict(ens,Name,Value)</pre>
<b>Description</b>	<p><code>Yfit = resubPredict(ens)</code> returns the response <code>ens</code> predicts for the data <code>ens.X</code>. <code>Yfit</code> is the predictions of <code>ens</code> on the data that <code>fitensemble</code> used to create <code>ens</code>.</p> <p><code>Yfit = resubPredict(ens,Name,Value)</code> predicts responses with additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Input Arguments</b>	<p><b>ens</b></p> <p>A regression ensemble created with <code>fitensemble</code>.</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>'learners'</b></p> <p>Indices of weak learners in the ensemble ranging from 1 to <code>NTrained</code>. <code>oobLoss</code> uses only these learners for calculating loss.</p> <p><b>Default:</b> <code>1:NTrained</code></p>
<b>Output Arguments</b>	<p><b>Yfit</b></p> <p>A vector of predicted responses to the training data, with <code>ens.X</code> elements.</p>

# RegressionEnsemble.resubPredict

---

## Examples

Find the resubstitution predictions of mileage from the `carsmall` data based on horsepower and weight, and look at their mean square difference from the training data.

```
load carsmall
X = [Horsepower Weight];
ens = fitensemble(X,MPG,'LSBoost',100,'Tree');
Yfit = resubPredict(ens);
MSE = mean((Yfit - ens.Y).^2)
```

```
MSE =
    6.4336
```

This is the same as the result of `resubLoss`:

```
resubLoss(ens)

ans =
    6.4336
```

## See Also

[resubLoss](#) | [resubPredict](#) | [predict](#)

## Purpose

Predict resubstitution response of tree

## Syntax

```
Yfit = resubPredict(tree)
[Yfit,node] = resubPredict(tree)
[Yfit,node] = resubPredict(tree,Name,Value)
```

## Description

`Yfit = resubPredict(tree)` returns the responses tree predicts for the data `tree.X`. `Yfit` is the predictions of tree on the data that `RegressionTree.fit` used to create tree.

`[Yfit,node] = resubPredict(tree)` returns the node numbers of tree for the resubstituted data.

`[Yfit,node] = resubPredict(tree,Name,Value)` predicts with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **tree**

A regression tree constructed by `RegressionTree.fit`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'subtrees'**

A vector with integer values from 0 (full unpruned tree) to the maximal pruning level `max(tree.PruneList)`. `subtrees` must be in ascending order.

**Default:** 0

## Output Arguments

### **Yfit**

The response tree predicts for the training data.

# RegressionTree.resubPredict

---

If the `subtrees` name-value argument is a scalar or is missing, `label` is the same data type as the training response data `tree.Y`.

If `subtrees` contains  $m > 1$  entries, `label` has  $m$  columns, each of which represents the predictions of the corresponding subtree.

## **node**

The tree node numbers where `tree` sends each data row.

If the `subtrees` name-value argument is a scalar or is missing, `node` is a numeric column vector with  $n$  rows, the same number of rows as `tree.X`.

If `subtrees` contains  $m > 1$  entries, `node` is a  $n$ -by- $m$  matrix. Each column represents the node predictions of the corresponding subtree.

## **Examples**

Find the mean square error of a model of the `carsmall` data:

```
load carsmall
X = [Displacement Horsepower Weight];
tree = RegressionTree.fit(X,MPG);
Yfit = resubPredict(tree);
mean((Yfit - tree.Y).^2)
```

```
ans =
    4.8952
```

You can get the same answer using `resubLoss`:

```
resubLoss(tree)

ans =
    4.8952
```

## **See Also**

`resubLoss` | `predict`



**Purpose** Resume training ensemble

**Syntax**  
`ens1 = resume(ens,nlearn)`  
`ens1 = resume(ens,nlearn,Name,Value)`

**Description** `ens1 = resume(ens,nlearn)` trains `ens` for `nlearn` more cycles. `resume` uses the same training options `fitensemble` used to create `ens`.

---

**Note** You cannot resume training when `ens` is a Subspace ensemble created with 'AllPredictorCombinations' number of learners.

---

`ens1 = resume(ens,nlearn,Name,Value)` trains `ens` with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A classification ensemble, created with `fitensemble`.

### **nlearn**

A positive integer, the number of cycles for additional training of `ens`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'nprint'**

Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak learners trained so far. Useful when you train ensembles with many learners on large data sets.

# ClassificationEnsemble.resume

---

**Default:** 'off'

## Output Arguments

**ens1**

The classification ensemble `ens`, augmented with additional training.

## Examples

Train a classification ensemble for 10 cycles. Examine the resubstitution error. Then train for 10 more cycles and examine the new resubstitution error.

```
load ionosphere
ens = fitensemble(X,Y,'GentleBoost',10,'Tree');
L = resubLoss(ens)
```

```
L =
    0.0484
```

```
ens1 = resume(ens,10);
L = resubLoss(ens1)
```

```
L =
    0.0256
```

The new ensemble has much less resubstitution error than the original.

## See Also

`fitensemble`

## Purpose

Resume training learners on cross-validation folds

## Syntax

```
ens1 = resume(ens,nlearn)
ens1 = resume(ens,nlearn,Name,Value)
```

## Description

`ens1 = resume(ens,nlearn)` trains `ens` in every fold for `nlearn` more cycles. `resume` uses the same training options `fitensemble` used to create `ens`.

`ens1 = resume(ens,nlearn,Name,Value)` trains `ens` with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A cross-validated classification ensemble. `ens` is the result of either:

- The `fitensemble` function with a cross-validation name-value pair. The names are 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.
- The `crossval` method applied to a classification ensemble.

### **nlearn**

A positive integer, the number of cycles for additional training of `ens`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'nprint'**

Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak

# ClassificationPartitionedEnsemble.resume

---

learners trained so far. Useful when you train ensembles with many learners on large data sets.

**Default:** 'off'

## Output Arguments

**ens1**

The cross-validated classification ensemble `ens`, augmented with additional training.

## Examples

Train a partitioned classification ensemble for 10 cycles. Examine the error. Then train for 10 more cycles and examine the new error.

```
load ionosphere
cvens = fitensemble(X,Y,'GentleBoost',10,'Tree',...
    'crossval','on');
L = kfoldLoss(cvens)
```

```
L =
    0.0883
```

```
cvens = resume(cvens,10);
L = kfoldLoss(cvens)
```

```
L =
    0.0769
```

The ensemble has less cross-validation error after training for ten more cycles.

## See Also

[kfoldPredict](#) | [kfoldEdge](#) | [kfoldMargin](#) | [kfoldLoss](#)

**Purpose** Resume training ensemble

**Syntax**  
`ens1 = resume(ens,nlearn)`  
`ens1 = resume(ens,nlearn,Name,Value)`

**Description**  
`ens1 = resume(ens,nlearn)` trains `ens` for `nlearn` more cycles.  
`resume` uses the same training options `fitensemble` used to create `ens`.  
`ens1 = resume(ens,nlearn,Name,Value)` trains `ens` with additional options specified by one or more `Name,Value` pair arguments.

**Input Arguments**  
**ens**  
A regression ensemble, created with `fitensemble`.

**nlearn**  
A positive integer, the number of cycles for additional training of `ens`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'nprint'**  
Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak learners trained so far. Useful when you train ensembles with many learners on large data sets.

**Default:** 'off'

# RegressionEnsemble.resume

---

## Output Arguments

**ens1**

The regression ensemble `ens`, augmented with additional training.

## Examples

Train a regression ensemble for 50 cycles. Examine the resubstitution error. Then train for 50 more cycles and examine the new resubstitution error.

```
load carsmall
X = [Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'LSBoost',50,'Tree');
L = resubLoss(ens)
```

```
L =
    6.2681
```

```
ens = resume(ens,50);
L = resubLoss(ens)
```

```
L =
    4.3904
```

The new ensemble has much less resubstitution error than the original.

## See Also

`fitensemble`

# RegressionPartitionedEnsemble.resume

---

## Purpose

Resume training ensemble

## Syntax

```
ens1 = resume(ens,nlearn)
ens1 = resume(ens,nlearn,Name,Value)
```

## Description

`ens1 = resume(ens,nlearn)` trains `ens` in every fold for `nlearn` more cycles. `resume` uses the same training options `fitensemble` used to create `ens`.

`ens1 = resume(ens,nlearn,Name,Value)` trains `ens` with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **ens**

A cross-validated regression ensemble. `ens` is the result of either:

- The `fitensemble` function with a cross-validation name-value pair. The names are 'crossval', 'kfold', 'holdout', 'leaveout', or 'cvpartition'.
- The `crossval` method applied to a regression ensemble.

### **nlearn**

A positive integer, the number of cycles for additional training of `ens`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'nprint'**

Printout frequency, a positive integer scalar or 'off' (no printouts). Returns to the command line the number of weak

# RegressionPartitionedEnsemble.resume

---

learners trained so far. Useful when you train ensembles with many learners on large data sets.

**Default:** 'off'

## Output Arguments

**ens1**

The cross-validated regression ensemble `ens`, augmented with additional training.

## Examples

Train a regression ensemble for 50 cycles, and cross validate it. Examine the cross-validation error. Then train for 50 more cycles and examine the new cross-validation error.

```
load carsmall
X = [Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'LSBoost',50,'Tree');
cvens = crossval(ens);
L = kfoldLoss(cvens)
```

```
L =
    25.6573
```

```
cvens = resume(cvens,50);
L = kfoldLoss(cvens)
```

```
L =
    26.7563
```

The additional training did not improve the cross-validation error.

## See Also

`fitensemble` | `kfoldLoss`



**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Rician probability distribution object

**Description** `prob.RicianDistribution` is an object consisting of parameters, a model description, and sample data for a Rician probability distribution. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Rician')` creates a Rician probability distribution object using the default parameter values.

`pd = makedist('Rician','s',s,'sigma',sigma)` creates a Rician probability distribution object using the specified parameter values.

### Input Arguments

#### **s - Noncentrality parameter**

1 (default) | nonnegative scalar value

Noncentrality parameter for the Rician distribution, specified as a nonnegative scalar value.

#### **Data Types**

single | double

#### **sigma - scale parameter**

1 (default) | positive scalar value

Scale parameter for the Rician distribution, specified as a positive scalar value.

#### **Data Types**

single | double

**Properties** **s**

Noncentrality parameter of the Rician distribution, stored as a nonnegative scalar value.

# prob.RicianDistribution

---

## Data Types

single | double

## sigma

Scale parameter for the Rician distribution, stored as a positive scalar value.

## Data Types

single | double

## DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

## InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## **NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## **Data Types**

single | double

## **ParameterCovariance**

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## **Data Types**

single | double

## **ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## **Data Types**

char

## **ParameterIsFixed**

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## **Data Types**

logical

## **ParameterNames**

# prob.RicianDistribution

---

Names of distribution parameters, stored as a cell array of strings.  
This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries.  
This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object

random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Rician Distribution

The Rician distribution is used in communications theory to model scattered signals that reach a receiver using multiple paths.

The Rician distribution uses the following parameters.

Name	Description	Support
s	Noncentrality parameter	$s \geq 0$
sigma	Scale parameter	$\sigma > 0$

The probability density function (pdf) is

# prob.RicianDistribution

---

$$f(x|s,\sigma) = I_0\left(\frac{xs}{\sigma^2}\right)\left(\frac{x}{\sigma^2}\right)\exp\left\{-\frac{x^2+s^2}{2\sigma^2}\right\} ; \quad x \geq 0,$$

where  $I_0$  is the zero-order modified Bessel function of the first kind.

## Examples

### Create a Rician Distribution Object Using Default Parameters

Create a Rician distribution object using the default parameter values.

```
pd = makedist('Rician')
```

```
pd =
```

```
RicianDistribution
```

```
Rician distribution
```

```
    s = 1
```

```
    sigma = 1
```

### Create a Rician Distribution Object Using Specified Parameters

Create a Rician distribution object by specifying the parameter values.

```
pd = makedist('Rician','s',0,'sigma',2)
```

```
pd =
```

```
RicianDistribution
```

```
Rician distribution
```

```
    s = 0
```

```
    sigma = 2
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

2.5066

## See Also

`makedist` | `fitdist` | `dffitool`

## Concepts

- “Rician Distribution” on page B-123
- Class Attributes
- Property Attributes

# ridge

---

**Purpose** Ridge regression

**Syntax**  
`b = ridge(y,X,k)`  
`b = ridge(y,X,k,scaled)`

**Description** `b = ridge(y,X,k)` returns a vector `b` of coefficient estimates for a multilinear ridge regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses. `k` is a vector of ridge parameters. If `k` has  $m$  elements, `b` is  $p$ -by- $m$ . By default, `b` is computed after centering and scaling the predictors to have mean 0 and standard deviation 1. The model does not include a constant term, and `X` should not contain a column of 1s.

`b = ridge(y,X,k,scaled)` uses the {0,1}-valued flag `scaled` to determine if the coefficient estimates in `b` are restored to the scale of the original data. `ridge(y,X,k,0)` performs this additional transformation. In this case, `b` contains  $p+1$  coefficients for each value of `k`, with the first row corresponding to a constant term in the model. `ridge(y,X,k,1)` is the same as `ridge(y,X,k)`. In this case, `b` contains  $p$  coefficients, without a coefficient for a constant term.

The relationship between `b0 = ridge(y,X,k,0)` and `b1 = ridge(y,X,k,1)` is given by

```
m = mean(X);  
s = std(X,0,1)';  
b1_scaled = b1./s;  
b0 = [mean(y)-m*b1_scaled; b1_scaled]
```

This can be seen by replacing the  $x_i$  ( $i = 1, \dots, n$ ) in the multilinear model  $y = b_0^0 + b_1^0 x_1 + \dots + b_n^0 x_n$  with the  $z$ -scores  $z_i = (x_i - \mu_i)/\sigma_i$ , and replacing  $y$  with  $y - \mu_y$ .

In general, `b1` is more useful for producing plots in which the coefficients are to be displayed on the same scale, such as a *ridge trace* (a plot of the regression coefficients as a function of the ridge parameter). `b0` is more useful for making predictions.



Coefficient estimates for multiple linear regression models rely on the independence of the model terms. When terms are correlated and the columns of the design matrix  $X$  have an approximate linear dependence, the matrix  $(X^T X)^{-1}$  becomes close to singular. As a result, the least-squares estimate

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

becomes highly sensitive to random errors in the observed response  $y$ , producing a large variance. This situation of *multicollinearity* can arise, for example, when data are collected without an experimental design.

*Ridge regression* addresses the problem by estimating regression coefficients using

$$\hat{\beta} = (X^T X + kI)^{-1} X^T y$$

where  $k$  is the *ridge parameter* and  $I$  is the identity matrix. Small positive values of  $k$  improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of ridge estimates often result in a smaller mean square error when compared to least-squares estimates.

## Examples

### Ridge Regression

Load the sample data.

```
load acetylene
```

acetylene has observations for the predictor variables  $x_1$ ,  $x_2$ ,  $x_3$ , and the response variable  $y$ .

Plot the predictor variables against each other.

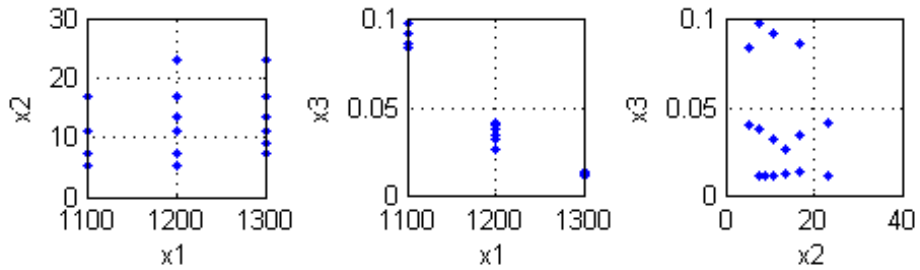
```
subplot(1,3,1)
plot(x1,x2, '.' )
xlabel('x1'); ylabel('x2'); grid on; axis square
```

# ridge

---

```
subplot(1,3,2)
plot(x1,x3,'.')
xlabel('x1'); ylabel('x3'); grid on; axis square

subplot(1,3,3)
plot(x2,x3,'.')
xlabel('x2'); ylabel('x3'); grid on; axis square
```



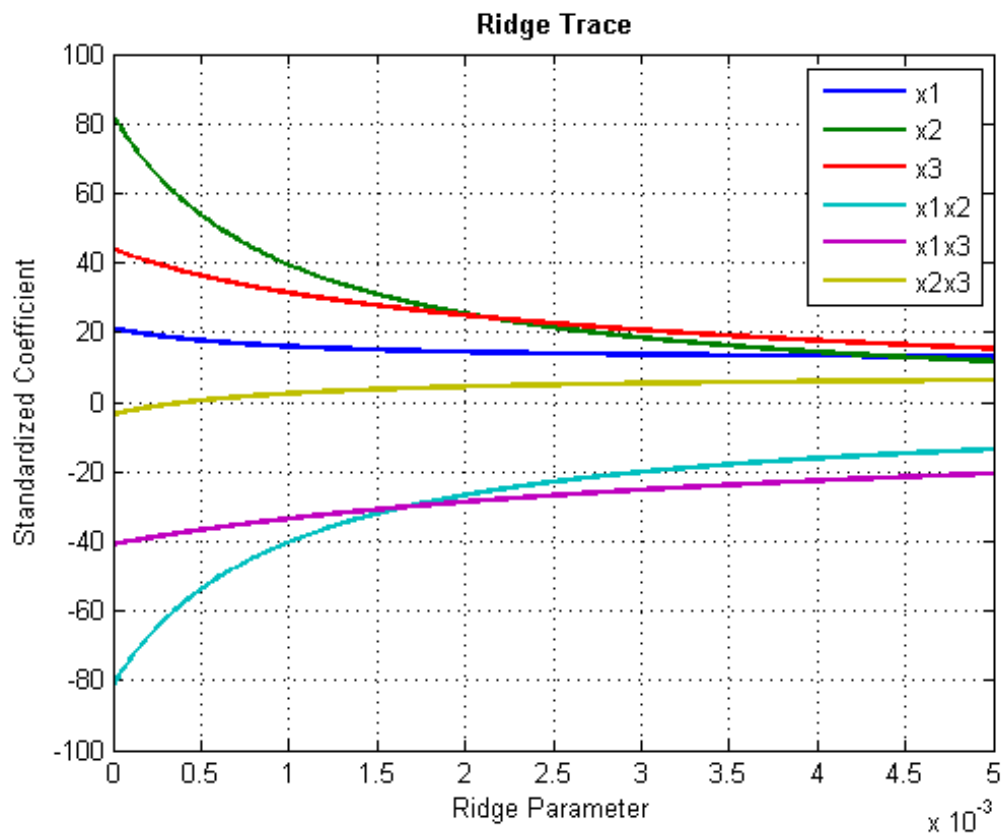
Note the correlation between  $x_1$  and the other two predictor variables.

Compute coefficient estimates for a multilinear model with interaction terms, for a range of ridge parameters using `ridge` and `x2fx`.

```
X = [x1 x2 x3];  
D = x2fx(X,'interaction');  
D(:,1) = []; % No constant term  
k = 0:1e-5:5e-3;  
b = ridge(y,D,k);
```

Plot the ridge trace.

```
figure  
plot(k,b,'LineWidth',2)  
ylim([-100 100])  
grid on  
xlabel('Ridge Parameter')  
ylabel('Standardized Coefficient')  
title('{\bf Ridge Trace}')  
legend('x1','x2','x3','x1x2','x1x3','x2x3')
```



The estimates stabilize to the right of the plot. Note that the coefficient of the x2x3 interaction term changes sign at a value of the ridge parameter  $\approx 5 \times 10^{-4}$ .

## References

- [1] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 55–67.

[2] Hoerl, A. E., and R. W. Kennard. "Ridge Regression: Applications to Nonorthogonal Problems." *Technometrics*. Vol. 12, No. 1, 1970, pp. 69–82.

[3] Marquardt, D.W. "Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation." *Technometrics*. Vol. 12, No. 3, 1970, pp. 591–612.

[4] Marquardt, D. W., and R.D. Snee. "Ridge Regression in Practice." *The American Statistician*. Vol. 29, No. 1, 1975, pp. 3–20.

**See Also**

regress | stepwise

# classregtree.risk

---

**Purpose** Node risks

**Syntax** `r = risk(t)`  
`r = risk(t,nodes)`

**Description** `r = risk(t)` returns an  $n$ -element vector  $r$  of the risk of the nodes in the tree  $t$ , where  $n$  is the number of nodes. The risk  $r(i)$  for node  $i$  is the node error  $e(i)$  (computed by `nodeerr`) weighted by the node probability  $p(i)$  (computed by `nodeprob`).

`r = risk(t,nodes)` takes a vector `nodes` of node numbers and returns the risk values for the specified nodes.

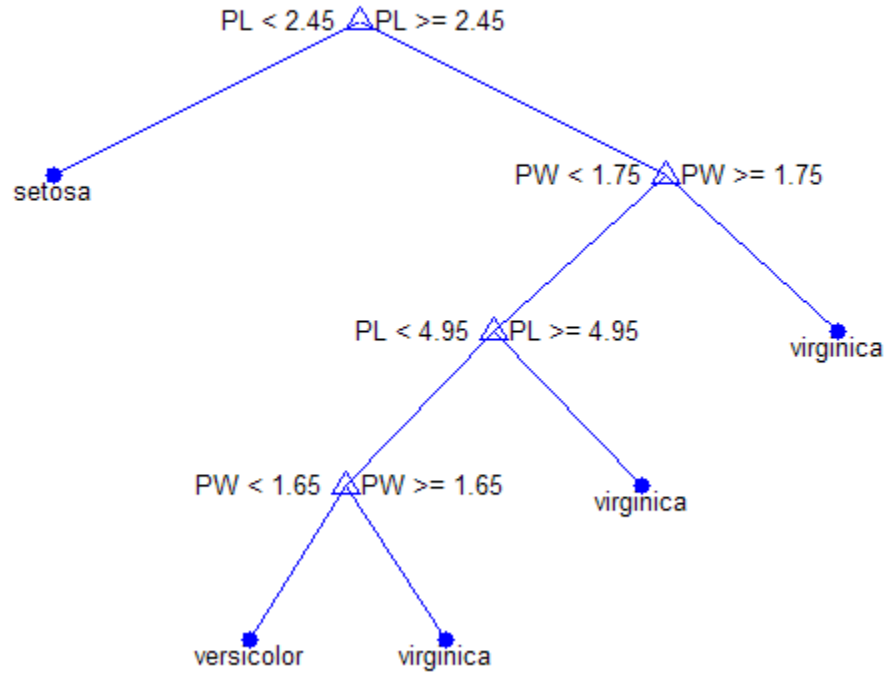
**Examples** Create a classification tree for Fisher's iris data:

```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

Click to display:  Magnification:  Pruning level:



```

e = nodeerr(t);
p = nodeprob(t);
r = risk(t);

```

```

r
r =
    0.6667
         0
    0.3333
    0.0333

```

# classregtree.risk

---

```
0.0067
0.0067
0.0133
0
0

e.*p
ans =
0.6667
0
0.3333
0.0333
0.0067
0.0067
0.0133
0
0
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree | nodeerr | nodeprob



**Purpose** Interactive robust regression

**Syntax** `robustdemo`  
`robustdemo(x,y)`

**Description** `robustdemo` shows the difference between ordinary least squares and robust regression for data with a single predictor. With no input arguments, `robustdemo` displays a scatter plot of a sample of roughly linear data with one outlier. The bottom of the figure displays equations of lines fitted to the data using ordinary least squares and robust methods, together with estimates of the root mean squared errors.

Use the right mouse button to click on a point and view its least-squares leverage and robust weight.

Use the left mouse button to click-and-drag a point. The displays will update.

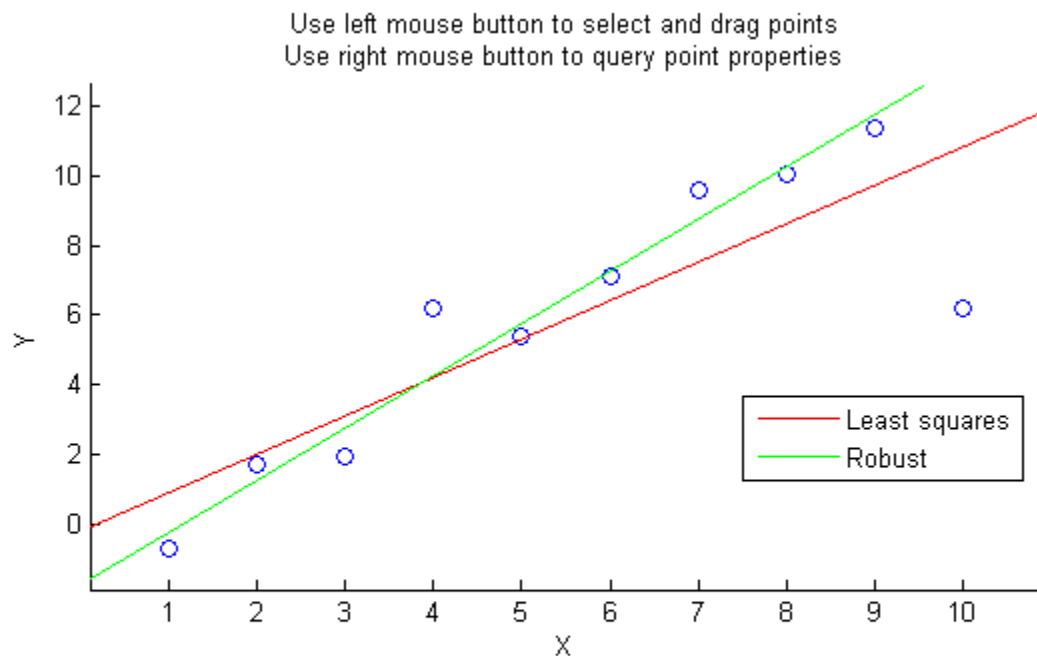
`robustdemo(x,y)` uses `x` and `y` data vectors you supply, in place of the sample data supplied with the function.

**Examples** The following steps show you how to use `robustdemo`.

**1 Start the example.** To begin using `robustdemo` with the built-in data, simply type the function name:

```
robustdemo
```

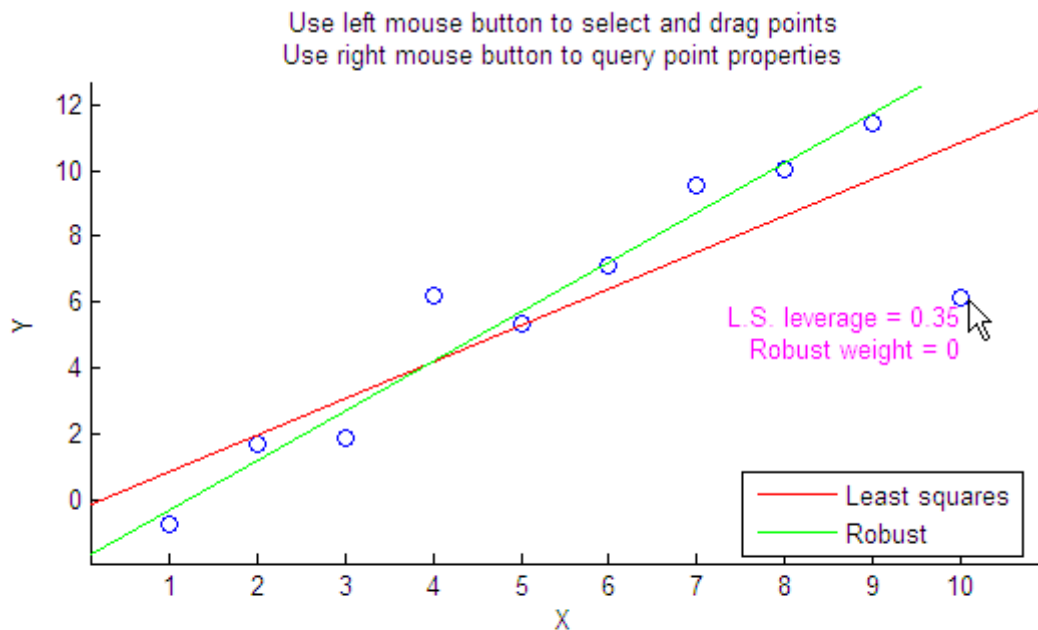
# robustdemo



Least squares:	$Y = -0.188327 + 1.10351 * X$	RMS error = 2.21375
Robust:	$Y = -1.77278 + 1.50415 * X$	RMS error = 1.43663

The resulting figure shows a scatter plot with two fitted lines. The red line is the fit using ordinary least-squares regression. The green line is the fit using robust regression. At the bottom of the figure are the equations for the fitted lines, together with the estimated root mean squared errors for each fit.

**2 View leverages and robust weights.** Right-click on any data point to see its least-squares leverage and robust weight:



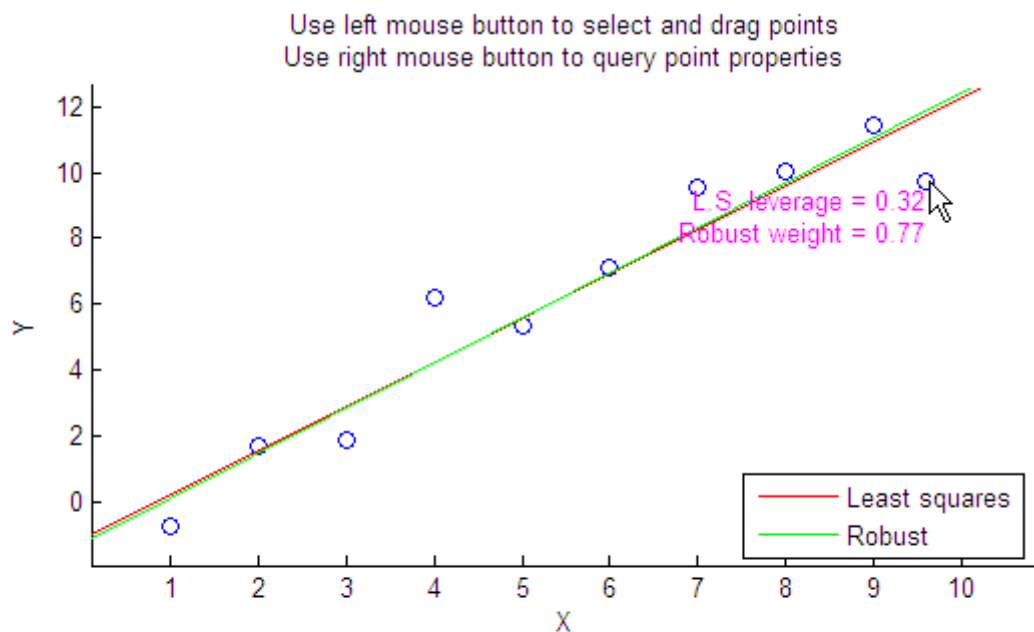
Least squares:  $Y = -0.188327 + 1.10351 \cdot X$       RMS error = 2.21375

Robust:  $Y = -1.77278 + 1.50415 \cdot X$       RMS error = 1.43663

In the built-in data, the right-most point has a relatively high leverage of 0.35. The point exerts a large influence on the least-squares fit, but its small robust weight shows that it is effectively excluded from the robust fit.

- 3 See how changes in the data affect the fits.** With the left mouse button, click and hold on any data point and drag it to a new location. When you release the mouse button, the displays update:

# robustdemo



Least squares:	$Y = -1.0661 + 1.33785 * X$	RMS error = 1.21477
Robust:	$Y = -1.18916 + 1.36459 * X$	RMS error = 1.27697

Bringing the right-most data point closer to the least-squares line makes the two fitted lines nearly identical. The adjusted right-most data point has significant weight in the robust fit.

**See Also** [robustfit](#) | [leverage](#)

**Purpose** Robust regression

**Syntax**

```
b = robustfit(X,y)
b = robustfit(X,y,wfun,tune)
b = robustfit(X,y,wfun,tune,const)
[b,stats] = robustfit(...)
```

**Description** `b = robustfit(X,y)` returns a  $(p + 1)$ -by-1 vector `b` of coefficient estimates for a robust multilinear regression of the responses in `y` on the predictors in `X`. `X` is an  $n$ -by- $p$  matrix of  $p$  predictors at each of  $n$  observations. `y` is an  $n$ -by-1 vector of observed responses. By default, the algorithm uses iteratively reweighted least squares with a bisquare weighting function.

---

**Note** By default, `robustfit` adds a first column of 1s to `X`, corresponding to a constant term in the model. Do not enter a column of 1s directly into `X`. You can change the default behavior of `robustfit` using the input `const`, below.

---

`robustfit` treats NaNs in `X` or `y` as missing values, and removes them.

`b = robustfit(X,y,wfun,tune)` specifies a weighting function `wfun`. `tune` is a tuning constant that is divided into the residual vector before computing weights.

The weighting function `wfun` can be any one of the following strings:

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385

Weight Function	Equation	Default Tuning Constant
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'ols'	Ordinary least squares (no weighting function)	None
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985

If `tune` is unspecified, the default value in the table is used. Default tuning constants give coefficient estimates that are approximately 95% as statistically efficient as the ordinary least-squares estimates, provided the response has a normal distribution with no outliers. Decreasing the tuning constant increases the downweight assigned to large residuals; increasing the tuning constant decreases the downweight assigned to large residuals.

The value  $r$  in the weight functions is

$$r = \text{resid} / (\text{tune} * s * \sqrt{1-h})$$

where `resid` is the vector of residuals from the previous iteration, `h` is the vector of leverage values from a least-squares fit, and `s` is an estimate of the standard deviation of the error term given by

$$s = \text{MAD} / 0.6745$$

Here `MAD` is the median absolute deviation of the residuals from their median. The constant 0.6745 makes the estimate unbiased for the normal distribution. If there are  $p$  columns in  $X$ , the smallest  $p$  absolute deviations are excluded when computing the median.

You can write your own weight function. The function must take a vector of scaled residuals as input and produce a vector of weights as

output. In this case, *wfun* is specified using a function handle @ (as in @myfun), and the input *tune* is required.

`b = robustfit(X,y,wfun,tune,const)` controls whether or not the model will include a constant term. *const* is 'on' to include the constant term (the default), or 'off' to omit it. When *const* is 'on', `robustfit` adds a first column of 1s to *X* and *b* becomes a  $(p + 1)$ -by-1 vector. When *const* is 'off', `robustfit` does not alter *X*, then *b* is a  $p$ -by-1 vector.

`[b,stats] = robustfit(...)` returns the structure *stats*, whose fields contain diagnostic statistics from the regression. The fields of *stats* are:

- `ols_s` — Sigma estimate (RMSE) from ordinary least squares
- `robust_s` — Robust estimate of sigma
- `mad_s` — Estimate of sigma computed using the median absolute deviation of the residuals from their median; used for scaling residuals during iterative fitting
- `s` — Final estimate of sigma, the larger of `robust_s` and a weighted average of `ols_s` and `robust_s`
- `resid` — Residual
- `rstud` — Studentized residual (see `regress` for more information)
- `se` — Standard error of coefficient estimates
- `covb` — Estimated covariance matrix for coefficient estimates
- `coeffcorr` — Estimated correlation of coefficient estimates
- `t` — Ratio of *b* to *se*
- `p` —  $p$ -values for *t*
- `w` — Vector of weights for robust fit
- `R` —  $R$  factor in  $QR$  decomposition of *X*
- `dfe` — Degrees of freedom for error

# robustfit

---

- `h` — Vector of leverage values for least-squares fit

The `robustfit` function estimates the variance-covariance matrix of the coefficient estimates using `inv(X'*X)*stats.s^2`. Standard errors and correlations are derived from this estimate.

## Examples

Generate data with the trend  $y = 10 - 2*x$ , then change one value to simulate an outlier:

```
x = (1:10)';  
y = 10 - 2*x + randn(10,1);  
y(10) = 0;
```

Use both ordinary least squares and robust regression to estimate a straight-line fit:

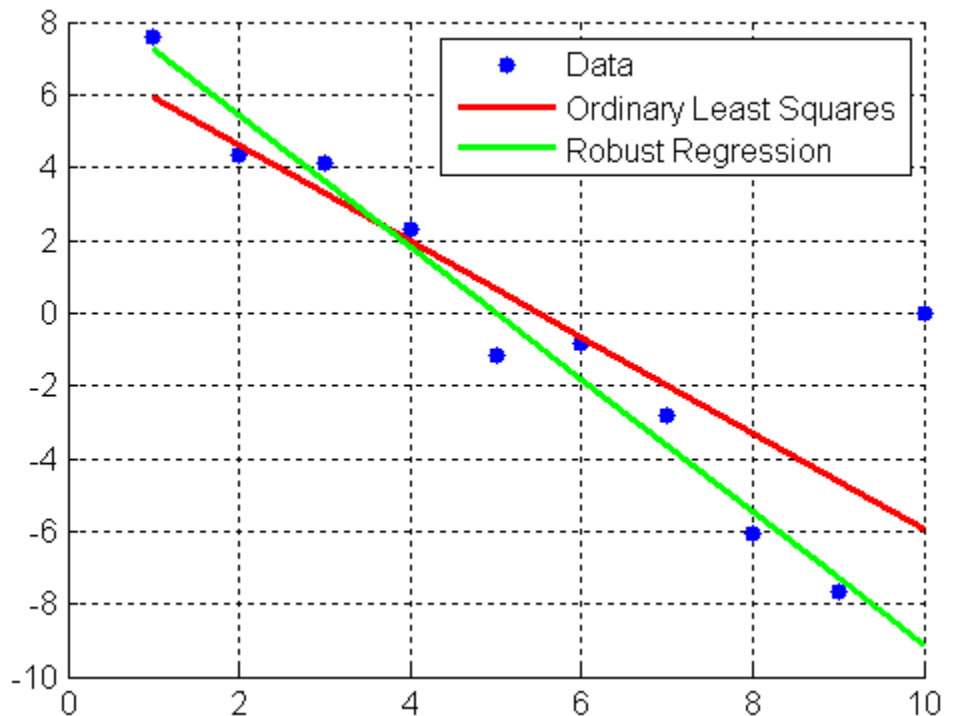
```
bls = regress(y,[ones(10,1) x])  
bls =  
    7.2481  
   -1.3208
```

```
brob = robustfit(x,y)  
brob =  
    9.1063  
   -1.8231
```

A scatter plot of the data together with the fits shows that the robust fit is less influenced by the outlier than the least-squares fit:

```
scatter(x,y,'filled'); grid on; hold on  
plot(x,bls(1)+bls(2)*x,'r','LineWidth',2);  
plot(x,brob(1)+brob(2)*x,'g','LineWidth',2)  
legend('Data','Ordinary Least Squares','Robust Regression')
```





## References

- [1] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.
- [2] Holland, P. W., and R. E. Welsch. "Robust Regression Using Iteratively Reweighted Least-Squares." *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.
- [3] Huber, P. J. *Robust Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1981.

[4] Street, J. O., R. J. Carroll, and D. Ruppert. “A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares.” *The American Statistician*. Vol. 42, 1988, pp. 152–154.

## See Also

`regress` | `robustdemo`

**Purpose** Rotate categorical matrix 90 degrees

**Syntax** B = rot90(A)  
B = rot90(A,k)

**Description** B = rot90(A) returns the 90 degree counterclockwise rotation of the 2-D categorical matrix A.

B = rot90(A,k) returns the  $k \times 90$  degree rotation of A,  $k = +1, +2, \dots$

**See Also** flipdim | flipplr | flipud

# rotatefactors

---

**Purpose** Rotate factor loadings

**Syntax**

```
B = rotatefactors(A)
B = rotatefactors(A, 'Method', 'orthomax', 'Coeff', gamma)
B = rotatefactors(A, 'Method', 'procrustes', 'Target', target)
B = rotatefactors(A, 'Method', 'pattern', 'Target', target)
B = rotatefactors(A, 'Method', 'promax')
[B,T] = rotatefactors(A, ...)
```

**Description** `B = rotatefactors(A)` rotates the  $d$ -by- $m$  loadings matrix  $A$  to maximize the varimax criterion, and returns the result in  $B$ . Rows of  $A$  and  $B$  correspond to variables and columns correspond to factors, for example, the  $(i, j)$ th element of  $A$  is the coefficient for the  $i$ th variable on the  $j$ th factor. The matrix  $A$  usually contains principal component coefficients created with `pca` or `pcacov`, or factor loadings estimated with `factoran`.

`B = rotatefactors(A, 'Method', 'orthomax', 'Coeff', gamma)` rotates  $A$  to maximize the orthomax criterion with the coefficient `gamma`, i.e.,  $B$  is the orthogonal rotation of  $A$  that maximizes

$$\text{sum}(D * \text{sum}(B.^4, 1) - \text{GAMMA} * \text{sum}(B.^2, 1).^2)$$

The default value of 1 for `gamma` corresponds to varimax rotation. Other possibilities include `gamma = 0`, `m/2`, and `d(m - 1)/(d + m - 2)`, corresponding to quartimax, equamax, and parsimax. You can also supply the strings `'varimax'`, `'quartimax'`, `'equamax'`, or `'parsimax'` for the `'method'` parameter and omit the `'Coeff'` parameter.

If `'Method'` is `'orthomax'`, `'varimax'`, `'quartimax'`, `'equamax'`, or `'parsimax'`, then additional parameters are

- `'Normalize'` — Flag indicating whether the loadings matrix should be row-normalized for rotation. If `'on'` (the default), rows of  $A$  are normalized prior to rotation to have unit Euclidean norm, and unnormalized after rotation. If `'off'`, the raw loadings are rotated and returned.

- 'Reltol' — Relative convergence tolerance in the iterative algorithm used to find T. The default is  $\sqrt{\text{eps}}$ .
- 'Maxit' — Iteration limit in the iterative algorithm used to find T. The default is 250.

`B = rotatefactors(A, 'Method', 'procrustes', 'Target', target)` performs an oblique procrustes rotation of A to the  $d$ -by- $m$  target loadings matrix target.

`B = rotatefactors(A, 'Method', 'pattern', 'Target', target)` performs an oblique rotation of the loadings matrix A to the  $d$ -by- $m$  target pattern matrix target, and returns the result in B. target defines the "restricted" elements of B, i.e., elements of B corresponding to zero elements of target are constrained to have small magnitude, while elements of B corresponding to nonzero elements of target are allowed to take on any magnitude.

If 'Method' is 'procrustes' or 'pattern', an additional parameter is 'Type', the type of rotation. If 'Type' is 'orthogonal', the rotation is orthogonal, and the factors remain uncorrelated. If 'Type' is 'oblique' (the default), the rotation is oblique, and the rotated factors might be correlated.

When 'Method' is 'pattern', there are restrictions on target. If A has  $m$  columns, then for orthogonal rotation, the  $j$ th column of target must contain at least  $m - j$  zeros. For oblique rotation, each column of target must contain at least  $m - 1$  zeros.

`B = rotatefactors(A, 'Method', 'promax')` rotates A to maximize the promax criterion, equivalent to an oblique Procrustes rotation with a target created by an orthomax rotation. Use the four orthomax parameters to control the orthomax rotation used internally by promax.

An additional parameter for 'promax' is 'Power', the exponent for creating promax target matrix. 'Power' must be 1 or greater. The default is 4.

`[B,T] = rotatefactors(A,...)` returns the rotation matrix T used to create B, that is,  $B = A * T$ . You can find the correlation matrix of the rotated factors by using  $\text{inv}(T' * T)$ . For orthogonal rotation, this is the

# rotatefactors

---

identity matrix, while for oblique rotation, it has unit diagonal elements but nonzero off-diagonal elements.

## Examples

```
rng('default') % for reproducibility
X = randn(100,10);

% Default (normalized varimax) rotation:
% first three principle components.
LPC = pca(X);
[L1,T] = rotatefactors(LPC(:,1:3));

% Equamax rotation:
% first three principle components.
[L2,T] = rotatefactors(LPC(:,1:3),...
                      'method','equamax');

% Promax rotation:
% first three factors.
LFA = factoran(X,3,'Rotate','none');
[L3,T] = rotatefactors(LFA(:,1:3),...
                      'method','promax',...
                      'power',2);

% Pattern rotation:
% first three factors.
Tgt = [1 1 1 1 1 0 1 0 1 1; ...
       0 0 0 1 1 1 0 0 0 0; ...
       1 0 0 1 0 1 1 1 1 0]';
[L4,T] = rotatefactors(LFA(:,1:3),...
                      'method','pattern',...
                      'target',Tgt);
inv(T'*T) % Correlation matrix of the rotated factors
ans =

    1.0000   -0.9593   -0.7098
   -0.9593    1.0000    0.5938
   -0.7098    0.5938    1.0000
```

**References**

[1] Harman, H. H. *Modern Factor Analysis*. 3rd ed. Chicago: University of Chicago Press, 1976.

[2] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd ed. New York: American Elsevier Publishing, 1971.

**See Also**

biplot | factoran | pca | pcacov | procrustes

**Purpose** Row exchange

**Syntax**

```
dRE = rowexch(nfactors,nruns)
[dRE,X] = rowexch(nfactors,nruns)
[dRE,X] = rowexch(nfactors,nruns,model)
[dRE,X] = rowexch(...,param1,val1,param2,val2,...)
```

**Description** `dRE = rowexch(nfactors,nruns)` uses a row-exchange algorithm to generate a  $D$ -optimal design `dRE` with `nruns` runs (the rows of `dRE`) for a linear additive model with `nfactors` factors (the columns of `dRE`). The model includes a constant term.

`[dRE,X] = rowexch(nfactors,nruns)` also returns the associated design matrix  $X$ , whose columns are the model terms evaluated at each treatment (row) of `dRE`.

`[dRE,X] = rowexch(nfactors,nruns,model)` uses the linear regression model specified in `model`. `model` is one of the following strings:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

The order of the columns of  $X$  for a full quadratic model with  $n$  terms is:

- 1** The constant term
- 2** The linear terms in order 1, 2, ...,  $n$
- 3** The interaction terms in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ )
- 4** The squared terms in order 1, 2, ...,  $n$

Other models use a subset of these terms, in the same order.



Alternatively, *model* can be a matrix specifying polynomial terms of arbitrary order. In this case, *model* should have one column for each factor and one row for each term in the model. The entries in any row of *model* are powers for the factors in the columns. For example, if a model has factors  $X_1$ ,  $X_2$ , and  $X_3$ , then a row  $[0 \ 1 \ 2]$  in *model* specifies the term  $(X_1.^0) \cdot (X_2.^1) \cdot (X_3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

`[dRE,X] = rowexch(...,param1,val1,param2,val2,...)` specifies additional parameter/value pairs for the design. Valid parameters and their values are listed in the following table.

Parameter	Value
'bounds'	Lower and upper bounds for each factor, specified as a 2-by-nfactors matrix. Alternatively, this value can be a cell array containing nfactors elements, each element specifying the vector of allowable values for the corresponding factor.
'categorical'	Indices of categorical predictors.
'display'	Either 'on' or 'off' to control display of the iteration counter. The default is 'on'.
'excludefun'	Handle to a function that excludes undesirable runs. If the function is $f$ , it must support the syntax $b = f(S)$ , where $S$ is a matrix of treatments with nfactors columns and $b$ is a vector of Boolean values with the same number of rows as $S$ . $b(i)$ is true if the $i$ th row $S$ should be excluded.
'init'	Initial design as an nruns-by-nfactors matrix. The default is a randomly selected set of points.
'levels'	Vector of number of levels for each factor.
'maxiter'	Maximum number of iterations. The default is 10.

Parameter	Value
options	<p>A structure that specifies whether to run in parallel, and specifies the random stream or streams. Create the options structure with <code>statset</code>. Option fields:</p> <ul style="list-style-type: none"> <li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li> <li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: <code>'mlfg6331_64'</code> or <code>'mrg32k3a'</code>.</li> <li>• <code>Streams</code> — A <code>RandStream</code> object or cell array of such objects. If you do not specify <code>Streams</code>, <code>rowexch</code> uses the default stream or streams. If you choose to specify <code>Streams</code>, use a single object except in the case <ul style="list-style-type: none"> <li>▪ You have an open Parallel pool</li> <li>▪ <code>UseParallel</code> is <code>true</code></li> <li>▪ <code>UseSubstreams</code> is <code>false</code></li> </ul> <p>In that case, use a cell array the same size as the Parallel pool.</p> </li> </ul>
'tries'	<p>Number of times to try to generate a design from a new starting point. The algorithm uses random points for each try, except possibly the first. The default is 1.</p>

## Algorithms

Both `cordexch` and `rowexch` use iterative search algorithms. They operate by incrementally changing an initial design matrix  $X$  to increase  $D = |X^T X|$  at each step. In both algorithms, there is randomness built into the selection of the initial design and into the choice of the incremental changes. As a result, both algorithms may return locally, but not globally,  $D$ -optimal designs. Run each algorithm multiple times

and select the best result for your final design. Both functions have a 'tries' parameter that automates this repetition and comparison.

At each step, the row-exchange algorithm exchanges an entire row of  $X$  with a row from a design matrix  $C$  evaluated at a candidate set of feasible treatments. The rowexch function automatically generates a  $C$  appropriate for a specified model, operating in two steps by calling the candgen and candexch functions in sequence. Provide your own  $C$  by calling candexch directly. In either case, if  $C$  is large, its static presence in memory can affect computation.

## Examples

Suppose you want a design to estimate the parameters in the following three-factor, seven-term interaction model:

$$y = \beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_3 + \beta_{12}x_1x_2 + \beta_{13}x_1x_3 + \beta_{23}x_2x_3 + \varepsilon$$

Use rowexch to generate a  $D$ -optimal design with seven runs:

```
nfactors = 3;
nruns = 7;
[dRE,X] = rowexch(nfactors,nruns,'interaction','tries',10)
dRE =
    -1    -1     1
     1    -1     1
     1    -1    -1
     1     1     1
    -1    -1    -1
    -1     1    -1
    -1     1     1
X =
     1    -1    -1     1     1    -1    -1
     1     1    -1     1    -1     1    -1
     1     1    -1    -1    -1    -1     1
     1     1     1     1     1     1     1
     1    -1    -1    -1     1     1     1
     1    -1     1    -1    -1     1    -1
     1    -1     1     1    -1    -1     1
```

## rowexch

---

Columns of the design matrix  $X$  are the model terms evaluated at each row of the design  $dRE$ . The terms appear in order from left to right: constant term, linear terms (1, 2, 3), interaction terms (12, 13, 23). Use  $X$  to fit the model, as described in “Linear Regression” on page 9-12, to response data measured at the design points in  $dRE$ .

### See Also

candgen | candexch | cordexch

**Purpose** Interactive response surface demonstration

**Syntax** rsmdemo

**Description** rsmdemo opens a group of three graphical user interfaces for interactively investigating response surface methodology (RSM), nonlinear fitting, and the design of experiments.

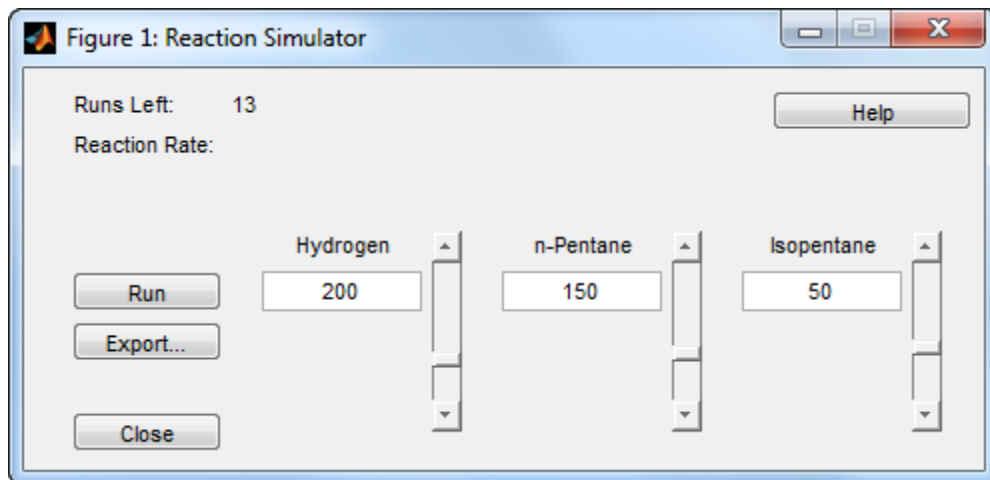
The interfaces allow you to collect and model data from a simulated chemical reaction. Experimental predictors are concentrations of three reactants (hydrogen, *n*-Pentane, and isopentane) and the response is the reaction rate. The reaction rate is simulated by a Hougen-Watson model (Bates and Watts, [2], pp. 271–272):

$$rate = \frac{\beta_1 x_2 - x_3 / \beta_5}{1 + \beta_2 x_1 + \beta_3 x_2 + \beta_4 x_3}$$

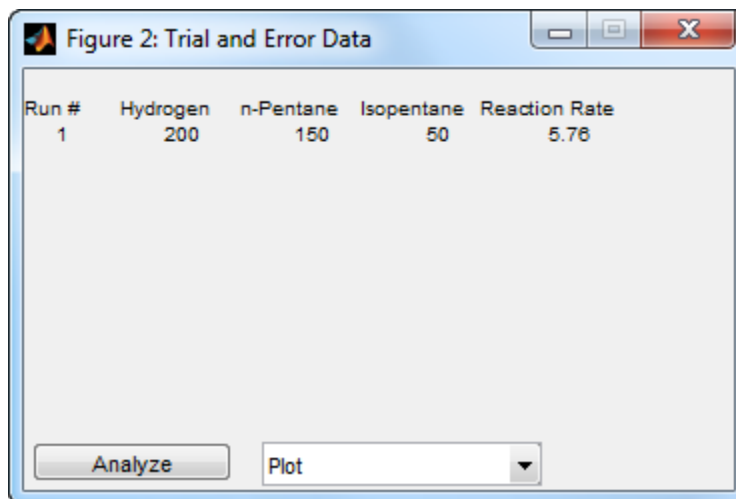
where *rate* is the reaction rate,  $x_1$ ,  $x_2$ , and  $x_3$  are the concentrations of hydrogen, *n*-pentane, and isopentane, respectively, and  $\beta_1, \beta_2, \dots, \beta_5$  are fixed parameters. Random errors are used to perturb the reaction rate for each combination of reactants.

Collect data using one of two methods:

- 1 Manually set reactant concentrations in the **Reaction Simulator** interface by editing the text boxes or by adjusting the associated sliders.

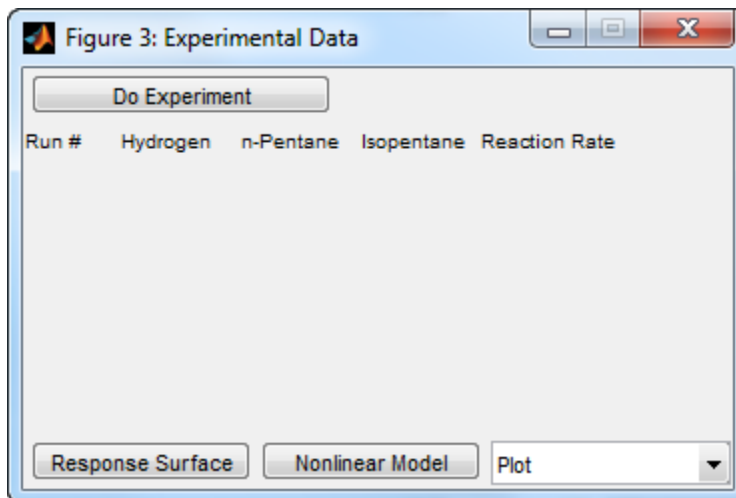


When you click **Run**, the concentrations and simulated reaction rate are recorded on the **Trial and Error Data** interface.

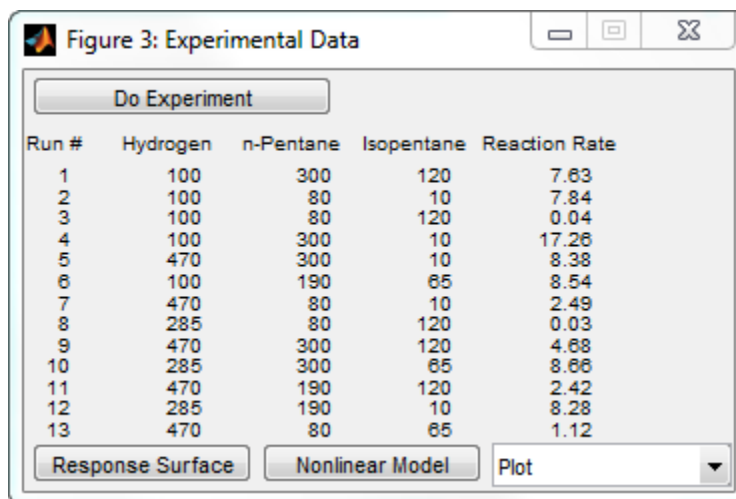


You are allowed up to 13 independent experimental runs for data collection.

- 2 Use a designed experiment to set reactant concentrations in the **Experimental Data** interface by clicking the **Do Experiment** button.



A 13-run  $D$ -optimal design for a full quadratic model is generated by the `cordexch` function, and the concentrations and simulated reaction rates are recorded on the same interface.



Once data is collected, scatter plots of reaction rates vs. individual predictors are generated by selecting one of the following from the **Plot** pop-up menu below the recorded data:

- **Hydrogen vs. Rate**
- **n-Pentane vs. Rate**
- **Isopentane vs. Rate**

Fit a response surface model to the data by clicking the **Analyze** button below the trial-and-error data or the **Response Surface** button below the experimental data. Both buttons load the data into the Response Surface Tool `rstool`. By default, trial-and-error data is fit with a linear additive model and experimental data is fit with a full quadratic model, but the models can be adjusted in the Response Surface Tool.

For experimental data, you have the additional option of fitting a Hougen-Watson model. Click the **Nonlinear Model** button to load the data and the model in `hougen` into the Nonlinear Fitting Tool `nlintool`.

## See Also

`hougen` | `cordexch` | `rstool` | `nlintool`

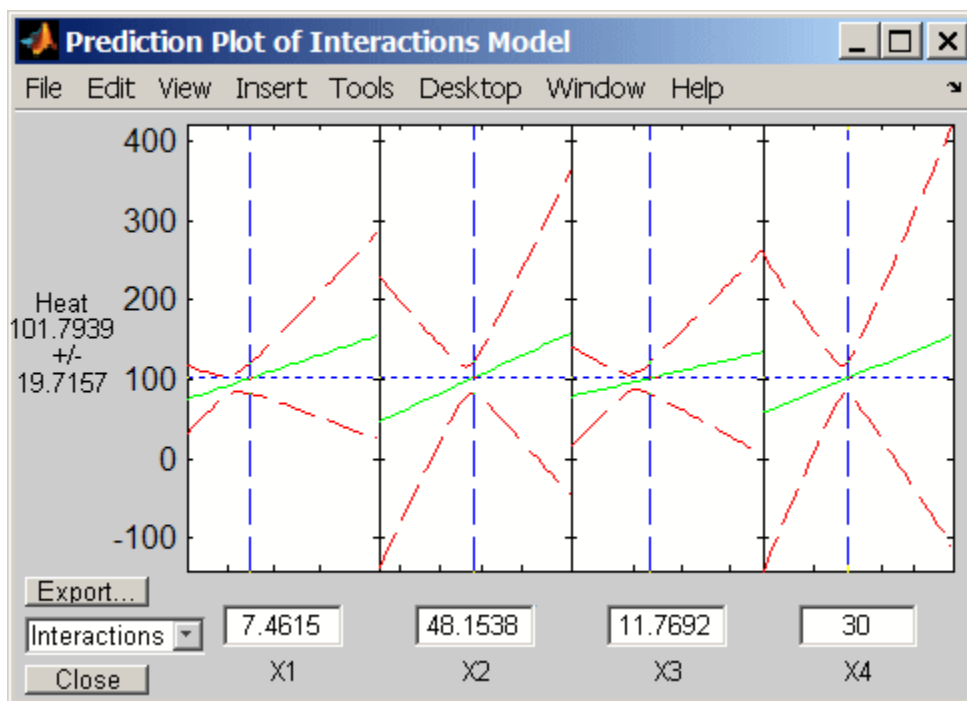


**Purpose** Interactive response surface modeling

**Syntax**

```
rstool
rstool(X,Y,model)
rstool(x,y,model,alpha)
rstool(x,y,model,alpha,xname,yname)
```

**Description** rstool opens a graphical user interface for interactively investigating one-dimensional contours of multidimensional response surface models.



By default, the interface opens with the data from `hald.mat` and a fitted response surface with constant, linear, and interaction terms.

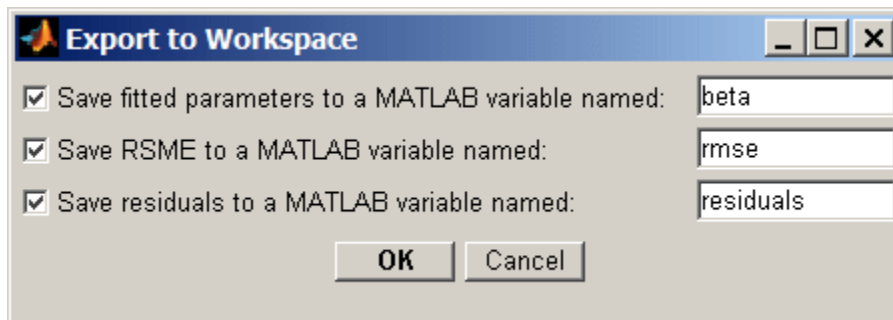
A sequence of plots is displayed, each showing a contour of the response surface against a single predictor, with all other predictors held fixed.

rstool plots a 95% simultaneous confidence band for the fitted response surface as two red curves. Predictor values are displayed in the text boxes on the horizontal axis and are marked by vertical dashed blue lines in the plots. Predictor values are changed by editing the text boxes or by dragging the dashed blue lines. When you change the value of a predictor, all plots update to show the new point in predictor space.

The pop-up menu at the lower left of the interface allows you to choose among the following models:

- Linear — Constant and linear terms (the default)
- Pure Quadratic — Constant, linear, and squared terms
- Interactions — Constant, linear, and interaction terms
- Full Quadratic — Constant, linear, interaction, and squared terms

Click **Export** to open the following dialog box:



The dialog allows you to save information about the fit to MATLAB workspace variables with valid names.

`rstool(X,Y,model)` opens the interface with the predictor data in *X*, the response data in *Y*, and the fitted model *model*. Distinct predictor variables should appear in different columns of *X*. *Y* can be a vector, corresponding to a single response, or a matrix, with columns corresponding to multiple responses. *Y* must have as many elements (or rows, if it is a matrix) as *X* has rows.

The optional input *model* can be any one of the following strings:

- 'linear' — Constant and linear terms (the default)
- 'purequadratic' — Constant, linear, and squared terms
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms

To specify a polynomial model of arbitrary order, or a model without a constant term, use a matrix for *model* as described in `x2fx`.

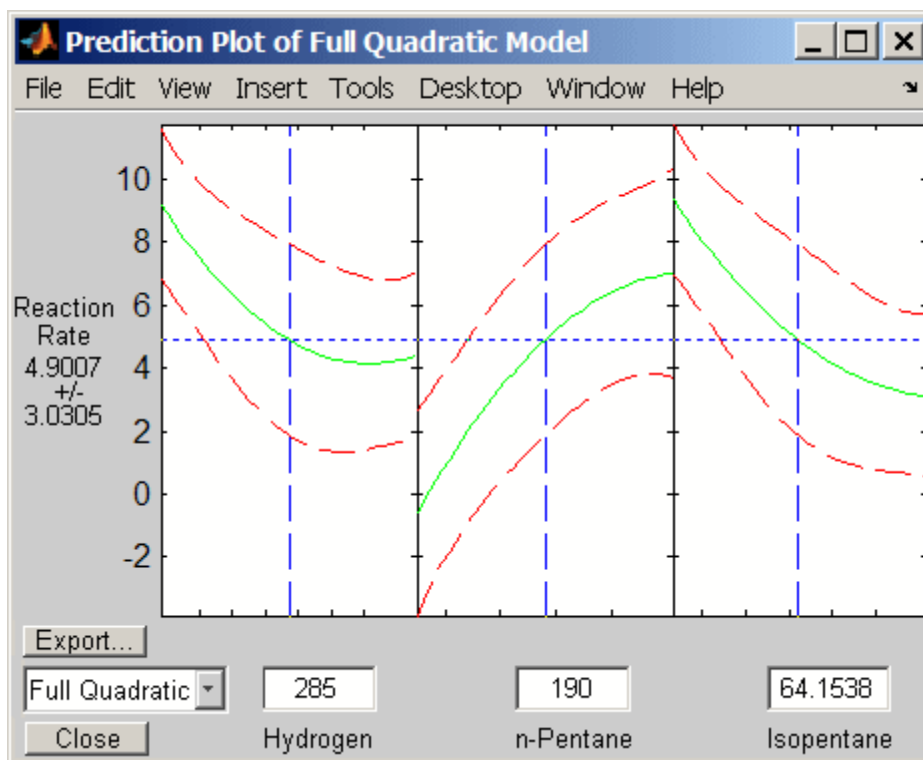
`rstool(x,y,model,alpha)` uses `100(1-alpha)%` global confidence intervals for new observations in the plots.

`rstool(x,y,model,alpha,xname,yname)` labels the axes using the strings in *xname* and *yname*. To label each subplot differently, *xname* and *yname* can be cell arrays of strings.

## Examples

The following uses `rstool` to visualize a quadratic response surface model of the 3-D chemical reaction data in `reaction.mat`:

```
load reaction
alpha = 0.01; % Significance level
rstool(reactants,rate,'quadratic',alpha,xn,yn)
```



The rstool interface is used by rsmdemo to visualize the results of simulated experiments with data like that in reaction.mat. As described in “Response Surface Designs” on page 18-9, rsmdemo uses a response surface model to generate simulated data at combinations of predictors specified by either the user or by a designed experiment.

## See Also

x2fx | rsmdemo | nlintool

**Purpose**

Run test for randomness

**Syntax**

```
h = runstest(x)
h = runstest(x,v)
h = runstest(x,'ud')
h = runstest(...,param1,val1,param2,val2,...)
[h,p] = runstest(...)
[h,p,stats] = runstest(...)
```

**Description**

`h = runstest(x)` performs a runs test on the sequence of observations in the vector `x`. This is a test of the null hypothesis that the values in `x` come in random order, against the alternative that they do not. The test is based on the number of runs of consecutive values above or below the mean of `x`. Too few runs indicate a tendency for high and low values to cluster. Too many runs indicate a tendency for high and low values to alternate. The test returns the logical value `h = 1` if it rejects the null hypothesis at the 5% significance level, and `h = 0` if it cannot. The test treats NaN values in `x` as missing values, and ignores them.

`runstest` uses a test statistic which is approximately normally distributed when the null hypothesis is true. It is the difference between the number of runs and its mean, divided by its standard deviation.

`h = runstest(x,v)` performs the test using runs above or below the value `v`. Values exactly equal to `v` are discarded. The default value of `v` is the mean of `x`.

`h = runstest(x,'ud')` performs a test for the number of runs up or down. This also tests the hypothesis that the values in `x` come in random order. Too few runs indicate a trend. Too many runs indicate an oscillation. Values exactly equal to the preceding value are discarded.

`h = runstest(...,param1,val1,param2,val2,...)` specifies additional parameters and their values. Valid parameter/value pairs are the following:

- `'alpha'` — A scalar giving the significance level of the test
- `'method'` — Either `'exact'` to compute the  $p$  value using an exact algorithm, or `'approximate'` to use a normal approximation. The

# runstest

---

default is 'exact', except for runs up/down when the length of is 51 or more. The 'exact' method is not available for runs up/down when the length of  $x$  is over 50.

- 'tail' — Performs the test against one of the following alternative hypotheses:
  - 'both' — two-tailed test (sequence is not random)
  - 'right' — right-tailed test (like values separate for runs above/below, direction alternates for runs up/down)
  - 'left' — left-tailed test (like values cluster for runs above/below, values trend for runs up/down)

`[h,p] = runstest(...)` returns the  $p$  value of the test. The output  $p$  is computed from either the test statistic or the exact distribution of the number of runs, depending on the value of the 'method' parameter.

`[h,p,stats] = runstest(...)` returns a structure `stats` with the following fields:

- `nruns` — The number of runs
- `n1` — The number of values above  $v$
- `n0` — The number of values below  $v$
- `z` — The test statistic

## Examples

```
x = randn(40,1);  
[h,p] = runstest(x,median(x))  
h =  
    0  
p =  
    0.6286
```

## See Also

`signrank` | `signtest`

# TreeBagger.SampleWithReplacement property

---

**Purpose** Flag to sample with replacement

**Description** The SampleWithReplacement property is a logical flag specifying if data are sampled for each decision tree with replacement. True if TreeBagger samples data with replacement and false otherwise. True by default.

# sampsizepwr

---

**Purpose** Sample size and power of test

**Syntax**

```
n = sampsizepwr(testtype,p0,p1)
n = sampsizepwr(testtype,p0,p1,power)
power = sampsizepwr(testtype,p0,p1,[],n)
p1 = sampsizepwr(testtype,p0,[],power,n)
[...] = sampsizepwr(...,n,param1,va11,param2,va12,...)
```

**Description** `n = sampsizepwr(testtype,p0,p1)` returns the sample size `n` required for a two-sided test of the specified type to have a power (probability of rejecting the null hypothesis when the alternative hypothesis is true) of 0.90 when the significance level (probability of rejecting the null hypothesis when the null hypothesis is true) is 0.05. `p0` specifies parameter values under the null hypothesis. `p1` specifies the single parameter value being tested under the alternative hypothesis.

The following values are available for `testtype`:

- 'z' —  $z$ -test for normally distributed data with known standard deviation. `p0` is a two-element vector [`mu0` `sigma0`] of the mean and standard deviation, respectively, under the null hypothesis. `p1` is the value of the mean under the alternative hypothesis.
- 't' —  $t$ -test for normally distributed data with unknown standard deviation. `p0` is a two-element vector [`mu0` `sigma0`] of the mean and standard deviation, respectively, under the null hypothesis. `p1` is the value of the mean under the alternative hypothesis.
- 'var' — Chi-square test of variance for normally distributed data. `p0` is the variance under the null hypothesis. `p1` is the variance under the alternative hypothesis.
- 'p' — Test of the  $p$  parameter (success probability) for a binomial distribution. `p0` is the value of  $p$  under the null hypothesis. `p1` is the value of  $p$  under the alternative hypothesis.

The 'p' test is a discrete test for which increasing the sample size does not always increase the power. For `n` values larger than 200, there may be values smaller than the returned `n` value that also produce the desired size and power.



`n = sampsizepwr(testtype,p0,p1,power)` returns the sample size `n` such that the power is `power` for the parameter value `p1`.

`power = sampsizepwr(testtype,p0,p1,[],n)` returns the power achieved for a sample size of `n` when the true parameter value is `p1`.

`p1 = sampsizepwr(testtype,p0,[],power,n)` returns the parameter value detectable with the specified sample size `n` and power `power`.

When computing `p1` for the 'p' test, if no alternative can be rejected for a given null hypothesis and significance level, the function displays a warning message and returns NaN.

`[...] = sampsizepwr(...,n,param1,val1,param2,val2,...)` specifies one or more of the following name/value pairs:

- 'alpha' — Significance level of the test (default 0.05)
- 'tail' — The type of test is one of the following:
  - 'both' — Two-sided test for an alternative not equal to `p0`
  - 'right' — One-sided test for an alternative larger than `p0`
  - 'left' — One-sided test for an alternative smaller than `p0`

## Examples

### Mean for Given Sample Size and Power

Compute the mean closest to 100 that can be determined to be significantly different from 100 using a *t*-test with a sample size of 60 and a power of 0.8.

```
mu1 = sampsizepwr('t',[100 10],[],0.8,60)
```

```
mu1 =
```

```
103.6770
```

## Compute Sample Size for a Binomial Test

Compute the sample size  $n$  required to distinguish  $p = 0.26$  from  $p = 0.6$  with a binomial test.

```
napprox = sampsizepwr('p',0.2,0.26,0.6)
```

Warning: Values  $N > 200$  are approximate. Plotting the power as a function of  $N$  may reveal lower  $N$  values that have the required power.

```
napprox =
```

```
244
```

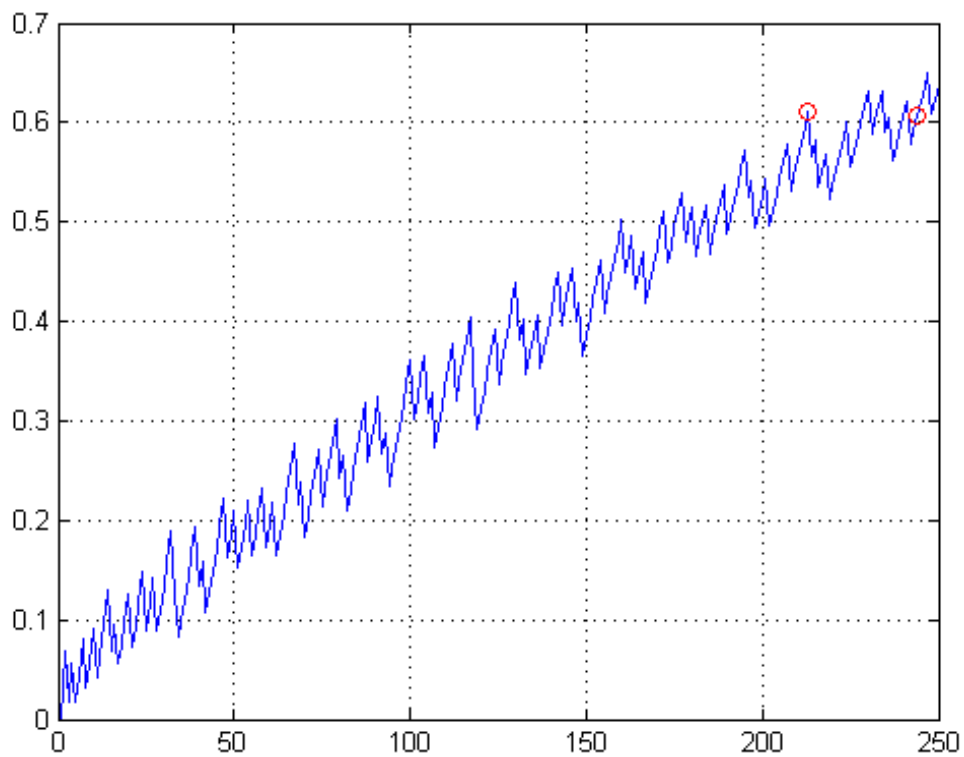
The result is approximate, so make a plot to see if any smaller  $n$  values also have the required power of 0.5.

```
nn = 1:250;  
pwr = sampsizepwr('p',0.2,0.26,[],nn);  
nexact = min(nn(pwr>=0.6))
```

```
nexact =
```

```
213
```

```
plot(nn,pwr,'b-',[napprox nexact],pwr([napprox nexact]),'ro');  
grid on
```



**See Also** `vartest` | `ttest` | `ztest` | `binocdf`

# scatterhist

---

## Purpose

Scatter plot with marginal histograms

## Syntax

```
scatterhist(x,y)
scatterhist(x,y,Name,Value)
```

```
h = scatterhist( ___ )
```

## Description

`scatterhist(x,y)` creates a 2-D scatter plot of the data in the vectors `x` and `y`, and puts a univariate histogram on the horizontal and vertical axes of the plot.

`scatterhist(x,y,Name,Value)` creates the plot using additional options specified by one or more name-value pair arguments. For example, you can specify a grouping variable or change the display options.

`h = scatterhist( ___ )` returns a vector of three axis handles for the scatter plot, the histogram along the horizontal axis, and the histogram along the vertical axis, respectively, using any of the input arguments in the previous syntaxes.

## Input Arguments

### **x** - Sample data

vector

Sample data, specified as a vector. The data vectors `x` and `y` must be the same length. If `x` or `y` contain NaN values, then `scatterhist`:

- Removes rows with NaN values in either `x` or `y` from both data vectors when generating the scatter plot
- Removes rows with NaN values only from the corresponding `x` or `y` data vector when generating the marginal histograms

### Data Types

single | double

### **y** - Sample data

vector

Sample data, specified as a vector. The data vectors `x` and `y` must be the same length.

### Data Types

single | double

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

**Example:** `'Location','SouthEast','Direction','out'` specifies a plot with histograms located below and to the right of the scatter plot, with the bars directed away from the scatter plot.

### 'NBins' - Number of bins for histograms

positive integer value | vector

Number of bins for histograms, specified as the comma-separated pair consisting of `'Nbins'` and a positive integer value greater than or equal to 2, or vector of two such values. If the number of bins is specified as a positive integer value, that value is the number of bins for both the `x` and `y` histograms. If the number of bins is specified by a vector, the first value is the number of bins for the `x` data, and the second value is the number of bins for the `y` data. By default, the number of bins is computed based on the sample standard deviation using Scott's rule.

**Example:** `'NBins',[5,7]`

### Data Types

single | double

### 'Location' - Location of marginal histograms

`'SouthWest'` (default) | `'SouthEast'` | `'NorthEast'` | `'NorthWest'`

Location of the marginal histograms in the figure, specified as the comma-separated pair consisting of 'Location' and one of the following.

'SouthWest' Plot the histograms below and to the left of the scatter plot.

'SouthEast' Plot the histograms below and to the right of the scatter plot.

'NorthEast' Plot the histograms above and to the right of the scatter plot.

'NorthWest' Plot the histograms above and to the left of the scatter plot.

**Example:** 'Location', 'SouthEast'

## 'Direction' - Direction of marginal histograms

'in' (default) | 'out'

Direction of the marginal histograms, specified as the comma-separated pair consisting of 'Direction' and one of the following.

'in' Plot the histograms with the bars directed toward the scatter plot.

'out' Plot the histograms with the bars directed away from the scatter plot.

**Example:** 'Direction', 'out'

## 'Group' - Grouping variable

categorical array | logical or numeric vector | cell array of strings

Grouping variable, specified as the comma-separated pair consisting of 'Group' and a categorical array, logical or numeric vector, or cell array of strings. Each unique value in a grouping variable defines a group.

For example, if Gender is a cell array of strings with values 'Male' and 'Female', you can use Gender as a grouping variable to plot your data by gender.

Multiple grouping variables can be used by specifying a cell array of grouping variable names. Observations are placed in the same group if they have common values of all specified grouping variables.

For example, if `Smoker` is a logical vector with values 0 for nonsmokers and 1 for smokers, then specifying the cell array `{Gender,Smoker}` divides observations into four groups: Male Smoker, Male Nonsmoker, Female Smoker, and Female Nonsmoker.

**Example:** `'Group',{Gender,Smoker}`

### Data Types

single | double | logical | cell | char

### 'Kernel' - Grouped kernel density plot indicator

`'on' | 'off' | 'overlay'`

Grouped kernel density plot indicator, specified as the comma-separated pair consisting of `'Kernel'` and one of the following.

- `'on'` Display kernel density plots for each group. This is the default if a `Group` parameter is specified.
- `'off'` Display the overall marginal distribution as histograms. This is the default if a `Group` parameter is not specified.
- `'overlay'` Display the overall marginal distribution as kernel density plots overlaid onto histograms, similar to `histfit`.

**Example:** `'Kernel','overlay'`

### 'Bandwidth' - Bandwidth of kernel smoothing window

matrix

Bandwidth of kernel smoothing window, specified as the comma-separated pair consisting of `'Bandwidth'` and a matrix of size 2-by- $K$ , where  $K$  is the number of unique groups. The first row of the matrix gives the bandwidth of each group in  $x$ , and the second row gives the bandwidth of each group in  $y$ . By default, `scatterhist` finds the optimal bandwidth for estimating normal densities. Specifying a different bandwidth value changes the smoothing characteristics of the

resulting kernel density plot. The value specified is a scaling factor for the normal distribution used to generate the kernel density plot.

**Example:** 'Bandwidth', [.5, .2, .1; .15, .25, .35]

## **Data Types**

single | double

## **'Legend' - Legend visibility indicator**

'on' | 'off'

Legend visibility indicator, specified as the comma-separated pair consisting of 'Legend' and one of the following.

'on' Set legend visible. This is the default if a **Group** parameter is specified.

'off' Set legend invisible. This is the default if a **Group** parameter is not specified.

**Example:** 'Legend', 'on'

## **'LineStyle' - Style of kernel density plot line**

valid line style string | cell array of strings

Style of kernel density plot line, specified as the comma-separated pair consisting of 'LineStyle' and a valid line style string or a cell array of valid line style strings. See `plot` for valid line style strings. The default is a solid line. Use a cell array to specify different line styles for each group. When the total number of groups exceeds the number of specified values, `scatterhist` cycles through the specified values.

**Example:** 'LineStyle', {'-', ':', '-.'}

## **'LineWidth' - Width of kernel density plot line**

0.5 (default) | nonnegative scalar value | vector

Width of kernel density plot line, specified as the comma-separated pair consisting of 'LineWidth' and a nonnegative scalar value or vector of nonnegative scalar values. The specified value is the size of the kernel



density plot line measured in points. The default size is 0.5 points. Use a vector to specify different line widths for each group. When the total number of groups is greater than the number of specified values, scatterhist cycles through the specified values.

**Example:** 'LineWidth',[0.5,1,2]

#### **Data Types**

single | double

#### **'Color' - Marker color for each scatter plot group**

valid color designation char | string of chars | matrix of RGB values

Marker color for each scatter plot group, specified as the comma-separated pair consisting of 'Color' and a valid color designation character, a string of valid color designation characters, or a 3-column matrix of RGB values in the range [0,1]. See ColorSpec for predefined colors and their RGB equivalents. If colors are specified using a matrix, each row of the matrix represents a group, and the three columns represent the R value, G value, and B value, respectively. When the total number of groups exceeds the number of specified colors, scatterhist cycles through the specified colors.

**Example:** 'Color','kcm'

**Example:** 'Color',[.5,0,1;0,.5,.5]

#### **Data Types**

single | double | char

#### **'Marker' - Marker symbol for each scatterplot group**

'o' (default) | valid marker symbol | string of valid marker symbols

Marker symbol for each scatter plot group, specified as the comma-separated pair consisting of 'Marker' and a valid marker symbol or string of valid marker symbols. See plot for valid symbols. The default is 'o', a circle. When the total number of groups exceeds the number of specified symbols, scatterhist cycles through the specified symbols.

**Example:** 'Marker','+do'

## 'MarkerSize' - Marker size for each scatter plot group

6 (default) | nonnegative scalar value | vector

Marker size for each scatter plot group, specified as the comma-separated pair consisting of 'MarkerSize' and a nonnegative scalar value or a vector of nonnegative scalar values, measured in points. When the total number of groups exceeds the number of specified values, scatterhist cycles through the specified values.

**Example:** 'MarkerSize',10

### Data Types

single | double

## Output Arguments

### h - Axes handles

vector

Axes handles for the three plots, returned as a vector. The vector contains the handles for the scatter plot, the histogram along the horizontal axis, and the histogram along the vertical axis, respectively.

## Examples

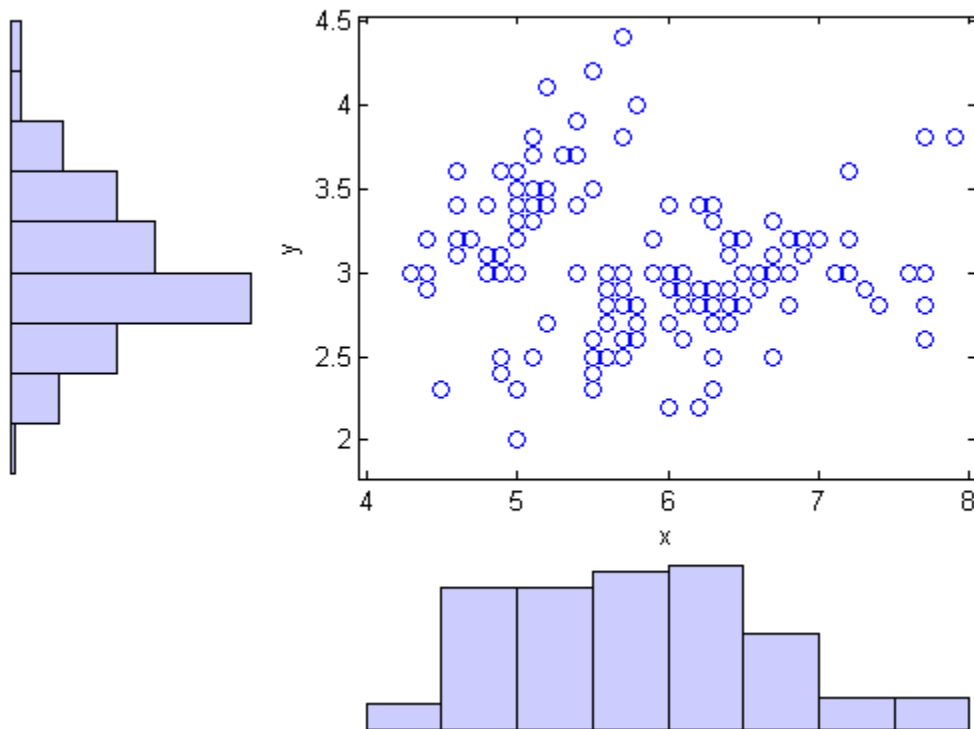
### Create a Scatterhist Plot

Load the sample data. Create data vector  $x$  from the first column of the data matrix, which contains sepal length measurements from iris flowers. Create data vector  $y$  from the second column of the data matrix, which contains sepal width measurements from the same flowers.

```
load fisheriris.mat;  
x = meas(:,1);  
y = meas(:,2);
```

Create a scatter plot and two marginal histograms to visualize the relationship between sepal length and sepal width.

```
scatterhist(x,y)
```



### Plot Grouped Data

Load the sample data. Create data vector  $x$  from the first column of the data matrix, which contains sepal length measurements from three species of iris flowers. Create data vector  $y$  from the second column of the data matrix, which contains sepal width measurements from the same flowers.

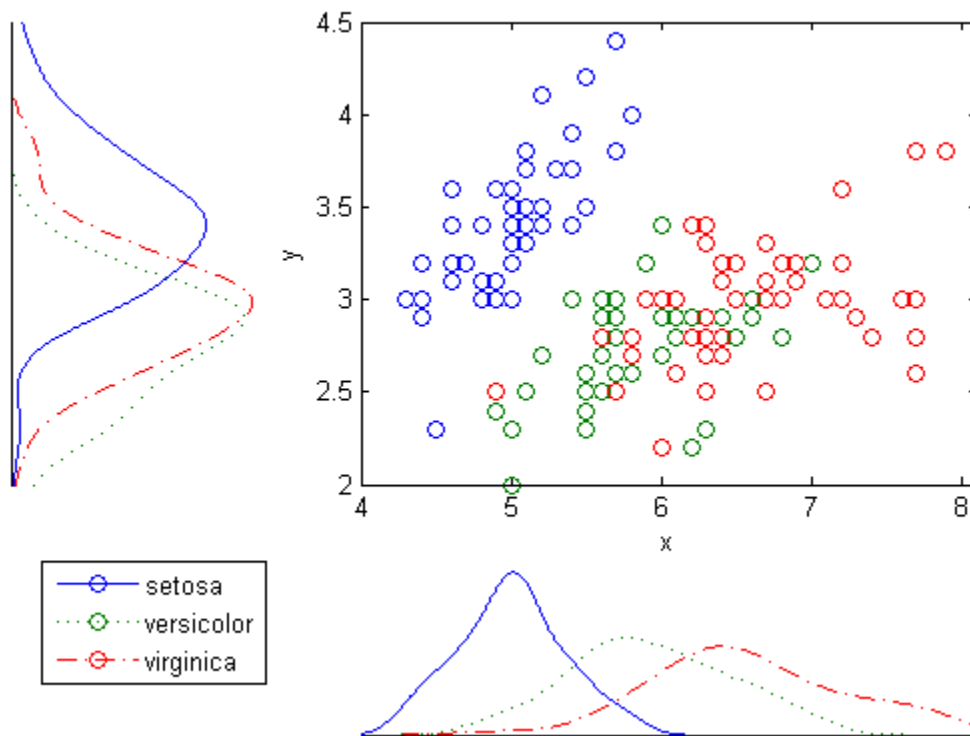
```
load fisheriris.mat;  
x = meas(:,1);
```

# scatterhist

```
y = meas(:,2);
```

Create a scatter plot and six kernel density plots to visualize the relationship between sepal length and sepal width, grouped by species.

```
scatterhist(x,y,'Group',species)
```



The plot shows that the relationship between sepal length and width varies depending on the flower species.

## Customize the Plot Display

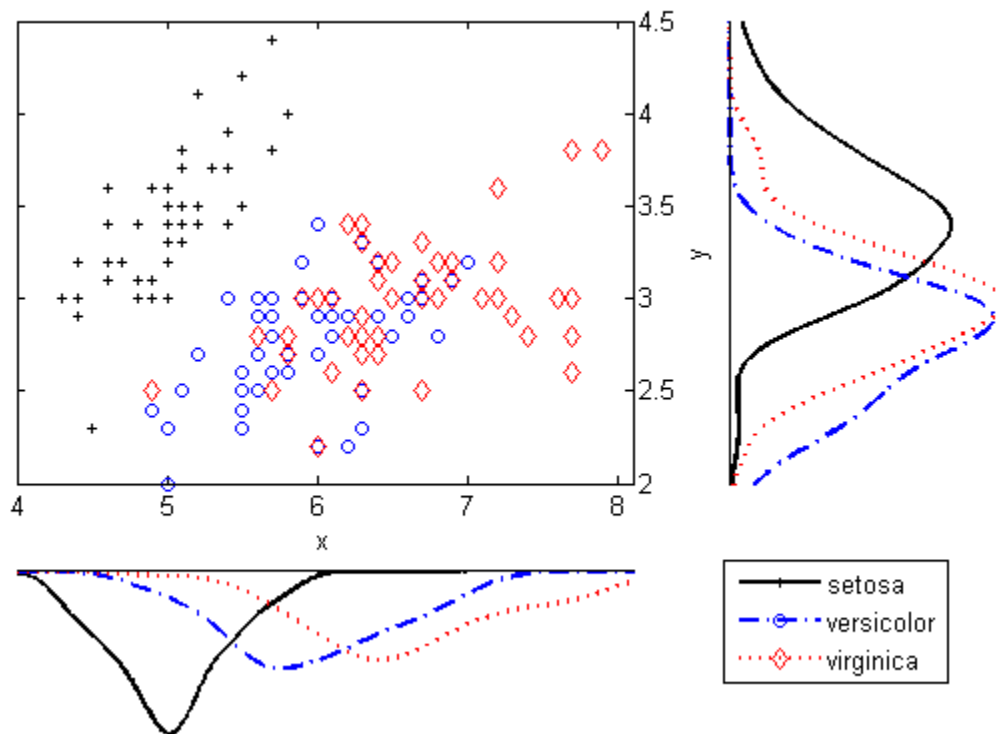
Load the sample data. Create data vector `x` from the first column of the data matrix, which contains sepal length measurements from three different species of iris flowers. Create data vector `y` from the second column of the data matrix, which contains sepal width measurements from the same flowers.

```
load fisheriris.mat;  
x = meas(:,1);  
y = meas(:,2);
```

Create a scatter plot and six kernel density plots to visualize the relationship between sepal length and sepal width as measured on three species of iris flowers, grouped by species. Customize the appearance of the plots.

```
scatterhist(x,y,'Group',species,'Location','SouthEast',...  
            'Direction','out','Color','kbr','LineStyle',{'-','-.'},...  
            'LineWidth',[2,2,2],'Marker','+od','MarkerSize',[4,5,6]);
```

# scatterhist



## Customize Plots Using Axes Handles

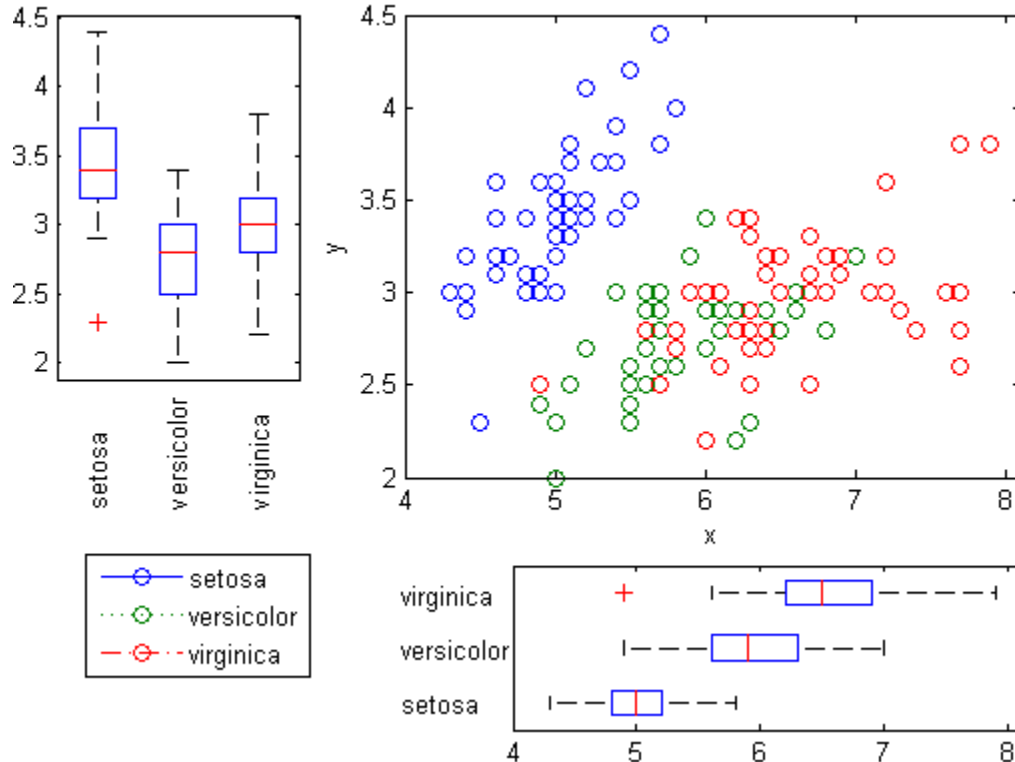
Load the sample data. Create data vector  $x$  from the first column of the data matrix, which contains sepal length measurements from three species of iris flowers. Create data vector  $y$  from the second column of the data matrix, which contains sepal width measurements from the same flowers.

```
load fisheriris.mat;  
x = meas(:,1);
```

```
y = meas(:,2);
```

Use axis handles to replace the marginal histograms with box plots.

```
h = scatterhist(x,y,'Group',species);
hold on;
boxplot(h(2),x,species,'orientation','horizontal');
boxplot(h(3),y,species,'labelorientation','inline');
hold off;
```



# scatterhist

---

## See Also

hist | gscatter

## Concepts

- “Grouping Variables” on page 2-51



**Purpose** Scramble quasi-random point set

**Syntax**

```
ps = scramble(p, type)
ps = scramble(p, 'clear')
ps = scramble(p)
```

**Description** `ps = scramble(p, type)` returns a scrambled copy `ps` of the point set `p` of the `grandset` class, created using the scramble type specified in the string `type`. Point sets from different subclasses of `grandset` support different scramble types, as indicated in the following table.

Subclass	Scramble Types
haltonset	'RR2' — A permutation of the radical inverse coefficients derived by applying a reverse-radix operation to all of the possible coefficient values. The scramble is described in [1].
sobolset	'MatousekAffineOwen' — A random linear scramble combined with a random digital shift. The scramble is described in [2].

`ps = scramble(p, 'clear')` removes all scramble settings from `p` and returns the result in `ps`.

`ps = scramble(p)` removes all scramble settings from `p` and then adds them back in the order they were originally applied. This typically results in a different point set because of the randomness of the scrambling algorithms.

**Examples** Use `haltonset` to generate a 3-D Halton point set, skip the first 1000 values, and then retain every 101st point:

```
p = haltonset(3, 'Skip', 1e3, 'Leap', 1e2)
p =
  Halton point set in 3 dimensions (8.918019e+013 points)
  Properties:
    Skip : 1000
```

# grandset.scramble

---

```
Leap : 100
ScrambleMethod : none
```

Use scramble to apply reverse-radix scrambling:

```
p = scramble(p, 'RR2')
p =
Halton point set in 3 dimensions (8.918019e+013 points)
Properties:
    Skip : 1000
    Leap : 100
    ScrambleMethod : RR2
```

Use net to generate the first four points:

```
X0 = net(p, 4)
X0 =
    0.0928    0.6950    0.0029
    0.6958    0.2958    0.8269
    0.3013    0.6497    0.4141
    0.9087    0.7883    0.2166
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11, :)
X =
    0.0928    0.6950    0.0029
    0.9087    0.7883    0.2166
    0.3843    0.9840    0.9878
    0.6831    0.7357    0.7923
```

## References

[1] Kocis, L., and W. J. Whiten. “Computational Investigations of Low-Discrepancy Sequences.” *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.

[2] Matousek, J. “On the L2-Discrepancy for Anchored Boxes.” *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

**See Also**     [haltonset](#) | [sobolset](#)

# grandset.ScrambleMethod property

---

**Purpose** Settings that control scrambling

**Description** The ScrambleMethod property contains a structure that defines which scrambles to apply to the sequence. The structure consists of two fields:

- **Type:** A string containing the name of the scramble.
- **Options:** A cell array of parameter values for the scramble.

Different point sets support different scramble types as outlined in the help for each point set class. An error occurs if you set an invalid scramble type for a given point set.

The ScrambleMethod property also accepts an empty matrix as a value. This will clear all scrambling and set the property to contain a (0x0) structure.

The `scramble` method provides an alternative, easier way to set scrambles.

**Examples** Apply a random linear scramble combined with a random digital shift to a `sobolset` point set class:

```
P = sobolset(5);  
P = scramble(P, 'MatousekAffineOwen');  
P.ScrambleMethod
```

**See Also** `sobolset` | `scramble`

**Purpose** Segments containing values

**Syntax** `S = segment(obj,X,P)`

**Description** `S = segment(obj,X,P)` returns an array `S` of integers indicating which segment of the piecewise distribution object `obj` contains each value of `X` or, alternatively, `P`. One of `X` and `P` must be empty (`[]`). If `X` is nonempty, `S` is determined by comparing `X` with the quantile boundary values defined for `obj`. If `P` is nonempty, `S` is determined by comparing `P` with the probability boundary values.

**Examples** Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);
obj = paretotails(t,0.1,0.9);

pvals = 0:0.2:1;
s = segment(obj,[],pvals)
s =
     1     2     2     2     2     3
```

**See Also** `paretotails` | `boundary` | `nsegments`

# sequentialfs

---

**Purpose** Sequential feature selection

**Syntax**

```
inmodel = sequentialfs(fun,X,y)
inmodel = sequentialfs(fun,X,Y,Z,...)
[inmodel,history] = sequentialfs(fun,X,...)
[] = sequentialfs(...,param1,val1,param2,val2,...)
```

**Description** `inmodel = sequentialfs(fun,X,y)` selects a subset of features from the data matrix `X` that best predict the data in `y` by sequentially selecting features until there is no improvement in prediction. Rows of `X` correspond to observations; columns correspond to variables or features. `y` is a column vector of response values or class labels for each observation in `X`. `X` and `y` must have the same number of rows. `fun` is a function handle to a function that defines the criterion used to select features and to determine when to stop. The output `inmodel` is a logical vector indicating which features are finally chosen.

Starting from an empty feature set, `sequentialfs` creates candidate feature subsets by sequentially adding each of the features not yet selected. For each candidate feature subset, `sequentialfs` performs 10-fold cross-validation by repeatedly calling `fun` with different training subsets of `X` and `y`, `XTRAIN` and `ytrain`, and test subsets of `X` and `y`, `XTEST` and `ytest`, as follows:

```
criterion = fun(XTRAIN,ytrain,XTEST,ytest)
```

`XTRAIN` and `ytrain` contain the same subset of rows of `X` and `Y`, while `XTEST` and `ytest` contain the complementary subset of rows. `XTRAIN` and `XTEST` contain the data taken from the columns of `X` that correspond to the current candidate feature set.

Each time it is called, `fun` must return a scalar value `criterion`. Typically, `fun` uses `XTRAIN` and `ytrain` to train or fit a model, then predicts values for `XTEST` using that model, and finally returns some measure of distance, or *loss*, of those predicted values from `ytest`. In the cross-validation calculation for a given candidate feature set, `sequentialfs` sums the values returned by `fun` and divides that sum

by the total number of test observations. It then uses that mean value to evaluate each candidate feature subset.

Typical loss measures include sum of squared errors for regression models (`sequentialfs` computes the mean-squared error in this case), and the number of misclassified observations for classification models (`sequentialfs` computes the misclassification rate in this case).

---

**Note** `sequentialfs` divides the sum of the values returned by `fun` across all test sets by the total number of test observations. Accordingly, `fun` should not divide its output value by the number of test observations.

---

After computing the mean criterion values for each candidate feature subset, `sequentialfs` chooses the candidate feature subset that minimizes the mean criterion value. This process continues until adding more features does not decrease the criterion.

`inmodel = sequentialfs(fun,X,Y,Z,...)` allows any number of input variables `X`, `Y`, `Z`, ... . `sequentialfs` chooses features (columns) only from `X`, but otherwise imposes no interpretation on `X`, `Y`, `Z`, ... . All data inputs, whether column vectors or matrices, must have the same number of rows. `sequentialfs` calls `fun` with training and test subsets of `X`, `Y`, `Z`, ... as follows:

```
criterion = fun(XTRAIN,YTRAIN,ZTRAIN,...,  
               XTEST,YTEST,ZTEST,...)
```

`sequentialfs` creates `XTRAIN`, `YTRAIN`, `ZTRAIN`, ... , `XTEST`, `YTEST`, `ZTEST`, ... by selecting subsets of the rows of `X`, `Y`, `Z`, ... . `fun` must return a scalar value `criterion`, but may compute that value in any way. Elements of the logical vector `inmodel` correspond to columns of `X` and indicate which features are finally chosen.

`[inmodel,history] = sequentialfs(fun,X,...)` returns information on which feature is chosen at each step. `history` is a scalar structure with the following fields:

# sequentialfs

---

- **Crit** — A vector containing the criterion values computed at each step.
- **In** — A logical matrix in which row *i* indicates the features selected at step *i*.

[ ] = sequentialfs(...,param1,val1,param2,val2,...) specifies optional parameter name/value pairs from the following table.

Parameter	Value
'cv'	<p>The validation method used to compute the criterion for each candidate feature subset.</p> <ul style="list-style-type: none"><li>• When the value is a positive integer <i>k</i>, sequentialfs uses <i>k</i>-fold cross-validation without stratification.</li><li>• When the value is an object of the cvpartition class, other forms of cross-validation can be specified.</li><li>• When the value is 'resubstitution', the original data are passed to fun as both the training and test data to compute the criterion.</li><li>• When the value is 'none', sequentialfs calls fun as criterion = fun(X,Y,Z,...), without separating test and training sets.</li></ul> <p>The default value is 10, that is, 10-fold cross-validation without stratification.</p> <p>So-called <i>wrapper methods</i> use a function fun that implements a learning algorithm. These methods usually apply cross-validation to select features. So-called <i>filter methods</i> use a function</p>



Parameter	Value
	fun that measures characteristics of the data (such as correlation) to select features.
'mcreps'	A positive integer indicating the number of Monte-Carlo repetitions for cross-validation. The default value is 1. The value must be 1 if the value of 'cv' is 'resubstitution' or 'none'.
'direction'	The direction of the sequential search. The default is 'forward'. A value of 'backward' specifies an initial candidate set including all features and an algorithm that removes features sequentially until the criterion increases.
'keepin'	A logical vector or a vector of column numbers specifying features that must be included. The default is empty.
'keepout'	A logical vector or a vector of column numbers specifying features that must be excluded. The default is empty.
'nfeatures'	The number of features at which <code>sequentialfs</code> should stop. <code>inmodel</code> includes exactly this many features. The default value is empty, indicating that <code>sequentialfs</code> should stop when a local minimum of the criterion is found. A nonempty value overrides values of 'MaxIter' and 'TolFun' in 'options'.
'nullmodel'	A logical value, indicating whether or not the null model (containing no features from X) should be included in feature selection and in the history output. The default is false.

# sequentialfs

Parameter	Value
'options'	<p>Options structure for the iterative sequential search algorithm, as created by <code>statset</code>. <code>sequentialfs</code> uses the following <code>statset</code> parameters:</p> <ul style="list-style-type: none"><li>• <code>Display</code> — Amount of information displayed by the algorithm. The default is 'off'.</li><li>• <code>MaxIter</code> — Maximum number of iterations allowed. The default is <code>Inf</code>.</li><li>• <code>TolFun</code> — Termination tolerance for the objective function value. The default is <code>1e-6</code> if 'direction' is 'forward'; <code>0</code> if 'direction' is 'backward'.</li><li>• <code>TolTypeFun</code> — Use absolute or relative objective function tolerances. The default is 'rel'.</li><li>• <code>UseParallel</code> — Set to <code>true</code> to compute in parallel. Default is <code>false</code>.</li><li>• <code>UseSubstreams</code> — Set to <code>true</code> to compute in parallel in a reproducible fashion. Default is <code>false</code>. To compute reproducibly, set <code>Streams</code> to a type allowing substreams: 'mlfg6331_64' or 'mrg32k3a'.</li><li>• <code>Streams</code> — A <code>RandStream</code> object or cell array consisting of one such object. If you do not specify <code>Streams</code>, <code>sequentialfs</code> uses the default stream.</li></ul>

## Examples

Perform sequential feature selection for classification of noisy features:

```
load fisheriris;  
X = randn(150,10);
```

```
X(:,[1 3 5 7 ])= meas;
y = species;

c = cvpartition(y,'k',10);
opts = statset('display','iter');
fun = @(XT,yT,Xt,yt)...
      (sum(~strcmp(yt,classify(Xt,XT,yT,'quadratic'))));

[fs,history] = sequentialfs(fun,X,y,'cv',c,'options',opts)
```

```
Start forward sequential feature selection:
Initial columns included: none
Columns that can not be included: none
Step 1, added column 7, criterion value 0.04
Step 2, added column 5, criterion value 0.0266667
Final columns included: 5 7
```

```
fs =
    0  0  0  0  1  0  1  0  0  0
history =
    In: [2x10 logical]
    Crit: [0.0400 0.0267]
```

```
history.In
ans =
    0  0  0  0  0  0  1  0  0  0
    0  0  0  0  1  0  1  0  0  0
```

## See Also

[crossval](#) | [cvpartition](#) | [stepwisefit](#) | [statset](#)

## Tutorials

- “Example: Sequential Feature Selection” on page 13-72

## How To

- “Sequential Feature Selection” on page 13-71

# dataset.set

---

## Purpose

Set and display properties

## Syntax

```
set(A)
set(A,PropertyName)
A = set(A,PropertyName,PropertyValue,...)
B = set(A,PropertyName,value)
```

## Description

`set(A)` displays all properties of the dataset array *A* and their possible values.

`set(A,PropertyName)` displays possible values for the property specified by the string *PropertyName*.

`A = set(A,PropertyName,PropertyValue,...)` sets property name/value pairs.

`B = set(A,PropertyName,value)` returns a dataset array *B* that is a copy of *A*, but with the property '*PropertyName*' set to the value *value*.

---

**Note** Using `set(A,'PropertyName',value)` without assigning to a variable does not modify *A*'s properties. Use `A = set(A,'PropertyName',value)` to modify *A*.

---

## Examples

Create a dataset array from Fisher's iris data and add a description:

```
load fisheriris
NumObs = size(meas,1);
NameObs = strcat({'Obs'},num2str((1:NumObs)','%-d'));
iris = dataset({nominal(species),'species'},...
              {meas,'SL','SW','PL','PW'},...
              'ObsNames',NameObs);
iris = set(iris,'Description','Fisher''s Iris Data');
get(iris)
Description: 'Fisher's Iris Data'
Units: {}
DimNames: {'Observations' 'Variables'}
```

```
UserData: []  
ObsNames: {150x1 cell}  
VarNames: {'species' 'SL' 'SW' 'PL' 'PW'}
```

**See Also** `get` | `summary`

# CompactTreeBagger.setDefaultYfit

---

**Purpose** Set default value for predict

**Syntax** B = setDefaultYfit(B,Yfit)

**Description** B = setDefaultYfit(B,Yfit) sets the default prediction for ensemble B to Yfit. The default prediction must be a character variable for classification or a numeric scalar for regression. This setting controls what predicted value CompactTreeBagger returns when no prediction is possible, for example when the predict method needs to predict for an observation which has only false values in the matrix supplied through 'useifort' argument.

**See Also** predict | TreeBagger.DefaultYfit

**Purpose** Set difference for categorical arrays

---

**Note** In a future release, the behavior of `categorical.setdiff` will change to be consistent with the MATLAB function `setdiff`. This behavior change is optional in R2012a. For a demonstration of using the 'R2012a' flag to preview the future behavior, or the 'legacy' flag to preserve the current behavior in your existing code, see the documentation for `setdiff`.

---

**Syntax**

```
C = setdiff(A,B)
[C,IA] = setdiff(A,B)
[...] = setdiff(A,B,'rows')
[...] = setdiff(...,'R2012a')
[...] = setdiff(...,'legacy')
[...] = setdiff(A,B,setOrder)
[...] = setdiff(A,B,'rows',setOrder)
```

**Description** `C = setdiff(A,B)` for categorical vectors `A` and `B`, returns a categorical vector `C` containing the values in `A` that are not in `B` with no repetitions. The result `C` is sorted. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs.

`[C,IA] = setdiff(A,B)` also returns an index vector `IA` such that `C = A(IA)`. If there are repeated values in `A` that are not in `B`, then the index of the last occurrence of each repeated value is returned.

`[...] = setdiff(A,B,'rows')` for categorical matrices `A` and `B` with the same number of columns, returns the rows from `A` that are not in `B`. The rows of the matrix `C` are sorted. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs. The optional output `IA` is an index vector such that `C = C(IA,:)`.

`[...] = setdiff(...,'R2012a')` adopts the future behavior of `setdiff`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

# categorical.setdiff

---

`[...] = setdiff(..., 'legacy')` preserves the current behavior of `setdiff`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

`[...] = setdiff(A,B,setOrder)` and `[...] = setdiff(A,B, 'rows', setOrder)` returns the observations of `C` in a specific order. `setOrder='sorted'` returns the values or rows of `C` in sorted order. `setOrder='stable'` returns the values or rows of `C` in the same order as `A`. If `A` is a row vector, then `C` is also a row vector. Otherwise, `C` is a column vector. If `A` is a column vector. If there are repeated values in `A` that are not in `B`, then the index of the first occurrence of each repeated value is returned.

## See Also

`intersect` | `ismember` | `setxor` | `union` | `unique`



---

<b>Purpose</b>	Set difference for dataset array observations
<b>Syntax</b>	<pre>C = setdiff(A,B) C = setdiff(A,B,vars) C = setxor(A,B,vars,setOrder) [C,iA] = setxor( ___ )</pre>
<b>Description</b>	<p><code>C = setdiff(A,B)</code> for dataset arrays <code>A</code> and <code>B</code> returns the set of observations that are in <code>A</code> but not <code>B</code>, with repetitions removed. The observations in the dataset array <code>C</code> are sorted.</p> <p><code>C = setdiff(A,B,vars)</code> returns the set of observations that are in <code>A</code> but not <code>B</code>, considering only the variables specified in <code>vars</code>, with repetitions removed. The observations in the dataset array <code>C</code> are sorted by these variables. The values for variables not specified in <code>vars</code> for each observation in <code>C</code> are taken from the corresponding observation in <code>A</code>. If there are multiple observations in <code>A</code> that correspond to an observation in <code>C</code>, those values are taken from the first occurrence.</p> <p><code>C = setxor(A,B,vars,setOrder)</code> returns the observations in <code>C</code> in the order specified by <code>setOrder</code>.</p> <p><code>[C,iA] = setxor( ___ )</code> also returns the index vector <code>iA</code> such that <code>C = A(iA,:)</code>. If there are repeated observations in <code>A</code>, then <code>setxor</code> returns the index of the first occurrence. You can use any of the previous input arguments.</p>
<b>Input Arguments</b>	<p><b>A,B</b> Input dataset arrays.</p> <p><b>vars</b> Cell array of strings containing variable names or a vector of integers containing variable column numbers, indicating the variables that <code>setdiff</code> considers.</p> <p>Specify <code>vars</code> as <code>[]</code> to use its default value of all variables.</p> <p><b>setOrder</b></p>

# dataset.setdiff

---

Flag indicating the sorting order for the observations in **C**. The possible values of `setOrder` are:

'sorted'	Observations in <b>C</b> are in sorted order (default).
'stable'	Observations in <b>C</b> are in the same order that they appear in <b>A</b> .

## Output Arguments

**C**

Dataset array with the observations that are in **A** but not **B**, with repetitions removed. **C** is in sorted order (by default), or the order specified by `setOrder`.

**iA**

Index vector, indicating the observations from **A** that are in **C**. The vector `iA` contains the index to the first occurrence of any repeated observations in **A**.

## Examples

### Set Difference of Two Dataset Arrays

Create a scalar structure array, and then convert it into two dataset arrays.

```
S(1,1).Name = 'CLARK';  
S(1,1).Gender = 'M';  
S(1,1).SystolicBP = 124;  
S(1,1).DiastolicBP = 93;
```

```
S(2,1).Name = 'BROWN';  
S(2,1).Gender = 'F';  
S(2,1).SystolicBP = 122;  
S(2,1).DiastolicBP = 80;
```

```
S(3,1).Name = 'MARTIN';  
S(3,1).Gender = 'M';  
S(3,1).SystolicBP = 130;
```

```
S(3,1).DiastolicBP = 92;  
  
A = struct2dataset(S(1:2));  
B = struct2dataset(S(2:3));
```

The intersection of A and B is the second observation, with last name BROWN.

Return the set difference of A and B.

```
[C,iA] = setdiff(A,B)
```

C =

Name	Gender	SystolicBP	DiastolicBP
'CLARK'	'M'	124	93

iA =

1

The first observation in A is not present in B.

## See Also

[dataset](#) | [intersect](#) | [ismember](#) | [setxor](#) | [sortrows](#) | [union](#) | [unique](#) |

## Concepts

- “Dataset Arrays” on page 2-135

# categorical.setlabels

---

**Purpose** Label levels

**Syntax**  
A = setlabels(A,labels)  
A = setlabels(A,labels,levels)

**Description** A = setlabels(A,labels) labels the levels in the categorical array A using the cell array of strings or 2-D character matrix labels. Labels are assigned in the order given in labels.

A = setlabels(A,labels,levels) labels only the levels specified in the cell array of strings or 2-D character matrix levels.

## Examples

### Example 1

Relabel the species in Fisher's iris data using new categories:

```
load fisheriris
species = nominal(species);
species = mergelevels(...
    species,{'setosa','virginica'},'parent');
species = setlabels(species,'hybrid','versicolor');
getlabels(species)
ans =
    'hybrid'    'parent'
```

### Example 2

- 1 Load patient data from the CSV file hospital.dat and store the information in a dataset array with observation names given by the first column in the data (patient identification):

```
patients = dataset('file','hospital.dat',...
    'delimiter',';',...
    'ReadObsNames',true);
```

- 2 Make the {0,1}-valued variable smoke nominal, and change the labels to 'No' and 'Yes':

```
patients.smoke = nominal(patients.smoke,{'No','Yes'});
```

- 3 Add new levels to `smoke` as placeholders for more detailed histories of smokers:

```
patients.smoke = addlevels(patients.smoke,...
                           {'0-5 Years', '5-10 Years', 'LongTerm'});
```

- 4 Assuming the nonsmokers have never smoked, relabel the 'No' level:

```
patients.smoke = setlabels(patients.smoke, 'Never', 'No');
```

- 5 Drop the undifferentiated 'Yes' level from `smoke`:

```
patients.smoke = droplevels(patients.smoke, 'Yes');
```

Warning: OLDLEVELS contains categorical levels that were present in A, caused some array elements to have undefined levels.

Note that smokers now have an undefined level.

- 6 Set each smoker to one of the new levels, by observation name:

```
patients.smoke('YPL-320') = '5-10 Years';
```

### See Also

`getlabels`

# categorical.setxor

---

**Purpose** Set exclusive-or for categorical arrays

---

**Note** In a future release, the behavior of `categorical.setxor` will change to be consistent with the MATLAB function `setxor`. This behavior change is optional in R2012a. For a demonstration of using the 'R2012a' flag to preview the future behavior, or the 'legacy' flag to preserve the current behavior in your existing code, see the documentation for `setxor`.

---

**Syntax**

```
C = setxor(A,B)
[C,IA,IB] = setxor(A,B)
[...] = setxor(A,B,'rows')
[...] = setxor(...,'R2012a')
[...] = setxor(...,'legacy')
[...] = setxor(A,B,setOrder)
[...] = setxor(A,B,'rows',setOrder)
```

**Description** `C = setxor(A,B)` for categorical vectors `A` and `B`, returns a categorical vector `C` containing the values not in the intersection of `A` and `B` with no repetitions. The result `C` is sorted. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs.

`[C,IA,IB] = setxor(A,B)` also returns index vectors `IA` and `IB` such that `C` is a sorted combination of the values `A(IA)` and `B(IB)`. If there are repeated values that are not in the intersection in `A` or `B` then the index of the last occurrence of each repeated value is returned.

`[...] = setxor(A,B,'rows')` for categorical matrices `A` and `B` with the same number of columns, returns the rows that are not in the intersection of `A` and `B`. The rows of the matrix `C` are sorted. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs. The optional outputs `IA` and `IB` are index vectors such that `C` is the sorted combination of rows `A(IA,:)` and `B(IB,:)`.

`[...] = setxor(..., 'R2012a')` adopts the future behavior of `setxor`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

`[...] = setxor(..., 'legacy')` preserves the current behavior of `setxor`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

`[...] = setxor(A,B,setOrder)` and `[...] = setxor(A,B,'rows',setOrder)` returns the observations of `C` in a specific order. `setOrder='sorted'` returns the values or rows of `C` in sorted order. `setOrder='stable'` returns the values or rows of `C` in the same order as `A`. If `A` and `B` are row vectors, then `C` is also a row vector. Otherwise, `C` is a column vector. `IA` and `IB` are column vectors. If there are repeated values that are not in the intersection of `A` and `B`, then the index of the first occurrence of each repeated value is returned.

### See Also

`intersect` | `ismember` | `setdiff` | `union` | `unique`

# dataset.setxor

---

**Purpose** Set exclusive or for dataset array observations

**Syntax**

```
C = setxor(A,B)
C = setxor(A,B,vars)
C = setxor(A,B,vars,setOrder)
[C,iA,iB] = setxor( ___ )
```

**Description** `C = setxor(A,B)` for dataset arrays `A` and `B` returns the set of observations that are not in the intersection of the two arrays, with repetitions removed. The observations in the dataset array `C` are sorted.

`C = setxor(A,B,vars)` returns the set of observations that are not in the intersection of the two arrays, considering only the variables specified in `vars`, with repetitions removed. The observations in the dataset array `C` are sorted by these variables. The values for variables not specified in `vars` for each observation in `C` are taken from the corresponding observation in `A` or `B`. If there are multiple observations in `A` or `B` that correspond to an observation in `C`, those values are taken from the first occurrence.

`C = setxor(A,B,vars,setOrder)` returns the observations in `C` in the order specified by `setOrder`.

`[C,iA,iB] = setxor( ___ )` also returns index vectors `iA` and `iB` such that `C` is a sorted combination of the values `A(iA,:)` and `B(iB,:)`. If there are repeated observations in `A` or `B`, then `setxor` returns the index of the first occurrence. You can use any of the previous input arguments.

## Input Arguments

**A,B**  
Input dataset arrays.

**vars**  
Cell array of strings containing variable names or a vector of integers containing variable column numbers, indicating the variables in `A` and `B` that `setxor` considers.

Specify `vars` as `[]` to use its default value of all variables.



**setOrder**

Flag indicating the sorting order for the observations in **C**. The possible values of **setOrder** are:

'sorted'	Observations in <b>C</b> are in sorted order (default).
'stable'	Observations in <b>C</b> are in the same order that they appear in <b>A</b> , then <b>B</b> .

**Output Arguments****C**

Dataset array with the observations not in the intersection of **A** and **B**, with repetitions removed. **C** is in sorted order (by default), or the order specified by **setOrder**.

**iA**

Index vector, indicating the observations from **A** that are in **C**. The vector **iA** contains the index to the first occurrence of any repeated observations in **A**.

**iB**

Index vector, indicating the observations from **B** that are in **C**. The vector **iB** contains the index to the first occurrence of any repeated observations in **B**.

**Examples****Symmetric Difference of Two Dataset Arrays**

Create a scalar structure array, and then convert it into two dataset arrays.

```
S(1,1).Name = 'CLARK';
S(1,1).Gender = 'M';
S(1,1).SystolicBP = 124;
S(1,1).DiastolicBP = 93;

S(2,1).Name = 'BROWN';
```

# dataset.setxor

---

```
S(2,1).Gender = 'F';  
S(2,1).SystolicBP = 122;  
S(2,1).DiastolicBP = 80;
```

```
S(3,1).Name = 'MARTIN';  
S(3,1).Gender = 'M';  
S(3,1).SystolicBP = 130;  
S(3,1).DiastolicBP = 92;
```

```
A = struct2dataset(S(1:2));  
B = struct2dataset(S(2:3));
```

The intersection of A and B is the second observation, with last name BROWN.

Return the symmetric difference of A and B.

```
[C,iA,iB] = setxor(A,B);  
C
```

C =

Name	Gender	SystolicBP	DiastolicBP
'CLARK'	'M'	124	93
'MARTIN'	'M'	130	92

```
[iA iB]
```

```
ans =
```

```
1    2
```

The symmetric difference contains the first observation from A, and the second observation from B.

## See Also

[dataset](#) | [intersect](#) | [ismember](#) | [setdiff](#) | [sortrows](#) | [union](#) | [unique](#) |

**Concepts**

- “Dataset Arrays” on page 2-135

# gmdistribution.SharedCov property

---

**Purpose** true if all covariance matrices are restricted to be the same

**Description** Logical true if all the covariance matrices are restricted to be the same (pooled estimate); logical false otherwise.

**Purpose** Shift dimensions of categorical array

**Syntax** `B = shiftdim(A,n)`  
`[B,nshifts] = shiftdim(A)`

**Description** `B = shiftdim(A,n)` shifts the dimensions of the categorical array `A` by `N`. When `n` is positive, `shiftdim` shifts the dimensions to the left and wraps the `n` leading dimensions to the end. When `n` is negative, `shiftdim` shifts the dimensions to the right and pads with singletons.

`[B,nshifts] = shiftdim(A)` returns the array `B` with the same number of elements as `A` but with any leading singleton dimensions removed. `nshifts` returns the number of dimensions that are removed. If `A` is a scalar, `shiftdim` has no effect.

**See Also** `circshift` | `reshape` | `squeeze`

# RegressionEnsemble.shrink

---

**Purpose** Prune ensemble

**Syntax**  
`cmp = shrink(ens)`  
`cmp = shrink(ens,Name,Value)`

**Description** `cmp = shrink(ens)` returns a compact shrunken version of `ens`, a regularized ensemble. `cmp` retains only learners with weights above a threshold.

`cmp = shrink(ens,Name,Value)` returns an ensemble with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

**ens**  
A regression ensemble created with `fitensemble`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### 'lambda'

Vector of nonnegative regularization parameter values for lasso. If `ens.Regularization` is nonempty (populate it with `regularize`), `shrink` regularizes `ens` using `lambda`. If `ens` contains a `Regularization` structure, you cannot pass `lambda`.

**Default:** []

### 'threshold'

Lower cutoff on weights for weak learners, a numeric nonnegative scalar. `shrink` creates `cmp` from those learners with weights above threshold.

**Default:** 0

## 'weightcolumn'

Column index of `ens.Regularization.TrainedWeights`, a positive integer. `shrink` creates `cmp` with learner weights from this column.

**Default:** 1

## Output Arguments

### `cmp`

A regression ensemble of class `CompactRegressionEnsemble`. Use `cmp` for making predictions exactly as you use `ens`, with the `predict` method.

`shrink` orders the members of `cmp` from largest to smallest.

## Examples

Shrink a 300-member bagged regression ensemble using 0.1 for the parameter `lambda`, and view the number of members of the resulting ensemble:

```
X = rand(2000,20);
Y = repmat(-1,2000,1);
Y(sum(X(:,1:5),2)>2.5) = 1;
bag = fitensemble(X,Y,'Bag',300,'Tree','type','regression');
cmp = shrink(bag,'lambda',0.1);
cmp.NTrained
```

```
ans =
    83
```

## See Also

`regularize` | `cvshrink` | `predict`

# RegressionEnsemble.shrink

---

## **Tutorials**

- “Ensemble Regularization” on page 16-103



## gmdistribution.Sigma property

---

**Purpose** Input array of covariances

**Description** Input array of covariances SIGMA.

# signrank

---

**Purpose** Wilcoxon signed rank test

**Syntax**

```
p = signrank(x)
p = signrank(x,y)
p = signrank(x,y,Name,Value)
[p,h] = signrank( __ )
[p,h,stats] = signrank( __ )
```

```
[ __ ] = signrank(x,m)
[ __ ] = signrank(x,m,Name,Value)
```

**Description** `p = signrank(x)` returns the  $p$ -value of a two-sided Wilcoxon signed rank test.

`signrank` tests the null hypothesis that data in the vector `x` come from a distribution whose median is zero at the 5% significance level. The test assumes that the data in `x` come from a continuous distribution symmetric about its median.

`p = signrank(x,y)` returns the  $p$ -value of a paired, two-sided test for the null hypothesis that  $x - y$  comes from a distribution with zero median.

`p = signrank(x,y,Name,Value)` returns the  $p$ -value for the sign test with additional options specified by one or more `Name,Value` pair arguments.

`[p,h] = signrank( __ )` also returns a logical value indicating the test decision.  $h = 1$  indicates a rejection of the null hypothesis, and  $h = 0$  indicates a failure to reject the null hypothesis at the 5% significance level. You can use any of the input arguments in the previous syntaxes.

`[p,h,stats] = signrank( __ )` also returns the structure `stats` with information about the test statistic.

[ \_\_\_ ] = `signrank(x,m)` returns any of the output arguments in the previous syntaxes for the null hypothesis that the data in `x` are observations from a distribution with median `m`.

[ \_\_\_ ] = `signrank(x,m,Name,Value)` returns any of the output arguments in the previous syntaxes for the signed rank test with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **x - Sample data**

vector

Sample data, specified as a vector.

#### **Data Types**

single | double

### **y - Sample data**

vector

Sample data, specified as a vector. `y` must be the same length as `x`.

#### **Data Types**

single | double

### **m - Hypothesized value of the median**

scalar

Hypothesized value of the median, specified as a scalar.

**Example:** `signrank(x,10)`

#### **Data Types**

single | double

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** 'alpha',0.01,'method','approximate','tail','right'  
specifies a right-tailed signed rank test with 1% significance level,  
which returns the approximate p-value.

## 'alpha' - Significance level

0.05 (default) | scalar value in the range 0 to 1

Significance level of the decision of a hypothesis test, specified as the comma-separated pair consisting of 'alpha' and a scalar value in the range 0 to 1. Significance level of h is  $100 * \text{alpha}\%$ .

**Example:** 'alpha', 0.01

## Data Types

double | single

## 'method' - Computation method of p

'exact' | 'approximate'

Computation method of p, specified as the comma-separated pair consisting of 'method' and one of the following.

'exact'	Exact computation of the $p$ -value, $p$ . Default value for 15 or fewer observations in $x$ , $x - m$ , or $x - y$ when <code>method</code> is unspecified.
'approximate'	Normal approximation while computing the $p$ -value, $p$ . Default value for more than 15 observations in $x$ , $x - m$ , or $x - y$ when 'method' is unspecified because the exact method can be slow on large samples.

**Example:** 'method','exact'

## Data Types

char

## 'tail' - Type of test

'both' (default) | 'right' | 'left'

Type of test, specified as the comma-separated pair consisting of 'tail' and one of the following:

'both'	Two-sided hypothesis test, which is the default test type. <ul style="list-style-type: none"> <li>• For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median different than 0 or <math>m</math>.</li> <li>• For a two-sample test, the alternate hypothesis states that the data in <math>x - y</math> come from a distribution with median different than 0.</li> </ul>
'right'	Right-tailed hypothesis test. <ul style="list-style-type: none"> <li>• For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median greater than 0 or <math>m</math>.</li> <li>• For a two-sample test, the alternate hypothesis states the data in <math>x - y</math> come from a distribution with median greater than 0.</li> </ul>
'left'	Left-tailed hypothesis test. <ul style="list-style-type: none"> <li>• For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median less than 0 or <math>m</math>.</li> <li>• For a two-sample test, the alternate hypothesis states the data in <math>x - y</math> come from a distribution with median less than 0.</li> </ul>

Example: 'tail','left'

## Output Arguments

### **p** - *p*-value of the test

nonnegative scalar

*p*-value of the test, returned as a nonnegative scalar from 0 to 1. *p* is the probability of observing a test statistic as or more extreme than the observed value under the null hypothesis. `signrank` computes the two-sided *p*-value by doubling the most significant one-sided value.

## **h - Result of the hypothesis test**

1 | 0

Result of the hypothesis test, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the  $100 * \alpha\%$  significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the  $100 * \alpha\%$  significance level.

## **stats - Test statistics**

structure

Test statistics, returned as a structure. The test statistics stored in `stats` are:

- `signrank`: Value of the sign rank test statistic.
- `zval`: Value of the  $z$ - statistic (computed when 'method' is 'approximate').

## **Examples**

### **Test for Zero Median of a Single Population**

Test the hypothesis of zero median.

Generate the sample data.

```
rng('default') % for reproducibility  
x = randn(1,25) + 1.30;
```

Test the hypothesis that the data in `x` has zero median.

```
[p,h] = signrank(x)
```

```
p =
```

```
3.2229e-05
```

```
h =
```

1

At the default 5% significance level, the value  $h = 1$  indicates that the test rejects the null hypothesis of zero median.

### Test the Median of Differences of Paired Samples

Test the hypothesis of zero median for the difference between paired samples.

Generate the sample data.

```
rng('default') % for reproducibility
x = lognrnd(2, .25, 10, 1);
y = x + trnd(2, 10, 1);
```

Test the hypothesis that  $x - y$  has zero median.

```
[p,h] = signrank(x,y)
```

```
p =
    0.3223
```

```
h =
    0
```

The results indicate that the test fails to reject the null hypothesis of zero median in the difference at the default 5% significance level.

### Signed Rank Test for Large Samples

Conduct a -sided test on a large sample using approximation.

Navigate to a folder containing sample data.

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

Load the sample data.

```
load gradespaired
```

Test the null hypothesis that the median of the grade differences of students before and after participating in a tutoring program is 0 against the alternate that it is less than 0.

```
[p,h,stats] = signrank(gradespaired(:,1),...  
                      gradespaired(:,2),'tail','left')
```

```
p =
```

```
    0.0047
```

```
h =
```

```
    1
```

```
stats =
```

```
      zval: -2.5982  
 signedrank: 2.0175e+03
```

Because the sample size is greater than 15, `signrank` uses an approximate method to calculate the  $p$ -value and also returns the value of the  $z$ -statistic. The value  $h = 1$  indicates that the test rejects the null hypothesis that there is no difference between the grade medians at the 5% significance level. There is enough statistical evidence to conclude that the median grade before the tutoring program is less than the median grade after the tutoring program.

Repeat the test using the exact method.

```
[p,h,stats] = signrank(gradespaired(:,1),gradespaired(:,2),...  
                      'tail','left','method','exact')
```

```
p =
```



```
0.0045
```

```
h =
```

```
1
```

```
stats =
```

```
signedrank: 2.0175e+03
```

The results obtained using the approximate method are consistent with the exact method.

### **Two-Sided Test for the Median of a Single Population**

Load the sample data.

```
load mileage
```

The data contains the mileages per gallon for three different types of cars in columns 1 to 3.

Test the hypothesis that the median mileage for the type of cars in the second column differs from 33.

```
[p,h,stats] = signrank(mileage(:,2),33)
```

```
p =
```

```
0.0313
```

```
h =
```

```
1
```

```
stats =
```

```
signedrank: 21
```

At the 5% significance level, the results indicate that the median mileage for the second type of cars differs from 33. Note that `signrank` uses an exact method to calculate the  $p$ -value for small samples and does not return the  $z$ -statistic.

## Right-Sided Test for the Median of a Single Population

Use the name-value pair arguments in `signrank`.

Load the sample data.

```
load mileage
```

The data contains the mileage per gallon for three different types of cars in columns 1 to 3.

Test the hypothesis that the median mileage for the type of cars in the second row are larger than 33.

```
[p,h,stats] = signrank(mileage(:,2),33,'tail','right')
```

```
p =  
    0.0156
```

```
h =  
    1
```

```
stats =  
    signedrank: 21
```

Repeat the same test at the 1% significance level using the approximate method.

```
[p,h,stats] = signrank(mileage(:,2),33,'tail','right',...  
    'alpha',0.01,'method','approximate')
```

```
p =  
    0.0180
```

```

h =
    0

stats =
    zval: 2.0966
    signedrank: 21

```

This result,  $h = 0$ , indicates that the null hypothesis cannot be rejected at the 1% significance level.

## Definitions

### Wilcoxon Signed Rank Test

The Wilcoxon signed rank test is a nonparametric test for two populations when the observations are paired. In this case, the test statistic,  $W$ , is the sum of the ranks of positive differences between the observations in the two samples (that is,  $x - y$ ). When you use the test for one sample, then  $W$  is the sum of the ranks of positive differences between the observations and the hypothesized median value  $M_0$  (which is 0 when you use `signrank(x)` and  $m$  when you use `signrank(x, m)`).

### z-Statistic

For large samples, or when `method` is `approximate`, the `signrank` function calculates the  $p$ -value using the  $z$ -statistic, given by

$$z = \frac{(W - n(n+1)/4)}{\sqrt{\frac{n(n+1)(2n+1) - \text{tieadj}}{24}}},$$

where  $n$  is the sample size of the difference  $x - y$  or  $x - m$ . For the two-sample case, `signrank` uses `[tie_rank, tieadj] = tiedrank(abs(diffxy), 0, 0, epsdiff)` to obtain the tie adjustment value `tieadj`.

# signrank

---

## Algorithms

signrank treats NaNs in  $x$  and  $y$  as missing values and ignores them.

For the two-sample case, signrank uses a tolerance based on the values  $\text{epsdiff} = \text{eps}(x) + \text{eps}(y)$ . The signrank function treats any pair of values with difference  $d(i) = x(i) - y(i)$  that differ by no more than the sum of their two eps values ( $\text{abs}(d(i)) < \text{epsdiff}(i)$ ) as ties.

## References

[1] Gibbons, J. D., and S. Chakraborti. *Nonparametric Statistical Inference*, 5th Ed., Boca Raton, FL: Chapman & Hall/CRC Press, Taylor & Francis Group, 2011.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

## See Also

ranksum | signtest | ttest | ztest

**Purpose**

Sign test

**Syntax**

```
p = signtest(x)
p = signtest(x,y)
p = signtest(x,y,Name,Value)
[p,h] = signtest( __ )
[p,h,stats] = signtest( __ )

[ __ ] = signtest(x,m)
[ __ ] = signtest(x,m,Name,Value)
```

**Description**

`p = signtest(x)` returns the  $p$ -value for a two-sided sign test.

`signtest` tests the hypothesis that data in `x` has a continuous distribution with zero median against the alternative that the distribution does not have zero median at the 5% significance level.

`p = signtest(x,y)` returns the  $p$ -value of a two-sided sign test. Here, `signtest` tests for the hypothesis that the data in `x - y` has a distribution with zero median against the alternative that the distribution does not have zero median. Note that a hypothesis of zero median for `x - y` is not equivalent to a hypothesis of equal median for `x` and `y`.

`p = signtest(x,y,Name,Value)` returns the  $p$ -value for the sign test with additional options specified by one or more `Name,Value` pair arguments.

`[p,h] = signtest( __ )` also returns a logical value indicating the test decision. The value `h = 1` indicates a rejection of the null hypothesis, and `h = 0` indicates a failure to reject the null hypothesis at the 5% significance level. You can use any of the input arguments in the previous syntaxes.

`[p,h,stats] = signtest( ___ )` also returns the structure `stats` containing information about the test statistic.

`[ ___ ] = signtest(x,m)` returns any of the output arguments in the previous syntaxes for the test whether the data in `x` are observations from a distribution with median `m` against the alternative that the median is different from `m`.

`[ ___ ] = signtest(x,m,Name,Value)` returns any of the output arguments in the previous syntaxes for the sign test with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **x - Sample data**

vector

Sample data, specified as a vector.

#### **Data Types**

single | double

### **y - Sample data**

vector

Sample data, specified as a vector. `y` must be the same length as `x`.

#### **Data Types**

single | double

### **m - Hypothesized value of the median**

scalar

Hypothesized value of the median, specified as a scalar.

**Example:** `signtest(x,35)`

#### **Data Types**

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'alpha', 0.01, 'method', 'approximate', 'tail', 'right'` specifies a right-tailed sign test with 1% significance level, which returns the approximate  $p$ -value.

### 'alpha' - Significance level

0.05 (default) | scalar value in the range 0 to 1

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'alpha'` and a scalar value in the range 0 to 1. The default value of `alpha` is 0.05. Significance level of  $h$  is  $100 * \alpha\%$ .

**Example:** `'alpha', 0.01`

### Data Types

double | single

### 'method' - $p$ -value computation method

`'exact'` | `'approximate'`

$p$ -value computation method, specified as the comma-separated pair consisting of `'method'` and one of the following:

<code>'exact'</code>	Exact computation of the $p$ -value, $p$ .
<code>'approximate'</code>	Normal approximation for computing the $p$ -value, $p$ .

The default computation method is `'exact'`, if there are fewer than 100 observations and `'approximate'` if there are 100 observations or more.

**Example:** `'method', 'exact'`

### Data Types

char

# signtest

---

## 'tail' - Type of test

'both' (default) | 'right' | 'left'

Type of test, specified as the comma-separated pair consisting of 'tail' and one of the following:

'both'	Two-sided hypothesis test, which is the default test type. <ul style="list-style-type: none"><li>• For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median different than zero (or <math>m</math>).</li><li>• For a two-sample test, the alternate hypothesis states that the data in <math>x - y</math> come from a distribution with median different than zero.</li></ul>
'right'	Right-tailed hypothesis test. <ul style="list-style-type: none"><li>• For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median greater than zero (or <math>m</math>).</li><li>• For a two-sample test, the alternate hypothesis states the data in <math>x - y</math> come from a distribution with median greater than zero.</li></ul>
'left'	Left-tailed hypothesis test. <ul style="list-style-type: none"><li>• For a one-sample test, the alternate hypothesis states that the data in <math>x</math> come from a continuous distribution with median less than zero (or <math>m</math>).</li><li>• For a two-sample test, the alternative hypothesis states the data in <math>x - y</math> come from a distribution with median less than zero.</li></ul>

Example: 'tail', 'left'



## Output Arguments

### **p** - *p*-value of the test

nonnegative scalar

*p*-value of the test, returned as a nonnegative scalar from 0 to 1. *p* is the probability of observing a test statistic as or more extreme than the observed value under the null hypothesis. `signtest` computes the two-sided *p*-value by doubling the most significant one-sided value.

### **h** - Result of the hypothesis test

1 | 0

Result of the hypothesis test, returned as a logical value.

- If  $h = 1$ , this indicates rejection of the null hypothesis at the  $100 * \alpha\%$  significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the  $100 * \alpha\%$  significance level.

### **stats** - Test statistics

structure

Test statistics, returned as a structure. The test statistics stored in `stats` are:

- `sign`: Value of the sign test statistic.
- `zval`: Value of the z-statistic (computed only for large samples).

## Examples

### **Test for Zero Median of a Single Population**

Test the hypothesis of zero median.

Generate the sample data.

```
rng('default') % for reproducibility
x = randn(1,25);
```

The sampling distribution of *x* is symmetric with zero median.

Test the null hypothesis that *x* comes from a distribution with a median different from zero median.

```
[p,h,stats] = signtest(x,0)
```

```
p =  
    0.1078
```

```
h =  
    0
```

```
stats =  
    zval: NaN  
    sign: 17
```

At the default 5% significance level, the result  $h = 0$  indicates that `signtest` fails to reject to the null hypothesis of zero median. `signtest` calculates the  $p$ -value using the exact method, hence it does not calculate `zval` and returns it as a NaN.

## Test for Zero Median for the Difference of Paired Samples

Test the hypothesis of zero median for the difference between paired samples.

Generate the sample data.

```
rng('default') % for reproducibility  
before = lognrnd(2, .25, 10, 1);  
after = before + (lognrnd(0, .5, 10, 1) - 1);
```

The sampling distribution of the difference between `before` and `after` is symmetric with zero median.

Test the null hypothesis that the difference of `before` and `after` has zero median.

```
[p,h] = signtest(before,after)
```

```
p =  
    0.7539
```

```
h =
```

```
0
```

At the default 5% significance level, the value  $h = 0$  indicates that `signtest` fails to reject to the null hypothesis of zero median in the difference.

### Medians of Paired Samples

Test the hypothesis of zero median for the difference between two paired samples using the exact and approximate methods.

Generate the sample data.

```
rng('default') % for reproducibility
x = lognrnd(2,.25,15,1);
y = x + trnd(2,15,1);
display([x y])
```

```
ans =
```

```
8.4521    7.8047
11.6869   11.4094
4.2009    5.1133
9.1664   12.1655
8.0020   10.0300
5.3285    6.0153
6.6300    5.1235
8.0499    8.6737
18.0763   19.2164
14.7665   15.3380
5.2726    8.4187
15.7798   16.2093
8.8583    8.5575
7.2735    7.4783
```

# signtest

---

```
8.8347 7.8894
```

Test the hypothesis that  $x - y$  has zero median.

```
[p,h,stats] = signtest(x,y)
```

```
p =
```

```
0.3018
```

```
h =
```

```
0
```

```
stats =
```

```
zval: NaN
```

```
sign: 5
```

At the default 5% significance level, the value  $h = 0$  indicates that the test fails to reject the null hypothesis of zero median in the difference.

Repeat the test using the approximate method.

```
[p,h,stats] = signtest(x,y,'method','approximate')
```

```
p =
```

```
0.3017
```

```
h =
```

```
0
```

```
stats =  
  
    zval: -1.0328  
    sign: 5
```

The approximate  $p$ -value, which `signtest` obtains using the  $z$ -statistic, is really close to the exact  $p$ -value.

### Test for Large Samples

Perform a left-sided sign test for large samples.

Navigate to a folder containing sample data.

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

Load the sample data.

```
load gradespaired
```

Test the null hypothesis that the median of the grade differences before and after the tutoring program is 0 against the alternate that it is less than 0.

```
[p,h,stats] = signtest(gradespaired(:,1),gradespaired(:,2),...  
'tail','left')
```

```
p =  
  
    0.0013
```

```
h =  
  
    1
```

```
stats =
```

# signtest

---

```
zval: -3.0110  
sign: 37
```

Because the sample size is large (greater than 100), `signtest` uses an approximate method to calculate the  $p$ -value and also returns the value of the  $z$ -statistic. The test rejects the null hypothesis that there is no difference between the grade medians at the 5% significance level.

## Test for Median of a Single Population

Test the hypothesis that the population median is different from a specified value.

Load the sample data.

```
load lawdata
```

The data set has 15 observations for variables `gpa` and `lsat`.

Test the hypothesis that the median `lsat` score is higher than 570.

```
[p,h,stats] = signtest(lsat,570,'tail','right')
```

```
p =
```

```
0.0176
```

```
h =
```

```
1
```

```
stats =
```

```
zval: NaN  
sign: 12
```

Both the  $p$ -value, 0.0176, and  $h = 1$  indicate that at the 5% significance level the test concludes in favor of the alternate hypothesis.

## Definitions

### Sign Test

The sign test is a nonparametric test for the median of a population or median of the difference of two populations.

For example, for tests on a single population median:

- If the test is two-sided, then the test statistic,  $S$ , is the minimum of the number of observations that are smaller or larger than the hypothesized median value,  $M_0$ .
- If the test is right-sided, then  $S$  is the number of observations that are larger than the hypothesized median value  $M_0$ .
- If the test is left-sided, then  $S$  is the number of observations that are smaller than the hypothesized median value  $M_0$ .

### z-Statistic

For a large sample, `signtest` uses the  $z$ -statistic to approximate the  $p$ -value.

The `signtest` test statistic is the number of elements that are greater than 0 (for `signtest(x)` or `signtest(x-y)`), or  $m$  (for `signtest(x, m)`). Hence, the  $z$ -statistic of the sign test, with the continuity correction, is:

$$z = \frac{(S - E(S))}{\sqrt{V(S)}} = \frac{(S - (0.5)n - 0.5\text{sign}(npos - nneg))}{\sqrt{(0.5)(0.5)n}},$$

where  $npos$  and  $nneg$  are the number of positive and negative differences from the hypothesized median value, respectively.

## Algorithms

For a one-sample test, `signtest` omits values in  $x$  that are zero or NaN.

For a two-sample test, `signtest` omits values in  $x - y$  that are zero or NaN.

## References

[1] Gibbons, J. D., and S. Chakraborti. *Nonparametric Statistical Inference*, 5th Ed. Boca Raton, FL: Chapman & Hall/CRC Press, Taylor & Francis Group, 2011.

[2] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.

## See Also

ranksum | signrank | ttest | ztest



**Purpose**

Silhouette plot

**Syntax**

```
silhouette(X,clust)
s = silhouette(X,clust)
[s,h] = silhouette(X,clust)
[...] = silhouette(X,clust,metric)
[...] = silhouette(X,clust,distfun,p1,p2,...)
```

**Description**

`silhouette(X,clust)` plots cluster silhouettes for the  $n$ -by- $p$  data matrix  $X$ , with clusters defined by `clust`. Rows of  $X$  correspond to points, columns correspond to coordinates. `clust` can be a categorical variable, numeric vector, character matrix, or cell array of strings containing a cluster name for each point. `silhouette` treats NaNs or empty strings in `clust` as missing values, and ignores the corresponding rows of  $X$ . By default, `silhouette` uses the squared Euclidean distance between points in  $X$ .

`s = silhouette(X,clust)` returns the silhouette values in the  $n$ -by-1 vector `s`, but does not plot the cluster silhouettes.

`[s,h] = silhouette(X,clust)` plots the silhouettes, and returns the silhouette values in the  $n$ -by-1 vector `s`, and the figure handle in `h`.

`[...] = silhouette(X,clust,metric)` plots the silhouettes using the inter-point distance function specified in `metric`. Choices for `metric` are given in the following table.

<b>Metric</b>	<b>Description</b>
'Euclidean'	Euclidean distance
'sqEuclidean'	Squared Euclidean distance (default)
'cityblock'	Sum of absolute differences
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values)

Metric	Description
'Hamming'	Percentage of coordinates that differ
'Jaccard'	Percentage of nonzero coordinates that differ
Vector	A numeric distance matrix in upper triangular vector form, such as is created by <code>pdist</code> . <code>X</code> is not used in this case, and can safely be set to <code>[]</code> .

For more information on each metric, see “Distance Metrics” on page 16-9.

`[...] = silhouette(X, clust, distfun, p1, p2, ...)` accepts a function handle `distfun` to a metric of the form

```
d = distfun(X0, X, p1, p2, ...)
```

where `X0` is a 1-by-`p` point, `X` is an `n`-by-`p` matrix of points, and `p1, p2, ...` are optional additional arguments. The function `distfun` returns an `n`-by-1 vector `d` of distances between `X0` and each point (row) in `X`. The arguments `p1, p2, ...` are passed directly to the function `distfun`.

## Definitions

### Silhouette Value

The silhouette value for each point is a measure of how similar that point is to points in its own cluster, when compared to points in other clusters. The silhouette value for the  $i$ th point,  $S_i$ , is defined as

$$S_i = (b_i - a_i) / \max(a_i, b_i)$$

where  $a_i$  is the average distance from the  $i$ th point to the other points in the same cluster as  $i$ , and  $b_i$  is the minimum average distance from the  $i$ th point to points in a different cluster, minimized over clusters.

The silhouette value ranges from -1 to +1. A high silhouette value indicates that  $i$  is well-matched to its own cluster, and poorly-matched to neighboring clusters. If most points have a high silhouette value, then the clustering solution is appropriate. If many points have a low or negative silhouette value, then the clustering solution may have

either too many or too few clusters. The silhouette clustering evaluation criterion can be used with any distance metric.

## Examples

### Create Silhouette Plot

Create a silhouette plot from clustered data.

Generate random sample data.

```
rng('default'); % For reproducibility  
X = [randn(10,2)+ones(10,2);randn(10,2)-ones(10,2)];
```

Cluster the data in  $X$  using `kmeans`.

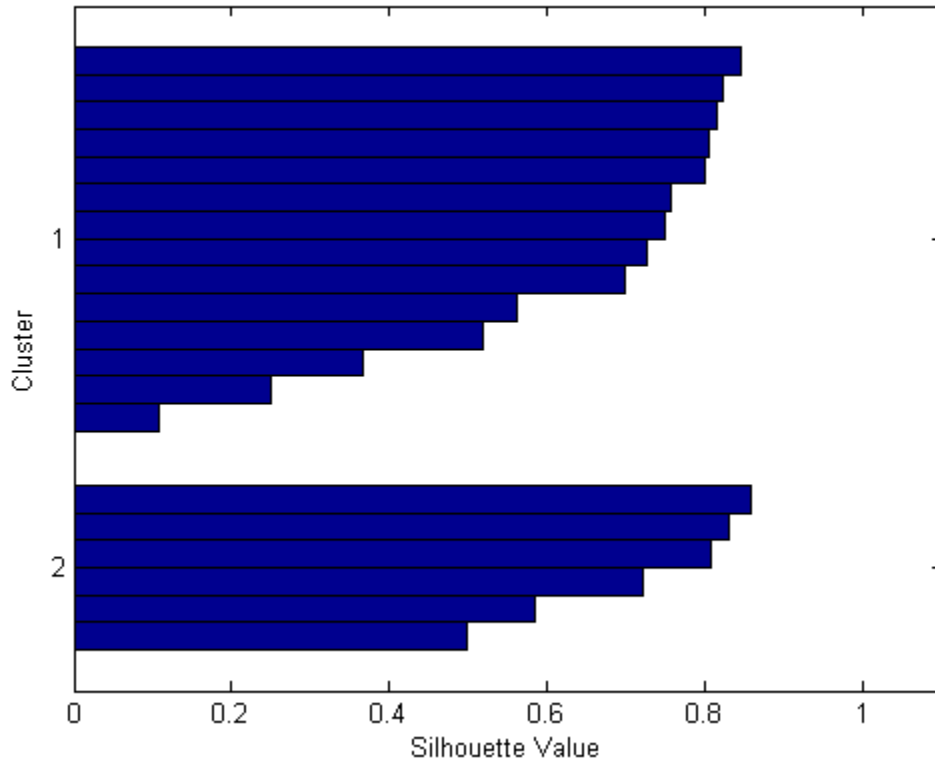
```
cidx = kmeans(X,2);
```

Create a silhouette plot from the clustered data.

```
silhouette(X,cidx)
```

# silhouette

---



## Compute Silhouette Values

Compute the silhouette values from clustered data.

Generate random sample data.

```
rng('default'); % For reproducibility  
X = [randn(10,2)+ones(10,2);randn(10,2)-ones(10,2)];
```

Use `kmeans` to cluster the data in `X` based on the sum of absolute differences in distance.

```
cidx = kmeans(X,2,'distance','cityblock');
```

Compute the silhouette values from the clustered data. Specify metric as 'cityblock' to indicate that the kmeans clustering is based on the sum of absolute differences.

```
s = silhouette(X,cidx,'cityblock')
```

```
s =
```

```
0.0816  
0.5848  
0.1906  
0.2781  
0.3954  
0.4050  
0.0897  
0.5416  
0.6203  
0.6664  
0.5814  
0.6022  
0.6540  
0.5223  
0.5566  
0.4227  
0.6225  
0.6558  
0.5284  
0.6034
```

## References

[1] Kaufman L., and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons, Inc., 1990.

## See Also

dendrogram | kmeans | linkage | pdist | evalclusters

## **How To**

- “Grouping Variables” on page 2-51

**Superclasses** ClusterCriterion

**Purpose** Silhouette criterion clustering evaluation object

**Description** clustering.evaluation.SilhouetteEvaluation is an object consisting of sample data, clustering data, and silhouette criterion values used to evaluate the optimal number of data clusters. Create a silhouette criterion clustering evaluation object using evalclusters.

**Construction** `eva = evalclusters(x,clust,'Silhouette')` creates a silhouette criterion clustering evaluation object.

`eva = evalclusters(x,clust,'Silhouette',Name,Value)` creates a silhouette criterion clustering evaluation object using additional options specified by one or more name-value pair arguments.

## Input Arguments

### **x - Input data**

matrix

Input data, specified as an  $N$ -by- $P$  matrix.  $N$  is the number of observations, and  $P$  is the number of variables.

### **Data Types**

single | double

### **clust - Clustering algorithm**

'kmeans' | 'linkage' | 'gmdistribution' | matrix of clustering solutions | function handle

Clustering algorithm, specified as one of the following.

# clustering.evaluation.SilhouetteEvaluation

---

'kmeans'	Cluster the data in $X$ using the kmeans clustering algorithm, with 'EmptyAction' set to 'singleton' and 'Replicates' set to 5.
'linkage'	Cluster the data in $X$ using the clusterdata agglomerative clustering algorithm, with 'Linkage' set to 'ward'.
'gmdistribution'	Cluster the data in $X$ using the gmdistribution Gaussian mixture distribution algorithm, with 'SharedCov' set to true and 'Replicates' set to 5.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can specify a clustering algorithm using the function\_handle (@) operator. The function must be of the form

```
C = clustfun(DATA,K)
```

where DATA is the data to be clustered, and K is the number of clusters. The output of clustfun must be one of the following:

- A vector of integers representing the cluster index for each observation in DATA. There must be K unique values in this vector.
- A numeric  $n$ -by- $K$  matrix of score for  $n$  observations and  $K$  classes. In this case, the cluster index for each observation is determined by taking the largest score value in each row.

If Criterion is 'CalinskHarabasz', 'DaviesBouldin', or 'silhouette', you can also specify clust as a  $n$ -by- $K$  matrix containing the proposed clustering solutions.  $n$  is the number of observations in the sample data, and  $K$  is the number of proposed clustering solutions. Column  $j$  contains the cluster indices for each of the  $N$  points in the  $j$ th clustering solution.



## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'KList',[1:5],'Distance','cityblock'` specifies to test 1, 2, 3, 4, and 5 clusters using the sum of absolute differences distance measure.

## **ClusterPriors - Prior probabilities for each cluster**

`'empirical'` (default) | `'equal'`

Prior probabilities for each cluster, specified as the comma-separated pair consisting of `'ClusterPriors'` and one of the following.

<code>'empirical'</code>	Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points. Each cluster contributes to the overall silhouette value proportionally to its size.
<code>'equal'</code>	Compute the overall silhouette value for the clustering solution by averaging the silhouette values for all points within each cluster, and then averaging those values across all clusters. Each cluster contributes equally to the overall silhouette value, regardless of its size.

**Example:** `'ClusterPriors','empirical'`

## **Distance - Distance metric**

`'sqEuclidean'` (default) | `'Euclidean'` | `'cityblock'` | vector | function | ...

# clustering.evaluation.SilhouetteEvaluation

---

Distance metric used for computing the criterion values, specified as the comma-separated pair consisting of 'Distance' and one of the following.

'sqEuclidean'	Squared Euclidean distance
'Euclidean'	Euclidean distance
'cityblock'	Sum of absolute differences
'cosine'	One minus the cosine of the included angle between points (treated as vectors)
'correlation'	One minus the sample correlation between points (treated as sequences of values)
'Hamming'	Percentage of coordinates that differ
'Jaccard'	Percentage of nonzero coordinates that differ

For detailed information about each distance metric, see `pdist`.

You can also specify a function for the distance metric by using the `function_handle` (@) operator. The distance function must be of the form

```
d2 = distfun(XI,XJ),
```

where `XI` is a 1-by- $n$  vector corresponding to a single row of the input matrix `X`, and `XJ` is an  $m_2$ -by- $n$  matrix corresponding to multiple rows of `X`. `distfun` must return an  $m_2$ -by-1 vector of distances `d2`, whose  $k$ th element is the distance between `XI` and `XJ(k, :)`.

If `Criterion` is 'silhouette', you can also specify `Distance` as the output vector output created by the function `pdist`.

When `Clust` a string representing a built-in clustering algorithm, `evalclusters` uses the distance metric specified for `Distance` to cluster the data, except for the following:

- If `Clust` is 'linkage', and `Distance` is either 'sqEuclidean' or 'Euclidean', then the clustering algorithm uses Euclidean distance and Ward linkage.
- If `Clust` is 'linkage' and `Distance` is any other metric, then the clustering algorithm uses the specified distance metric and average linkage.

In all other cases, the distance metric specified for `Distance` must match the distance metric used in the clustering algorithm to obtain meaningful results.

**Example:** 'Distance', 'Euclidean'

## **KList - List of number of clusters to evaluate**

vector

List of number of clusters to evaluate, specified as the comma-separated pair consisting of 'KList' and a vector of positive integer values. You must specify `KList` when `clust` is a clustering algorithm name string or a function handle. When `criterion` is 'gap', `clust` must be a string or a function handle, and you must specify `KList`.

**Example:** 'KList', [1:6]

## **Properties**

### **ClusteringFunction**

Clustering algorithm used to cluster the input data, stored as a valid clustering algorithm name string or function handle. If the clustering solutions are provided in the input, `ClusteringFunction` is empty.

### **ClusterPriors**

Prior probabilities for each cluster, stored as valid prior probability name string.

### **ClusterSilhouettes**

Silhouette values corresponding to each proposed number of clusters in `InspectedK`, stored as a cell array of vectors.

# clustering.evaluation.SilhouetteEvaluation

---

## Data Types

single | double

## CriterionName

Name of the criterion used for clustering evaluation, stored as a valid criterion name string.

## CriterionValues

Criterion values corresponding to each proposed number of clusters in `InspectedK`, stored as a vector of numerical values.

## Distance

Distance measure used for clustering data, stored as a valid distance measure name string.

## InspectedK

List of the number of proposed clusters for which to compute criterion values, stored as a vector of positive integer values.

## Missing

Logical flag for excluded data, stored as a column vector of logical values. If `Missing` equals `true`, then the corresponding value in the data matrix `x` is not used in the clustering solution.

## NumObservations

Number of observations in the data matrix `X`, minus the number of missing (NaN) values in `X`, stored as a positive integer value.

## OptimalK

Optimal number of clusters, stored as a positive integer value.

## OptimalY

Optimal clustering solution corresponding to `OptimalK`, stored as a column vector of positive integer values. If the clustering solutions are provided in the input, `OptimalY` is empty.

## X

Data used for clustering, stored as a matrix of numerical values.

## Methods

### Inherited Methods

compact	Compact clustering evaluation object
plot	Plot clustering evaluation object criterion values

## Definitions

### Silhouette Value

The silhouette value for each point is a measure of how similar that point is to points in its own cluster, when compared to points in other clusters. The silhouette value for the  $i$ th point,  $S_i$ , is defined as

$$S_i = (b_i - a_i) / \max(a_i, b_i)$$

where  $a_i$  is the average distance from the  $i$ th point to the other points in the same cluster as  $i$ , and  $b_i$  is the minimum average distance from the  $i$ th point to points in a different cluster, minimized over clusters.

The silhouette value ranges from -1 to +1. A high silhouette value indicates that  $i$  is well-matched to its own cluster, and poorly-matched to neighboring clusters. If most points have a high silhouette value, then the clustering solution is appropriate. If many points have a low or negative silhouette value, then the clustering solution may have either too many or too few clusters. The silhouette clustering evaluation criterion can be used with any distance metric.

## Examples

### Evaluate the Clustering Solution Using Silhouette Criterion

Evaluate the optimal number of clusters using the silhouette clustering evaluation criterion.

Generate sample data containing random numbers from three multivariate distributions with different parameter values.

```
rng('default'); % For reproducibility
```

# clustering.evaluation.SilhouetteEvaluation

---

```
mu1 = [2 2];
sigma1 = [0.9 -0.0255; -0.0255 0.9];

mu2 = [5 5];
sigma2 = [0.5 0 ; 0 0.3];

mu3 = [-2, -2];
sigma3 = [1 0 ; 0 0.9];

N = 200;

X = [mvnrnd(mu1,sigma1,N);...
     mvnrnd(mu2,sigma2,N);...
     mvnrnd(mu3,sigma3,N)];
```

Evaluate the optimal number of clusters using the silhouette criterion.  
Cluster the data using kmeans.

```
E = evalclusters(X,'kmeans','silhouette','klist',[1:6])
```

```
E =
```

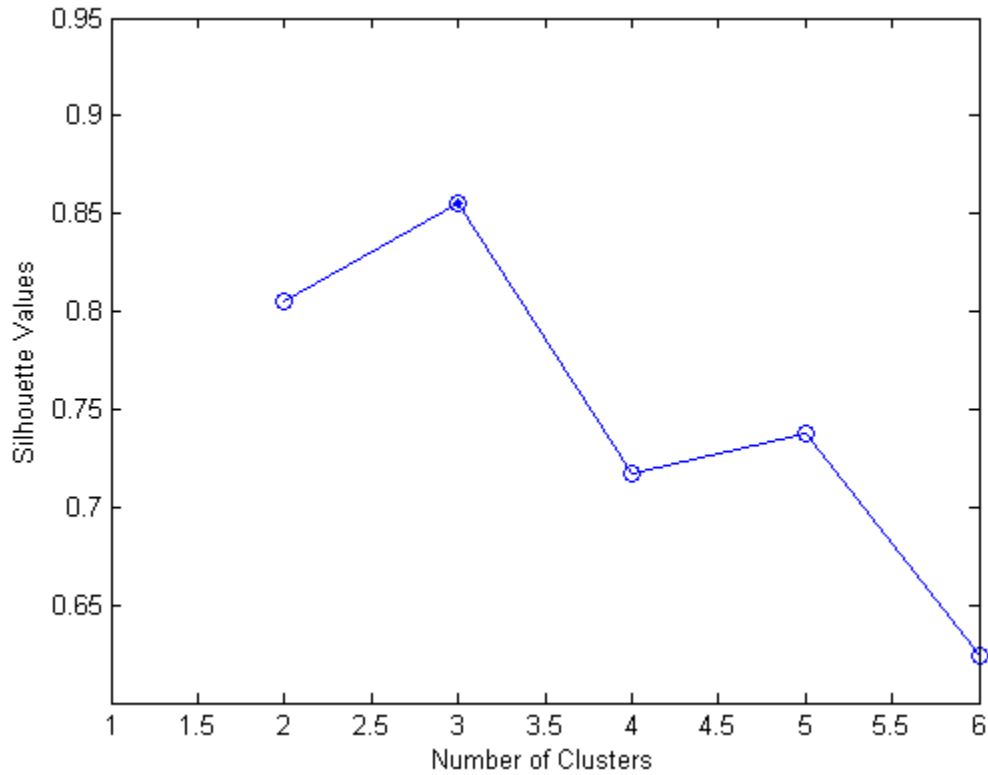
```
SilhouetteEvaluation with properties:
```

```
NumObservations: 600
   InspectedK: [1 2 3 4 5 6]
 CriterionValues: [NaN 0.8055 0.8551 0.7170 0.7376 0.6239]
   OptimalK: 3
```

The OptimalK value indicates that, based on the silhouette criterion,  
the optimal number of clusters is three.

Plot the silhouette criterion values for each number of clusters tested.

```
figure;
plot(E)
```



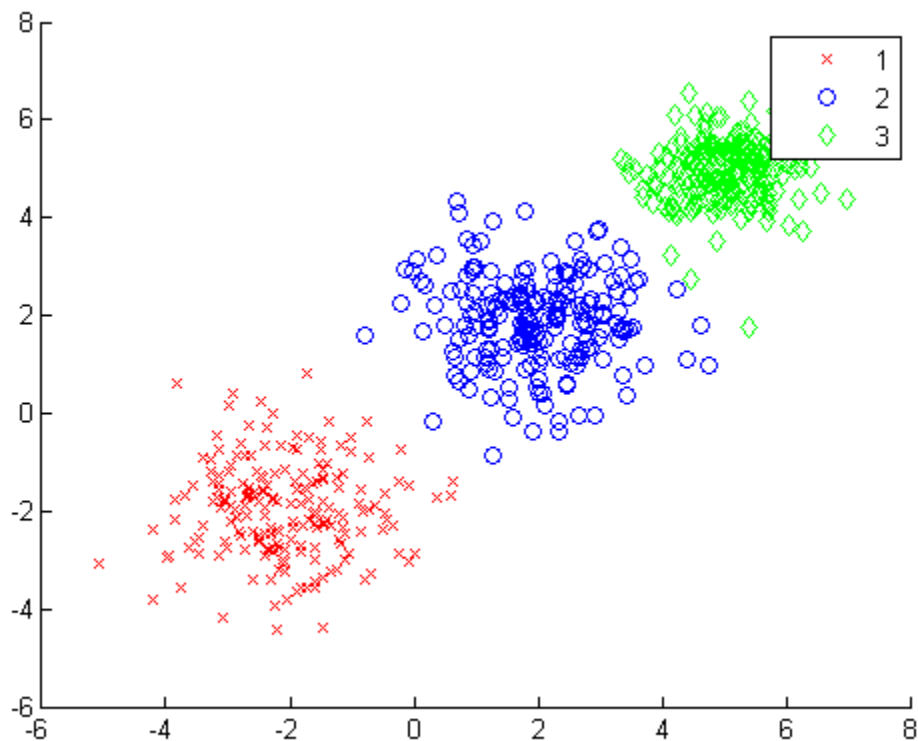
The plot shows that the highest silhouette value occurs at three clusters, suggesting that the optimal number of clusters is three.

Create a grouped scatter plot to visually examine the suggested clusters.

```
figure;  
gscatter(X(:,1),X(:,2),E.OptimalY,'rbg','xod')
```

# clustering.evaluation.SilhouetteEvaluation

---



The plot shows three distinct clusters within the data: Cluster 1 is in the lower-left corner, cluster 2 is near the center of the plot, and cluster 3 is in the upper-right corner.

## References

[1] Kaufman L. and P. J. Rouseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons, Inc., 1990.



[2] Rouseeuw, P. J. “Silhouettes: a graphical aid to the interpretation and validation of cluster analysis.” *Journal of Computational and Applied Mathematics*. Vol. 20, No. 1, 1987, pp. 53–65.

## See Also

[evalclusters](#) |  
[silhouetteclustering.evaluation.CalinskiHarabaszEvaluation](#) |  
[clustering.evaluation.DaviesBouldinEvaluation](#) |  
[clustering.evaluation.GapEvaluation](#) |

## Concepts

- Class Attributes
- Property Attributes

# categorical.single

---

**Purpose** Convert categorical array to single array

**Syntax** `B = single(A)`

**Description** `B = single(A)` converts the categorical array `A` to a single array. Each element of `B` contains the internal categorical level code for the corresponding element of `A`.

**See Also** `double`

**Purpose** Convert dataset variables to single array

**Syntax** `B = single(A)`  
`B = single(A,vars)`

**Description** `B = single(A)` returns the contents of the dataset A, converted to one single array. The classes of the variables in the dataset must support the conversion.

`B = single(A,vars)` returns the contents of the dataset variables specified by vars. vars is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector.

**See Also** `dataset` | `double` | `replacedata`

# categorical.size

---

**Purpose** Size of categorical array

**Syntax**

```
d = size(A)
[m,n] = size(A)
[m1,m2,m3,...,mn] = size(A)
m = size(A,dim)
```

**Description** `d = size(A)` returns the two-element row vector `d = [m,n]` containing the number of rows and columns in the matrix for an `m`-by-`n` categorical matrix `A`. For `n`-D categorical arrays, `size(A)` returns a 1-by-`n` vector of dimension lengths. Trailing singleton dimensions are ignored.

`[m,n] = size(A)` for a categorical matrix `A`, returns the number of rows and columns in `A` as separate output variables.

`[m1,m2,m3,...,mn] = size(A)`, for `n > 1`, returns the sizes of the first `n` dimensions of the categorical array `A`. If the number of output arguments `n` does not equal `ndims(A)`, then for:

`n > ndims(A)`                    `size` returns ones in the "extra" variables, i.e., outputs `ndims(A)+1` through `n`.

`n < ndims(A)`                    `mn` contains the product of the sizes of dimensions `n` through `ndims(A)`.

`m = size(A,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(A,1)` returns the number of rows. If `dim > ndims(A)`, `m` will be 1.

**See Also** `length` | `ndims` | `numel`

**Purpose**

Size of dataset array

**Syntax**

```
D = SIZE(A)
[NOBS,NVARS] = SIZE(A)
[M1,M2,M3,...,MN] = SIZE(A)
M = size(A,dim)
```

**Description**

`D = SIZE(A)` returns the two-element row vector `D = [NOBS,NVARS]` containing the number of observations and number of variables in the dataset `A`. A dataset array always has two dimensions.

`[NOBS,NVARS] = SIZE(A)` returns the numbers of observations and variables in the dataset `A` as separate output variables.

`[M1,M2,M3,...,MN] = SIZE(A)`, for  $N > 2$ , returns `M1 = NOBS`, `M2 = NVARS`, and `M3,...,MN = 1`.

`M = size(A,dim)` returns the length of the dimension specified by the scalar `dim`:

- `M = size(A,1)` returns `NOBS`
- `M = size(A,2)` returns `NVARS`
- `M = size(A,k)` returns 1 for  $k > 2$

**See Also**

`length` | `ndims` | `numel`

# grandset.size

---

**Purpose** Number of dimensions in matrix

**Syntax**  
`d = size(p)`  
`[m,n] = size(p)`  
`m = size(p,dim)`

**Description** `d = size(p)` returns the two-element row vector `d = [m,n]` containing the number of points in the point set and the number of dimensions the points are in, for the point set `p`. These correspond to the number of rows and columns in the matrix that would be produced by the expression `p(:, :)`.

`[m,n] = size(p)` returns the number of points and dimensions for `p` as separate output variables.

`m = size(p,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(p,1)` returns the number of rows (points in the point set). If `dim` is greater than 2, `m` will be 1.

**Examples** The commands

```
P = sobolset(12);  
d = size(P)
```

return

```
d = [9.0072e+015 12]
```

The command

```
[m,n] = size(P)
```

returns

```
m = 9.0072e+015  
n = 12
```

The command

```
m2 = size(P, 2)
```

returns

```
m2 = 12
```

## See Also

[length](#) | [ndims](#) | [grandset](#)

# slicesample

---

**Purpose** Slice sampler

**Syntax**

```
rnd = slicesample(initial,nsamples,'pdf',pdf)
rnd = slicesample(initial,nsamples,'logpdf',logpdf)
[rnd,neval] = slicesample(initial,...)
[rnd,neval] = slicesample(initial,...,Name,Value)
```

**Description**

`rnd = slicesample(initial,nsamples,'pdf',pdf)` generates `nsamples` random samples using the slice sampling method (see “Algorithms” on page 21-3180). `pdf` gives the target probability density function (`pdf`). `initial` is a row vector or scalar containing the initial value of the random sample sequences.

`rnd = slicesample(initial,nsamples,'logpdf',logpdf)` generates samples using the logarithm of the `pdf`.

`[rnd,neval] = slicesample(initial,...)` returns the average number of function evaluations that occurred in the slice sampling.

`[rnd,neval] = slicesample(initial,...,Name,Value)` generates random samples with additional options specified by one or more `Name,Value` pair arguments.

**Tips**

- There are no definitive suggestions for choosing appropriate values for `burnin`, `thin`, or `width`. Choose starting values of `burnin` and `thin`, and increase them, if necessary, to give the requisite independence and marginal distributions. See Neal [1] for details of the effect of adjusting `width`.

**Input Arguments**

**initial**  
Initial point, a scalar or row vector. Set `initial` so `pdf(initial)` is a strictly positive scalar. `length(initial)` is the number of dimensions of each sample.

**nsamples**  
Positive integer, the number of samples that `slicesample` generates.



**pdf**

Handle to a function that generates the probability density function, specified with @. pdf can be unnormalized, meaning it need not integrate to 1.

**logpdf**

Handle to a function that generates the logarithm of the probability density function, specified with @. logpdf can be the logarithm of an unnormalized pdf.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**'burnin'**

Nonnegative integer, the number of samples to generate and discard before generating the samples to return. The slice sampling algorithm is a Markov chain whose stationary distribution is proportional to that of the pdf argument. Set `burnin` to a high enough value that you believe the Markov chain approximately reaches stationarity after `burnin` samples.

**Default:** 0

**'thin'**

Positive integer, where `slicesample` discards every `thin - 1` samples and returns the next. The slice sampling algorithm is a Markov chain, so the samples are serially correlated. To reduce the serial correlation, choose a larger value of `thin`.

**Default:** 1

# slicesample

---

## 'width'

Width of the interval around the current sample, a scalar or vector of positive values. `slicesample` begins with this interval and searches for an appropriate region containing the points of pdf that evaluate to a large enough value.

- If `width` is a scalar and the samples have multiple dimensions, `slicesample` uses `width` for each dimension.
- If `width` is a vector, it should have the same length as `initial`.

**Default:** 10

## Output Arguments

### `rnd`

`nsamples-by-length(initial)` matrix, where each row is one sample.

### `neval`

Scalar, the mean number of function evaluations per sample. `neval` includes the `burnin` and `thin` evaluations, not just the evaluations of samples returned in `rnd`. Therefore the total number of function evaluations is

```
neval*(nsamples*thin + burnin).
```

## Examples

Generate random samples from a multimodal density using `slicesample`.

**1** Define a function proportional to a multimodal density:

```
f = @(x) exp(-x.^2/2).*(1 + (sin(3*x)).^2).*...  
    (1 + (cos(5*x)).^2));  
area = integral(f,-5,5);
```

**2** Generate 2000 samples from the density, using a burn-in period of 1000, and keeping one in five samples:

```
N = 2000;
```

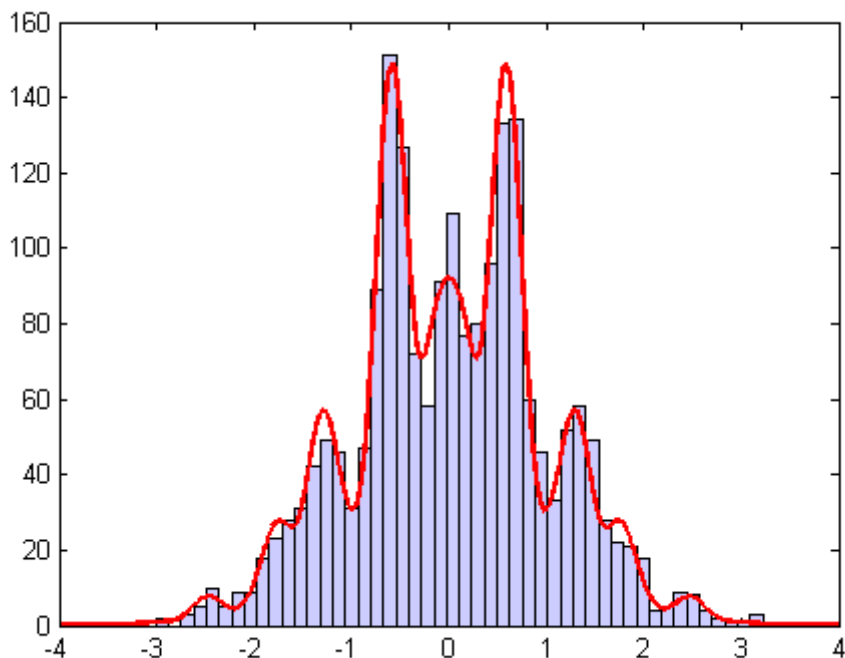
```
x = slicesample(1,N,'pdf',f,'thin',5,'burnin',1000);
```

- 3** Plot a histogram of the sample:

```
[binheight,bincenter] = hist(x,50);  
h = bar(bincenter,binheight,'hist');  
set(h,'facecolor',[0.8 .8 1]);
```

- 4** Scale the density to have the same area as the histogram, and superimpose it on the histogram:

```
hold on  
xd = get(gca,'XLim');  
xgrid = linspace(xd(1),xd(2),1000);  
binwidth = (bincenter(2)-bincenter(1));  
y = (N*binwidth/area) * f(xgrid);  
plot(xgrid,y,'r','LineWidth',2)  
hold off
```



The samples seem to fit the theoretical distribution well, so the burnin value seems adequate.

## Algorithms

At each point in the sequence of random samples, `slicesample` selects the next point by “slicing” the density to form a neighborhood around the previous point where the density is above some value. Consequently, the sample points are not independent. Nearby points in the sequence tend to be closer together than they would be from a sample of independent values. For many purposes, the entire set of points can be used as a sample from the target distribution. However, when this type of serial correlation is a problem, the `burnin` and `thin` parameters can help reduce that correlation.

`slicesample` uses the slice sampling algorithm of Neal [1]. For numerical stability, it converts a `pdf` function into a `logpdf` function. The algorithm to resize the support region for each level, called “stepping-out” and “stepping-in,” was suggested by Neal.

## References

[1] Neal, Radford M. *Slice Sampling*. Ann. Stat. Vol. 31, No. 3, pp. 705–767, 2003. Available at Project Euclid.

## See Also

`mhsample` | `rand` | `randsample`

# skewness

---

**Purpose** Skewness

**Syntax**  
`y = skewness(X)`  
`y = skewness(X, flag)`  
`y = skewness(X, flag, dim)`

**Description** `y = skewness(X)` returns the sample skewness of `X`. For vectors, `skewness(x)` is the skewness of the elements of `x`. For matrices, `skewness(X)` is a row vector containing the sample skewness of each column. For N-dimensional arrays, `skewness` operates along the first nonsingleton dimension of `X`.

`y = skewness(X, flag)` specifies whether to correct for bias (`flag = 0`) or not (`flag = 1`, the default). When `X` represents a sample from a population, the skewness of `X` is biased; that is, it will tend to differ from the population skewness by a systematic amount that depends on the size of the sample. You can set `flag = 0` to correct for this systematic bias.

`y = skewness(X, flag, dim)` takes the skewness along dimension `dim` of `X`.

`skewness` treats NaNs as missing values and removes them.

**Algorithms** Skewness is a measure of the asymmetry of the data around the sample mean. If skewness is negative, the data are spread out more to the left of the mean than to the right. If skewness is positive, the data are spread out more to the right. The skewness of the normal distribution (or any perfectly symmetric distribution) is zero.

The skewness of a distribution is defined as

$$s = \frac{E(x - \mu)^3}{\sigma^3}$$

where  $\mu$  is the mean of  $x$ ,  $\sigma$  is the standard deviation of  $x$ , and  $E(t)$  represents the expected value of the quantity  $t$ . `skewness` computes a sample version of this population value.

When you set `flag` to 1, the following equation applies:

$$s_1 = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^3}{\left( \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2} \right)^3}$$

When you set `flag` to 0, the following equation applies:

$$s_0 = \frac{\sqrt{n(n-1)}}{n-2} s_1$$

This bias-corrected formula requires that `X` contain at least three elements.

## Examples

```
X = randn([5 4])
X =
    1.1650    1.6961   -1.4462   -0.3600
    0.6268    0.0591   -0.7012   -0.1356
    0.0751    1.7971    1.2460   -1.3493
    0.3516    0.2641   -0.6390   -1.2704
   -0.6965    0.8717    0.5774    0.9846

y = skewness(X)
y =
   -0.2933    0.0482    0.2735    0.4641
```

## See Also

[kurtosis](#) | [mean](#) | [moment](#) | [std](#) | [var](#)

# grandset.Skip property

---

**Purpose** Number of initial points to omit from sequence

**Description** The Skip property of a point set contains a positive integer which specifies the number of initial points in the sequence to omit from the point set. The default Skip value is 0.

Initial points of a sequence sometimes exhibit undesirable properties, for example the first point is often  $(0,0,0,\dots)$  and this may "unbalance" the sequence since its counterpart,  $(1,1,1,\dots)$ , never appears. Another common reason is that initial points often exhibit correlations among different dimensions which disappear later in the sequence.

**Examples** Examine the difference between skipping and not skipping points:

```
% No skipping produces the standard Sobol sequence.  
P = sobolset(5);  
P(1:3,:)
```

```
% Skip the first point of the sequence. The point set now  
% starts at the second point of the basic Sobol sequence.  
P.Skip = 1;  
P(1:3,:)
```

**See Also** Leap | net | grandset | subsref



<b>Superclasses</b>	grandset
<b>Purpose</b>	Sobol quasi-random point sets
<b>Description</b>	sobolset is a quasi-random point set class that produces points from the Sobol sequence. The Sobol sequence is a base-2 digital sequence that fills space in a highly uniform manner.

<b>Construction</b>	sobolset	Construct Sobol quasi-random point set
---------------------	----------	--

**Methods****Inherited Methods**

Methods in the following table are inherited from grandset.

disp	Display grandset object
end	Last index in indexing expression for point set
length	Length of point set
ndims	Number of dimensions in matrix
net	Generate quasi-random point set
scramble	Scramble quasi-random point set
size	Number of dimensions in matrix
subsref	Subscripted reference for grandset

**Properties**

PointOrder	Point generation method
------------	-------------------------

**Inherited Properties**

Properties in the following table are inherited from grandset.

Dimensions	Number of dimensions
Leap	Interval between points
ScrambleMethod	Settings that control scrambling
Skip	Number of initial points to omit from sequence
Type	Name of sequence on which point set P is based

## Copy Semantics

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## References

[1] Bratley, P., and B. L. Fox, "ALGORITHM 659 Implementing Sobol's Quasirandom Sequence Generator," *ACM Transactions on Mathematical Software*, Vol. 14, No. 1, pp. 88-100, 1988.

[2] Joe, S., and F. Y. Kuo, "Remark on Algorithm 659: Implementing Sobol's Quasirandom Sequence Generator," *ACM Transactions on Mathematical Software*, Vol. 29, No. 1, pp. 49-57, 2003.

[3] Hong, H. S., and F. J. Hickernell, "ALGORITHM 823: Implementing Scrambled Digital Sequences," *ACM Transactions on Mathematical Software*, Vol. 29, No. 2, pp. 95-109, 2003.

[4] Matousek, J., "On the L2-discrepancy for anchored boxes," *Journal of Complexity*, Vol. 14, pp. 527-556, 1998.

## See Also

haltonset

## How To

- "Quasi-Random Point Sets" on page 6-16

**Purpose**

Construct Sobol quasi-random point set

**Syntax**

```
p = sobolset(d)
p = sobolset(d,prop1,va11,prop2,va12,...)
```

**Description**

`p = sobolset(d)` constructs a  $d$ -dimensional point set `p` of the `sobolset` class, with default property settings.

`p = sobolset(d,prop1,va11,prop2,va12,...)` specifies property name/value pairs used to construct `p`.

The object `p` returned by `sobolset` encapsulates properties of a specified quasi-random sequence. The point set is finite, with a length determined by the `Skip` and `Leap` properties and by limits on the size of point set indices (maximum value of  $2^{53}$ ). Values of the point set are not generated and stored in memory until you access `p` using `net` or parenthesis indexing.

**Examples**

Generate a 3-D Sobol point set, skip the first 1000 values, and then retain every 101st point:

```
p = sobolset(3,'Skip',1e3,'Leap',1e2)
p =
    Sobol point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
        Leap : 100
    ScrambleMethod : none
        PointOrder : standard
```

Use `scramble` to apply a random linear scramble combined with a random digital shift:

```
p = scramble(p,'MatousekAffineOwen')
p =
    Sobol point set in 3 dimensions (8.918019e+013 points)
    Properties:
        Skip : 1000
```

```
Leap : 100
ScrambleMethod : MatousekAffineOwen
PointOrder : standard
```

Use net to generate the first four points:

```
X0 = net(p,4)
X0 =
    0.7601    0.5919    0.9529
    0.1795    0.0856    0.0491
    0.5488    0.0785    0.8483
    0.3882    0.8771    0.8755
```

Use parenthesis indexing to generate every third point, up to the 11th point:

```
X = p(1:3:11,:)
X =
    0.7601    0.5919    0.9529
    0.3882    0.8771    0.8755
    0.6905    0.4951    0.8464
    0.1955    0.5679    0.3192
```

## References

- [1] Bratley, P., and B. L. Fox. “Algorithm 659 Implementing Sobol’s Quasirandom Sequence Generator.” *ACM Transactions on Mathematical Software*. Vol. 14, No. 1, 1988, pp. 88–100.
- [2] Joe, S., and F. Y. Kuo. “Remark on Algorithm 659: Implementing Sobol’s Quasirandom Sequence Generator.” *ACM Transactions on Mathematical Software*. Vol. 29, No. 1, 2003, pp. 49–57.
- [3] Hong, H. S., and F. J. Hickernell. “Algorithm 823: Implementing Scrambled Digital Sequences.” *ACM Transactions on Mathematical Software*. Vol. 29, No. 2, 2003, pp. 95–109.
- [4] Matousek, J. “On the L2-Discrepancy for Anchored Boxes.” *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

**See Also**      haltonset | net | scramble

# ordinal.sort

---

**Purpose** Sort elements of ordinal array

**Syntax**

```
B = sort(A)
B = sort(A,dim)
B = sort(A,dim,mode)
[B,I] = sort(A,...)
```

**Description** `B = sort(A)`, when `A` is an ordinal vector, sorts the elements of `A` in ascending order. For ordinal matrices, `sort(A)` sorts each column of `A` in ascending order. For  $N$ -D ordinal arrays, `sort(A)` sorts along the first nonsingleton dimension of `A`. `B` is an ordinal array with the same levels as `A`.

`B = sort(A,dim)` sorts `A` along dimension `dim`.

`B = sort(A,dim,mode)` sorts `A` in the order specified by `mode`. `mode` is 'ascend' for ascending order, or 'descend' for descending order.

`[B,I] = sort(A,...)` also returns an index matrix `I`. If `A` is a vector, then `B = A(I)`. If `A` is an  $m$ -by- $n$  matrix and `dim` is 1, then `B(:,j) = A(I(:,j),j)` for `j = 1:n`.

Elements with undefined levels are sorted to the end.

## Examples

Sort the columns of an ordinal array in ascending order:

```
A = ordinal([6 2 5; 2 4 1; 3 2 4],...
            {'lo','med','hi'},[],[0 2 4 6])
```

```
A =
      hi      med      hi
      med     hi      lo
      med     med     hi
```

```
B = sort(A)
B =
      med     med     lo
      med     med     hi
      hi      hi      hi
```

**See Also**

sortrows

# dataset.sortrows

---

**Purpose** Sort rows of dataset array

**Syntax**

```
B = sortrows(A)
B = sortrows(A,vars)
B = sortrows(A,'obsnames')
B = sortrows(A,vars,mode)
[B,idx] = sortrows(A)
```

**Description** `B = sortrows(A)` returns a copy of the dataset array `A`, with the observations sorted in ascending order by all of the variables in `A`. The observations in `B` are sorted first by the first variable, next by the second variable, and so on. Each variable in `A` must be a valid input to `sort`, or, if a variable contains multiple columns, to the MATLAB `sortrows` function or the `sortrows` function for ordinal data.

`B = sortrows(A,vars)` sorts the observations in `A` by the variables specified by `vars`. `vars` is a positive integer, a vector of positive integers, variable names, a cell array containing one or more variable names, or a logical vector.

`B = sortrows(A,'obsnames')` sorts the observations in `A` by observation name.

`B = sortrows(A,vars,mode)` sorts in the direction specified by `mode`. When `mode` is the single string `'ascend'` (the default) or `'descend'`, `sortrows` sorts `A` by the variables specified by `vars` in ascending or descending order, respectively. `mode` can also be a cell array containing the strings `'ascend'` or `'descend'`, to specify a different sorting direction for each variable in `vars`. Specify `[]` for `vars` to sort using all variables.

`[B,idx] = sortrows(A)` also returns an index vector `idx` such that `B = A(idx,:)`.

**Examples** Sort the data in `hospital.mat` by age and then by last name:

```
load hospital
hospital(1:5,1:3)
ans =
```



	LastName	Sex	Age
YPL-320	'SMITH'	Male	38
GLI-532	'JOHNSON'	Male	43
PNI-258	'WILLIAMS'	Female	38
MIJ-579	'JONES'	Female	40
XLK-030	'BROWN'	Female	49

```
hospital = sortrows(hospital,{'Age','LastName'});
hospital(1:5,1:3)
ans =
```

	LastName	Sex	Age
REV-997	'ALEXANDER'	Male	25
FZR-250	'HALL'	Male	25
LIM-480	'HILL'	Female	25
XUE-826	'JACKSON'	Male	25
SCQ-914	'JAMES'	Male	25

Sort the data in hospital by gender in ascending order, and age in descending order.

```
hospital = sortrows(hospital,{'Sex','Age'},{'ascend','descend'});
hospital(1:5,1:3)
ans =
```

	LastName	Sex	Age
XLK-030	'BROWN'	Female	49
GGU-691	'HUGHES'	Female	49
KKL-155	'ADAMS'	Female	48
HQO-561	'BRYANT'	Female	48
BKD-785	'CLARK'	Female	48

```
hospital(end-4:end,1:3)
ans =
```

	LastName	Sex	Age
VNL-702	'MOORE'	Male	28
REV-997	'ALEXANDER'	Male	25

# dataset.sortrows

---

FZR-250	'HALL'	Male	25
XUE-826	'JACKSON'	Male	25
SCQ-914	'JAMES'	Male	25

## See Also

[dataset](#) | [sortrowsunique](#) |

## Related Examples

- “Sort Observations in Dataset Arrays” on page 2-97

## Concepts

- “Dataset Arrays” on page 2-135

**Purpose**

Sort rows

**Syntax**

```

B = sortrows(A)
B = sortrows(A,col)
[B,I] = sortrows(A)
[B,I] = sortrows(A,col)

```

**Description**

`B = sortrows(A)` sorts the rows of the 2-D ordinal matrix `A` in ascending order, as a group. `B` is an ordinal array with the same levels as `A`.

`B = sortrows(A,col)` sorts `A` based on the columns specified in the vector `col`. If an element of `col` is positive, the corresponding column in `A` is sorted in ascending order; if an element of `col` is negative, the corresponding column in `A` is sorted in descending order.

`[B,I] = sortrows(A)` and `[B,I] = sortrows(A,col)` also returns an index matrix `I` such that `B = A(I,:)`.

Elements with undefined levels are sorted to the end.

**Examples**

Sort the rows of an ordinal array in ascending order for the first column, and then in descending order for the second column:

```

A = ordinal([6 2 5; 2 4 1; 3 2 4],...
            {'lo','med','hi'},[],[0 2 4 6])

```

```

A =
    hi      med      hi
    med     hi      lo
    med     med     hi

```

```

B = sortrows(A,[1 -2])

```

```

B =
    med     hi      lo
    med     med     hi
    hi      med     hi

```

**See Also**

`sort` | `sortrows`

# squareform

---

**Purpose** Format distance matrix

**Syntax**

```
Z = squareform(y)
y = squareform(Z)
Z = squareform(y, 'tovector')
Y = squareform(Z, 'tomatrix')
```

**Description** `Z = squareform(y)`, where `y` is a vector as created by the `pdist` function, converts `y` into a square, symmetric format `Z`, in which `Z(i, j)` denotes the distance between the `i`th and `j`th objects in the original data.

`y = squareform(Z)`, where `Z` is a square, symmetric matrix with zeros along the diagonal, creates a vector `y` containing the `Z` elements below the diagonal. `y` has the same format as the output from the `pdist` function.

`Z = squareform(y, 'tovector')` forces `squareform` to treat `y` as a vector.

`Y = squareform(Z, 'tomatrix')` forces `squareform` to treat `Z` as a matrix.

The last two formats are useful if the input has a single element, so that it is ambiguous whether the input is a vector or square matrix.

## Examples

```
y = 1:6
y =
    1    2    3    4    5    6

X = [0 1 2 3; 1 0 4 5; 2 4 0 6; 3 5 6 0]
X =
    0    1    2    3
    1    0    4    5
    2    4    0    6
    3    5    6    0
```

Then `squareform(y) = X` and `squareform(X) = y`.

**See Also**      `pdist`

# categorical.squeeze

---

**Purpose** Squeeze singleton dimensions from categorical array

**Syntax** `B = squeeze(A)`

**Description** `B = squeeze(A)` returns an array `B` with the same elements as the categorical array `A` but with all the singleton dimensions removed. A singleton is a dimension such that `size(A,dim)=1`. 2-D arrays are unaffected by `squeeze` so that row vectors remain rows.

**See Also** `shiftdim`

**Purpose**

Stack data from multiple variables into single variable

**Syntax**

```
tall = stack(wide,datavars)
[tall,iwide] = stack(wide,datavars)
tall = stack(wide,datavars,Parameter,value)
```

**Description**

`tall = stack(wide,datavars)` converts a wide-format dataset array into a tall-format array, by stacking multiple variables in `wide` into a single variable in `tall`. In general, `tall` contains fewer variables but more observations than `wide`.

`datavars` specifies a group of `m` data variables in `wide`. `stack` creates a single data variable in `tall` by interleaving their values, and if `wide` has `n` observations, then `tall` has `m-by-n` observations. In other words, `stack` takes the `m` data values from each observation in `wide` and stacks them up to create `m` observations in `tall`. `datavars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. `stack` also creates a grouping variable in `tall` to indicate which of the `m` data variables in `wide` each observation in `tall` corresponds to.

`stack` assigns values for the "per-variable properties (e.g., `Units` and `VarDescription`) for the new data variable in `tall` from the corresponding property values for the first variable listed in `datavars`.

`stack` copies the remaining variables from `wide` to `tall` without stacking, by replicating each of their values `m` times. These variables are typically grouping variables. Because their values are constant across each group of `m` observations in `tall`, they identify which observation in `wide` an observation in `tall` came from.

`[tall,iwide] = stack(wide,datavars)` returns an index vector `iwide` indicating the correspondence between observations in `tall` and those in `wide`. `stack` creates `tall(j,:)` using `wide(iwide(j),datavars)`.

For more information on grouping variables, see "Grouping Variables" on page 2-51.

# dataset.stack

---

## Input Arguments

`tall = stack(wide,datavars,Parameter,value)` uses the following parameter name/value pairs to control how `stack` converts variables in `wide` to variables in `tall`:

'ConstVars '	Variables in <code>wide</code> to copy to <code>tall</code> without stacking. <code>ConstVars</code> is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. The default is all variables in <code>wide</code> not specified in <code>datavars</code> .
'NewDataVarName '	A name for the data variable to be created in <code>tall</code> . The default is a concatenation of the names of the <code>m</code> variables that are stacked up.
'IndVarName '	A name for the grouping variable to create in <code>tall</code> to indicate the source of each value in the new data variable. The default is based on the 'NewDataVarName' parameter.

You can also specify multiple groups of data variables in `wide`, each of which becomes a variable in `tall`. All groups must contain the same number of variables. Use a cell array to contain multiple parameter values for `datavars`, and a cell array of strings to contain multiple 'NewDataVarName'.

## Examples

Convert a wide format data set to tall format, and then back to a different wide format:

```
load flu
```

```
% FLU has a 'Date' variable, and 10 variables for estimated  
% influenza rates (in 9 different regions, estimated from
```



```
% Google searches, plus a nationwide estimate from the
% CDC). Combine those 10 variables into a "tall" array that
% has a single data variable, 'FluRate', and an indicator
% variable, 'Region', that says which region each estimate
% is from.
[flu2,iflu] = stack(flu, 2:11, 'NewDataVarName','FluRate', ...
    'IndVarName','Region')

% The second observation in FLU is for 10/16/2005. Find the
% observations in FLU2 that correspond to that date.
flu(2,:)
flu2(iflu==2,:)

% Use the 'Date' variable from that tall array to split
% 'FluRate' into 52 separate variables, each containing the
% estimated influenza rates for each unique date. The new
% "wide" array has one observation for each region. In
% effect, this is the original array FLU "on its side".
dateNames = cellstr(datestr(flu.Date,'mmm_DD_YYYY'));
[flu3,iflu2] = unstack(flu2, 'FluRate', 'Date', ...
    'NewDataVarNames',dateNames)

% Since observations in FLU3 represent regions, IFLU2
% indicates the first occurrence in FLU2 of each region.
flu2(iflu2,:)
```

**See Also**

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

**How To**

- “Grouping Variables” on page 2-51

# grandstream.State property

---

**Purpose** Current state of the stream

**Description** The State property of a quasi-random stream contains the index into the associated point set of the next point to draw in the stream. Getting and resetting the State property allows you to return a stream to a previous state. The initial value of State is 1.

**Examples**

```
Q = grandstream('sobol', 5);  
s = Q.State;  
u1 = grand(Q, 10)  
Q.State = s;  
u2 = grand(Q, 10) % contains exactly the same values as u1
```

**See Also** grand

**Purpose**

Access values in statistics options structure

**Syntax**

```
val = statget(options,param)
val = statget(options,param,default)
```

**Description**

`val = statget(options,param)` returns the value of the parameter specified by the string `param` in the statistics options structure `options`. If the parameter is undefined in `options`, `statget` returns `[]`. You need to type only enough leading characters to define the parameter name uniquely. `statget` ignores case for parameter names. For available options, see Inputs.

`val = statget(options,param,default)` returns `default` if the specified parameter is undefined in the optimization options structure `options`.

**Input Arguments****DerivStep**

Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics Toolbox function using the options structure.

**Display**

Amount of information displayed by the algorithm.

- 'off' — Displays no information.
- 'final' — Displays the final output.
- 'iter' — Displays iterative output to the command window for some functions; otherwise displays the final output.

**FunValCheck**

Check for invalid values, such as NaN or Inf, from the objective function.

- 'off'
- 'on'

## **GradObj**

Flags whether the objective function returns a gradient vector as a second output.

- 'off'
- 'on'

## **Jacobian**

Flags whether the objective function returns a Jacobian as a second output.

- 'off'
- 'on'

## **MaxFunEvals**

Maximum number of objective function evaluations allowed. Positive integer.

## **MaxIter**

Maximum number of iterations allowed. Positive integer.

## **OutputFcn**

The solver calls all output functions after each iteration.

- Function handle specified using @
- a cell array with function handles
- an empty array (default)

## **Robust**

Invoke robust fitting option.

- 'off'
- 'on'

### **RobustWgtFun**

A weight function for robust fitting. Valid only when Robust is 'on'. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output.

- 'bisquare'
- 'andrews'
- 'cauchy'
- 'fair'
- 'huber'
- 'logistic'
- 'talwar'
- 'welsch'

### **Streams**

A single instance of the RandStream class, or a cell array of RandStream instances. The Streams option is accepted by some functions to govern what stream(s) to use in generating random numbers within the function. If 'UseSubstreams' is true, the Streams value must be a scalar, or must be empty. If 'UseParallel' is true and 'UseSubstreams' is false, then the Streams argument must either be empty, or its length must match the number of processors used in the computation: equal to the *parpool* size if a *parpool* is open, a scalar otherwise.

### **TolBnd**

Parameter bound tolerance. Positive scalar.

### **TolFun**

Termination tolerance for the objective function value. Positive scalar.

### **TolTypeFun**

Use `TolFun` for absolute or relative objective function tolerances.

- 'abs'
- 'rel'

## **TolTypeX**

Use `TolX` for absolute or relative parameter tolerances.

- 'abs'
- 'rel'

## **TolX**

Termination tolerance for the parameters. Positive scalar.

## **Tune**

The tuning constant used in robust fitting to normalize the residuals before applying the weight function. The default value depends upon the weight function. This parameter is necessary if you specify the weight function as a function handle. Positive scalar.

## **UseParallel**

Flag indicating whether eligible functions should use capabilities of the Parallel Computing Toolbox (PCT), if the capabilities are available. That is, if the PCT is installed, and a PCT parpool is in effect. Valid values are `false` (the default), for serial computation, and `true`, for parallel computation.

## **UseSubstreams**

Flag indicating whether the random number generator in eligible functions should use `Substream` property of the `RandStream` class. `false` (default) or `true`. When `true`, high level iterations within the function will set the `Substream` property to the value of the iteration. This behavior helps to generate reproducible random number streams in parallel and/or serial mode computation.

### WgtFun

A weight function for robust fitting. Valid only when Robust is 'on'. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output.

- 'bisquare'
- 'andrews'
- 'cauchy'
- 'fair'
- 'huber'
- 'logistic'
- 'talwar'
- 'welsch'

### Examples

This statement returns the value of the Display statistics options parameter from the structure called my\_options.

```
val = statget(my_options, 'Display')
```

---

Return the value of the Display statistics options parameter from the structure called my\_options (as in the previous example). If the Display parameter is undefined, statget returns the value 'final'.

```
optnew = statget(my_options, 'Display', 'final');
```

### See Also

statset

# statset

---

**Purpose** Create statistics options structure

**Syntax**

```
statset
statset(statfun)
options = statset(...)
options = statset(fieldname1,val1,fieldname2,val2,...)
options = statset(olddopts,fieldname1,val1,fieldname2,val2,
    ...)
options = statset(olddopts,newopts)
```

**Description** `statset` with no input arguments and no output arguments displays all fields of a statistics options structure and their possible values.

`statset(statfun)` displays fields and default values used by the Statistics Toolbox function `statfun`. Specify `statfun` using a string name or a function handle.

`options = statset(...)` creates a statistics options structure `options`. With no input arguments, all fields of the options structure are an empty array (`[]`). With a specified `statfun`, function-specific fields are default values and the remaining fields are `[]`. Function-specific fields set to `[]` indicate that the function is to use its default value for that parameter. For available options, see [Inputs](#).

`options = statset(fieldname1,val1,fieldname2,val2,...)` creates an options structure in which the named fields have the specified values. Any unspecified values are `[]`. Use strings for field names. For fields that are string-valued, you must input the complete string for the value. If you provide an invalid string for a value, `statset` uses the default.

`options = statset(olddopts,fieldname1,val1,fieldname2,val2,...)` creates a copy of `olddopts` with the named parameters changed to the specified values.

`options = statset(olddopts,newopts)` combines an existing options structure, `olddopts`, with a new options structure, `newopts`. Any parameters in `newopts` with nonempty values overwrite corresponding parameters in `olddopts`.



## Input Arguments

### DerivStep

Relative difference used in finite difference derivative calculations. A positive scalar, or a vector of positive scalars the same size as the vector of parameters estimated by the Statistics Toolbox function using the options structure.

### Display

Amount of information displayed by the algorithm.

- 'off' — Displays no information.
- 'final' — Displays the final output.
- 'iter' — Displays iterative output to the command window for some functions; otherwise displays the final output.

### FunValCheck

Check for invalid values, such as NaN or Inf, from the objective function.

- 'off'
- 'on'

### GradObj

Flags whether the objective function returns a gradient vector as a second output.

- 'off'
- 'on'

### Jacobian

Flags whether the objective function returns a Jacobian as a second output.

- 'off'
- 'on'

## **MaxFunEvals**

Maximum number of objective function evaluations allowed. Positive integer.

## **MaxIter**

Maximum number of iterations allowed. Positive integer.

## **OutputFcn**

The solver calls all output functions after each iteration.

- Function handle specified using @
- a cell array with function handles
- an empty array (default)

## **Robust**

Invoke robust fitting option.

- 'off'
- 'on'

Robust will be removed in a future software release. Use `RobustWgtFun` for robust fitting.

## **RobustWgtFun**

Weight function for robust fitting. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. If you use a function handle, give a Tune constant. See “Robust Options” on page 21-3212.

## **Streams**

A single instance of the `RandStream` class, or a cell array of `RandStream` instances. The `Streams` option is accepted by some functions to govern what stream(s) to use in generating random numbers within the function. If `'UseSubstreams'` is true, the `Streams` value must

be a scalar, or must be empty. If 'UseParallel' is true and 'UseSubstreams' is false, then the Streams argument must either be empty, or its length must match the number of processors used in the computation: equal to the *parpool* size if a *parpool* is open, a scalar otherwise.

**TolBnd**

Parameter bound tolerance. Positive scalar.

**TolFun**

Termination tolerance for the objective function value. Positive scalar.

**TolTypeFun**

Use TolFun for absolute or relative objective function tolerances.

- 'abs'
- 'rel'

**TolTypeX**

Use TolX for absolute or relative parameter tolerances.

- 'abs'
- 'rel'

**TolX**

Termination tolerance for the parameters. Positive scalar.

**Tune**

Tuning constant used in robust fitting to normalize the residuals before applying the weight function. The default value depends upon the weight function. This parameter is necessary if you specify the weight function as a function handle. Positive scalar. See “Robust Options” on page 21-3212.

## UseParallel

Flag indicating whether eligible functions should use capabilities of the Parallel Computing Toolbox (PCT), if the capabilities are available. That is, if the PCT is installed, and a PCT parpool is in effect. Valid values are `false` (the default), for serial computation, and `true`, for parallel computation.

## UseSubstreams

Flag indicating whether the random number generator in eligible functions should use `Substream` property of the `RandStream` class. `false` (default) or `true`. When `true`, high level iterations within the function will set the `Substream` property to the value of the iteration. This behavior helps to generate reproducible random number streams in parallel and/or serial mode computation.

## WgtFun

Weight function for robust fitting. Valid only when `Robust` is `'on'`. Can also be a function handle that accepts a normalized residual as input and returns the robust weights as output. See “Robust Options” on page 21-3212.

`WgtFun` will be removed in a future software release. Use `RobustWgtFun` instead.

## Definitions

### Robust Options

Weight Function	Equation	Default Tuning Constant
'andrews'	$w = (\text{abs}(r) < \pi) .* \sin(r) ./ r$	1.339
'bisquare' (default)	$w = (\text{abs}(r) < 1) .* (1 - r.^2).^2$	4.685
'cauchy'	$w = 1 ./ (1 + r.^2)$	2.385

Weight Function	Equation	Default Tuning Constant
'fair'	$w = 1 ./ (1 + \text{abs}(r))$	1.400
'huber'	$w = 1 ./ \max(1, \text{abs}(r))$	1.345
'logistic'	$w = \tanh(r) ./ r$	1.205
'talwar'	$w = 1 * (\text{abs}(r) < 1)$	2.795
'welsch'	$w = \exp(-(r.^2))$	2.985
[]	No robust fitting	—

## Examples

Suppose you want to change the default parameter values for the function `evfit`, which fits an extreme value distribution to data. The defaults parameter values are:

```
statset('evfit')
ans =
    Display: 'off'
    MaxFunEvals: []
    MaxIter: []
    TolBnd: []
    TolFun: []
    TolTypeFun: []
    TolX: 1.0000e-06
    TolTypeX: []
    GradObj: []
    Jacobian: []
    DerivStep: []
    FunValCheck: []
    Robust: []
    RobustWgtFun: []
    WgtFun: []
    Tune: []
    UseParallel: []
    UseSubstreams: []
```

## statset

---

```
Streams: []  
OutputFcn: []
```

---

The only parameters that `evfit` uses are `Display` and `TolX`. To create an options structure with the value of `TolX` set to `1e-8`, enter:

```
options = statset('TolX',1e-8)  
% Pass options to evfit:  
mu = 1;  
sigma = 1;  
data = evrnd(mu,sigma,1,100);  
  
paramhat = evfit(data,[],[],[],options)
```

### See Also

`statget`

<b>Purpose</b>	Standard deviation of probability distribution object
<b>Syntax</b>	<code>s = std(pd)</code>
<b>Description</b>	<code>s = std(pd)</code> returns the standard deviation <code>s</code> of the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<p><b>pd - Probability distribution</b> probability distribution object</p> <p>Probability distribution, specified as a probability distribution object. Fit a probability distribution object to data using <code>fitdist</code> or the Distribution Fitting app.</p>
<b>Output Arguments</b>	<p><b>s - Standard deviation</b> nonnegative scalar value</p> <p>Standard deviation of the probability distribution, returned as a nonnegative scalar value.</p>
<b>Examples</b>	<p><b>Standard Deviation of a Fitted Distribution</b></p> <p>Load the sample data. Create a vector containing the first column of students' exam grade data.</p> <pre>load examgrades; x = grades(:,1);</pre> <p>Create a probability distribution object by fitting a kernel distribution to the data.</p> <pre>pd = fitdist(x,'Kernel')</pre> <pre>pd =  KernelDistribution  Kernel = normal</pre>

# prob.KernelDistribution.std

---

```
Bandwidth = 3.61677  
Support = unbounded
```

Compute the standard deviation of the fitted distribution.

```
s = std(pd)
```

```
s =
```

```
9.4069
```

## See Also

```
fitdist | dfittool
```



<b>Purpose</b>	Return standard deviation of ProbDistUnivParam object	
<b>Syntax</b>	$S = \text{std}(PD)$	
<b>Description</b>	$S = \text{std}(PD)$ returns $S$ , the standard deviation of the ProbDistUnivParam object $PD$ .	
<b>Input Arguments</b>	$PD$	An object of the class ProbDistUnivParam.
<b>Output Arguments</b>	$S$	The standard deviation of the ProbDistUnivParam object $PD$ .
<b>See Also</b>	std	

# prob.ParametricTruncatableDistribution.std

---

**Purpose** Standard deviation of probability distribution object

**Syntax** `s = std(pd)`

**Description** `s = std(pd)` returns the standard deviation `s` of the probability distribution `pd`.

**Input Arguments** **pd - Probability distribution**  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`.

**Output Arguments** **s - Standard deviation**  
nonnegative scalar value

Standard deviation of the probability distribution, returned as a nonnegative scalar value.

**Examples** **Standard Deviation of a Triangular Distribution**

Create a triangular distribution object.

```
pd = makedist('Triangular','a',-3,'b',1,'c',3)
```

```
pd =
```

```
TriangularDistribution
```

```
A = -3, B = 1, C = 3
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

1.2472

**See Also** `makedist`

# prob.ToolboxFittableParametricDistribution.std

---

<b>Purpose</b>	Standard deviation of probability distribution object
<b>Syntax</b>	<code>s = std(pd)</code>
<b>Description</b>	<code>s = std(pd)</code> returns the standard deviation <code>s</code> of the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code> . Alternatively, create a probability distribution object by fitting it to data using <code>fitdist</code> or the Distribution Fitting app.
<b>Output Arguments</b>	<b>s - Standard deviation</b> nonnegative scalar value  Standard deviation of the probability distribution, returned as a nonnegative scalar value.
<b>Examples</b>	<b>Standard Deviation of a Fitted Distribution</b>  Load the sample data. Create a vector containing the first column of students' exam grade data.  <code>load examgrades;</code> <code>x = grades(:,1);</code>  Fit a normal distribution object to the data.  <code>pd = fitdist(x,'Normal')</code>  <code>pd =</code>  <code>NormalDistribution</code>

```
Normal distribution
mu = 75.0083 [73.4321, 76.5846]
sigma = 8.7202 [7.7391, 9.98843]
```

Compute the standard deviation of the fitted distribution.

```
s = std(pd)
```

```
s =
```

```
8.7202
```

For a normal distribution, the standard deviation is equal to the parameter `sigma`.

## Standard Deviation of a Skewed Distribution

Create a Weibull probability distribution object

```
pd = makedist('Weibull','a',5,'b',2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the standard deviation of the distribution.

```
s = std(pd)
```

```
s =
```

```
2.3163
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

# GeneralizedLinearModel.step

---

**Purpose** Improve generalized linear regression model by adding or removing terms

**Syntax**  
`mdl1 = step(mdl)`  
`mdl1 = step(mdl, Name, Value)`

**Description** `mdl1 = step(mdl)` returns an improved generalized linear model based on `mdl`, with one predictor added or removed.  
`mdl1 = step(mdl, Name, Value)` improves a generalized linear model with additional options specified by one or more `Name, Value` pair arguments.

**Tips**

- Use `addTerms` or `removeTerms` to control exactly which terms enter or leave the model.

**Input Arguments** **mdl - Generalized linear model**  
`GeneralizedLinearModel` object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a `GeneralizedLinearModel` object.

For properties and methods of the generalized linear model object, `mdl`, see the `GeneralizedLinearModel` class page.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## 'Criterion' - Criterion to add or remove terms

'Deviance' | 'sse' | 'aic' | 'bic' | 'rsquared' | 'adjrsquared'

Criterion to add or remove terms, specified as the comma-separated pair consisting of 'Criterion' and one of the following:

- 'Deviance' — Default for `stepwiseglm`.  $p$ -value for  $F$  or chi-squared test of the change in the deviance by adding or removing the term.  $F$ -test is for testing a single model. Chi-squared test is for comparing two different models. This option is not valid for `stepwiselm`.
- 'sse' — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- 'aic' — Change in the value of Akaike information criterion (AIC).
- 'bic' — Change in the value of Bayesian information criterion (BIC).
- 'rsquared' — Increase in the value of  $R^2$ .
- 'adjrsquared' — Increase in the value of adjusted  $R^2$ .

**Example:** 'Criterion','bic'

## 'Lower' - Model specification describing smallest set of terms in fit

'constant' (default)

Model specification describing the smallest set of terms in the fit, specified as the comma-separated pair consisting of 'Lower' and one of the string options for `modelspec` naming the model.

**Example:** 'Lower','linear'

## 'NSteps' - Number of steps to take

1 (default) | positive integer

Number of steps to take, specified as the comma-separated pair consisting of 'NSteps' and a positive integer.

# GeneralizedLinearModel.step

---

## Data Types

single | double

## 'PEnter' - Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than PEnter, add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than PEnter, add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than PEnter, add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than PEnter, add the term to the model.



Criterion	Default value	Decision
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than PEnter, add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add the term to the model.

For more information on the criteria, see Criterion name-value pair argument.

**Example:** 'PEnter',0.075

**'PRemove' - Improvement measure for removing term**  
scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of 'PRemove' and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than PRemove, remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than PRemove,

# GeneralizedLinearModel.step

---

Criterion	Default value	Decision
		remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

**Example:** `'PRemove',0.05`

**'Upper' - Model specification describing largest set of terms in fit**  
`'interaction' (default) | string`

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of `'Upper'` and one of the string options for `modelspec` naming the model.

**Example:** `'Upper','quadratic'`

**'Verbose' - Control for display of information**

`1 (default) | 0 | 2`

Control for display of information, specified as the comma-separated pair consisting of `'Verbose'` and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

**Example:** `'Verbose',2`

## Output Arguments

### **mdl1**

Linear model, the same as `mdl` but with additional terms given in terms. You can set `mdl1` equal to `mdl` to overwrite `mdl`.

## Examples

### **Add Predictors One at a Time**

Fit a Poisson regression model using random data and a single predictor, then step in other predictors.

Generate artificial data with 20 predictors, using three of the predictors for the responses.

# GeneralizedLinearModel.step

---

```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:,[5 10 15])*[.4;.2;.3] + 1);
y = poissrnd(mu);
```

Construct a generalized linear model using  $X(:, 1)$  as the only predictor.

```
mdl = fitglm(X,y,...
    'y ~ x1','Distribution','poisson')
```

```
mdl =
```

Generalized Linear regression model:

```
log(y) ~ 1 + x1
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.1278	0.057487	19.618	1.0904e-85
x1	0.061287	0.04848	1.2642	0.20617

100 observations, 98 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 1.59, p-value = 0.208

Add a variable to the model using `step`.

```
mdl1 = step(mdl)
```

```
1. Adding x5, Deviance = 134.2976, Chi2Stat = 50.80176, PValue = 1.021821e-12
```

```
mdl1 =
```

Generalized Linear regression model:

```
log(y) ~ 1 + x1 + x5
Distribution = Poisson
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0418	0.062341	16.712	1.07e-62
x1	0.018803	0.049916	0.37671	0.70639
x5	0.47881	0.067875	7.0542	1.7357e-12

100 observations, 97 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 52.4, p-value = 4.21e-12

Add another variable to the model using step.

```
mdl1 = step(mdl1)
```

2. Adding x15, Deviance = 105.9973, Chi2Stat = 28.30027, PValue = 1.038814e-07

```
mdl1 =
```

Generalized Linear regression model:

$\log(y) \sim 1 + x1 + x5 + x15$

Distribution = Poisson

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0459	0.0627	16.681	1.7975e-62
x1	0.026907	0.05003	0.53782	0.5907
x5	0.3983	0.068376	5.8251	5.7073e-09
x15	0.28949	0.053992	5.3618	8.2375e-08

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi<sup>2</sup>-statistic vs. constant model: 80.7, p-value = 2.18e-17

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

# GeneralizedLinearModel.step

---

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1** Fit the initial model.
- 2** If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3** If any terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps

will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

Use `stepwiseglm` to select a model from a starting model, continuing until no single step is beneficial.

Use `addTerms` or `removeTerms` to add or remove particular terms.

## See Also

`addTerms` | `GeneralizedLinearModel` | `removeTerms` | `stepwiseglm`

## Related Examples

- “Plots to Understand Predictor Effects and How to Modify a Model” on page 10-32

## Concepts

- “Generalized Linear Models” on page 10-12

# LinearModel.step

---

**Purpose** Improve linear regression model by adding or removing terms

**Syntax**  
`mdl1 = step(mdl)`  
`mdl1 = step(mdl,Name,Value)`

**Description** `mdl1 = step(mdl)` returns an improved linear model based on `mdl`, with one predictor added or removed.

---

**Note** You can use `step` only if `mdl.Robust = []`. This holds when you create `mdl` with `fitlm` having the `RobustOpts` name-value pair set to the default 'off'.

---

`mdl1 = step(mdl,Name,Value)` improves a linear model with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **mdl**

Linear model, as constructed by `fitlm` or `stepwiselm`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'Criterion'**

Criterion that `step` uses in selecting terms to add or remove:

<b>Criterion</b>	<b>PEnter</b>	<b>PRemove</b>	<b>Compared Against</b>
'SSE'	0.05	< 0.1	$p$ -value for $F$ test
'AIC'	0	< 0.01	Change in AIC



Criterion	PEnter	PRemove	Compared Against
'BIC'	0	< 0.01	Change in BIC
'Rsquared'	0.1	> 0.05	Increase in R-squared
'AdjRsquared'	0	> -0.05	Increase in adjusted R-squared

Default: 'sse'

### 'Lower' - Model specification describing smallest set of terms in fit

'constant' (default)

Model specification describing the smallest set of terms in the fit, specified as the comma-separated pair consisting of 'Lower' and one of the string options for `modelspec` naming the model.

Example: 'Lower', 'linear'

### 'NSteps' - Number of steps to take

1 (default) | positive integer

Number of steps to take, specified as the comma-separated pair consisting of 'NSteps' and a positive integer.

### Data Types

single | double

### 'PEnter' - Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

## LinearModel.step

---

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than PEnter, add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than PEnter, add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than PEnter, add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than PEnter, add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than PEnter, add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add

Criterion	Default value	Decision
		the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.

**Example:** `'PEnter',0.075`

**'PRemove' - Improvement measure for removing term**  
 scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of `'PRemove'` and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.

# LinearModel.step

---

Criterion	Default value	Decision
'BIC'	0.01	If the change in the BIC of the model is larger than PRemove, remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than PRemove, remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than PRemove, remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see Criterion name-value pair argument.

**Example:** 'PRemove',0.05

**'Upper' - Model specification describing largest set of terms in fit**  
'interaction' (default) | string

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of 'Upper' and one of the string options for `modelspec` naming the model.

**Example:** 'Upper', 'quadratic'

## 'Verbose' - Control for display of information

1 (default) | 0 | 2

Control for display of information, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

**Example:** 'Verbose', 2

## Output Arguments

### **mdl1**

Linear model. Typically you set `mdl1` equal to `mdl`.

## Examples

### **Modify a Linear Model**

Fit a linear model to car data. Use `step` to see if a quadratic model would help the fit quality.

Load `carsmall` data, and make a dataset from weight and model year predictors with MPG response.

```
load carsmall
ds = dataset(MPG,Weight);
ds.Year = ordinal(Model_Year);
```

Make a linear model of MPG as a function of Year and Weight.

```
mdl = fitlm(ds,'MPG ~ Year + Weight')
```

# LinearModel.step

---

```
mdl =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Year
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	40.11	1.5418	26.016	1.2024e-43
Weight	-0.0066475	0.00042802	-15.531	3.3639e-27
Year_76	1.9291	0.74761	2.5804	0.011488
Year_82	7.9093	0.84975	9.3078	7.8681e-15

```
Number of observations: 94, Error degrees of freedom: 90
```

```
Root Mean Squared Error: 2.92
```

```
R-squared: 0.873, Adjusted R-Squared 0.868
```

```
F-statistic vs. constant model: 206, p-value = 3.83e-40
```

Use step to adjust the model to potentially include full quadratic terms.

```
mdl1 = step(mdl, 'upper', 'quadratic')
```

```
1. Adding Weight^2, FStat = 9.9164, pValue = 0.0022303
```

```
mdl1 =
```

```
Linear regression model:
```

```
MPG ~ 1 + Weight + Year + Weight^2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	54.206	4.7117	11.505	2.6648e-19
Weight	-0.016404	0.0031249	-5.2493	1.0283e-06
Year_76	2.0887	0.71491	2.9215	0.0044137
Year_82	8.1864	0.81531	10.041	2.6364e-16
Weight^2	1.5573e-06	4.9454e-07	3.149	0.0022303

```
Number of observations: 94, Error degrees of freedom: 89
```

Root Mean Squared Error: 2.78  
R-squared: 0.885, Adjusted R-Squared 0.88  
F-statistic vs. constant model: 172, p-value = 5.52e-41

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the `'Criterion'` argument. The default value of `'Criterion'` is `'sse'`, and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when `'Criterion'` is `'sse'`:

- 1** Fit the initial model.
- 2** If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3** If any terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

The default for `stepwiseglm` is `'Deviance'` and it follows a similar procedure for adding or removing terms.

# LinearModel.step

---

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

Use `stepwiselm` to select a model from a starting model, continuing until no single step is beneficial.

Use `addTerms` or `removeTerms` to add or remove particular terms.

## See Also

`addTerms` | `LinearModel` | `removeTerms` | `stepwiselm`

## Tutorials

- “Linear Regression Workflow” on page 9-44
- “Change Models” on page 9-37

## How To

- “Linear Regression” on page 9-12

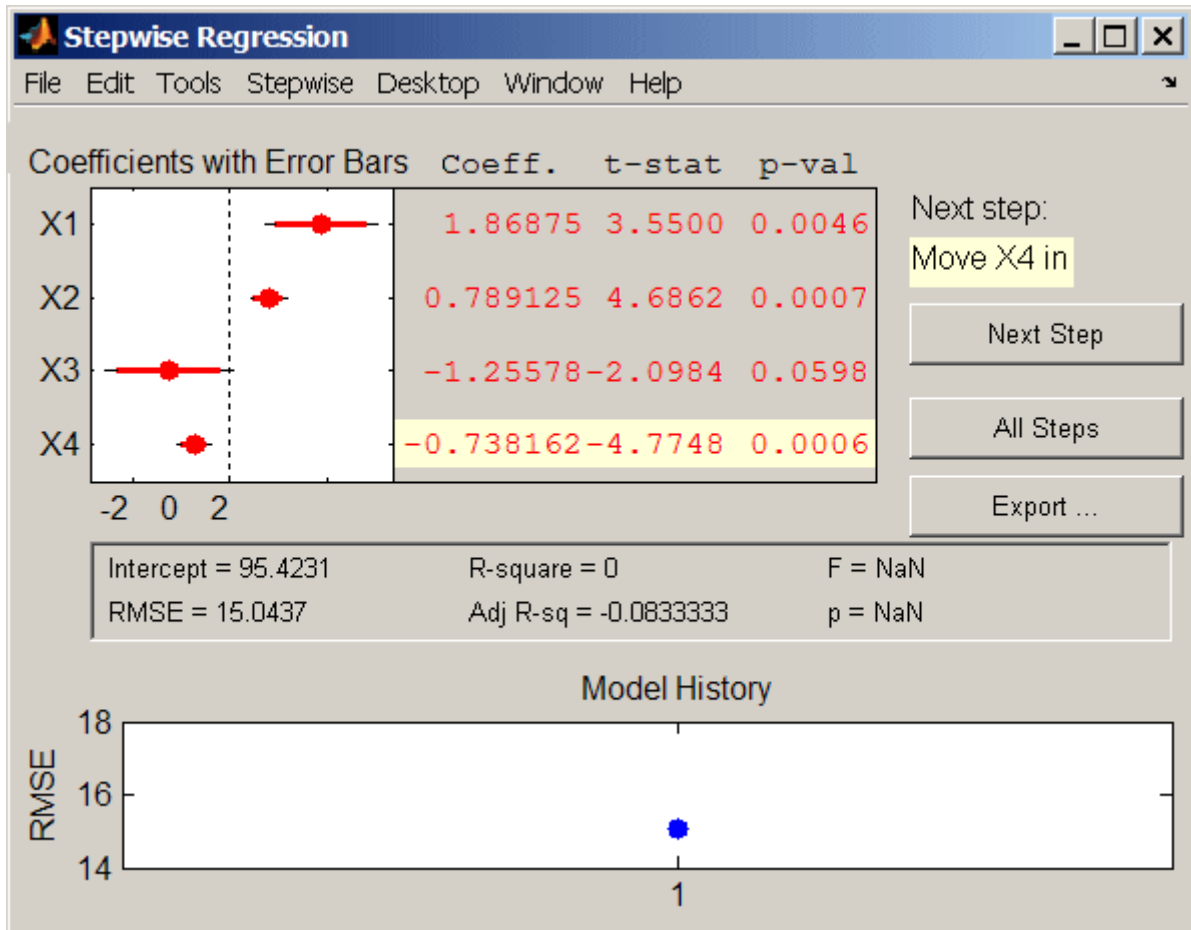


**Purpose** Interactive stepwise regression

**Syntax**  
`stepwise`  
`stepwise(X,y)`  
`stepwise(X,y,inmodel,penter,premove)`

**Description** `stepwise` uses the sample data in `hald.mat` to display a graphical user interface for performing stepwise regression of the response values in `heat` on the predictive terms in `ingredients`.

# stepwise



The upper left of the interface displays estimates of the coefficients for all potential terms, with horizontal bars indicating 90% (colored) and 95% (grey) confidence intervals. The red color indicates that, initially, the terms are not in the model. Values displayed in the table are those that would result if the terms were added to the model.

The middle portion of the interface displays summary statistics for the entire model. These statistics are updated with each step.

The lower portion of the interface, **Model History**, displays the RMSE for the model. The plot tracks the RMSE from step to step, so you can compare the optimality of different models. Hover over the blue dots in the history to see which terms were in the model at a particular step. Click on a blue dot in the history to open a copy of the interface initialized with the terms in the model at that step.

Initial models, as well as entrance/exit tolerances for the  $p$ -values of  $F$ -statistics, are specified using additional input arguments to **stepwise**. Defaults are an initial model with no terms, an entrance tolerance of 0.05, and an exit tolerance of 0.10.

To center and scale the input data (compute  $z$ -scores) to improve conditioning of the underlying least-squares problem, select **Scale Inputs** from the **Stepwise** menu.

You proceed through a stepwise regression in one of two ways:

- 1** Click **Next Step** to select the recommended next step. The recommended next step either adds the most significant term or removes the least significant term. When the regression reaches a local minimum of RMSE, the recommended next step is “Move no terms.” You can perform all of the recommended steps at once by clicking **All Steps**.
- 2** Click a line in the plot or in the table to toggle the state of the corresponding term. Clicking a red line, corresponding to a term not currently in the model, adds the term to the model and changes the line to blue. Clicking a blue line, corresponding to a term currently in the model, removes the term from the model and changes the line to red.

To call `addedvarplot` and produce an added variable plot from the **stepwise** interface, select **Added Variable Plot** from the **Stepwise** menu. A list of terms is displayed. Select the term you want to add, and then click **OK**.

Click **Export** to display a dialog box that allows you to select information from the interface to save to the MATLAB workspace.

Check the information you want to export and, optionally, change the names of the workspace variables to be created. Click **OK** to export the information.

`stepwise(X,y)` displays the interface using the  $p$  predictive terms in the  $n$ -by- $p$  matrix  $X$  and the response values in the  $n$ -by-1 vector  $y$ . Distinct predictive terms should appear in different columns of  $X$ .

---

**Note** `stepwise` automatically includes a constant term in all models. Do not enter a column of 1s directly into  $X$ .

---

`stepwise` treats NaN values in either  $X$  or  $y$  as missing values, and ignores them.

`stepwise(X,y,inmodel,penter,premove)` additionally specifies the initial model (`inmodel`) and the entrance (`penter`) and exit (`premove`) tolerances for the  $p$ -values of  $F$ -statistics. `inmodel` is either a logical vector with length equal to the number of columns of  $X$ , or a vector of indices, with values ranging from 1 to the number of columns in  $X$ . The value of `penter` must be less than or equal to the value of `premove`.

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a multilinear model based on their statistical significance in a regression. The method begins with an initial model and then compares the explanatory power of incrementally larger and smaller models. At each step, the  $p$  value of an  $F$ -statistic is computed to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model. The method proceeds as follows:

- 1 Fit the initial model.

- 2** If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$  value and repeat this step; otherwise, go to step 3.
- 3** If any terms in the model have  $p$ -values greater than an exit tolerance (that is, if it is unlikely that the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$  value and go to step 2; otherwise, end.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method may build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but may not be globally optimal.

**See Also**

`addedvarplot` | `regress` | `stepwisefit`

# GeneralizedLinearModel.stepwise

---

**Purpose** Create generalized linear regression model by stepwise regression

**Syntax**

```
mdl = GeneralizedLinearModel.stepwise(ds,modelspec)
mdl = GeneralizedLinearModel.stepwise(X,y,modelspec)
mdl = GeneralizedLinearModel.stepwise(...,modelspec,Name,
    Value)
```

**Description** `mdl = GeneralizedLinearModel.stepwise(ds,modelspec)` creates a generalized linear model of a dataset array `ds`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = GeneralizedLinearModel.stepwise(X,y,modelspec)` creates a generalized linear model of the responses `y` to a data matrix `X`, using stepwise regression to add or remove predictors.

`mdl = GeneralizedLinearModel.stepwise(...,modelspec,Name,Value)` creates a generalized linear model with additional options specified by one or more `Name,Value` pair arguments.

- Tips**
- Use the function `stepwiseglm` instead of `GeneralizedLinearModel.stepwise`.
  - The generalized linear model `mdl` is a standard linear model unless you specify otherwise with the `Distribution` name-value pair.
  - For other methods such as `devianceTest`, or properties of the `GeneralizedLinearModel` object, see `GeneralizedLinearModel`.

**Input Arguments**

**ds - Input data**  
dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

## Data Types

single | double | logical

## X - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

## Data Types

single | double | logical

## y - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

## Data Types

single | double

## modelspec - Starting model

string specifying the model |  $t$ -by- $(p+1)$  terms matrix | string of the form 'Y ~ terms'

Starting model for `stepwiseglm`, specified as one of the following:

# GeneralizedLinearModel.stepwise

- String specifying the type of model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p+1)$  matrix, namely terms matrix, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- String representing a formula in the form

`'Y ~ terms'`,

where the terms are in “Wilkinson Notation” on page 21-3265.



## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## 'BinomialSize' - Number of trials for binomial distribution

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the response. This is the parameter `n` for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is 'binomial'.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

## Data Types

single | double

## 'CategoricalVars' - Categorical variables

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of 'CategoricalVars' and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.

# GeneralizedLinearModel.stepwise

---

- If data is in matrix  $X$ , then the default value of this name-value pair argument is an empty matrix `[]`. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** `'CategoricalVars',[2,3]`

**Example:** `'CategoricalVars',logical([0 1 1 0 0 0])`

## Data Types

`single` | `double` | `logical`

## 'Criterion' - Criterion to add or remove terms

`'Deviance'` | `'sse'` | `'aic'` | `'bic'` | `'rsquared'` | `'adjrsquared'`

Criterion to add or remove terms, specified as the comma-separated pair consisting of `'Criterion'` and one of the following:

- `'Deviance'` — Default for `stepwiseglm`.  $p$ -value for  $F$  or chi-squared test of the change in the deviance by adding or removing the term.  $F$ -test is for testing a single model. Chi-squared test is for comparing two different models. This option is not valid for `stepwiselm`.
- `'sse'` — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- `'aic'` — Change in the value of Akaike information criterion (AIC).
- `'bic'` — Change in the value of Bayesian information criterion (BIC).
- `'rsquared'` — Increase in the value of  $R^2$ .
- `'adjrsquared'` — Increase in the value of adjusted  $R^2$ .

**Example:** `'Criterion','bic'`

## 'DispersionFlag' - Indicator to compute dispersion parameter

false for 'binomial' and 'poisson' distributions (default) | true

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

true	Estimate a dispersion parameter when computing standard errors
false	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

**Example:** 'DispersionFlag', true

## 'Distribution' - Distribution of the response variable

'normal' (default) | 'binomial' | 'poisson' | 'gamma' | 'inverse gaussian'

Distribution of the response variable, specified as the comma-separated pair consisting of 'Distribution' and one of the following.

'normal'	Normal distribution
'binomial'	Binomial distribution
'poisson'	Poisson distribution
'gamma'	Gamma distribution
'inverse gaussian'	Inverse Gaussian distribution

**Example:** 'Distribution', 'gamma'

## 'Exclude' - Observations to exclude

logical or numeric index vector

# GeneralizedLinearModel.stepwise

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

## Data Types

single | double | logical

## 'Intercept' - Indicator for constant term

true (default) | false

Indicator the for constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

**Example:** 'Intercept',false

## 'Link' - Link function

The canonical link function (default) | scalar value | structure

Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
'identity'	$f(\mu) = \mu$	$\mu = Xb$
'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'probit'	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$

Link Function Name	Link Function	Mean (Inverse) Function
'comploglog'	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
'reciprocal'	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
p (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$
S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size: <ul style="list-style-type: none"> <li>• S.Link — The link function</li> <li>• S.Inverse — The inverse link function</li> <li>• S.Derivative — The derivative of the link function</li> </ul>	$f(\mu) = S.Link(\mu)$	$\mu = S.Inverse(Xb)$

The link function defines the relationship  $f(\mu) = X*b$  between the mean response  $\mu$  and the linear combination of predictors  $X*b$ .

For more information on the canonical link functions, see [Definitions](#).

**Example:** 'Link', 'probit'

### 'Lower' - Model specification describing smallest set of terms in fit

'constant' (default)

Model specification describing the smallest set of terms in the fit, specified as the comma-separated pair consisting of 'Lower' and one of the string options for `modelspec` naming the model.

**Example:** 'Lower', 'linear'

# GeneralizedLinearModel.stepwise

---

## 'Offset' - Offset variable

[] (default) | vector | string

Offset variable in the fit, specified as the comma-separated pair consisting of 'Offset' and a vector or name of a variable with the same length as the response.

fitglm and stepwiseglm use Offset as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the Offset predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor A. By using the log link function and by specifying  $\log(A)$  as an offset, you can force the model to satisfy this theoretical constraint.

## Data Types

single | double | char

## 'PEnter' - Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

# GeneralizedLinearModel.stepwise

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than PEnter, add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than PEnter, add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than PEnter, add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than PEnter, add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than PEnter, add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add

# GeneralizedLinearModel.stepwise

Criterion	Default value	Decision
		the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.

**Example:** `'PEnter',0.075`

## 'PredictorVars' - Predictor variables

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of `'PredictorVars'` and either a cell array of strings of the variable names in the dataset array `ds`, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the `ds` dataset array, or the names you specify using the `'VarNames'` name-value pair argument.

The default is all variables in `X`, or all variables in `ds` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** `'PredictorVars',[2,3]`

**Example:** `'PredictorVars',logical([0 1 1 0 0 0])`

## Data Types

single | double | logical | cell

## 'PRemove' - Improvement measure for removing term

scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of `'PRemove'` and a scalar value.



# GeneralizedLinearModel.stepwise

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <code>PRemove</code> , remove the term from the model.

# GeneralizedLinearModel.stepwise

Criterion	Default value	Decision
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

**Example:** `'PRemove', 0.05`

## 'ResponseVar' - Response variable

last column in dataset array `ds` (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of `'ResponseVar'` and either a string of the variable name in the dataset array `ds`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a dataset array `ds`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

**Example:** `'ResponseVar','yield'`

**Example:** `'ResponseVar',[4]`

**Example:** `'ResponseVar',logical([0 0 0 1 0 0])`

## Data Types

`single` | `double` | `logical` | `char`

## 'Upper' - Model specification describing largest set of terms in fit

`'interaction'` (default) | `string`

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of `'Upper'` and one of the string options for `modelspec` naming the model.

**Example:** `'Upper','quadratic'`

## 'VarNames' - Names of variables in fit

`{'x1','x2',...,'xn','y'}` (default) | `cell array of strings`

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings including the names for the columns of `X` first, and the name for the response variable `y` last.

`'VarNames'` is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

## Example:

`'VarNames',{ 'Horsepower','Acceleration','Model_Year','MPG' }`

## Data Types

`cell`

## 'Weights' - Observation weights

# GeneralizedLinearModel.stepwise

---

ones(n,1) (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

## Data Types

single | double

## Output Arguments

### mdl - Generalized linear model

GeneralizedLinearModel object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a GeneralizedLinearModel object.

For properties and methods of the generalized linear model object, mdl, see the GeneralizedLinearModel class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

# GeneralizedLinearModel.stepwise

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

# GeneralizedLinearModel.stepwise

---

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
    0    0    0
    1    0    0
    0    1    0
    1    1    0
    0    2    0
```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

```
y ~ 1 + x1*x2 + x2^2
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337

# GeneralizedLinearModel.stepwise

x2	-0.012781	0.0060312	-2.1192	0.0368
x1:x2	-0.00010892	0.00017925	-0.6076	0.5
x2^2	9.7518e-07	7.5389e-07	1.2935	0.199

Number of observations: 94, Error degrees of freedom: 89  
Root Mean Squared Error: 4.1  
R-squared: 0.751, Adjusted R-Squared 0.739  
F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];  
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

mdl =

Linear regression model:

$$y \sim 1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92  
Root Mean Squared Error: 4.13  
R-squared: 0.738, Adjusted R-Squared 0.735  
F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of fitlm in the previous step.

# GeneralizedLinearModel.stepwise

---

## Formula

A formula for model specification is a string of the form

`'Y ~ terms'`,

where

- *Y* is the response name.
- *terms* contains
  - Variable names
  - + means include the next variable
  - - means do not include the next variable
  - : defines an interaction, a product of terms
  - \* defines an interaction **and all lower-order terms**
  - ^ raises the predictor to a power, exactly as in \* repeated, so ^ includes lower order terms as well
  - () groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include - 1 in the formula.

---

For example,

`'Y ~ A + B + C'` means a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept and a  $B^2$  term.

`'Y ~ A + B^2 + C'` is the same as the previous example because



$B^2$  includes a B term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among A, B, and C, except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where k is a positive integer	A, $A^2$ , ..., $A^k$
$A + B$	A, B
$A*B$	A, B, $A*B$
$A:B$	$A*B$ only
-B	Do not include B
$A*B + C$	A, B, C, $A*B$
$A + B + C + A:B$	A, B, C, $A*B$
$A*B*C - A:B:C$	A, B, C, $A*B$ , $A*C$ , $B*C$
$A*(B + C)$	A, B, C, $A*B$ , $A*C$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using -1.

## Canonical Link Function

The default link function for a generalized linear model is the *canonical link function*.

# GeneralizedLinearModel.stepwise

## Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

## Examples

### Create a Generalized Linear Model Stepwise

Create response data using just three of 20 predictors, and create a generalized linear model stepwise to see if it uses just the correct predictors.

Create data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:, [5 10 15]) * [.4;.2;.3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = stepwiseglm(X,y,...
    'constant', 'upper', 'linear', 'Distribution', 'poisson')
```

```
1. Adding x5, Deviance = 134.439, Chi2Stat = 52.24814, PValue = 4.891229e-13
2. Adding x15, Deviance = 106.285, Chi2Stat = 28.15393, PValue = 1.1204e-07
3. Adding x10, Deviance = 95.0207, Chi2Stat = 11.2644, PValue = 0.000790094
```

```
mdl =
```

Generalized Linear regression model:

$\log(y) \sim 1 + x5 + x10 + x15$

Distribution = Poisson

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0115	0.064275	15.737	8.4217e-56
x5	0.39508	0.066665	5.9263	3.0977e-09
x10	0.18863	0.05534	3.4085	0.0006532
x15	0.29295	0.053269	5.4995	3.8089e-08

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 91.7, p-value = 9.61e-20

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the `'Criterion'` argument. The default value of `'Criterion'` is `'sse'`, and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when `'Criterion'` is `'sse'`:

# GeneralizedLinearModel.stepwise

---

- 1 Fit the initial model.
- 2 If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Alternatives

You can also create a stepwise generalized linear model using `stepwiseglm`.

Use `fitglm` to create a model with a fixed specification. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

## References

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

## See Also

GeneralizedLinearModel | fitglm

## Related Examples

- “Compare large and small stepwise models” on page 9-119

## Concepts

- “Generalized Linear Models” on page 10-12

# LinearModel.stepwise

---

**Purpose** Create linear regression model by stepwise regression

**Syntax**

```
mdl = LinearModel.stepwise(ds,modelspec)
mdl = LinearModel.stepwise(X,y,modelspec)
mdl = LinearModel.stepwise( __ ,modelspec,Name,Value)
```

**Description** `mdl = LinearModel.stepwise(ds,modelspec)` returns a linear model of a dataset array `ds`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = LinearModel.stepwise(X,y,modelspec)` creates a linear model of the responses `y` to a data matrix `X`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = LinearModel.stepwise( __ ,modelspec,Name,Value)` creates a linear model for any of the inputs in the previous syntaxes, with additional options specified by one or more `NAME,Value` pair arguments.

For example, you can specify the categorical variables, the smallest or largest set of terms to use in the model, the maximum number of steps to take, or the criterion `LinearModel.stepwise` uses to add or remove terms.

- Tips**
- Use the function `stepwiselm` instead of `LinearModel.stepwise`.
  - You cannot use robust regression with stepwise regression. Check your data for outliers before using `LinearModel.stepwise`.
  - For other methods or properties of the `LinearModel` object, see `LinearModel`.

**Input Arguments**

**ds - Input data**  
dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and

response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### Data Types

single | double | logical

### X - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

### Data Types

single | double | logical

### y - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### Data Types

single | double

### modelspec - Starting model

string specifying the model |  $t$ -by- $(p+1)$  terms matrix | string of the form 'Y ~ terms'

# LinearModel.stepwise

---

Starting model for the stepwise regression, specified as one of the following:

- String specifying the type of starting model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

If you want to specify the smallest or largest set of terms in the model, use the **Lower** and **Upper** name-value pair arguments.

- $t$ -by- $(p+1)$  matrix, namely a terms matrix, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- String representing a formula in the form



```
'Y ~ terms',
```

where the terms are in “Wilkinson Notation” on page 21-3286.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

### 'CategoricalVars' - Categorical variables

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.
- If data is in matrix `X`, then the default value of this name-value pair argument is an empty matrix `[]`. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** `'CategoricalVars', [2,3]`

**Example:** `'CategoricalVars', logical([0 1 1 0 0 0])`

### Data Types

single | double | logical

### 'Criterion' - Criterion to add or remove terms

`'Deviance'` | `'sse'` | `'aic'` | `'bic'` | `'rsquared'` | `'adjrsquared'`

# LinearModel.stepwise

---

Criterion to add or remove terms, specified as the comma-separated pair consisting of 'Criterion' and one of the following:

- 'Deviance' — Default for `stepwiseglm`.  $p$ -value for  $F$  or chi-squared test of the change in the deviance by adding or removing the term.  $F$ -test is for testing a single model. Chi-squared test is for comparing two different models. This option is not valid for `stepwiselm`.
- 'sse' — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- 'aic' — Change in the value of Akaike information criterion (AIC).
- 'bic' — Change in the value of Bayesian information criterion (BIC).
- 'rsquared' — Increase in the value of  $R^2$ .
- 'adjrsquared' — Increase in the value of adjusted  $R^2$ .

**Example:** 'Criterion','bic'

## 'Exclude' - Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

## Data Types

single | double | logical

### 'Intercept' - Indicator for constant term

true (default) | false

Indicator the for constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

**Example:** 'Intercept',false

### 'Lower' - Model specification describing smallest set of terms in fit

'constant' (default)

Model specification describing the smallest set of terms in the fit, specified as the comma-separated pair consisting of 'Lower' and one of the string options for `modelspec` naming the model.

**Example:** 'Lower', 'linear'

### 'NSteps' - Number of steps to take

1 (default) | positive integer

Number of steps to take, specified as the comma-separated pair consisting of 'NSteps' and a positive integer.

## Data Types

single | double

### 'PEnter' - Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

## LinearModel.stepwise

---

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than $P_{Enter}$ , add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than $P_{Enter}$ , add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than $P_{Enter}$ , add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than $P_{Enter}$ , add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than $P_{Enter}$ , add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than $P_{Enter}$ , add

Criterion	Default value	Decision
		the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.

**Example:** `'PEnter',0.075`

### **'PredictorVars' - Predictor variables**

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of `'PredictorVars'` and either a cell array of strings of the variable names in the dataset array `ds`, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the `ds` dataset array, or the names you specify using the `'VarNames'` name-value pair argument.

The default is all variables in `X`, or all variables in `ds` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** `'PredictorVars',[2,3]`

**Example:** `'PredictorVars',logical([0 1 1 0 0 0])`

### **Data Types**

single | double | logical | cell

### **'PRemove' - Improvement measure for removing term**

scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of `'PRemove'` and a scalar value.

## LinearModel.stepwise

---

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <code>PRemove</code> , remove the term from the model.

Criterion	Default value	Decision
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <code>PRemove</code> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

**Example:** `'PRemove', 0.05`

### 'ResponseVar' - Response variable

last column in dataset array `ds` (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of `'ResponseVar'` and either a string of the variable name in the dataset array `ds`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a dataset array `ds`.

# LinearModel.stepwise

---

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

**Example:** `'ResponseVar','yield'`

**Example:** `'ResponseVar',[4]`

**Example:** `'ResponseVar',logical([0 0 0 1 0 0])`

## Data Types

`single` | `double` | `logical` | `char`

## 'Upper' - Model specification describing largest set of terms in fit

`'interaction'` (default) | `string`

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of `'Upper'` and one of the string options for `modelspec` naming the model.

**Example:** `'Upper','quadratic'`

## 'VarNames' - Names of variables in fit

`{'x1','x2',...,'xn','y'}` (default) | `cell array of strings`

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings including the names for the columns of `X` first, and the name for the response variable `y` last.

`'VarNames'` is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

## Example:

`'VarNames',{'Horsepower','Acceleration','Model_Year','MPG'}`

## Data Types

`cell`

## 'Verbose' - Control for display of information



1 (default) | 0 | 2

Control for display of information, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

**Example:** 'Verbose',2

### 'Weights' - Observation weights

ones(n,1) (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

### Data Types

single | double

## Output Arguments

### mdl - Linear model

LinearModel object

Linear model representing a least-squares fit of the response to the data, returned as a LinearModel object.

For the properties and methods of the linear model object, mdl, see the LinearModel class page.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

# LinearModel.stepwise

---

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```
    0    0    0
    1    0    0
    0    1    0
    1    1    0
    0    2    0
```

# LinearModel.stepwise

---

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

$$y \sim 1 + x1*x2 + x2^2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

```
mdl =
```

Linear regression model:

$$y \sim 1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

## Formula

A formula for model specification is a string of the form

`'Y ~ terms'`,

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

# LinearModel.stepwise

---

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include - 1 in the formula.

---

For example,

'Y ~ A + B + C' means a three-variable linear model with intercept.

'Y ~ A + B + C - 1' is a three-variable linear model without intercept.

'Y ~ A + B + C + B^2' is a three-variable model with intercept and a B^2 term.

'Y ~ A + B^2 + C' is the same as the previous example because B^2 includes a B term.

'Y ~ A + B + C + A:B' includes an A\*B term.

'Y ~ A\*B + C' is the same as the previous example because  $A*B = A + B + A:B$ .

'Y ~ A\*B\*C - A:B:C' has all interactions among A, B, and C, except the three-way interaction.

'Y ~ A\*(B + C + D)' has all linear terms, plus products of A with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
A^k, where k is a positive integer	A, A <sup>2</sup> , ..., A <sup>k</sup>
A + B	A, B
A*B	A, B, A*B
A:B	A*B only

Wilkinson Notation	Factors in Standard Notation
-B	Do not include B
A*B + C	A, B, C, A*B
A + B + C + A:B	A, B, C, A*B
A*B*C - A:B:C	A, B, C, A*B, A*C, B*C
A*(B + C)	A, B, C, A*B, A*C

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using -1.

## Examples

### Linear Model from Stepwise Regression

Fit a linear model of the Hald data using stepwise regression.

Load the data.

```
load hald
```

Fit a linear model to the data.

```
mdl = LinearModel.stepwise(ingredients,heat,'PEnter',0.06)
```

1. Adding x4, FStat = 22.7985, pValue = 0.000576232
2. Adding x1, FStat = 108.2239, pValue = 1.105281e-06
3. Adding x2, FStat = 5.0259, pValue = 0.051687
4. Removing x4, FStat = 1.8633, pValue = 0.2054

```
mdl =
```

Linear regression model:

$$y \sim 1 + x1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	52.577	2.2862	22.998	5.4566e-10

# LinearModel.stepwise

---

x1	1.4683	0.1213	12.105	2.6922e-07
x2	0.66225	0.045855	14.442	5.029e-08

Number of observations: 13, Error degrees of freedom: 10  
Root Mean Squared Error: 2.41  
R-squared: 0.979, Adjusted R-Squared 0.974  
F-statistic vs. constant model: 230, p-value = 4.41e-09

## Simultaneously Specify the Variables and Use Formula

Simultaneously identify response and predictor variables and specify the initial model using formula in stepwise regression.

Load sample data.

```
load hospital
```

Fit a linear model to the data.

```
mdl = LinearModel.stepwise(hospital, 'Weight~1+Smoker', ...  
  'ResponseVar', 'Weight', 'PredictorVars', {'Sex', 'Age', 'Smoker'}, ...  
  'CategoricalVar', {'Sex', 'Smoker'})
```

1. Adding Sex, FStat = 770.0158, pValue = 6.262758e-48
2. Removing Smoker, FStat = 0.21224, pValue = 0.64605

```
mdl =
```

```
Linear regression model:  
Weight ~ 1 + Sex
```

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	130.47	1.1995	108.77	5.2762e-104
Sex_Male	50.06	1.7496	28.612	2.2464e-49

Number of observations: 100, Error degrees of freedom: 98  
Root Mean Squared Error: 8.73



R-squared: 0.893, Adjusted R-Squared 0.892  
F-statistic vs. constant model: 819, p-value = 2.25e-49

The weight of the patients do not seem to differ significantly according to age or the status of smoking, or interaction of these factors with gender. `LinearModel.stepwise` only includes `Sex` in the final linear model.

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the `'Criterion'` argument. The default value of `'Criterion'` is `'sse'`, and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when `'Criterion'` is `'sse'`:

- 1 Fit the initial model.
- 2 If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3 If any terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

# LinearModel.stepwise

---

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## References

[1] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, pp. 307–312, 1998.

## Alternatives

You can also construct a stepwise linear model using `stepwiselm`.

You can construct a model using `fitlm`, then manually adjust the model using `step`, `addTerms`, or `removeTerms`. Use `fitlm` for robust regression. You cannot use robust regression and stepwise regression together.

## See Also

`LinearModel` | `fitlm` | `step`

## Tutorials

- “Compare large and small stepwise models” on page 9-119

## How To

- “Linear Regression” on page 9-12
- “Stepwise Regression” on page 9-119

**Purpose** Create generalized linear regression model by stepwise regression

**Syntax**

```
mdl = stepwiseglm(ds,modelspec)
mdl = stepwiseglm(X,y,modelspec)
mdl = stepwiseglm(...,modelspec,Name,Value)
```

**Description** `mdl = stepwiseglm(ds,modelspec)` creates a generalized linear model of a dataset array `ds`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = stepwiseglm(X,y,modelspec)` creates a generalized linear model of the responses `y` to a data matrix `X`, using stepwise regression to add or remove predictors.

`mdl = stepwiseglm(...,modelspec,Name,Value)` creates a generalized linear model with additional options specified by one or more `Name, Value` pair arguments.

## Input Arguments

### **ds - Input data**

dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### **Data Types**

single | double | logical

# stepwiseglm

---

## **X - Predictor variables**

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

### **Data Types**

single | double | logical

## **y - Response variable**

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### **Data Types**

single | double

## **modelspec - Starting model**

string specifying the model |  $t$ -by- $(p+1)$  terms matrix | string of the form 'Y ~ terms'

Starting model for `stepwiseglm`, specified as one of the following:

- String specifying the type of model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.

String	Model Type
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

- $t$ -by- $(p+1)$  matrix, namely terms matrix, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- String representing a formula in the form

`'Y ~ terms'`,

where the terms are in “Wilkinson Notation” on page 21-3265.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

#### Example:

`'Criterion', 'aic', 'Distribution', 'poisson', 'Upper', 'interactions'` specifies Akaike Information Criterion as the criterion to add or remove

variables to the model, Poisson distribution as the distribution of the response variable, and a model with all possible interactions as the largest model to consider as the fit.

## **'BinomialSize' - Number of trials for binomial distribution**

1 (default) | scalar value | vector

Number of trials for binomial distribution, that is the sample size, specified as the comma-separated pair consisting of a scalar value or a vector of the same length as the response. This is the parameter  $n$  for the fitted binomial distribution. `BinomialSize` applies only when the `Distribution` parameter is `'binomial'`.

If `BinomialSize` is a scalar value, that means all observations have the same number of trials.

As an alternative to `BinomialSize`, you can specify the response as a two-column vector with counts in column 1 and `BinomialSize` in column 2.

### **Data Types**

single | double

## **'CategoricalVars' - Categorical variables**

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.
- If data is in matrix `X`, then the default value of this name-value pair argument is an empty matrix `[]`. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** 'CategoricalVars',[2,3]

**Example:** 'CategoricalVars',logical([0 1 1 0 0 0])

### Data Types

single | double | logical

### 'Criterion' - Criterion to add or remove terms

'Deviance' | 'sse' | 'aic' | 'bic' | 'rsquared' | 'adjrsquared'

Criterion to add or remove terms, specified as the comma-separated pair consisting of 'Criterion' and one of the following:

- 'Deviance' — Default for `stepwiseglm`.  $p$ -value for  $F$  or chi-squared test of the change in the deviance by adding or removing the term.  $F$ -test is for testing a single model. Chi-squared test is for comparing two different models. This option is not valid for `stepwiselm`.
- 'sse' — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- 'aic' — Change in the value of Akaike information criterion (AIC).
- 'bic' — Change in the value of Bayesian information criterion (BIC).
- 'rsquared' — Increase in the value of  $R^2$ .
- 'adjrsquared' — Increase in the value of adjusted  $R^2$ .

**Example:** 'Criterion','bic'

### 'DispersionFlag' - Indicator to compute dispersion parameter

false for 'binomial' and 'poisson' distributions (default) | true

Indicator to compute dispersion parameter for 'binomial' and 'poisson' distributions, specified as the comma-separated pair consisting of 'DispersionFlag' and one of the following.

<code>true</code>	Estimate a dispersion parameter when computing standard errors
<code>false</code>	Default. Use the theoretical value when computing standard errors

The fitting function always estimates the dispersion for other distributions.

**Example:** `'DispersionFlag',true`

## **'Distribution' - Distribution of the response variable**

`'normal' (default) | 'binomial' | 'poisson' | 'gamma' | 'inverse gaussian'`

Distribution of the response variable, specified as the comma-separated pair consisting of `'Distribution'` and one of the following.

<code>'normal'</code>	Normal distribution
<code>'binomial'</code>	Binomial distribution
<code>'poisson'</code>	Poisson distribution
<code>'gamma'</code>	Gamma distribution
<code>'inverse gaussian'</code>	Inverse Gaussian distribution

**Example:** `'Distribution','gamma'`

## **'Exclude' - Observations to exclude**

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of `'Exclude'` and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** `'Exclude',[2,3]`



**Example:** 'Exclude',logical([0 1 1 0 0 0])

### Data Types

single | double | logical

### 'Intercept' - Indicator for constant term

true (default) | false

Indicator the for constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

**Example:** 'Intercept',false

### 'Link' - Link function

The canonical link function (default) | scalar value | structure

Link function to use in place of the canonical link function, specified as the comma-separated pair consisting of 'Link' and one of the following.

Link Function Name	Link Function	Mean (Inverse) Function
'identity'	$f(\mu) = \mu$	$\mu = Xb$
'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'probit'	$f(\mu) = \Phi^{-1}(\mu)$	$\mu = \Phi(Xb)$
'comloglog'	$f(\mu) = \log(-\log(1 - \mu))$	$\mu = 1 - \exp(-\exp(Xb))$
'reciprocal'	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$

# stepwiseglm

Link Function Name	Link Function	Mean (Inverse) Function
p (a number)	$f(\mu) = \mu^p$	$\mu = Xb^{1/p}$
S (a structure) with three fields. Each field holds a function handle that accepts a vector of inputs and returns a vector of the same size: <ul style="list-style-type: none"> <li>• S.Link — The link function</li> <li>• S.Inverse — The inverse link function</li> <li>• S.Derivative — The derivative of the link function</li> </ul>	$f(\mu) = \text{S.Link}(\mu)$	$\mu = \text{S.Inverse}(Xb)$

The link function defines the relationship  $f(\mu) = X^*b$  between the mean response  $\mu$  and the linear combination of predictors  $X^*b$ .

For more information on the canonical link functions, see [Definitions](#).

**Example:** 'Link', 'probit'

## 'Lower' - Model specification describing smallest set of terms in fit

'constant' (default)

Model specification describing the smallest set of terms in the fit, specified as the comma-separated pair consisting of 'Lower' and one of the string options for `modelspec` naming the model.

**Example:** 'Lower', 'linear'

## 'Offset' - Offset variable

[] (default) | vector | string

Offset variable in the fit, specified as the comma-separated pair consisting of 'Offset' and a vector or name of a variable with the same length as the response.

fitglm and stepwiseglm use Offset as an additional predictor, with a coefficient value fixed at 1.0. In other words, the formula for fitting is

$$\mu \sim \text{Offset} + (\text{terms involving real predictors})$$

with the Offset predictor having coefficient 1.

For example, consider a Poisson regression model. Suppose the number of counts is known for theoretical reasons to be proportional to a predictor A. By using the log link function and by specifying  $\log(A)$  as an offset, you can force the model to satisfy this theoretical constraint.

### Data Types

single | double | char

### 'PEnter' - Improvement measure for adding term

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than PEnter, add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than PEnter, add the term to the model.

# stepwiseglm

Criterion	Default value	Decision
'AIC'	0	If the change in the AIC of the model is smaller than PEnter, add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than PEnter, add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than PEnter, add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add the term to the model.

For more information on the criteria, see **Criterion** name-value pair argument.

**Example:** 'PEnter',0.075

## 'PredictorVars' - Predictor variables

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array **ds**, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the `ds` dataset array, or the names you specify using the `'VarNames'` name-value pair argument.

The default is all variables in `X`, or all variables in `ds` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** `'PredictorVars',[2,3]`

**Example:** `'PredictorVars',logical([0 1 1 0 0 0])`

### Data Types

single | double | logical | cell

### 'PRemove' - Improvement measure for removing term

scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of `'PRemove'` and a scalar value.

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <code>PRemove</code> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <code>PRemove</code> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <code>PRemove</code> , remove the term from the model.

# stepwiseglm

Criterion	Default value	Decision
'BIC'	0.01	If the change in the BIC of the model is larger than PRemove, remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than PRemove, remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than PRemove, remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see **Criterion** name-value pair argument.

**Example:** 'PRemove',0.05

## 'ResponseVar' - Response variable

last column in dataset array ds (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of 'ResponseVar' and either a string of the variable name in the dataset array ds, or a logical or numeric index vector

indicating which column is the response variable. You typically need to use 'ResponseVar' when fitting a dataset array ds.

For example, you can specify the fourth variable, say yield, as the response out of six variables, in one of the following ways.

**Example:** 'ResponseVar', 'yield'

**Example:** 'ResponseVar', [4]

**Example:** 'ResponseVar', logical([0 0 0 1 0 0])

### Data Types

single | double | logical | char

### 'Upper' - Model specification describing largest set of terms in fit

'interaction' (default) | string

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of 'Upper' and one of the string options for `modelspec` naming the model.

**Example:** 'Upper', 'quadratic'

### 'VarNames' - Names of variables in fit

{ 'x1', 'x2', ..., 'xn', 'y' } (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of 'VarNames' and a cell array of strings including the names for the columns of X first, and the name for the response variable y last.

'VarNames' is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

**Example:**

'VarNames', {'Horsepower', 'Acceleration', 'Model\_Year', 'MPG'}

### Data Types

cell

## 'Weights' - Observation weights

`ones(n,1)` (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of 'Weights' and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

## Data Types

single | double

## Output Arguments

### mdl - Generalized linear model

GeneralizedLinearModel object

Generalized linear model representing a least-squares fit of the link of the response to the data, returned as a GeneralizedLinearModel object.

For properties and methods of the generalized linear model object, mdl, see the GeneralizedLinearModel class page.

## Examples

### Generalized Linear Model Using Stepwise Algorithm

Create response data using just three of 20 predictors, and create a generalized linear model using stepwise algorithm to see if it uses just the correct predictors.

Create data with 20 predictors, and Poisson response using just three of the predictors, plus a constant.

```
rng('default') % for reproducibility
X = randn(100,20);
mu = exp(X(:,[5 10 15])*[.4;.2;.3] + 1);
y = poissrnd(mu);
```

Fit a generalized linear model using the Poisson distribution.

```
mdl = stepwiseglm(X,y,...
    'constant','upper','linear','Distribution','poisson')
```

1. Adding x5, Deviance = 134.439, Chi2Stat = 52.24814, PValue = 4.891229e-13
2. Adding x15, Deviance = 106.285, Chi2Stat = 28.15393, PValue = 1.1204e-07



3. Adding x10, Deviance = 95.0207, Chi2Stat = 11.2644, PValue = 0.000790094

mdl =

Generalized Linear regression model:

$\log(y) \sim 1 + x5 + x10 + x15$

Distribution = Poisson

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	1.0115	0.064275	15.737	8.4217e-56
x5	0.39508	0.066665	5.9263	3.0977e-09
x10	0.18863	0.05534	3.4085	0.0006532
x15	0.29295	0.053269	5.4995	3.8089e-08

100 observations, 96 error degrees of freedom

Dispersion: 1

Chi^2-statistic vs. constant model: 91.7, p-value = 9.61e-20

The starting model is the constant model. `stepwiseglm` by default uses deviance of the model as the criterion. It first adds `x5` into the model, as the  $p$ -value for the test statistic, deviance (the differences in the deviances of the two models), is less than the default threshold value 0.05. Then, it adds `x15` because given `x5` is in the model, when `x15` is added, the  $p$ -value for chi-squared test is smaller than 0.05. It then adds `x10` because given `x5` and `x15` are in the model, when `x10` is added, the  $p$ -value for the chi-square test statistic is again less than 0.05.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables A, B, and C:

# stepwiseglm

---

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

```

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

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```
load carsmall
X = [Acceleration,Weight];
```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```

0     0     0
1     0     0
0     1     0
1     1     0
0     2     0
```

# stepwiseglm

---

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

$$y \sim 1 + x1*x2 + x2^2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

Number of observations: 94, Error degrees of freedom: 89

Root Mean Squared Error: 4.1

R-squared: 0.751, Adjusted R-Squared 0.739

F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];
```

```
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

```
mdl =
```

Linear regression model:

$$y \sim 1 + x2$$

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92

Root Mean Squared Error: 4.13

R-squared: 0.738, Adjusted R-Squared 0.735

F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

### Formula

A formula for model specification is a string of the form

`'Y ~ terms'`,

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include -1 in the formula.

---

For example,

' $Y \sim A + B + C$ ' means a three-variable linear model with intercept.

' $Y \sim A + B + C - 1$ ' is a three-variable linear model without intercept.

' $Y \sim A + B + C + B^2$ ' is a three-variable model with intercept and a  $B^2$  term.

' $Y \sim A + B^2 + C$ ' is the same as the previous example because  $B^2$  includes a  $B$  term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of  $A$  with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only

Wilkinson Notation	Factors in Standard Notation
-B	Do not include B
A*B + C	A, B, C, A*B
A + B + C + A:B	A, B, C, A*B
A*B*C - A:B:C	A, B, C, A*B, A*C, B*C
A*(B + C)	A, B, C, A*B, A*C

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using -1.

### Canonical Function

The default link function for a generalized linear model is the *canonical link function*.

### Canonical Link Functions for Generalized Linear Models

Distribution	Link Function Name	Link Function	Mean (Inverse) Function
'normal'	'identity'	$f(\mu) = \mu$	$\mu = Xb$
'binomial'	'logit'	$f(\mu) = \log(\mu/(1-\mu))$	$\mu = \exp(Xb) / (1 + \exp(Xb))$
'poisson'	'log'	$f(\mu) = \log(\mu)$	$\mu = \exp(Xb)$
'gamma'	-1	$f(\mu) = 1/\mu$	$\mu = 1/(Xb)$
'inverse gaussian'	-2	$f(\mu) = 1/\mu^2$	$\mu = (Xb)^{-1/2}$

### Tips

- The generalized linear model mdl is a standard linear model unless you specify otherwise with the Distribution name-value pair.
- For other methods such as devianceTest, or properties of the GeneralizedLinearModel object, see GeneralizedLinearModel.

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. The default value of 'Criterion' is 'sse', and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when 'Criterion' is 'sse':

- 1** Fit the initial model.
- 2** If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3** If any terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

The default for `stepwiseglm` is 'Deviance' and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the 'Criterion' argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.



Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

**Alternatives**

Use `fitglm` to create a model with a fixed specification. Use `step`, `addTerms`, or `removeTerms` to adjust a fitted model.

**References**

- [1] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [2] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [3] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.

**See Also**

GeneralizedLinearModel | `fitglm`

**Related Examples**

- “Compare large and small stepwise models” on page 9-119

**Concepts**

- “Generalized Linear Models” on page 10-12

# stepwiselm

---

**Purpose** Create linear regression model using stepwise regression

**Syntax**

```
mdl = stepwiselm(ds,modelspec)
mdl = stepwiselm(X,y,modelspec)
mdl = stepwiselm(___,Name,Value)
```

**Description** `mdl = stepwiselm(ds,modelspec)` returns a linear model for the variables in the dataset array `ds` using stepwise regression to add or remove predictors. `stepwiselm` uses forward and backward stepwise regression to determine a final model. At each step, the function searches for terms to add to or remove from the model based on the value of the 'Criterion' argument. `modelspec` is the starting model for the stepwise procedure.

`mdl = stepwiselm(X,y,modelspec)` creates a linear model of the responses `y` to the predictor variables in the data matrix `X`, using stepwise regression to add or remove predictors. `modelspec` is the starting model for the stepwise procedure.

`mdl = stepwiselm(___,Name,Value)` creates a linear model for any of the inputs in the previous syntaxes, with additional options specified by one or more `Name,Value` pair arguments.

For example, you can specify the categorical variables, the smallest or largest set of terms to use in the model, the maximum number of steps to take, or the criterion `stepwiselm` uses to add or remove terms.

## Input Arguments

**ds - Input data**  
dataset array

Input data, specified as a dataset array. When `modelspec` is a formula, it specifies the variables to be used as the predictors and response. Otherwise, if you do not specify the predictor and response variables, the last variable is the response variable and the others are the predictor variables by default.

Predictor variables can be numeric, or any grouping variable type, such as logical or categorical (see “Grouping Variables” on page 2-51). The response must be numeric or logical.

To set a different column as the response variable, use the `ResponseVar` name-value pair argument. To use a subset of the columns as predictors, use the `PredictorVars` name-value pair argument.

### Data Types

single | double | logical

### X - Predictor variables

matrix

Predictor variables, specified as an  $n$ -by- $p$  matrix, where  $n$  is the number of observations and  $p$  is the number of predictor variables. Each column of  $X$  represents one variable, and each row represents one observation.

By default, there is a constant term in the model, unless you explicitly remove it, so do not include a column of 1s in  $X$ .

### Data Types

single | double | logical

### y - Response variable

vector

Response variable, specified as an  $n$ -by-1 vector, where  $n$  is the number of observations. Each entry in  $y$  is the response for the corresponding row of  $X$ .

### Data Types

single | double

### modelspec - Starting model

string specifying the model |  $t$ -by- $(p+1)$  terms matrix | string of the form 'Y ~ terms'

Starting model for the stepwise regression, specified as one of the following:

- String specifying the type of starting model.

String	Model Type
'constant'	Model contains only a constant (intercept) term.
'linear'	Model contains an intercept and linear terms for each predictor.
'interactions'	Model contains an intercept, linear terms, and all products of pairs of distinct predictors (no squared terms).
'purequadratic'	Model contains an intercept, linear terms, and squared terms.
'quadratic'	Model contains an intercept, linear terms, interactions, and squared terms.
'polyijk'	Model is a polynomial with all terms up to degree $i$ in the first predictor, degree $j$ in the second predictor, etc. Use numerals 0 through 9. For example, 'poly2111' has a constant plus all linear and product terms, and also contains terms with predictor 1 squared.

If you want to specify the smallest or largest set of terms in the model, use the **Lower** and **Upper** name-value pair arguments.

- $t$ -by- $(p+1)$  matrix, namely a terms matrix, specifying terms to include in model, where  $t$  is the number of terms and  $p$  is the number of predictor variables, and plus one is for the response variable.
- String representing a formula in the form

`'Y ~ terms'`,

where the terms are in “Wilkinson Notation” on page 21-3332.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'Criterion', 'aic', 'Upper', interactions, 'Verbose', 1` instructs `stepwiselm` to use the Akaike information criterion, display the action it takes at each step, and include at most the interaction terms in the model.

### 'CategoricalVars' - Categorical variables

cell array of strings | logical or numeric index vector

Categorical variables in the fit, specified as the comma-separated pair consisting of `'CategoricalVars'` and either a cell array of strings of the names of the categorical variables in the dataset array `ds`, or a logical or numeric index vector indicating which columns are categorical.

- If data is in dataset array `ds`, then the default is to treat all categorical or logical variables, character arrays, or cell arrays of strings as categorical variables.
- If data is in matrix `X`, then the default value of this name-value pair argument is an empty matrix `[]`. That is, no variable is categorical unless you specify it.

For example, you can specify the observations 2 and 3 out of 6 as categorical using either of the following examples.

**Example:** `'CategoricalVars', [2,3]`

**Example:** `'CategoricalVars', logical([0 1 1 0 0 0])`

### Data Types

single | double | logical

### 'Criterion' - Criterion to add or remove terms

`'Deviance'` | `'sse'` | `'aic'` | `'bic'` | `'rsquared'` | `'adjrsquared'`

Criterion to add or remove terms, specified as the comma-separated pair consisting of 'Criterion' and one of the following:

- 'Deviance' — Default for `stepwiseglm`.  $p$ -value for  $F$  or chi-squared test of the change in the deviance by adding or removing the term.  $F$ -test is for testing a single model. Chi-squared test is for comparing two different models. This option is not valid for `stepwiselm`.
- 'sse' — Default for `stepwiselm`.  $p$ -value for an  $F$ -test of the change in the sum of squared error by adding or removing the term.
- 'aic' — Change in the value of Akaike information criterion (AIC).
- 'bic' — Change in the value of Bayesian information criterion (BIC).
- 'rsquared' — Increase in the value of  $R^2$ .
- 'adjrsquared' — Increase in the value of adjusted  $R^2$ .

**Example:** 'Criterion','bic'

## 'Exclude' - Observations to exclude

logical or numeric index vector

Observations to exclude from the fit, specified as the comma-separated pair consisting of 'Exclude' and a logical or numeric index vector indicating which observations to exclude from the fit.

For example, you can exclude observations 2 and 3 out of 6 using either of the following examples.

**Example:** 'Exclude',[2,3]

**Example:** 'Exclude',logical([0 1 1 0 0 0])

## Data Types

single | double | logical

## 'Intercept' - Indicator for constant term

true (default) | false

Indicator the for constant term (intercept) in the fit, specified as the comma-separated pair consisting of 'Intercept' and either true to include or false to remove the constant term from the model.

Use 'Intercept' only when specifying the model using a string, not a formula or matrix.

**Example:** 'Intercept',false

### **'Lower' - Model specification describing smallest set of terms in fit**

'constant' (default)

Model specification describing the smallest set of terms in the fit, specified as the comma-separated pair consisting of 'Lower' and one of the string options for `modelspec` naming the model.

**Example:** 'Lower', 'linear'

### **'NSteps' - Number of steps to take**

1 (default) | positive integer

Number of steps to take, specified as the comma-separated pair consisting of 'NSteps' and a positive integer.

### **Data Types**

single | double

### **'PEnter' - Improvement measure for adding term**

scalar value

Improvement measure for adding a term, specified as the comma-separated pair consisting of 'PEnter' and a scalar value. The default values are below.

# stepwiselm

---

Criterion	Default value	Decision
'Deviance'	0.05	If the $p$ -value of $F$ or chi-squared statistic is smaller than PEnter, add the term to the model.
'SSE'	0.05	If the SSE of the model is smaller than PEnter, add the term to the model.
'AIC'	0	If the change in the AIC of the model is smaller than PEnter, add the term to the model.
'BIC'	0	If the change in the BIC of the model is smaller than PEnter, add the term to the model.
'Rsquared'	0.1	If the increase in the R-squared of the model is larger than PEnter, add the term to the model.
'AdjRsquared'	0	If the increase in the adjusted R-squared of the model is larger than PEnter, add the term to the model.

For more information on the criteria, see `Criterion` name-value pair argument.



**Example:** 'PEnter',0.075

### **'PredictorVars' - Predictor variables**

cell array of strings | logical or numeric index vector

Predictor variables to use in the fit, specified as the comma-separated pair consisting of 'PredictorVars' and either a cell array of strings of the variable names in the dataset array `ds`, or a logical or numeric index vector indicating which columns are predictor variables.

The strings should be among the names in the `ds` dataset array, or the names you specify using the 'VarNames' name-value pair argument.

The default is all variables in `X`, or all variables in `ds` except for `ResponseVar`.

For example, you can specify the second and third variables as the predictor variables using either of the following examples.

**Example:** 'PredictorVars',[2,3]

**Example:** 'PredictorVars',logical([0 1 1 0 0 0])

### **Data Types**

single | double | logical | cell

### **'PRemove' - Improvement measure for removing term**

scalar value

Improvement measure for removing a term, specified as the comma-separated pair consisting of 'PRemove' and a scalar value.

# stepwiselm

---

Criterion	Default value	Decision
'Deviance'	0.10	If the $p$ -value of $F$ or chi-squared statistic is larger than <b>PRemove</b> , remove the term from the model.
'SSE'	0.10	If the $p$ -value of the $F$ statistic is larger than <b>PRemove</b> , remove the term from the model.
'AIC'	0.01	If the change in the AIC of the model is larger than <b>PRemove</b> , remove the term from the model.
'BIC'	0.01	If the change in the BIC of the model is larger than <b>PRemove</b> , remove the term from the model.
'Rsquared'	0.05	If the increase in the R-squared value of the model is smaller than <b>PRemove</b> , remove the term from the model.
'AdjRsquared'	-0.05	If the increase in the adjusted R-squared value of the model is smaller than <b>PRemove</b> , remove the term from the model.

At each step, stepwise algorithm also checks whether any term is redundant (linearly dependent) with other terms in the current model. When any term is linearly dependent with other terms in the current model, it is removed, regardless of the criterion value.

For more information on the criteria, see `Criterion` name-value pair argument.

**Example:** `'PRemove',0.05`

### **'ResponseVar' - Response variable**

last column in dataset array `ds` (default) | string for variable name  
| logical or numeric index vector

Response variable to use in the fit, specified as the comma-separated pair consisting of `'ResponseVar'` and either a string of the variable name in the dataset array `ds`, or a logical or numeric index vector indicating which column is the response variable. You typically need to use `'ResponseVar'` when fitting a dataset array `ds`.

For example, you can specify the fourth variable, say `yield`, as the response out of six variables, in one of the following ways.

**Example:** `'ResponseVar','yield'`

**Example:** `'ResponseVar',[4]`

**Example:** `'ResponseVar',logical([0 0 0 1 0 0])`

### **Data Types**

single | double | logical | char

### **'Upper' - Model specification describing largest set of terms in fit**

`'interaction'` (default) | string

Model specification describing the largest set of terms in the fit, specified as the comma-separated pair consisting of `'Upper'` and one of the string options for `modelspec` naming the model.

**Example:** `'Upper','quadratic'`

### **'VarNames' - Names of variables in fit**

`{'x1','x2',...,'xn','y'}` (default) | cell array of strings

Names of variables in fit, specified as the comma-separated pair consisting of `'VarNames'` and a cell array of strings including the names for the columns of  $X$  first, and the name for the response variable  $y$  last.

`'VarNames'` is not applicable to variables in a dataset array, because those variables already have names.

For example, if in your data, horsepower, acceleration, and model year of the cars are the predictor variables, and miles per gallon (MPG) is the response variable, then you can name the variables as follows.

**Example:**

`'VarNames',{ 'Horsepower','Acceleration','Model_Year','MPG'}`

**Data Types**

cell

**'Verbose' - Control for display of information**

1 (default) | 0 | 2

Control for display of information, specified as the comma-separated pair consisting of `'Verbose'` and one of the following:

- 0 — Suppress all display.
- 1 — Display the action taken at each step.
- 2 — Also display the actions evaluated at each step.

**Example:** `'Verbose',2`

**'Weights' - Observation weights**

`ones(n,1)` (default) |  $n$ -by-1 vector of nonnegative scalar values

Observation weights, specified as the comma-separated pair consisting of `'Weights'` and an  $n$ -by-1 vector of nonnegative scalar values, where  $n$  is the number of observations.

**Data Types**

single | double

**Output Arguments****mdl - Linear model**

LinearModel object

Linear model representing a least-squares fit of the response to the data, returned as a LinearModel object.

For the properties and methods of the linear model object, mdl, see the LinearModel class page.

**Examples****Linear Model Using Stepwise Regression**

Load the sample data.

```
load hald
```

hald contains hardening data for 13 different concrete compositions. heat is the heat of hardening after 180 days. ingredients is the percentage of each different ingredient in the cement sample.

Fit a linear model to the data. Set the criterion value to enter the model as 0.06.

```
mdl = stepwiselm(ingredients,heat,'PEnter',0.06)
```

1. Adding x4, FStat = 22.7985, pValue = 0.000576232
2. Adding x1, FStat = 108.2239, pValue = 1.105281e-06
3. Adding x2, FStat = 5.0259, pValue = 0.051687
4. Removing x4, FStat = 1.8633, pValue = 0.2054

```
mdl =
```

```
Linear regression model:
```

```
y ~ 1 + x1 + x2
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	52.577	2.2862	22.998	5.4566e-10
x1	1.4683	0.1213	12.105	2.6922e-07

# stepwiselm

---

```
x2                0.66225      0.045855      14.442      5.029e-08
```

```
Number of observations: 13, Error degrees of freedom: 10
Root Mean Squared Error: 2.41
R-squared: 0.979, Adjusted R-Squared 0.974
F-statistic vs. constant model: 230, p-value = 4.41e-09
```

By default, the starting model is the constant model. `stepwiselm` performs forward selection and `x4`, `x1`, and `x2`, respectively, as the corresponding  $p$ -values are less than the `PEnter` value of 0.06. `stepwiselm` later uses backward elimination and eliminates `x4` from the model. Because, given that `x2` is in the model, the  $p$ -value of `x4` is higher than the default value of `PRemove`, 0.1.

## Specify Model Using Formula and Specify Variables

Perform stepwise regression with variables in a dataset array. Specify the starting model using formula, and identify the response and predictor variables with optional arguments.

Load the sample data.

```
load hospital
```

The `hospital` dataset array includes the gender, age, weight, and smoking status of patients.

Fit a linear model with a starting model of a constant term and `Smoker` as the predictor variable. Specify the response variable, `Weight`, and categorical predictor variables, `Sex`, `Age`, and `Smoker`.

```
mdl = stepwiselm(hospital, 'Weight~1+Smoker', ...
'ResponseVar', 'Weight', 'PredictorVars', {'Sex', 'Age', 'Smoker'}, ...
'CategoricalVar', {'Sex', 'Smoker'})
```

1. Adding `Sex`, `FStat` = 770.0158, `pValue` = 6.262758e-48
2. Removing `Smoker`, `FStat` = 0.21224, `pValue` = 0.64605

```
mdl =
```

```
Linear regression model:
```

```
Weight ~ 1 + Sex
```

```
Estimated Coefficients:
```

	Estimate	SE	tStat	pValue
(Intercept)	130.47	1.1995	108.77	5.2762e-104
Sex_Male	50.06	1.7496	28.612	2.2464e-49

```
Number of observations: 100, Error degrees of freedom: 98
```

```
Root Mean Squared Error: 8.73
```

```
R-squared: 0.893, Adjusted R-Squared 0.892
```

```
F-statistic vs. constant model: 819, p-value = 2.25e-49
```

At each step, `stepwiselm` searches for terms to add and remove. At first step, stepwise algorithm adds `Sex` to the model with a  $p$ -value of  $6.26e-48$ . Then, removes `Smoker` from the model, since given `Sex` in the model, the variable `Smoker` becomes redundant. `stepwiselm` only includes `Sex` in the final linear model. The weight of the patients do not seem to differ significantly according to age or the status of smoking.

## Definitions

### Terms Matrix

A terms matrix is a  $t$ -by- $(p + 1)$  matrix specifying terms in a model, where  $t$  is the number of terms,  $p$  is the number of predictor variables, and plus one is for the response variable. The value of  $T(i, j)$  is the exponent of variable  $j$  in term  $i$ . Suppose there are three predictor variables `A`, `B`, and `C`:

```
[0 0 0 0] % Constant term or intercept
[0 1 0 0] % B; equivalently, A^0 * B^1 * C^0
[1 0 1 0] % A*C
[2 0 0 0] % A^2
[0 1 2 0] % B*(C^2)
```

The 0 at the end of each term represents the response variable. In general,

# stepwiselm

---

- If you have the variables in a dataset array, then 0 must represent the response variable depending on the position of the response variable in the dataset array. The following example illustrates this.

Load the sample data and define the dataset array.

```
load hospital
ds = dataset(hospital.Sex,hospital.BloodPressure(:,1),hospital.Age,...
hospital.Smoker,'VarNames',{'Sex','BloodPressure','Age','Smoker'});
```

Represent the linear model 'BloodPressure ~ 1 + Sex + Age + Smoker' in a terms matrix. The response variable is in the second column of the dataset array, so there must be a column of 0s for the response variable in the second column of the terms matrix.

```
T = [0 0 0 0;1 0 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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

Redefine the dataset array.

```
ds = dataset(hospital.BloodPressure(:,1),hospital.Sex,hospital.Age,...
hospital.Smoker,'VarNames',{'BloodPressure','Sex','Age','Smoker'});
```

Now, the response variable is the first term in the dataset array. Specify the same linear model, 'BloodPressure ~ 1 + Sex + Age + Smoker', using a terms matrix.

```
T = [0 0 0 0;0 1 0 0;0 0 1 0;0 0 0 1]
```

```
T =
```

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



```

0 0 1 0
0 0 0 1

```

- If you have the predictor and response variables in a matrix and column vector, then you must include 0 for the response variable at the end of each term. The following example illustrates this.

Load the sample data and define the matrix of predictors.

```

load carsmall
X = [Acceleration,Weight];

```

Specify the model 'MPG ~ Acceleration + Weight + Acceleration:Weight + Weight^2' using a term matrix and fit the model to the data. This model includes the main effect and two-way interaction terms for the variables, Acceleration and Weight, and a second-order term for the variable, Weight.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0;0 2 0]
```

```
T =
```

```

0 0 0
1 0 0
0 1 0
1 1 0
0 2 0

```

Fit a linear model.

```
mdl = fitlm(X,MPG,T)
```

```
mdl =
```

Linear regression model:

```
y ~ 1 + x1*x2 + x2^2
```

Estimated Coefficients:

# stepwiselm

	Estimate	SE	tStat	pValue
(Intercept)	48.906	12.589	3.8847	0.00019665
x1	0.54418	0.57125	0.95261	0.34337
x2	-0.012781	0.0060312	-2.1192	0.036857
x1:x2	-0.00010892	0.00017925	-0.6076	0.545
x2^2	9.7518e-07	7.5389e-07	1.2935	0.19917

Number of observations: 94, Error degrees of freedom: 89  
Root Mean Squared Error: 4.1  
R-squared: 0.751, Adjusted R-Squared 0.739  
F-statistic vs. constant model: 67, p-value = 4.99e-26

Only the intercept and x2 term, which correspond to the Weight variable, are significant at the 5% significance level.

Now, perform a stepwise regression with a constant model as the starting model and a linear model with interactions as the upper model.

```
T = [0 0 0;1 0 0;0 1 0;1 1 0];  
mdl = stepwiselm(X,MPG,[0 0 0],'upper',T)
```

1. Adding x2, FStat = 259.3087, pValue = 1.643351e-28

mdl =

Linear regression model:  
y ~ 1 + x2

Estimated Coefficients:

	Estimate	SE	tStat	pValue
(Intercept)	49.238	1.6411	30.002	2.7015e-49
x2	-0.0086119	0.0005348	-16.103	1.6434e-28

Number of observations: 94, Error degrees of freedom: 92  
Root Mean Squared Error: 4.13  
R-squared: 0.738, Adjusted R-Squared 0.735  
F-statistic vs. constant model: 259, p-value = 1.64e-28

The results of the stepwise regression are consistent with the results of `fitlm` in the previous step.

### Formula

A formula for model specification is a string of the form

`'Y ~ terms'`,

where

- `Y` is the response name.
- `terms` contains
  - Variable names
  - `+` means include the next variable
  - `-` means do not include the next variable
  - `:` defines an interaction, a product of terms
  - `*` defines an interaction **and all lower-order terms**
  - `^` raises the predictor to a power, exactly as in `*` repeated, so `^` includes lower order terms as well
  - `()` groups terms

---

**Note** Formulas include a constant (intercept) term by default. To exclude a constant term from the model, include `-1` in the formula.

---

For example,

`'Y ~ A + B + C'` means a three-variable linear model with intercept.

`'Y ~ A + B + C - 1'` is a three-variable linear model without intercept.

`'Y ~ A + B + C + B^2'` is a three-variable model with intercept

and a  $B^2$  term.

' $Y \sim A + B^2 + C$ ' is the same as the previous example because  $B^2$  includes a  $B$  term.

' $Y \sim A + B + C + A:B$ ' includes an  $A*B$  term.

' $Y \sim A*B + C$ ' is the same as the previous example because  $A*B = A + B + A:B$ .

' $Y \sim A*B*C - A:B:C$ ' has all interactions among  $A$ ,  $B$ , and  $C$ , except the three-way interaction.

' $Y \sim A*(B + C + D)$ ' has all linear terms, plus products of  $A$  with each of the other variables.

## Wilkinson Notation

Wilkinson notation describes the factors present in models. The notation relates to factors present in models, not to the multipliers (coefficients) of those factors.

Wilkinson Notation	Factors in Standard Notation
1	Constant (intercept) term
$A^k$ , where $k$ is a positive integer	$A, A^2, \dots, A^k$
$A + B$	$A, B$
$A*B$	$A, B, A*B$
$A:B$	$A*B$ only
$-B$	Do not include $B$
$A*B + C$	$A, B, C, A*B$
$A + B + C + A:B$	$A, B, C, A*B$
$A*B*C - A:B:C$	$A, B, C, A*B, A*C, B*C$
$A*(B + C)$	$A, B, C, A*B, A*C$

Statistics Toolbox notation always includes a constant term unless you explicitly remove the term using  $-1$ .

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a linear or generalized linear model based on their statistical

significance in explaining the response variable. The method begins with an initial model, specified using `modelspec`, and then compares the explanatory power of incrementally larger and smaller models.

MATLAB uses forward and backward stepwise regression to determine a final model. At each step, the method searches for terms to add to or remove from the model based on the value of the `'Criterion'` argument. The default value of `'Criterion'` is `'sse'`, and in this case, `stepwiselm` uses the  $p$ -value of an  $F$ -statistic to test models with and without a potential term at each step. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model.

Here is how stepwise proceeds when `'Criterion'` is `'sse'`:

- 1** Fit the initial model.
- 2** If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$ -value and repeat this step; otherwise, go to step 3.
- 3** If any terms in the model have  $p$ -values greater than an exit tolerance (that is, the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$ -value and go to step 2; otherwise, end.

The default for `stepwiseglm` is `'Deviance'` and it follows a similar procedure for adding or removing terms.

There are several other criteria available, which you can specify using the `'Criterion'` argument. You can use the change in the value of the Akaike information criterion, Bayesian information criterion, R-squared, adjusted R-squared as a criterion to add or remove terms.

# stepwiselm

---

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method might build different models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but might not be globally optimal.

## Tips

- You cannot use robust regression with stepwise regression. Check your data for outliers before using `stepwiselm`.
- For other methods such as `anova`, or properties of the `LinearModel` object, see `LinearModel`.

## Alternative Functionality

You can construct a model using `fitlm`, and then manually adjust the model using `step`, `addTerms`, or `removeTerms`.

## See Also

`LinearModel` | `step` | `fitlm`

## Related Examples

- “Compare large and small stepwise models” on page 9-119
- “Linear Regression” on page 9-12

## Concepts

- “Stepwise Regression” on page 9-119

**Purpose**

Stepwise regression

**Syntax**

```
b = stepwisefit(X,y)
[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)
[...] = stepwisefit(X,y,param1,va11,param2,va12,...)
```

**Description**

`b = stepwisefit(X,y)` uses a stepwise method to perform a multilinear regression of the response values in the  $n$ -by-1 vector  $y$  on the  $p$  predictive terms in the  $n$ -by- $p$  matrix  $X$ . Distinct predictive terms should appear in different columns of  $X$ .

$b$  is a  $p$ -by-1 vector of estimated coefficients for all of the terms in  $X$ . The `stepwisefit` function calculates the coefficient estimate values in  $b$  as follows:

- If a term is not in the final model, then the corresponding coefficient estimate in  $b$  results from adding only that term to the predictors in the final model.
- If a term is in the final model, then the coefficient estimate in  $b$  for that term is a result of the final model, that is `stepwise` does not consider the terms it excluded from the model while computing these values.

---

**Note** `stepwisefit` automatically includes a constant term in all models. Do not enter a column of 1s directly into  $X$ .

---

`stepwisefit` treats NaN values in either  $X$  or  $y$  as missing values, and ignores them.

`[b,se,pval,inmodel,stats,nextstep,history] = stepwisefit(...)` returns the following additional information:

- `se` — A vector of standard errors for  $b$
- `pval` — A vector of  $p$ -values for testing whether elements of  $b$  are 0

## stepwisefit

---

- `inmodel` — A logical vector, with length equal to the number of columns in `X`, specifying which terms are in the final model
- `stats` — A structure of additional statistics with the following fields. All statistics pertain to the final model except where noted.
  - `source` — The string 'stepwisefit'
  - `dfe` — Degrees of freedom for error
  - `df0` — Degrees of freedom for the regression
  - `SStotal` — Total sum of squares of the response
  - `SSresid` — Sum of squares of the residuals
  - `fstat` —  $F$ -statistic for testing the final model vs. no model (mean only)
  - `pval` —  $p$  value of the  $F$ -statistic
  - `rmse` — Root mean square error
  - `xr` — Residuals for predictors not in the final model, after removing the part of them explained by predictors in the model
  - `yr` — Residuals for the response using predictors in the final model
  - `B` — Coefficients for terms in final model, with values for a term not in the model set to the value that would be obtained by adding that term to the model
  - `SE` — Standard errors for coefficient estimates
  - `TSTAT` —  $t$  statistics for coefficient estimates
  - `PVAL` —  $p$ -values for coefficient estimates
  - `intercept` — Estimated intercept
  - `wasnan` — Indicates which rows in the data contained NaN values
- `nextstep` — The recommended next step—either the index of the next term to move in or out of the model, or 0 if no further steps are recommended



- **history** — Structure containing information on steps taken, with the following fields:
  - **B** — Matrix of regression coefficients, where each column is one step, and each row is one coefficient.
  - **rmse** — Root mean square errors for the model at each step.
  - **df0** — Degrees of freedom for the regression at each step.
  - **in** — Logical array indicating which predictors are in the model at each step, where each row is one step, and each column is one predictor.

[...] = stepwisefit(X,y,param1,val1,param2,val2,...)

specifies one or more of the name/value pairs described in the following table.

Parameter	Value
'inmodel'	A logical vector specifying terms to include in the initial fit. The default is to specify no terms.
'penter'	The maximum $p$ value for a term to be added. The default is 0.05.
'premove'	The minimum $p$ value for a term to be removed. The default is the maximum of the value of 'penter' and 0.10.
'display'	'on' displays information about each step in the command window. This is the default. 'off' omits the display.
'maxiter'	The maximum number of steps in the regression. The default is Inf.

Parameter	Value
'keep'	A logical vector specifying terms to keep in their initial state. The default is to specify no terms.
'scale'	'on' centers and scales each column of X (computes z-scores) before fitting.  'off' does not scale the terms. This is the default.

## Algorithms

*Stepwise regression* is a systematic method for adding and removing terms from a multilinear model based on their statistical significance in a regression. The method begins with an initial model and then compares the explanatory power of incrementally larger and smaller models. At each step, the  $p$  value of an  $F$ -statistic is computed to test models with and without a potential term. If a term is not currently in the model, the null hypothesis is that the term would have a zero coefficient if added to the model. If there is sufficient evidence to reject the null hypothesis, the term is added to the model. Conversely, if a term is currently in the model, the null hypothesis is that the term has a zero coefficient. If there is insufficient evidence to reject the null hypothesis, the term is removed from the model. The method proceeds as follows:

- 1 Fit the initial model.
- 2 If any terms not in the model have  $p$ -values less than an entrance tolerance (that is, if it is unlikely that they would have zero coefficient if added to the model), add the one with the smallest  $p$  value and repeat this step; otherwise, go to step 3.
- 3 If any terms in the model have  $p$ -values greater than an exit tolerance (that is, if it is unlikely that the hypothesis of a zero coefficient can be rejected), remove the one with the largest  $p$  value and go to step 2; otherwise, end.

Depending on the terms included in the initial model and the order in which terms are moved in and out, the method may build different

models from the same set of potential terms. The method terminates when no single step improves the model. There is no guarantee, however, that a different initial model or a different sequence of steps will not lead to a better fit. In this sense, stepwise models are locally optimal, but may not be globally optimal.

## Examples

Load the data in `hald.mat`, which contains observations of the heat of reaction of various cement mixtures:

```
load hald
whos
```

Name	Size	Bytes	Class	Attributes
Description	22x58	2552	char	
hald	13x5	520	double	
heat	13x1	104	double	
ingredients	13x4	416	double	

The response (heat) depends on the quantities of the four predictors (the columns of ingredients).

Use `stepwisefit` to carry out the stepwise regression algorithm, beginning with no terms in the model and using entrance/exit tolerances of 0.05/0.10 on the  $p$ -values:

```
stepwisefit(ingredients,heat,...
            'penter',0.05,'premove',0.10);
Initial columns included: none
Step 1, added column 4, p=0.000576232
Step 2, added column 1, p=1.10528e-006
Final columns included: 1 4
```

'Coeff'	'Std.Err.'	'Status'	'P'
[ 1.4400]	[ 0.1384]	'In'	[1.1053e-006]
[ 0.4161]	[ 0.1856]	'Out'	[ 0.0517]
[-0.4100]	[ 0.1992]	'Out'	[ 0.0697]
[-0.6140]	[ 0.0486]	'In'	[1.8149e-007]

# stepwisefit

---

stepwisefit automatically includes an intercept term in the model, so you do not add it explicitly to ingredients as you would for regress. For terms not in the model, coefficient estimates and their standard errors are those that result by adding the corresponding term to the final model.

The inmodel parameter is used to specify terms in an initial model:

```
initialModel = ...
    [false true false false]; % Force in 2nd term
stepwisefit(ingredients,heat,...
    'inmodel',initialModel,...
    'penter',.05,'premove',0.10);
Initial columns included: 2
Step 1, added column 1, p=2.69221e-007
Final columns included: 1 2
    'Coeff'      'Std.Err.'    'Status'    'P'
    [ 1.4683]    [ 0.1213]    'In'        [2.6922e-007]
    [ 0.6623]    [ 0.0459]    'In'        [5.0290e-008]
    [ 0.2500]    [ 0.1847]    'Out'       [ 0.2089]
    [-0.2365]    [ 0.1733]    'Out'       [ 0.2054]
```

The preceding two models, built from different initial models, use different subsets of the predictive terms. Terms 2 and 4, swapped in the two models, are highly correlated:

```
term2 = ingredients(:,2);
term4 = ingredients(:,4);
R = corrcoef(term2,term4)
R =
    1.0000    -0.9730
   -0.9730     1.0000
```

To compare the models, use the stats output of stepwisefit:

```
[betahat1,se1,pval1,inmodel1,stats1] = ...
    stepwisefit(ingredients,heat,...
    'penter',.05,'premove',0.10,...
```

```
        'display','off');
[betahat2,se2,pval2,inmodel2,stats2] = ...
    stepwisefit(ingredients,heat,...
                'inmodel',initialModel,...
                'penter',.05,'premove',0.10,...
                'display','off');
```

RMSE1 = stats1.rmse  
RMSE1 =  
    2.7343  
RMSE2 = stats2.rmse  
RMSE2 =  
    2.4063

The second model has a lower Root Mean Square Error (RMSE).

## References

[1] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 1998. pp. 307–312.

## See Also

[stepwise](#) | [addedvarplot](#) | [regress](#)

# categorical.subsasgn

---

**Purpose** Subscripted assignment for categorical array

**Syntax** `A = subsasgn(A,S,B)`

**Description** `A = subsasgn(A,S,B)` is called for the syntax `A(I) = B`, or `A.I = B` when `A` is an object. `S` is a structure array with the following fields.

<code>type</code>	String containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or string containing the actual subscripts.

**See Also** `categorical` | `subsref`

**Purpose** Subscripted reference for classregtree object

**Syntax**

**Description** Subscript assignment is not allowed for a classregtree object.

**See Also** classregtree

# dataset.subsasgn

---

**Purpose** Subscripted assignment to dataset array

**Description** `A = subsasgn(A,S,B)` is called for the syntax `A(i,j)=B`, `A{i,j}=B`, or `A.var=B` when `A` is a dataset array. `S` is a structure array with the fields:

<code>type</code>	String containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or string containing the actual subscripts.

`A(i,j) = B` assigns the contents of the dataset array `B` to a subset of the observations and variables in the dataset array `A`. `i` and `j` are one of the following types:

- positive integers
- vectors of positive integers
- observation/variable names
- cell arrays containing one or more observation/variable names
- logical vectors

The assignment does not use observation names, variable names, or any other properties of `B` to modify properties of `A`; however properties of `A` are extended with default values if the assignment expands the number of observations or variables in `A`. Elements of `B` are assigned into `A` by position, not by matching names.

`A{i,j} = B` assigns the value `B` into an element of the dataset array `A`. `i` and `J` are positive integers, or logical vectors. Cell indexing cannot assign into multiple dataset elements, that is, the subscripts `i` and `j` must each refer to only a single observation or variable. `B` is cast to the type of the target variable if necessary. If the dataset element already exists, `A{i,j}` may also be followed by further subscripting as supported by the variable.



For dataset variables that are cell arrays, assignments such as `A{1, 'CellVar'} = B` assign into the contents of the target dataset element in the same way that `{}`-indexing of an ordinary cell array does.

For dataset variables that are n-D arrays, i.e., each observation is a matrix or array, an assignment such as `A{1, 'ArrayVar'} = B` assigns into the second and following dimensions of the target dataset element, i.e., the assignment adds a leading singleton dimension to `B` to account for the observation dimension of the dataset variable.

`A.var = B` or `A.(varname) = B` assigns `B` to a dataset variable. `var` is a variable name literal, or `varname` is a character variable containing a variable name. If the dataset variable already exists, the assignment completely replaces that variable. To assign into an element of the variable, `A.var` or `A.(varname)` may be followed by further subscripting as supported by the variable. In particular, `A.var(obsnames, ...)` = `B` and `A.var{obsnames, ...}` = `B` (when supported by `var`) provide assignment into a dataset variable using observation names.

`A.properties.propertyname = P` assigns to a dataset property. `propertyname` is one of the following:

- 'ObsNames'
- 'VarNames'
- 'Description'
- 'Units'
- 'DimNames'
- 'UserData'
- 'VarDescription'

To assign into an element of the property, `A.properties.propertyname` may also be followed by further subscripting as supported by the property.

You cannot assign multiple values into dataset variables or properties using assignments such as `[A.CellVar{1:2}] = B`,

## dataset.subsasgn

---

`[A.StructVar(1:2).field] = B`, or `[A.Properties.ObsNames{1:2}] = B`. Use multiple assignments of the form `A.CellVar{1} = B` instead.

Similarly, if a dataset variable is a cell array with multiple columns or is an n-D cell array, then the contents of that variable for a single observation consists of multiple cells, and you cannot assign to all of them using the syntax `A{1, 'CellVar'} = B`. Use multiple assignments of the form `[A.CellVar{1,1}] = B` instead.

### See Also

[dataset](#) | [set](#) | [subsref](#)

<b>Purpose</b>	Subscripted reference for Gaussian mixture distribution object
<b>Description</b>	Subscript assignment is not allowed for gmdistribution objects.
<b>See Also</b>	<code>gmdistribution</code>

# NaiveBayes.subsasgn

---

**Purpose**            Subscripted reference for NaiveBayes object

**Description**      Subscript assignment is not allowed for a NaiveBayes object.

**Purpose** Subscript index for categorical array

**Syntax** `I = subsindex(A)`

**Description** `I = subsindex(A)` is called for the syntax '`X(A)`' when `A` is a categorical array and `X` is one of the built-in types (most commonly 'double'). `subsindex` returns the internal categorical level codes of `A` converted to zero-based integer indices. `subsindex` is invoked separately on all the subscripts in an expression such as `X(A,B)`.

**Examples**

```
load fisheriris
a = ordinal(species,[],unique(species));
colmeans = grpstats(meas,a,@mean);
residuals = meas - colmeans(a,:);
```

**See Also** `categorical` | `double`

# classregtree.subsref

---

**Purpose** Subscripted reference for classregtree object

**Syntax** `B = subsref(T,S)`

**Description** `B = subsref(T,S)` is called for the syntax `T(X)` when `T` is a classregtree object. `S` is a structure array with the fields:

<code>type</code>	String containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or string containing the actual subscripts.

`[...] = T(...)` invokes the `eval` method for the tree `T`.

**See Also** classregtree | eval

**Purpose** Subscripted reference for categorical array

**Syntax** `B = subsref(A,S)`

**Description** `B = subsref(A,S)` is called for the syntax `A(I)`, `A{I}`, or `A.I` when `A` is an object. `S` is a structure array with the following fields.

<code>type</code>	String containing <code>'()'</code> , <code>'{'}</code> , or <code>'.'</code> specifying the subscript type.
<code>subs</code>	Cell array or string containing the actual subscripts.

**See Also** `categorical` | `subsasgn`

# dataset.subsref

---

**Purpose** Subscripted reference for dataset array

**Syntax** `B = subsref(A,S)`

**Description** `B = subsref(A,S)` is called for the syntax `A(i,j)`, `A{i,j}`, or `A.var` when `A` is a dataset array. `S` is a structure array with the fields:

<code>type</code>	String containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or string containing the actual subscripts.

`B = A(i,j)` returns a dataset array that contains a subset of the observations and variables in the dataset array `A`. `i` and `j` are one of the following types:

- positive integers
- vectors of positive integers
- observation/variable names
- cell arrays containing one or more observation/variable names
- logical vectors

`B` contains the same property values as `A`, subsetted for observations or variables where appropriate.

`B = A{i,j}` returns an element of a dataset variable. `i` and `j` are positive integers, or logical vectors. Cell indexing cannot return multiple dataset elements, that is, the subscripts `i` and `j` must each refer to only a single observation or variable. `A{i,j}` may also be followed by further subscripting as supported by the variable.

For dataset variables that are cell arrays, expressions such as `A{1, 'CellVar'}` return the contents of the referenced dataset element in the same way that `{}`-indexing on an ordinary cell array does. If the dataset variable is a single column of cells, the contents of a single cell



is returned. If the dataset variable has multiple columns or is n-D, multiple outputs containing the contents of multiple cells are returned.

For dataset variables that are n-D arrays, i.e., each observation is a matrix or an array, expressions such as `A{1, 'ArrayVar'}` return `A.ArrayVar(1, :, ...)` with the leading singleton dimension squeezed out.

`B = A.var` or `A.(varname)` returns a dataset variable. `var` is a variable name literal, or `varname` is a character variable containing a variable name. `A.var` or `A.(varname)` may also be followed by further subscripting as supported by the variable. In particular, `A.var(obsnames, ...)` and `A.var{obsnames, ...}` (when supported by `var`) provide subscripting into a dataset variable using observation names.

`P = A.Properties.propertyname` returns a dataset property. `propertyname` is one of the following:

- 'ObsNames'
- 'VarNames'
- 'Description'
- 'Units'
- 'DimNames'
- 'UserData'
- 'VarDescription'

`A.properties.propertyname` may also be followed by further subscripting as supported by the property.

### Limitations

Subscripting expressions such as `A.CellVar{1:2}`, `A.StructVar(1:2).field`, or `A.Properties.ObsNames{1:2}` are valid, but result in `subsref` returning multiple outputs in the form of a comma-separated list. If you explicitly assign to output

## dataset.subsref

---

arguments on the left-hand side of an assignment, for example, `[cellval1,cellval2] = A.CellVar{1:2}`, those variables will receive the corresponding values. However, if there are no output arguments, only the first output in the comma-separated list is returned.

Similarly, if a dataset variable is a cell array with multiple columns or is an n-D cell array, then subscripting expressions such as `A{1, 'CellVar'}` result in `subsref` returning the contents of multiple cells. You should explicitly assign to output arguments on the left-hand side of an assignment, for example, `[cellval1,cellval2] = A{1, 'CellVar'}`.

### See Also

[dataset](#) | [set](#) | [subsasgn](#)

**Purpose** Subscripted reference for Gaussian mixture distribution object

**Syntax** `B = subsref(T,S)`

**Description** `B = subsref(T,S)` is called for the syntax `T(X)` when `T` is a `gmdistribution` object. `S` is a structure array with the following fields:

<code>type</code>	String containing '()', '{}', or '.' specifying the subscript type.
<code>subs</code>	Cell array or string containing the actual subscripts.

**See Also** `gmdistribution`

# NaiveBayes.subsref

---

**Purpose** Subscripted reference for NaiveBayes object

**Syntax** `b = subsref(nb,s)`

**Description** `b = subsref(nb,s)` is called for the syntax `nb(s)` when `nb` is a NaiveBayes object. `S` is a structure array with the fields:

<code>type</code>	string containing '()', '{}', or '' specifying the subscript type.
<code>subs</code>	Cell array or string containing the actual subscripts.

**Purpose** Subscripted reference for grandset

**Syntax**  
`x = p(i,j)`  
`x = subsref(p,s)`

**Description** `x = p(i,j)` returns a matrix that contains a subset of the points from the point set `p`. The indices in `i` select points from the set and the indices in `j` select columns from those points. `i` and `j` are vector of positive integers or logical vectors. A colon used as a subscript, as in `p(i,:)`, indicates the entire row (or column).

`x = subsref(p,s)` is called for the syntax `p(i)`, `p{i}`, or `p.i`. `s` is a structure array with the fields:

`type` string containing '()', '{}', or '.' specifying the subscript type.

`subs` Cell array or string containing the actual subscripts.

## Examples

Command	Returns
<code>p = sobolset(5);</code>	The fifth point
<code>x = p(1:10,:)</code>	All columns of the first 10 points
<code>x = p(end,1)</code>	The first column of the last point
<code>x = p([1,4,5], :)</code>	Points 1, 4, and 5

**See Also** grandset

# nominal.summary

---

**Purpose** Summary statistics for nominal array

**Syntax**  
`summary(A)`  
`C = summary(A)`  
`[C,labels] = summary(A)`

**Description** `summary(A)` displays the number of elements in the nominal array `A` equal to each of the possible levels in `A`. If `A` contains any undefined elements, the output also includes the number of undefined elements.

`C = summary(A)` returns counts of the number of elements in the nominal array `A` equal to each of the possible levels in `A`. If `A` is a matrix or  $N$ -dimensional array, `C` is a matrix or array with rows corresponding to the levels of `A`. If `A` contains any undefined elements, `C` contains one more row than the number of levels of `A`, with the number of undefined elements in `c(end)` or `c(end, :)`.

`[C,labels] = summary(A)` also returns the list of categorical level labels corresponding to the counts in `C`.

## **Examples**      **Create and Summarize Nominal Array**

Load the sample data.

```
load fisheriris;
```

The data contains length and width measurements from the sepals and petals of three species of iris flowers.

Create a nominal array containing the flower species data.

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

Display a summary of the flower species data.

```
summary(species)

      setosa      50
versicolor      50
```

```
virginica      50
```

The nominal array contains 50 elements in each level, which corresponds to measurements for 50 flowers each of the setosa, versicolor, and virginica species.

## See Also

`islevel` | `ismember` | `levelcounts`

# ordinal.summary

---

**Purpose** Summary statistics for ordinal array

**Syntax**  
`summary(A)`  
`C = summary(A)`  
`[C,labels] = summary(A)`

**Description** `summary(A)` displays the number of elements in the ordinal array `A` equal to each of the possible levels in `A`. If `A` contains any undefined elements, the output also includes the number of undefined elements.

`C = summary(A)` returns counts of the number of elements in the ordinal array `A` equal to each of the possible levels in `A`. If `A` is a matrix or  $N$ -dimensional array, `C` is a matrix or array with rows corresponding to the levels of `A`. If `A` contains any undefined elements, `C` contains one more row than the number of levels of `A`, with the number of undefined elements in `c(end)` or `c(end, :)`.

`[C,labels] = summary(A)` also returns the list of categorical level labels corresponding to the counts in `C`.

## **Examples**      **Create and Summarize Ordinal Array**

Load the sample data.

```
load hospital;
```

The file contains data obtained from hospital patients, including their age, gender, weight, blood pressure, and smoking status.

Create an ordinal array of the age groups represented in the data.

```
edges = 0:10:100;  
labels = strcat(num2str((0:10:90)', '%d'), {'s'});  
AgeGroup = ordinal(hospital.Age, labels, [], edges);
```

Display the number of patients in each age group.

```
summary(AgeGroup)
```



0s	0
10s	0
20s	15
30s	41
40s	42
50s	2
60s	0
70s	0
80s	0
90s	0

The results show, for example, that the data contains 15 patients of ages 20 through 29, inclusive. These patients are members of the '20s' level in the ordinal array AgeGroup.

## See Also

`islevel` | `ismember` | `levelcounts`

# dataset.summary

---

**Purpose** Print summary of dataset array

**Syntax**  
`summary(A)`  
`s = summary(A)`

**Description** `summary(A)` prints a summary of a dataset array and the variables that it contains.

`s = summary(A)` returns a scalar structure `s` that contains a summary of the dataset `A` and the variables that `A` contains. For more information on the fields in `s`, see [Outputs](#).

Summary information depends on the type of the variables in the data set:

- For numerical variables, `summary` computes a five-number summary of the data, giving the minimum, the first quartile, the median, the third quartile, and the maximum.
- For logical variables, `summary` counts the number of `true`s and `false`s in the data.
- For categorical variables, `summary` counts the number of data at each level.

## Output Arguments

The following list describes the fields in the structure `s`:

- **Description** — A character array containing the dataset description.
- **Variables** — A structure array with one element for each dataset variable in `A`. Each element has the following fields:
  - **Name** — A character string containing the name of the variable.
  - **Description** — A character string containing the variable's description.
  - **Units** — A character string containing the variable's units.
  - **Size** — A numeric vector containing the size of the variable.

- **Class** — A character string containing the class of the variable.
- **Data** — A scalar structure containing the following fields.

For numeric variables:

- **Probabilities** — A numeric vector containing the probabilities [0.0 .25 .50 .75 1.0] and NaN (if any are present in the corresponding dataset variable).
- **Quantiles** — A numeric vector containing the values that correspond to 'Probabilities' for the corresponding dataset variable, and a count of NaNs (if any are present).

For logical variables:

- **Values** — The logical vector [true false].
- **Counts** — A numeric vector of counts for each logical value.

For categorical variables:

- **Levels** — A cell array containing the labels for each level of the corresponding dataset variable.
- **Counts** — A numeric vector of counts for each level.

'Data' is empty if variable is not numeric, categorical, or logical. If a dataset variable has more than one column, then the corresponding 'Quantiles' or 'Counts' field is a matrix or an array.

## Examples

Summarize Fisher's iris data:

```
load fisheriris
species = nominal(species);
data = dataset(species,meas);
summary(data)
species: [150x1 nominal]
      setosa  versicolor  virginica
           50           50           50
```

# dataset.summary

---

```
meas: [150x4 double]
  min      4.3000      2      1      0.1000
  1st Q    5.1000    2.8000    1.6000    0.3000
  median   5.8000      3      4.3500    1.3000
  3rd Q    6.4000    3.3000    5.1000    1.8000
  max      7.9000    4.4000    6.9000    2.5000
```

---

Summarize the data in hospital.mat:

```
load hospital
summary(hospital)
```

Dataset array created from the data file hospital.dat.

The first column of the file ("id") is used for observation names. Other columns ("sex" and "smoke") have been converted from their original coded values into categorical and logical variables. Two sets of columns ("sys" and "dia", "trial1" through "trial4") have been combined into single variables with multivariate observations. Column headers have been replaced with more descriptive variable names. Units have been added where appropriate.

```
LastName: [100x1 cell string]
```

```
Sex: [100x1 nominal]
      Female      Male
      53          47
```

```
Age: [100x1 double, Units = Yrs]
```

```
  min      1st Q      median      3rd Q      max
  25        32        39        44        50
```

```
Weight: [100x1 double, Units = Lbs]
```

```
  min      1st Q      median      3rd Q      max
  111     130.5000    142.5000    180.5000    202
```

```
Smoker: [100x1 logical]
  true    false
   34     66
```

```
BloodPressure: [100x2 double, Units = mm Hg]
Systolic/Diastolic
  min          109          68
 1st Q        117.5000      77.5000
 median         122         81.5000
 3rd Q        127.5000      89
  max           138          99
```

```
Trials: [100x1 cell, Units = Counts]
From zero to four measurement trials performed
```

## See Also

`get` | `set` | `grpstats`

# ProbDist.Support property

---

**Purpose** Read-only structure containing information about support of ProbDist object

**Description** Support is a read-only property of the ProbDist class. Support is a structure containing information about the support of a ProbDist object. It includes the following fields:

- range
- closedbound
- iscontinuous

**Values** The values for the three fields in the structure are:

- range — A two-element vector [L, U], such that all of the probability is contained from L to U.
- closedbound — A two-element logical vector indicating whether the corresponding range endpoint is included. Possible values for each endpoint are 1 (true) or 0 (false).
- iscontinuous — A logical value indicates if the distribution takes values on the entire interval from L to U (true), or if it takes only integer values within this range (false). Possible values are 1 (true) or 0 (false).

Use this information to view and compare information about the support of distributions.

<b>Purpose</b>	Convert structure array to dataset array
<b>Syntax</b>	<pre>ds = struct2dataset(S) ds = struct2dataset(S,Name,Value)</pre>
<b>Description</b>	<p><code>ds = struct2dataset(S)</code> converts a structure array to a dataset array.</p> <p><code>ds = struct2dataset(S,Name,Value)</code> performs the conversion using additional options specified by one or more <code>Name,Value</code> pair arguments.</p>
<b>Input Arguments</b>	<p><b>S - Input structure array</b> structure array</p> <p>Input structure array to convert to a dataset array, specified as a scalar structure array with <math>N</math> fields, each with <math>M</math> rows, or a nonscalar <math>M</math>-by-1 structure array with <math>N</math> fields.</p> <p><b>Data Types</b> struct</p> <p><b>Name-Value Pair Arguments</b></p> <p>Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code>.</p> <p><b>Example:</b> <code>'ReadObsNames','myField'</code> specifies that the structure field, <code>myField</code>, contains observation names.</p> <p><b>'ReadObsNames' - Name of structure field containing observation names for dataset array</b> <code>false</code> (default)   string</p> <p>Name of structure field containing observation names for the output dataset array, specified as the comma-separated pair consisting of <code>'ReadObsNames'</code> and a string containing a field name from</p>

# struct2dataset

---

the input structure array, `S`. When you specify a field name, `struct2dataset` uses that field to create observation names, and sets `ds.Properties.DimNames` equal to `{ReadObsNames, 'Variables'}`.

For example, to specify that observation names are in the structure field, `Names`, use

**Example:** `'ReadObsNames', 'Names'`

By default, or if `ReadObsNames` is equal to `false`, `struct2dataset` does not create observation names unless you specify names using the name-value pair argument `ObsNames`.

## **'ObsNames' - Observation names for dataset array**

cell array of strings

Observation names for the output dataset array, specified as the comma-separated pair consisting of `'ObsNames'` and a cell array of strings containing observation names. The names do not need to be valid MATLAB identifiers, but they must be unique.

## **'AsScalar' - Indicator for how to treat scalar structure**

`false` | `true`

Indicator for how to treat a scalar input structure array, specified as the comma-separated pair consisting of `'AsScalar'` and either `true` or `false`. The default value is `true` if `S` is a scalar structure array, and `false` otherwise.

By default, `struct2dataset` converts a scalar structure array with  $N$  fields, each with  $M$  rows, into an  $M$ -by- $N$  dataset array.

If instead you set `AsScalar` equal to `false` for a scalar input structure array, then `struct2dataset` converts `S` to a dataset array with  $N$  observations.

## **Output Arguments**

### **ds - Output dataset array**

dataset array

Output dataset array, returned by default with  $M$  observations and  $N$  variables.



- If **S** is a scalar structure array with  $N$  fields, each with  $M$  rows, then **ds** is an  $M$ -by- $N$  dataset array.
- If **S** is a nonscalar  $M$ -by-1 structure array with  $N$  fields, then **ds** is an  $M$ -by- $N$  dataset array.
- If **S** is a scalar structure array with  $N$  fields, each with  $M$  rows, and **AsScalar** is set equal to **false**, then **ds** is a dataset array with  $N$  observations.

## Examples

### Convert Scalar Structure Array to Dataset Array

Convert a scalar structure array to a dataset array using the default options.

Create a structure array to convert.

```
S.Name = {'CLARK'; 'BROWN'; 'MARTIN'};
S.Gender = {'M'; 'F'; 'M'};
S.SystolicBP = [124; 122; 130];
S.DiastolicBP = [93; 80; 92];
S
```

S =

```
      Name: {3x1 cell}
      Gender: {3x1 cell}
  SystolicBP: [3x1 double]
  DiastolicBP: [3x1 double]
```

The scalar structure array has four fields, each with three rows.

Convert the structure array to a dataset array.

```
ds = struct2dataset(S)
```

ds =

Name	Gender	SystolicBP	DiastolicBP
'CLARK'	'M'	124	93

# struct2dataset

---

```
'BROWN'      'F'      122      80
'MARTIN'     'M'      130      92
```

The structure field names in `S` become the variable names in the output dataset array. The size of `ds` is 3-by-4.

## Convert Nonscalar Structure Array to Dataset Array

Convert a nonscalar structure array to a dataset array, using one of the structure fields for observation names.

Create a nonscalar structure array to convert.

```
S(1,1).Name = 'CLARK';
S(1,1).Gender = 'M';
S(1,1).SystolicBP = 124;
S(1,1).DiastolicBP = 93;
```

```
S(2,1).Name = 'BROWN';
S(2,1).Gender = 'F';
S(2,1).SystolicBP = 122;
S(2,1).DiastolicBP = 80;
```

```
S(3,1).Name = 'MARTIN';
S(3,1).Gender = 'M';
S(3,1).SystolicBP = 130;
S(3,1).DiastolicBP = 92;
```

```
S
```

```
S =
```

```
3x1 struct array with fields:
    Name
    Gender
    SystolicBP
    DiastolicBP
```

This is a 3-by-1 structure array with 4 fields.

Convert the structure array to a dataset array, using the Name field for observation names.

```
ds = struct2dataset(S, 'ReadObsNames', 'Name')
```

```
ds =
```

	Gender	SystolicBP	DiastolicBP
CLARK	'M'	124	93
BROWN	'F'	122	80
MARTIN	'M'	130	92

The size of `ds` is 3-by-3 because the structure field `Name` is used for observation names, and not as a dataset array variable.

```
ds.Properties.DimNames
```

```
ans =
```

```
    'Name'    'Variables'
```

```
ds.Properties.ObsNames
```

```
ans =
```

```
    'CLARK'  
    'BROWN'  
    'MARTIN'
```

## See Also

[dataset](#) | [dataset2struct](#) | [cell2dataset](#)

## Related Examples

- “Create a Dataset Array from Workspace Variables” on page 2-65
- “Create a Dataset Array from a File” on page 2-71

## Concepts

- “Dataset Arrays” on page 2-135

# surfht

---

**Purpose** Interactive contour plot

**Syntax** `surfht(Z)`  
`surfht(x,y,Z)`

**Description** `surfht(Z)` is an interactive contour plot of the matrix  $Z$  treating the values in  $Z$  as height above the plane. The  $x$ -values are the column indices of  $Z$  while the  $y$ -values are the row indices of  $Z$ .

`surfht(x,y,Z)` where  $x$  and  $y$  are vectors specify the  $x$  and  $y$ -axes on the contour plot. The length of  $x$  must match the number of columns in  $Z$ , and the length of  $y$  must match the number of rows in  $Z$ .

There are vertical and horizontal reference lines on the plot whose intersection defines the current  $x$  value and  $y$  value. You can drag these dotted white reference lines and watch the interpolated  $z$  value (at the top of the plot) update simultaneously. Alternatively, you can get a specific interpolated  $z$  value by typing the  $x$  value and  $y$  value into editable text fields on the  $x$ -axis and  $y$ -axis respectively.

**Purpose** Categories used for surrogate splits in decision tree

**Syntax** `C = surrcutcategories(T)`  
`C = surrcutcategories(T,J)`

**Description** `C = surrcutcategories(T)` returns an  $n$ -element cell array `C` of the categories used for surrogate splits in the decision tree `T`, where  $n$  is the number of nodes in the tree. For each node `K`, `C{K}` is a cell array. The length of `C{K}` is equal to the number of surrogate predictors found at this node. Every element of `C{K}` is either an empty string for a continuous surrogate predictor or a two-element cell array with categories for a categorical surrogate predictor. The first element of this two-element cell array lists categories assigned to the left child by this surrogate split and the second element of this two-element cell array lists categories assigned to the right child by this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrcutvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `C` contains an empty cell.

`C = surrcutcategories(T,J)` takes an array `J` of node numbers and returns the categories for the specified nodes.

**See Also** `classregtree` | `surrcutvar` | `cutcategories` | `surrcuttype` | `surrcutpoint`

# classregtree.surrcutflip

---

**Purpose** Numeric cutpoint assignments used for surrogate splits in decision tree

**Syntax**  
`V = surrcutflip(T)`  
`V = surrcutflip(T,J)`

**Description** `V = surrcutflip(T)` returns an  $n$ -element cell array  $V$  of the numeric cut assignments used for surrogate splits in the decision tree  $T$ , where  $n$  is the number of nodes in the tree. For each node  $K$ ,  $V\{K\}$  is a numeric vector. The length of  $V\{K\}$  is equal to the number of surrogate predictors found at this node. Every element of  $V\{K\}$  is either zero for a categorical surrogate predictor or a numeric cut assignment for a continuous surrogate predictor. The numeric cut assignment can be either  $-1$  or  $+1$ . For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z \geq C$  and the cut assignment for this surrogate split is  $-1$ . Similarly, the right child is chosen if  $Z \geq C$  and the cut assignment for this surrogate split is  $+1$ , or if  $Z < C$  and the cut assignment for this surrogate split is  $-1$ . The order of the surrogate split variables at each node is matched to the order of variables returned by `surrcutvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes,  $V$  contains an empty array.

`V = surrcutflip(T,J)` takes an array  $J$  of node numbers and returns the cutpoint assignments for the specified nodes.

**See Also** `classregtree` | `surrcutvar` | `surrcutcategories` | `surrcuttype` | `surrcutpoint` | `cutpoint`

**Purpose** Cutpoints used for surrogate splits in decision tree

**Syntax** `V = surrcutpoint(T)`  
`V = surrcutpoint(T,J)`

**Description** `V = surrcutpoint(T)` returns an  $n$ -element cell array  $V$  of the numeric values used for surrogate splits in the decision tree  $T$ , where  $n$  is the number of nodes in the tree. For each node  $K$ ,  $V\{K\}$  is a numeric vector. The length of  $V\{K\}$  is equal to the number of surrogate predictors found at this node. Every element of  $V\{K\}$  is either either NaN for a categorical surrogate predictor or a numeric cut for a continuous surrogate predictor. For every surrogate split with a numeric cut  $C$  based on a continuous predictor variable  $Z$ , the left child is chosen if  $Z < C$  and `surrcutflip` for this surrogate split is -1. Similarly, the right child is chosen if  $Z \geq C$  and `surrcutflip` for this surrogate split is +1, or if  $Z < C$  and `surrcutflip` for this surrogate split is -1. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrcutvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes,  $V$  contains an empty cell.

`V = surrcutpoint(T,J)` takes an array  $J$  of node numbers and returns the cutpoint assignments for the specified nodes.

**See Also** `classregtree` | `surrcutvar` | `surrcutcategories` | `surrcuttype` | `surrcutflip` | `cutpoint`

# classregtree.surrcuttype

---

**Purpose** Types of surrogate splits used at branches in decision tree

**Syntax**  
`C = surrcuttype(T)`  
`C = surrcuttype(T,J)`

**Description** `C = surrcuttype(T)` returns an  $n$ -element cell array `C` indicating types of surrogate splits at each node in the tree `T`, where  $n$  is the number of nodes in the tree. For each node `K`, `C{K}` is a cell array with the types of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The order of the surrogate split variables at each node is matched to the order of variables returned by `surrcutvar`. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `C` contains an empty cell. A surrogate split type can be either 'continuous' if the cut is defined in the form  $Z < V$  for a variable `Z` and cutpoint `V` or 'categorical' if the cut is defined by whether `Z` takes a value in a set of categories.

`C = surrcuttype(T,J)` takes an array `J` of node numbers and returns the cut types for the specified nodes.

**See Also** `classregtree` | `numnodes` | `cuttype` | `surrcutvar`



**Purpose** Variables used for surrogate splits in decision tree

**Syntax**  
`V = surrcutvar(T)`  
`V = surrcutvar(T,J)`  
`[V,NUM] = surrcutvar(...)`

**Description** `V = surrcutvar(T)` returns an  $n$ -element cell array `V` of the names of the variables used for surrogate splits in each node of the tree `T`, where  $n$  is the number of nodes in the tree. Every element of `V` is a cell array with the names of the surrogate split variables at this node. The variables are sorted by the predictive measure of association with the optimal predictor in the descending order, and only variables with the positive predictive measure are included. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, `V` contains an empty cell.

`V = surrcutvar(T,J)` takes an array `J` of node numbers and returns the cut types for the specified nodes.

`[V,NUM] = surrcutvar(...)` also returns a cell array `NUM` with indices for each variable.

**See Also** `classregtree` | `numnodes` | `children` | `cutvar`

# classregtree.surrvarassoc

---

**Purpose** Predictive measure of association for surrogate splits in decision tree

**Syntax**  
A = surrvarassoc(T)  
A = surrvarassoc(T,J)

**Description** A = surrvarassoc(T) returns an  $n$ -element cell array A of the predictive measures of association for surrogate splits in the decision tree T, where  $n$  is the number of nodes in the tree. For each node K, A{K} is a numeric vector. The length of A{K} is equal to the number of surrogate predictors found at this node. Every element of A{K} gives the predictive measure of association between the optimal split and this surrogate split. The order of the surrogate split variables at each node is matched to the order of variables returned by surrcutvar. The optimal-split variable at this node is not included. For non-branch (leaf) nodes, V contains an empty cell.

A = surrvarassoc(T,J) takes an array J of node numbers and returns the predictive measure of association for the specified nodes.

**See Also** classregtree | surrcutcategories | surrcuttype | surrcutflip | surrcutpoint | surrcutvar

## Purpose

Classify using support vector machine (SVM)

## Syntax

```
Group = svmclassify(SVMStruct,Sample)
Group = svmclassify(SVMStruct,Sample, 'Showplot', true)
```

## Description

`Group = svmclassify(SVMStruct,Sample)` classifies each row of the data in `Sample`, a matrix of data, using the information in a support vector machine classifier structure `SVMStruct`, created using the `svmtrain` function. Like the training data used to create `SVMStruct`, `Sample` is a matrix where each row corresponds to an observation or replicate, and each column corresponds to a feature or variable. Therefore, `Sample` must have the same number of columns as the training data. This is because the number of columns defines the number of features. `Group` indicates the group to which each row of `Sample` has been assigned.

`Group = svmclassify(SVMStruct,Sample, 'Showplot', true)` plots the `Sample` data in the figure created using the `Showplot` property with the `svmtrain` function. This plot appears only when the data is two-dimensional.

## Input Arguments

### **SVMStruct**

Support vector machine classifier structure created using the `svmtrain` function.

### **Sample**

A matrix where each row corresponds to an observation or replicate, and each column corresponds to a feature or variable. Therefore, `Sample` must have the same number of columns as the training data. This is because the number of columns defines the dimensionality of the data space.

### **Showplot**

# svmclassify

---

Describes whether to display a plot of the classification. Displays only for 2-D problems. Follow with a Boolean argument: `true` to display the plot, `false` to give no display.

## Output Arguments

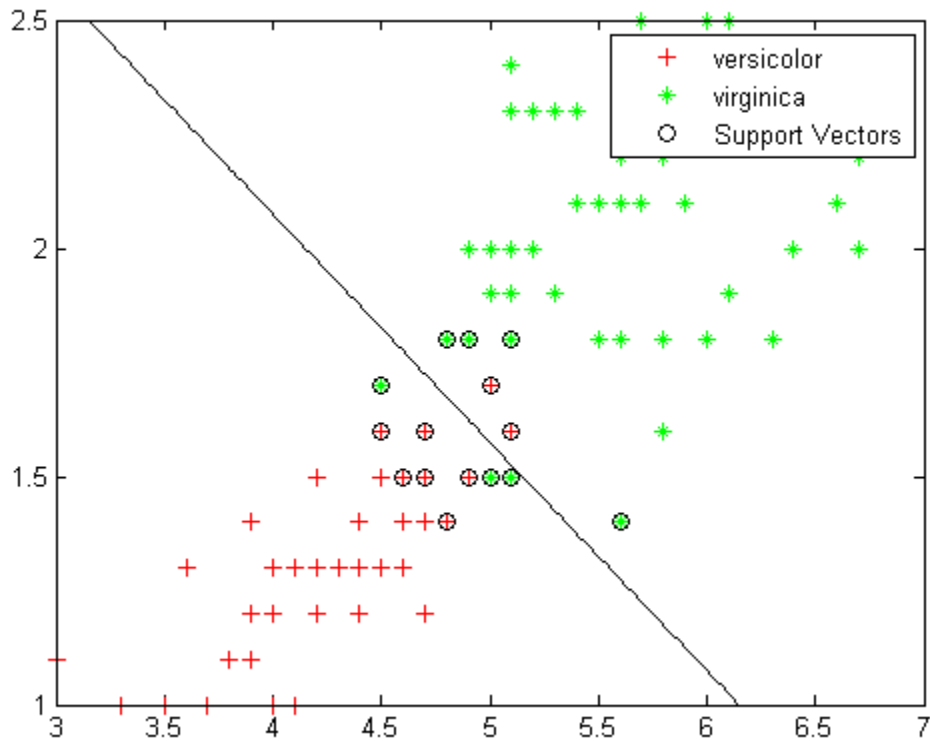
### Group

Column vector with the same number of rows as `Sample`. Each entry (row) in `Group` represents the class of the corresponding row of `Sample`.

## Examples

Find a line separating the Fisher iris data on `versicolor` and `virginica` species, according to the petal length and petal width measurements. These two species are in rows 51 and higher of the data set, and the petal length and width are the third and fourth columns.

```
load fisheriris
xdata = meas(51:end,3:4);
group = species(51:end);
svmStruct = svmtrain(xdata,group,'showplot',true);
```

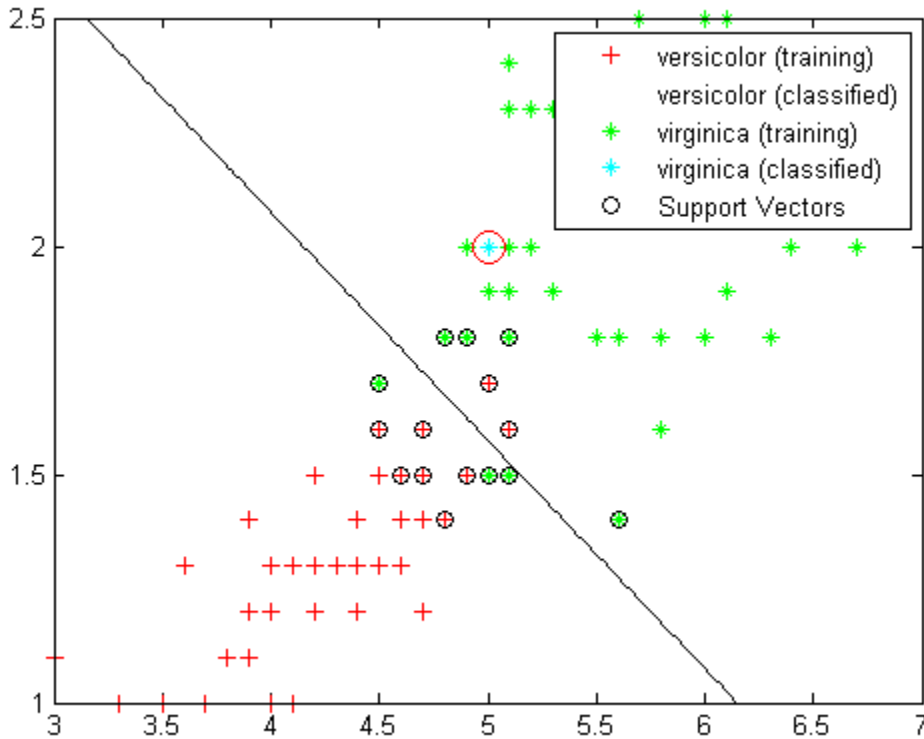


Classify a new flower with petal length 5 and petal width 2, and circle the new point:

```
species = svmclassify(svmStruct,[5 2],'showplot',true)
hold on;plot(5,2,'ro','MarkerSize',12);hold off
```

```
species =
    'virginica'
```

# svmclassify



## Algorithms

The `svmclassify` function uses results from `svmtrain` to classify vectors  $x$  according to the following equation:

$$c = \sum_i \alpha_i k(s_i, x) + b,$$

where  $s_i$  are the support vectors,  $\alpha_i$  are the weights,  $b$  is the bias, and  $k$  is a kernel function. In the case of a linear kernel,  $k$  is the dot product.

If  $c \geq 0$ , then  $x$  is classified as a member of the first group, otherwise it is classified as a member of the second group.

## References

- [1] Kecman, V., Learning and Soft Computing, MIT Press, Cambridge, MA. 2001.
- [2] Suykens, J.A.K., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J., Least Squares Support Vector Machines, World Scientific, Singapore, 2002.
- [3] Scholkopf, B., and Smola, A.J., Learning with Kernels, MIT Press, Cambridge, MA. 2002.
- [4] Cristianini, N., and Shawe-Taylor, J. (2000). An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, First Edition (Cambridge: Cambridge University Press). <http://www.support-vector.net/>

## See Also

svmtrain

## How To

- “Support Vector Machines (SVM)” on page 16-162

# svmtrain

---

## Purpose

Train support vector machine classifier

## Syntax

```
SVMStruct = svmtrain(Training,Group)
SVMStruct = svmtrain(Training,Group,Name,Value)
```

## Description

`SVMStruct = svmtrain(Training,Group)` returns a structure, `SVMStruct`, containing information about the trained support vector machine (SVM) classifier.

`SVMStruct = svmtrain(Training,Group,Name,Value)` returns a structure with additional options specified by one or more `Name,Value` pair arguments.

## Tips

- To classify new data, use the result of training, `SVMStruct`, with the `svmclassify` function.

## Input Arguments

### Training

Matrix of training data, where each row corresponds to an observation or replicate, and each column corresponds to a feature or variable. `svmtrain` treats NaNs or empty strings in `Training` as missing values and ignores the corresponding rows of `Group`.

### Group

Grouping variable, which can be a categorical, numeric, or logical vector, a cell vector of strings, or a character matrix with each row representing a class label. Each element of `Group` specifies the group of the corresponding row of `Training`. `Group` should divide `Training` into two groups. `Group` has the same number of elements as there are rows in `Training`. `svmtrain` treats each NaN, empty string, or 'undefined' in `Group` as a missing value, and ignores the corresponding row of `Training`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can



specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

**'autoscale'**

Boolean specifying whether `svmtrain` automatically centers the data points at their mean, and scales them to have unit standard deviation, before training.

**Default:** true

**'boxconstraint'**

Value of the box constraint `C` for the soft margin. `C` can be a scalar, or a vector of the same length as the training data.

If `C` is a scalar, it is automatically rescaled by  $N / (2 * N1)$  for the data points of group one and by  $N / (2 * N2)$  for the data points of group two, where  $N1$  is the number of elements in group one,  $N2$  is the number of elements in group two, and  $N = N1 + N2$ . This rescaling is done to take into account unbalanced groups, that is cases where  $N1$  and  $N2$  have very different values.

If `C` is an array, then each array element is taken as a box constraint for the data point with the same index.

**Default:** 1

**'kernelcachelimit'**

Value that specifies the size of the kernel matrix cache for the SMO training method. The algorithm keeps a matrix with up to `kernelcachelimit × kernelcachelimit` double-precision, floating-point numbers in memory.

**Default:** 5000

**'kernel\_function'**

Kernel function `svmtrain` uses to map the training data into kernel space. The default kernel function is the dot product. The kernel function can be one of the following strings or a function handle:

- `'linear'` — Linear kernel, meaning dot product.
- `'quadratic'` — Quadratic kernel.
- `'polynomial'` — Polynomial kernel (default order 3). Specify another order with the `polyorder` name-value pair.
- `'rbf'` — Gaussian Radial Basis Function kernel with a default scaling factor, `sigma`, of 1. Specify another value for `sigma` with the `rbf_sigma` name-value pair.
- `'mlp'` — Multilayer Perceptron kernel with default scale [1 1]. Specify another scale with the `mlp_params` name-value pair.
- `@kfun` — Function handle to a kernel function. A kernel function must be of the form

```
function K = kfun(U, V)
```

The returned value, `K`, is a matrix of size M-by-N, where `U` and `V` have M and N rows respectively.

If `kfun` has extra parameters, include the extra parameters via an anonymous function. For example, suppose that your kernel function is:

```
function k = kfun(u,v,p1,p2)
k = tanh(p1*(u*v')+p2);
```

Set values for `p1` and `p2`, and then use an anonymous function:

```
@(u,v) kfun(u,v,p1,p2)
```

**Default:** `'linear'`

**'kktviolationlevel'**

Value that specifies the fraction of variables allowed to violate the Karush-Kuhn-Tucker (KKT) conditions for the SMO training method. Set any value in  $[0, 1)$ . For example, if you set `kktviolationlevel` to `0.05`, then 5% of the variables are allowed to violate the KKT conditions.

---

**Tip** Set this option to a positive value to help the algorithm converge if it is fluctuating near a good solution.

---

For more information on KKT conditions, see Cristianini and Shawe-Taylor [4].

**Default:** 0

#### **'method'**

Method used to find the separating hyperplane. Options are:

- 'QP' — Quadratic programming (requires an Optimization Toolbox license). The classifier is a 2-norm soft-margin support vector machine. Give quadratic programming options with the options name-value pair, and create options with `optimset`.
- 'SMO' — Sequential Minimal Optimization. Give SMO options with the options name-value pair, and create options with `statset`.
- 'LS' — Least squares.

**Default:** SMO

#### **'mlp\_params'**

Parameters of the Multilayer Perceptron (mlp) kernel. The mlp kernel requires two parameters,  $[P1 \ P2]$ . The kernel  $K = \tanh(P1*U*V' + P2)$ , where  $P1 > 0$  and  $P2 < 0$ .

**Default:** [1 -1]

## 'options'

Options structure for training.

- When you set 'method' to 'SMO' (default), create the options structure using `statset`. Options are:

Display	String that specifies the level of information about the optimization iterations that is displayed as the algorithm runs. Choices are: <ul style="list-style-type: none"><li>▪ <code>off</code> (default) — Reports nothing.</li><li>▪ <code>iter</code> — Reports every 500 iterations.</li><li>▪ <code>final</code> — Reports only when the algorithm finishes.</li></ul>
MaxIter	Integer that specifies the maximum number of iterations of the main loop. If this limit is exceeded before the algorithm converges, then the algorithm stops and returns an error. Default is 15000.

The other name-value pairs that relate specifically to the 'SMO' method are `kernelcachelimit`, `kktviolationlevel`, and `tolkkt`.

- When you set `method` to 'QP', create the options structure using `optimset`. For details of applicable option choices, see `quadprog` options. SVM uses a convex quadratic program, so you can choose the 'interior-point-convex' `quadprog` algorithm. In limited testing, the 'interior-point-convex' algorithm was the best `quadprog` option for `svmtrain`, in both speed and memory utilization.

## 'polyorder'

Order of the polynomial kernel.

**Default:** 3

## 'rbf\_sigma'

Scaling factor (sigma) in the radial basis function kernel.

**Default:** 1

**'showplot'**

Boolean indicating whether to plot the grouped data and separating line. Creates a plot only when the data has two columns (features).

**Default:** false

**'tolkkt'**

Value that specifies the tolerance with which the Karush-Kuhn-Tucker (KKT) conditions are checked for the SMO training method. For a definition of KKT conditions, see “Karush-Kuhn-Tucker (KKT) Conditions” on page 21-3390.

**Default:** 1e-3

## Output Arguments

### **SVMStruct**

Structure containing information about the trained SVM classifier in the following fields:

- **SupportVectors** — Matrix of data points with each row corresponding to a support vector in the normalized data space. This matrix is a subset of the *Training* input data matrix, after normalization has been applied according to the 'AutoScale' argument.
- **Alpha** — Vector of weights for the support vectors. The sign of the weight is positive for support vectors belonging to the first group, and negative for the second group.
- **Bias** — Intercept of the hyperplane that separates the two groups in the normalized data space (according to the 'AutoScale' argument).
- **KernelFunction** — Handle to the function that maps the training data into kernel space.

- **KernelFunctionArgs** — Cell array of any additional arguments required by the kernel function.
- **GroupNames** — Categorical, numeric, or logical vector, a cell vector of strings, or a character matrix with each row representing a class label. Specifies the group identifiers for the support vectors. It has the same number of elements as there are rows in **SupportVectors**. Each element specifies the group to which the corresponding row in **SupportVectors** belongs.
- **SupportVectorIndices** — Vector of indices that specify the rows in **Training**, the training data, that were selected as support vectors after the data was normalized, according to the **AutoScale** argument.
- **ScaleData** — Field containing normalization factors. When 'AutoScale' is set to **false**, it is empty. When **AutoScale** is set to **true**, it is a structure containing two fields:
  - **shift** — Row vector of values. Each value is the negative of the mean across an observation in *Training*, the training data.
  - **scaleFactor** — Row vector of values. Each value is 1 divided by the standard deviation of an observation in *Training*, the training data.Both **svmtrain** and **svmclassify** apply the scaling in **ScaleData**.
- **FigureHandles** — Vector of figure handles created by **svmtrain** when using the 'Showplot' argument.

## Definitions

### Karush-Kuhn-Tucker (KKT) Conditions

The Karush-Kuhn-Tucker (KKT) conditions are analogous to the condition that the gradient must be zero at a minimum, modified to take constraints into account. The difference is that the KKT conditions hold for constrained problems. The KKT conditions use the auxiliary Lagrangian function:

$$L(x, \lambda) = f(x) + \sum \lambda_{g,i} g_i(x) + \sum \lambda_{h,i} h_i(x).$$

Here  $f(x)$  is the objective function,  $g(x)$  is a vector of constraint functions  $g(x) \leq 0$ , and  $h(x)$  is a vector of constraint functions  $h(x) = 0$ . The vector  $\lambda$ , which is the concatenation of  $\lambda_g$  and  $\lambda_h$ , is the Lagrange multiplier vector. Its length is the total number of constraints.

The KKT conditions are:

$$\begin{aligned}\nabla_x L(x, \lambda) &= 0 \\ \lambda_{g,i} g_i(x) &= 0 \quad \forall i \\ g(x) &\leq 0 \\ h(x) &= 0 \\ \lambda_{g,i} &\geq 0.\end{aligned}$$

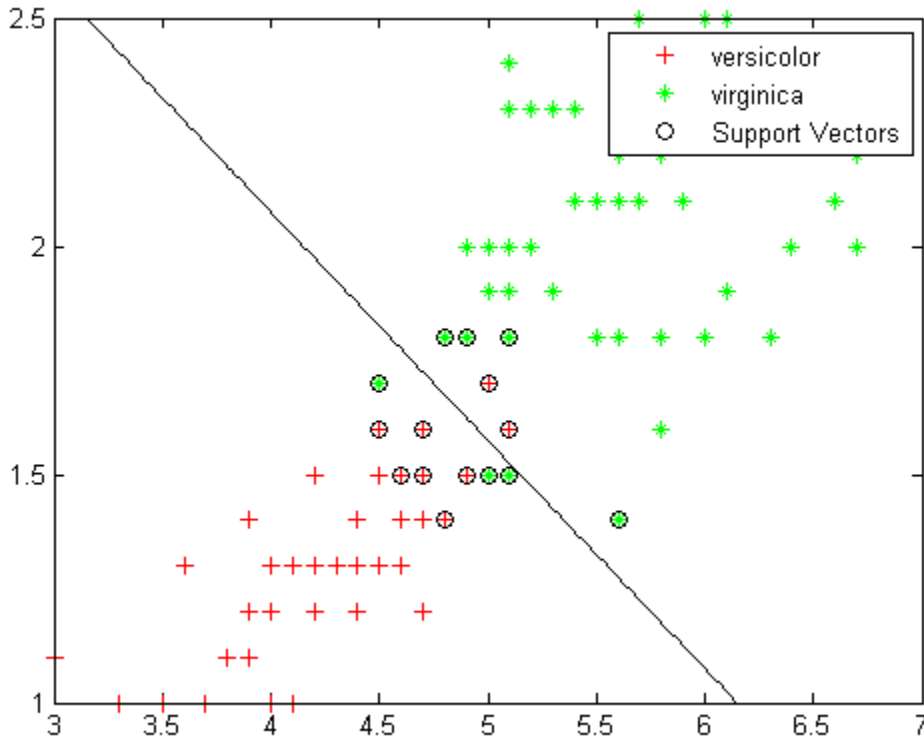
For more information, see Karush-Kuhn-Tucker conditions.

## Examples

Find a line separating the Fisher iris data on versicolor and virginica species, according to the petal length and petal width measurements. These two species are in rows 51 and higher of the data set, and the petal length and width are the third and fourth columns.

```
load fisheriris
xdata = meas(51:end,3:4);
group = species(51:end);
svmStruct = svmtrain(xdata,group,'showplot',true);
```

# svmtrain



## Algorithms

The `svmtrain` function uses an optimization method to identify support vectors  $s_i$ , weights  $a_i$ , and bias  $b$  that are used to classify vectors  $x$  according to the following equation:

$$c = \sum_i \alpha_i k(s_i, x) + b,$$



where  $k$  is a kernel function. In the case of a linear kernel,  $k$  is the dot product. If  $c \geq 0$ , then  $x$  is classified as a member of the first group, otherwise it is classified as a member of the second group.

### Memory Usage and Out of Memory Error

When you set 'Method' to 'QP', the svmtrain function operates on a data set containing  $N$  elements, and it creates an  $(N+1)$ -by- $(N+1)$  matrix to find the separating hyperplane. This matrix needs at least  $8 \cdot (n+1)^2$  bytes of contiguous memory. If this size of contiguous memory is not available, the software displays an “out of memory” error message.

When you set 'Method' to 'SMO' (default), memory consumption is controlled by the kernelcachelimit option. The SMO algorithm stores only a submatrix of the kernel matrix, limited by the size specified by the kernelcachelimit option. However, if the number of data points exceeds the size specified by the kernelcachelimit option, the SMO algorithm slows down because it has to recalculate the kernel matrix elements.

When using svmtrain on large data sets, and you run out of memory or the optimization step is very time consuming, try either of the following:

- Use a smaller number of samples and use cross-validation to test the performance of the classifier.
- Set 'Method' to 'SMO', and set the kernelcachelimit option as large as your system permits.

## References

- [1] Kecman, V., Learning and Soft Computing, MIT Press, Cambridge, MA. 2001.
- [2] Suykens, J.A.K., Van Gestel, T., De Brabanter, J., De Moor, B., and Vandewalle, J., Least Squares Support Vector Machines, World Scientific, Singapore, 2002.
- [3] Scholkopf, B., and Smola, A.J., Learning with Kernels, MIT Press, Cambridge, MA. 2002.

[4] Cristianini, N., and Shawe-Taylor, J. (2000). An Introduction to Support Vector Machines and Other Kernel-based Learning Methods, First Edition (Cambridge: Cambridge University Press). <http://www.support-vector.net/>

## See Also

`svmclassify` | `classify`

## How To

- “Support Vector Machines (SVM)” on page 16-162
- “Grouping Variables” on page 2-51

**Purpose** Convert table to dataset array

**Syntax** `ds = table2dataset(t)`

**Description** `ds = table2dataset(t)` converts a table to a dataset array.

**Input Arguments** **t** - Input table  
table

Input table to convert to a dataset array, specified as a table. Each variable in **t** becomes a variable in the output dataset array **ds**.

**Example:**

**Data Types**  
table

**Output Arguments** **ds** - Output dataset array  
dataset array

Output dataset array, returned as a dataset array containing the variables from the input table **t**.

**Examples** **Convert a Table to a Dataset Array**

Load the sample data, which contains nutritional information for 77 cereals.

```
load cereal;
```

Create a table containing the calorie, protein, fat, and name data for the first five cereals. Label the variables.

```
Calories = Calories(1:5);  
Protein = Protein(1:5);  
Fat = Fat(1:5);  
Name = Name(1:5);
```

```
cereal = table(Calories,Protein,Fat,'RowNames',Name)
```

# table2dataset

---

```
cereal =
```

	Calories	Protein	Fat
	-----	-----	---
100% Bran	70	4	1
100% Natural Bran	120	3	5
All-Bran	70	4	1
All-Bran with Extra Fiber	50	4	0
Almond Delight	110	2	2

Convert the table to a dataset array.

```
ds = table2dataset(cereal)
```

```
ds =
```

	Calories	Protein	Fat
100% Bran	70	4	1
100% Natural Bran	120	3	5
All-Bran	70	4	1
All-Bran with Extra Fiber	50	4	0
Almond Delight	110	2	2

## See Also

[table](#) | [dataset](#)

## Concepts

- “Array Dimensions”
- “Dataset Arrays” on page 2-135

**Purpose** Frequency table

**Syntax** `TABLE = tabulate(x)`  
`tabulate(x)`

**Description** `TABLE = tabulate(x)` creates a frequency table of data in vector `x`. Information in `TABLE` is arranged as follows:

- 1st column — The unique values of `x`
- 2nd column — The number of instances of each value
- 3rd column — The percentage of each value

If `x` is a numeric array, `TABLE` is a numeric matrix. If the elements of `x` are nonnegative integers, `TABLE` includes 0 counts for integers between 1 and `max(x)` that do not appear in `x`.

If `x` is a categorical variable, character array, or cell array of strings, `TABLE` is a cell array.

`tabulate(x)` with no output arguments displays the table in the command window.

**Examples**

```
tabulate([1 2 4 4 3 4])
  Value  Count  Percent
   1     1    16.67%
   2     1    16.67%
   3     1    16.67%
   4     3    50.00%
```

**See Also** `pareto`

**How To** • “Grouping Variables” on page 2-51

# tblread

---

**Purpose** Read tabular data from file

**Syntax**

```
[data,varnames,casenames] = tblread
[data,varnames,casenames] = tblread(filename)
[data,varnames,casenames] = tblread(filename,delimiter)
```

**Description** [data,varnames,casenames] = tblread displays the File Open dialog box for interactive selection of a tabular data file. The file format has variable names in the first row, case names in the first column and data starting in the (2, 2) position. Outputs are:

- data — Numeric matrix with a value for each variable-case pair
- varnames — String matrix containing the variable names in the first row of the file
- casenames — String matrix containing the names of each case in the first column of the file

[data,varnames,casenames] = tblread(filename) allows command line specification of the name of a file in the current folder, or the complete path name of any file, using the string *filename*.

[data,varnames,casenames] = tblread(filename,delimiter) reads from the file using *delimiter* as the delimiting character. Accepted values for *delimiter* are:

- ' ' or 'space'
- '\t' or 'tab'
- ',' or 'comma'
- ';' or 'semi'
- '|' or 'bar'

The default value of *delimiter* is 'space'.

**Examples**

```
[data,varnames,casenames] = tblread('sat.dat')
data =
    470  530
```

```
520 480
```

```
varnames =
```

```
Male
```

```
Female
```

```
casenames =
```

```
Verbal
```

```
Quantitative
```

**See Also**

```
tblwrite | tdfread | caseread
```

# tblwrite

---

**Purpose** Write tabular data to file

**Syntax**

```
tblwrite(data,varnames,casenames)
tblwrite(data,varnames,casenames,filename)
tblwrite(data,varnames,casenames,filename,delimiter)
```

**Description** `tblwrite(data,varnames,casenames)` displays the **File Open** dialog box for interactive specification of the tabular data output file. The file format has variable names in the first row, case names in the first column and data starting in the (2,2) position.

`varnames` is a string matrix containing the variable names. `casenames` is a string matrix containing the names of each case in the first column. `data` is a numeric matrix with a value for each variable-case pair.

`tblwrite(data,varnames,casenames,filename)` specifies a file in the current folder, or the complete path name of any file in the string `filename`.

`tblwrite(data,varnames,casenames,filename,delimiter)` writes to the file using `delimiter` as the delimiting character. The following table lists the accepted character values for `delimiter` and their equivalent string values.

Character	String
' '	'space'
'\t'	'tab'
','	'comma'
';'	'semi'
' '	'bar'

The default value of `delimiter` is 'space'.

**Examples** Continuing the example from `tblread`:

```
tblwrite(data,varnames,casenames,'sattest.dat')
```



```
type satestest.dat
      Male Female
Verbal    470  530
Quantitative 520  480
```

**See Also**

casewrite | tblread

# tcdf

---

**Purpose** Student's  $t$  cumulative distribution function

**Syntax**  
`p = tcdf(x,v)`  
`p = tcdf(x,v,'upper')`

**Description** `p = tcdf(x,v)` returns Student's  $t$  cdf at each value in  $x$  using the corresponding degrees of freedom in  $v$ .  $x$  and  $v$  can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

`p = tcdf(x,v,'upper')` returns the complement of the Student's  $t$  cdf at each value in  $x$ , using an algorithm that more accurately computes the extreme upper tail probabilities.

The  $t$  cdf is

$$p = F(x|v) = \int_{-\infty}^x \frac{\Gamma\left(\frac{v+1}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} \frac{1}{\sqrt{v\pi}} \frac{1}{\left(1 + \frac{t^2}{v}\right)^{\frac{v+1}{2}}} dt$$

The result,  $p$ , is the probability that a single observation from the  $t$  distribution with  $v$  degrees of freedom will fall in the interval  $[-\infty, x)$ .

## Examples **Compute Student's $t$ cdf**

```
mu = 1;      % Population mean
sigma = 2;   % Population standard deviation
n = 100;     % Sample size
x = normrnd(mu,sigma,n,1); % Random sample from population
xbar = mean(x); % Sample mean
s = std(x);  % Sample standard deviation

t =
    0.2489
```

```
p = 1-tcdf(t,n-1) % Probability of larger t-statistic
```

```
p =  
    0.4020
```

This probability is the same as the  $p$  value returned by a  $t$ -test of the null hypothesis that the sample comes from a normal population with mean  $\mu$ .

```
[h,ptest] = ttest(x,mu,0.05,'right')
```

```
h =  
    0  
ptest =  
    0.4020
```

**See Also**

[cdf](#) | [tpdf](#) | [tinv](#) | [tstat](#) | [trnd](#)

**How To**

- “Student’s  $t$  Distribution” on page B-124

# tdfread

---

**Purpose** Read tab-delimited file

**Syntax**

```
tdfread
tdfread(filename)
tdfread(filename,delimiter)
s = tdfread(filename,...)
```

**Description** `tdfread` displays the **File Open** dialog box for interactive selection of a data file, then reads data from the file. The file should have variable names separated by tabs in the first row, and data values separated by tabs in the remaining rows. `tdfread` creates variables in the workspace, one for each column of the file. The variable names are taken from the first row of the file. If a column of the file contains only numeric data in the second and following rows, `tdfread` creates a double variable. Otherwise, `tdfread` creates a char variable. After all values are imported, `tdfread` displays information about the imported values using the format of the `tdfread` command.

`tdfread(filename)` allows command line specification of the name of a file in the current folder, or the complete path name of any file, using the string *filename*.

`tdfread(filename,delimiter)` indicates that the character specified by *delimiter* separates columns in the file. Accepted values for *delimiter* are:

- ' ' or 'space'
- '\t' or 'tab'
- ',' or 'comma'
- ';' or 'semi'
- '|' or 'bar'

The default delimiter is 'tab'.

`s = tdfread(filename,...)` returns a scalar structure `s` whose fields each contain a variable.

**Examples**

The following displays the contents of the file `sat2.dat`:

```
type sat2.dat

Test,Gender,Score
Verbal,Male,470
Verbal,Female,530
Quantitative,Male,520
Quantitative,Female,480
```

The following creates the variables `Gender`, `Score`, and `Test` from the file `sat2.dat` and displays the contents of the MATLAB workspace:

```
tdfread('sat2.dat','')
```

Name	Size	Bytes	Class	Attributes
Gender	4x6	48	char	
Score	4x1	32	double	
Test	4x12	96	char	

**See Also**

`tblread` | `caseread`

# ClassificationDiscriminant.template

---

**Purpose** Discriminant analysis classifier template for ensemble

**Syntax**  
`t = ClassificationDiscriminant.template()`  
`t = ClassificationDiscriminant.template(Name,Value)`

**Description**  
`t = ClassificationDiscriminant.template()` returns a learner template suitable to use in the `fitensemble` function.  
`t = ClassificationDiscriminant.template(Name,Value)` creates a template with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

#### 'delta'

Threshold on linear coefficients, a nonnegative scalar. If a coefficient of `obj` has magnitude smaller than `delta`, `obj` sets this coefficient to 0, and so you can eliminate the corresponding predictor from the model. Set `delta` to a higher value to eliminate more predictors.

`delta` must be 0 for quadratic discriminant models.

**Default:** 0

#### 'discrimType'

String specifying the discriminant type. Case-insensitive. One of:

- `'linear'`
- `'quadratic'`

- 'diagLinear'
- 'diagQuadratic'
- 'pseudoLinear'
- 'pseudoQuadratic'

**Default:** 'linear'

## 'fillCoeffs'

String, either 'on' or 'off', specifying whether to populate the Coeffs property in the classifier object. Setting to 'on' can be computationally intensive, especially when cross validating.

**Default:** 'on', except 'off' when cross validating

## 'gamma'

Parameter for regularizing the correlation matrix of predictors.

- Linear discriminant — Scalar from 0 to 1.
  - If you pass a value strictly between 0 and 1, `ClassificationDiscriminant.fit` sets the discriminant type to 'Linear'.
  - If you pass 0 for gamma and 'Linear' for `DiscrimType`, and if the correlation matrix is singular, `ClassificationDiscriminant.fit` sets gamma to the minimal value required for inverting the covariance matrix.
  - If you set gamma to 1, `ClassificationDiscriminant.fit` sets the discriminant type to 'diagLinear'.
- Quadratic discriminant — Either 0 or 1.
  - If you pass 0 for gamma and 'Quadratic' for `DiscrimType`, and if one of the classes has a singular covariance matrix, `ClassificationDiscriminant.fit` errors.

# ClassificationDiscriminant.template

---

- If you set gamma to 1, `ClassificationDiscriminant.fit` sets the discriminant type to `'diagQuadratic'`.

## 'SaveMemory'

When `'on'`, `ClassificationDiscriminant.fit` does not store the full covariance matrix, but instead stores enough information to compute the matrix. The `predict` method computes the full covariance matrix for prediction, and does not store the matrix. When `'off'`, `ClassificationDiscriminant.fit` computes and stores the full covariance matrix in `obj`.

Set `SaveMemory` to `'on'` when `X` has thousands of predictors.

**Default:** `'off'`

## Output Arguments

**t**

Discriminant analysis classification template suitable to use in the `fitensemble` function. In an ensemble, `t` specifies how to create the discriminant analysis classifier.

## Examples

### Discriminant analysis template for nondefault options

Create a nondefault discriminant analysis template for use in `fitensemble`.

Create a template for pseudolinear discriminant analysis.

```
t = ClassificationDiscriminant.template('discrimType','pseudoLinear')
```

```
t =
```

```
Fit template for classification Discriminant.  
classreg.learning.modelparams.DiscriminantParams  
Package: classreg.learning.modelparams
```

```
Properties:
```

```
DiscrimType: 'pseudoLinear'
```



```
Gamma: []
Delta: []
FillCoeffs: []
SaveMemory: []
Method: 'Discriminant'
Type: 'classification'
```

Superclasses

You can use `t` for ensemble learning.

## See Also

`ClassificationDiscriminant` | `fitensemble`

# ClassificationKNN.template

---

**Purpose** *k*-nearest neighbor classifier template for ensemble

**Syntax**  
`t = ClassificationKNN.template()`  
`t = ClassificationKNN.template(Name,Value)`

**Description**  
`t = ClassificationKNN.template()` returns a learner template suitable to use in the `fitensemble` function.  
`t = ClassificationKNN.template(Name,Value)` creates a template with additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

#### 'BreakTies'

String specifying the method `predict` uses to break ties if multiple classes have the same smallest cost. By default, ties occur when multiple classes have the same number of nearest points among the `K` nearest neighbors.

- 'nearest' — Use the class with the nearest neighbor among tied groups.
- 'random' — Use a random tiebreaker among tied groups.
- 'smallest' — Use the smallest index among tied groups.

**Default:** 'smallest'

#### 'BucketSize'

Maximum number of data points in the leaf node of the *kd*-tree. This argument is meaningful only when `NSMethod` is `'kdtree'`.

**Default:** 50

## 'Cov'

Positive definite matrix, the covariance matrix when computing the Mahalanobis distance. This argument is only valid when `'Distance'` is `'mahalanobis'`.

**Default:** `nancov(X)`

## 'Distance'

String or function handle specifying the distance metric. The allowable strings depend on the `NSMethod` parameter, which you set in `ClassificationKNN.fit`, and which exists as a field in `ModelParams`.

<b>NSMethod</b>	<b>Distance Metric Names</b>
<code>exhaustive</code>	Any distance metric of <code>ExhaustiveSearcher</code>
<code>kdtree</code>	<code>'cityblock'</code> , <code>'chebychev'</code> , <code>'euclidean'</code> , or <code>'minkowski'</code>

For definitions, see “Distance Metrics” on page 16-9.

The distance metrics of `ExhaustiveSearcher`:

<b>Value</b>	<b>Description</b>
<code>'cityblock'</code>	City block distance.
<code>'chebychev'</code>	Chebychev distance (maximum coordinate difference).
<code>'correlation'</code>	One minus the sample linear correlation between observations (treated as sequences of values).

# ClassificationKNN.template

---

Value	Description
'cosine'	One minus the cosine of the included angle between observations (treated as vectors).
'euclidean'	Euclidean distance.
'hamming'	Hamming distance, percentage of coordinates that differ.
'jaccard'	One minus the Jaccard coefficient, the percentage of nonzero coordinates that differ.
'mahalanobis'	Mahalanobis distance, computed using a positive definite covariance matrix $C$ . The default value of $C$ is the sample covariance matrix of $X$ , as computed by <code>nancov(X)</code> . To specify a different value for $C$ , use the 'Cov' name-value pair.
'minkowski'	Minkowski distance. The default exponent is 2. To specify a different exponent, use the 'P' name-value pair.
'seuclidean'	Standardized Euclidean distance. Each coordinate difference between $X$ and a query point is scaled, meaning divided by a scale value $S$ . The default value of $S$ is the standard deviation computed from $X$ , $S = \text{nanstd}(X)$ . To specify another value for $S$ , use the Scale name-value pair.

Value	Description
'spearman'	One minus the sample Spearman's rank correlation between observations (treated as sequences of values).
@ <i>distfun</i>	<p>Distance function handle. <i>distfun</i> has the form</p> <pre>function D2 = DISTFUN(ZI,ZJ) % calculation of distance ...</pre> <p>where</p> <ul style="list-style-type: none"> <li>• ZI is a 1-by-N vector containing one row of X or Y.</li> <li>• ZJ is an M2-by-N matrix containing multiple rows of X or Y.</li> <li>• D2 is an M2-by-1 vector of distances, and D2(k) is the distance between observations ZI and ZJ(J, :).</li> </ul>

**Default:** 'euclidean', except the default is 'hamming' when CategoricalPredictors is 'all'

### 'DistanceWeight'

String or function handle specifying the distance weighting function.

DistanceWeight	Meaning
'equal'	No weighting
'inverse'	Weight is 1/distance

# ClassificationKNN.template

---

DistanceWeight	Meaning
'squaredinverse'	Weight is $1/\text{distance}^2$
@fcn	<i>fcn</i> is a function that accepts a matrix of nonnegative distances, and returns a matrix the same size containing nonnegative distance weights. For example, 'squaredinverse' is equivalent to $\text{@}(d)d.^{-2}$ .

**Default:** 'equal'

## 'Exponent'

Positive scalar specifying the exponent of Minkowski distance. This argument is only valid when 'Distance' is 'minkowski'.

**Default:** 2

## 'IncludeTies'

Logical value indicating whether predict includes all the neighbors whose distance values are equal to the Kth smallest distance. If IncludeTies is true, predict includes all these neighbors. Otherwise, predict uses exactly K neighbors.

**Default:** false

## 'NSMethod'

String specifying the nearest neighbor search method:

- 'kdtree' — Create and use a *kd*-tree to find nearest neighbors. 'kdtree' is valid when the distance metric is one of the following:
  - 'euclidean'
  - 'cityblock'

- 'minkowski'
- 'chebyshev'
- 'exhaustive' — Use the exhaustive search algorithm. The distance values from all points in  $X$  to each point in  $Y$  are computed to find nearest neighbors.

**Default:** 'kdtree' when  $X$  has 10 or fewer columns,  $X$  is not sparse, and the distance metric is a 'kdtree' type; otherwise, 'exhaustive'

## 'NumNeighbors'

Positive integer specifying the number of nearest neighbors in  $X$  to find for classifying each point when predicting.

**Default:** 1

## 'Scale'

Vector containing nonnegative values, with length equal to the number of columns in  $X$ . Each coordinate difference between  $X$  and a query point is scaled by the corresponding element of `Scale`. This argument is only valid when 'Distance' is 'seuclidean'.

**Default:** `nanstd(X)`

## Output Arguments

**t**

$K$ -nearest neighbor classification template suitable to use in the `fitensemble` function. In an ensemble, `t` specifies how to create the KNN classifier.

## Examples

### KNN template for nondefault options

Create a nondefault  $k$ -nearest neighbor template for use in `fitensemble`.

Create a template for 5-nearest neighbor search.

# ClassificationKNN.template

---

```
t = ClassificationKNN.template('NumNeighbors',5)
```

```
t =
```

```
Fit template for classification KNN.
```

```
classreg.learning.modelparams.KNNParams
```

```
Package: classreg.learning.modelparams
```

```
Properties:
```

```
  NumNeighbors: 5
```

```
  NSMethod: ''
```

```
  Distance: ''
```

```
  BucketSize: []
```

```
  IncludeTies: ''
```

```
  DistanceWeight: []
```

```
  BreakTies: []
```

```
  Exponent: []
```

```
    Cov: []
```

```
    Scale: []
```

```
  Method: 'KNN'
```

```
  Type: 'classification'
```

```
Superclasses
```

You can use `t` for ensemble learning.

## See Also

`ClassificationKNN` | `fitensemble`

## Related Examples

- “Random Subspace Classification” on page 16-119



**Purpose** Create classification template

**Syntax**  
`t = ClassificationTree.template`  
`t = ClassificationTree.template(Name,Value)`

**Description**  
`t = ClassificationTree.template` returns a learner template suitable to use in the `fitensemble` function.  
`t = ClassificationTree.template(Name,Value)` creates a template with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

### 'AlgorithmForCategorical'

Algorithm to find the best split on a categorical predictor for data with  $K = 3$  or more classes. The available algorithms are:

'Exact'

For a categorical predictor with  $C$  categories, consider all  $2^{C-1} - 1$  combinations.

'PullLeft'

Start with all  $C$  categories on the right branch. Consider moving each category to the left branch as it achieves the minimum impurity for the  $K$  classes among the remaining categories. Out of

# ClassificationTree.template

---

'PCA'

this sequence, choose the split that has the lowest impurity.

Compute a score for each category using the inner product between the first principal component of a weighted covariance matrix (of the centered class probability matrix) and the vector of class probabilities for that category. Sort the scores in ascending order, and consider all  $C - 1$  splits [1].

'OVAbyClass'

Start with all  $C$  categories on the right branch. For each class, order the categories based on their probability for that class. For the first class, consider moving each category to the left branch in order, recording the impurity criterion at each move. Repeat for the remaining classes. Out of this sequence, choose the split that has the minimum impurity.

**Default:** `ClassificationTree` selects the optimal subset of algorithms for each split using the known number of classes and levels of a categorical predictor. For two classes, `ClassificationTree` always performs the exact search.

**'MaxCat'**

`ClassificationTree` splits a categorical predictor using the exact search algorithm if the predictor has at most `MaxCat` levels in

the split node. Otherwise, `ClassificationTree` finds the best categorical split using one of the inexact algorithms.

Specify `MaxCat` as a numeric nonnegative scalar value. Passing a small value can lead to long computation time and memory overload.

**Default:** 10

## **'MergeLeaves'**

String that specifies whether to merge leaves after the tree is grown. Values are 'on' or 'off'.

When 'on', `ClassificationTree` merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', `ClassificationTree` does not merge leaves.

**Default:** 'off'

## **'MinLeaf'**

Each leaf has at least `MinLeaf` observations per tree leaf. If you supply both `MinParent` and `MinLeaf`, `ClassificationTree` uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** Half the number of training observations for boosting, 1 for bagging

## **'MinParent'**

Each branch node in the tree has at least `MinParent` observations. If you supply both `MinParent` and `MinLeaf`, `ClassificationTree` uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

# ClassificationTree.template

---

**Default:** Number of training observations for boosting, 2 for bagging

## **'NVarToSample'**

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all' for boosting, square root of number of predictors for bagging

## **'Prune'**

When 'on', ClassificationTree grows the classification tree and computes the optimal sequence of pruned subtrees. When 'off' ClassificationTree grows the tree without pruning.

**Default:** 'off'

## **'PruneCriterion'**

String with the pruning criterion, either 'error' or 'impurity'.

**Default:** 'error'

## **'SplitCriterion'**

Criterion for choosing a split. One of 'gdi' (Gini's diversity index), 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction (also known as cross entropy).

**Default:** 'gdi'

## **'Surrogate'**

String describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', ClassificationTree finds at most 10 surrogate splits at each branch node.
- When set to a positive integer value, ClassificationTree finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', ClassificationTree finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## Output Arguments

**t**

Classification tree template suitable to use in the `fitensemble` function. In an ensemble, `t` specifies how to grow the classification trees.

## Examples

Create a classification template with surrogate splits, and train an ensemble for the Fisher iris model with the template:

```
t = ClassificationTree.template('surrogate','on');
load fisheriris
ens = fitensemble(meas,species,'AdaBoostM2',100,t);
```

## References

[1] Coppersmith, D., S. J. Hong, and J. R. M. Hosking. "Partitioning Nominal Attributes in Decision Trees." *Data Mining and Knowledge Discovery*, Vol. 3, 1999, pp. 197–217.

## See Also

ClassificationTree | ClassificationTree.fit | fitensemble

# RegressionTree.template

---

**Purpose** Create regression template

**Syntax**  
`t = RegressionTree.template`  
`t = RegressionTree.template(Name,Value)`

**Description** `t = RegressionTree.template` returns a learner template suitable to use in the `fitensemble` function.

`t = RegressionTree.template(Name,Value)` creates a template with additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1, ...,NameN,ValueN`.

#### 'MergeLeaves'

String that specifies whether to merge leaves after the tree is grown. Values are 'on' or 'off'.

When 'on', `RegressionTree` merges leaves that originate from the same parent node, and that give a sum of risk values greater or equal to the risk associated with the parent node. When 'off', `RegressionTree` does not merge leaves.

**Default:** 'off'

#### 'MinLeaf'

Each leaf has at least `MinLeaf` observations per tree leaf. If you supply both `MinParent` and `MinLeaf`, `RegressionTree` uses the setting that gives larger leaves: `MinParent=max(MinParent,2*MinLeaf)`.

**Default:** Half the number of training observations for boosting, 5 for bagging

## **'MinParent'**

Each branch node in the tree has at least `MinParent` observations. If you supply both `MinParent` and `MinLeaf`, `RegressionTree` uses the setting that gives larger leaves:  $\text{MinParent} = \max(\text{MinParent}, 2 * \text{MinLeaf})$ .

**Default:** Number of training observations for boosting, 10 for bagging

## **'NVarToSample'**

Number of predictors to select at random for each split. Can be a positive integer or 'all', which means use all available predictors.

**Default:** 'all' for boosting, one third of the number of predictors for bagging

## **'Prune'**

When 'on', `RegressionTree` grows the regression tree and computes the optimal sequence of pruned subtrees. When 'off' `RegressionTree` grows the tree without pruning.

**Default:** 'off'

## **'Surrogate'**

String describing whether to find surrogate decision splits at each branch node. Specify as 'on', 'off', 'all', or a positive scalar value.

- When 'on', `RegressionTree` finds at most 10 surrogate splits at each branch node.

# RegressionTree.template

---

- When set to a positive integer value, `RegressionTree` finds at most the specified number of surrogate splits at each branch node.
- When set to 'all', `RegressionTree` finds all surrogate splits at each branch node. The 'all' setting can use much time and memory.

Use surrogate splits to improve the accuracy of predictions for data with missing values. The setting also enables you to compute measures of predictive association between predictors.

**Default:** 'off'

## Output Arguments

**t**

Regression tree template suitable to use in the `fitensemble` function. In an ensemble, `t` specifies how to grow the regression trees.

## Examples

Create a regression template with surrogate splits, and train an ensemble for the `carsmall` data with the template:

```
t = RegressionTree.template('surrogate','on');
load carsmall
X = [Acceleration Displacement Horsepower Weight];
ens = fitensemble(X,MPG,'LSBoost',100,t);
```

## See Also

[RegressionTree](#) | [RegressionTree.fit](#) | [fitensemble](#)



## Purpose

Error rate

## Syntax

```
cost = test(t,'resubstitution')
cost = test(t,'test',X,y)
cost = test(t,'crossvalidate',X,y)
[cost,secost,ntnodes,bestlevel] = test(...)
[...] = test(...,param1,val1,param2,val2,...)
```

## Description

`cost = test(t,'resubstitution')` computes the cost of the tree `t` using a resubstitution method. `t` is a decision tree as created by `classregtree`. The cost of the tree is the sum over all terminal nodes of the estimated probability of a node times the cost of a node. If `t` is a classification tree, the cost of a node is the sum of the misclassification costs of the observations in that node. If `t` is a regression tree, the cost of a node is the average squared error over the observations in that node. `cost` is a vector of cost values for each subtree in the optimal pruning sequence for `t`. The resubstitution cost is based on the same sample that was used to create the original tree, so it under estimates the likely cost of applying the tree to new data.

`cost = test(t,'test',X,y)` uses the matrix of predictors `X` and the response vector `y` as a test sample, applies the decision tree `t` to that sample, and returns a vector `cost` of cost values computed for the test sample. `X` and `y` should not be the same as the learning sample, that is, the sample that was used to fit the tree `t`.

`cost = test(t,'crossvalidate',X,y)` uses 10-fold cross-validation to compute the cost vector. `X` and `y` should be the learning sample, that is, the sample that was used to fit the tree `t`. The function partitions the sample into 10 subsamples, chosen randomly but with roughly equal size. For classification trees, the subsamples also have roughly the same class proportions. For each subsample, `test` fits a tree to the remaining data and uses it to predict the subsample. It pools the information from all subsamples to compute the cost for the whole sample.

`[cost,secost,ntnodes,bestlevel] = test(...)` also returns the vector `secost` containing the standard error of each cost value, the vector `ntnodes` containing the number of terminal nodes for each

# classregtree.test

---

subtree, and the scalar `bestlevel` containing the estimated best level of pruning. A `bestlevel` of 0 means no pruning. The best level is the one that produces the smallest tree that is within one standard error of the minimum-cost subtree.

[...] = `test(...,param1,val1,param2,val2,...)` specifies optional parameter name/value pairs for methods other than 'resubstitution', chosen from the following:

- 'weights' — Observation weights.
- 'nsamples' — The number of cross-validation samples (default is 10).
- 'treesize' — Either 'se' (default) to choose the smallest tree whose cost is within one standard error of the minimum cost, or 'min' to choose the minimal cost tree.

## Examples

Find the best tree for Fisher's iris data using cross-validation. Start with a large tree:

```
load fisheriris;

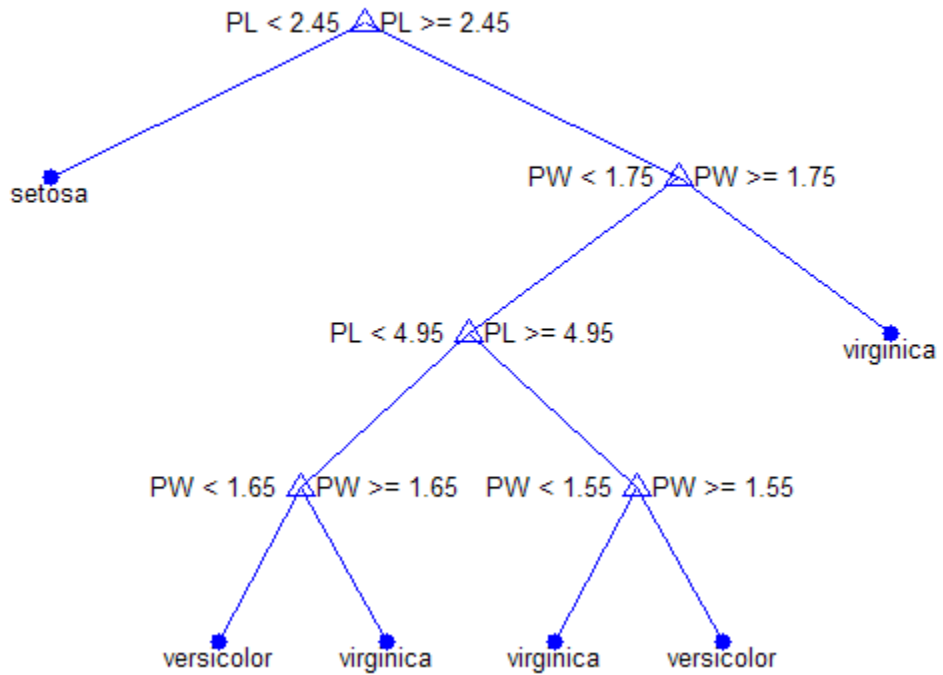
t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'},...
                'splitmin',5)

t =
Decision tree for classification
 1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
 2  class = setosa
 3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
 4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
 5  class = virginica
 6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
 7  if PW<1.55 then node 10 elseif PW>=1.55 then node 11 else virginica
 8  class = versicolor
 9  class = virginica
10  class = virginica
```

```
11 class = versicolor
```

```
view(t)
```

Click to display:  Magnification:  Pruning level:



Find the minimum-cost tree:

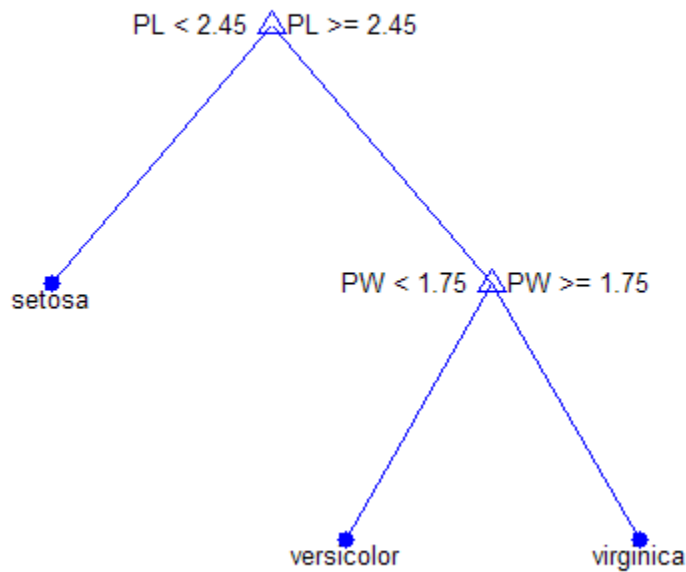
```
[c,s,n,best] = test(t,'cross',meas,species);
tmin = prune(t,'level',best)
tmin =
Decision tree for classification
```

# classregtree.test

```
1 if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2 class = setosa
3 if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4 class = versicolor
5 class = virginica
```

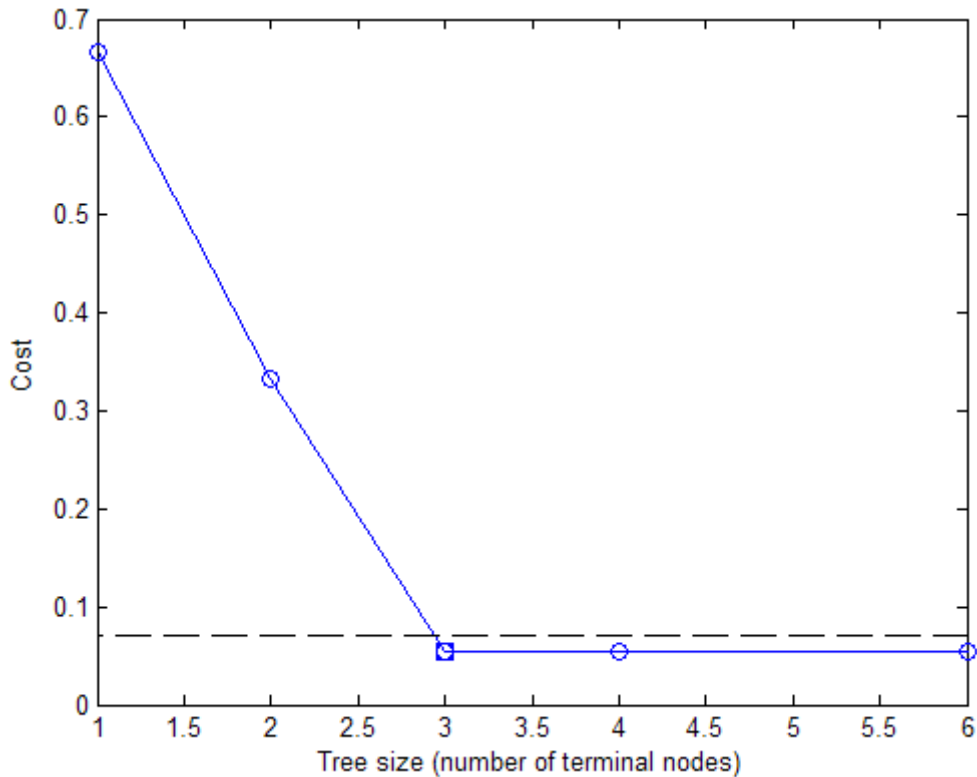
```
view(tmin)
```

Click to display:  Magnification:  Pruning level:



Plot the smallest tree within one standard error of the minimum cost tree:

```
[mincost,minloc] = min(c);  
plot(n,c,'b-o',...  
      n(best+1),c(best+1),'bs',...  
      n,(mincost+s(minloc))*ones(size(n)), 'k--')  
xlabel('Tree size (number of terminal nodes)')  
ylabel('Cost')
```



The solid line shows the estimated cost for each tree size, the dashed line marks one standard error above the minimum, and the square marks the smallest tree under the dashed line.

# classregtree.test

---

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`classregtree` | `eval` | `view` | `prune`

**Purpose** Test indices for cross-validation

**Syntax** `idx = test(c)`  
`idx = test(c,i)`

**Description** `idx = test(c)` returns the logical vector `idx` of test indices for an object `c` of the `cvpartition` class of type `'holdout'` or `'resubstitution'`.

If `c.Type` is `'holdout'`, `idx` specifies the observations in the test set.

If `c.Type` is `'resubstitution'`, `idx` specifies all observations.

`idx = test(c,i)` returns the logical vector `idx` of test indices for repetition `i` of an object `c` of the `cvpartition` class of type `'kfold'` or `'leaveout'`.

If `c.Type` is `'kfold'`, `idx` specifies the observations in the test set in fold `i`.

If `c.Type` is `'leaveout'`, `idx` specifies the observation left out at repetition `i`.

**Examples** Identify the test indices in the first fold of a partition of 10 observations for 3-fold cross-validation:

```
c = cvpartition(10,'kfold',3)
c =
K-fold cross validation partition
      N: 10
  NumTestSets: 3
   TrainSize: 7 6 7
   TestSize: 3 4 3

test(c,1)
ans =
     1
     1
     0
     0
```

# cvpartition.test

---

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

**See Also** [cvpartition](#) | [training](#)



**Purpose**

Size of each test set

**Description**

Value is a vector in partitions of type 'kfold' and 'leaveout'.  
Value is a scalar in partitions of type 'holdout' and  
'resubstitution'.

# tiedrank

---

**Purpose** Rank adjusted for ties

**Syntax** `[R,TIEADJ] = tiedrank(X)`  
`[R,TIEADJ] = tiedrank(X,1)`  
`[R,TIEADJ] = tiedrank(X,0,1)`

**Description** `[R,TIEADJ] = tiedrank(X)` computes the ranks of the values in the vector `X`. If any `X` values are tied, `tiedrank` computes their average rank. The return value `TIEADJ` is an adjustment for ties required by the nonparametric tests `signrank` and `ranksum`, and for the computation of Spearman's rank correlation.

`[R,TIEADJ] = tiedrank(X,1)` computes the ranks of the values in the vector `X`. `TIEADJ` is a vector of three adjustments for ties required in the computation of Kendall's tau. `tiedrank(X,0)` is the same as `tiedrank(X)`.

`[R,TIEADJ] = tiedrank(X,0,1)` computes the ranks from each end, so that the smallest and largest values get rank 1, the next smallest and largest get rank 2, etc. These ranks are used in the Ansari-Bradley test.

**Examples** Counting from smallest to largest, the two 20 values are 2nd and 3rd, so they both get rank 2.5 (average of 2 and 3):

```
tiedrank([10 20 30 40 20])
ans =
    1.0000    2.5000    4.0000    5.0000    2.5000
```

**See Also** `ansaribradley` | `corr` | `partialcorr` | `ranksum` | `signrank`

**Purpose** Product of categorical arrays

**Syntax** `C = times(A,B)`

**Description** `C = times(A,B)` returns a categorical array each of whose elements has the level formed from the concatenation of the levels of the corresponding elements of `A` and `B`. The set of levels of `C` is the cartesian product of the sets of levels of `A` and of `B`. The syntax `A .* B` calls `C = times(A,B)`.

**See Also** `categorical`

# tinvs

---

**Purpose** Student's *t* inverse cumulative distribution function

**Syntax** `X = tinvs(P,V)`

**Description** `X = tinvs(P,V)` computes the inverse of Student's *t* cdf using the degrees of freedom in *V* for the corresponding probabilities in *P*. *P* and *V* can be vectors, matrices, or multidimensional arrays that are the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The values in *P* must lie on the interval [0 1].

The *t* inverse function in terms of the *t* cdf is

$$x = F^{-1}(p | \nu) = \{x : F(x | \nu) = p\}$$

where

$$p = F(x | \nu) = \int_{-\infty}^x \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{t^2}{\nu}\right)^{\frac{\nu+1}{2}}} dt$$

The result, *x*, is the solution of the cdf integral with parameter *ν*, where you supply the desired probability *p*.

**Examples** What is the 99th percentile of the *t* distribution for one to six degrees of freedom?

```
percentile = tinvs(0.99,1:6)
percentile =
    31.8205    6.9646    4.5407    3.7469    3.3649    3.1427
```

**See Also** `icdf` | `tcdf` | `tpdf` | `trnd` | `tstat`

**How To** • “Student's *t* Distribution” on page B-124

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** *t* Location-Scale probability distribution object

**Description** prob.tLocationScaleDistribution is an object consisting of parameters, a model description, and sample data for a *t* location-scale probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('tLocationScale')` creates a *t* location-scale probability distribution object using the default parameter values.

`pd = makedist('tLocationScale', 'mu', mu, 'sigma', sigma, 'nu', nu)` creates a *t* location-scale probability distribution object using the specified parameter values.

## Input Arguments

### **mu - Location parameter**

0 (default) | scalar value

Location parameter for the *t* location-scale distribution, specified as a scalar value.

### **Data Types**

single | double

### **sigma - Scale parameter**

1 (default) | positive scalar value

Scale parameter for the *t* location-scale distribution, specified as a positive scalar value.

### **Data Types**

single | double

### **nu - Degrees of freedom**

# prob.tLocationScaleDistribution

---

5 (default) | positive scalar value

Degrees of freedom for the  $t$  location-scale distribution, specified as a positive scalar value.

## Data Types

single | double

## Properties

### **mu**

Location parameter of the  $t$  location-scale distribution, stored as a scalar value.

## Data Types

single | double

### **sigma**

Scale parameter of the  $t$  location-scale distribution, stored as a positive scalar value.

## Data Types

single | double

### **nu**

Degrees of freedom of the  $t$  location-scale distribution, stored as a positive scalar value.

## Data Types

single | double

### **DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

### **InputData**

Data used for distribution fitting, stored as a structure containing the following:

- **data**: Data vector used for distribution fitting.
- **cens**: Censoring vector, or empty if none.
- **freq**: Frequency vector, or empty if none.

This property is read-only.

## Data Types

single | double

## IsTruncated

Logical flag for truncated distribution, stored as a logical value. If `IsTruncated` equals 0, the distribution is not truncated. If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

# prob.tLocationScaleDistribution

---

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the ParameterNames array is not fixed. If 1, the corresponding parameter in the ParameterNames array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double



## Methods

## Inherited Methods

<code>cdf</code>	Cumulative distribution function of probability distribution object
<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
<code>iqr</code>	Interquartile range of probability distribution object
<code>median</code>	Median of probability distribution object
<code>pdf</code>	Probability density function of probability distribution object
<code>random</code>	Generate random numbers from probability distribution object
<code>truncate</code>	Truncate probability distribution object
<code>mean</code>	Mean of probability distribution object
<code>negloglik</code>	Negative loglikelihood of probability distribution object
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>proflik</code>	Profile likelihood function for probability distribution object
<code>std</code>	Standard deviation of probability distribution object
<code>var</code>	Variance of probability distribution object

# prob.tLocationScaleDistribution

## Definitions

### t Location-Scale Distribution

The  $t$  location-scale distribution is useful for modeling data distributions with heavier tails (more prone to outliers) than the normal distribution. It approaches the normal distribution as  $\nu$  approaches infinity, and smaller values of  $\nu$  yield heavier tails.

The  $t$  location-scale distribution uses the following parameters.

Parameter	Description	Support
mu	Location parameter	$-\infty < \mu < \infty$
sigma	Scale parameter	$\sigma > 0$
nu	Shape parameter	$\nu > 0$

The probability density function (pdf) is

$$f(x | \mu, \sigma, \nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sigma\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left[ \frac{\nu + \left(\frac{x-\mu}{\sigma}\right)^2}{\nu} \right]^{-\left(\frac{\nu+1}{2}\right)} ; \quad -\infty < x < \infty,$$

where  $\Gamma(\cdot)$  is the Gamma function.

## Examples

### Create a t Location-Scale Distribution Object Using Default Parameters

Create a  $t$  location scale distribution object using the default parameter values.

```
pd = makedist('tLocationScale')
```

```
pd =
```

```
tLocationScaleDistribution

t Location-Scale distribution
  mu = 0
  sigma = 1
  nu = 5
```

## Create a *t* Location-Scale Distribution Object Using Specified Parameters

Create a *t* location-scale distribution object by specifying the parameter values.

```
pd = makedist('tLocationScale','mu',-2,'sigma',1,'nu',20)
```

```
pd =
```

```
tLocationScaleDistribution

t Location-Scale distribution
  mu = -2
  sigma = 1
  nu = 20
```

Compute the interquartile range of the distribution.

```
r = iqr(pd)
```

```
r =
```

```
1.3739
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

## Concepts

- “*t* Location-Scale Distribution” on page B-126
- Class Attributes
- Property Attributes

# prob.ToolboxFittableParametricDistribution

---

**Superclasses** TruncatableDistribution

**Purpose** Toolbox-integrated fittable parametric probability distribution object

**Description** Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Methods**

<code>mean</code>	Mean of probability distribution object
<code>negloglik</code>	Negative loglikelihood of probability distribution object
<code>paramci</code>	Confidence intervals for probability distribution parameters
<code>proflik</code>	Profile likelihood function for probability distribution object
<code>std</code>	Standard deviation of probability distribution object
<code>var</code>	Variance of probability distribution object

## Inherited Methods

<code>cdf</code>	Cumulative distribution function of probability distribution object
<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
<code>iqr</code>	Interquartile range of probability distribution object

# prob.ToolboxFittableParametricDistribution

---

median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object

## See Also

`makedist` | `fitdist` | `dfittool`

## Concepts

- Class Attributes
- Property Attributes

**Purpose** Student's  $t$  probability density function

**Syntax** `Y = tpdf(X,V)`

**Description** `Y = tpdf(X,V)` computes Student's  $t$  pdf at each of the values in  $X$  using the corresponding degrees of freedom in  $V$ .  $X$  and  $V$  can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.

Student's  $t$  pdf is

$$y = f(x | \nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

## Examples

The mode of the  $t$  distribution is at  $x = 0$ . This example shows that the value of the function at the mode is an increasing function of the degrees of freedom.

```
tpdf(0,1:6)
ans =
    0.3183    0.3536    0.3676    0.3750    0.3796    0.3827
```

The  $t$  distribution converges to the standard normal distribution as the degrees of freedom approaches infinity. How good is the approximation for  $\nu = 30$ ?

```
difference = tpdf(-2.5:2.5,30) - normpdf(-2.5:2.5)
difference =
    0.0035   -0.0006   -0.0042   -0.0042   -0.0006   0.0035
```

## See Also

`pdf` | `tcdf` | `tinvs` | `tstat` | `trnd`

**How To**

- “Student’s t Distribution” on page B-124

# cvpartition.training

---

**Purpose** Training indices for cross-validation

**Syntax** `idx = training(c)`  
`idx = training(c,i)`

**Description** `idx = training(c)` returns the logical vector `idx` of training indices for an object `c` of the `cvpartition` class of type `'holdout'` or `'resubstitution'`.

If `c.Type` is `'holdout'`, `idx` specifies the observations in the training set.

If `c.Type` is `'resubstitution'`, `idx` specifies all observations.

`idx = training(c,i)` returns the logical vector `idx` of training indices for repetition `i` of an object `c` of the `cvpartition` class of type `'kfold'` or `'leaveout'`.

If `c.Type` is `'kfold'`, `idx` specifies the observations in the training set in fold `i`.

If `c.Type` is `'leaveout'`, `idx` specifies the observations left in at repetition `i`.

## Examples

Identify the training indices in the first fold of a partition of 10 observations for 3-fold cross-validation:

```
c = cvpartition(10,'kfold',3)
c =
K-fold cross validation partition
      N: 10
  NumTestSets: 3
   TrainSize: 7 6 7
   TestSize: 3 4 3

training(c,1)
ans =
     0
     0
```



1  
1  
1  
1  
1  
1  
1  
0  
1

**See Also** [cvpartition | test](#)

# cvpartition.TrainSize property

---

- Purpose** Size of each training set
- Description** Value is a vector in partitions of type 'kfold' and 'leaveout'.  
Value is a scalar in partitions of type 'holdout' and 'resubstitution'.
- See Also** type

**Purpose** Transpose categorical matrix

**Syntax** `B = transpose(A)`

**Description** `B = transpose(A)` returns the transpose of the 2-D categorical matrix `A`. `ctranspose` is identical to `transpose` for categorical arrays. The syntax `A.'` calls `transpose`.

**See Also** `ctranspose` | `permute`

# TreeBagger.TreeArgs property

---

**Purpose** Cell array of arguments for `classregtree`

**Description** The `TreeArgs` property is a cell array of arguments for the `classregtree` constructor. `TreeBagger` uses these arguments in growing new trees for the ensemble.

## Purpose

Bootstrap aggregation for ensemble of decision trees

## Description

TreeBagger bags an ensemble of decision trees for either classification or regression. Bagging stands for bootstrap aggregation. Every tree in the ensemble is grown on an independently drawn bootstrap replica of input data. Observations not included in this replica are "out of bag" for this tree. To compute prediction of an ensemble of trees for unseen data, TreeBagger takes an average of predictions from individual trees. To estimate the prediction error of the bagged ensemble, you can compute predictions for each tree on its out-of-bag observations, average these predictions over the entire ensemble for each observation and then compare the predicted out-of-bag response with the true value at this observation.

TreeBagger relies on the `ClassificationTree` and `RegressionTree` functionality for growing individual trees. In particular, `ClassificationTree` and `RegressionTree` accepts the number of features selected at random for each decision split as an optional input argument.

The `compact` method returns an object of another class, `CompactTreeBagger`, with sufficient information to make predictions using new data. This information includes the tree ensemble, variable names, and class names (for classification). `CompactTreeBagger` requires less memory than `TreeBagger`, but only `TreeBagger` has methods for growing more trees for the ensemble. Once you grow an ensemble of trees using `TreeBagger` and no longer need access to the training data, you can opt to work with the compact version of the trained ensemble from then on.

## Construction

`TreeBagger`

Create ensemble of bagged decision trees

## Methods

append	Append new trees to ensemble
compact	Compact ensemble of decision trees
error	Error (misclassification probability or MSE)
fillProximities	Proximity matrix for training data
growTrees	Train additional trees and add to ensemble
margin	Classification margin
mdsProx	Multidimensional scaling of proximity matrix
meanMargin	Mean classification margin
oobError	Out-of-bag error
oobMargin	Out-of-bag margins
oobMeanMargin	Out-of-bag mean margins
oobPredict	Ensemble predictions for out-of-bag observations
predict	Predict response

## Properties

### ClassNames

A cell array containing the class names for the response variable  $Y$ . This property is empty for regression trees.

### ComputeOOBPrediction

A logical flag specifying whether out-of-bag predictions for training observations should be computed. The default is `false`.

If this flag is true, the following properties are available:

- `OOBIndices`
- `OOBInstanceWeight`

If this flag is true, the following methods can be called:

- `oobError`
- `oobMargin`
- `oobMeanMargin`

See also `oobError`, `OOBIndices`, `OOBInstanceWeight`, `oobMargin`, `oobMeanMargin`.

### **ComputeOOBVarImp**

A logical flag specifying whether out-of-bag estimates of variable importance should be computed. The default is `false`. If this flag is true, then `ComputeOOBPrediction` is true as well.

If this flag is true, the following properties are available:

- `OOBPermutedVarDeltaError`
- `OOBPermutedVarDeltaMeanMargin`
- `OOBPermutedVarCountRaiseMargin`

### **Cost**

A matrix with misclassification costs. This property is empty for ensembles of regression trees.

### **DefaultYfit**

Default value returned by `predict` and `oobPredict`. The `DefaultYfit` property controls what predicted value is returned when no prediction is possible. For example, when `oobPredict` needs to predict for an observation that is in-bag for all trees in the ensemble.

- For classification, you can set this property to either '' or 'MostPopular'. If you choose 'MostPopular' (the default), the property value becomes the name of the most probably class in the training data. If you choose '', the in-bag observations are excluded from computation of the out-of-bag error and margin.
- For regression, you can set this property to any numeric scalar. The default value is the mean of the response for the training data. If you set this property to NaN, the in-bag observations are excluded from computation of the out-of-bag error and margin.

## **DeltaCritDecisionSplit**

A numeric array of size 1-by-*Nvars* of changes in the split criterion summed over splits on each variable, averaged across the entire ensemble of grown trees.

See also `ClassificationTree.predictorImportance` and `RegressionTree.predictorImportance`.

## **FBoot**

Fraction of observations that are randomly selected with replacement for each bootstrap replica. The size of each replica is  $Nobs \times FBoot$ , where *Nobs* is the number of observations in the training set. The default value is 1.

## **MergeLeaves**

A logical flag specifying whether decision tree leaves with the same parent are merged for splits that do not decrease the total risk. The default value is `false`.

## **Method**

Method used by trees. The possible values are 'classification' for classification ensembles, and 'regression' for regression ensembles.

## **MinLeaf**



Minimum number of observations per tree leaf. By default, `MinLeaf` is 1 for classification and 5 for regression. For decision tree training, the `MinParent` value is set equal to  $2 * \text{MinLeaf}$ .

## **NTrees**

Scalar value equal to the number of decision trees in the ensemble.

## **NVarSplit**

A numeric array of size 1-by-*Nvars*, where every element gives a number of splits on this predictor summed over all trees.

## **NVarToSample**

Number of predictor or feature variables to select at random for each decision split. By default, `NVarToSample` is equal to the square root of the total number of variables for classification, and one third of the total number of variables for regression.

## **OOBIndices**

Logical array of size *Nobs*-by-*NTrees*, where *Nobs* is the number of observations in the training data and *NTrees* is the number of trees in the ensemble. A true value for the  $(i,j)$  element indicates that observation *i* is out-of-bag for tree *j*. In other words, observation *i* was not selected for the training data used to grow tree *j*.

## **OOBInstanceWeight**

Numeric array of size *Nobs*-by-1 containing the number of trees used for computing the out-of-bag response for each observation. *Nobs* is the number of observations in the training data used to create the ensemble.

## **OOBPermutedVarCountRaiseMargin**

A numeric array of size 1-by-*Nvars* containing a measure of variable importance for each predictor variable (feature). For any variable, the measure is the difference between the number of raised margins and the number of lowered margins if the values of that variable are permuted across the out-of-bag observations.

This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

## **OOBPermutedVarDeltaError**

A numeric array of size 1-by-*Nvars* containing a measure of importance for each predictor variable (feature). For any variable, the measure is the increase in prediction error if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble.

## **OOBPermutedVarDeltaMeanMargin**

A numeric array of size 1-by-*Nvars* containing a measure of importance for each predictor variable (feature). For any variable, the measure is the decrease in the classification margin if the values of that variable are permuted across the out-of-bag observations. This measure is computed for every tree, then averaged over the entire ensemble and divided by the standard deviation over the entire ensemble. This property is empty for regression trees.

## **OutlierMeasure**

A numeric array of size *Nobs*-by-1, where *Nobs* is the number of observations in the training data, containing outlier measures for each observation.

See also `CompactTreeBagger.OutlierMeasure`.

## **Prior**

A vector with prior probabilities for classes. This property is empty for ensembles of regression trees.

## **Proximity**

A numeric matrix of size *Nobs*-by-*Nobs*, where *Nobs* is the number of observations in the training data, containing measures of

the proximity between observations. For any two observations, their proximity is defined as the fraction of trees for which these observations land on the same leaf. This is a symmetric matrix with 1s on the diagonal and off-diagonal elements ranging from 0 to 1.

See also `CompactTreeBagger.proximity`.

## **Prune**

The `Prune` property is true if decision trees are pruned and false if they are not. Pruning decision trees is not recommended for ensembles. The default value is false.

See also `ClassificationTree.prune` and `RegressionTree.prune`.

## **SampleWithReplacement**

A logical flag specifying if data are sampled for each decision tree with replacement. True if `TreeBagger` samples data with replacement and false otherwise. True by default.

## **TreeArgs**

Cell array of arguments for `ClassificationTree.fit` or `RegressionTree.fit`. These arguments are used by `TreeBagger` when growing new trees for the ensemble.

## **Trees**

A cell array of size *NTrees*-by-1 containing the trees in the ensemble.

See also `NTrees`.

## **VarAssoc**

A matrix of size *Nvars*-by-*Nvars* with predictive measures of variable association, averaged across the entire ensemble of grown trees. If you grew the ensemble setting `'surrogate'` to `'on'`, this matrix for each tree is filled with predictive measures of association averaged over the surrogate splits. If you grew

the ensemble setting 'surrogate' to 'off' (default), `VarAssoc` is diagonal.

## **VarNames**

A cell array containing the names of the predictor variables (features). `TreeBagger` takes these names from the optional 'names' parameter. The default names are 'x1', 'x2', etc.

## **W**

Numeric vector of weights of length *Nobs*, where *Nobs* is the number of observations (rows) in the training data. `TreeBagger` uses these weights for growing every decision tree in the ensemble. The default `W` is `ones(Nobs, 1)`.

## **X**

A numeric matrix of size *Nobs*-by-*Nvars*, where *Nobs* is the number of observations (rows) and *Nvars* is the number of variables (columns) in the training data. This matrix contains the predictor (or feature) values.

## **Y**

An array of true class labels for classification, or response values for regression. `Y` can be a numeric column vector, a character matrix, or a cell array of strings.

## **Copy Semantics**

Value. To learn how this affects your use of the class, see [Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation](#).

## **How To**

- “Ensemble Methods” on page 16-59
- “Classification Trees and Regression Trees” on page 16-31
- “Grouping Variables” on page 2-51

## Purpose

Create ensemble of bagged decision trees

## Syntax

```
B = TreeBagger(ntrees,X,Y)
B = TreeBagger(ntrees,X,Y,'param1',val1,'param2',val2,...)
```

## Description

`B = TreeBagger(ntrees,X,Y)` creates an ensemble `B` of `ntrees` decision trees for predicting response `Y` as a function of predictors `X`. By default `TreeBagger` builds an ensemble of classification trees. The function can build an ensemble of regression trees by setting the optional input argument `'method'` to `'regression'`.

`X` is a numeric matrix of training data. Each row represents an observation and each column represents a predictor or feature. `Y` is an array of true class labels for classification or numeric function values for regression. True class labels can be a numeric vector, character matrix, vector cell array of strings or categorical vector. `TreeBagger` converts labels to a cell array of strings for classification.

For more information on grouping variables, see “Grouping Variables” on page 2-51.

`B = TreeBagger(ntrees,X,Y,'param1',val1,'param2',val2,...)` specifies optional parameter name/value pairs:

- |                                      |   |
|--------------------------------------|---|
| <code>'FBoot'</code>                 | Fraction of input data to sample with replacement from the input data for growing each new tree. Default value is 1.  |
| <code>'SampleWithReplacement'</code> | to sample with replacement or <code>'off'</code> to sample without replacement. If you sample without replacement, you need to set <code>'FBoot'</code> to a value less than one. Default is <code>'on'</code> .                                |
| <code>'OOBPred'</code>               | <code>'on'</code> to store info on what observations are out of bag for each tree. This info can be used by <code>oobPredict</code> to compute the predicted class probabilities for each tree in the ensemble. Default is <code>'off'</code> . |

# TreeBagger

---

'OOBVarImp'	'on' to store out-of-bag estimates of feature importance in the ensemble. Default is 'off'. Specifying 'on' also sets the 'OOBPred' value to 'on'.
'Method'	Either 'classification' or 'regression'. Regression requires a numeric Y.
'NVarToSample'	Number of variables to select at random for each decision split. Default is the square root of the number of variables for classification and one third of the number of variables for regression. Valid values are 'all' or a positive integer. Setting this argument to any valid value but 'all' invokes Breiman's 'random forest' algorithm.
'NPrint'	Number of training cycles (grown trees) after which TreeBagger displays a diagnostic message showing training progress. Default is no diagnostic messages.
'MinLeaf'	Minimum number of observations per tree leaf. Default is 1 for classification and 5 for regression.
'Options'	<p>A structure that specifies options that govern the computation when growing the ensemble of decision trees. One option requests that the computation of decision trees on multiple bootstrap replicates uses multiple processors, if the Parallel Computing Toolbox is available. Two options specify the random number streams to use in selecting bootstrap replicates. You can create this argument with a call to <code>statset</code>. You can retrieve values of the individual fields with a call to <code>statget</code>. Applicable <code>statset</code> parameters are:</p> <ul style="list-style-type: none"><li>• 'UseParallel' — If true and if a parpool of the Parallel Computing Toolbox is open, compute decision trees drawn on separate</li></ul>

bootstrap replicates in parallel. If the Parallel Computing Toolbox is not installed, or a `parpool` is not open, computation occurs in serial mode. Default is `false`, or serial computation.

- `'UseSubstreams'` — If `true` select each bootstrap replicate using a separate Substream of the random number generator (aka Stream). This option is available only with `RandStream` types that support Substreams: `'mlfg6331_64'` or `'mrg32k3a'`. Default is `false`, do not use a different Substream to compute each bootstrap replicate.
- `Streams` — A `RandStream` object or cell array of such objects. If you do not specify `Streams`, `TreeBagger` uses the default stream or streams. If you choose to specify `Streams`, use a single object except in the case
  - You have an open Parallel pool
  - `UseParallel` is `true`
  - `UseSubstreams` is `false`In that case, use a cell array the same size as the Parallel pool.

In addition to the optional arguments above, this method accepts all optional `ClassificationTree.fit` and `RegressionTree.fit` arguments with the exception of `'minparent'`. Refer to the documentation for `ClassificationTree.fit` and `RegressionTree.fit` for more detail.

## Examples

```
load fisheriris
b = TreeBagger(50,meas,species,'OOBPred','on')
plot(oobError(b))
xlabel('number of grown trees')
ylabel('out-of-bag classification error')
```

# TreeBagger

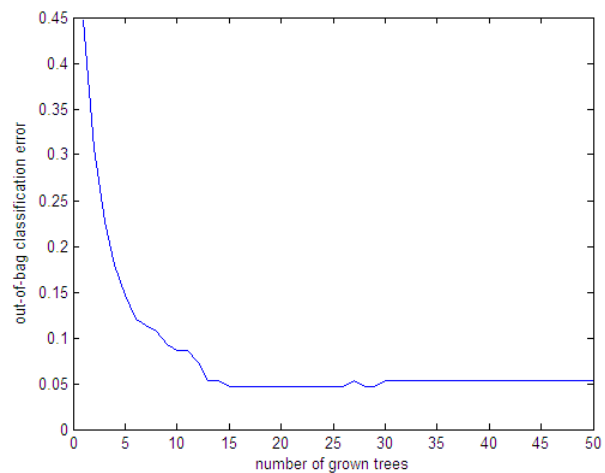
---

returns

b =

Ensemble with 50 bagged decision trees:

```
Training X: [150x4]
Training Y: [150x1]
Method: classification
Nvars: 4
NVarToSample: 2
MinLeaf: 1
FBoot: 1
SampleWithReplacement: 1
ComputeOOBPrediction: 1
ComputeOOBVarImp: 0
Proximity: []
Prune: 0
MergeLeaves: 0
TreeArgs:
ClassNames: 'setosa' 'versicolor' 'virginica'
```





## See Also

TreeBagger | ClassificationTree.fit | RegressionTree.fit | CompactTreeBagger | statset

## How To

- “Ensemble Methods” on page 16-59
- “Grouping Variables” on page 2-51

# treedisp

---

## Purpose

Plot tree

---

**Note** `treedisp` will be removed in a future release. Use `classregtree.view` instead.

---

## Syntax

```
treedisp(t)
treedisp(t,param1,val1,param2,val2,...)
```

## Description

---

**Note** This function is superseded by the `view` method of the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t` created with the `classregtree` constructor.

---

`treedisp(t)` takes as input a decision tree `t` as computed by the `treefit` function, and displays it in a figure window. Each branch in the tree is labeled with its decision rule, and each terminal node is labeled with the predicted value for that node.

For each branch node, the left child node corresponds to the points that satisfy the condition, and the right child node corresponds to the points that do not satisfy the condition.

The **Click to display** pop-up menu at the top of the figure enables you to display more information about each node, as described in the following table.

Menu Choice	Displays
Identity	The node number, whether the node is a branch or a leaf, and the rule that governs the node
Variable ranges	The range of each of the predictor variables for that node
Node statistics	Descriptive statistics for the observations falling into this node

After you select the type of information you want, click any node to display the information for that node.

The **Pruning level** button displays the number of levels that have been cut from the tree and the number of levels in the unpruned tree. For example, 1 of 6 indicates that the unpruned tree has six levels, and that one level has been cut from the tree. Use the spin button to change the pruning level.

`treedisp(t,param1,val1,param2,val2,...)` specifies optional parameter name-value pairs, listed in the following table.

Parameter	Value
'names'	A cell array of names for the predictor variables, in the order in which they appear in the X matrix from which the tree was created (see <code>treefit</code> )
'prunelevel'	Initial pruning level to display

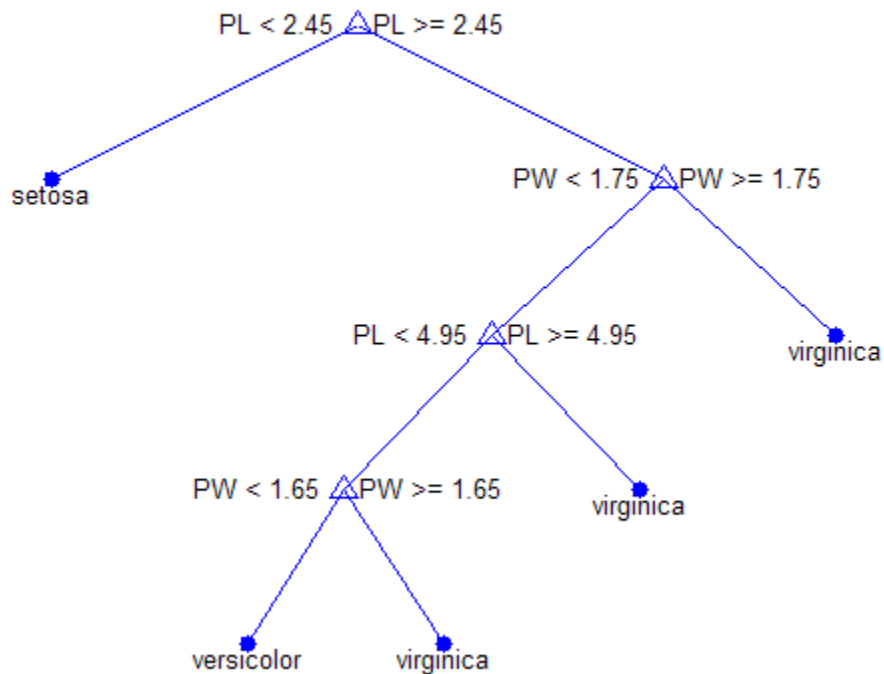
## Examples

Create and graph classification tree for Fisher's iris data. The names in this example are abbreviations for the column contents (sepal length, sepal width, petal length, and petal width).

```
load fisheriris;
t = treefit(meas,species);
treedisp(t,'names',{'SL' 'SW' 'PL' 'PW'});
```

# treedisp

Click to display:  Magnification:  Pruning level:



## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[treefit](#) | [treeprune](#) | [treetest](#)

**Purpose** Fit tree

---

**Note** treefit will be removed in a future release. Use classregtree instead.

---

**Syntax**

```
t = treefit(X,y)
t = treefit(X,y,param1,val1,param2,val2,...)
```

**Description**

---

**Note** This function is superseded by the classregtree constructor of the classregtree class and is maintained only for backwards compatibility. It returns objects *t* in the classregtree class.

---

*t* = treefit(*X*,*y*) creates a decision tree *t* for predicting response *y* as a function of predictors *X*. *X* is an *n*-by-*m* matrix of predictor values. *y* is either a vector of *n* response values (for regression), or a character array or cell array of strings containing *n* class names (for classification). Either way, *t* is a binary tree where each non-terminal node is split based on the values of a column of *X*.

*t* = treefit(*X*,*y*,*param1*,*val1*,*param2*,*val2*,...) specifies optional parameter name-value pairs. Valid parameter strings are:

The following table lists parameters available for all trees.

Parameter	Value
'catidx'	Vector of indices of the columns of <i>X</i> . treefit treats these columns as unordered categorical values.
'method'	Either 'classification' (default if <i>y</i> is text) or 'regression' (default if <i>y</i> is numeric).

Parameter	Value
'splitmin'	A number $n$ such that impure nodes must have $n$ or more observations to be split (default 10).
'prune'	'on' (default) to compute the full tree and a sequence of pruned subtrees, or 'off' for the full tree without pruning.

The following table lists parameters available for classification trees only.

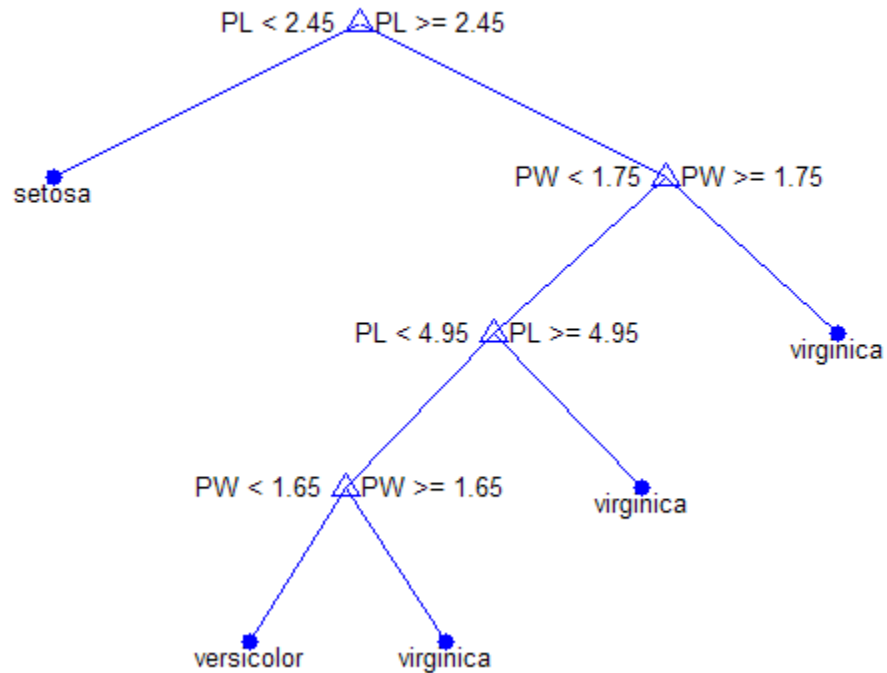
Parameter	Value
'cost'	$p$ -by- $p$ matrix $C$ , where $p$ is the number of distinct response values or class names in the input $y$ . $C(i, j)$ is the cost of classifying a point into class $j$ if its true class is $i$ . (The default has $C(i, j)=1$ if $i \neq j$ , and $C(i, j)=0$ if $i=j$ .) $C$ can also be a structure $S$ with two fields: $S.group$ containing the group names, and $S.cost$ containing a matrix of cost values.
'splitcriterion'	Criterion for choosing a split: either 'gdi' (default) for Gini's diversity index, 'twoing' for the twoing rule, or 'deviance' for maximum deviance reduction.
'priorprob'	Prior probabilities for each class, specified as a vector (one value for each distinct group name) or as a structure $S$ with two fields: $S.group$ containing the group names, and $S.prob$ containing a vector of corresponding probabilities.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;
t = treefit(meas,species);
treedisp(t,'names',{'SL' 'SW' 'PL' 'PW'});
```

Click to display:  Magnification:  Pruning level:



## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

`treedisp` | `treetest`

## How To

- “Grouping Variables” on page 2-51

# treeprune

---

## Purpose

Prune tree

---

**Note** `treeprune` will be removed in a future release. Use `classregtree.prune` instead.

---

## Syntax

```
t2 = treeprune(t1,'level',level)
t2 = treeprune(t1,'nodes',nodes)
t2 = treeprune(t1)
```

## Description

---

**Note** This function is superseded by the `prune` method of the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t1` created with the `classregtree` constructor and returns objects `t2` in the `classregtree` class.

---

`t2 = treeprune(t1,'level',level)` takes a decision tree `t1` as created by the `treefit` function, and a pruning level, and returns the decision tree `t2` pruned to that level. Setting `level` to 0 means no pruning. Trees are pruned based on an optimal pruning scheme that first prunes branches giving less improvement in error cost.

`t2 = treeprune(t1,'nodes',nodes)` prunes the nodes listed in the `nodes` vector from the tree. Any `t1` branch nodes listed in `nodes` become leaf nodes in `t2`, unless their parent nodes are also pruned. The `treedisp` function can display the node numbers for any node you select.

`t2 = treeprune(t1)` returns the decision tree `t2` that is the same as `t1`, but with the optimal pruning information added. This is useful only if you created `t1` by pruning another tree, or by using the `treefit` function with pruning set 'off'. If you plan to prune a tree multiple times, it is more efficient to create the optimal pruning sequence first.

Pruning is the process of reducing a tree by turning some branch nodes into leaf nodes, and removing the leaf nodes under the original branch.

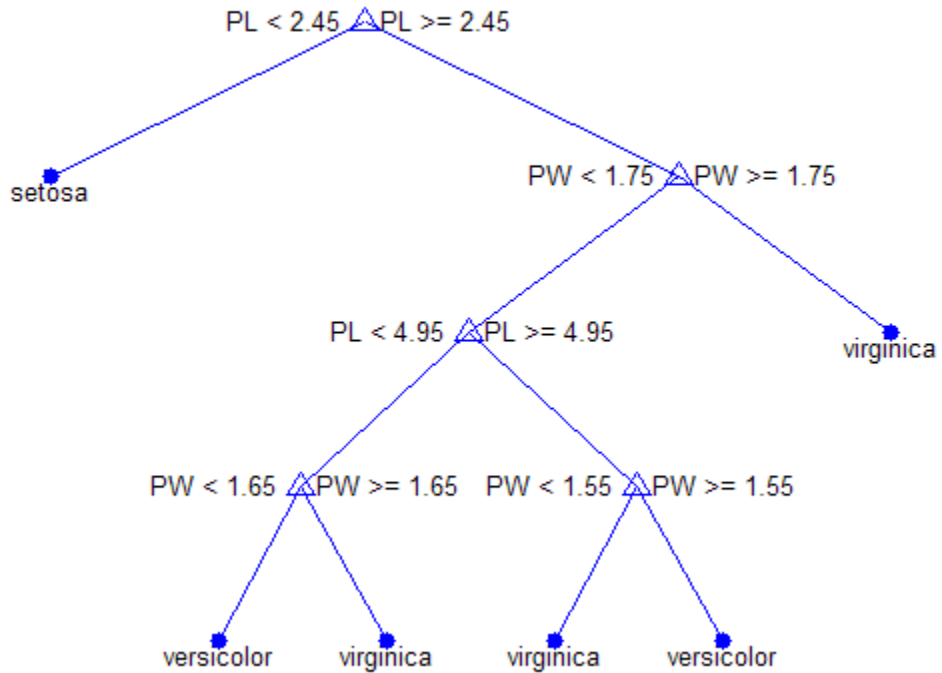


**Examples**

Display the full tree for Fisher's iris data, as well as the next largest tree from the optimal pruning sequence:

```
load fisheriris;
t1 = treefit(meas,species,'splitmin',5);
treedisp(t1,'names',{'SL' 'SW' 'PL' 'PW'});
```

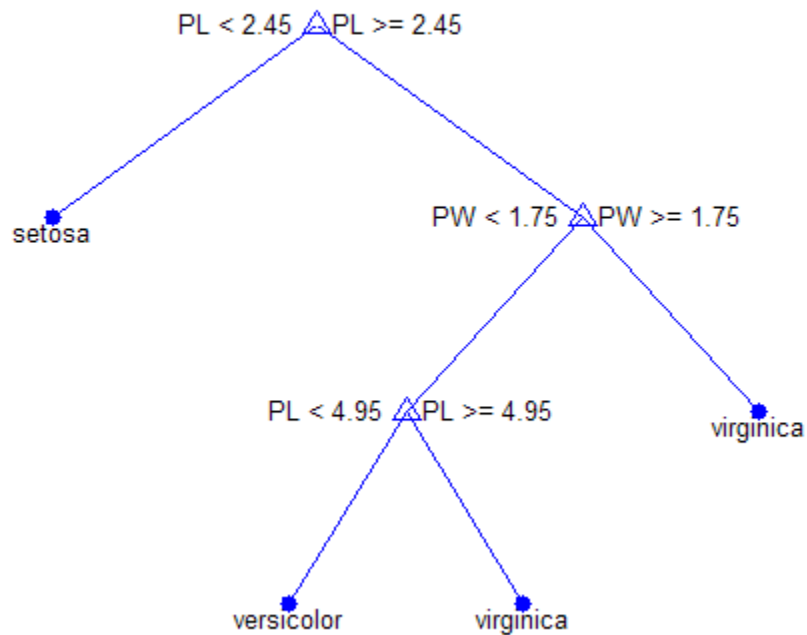
Click to display:  Magnification:  Pruning level:



```
t2 = treeprune(t1,'level',1);
treedisp(t2,'names',{'SL' 'SW' 'PL' 'PW'});
```

# treeprune

Click to display:  Magnification:  Pruning level:



## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

[treefit](#) | [treetest](#) | [treedisp](#)

<b>Purpose</b>	Decision trees in ensemble
<b>Description</b>	The Trees property is a cell array of size NTrees-by-1 containing the trees in the ensemble.
<b>See Also</b>	NTrees

# treetest

---

## Purpose

Error rate

## Syntax

```
cost = treetest(t,'resubstitution')
cost = treetest(t,'test',X,y)
cost = treetest(t,'crossvalidate',X,y)
[cost,secost,ntnodes,bestlevel] = treetest(...)
[...] = treetest(...,param1,val1,param2,val2,...)
```

---

**Note** `treetest` will be removed in a future release. Use `classregtree.test` instead.

---

## Description

---

**Note** This function is superseded by the `test` method of the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t` created with the `classregtree` constructor.

---

`cost = treetest(t,'resubstitution')` computes the cost of the tree `t` using a resubstitution method. `t` is a decision tree as created by the `treefit` function. The cost of the tree is the sum over all terminal nodes of the estimated probability of that node times the node's cost. If `t` is a classification tree, the cost of a node is the sum of the misclassification costs of the observations in that node. If `t` is a regression tree, the cost of a node is the average squared error over the observations in that node. `cost` is a vector of cost values for each subtree in the optimal pruning sequence for `t`. The resubstitution cost is based on the same sample that was used to create the original tree, so it underestimates the likely cost of applying the tree to new data.

`cost = treetest(t,'test',X,y)` uses the predictor matrix `X` and response `y` as a test sample, applies the decision tree `t` to that sample, and returns a vector `cost` of cost values computed for the test sample. `X` and `y` should not be the same as the learning sample, which is the sample that was used to fit the tree `t`.

`cost = treetest(t, 'crossvalidate', X, y)` uses 10-fold cross-validation to compute the cost vector. `X` and `y` should be the learning sample, which is the sample that was used to fit the tree `t`. The function partitions the sample into 10 subsamples, chosen randomly but with roughly equal size. For classification trees, the subsamples also have roughly the same class proportions. For each subsample, `treetest` fits a tree to the remaining data and uses it to predict the subsample. It pools the information from all subsamples to compute the cost for the whole sample.

`[cost, secost, ntnodes, bestlevel] = treetest(...)` also returns the vector `secost` containing the standard error of each cost value, the vector `ntnodes` containing number of terminal nodes for each subtree, and the scalar `bestlevel` containing the estimated best level of pruning. `bestlevel = 0` means no pruning, i.e., the full unpruned tree. The best level is the one that produces the smallest tree that is within one standard error of the minimum-cost subtree.

`[...] = treetest(..., param1, val1, param2, val2, ...)` specifies optional parameter name-value pairs chosen from the following table.

Parameter	Value
'nsamples'	The number of cross-validations samples (default is 10)
'treesize'	Either 'se' (default) to choose the smallest tree whose cost is within one standard error of the minimum cost, or 'min' to choose the minimal cost tree.

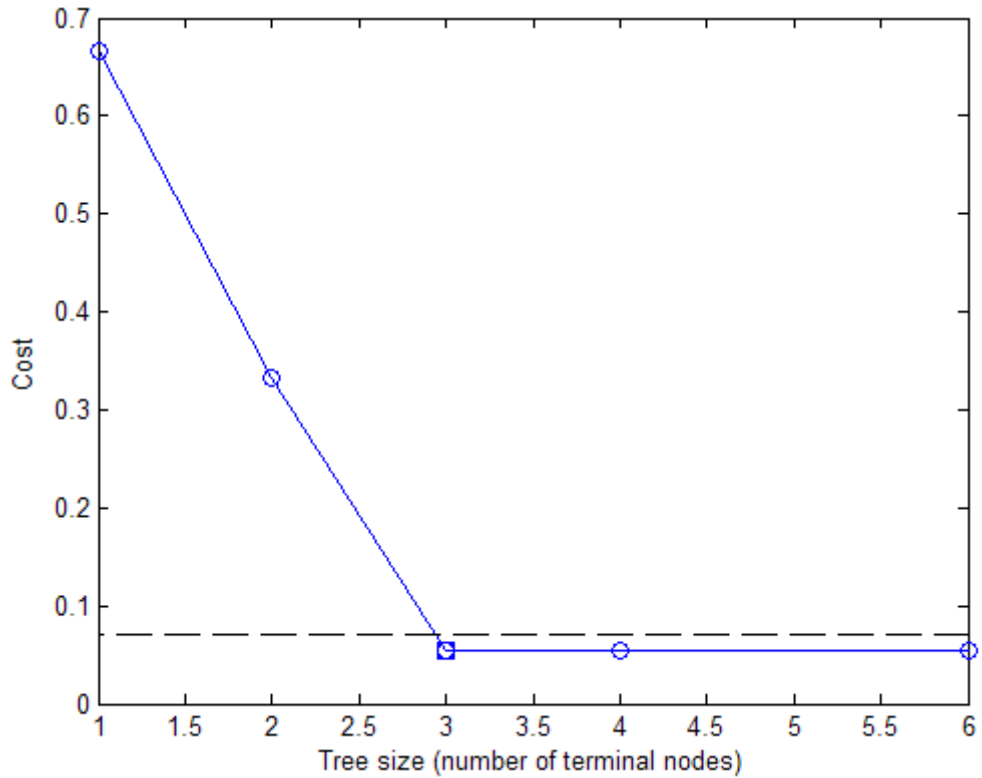
## Examples

Find the best tree for Fisher's iris data using cross-validation. The solid line shows the estimated cost for each tree size, the dashed line marks one standard error above the minimum, and the square marks the smallest tree under the dashed line.

```
% Start with a large tree.
load fisheriris;
t = treefit(meas, species', 'splitmin', 5);
```

# treetest

```
% Find the minimum-cost tree.  
[c,s,n,best] = treetest(t,'cross',meas,species);  
tmin = treeprune(t,'level',best);  
  
% Plot smallest tree within 1 std of minimum cost tree.  
[mincost,minloc] = min(c);  
plot(n,c,'b-o',...  
      n(best+1),c(best+1),'bs',...  
      n,(mincost+s(minloc))*ones(size(n)),'k--');  
xlabel('Tree size (number of terminal nodes)')  
ylabel('Cost')
```



**References**

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

**See Also**

treefit | treedisp

# treeval

---

## Purpose

Predicted responses

---

**Note** `treeval` will be removed in a future release. Use `classregtree.eval` instead.

---

## Syntax

```
yfit = treeval(t,X)
yfit = treeval(t,X,subtrees)
[yfit,node] = treeval(...)
[yfit,node,cname] = treeval(...)
```

## Description

---

**Note** This function is superseded by the `eval` method of the `classregtree` class and is maintained only for backwards compatibility. It accepts objects `t` created with the `classregtree` constructor.

---

`yfit = treeval(t,X)` takes a classification or regression tree `t` as produced by the `treefit` function and a matrix `X` of predictor values, and produces a vector `yfit` of predicted response values. For a regression tree, `yfit(i)` is the fitted response value for a point having the predictor values `X(i,:)`. For a classification tree, `yfit(i)` is the class number into which the tree would assign the point with data `X(i,:)`. To convert the number into a class name, use the third output argument, `cname` (described below).

`yfit = treeval(t,X,subtrees)` takes an additional vector `subtrees` of pruning levels, with 0 representing the full, unpruned tree. `T` must include a pruning sequence as created by the `treefit` or `prunetree` function. If `subtree` has  $k$  elements and `X` has  $n$  rows, the output `yfit` is an  $n$ -by- $k$  matrix, with the  $j$ th column containing the fitted values produced by the `subtrees(j)` subtree. `subtrees` must be sorted in ascending order.

`[yfit,node] = treeval(...)` also returns an array `node` of the same size as `yfit` containing the node number assigned to each row of `X`. The `treedisp` function can display the node numbers for any node you select.



---

[yfit,node,cname] = treeval(...) is valid only for classification trees. It returns a cell array cname containing the predicted class names.

## Examples

Find the predicted classifications for Fisher's iris data:

```
load fisheriris;
t = treefit(meas,species); % Create decision tree
sfit = treeval(t,meas); % Find assigned class numbers
sfit = t.classname(sfit); % Get class names
mean(strcmp(sfit,species)) % Proportion in correct class
ans =
    0.9800
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

treefit | treeprune | treetest

# prob.TriangularDistribution

---

**Superclasses** ParametricTruncatableDistribution

**Purpose** Triangular probability distribution object

**Description** `prob.TriangularDistribution` is an object consisting of parameters and a model description for a triangular probability distribution. Create a probability distribution object with specified parameters using `makedist`.

**Construction** `pd = makedist('Triangular')` creates a triangular probability distribution object using the default parameter values.

`pd = makedist('Triangular', 'a', a, 'b', b, 'c', c)` creates a triangular distribution object using the specified parameter values.

## Input Arguments

### **a - Lower limit**

0 (default) | scalar value

Lower limit for the triangular distribution, specified as a scalar value.

### **Data Types**

single | double

### **b - Peak location**

0.5 (default) | scalar value

Peak location for the triangular distribution, specified as a scalar value greater than or equal to `a`.

### **Data Types**

single | double

### **c - Upper limit**

1 (default) | scalar value

Upper limit for the triangular distribution, specified as a scalar value greater than or equal to `b`.

## Data Types

single | double

## Properties

### **a**

Lower limit for the triangular distribution, stored as a scalar value.

## Data Types

single | double

### **b**

Location of the peak for the triangular distribution, stored as a scalar value greater than or equal to a.

## Data Types

single | double

### **c**

Upper limit for the triangular distribution, stored as a scalar value greater than or equal to b.

## Data Types

single | double

## DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

## Data Types

char

## IsTruncated

Logical flag for truncated distribution, stored as a logical value.

If IsTruncated equals 0, the distribution is not truncated.

If IsTruncated equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

# prob.TriangularDistribution

---

## **NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## **Data Types**

single | double

## **ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## **Data Types**

char

## **ParameterNames**

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## **Data Types**

char

## **ParameterValues**

Values of distribution parameters, stored as a vector. This property is read-only.

## **Data Types**

single | double

## **Truncation**

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## **Data Types**

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object
pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Triangular Distribution

The triangular distribution is frequently used in simulations when limited sample data is available. The lower and upper limits represent the smallest and largest values, and the location of the peak represents an estimate of the mode.

The triangular distribution uses the following parameters.

# prob.TriangularDistribution

---

Parameter	Description	Support
a	Lower limit	$a \leq b$
b	Peak location	$a \leq b \leq c$
c	Upper limit	$c \geq b$

The probability density function (pdf) is

$$f(x | a, b, c) = \begin{cases} \frac{2(x-a)}{(c-a)(b-a)} & ; a \leq x \leq b \\ \frac{2(c-x)}{(c-a)(c-b)} & ; b < x \leq c \\ 0 & ; x < a, x > c \end{cases} .$$

and

$$f(x | a, b, c) = \frac{2(c-x)}{(c-a)(c-b)} ; b < x \leq c .$$

The value of the pdf is 0 when  $x < a$  or  $x > c$ .

## Examples

### Create a Triangular Distribution Object Using Default Parameters

Create a triangular distribution object using the default parameter values.

```
pd = makedist('Triangular')
```

```
pd =
```

```
TriangularDistribution
```

```
A = 0, B = 0.5, C = 1
```

## Create a Triangular Distribution Object Using Specified Parameters

Create a triangular distribution object by specifying parameter values.

```
pd = makedist('Triangular', 'a',-2,'b',1,'c',5)
```

```
pd =
```

```
    TriangularDistribution
```

```
A = -2, B = 1, C = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
    1.3333
```

**See Also** [makedist](#)

**Concepts**

- Class Attributes
- Property Attributes

# trimmean

---

**Purpose** Mean excluding outliers

**Syntax**

```
m = trimmean(X,percent)
trimmean(X,percent,dim)
m = trimmean(X,percent,flag)
m = trimmean(x,percent,flag,dim)
```

**Description** `m = trimmean(X,percent)` calculates the trimmed mean of the values in `X`. For a vector input, `m` is the mean of `X`, excluding the highest and lowest `k` data values, where  $k = n * (\text{percent} / 100) / 2$  and where `n` is the number of values in `X`. For a matrix input, `m` is a row vector containing the trimmed mean of each column of `X`. For `n`-D arrays, `trimmean` operates along the first non-singleton dimension. `percent` is a scalar between 0 and 100.

`trimmean(X,percent,dim)` takes the trimmed mean along dimension `dim` of `X`.

`m = trimmean(X,percent,flag)` controls how to trim when `k` is not an integer. `flag` can be chosen from the following:

'round'	Round <code>k</code> to the nearest integer (round to a smaller integer if <code>k</code> is a half integer). This is the default.
'floor'	Round <code>k</code> down to the next smaller integer.
'weight'	If $k = i + f$ where <code>i</code> is the integer part and <code>f</code> is the fraction, compute a weighted mean with weight $(1 - f)$ for the $(i + 1)$ th and $(n - i)$ th values, and full weight for the values between them.

`m = trimmean(x,percent,flag,dim)` takes the trimmed mean along dimension `dim` of `x`.

**Tips** The trimmed mean is a robust estimate of the location of a sample. If there are outliers in the data, the trimmed mean is a more representative estimate of the center of the body of the data than the mean. However, if the data is all from the same probability distribution,



then the trimmed mean is less efficient than the sample mean as an estimator of the location of the data.

## Examples

### Example 1

This example shows a Monte Carlo simulation of the efficiency of the 10% trimmed mean relative to the sample mean for normal data.

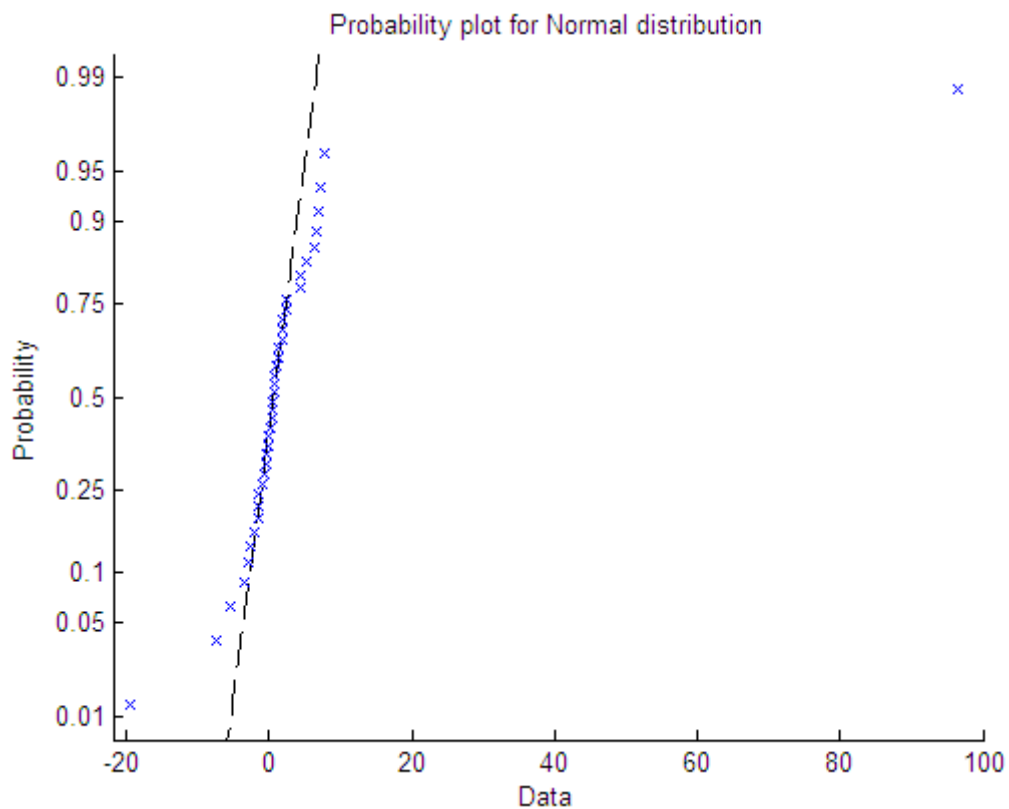
```
x = normrnd(0,1,100,100);
m = mean(x);
trim = trimmean(x,10);
sm = std(m);
strim = std(trim);
efficiency = (sm/strim).^2
efficiency =
    0.9702
```

### Example 2

Generate random data from the t distribution, which tends to have outliers:

```
rng('default') % to reproduce the plot exactly
x = trnd(1,40,1);
probplot(x)
```

# trimmean



Though the distribution is symmetric around zero, there are several outliers which will affect the mean. The trimmed mean is much closer to zero, which is much more representative of the data:

```
mean(x)
```

```
ans =  
2.7991
```

```
trimmean(x,25)
```

```
ans =
```

0.8797

**See Also**

mean | median | geomean | harmmean

# trnd

---

**Purpose** Student's *t* random numbers

**Syntax**  
R = trnd(V)  
R = trnd(V,m,n,...)  
R = trnd(V,[m,n,...])

**Description** R = trnd(V) generates random numbers from Student's *t* distribution with V degrees of freedom. V can be a vector, a matrix, or a multidimensional array. The size of R is the size of V.  
R = trnd(V,m,n,...) or R = trnd(V,[m,n,...]) generates an m-by-n-by-... array. The V parameter can be a scalar or an array of the same size as R.

**Examples**

```
noisy = trnd(ones(1,6))
noisy =
    19.7250    0.3488    0.2843    0.4034    0.4816   -2.4190

numbers = trnd(1:6,[1 6])
numbers =
   -1.9500   -0.9611   -0.9038    0.0754    0.9820    1.0115

numbers = trnd(3,2,6)
numbers =
   -0.3177  -0.0812  -0.6627    0.1905  -1.5585  -0.0433
    0.2536    0.5502    0.8646    0.8060  -0.5216    0.0891
```

**See Also** random | tpdf | tcdf | tinvs | tstat

**How To** • “Student's *t* Distribution” on page B-124

<b>Purpose</b>	Truncatable probability distribution object	
<b>Description</b>	Create a probability distribution object with specified parameter values using <code>makedist</code> . Alternatively, fit a distribution to data using <code>fitdist</code> or the Distribution Fitting app.	
<b>Methods</b>	<code>cdf</code>	Cumulative distribution function of probability distribution object
	<code>icdf</code>	Inverse cumulative distribution function of probability distribution object
	<code>iqr</code>	Interquartile range of probability distribution object
	<code>median</code>	Median of probability distribution object
	<code>pdf</code>	Probability density function of probability distribution object
	<code>random</code>	Generate random numbers from probability distribution object
	<code>truncate</code>	Truncate probability distribution object
<b>See Also</b>	<code>makedist</code>   <code>fitdist</code>   <code>dfittool</code>	
<b>Concepts</b>	<ul style="list-style-type: none"><li>• Class Attributes</li><li>• Property Attributes</li></ul>	

# prob.TruncatableDistribution.truncate

---

**Purpose** Truncate probability distribution object

**Syntax** `t = truncate(pd, lower, upper)`

**Description** `t = truncate(pd, lower, upper)` returns a probability distribution **t**, which is the probability distribution **pd** truncated to the specified interval with lower limit, **lower**, and upper limit, **upper**.

## Input Arguments

### **pd - Probability distribution**

probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, for fittable distributions, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

### **lower - Lower truncation limit**

scalar value

Lower truncation limit, specified as a scalar value.

### **Data Types**

single | double

### **upper - Upper truncation limit**

scalar value

Upper truncation limit, specified as a scalar value.

### **Data Types**

single | double

## Output Arguments

### **t - Truncated distribution**

probability distribution object

Truncated distribution, returned as a probability distribution object. The pdf of **t** is 0 outside the truncation interval. Inside the truncation interval, the pdf of **t** is equal to the pdf of **pd**, but divided by the probability assigned to that interval by **pd**.

## Examples

### Truncate a Probability Distribution

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

Truncate the distribution to have a lower limit of -2 and an upper limit of 2.

```
t = truncate(pd,-2,2)
```

```
t =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 0
```

```
sigma = 1
```

```
Truncated to the interval [-2, 2]
```

Plot the pdf of the original and truncated distributions for a visual comparison.

```
x = -3:.1:3;
```

```
figure;
```

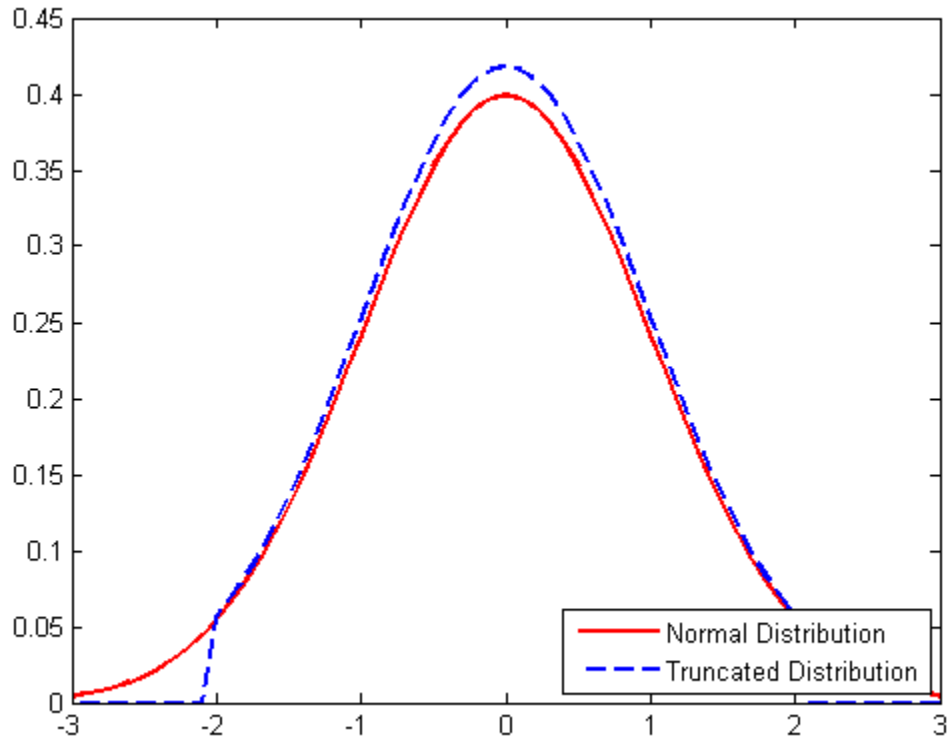
```
plot(x,pdf(pd,x),'Color','red','LineWidth',2)
```

```
hold on;
```

```
plot(x,pdf(t,x),'Color','blue','LineWidth',2)
```

```
hold off;
```

# prob.TruncatableDistribution.truncate



## Generate Random Numbers from a Truncated Distribution

Create a standard normal probability distribution object.

```
pd = makedist('Normal')
```

```
pd =
```

```
NormalDistribution
```



```
Normal distribution
  mu = 0
  sigma = 1
```

Truncate the distribution by restricting it to positive values. Set the lower limit to 0 and the upper limit to infinity.

```
t = truncate(pd,0,inf)
```

```
t =
```

```
NormalDistribution
```

```
Normal distribution
  mu = 0
  sigma = 1
```

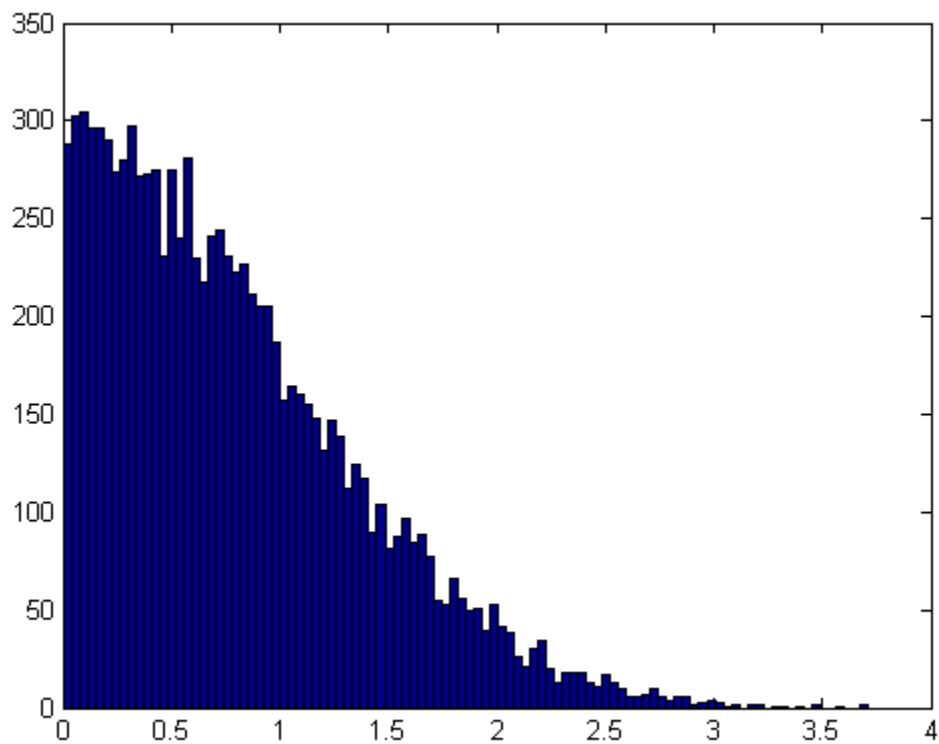
```
Truncated to the interval [0, Inf]
```

Generate random numbers from the truncated distribution and visualize with a histogram.

```
r = random(t,10000,1);
hist(r,100)
```

# prob.TruncatableDistribution.truncate

---



**See Also**      `makedist` | `fitdist` | `dfittool`

**Purpose** Student's  $t$  mean and variance

**Syntax** `[M,V] = tstat(NU)`

**Description** `[M,V] = tstat(NU)` returns the mean of and variance for Student's  $t$  distribution using the degrees of freedom in `NU`. `M` and `V` are the same size as `NU`.

The mean of the Student's  $t$  distribution with parameter  $\nu$  is zero for values of  $\nu$  greater than 1. If  $\nu$  is one, the mean does not exist. The variance for values of  $\nu$  greater than 2 is  $\nu/(\nu-2)$ .

**Examples** Find the mean of and variance for 1 to 30 degrees of freedom.

```
[m,v] = tstat(reshape(1:30,6,5))
```

```
m =
```

```
NaN  0  0  0  0
  0  0  0  0  0
  0  0  0  0  0
  0  0  0  0  0
  0  0  0  0  0
  0  0  0  0  0
```

```
v =
```

```
NaN  1.4000  1.1818  1.1176  1.0870
NaN  1.3333  1.1667  1.1111  1.0833
3.0000  1.2857  1.1538  1.1053  1.0800
2.0000  1.2500  1.1429  1.1000  1.0769
1.6667  1.2222  1.1333  1.0952  1.0741
1.5000  1.2000  1.1250  1.0909  1.0714
```

Note that the variance does not exist for one and two degrees of freedom.

**See Also** `tpdf` | `tcdf` | `tinu` | `trnd`

# ttest

---

**Purpose** One-sample and paired-sample *t*-test

**Syntax**

```
h = ttest(x)

h = ttest(x,y)
h = ttest(x,y,Name,Value)

h = ttest(x,m)
h = ttest(x,m,Name,Value)

[h,p] = ttest( ___ )
[h,p,ci,stats] = ttest( ___ )
```

**Description** `h = ttest(x)` returns a test decision for the null hypothesis that the data in `x` comes from a normal distribution with mean equal to zero and unknown variance, using the one-sample *t*-test. The alternative hypothesis is that the population distribution does not have a mean equal to zero. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = ttest(x,y)` returns a test decision for the null hypothesis that the data in `x` `y` comes from a normal distribution with mean equal to zero and unknown variance, using the paired-sample *t*-test.

`h = ttest(x,y,Name,Value)` returns a test decision for the paired-sample *t*-test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`h = ttest(x,m)` returns a test decision for the null hypothesis that the data in `x` comes from a normal distribution with mean `m` and unknown variance. The alternative hypothesis is that the mean is not `m`.

`h = ttest(x,m,Name,Value)` returns a test decision for the one-sample *t*-test with additional options specified by one or more name-value

pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = ttest( ___ )` also returns the  $p$ -value,  $p$ , of the test, using any of the input arguments from the previous syntax groups.

`[h,p,ci,stats] = ttest( ___ )` also returns the confidence interval  $ci$  for the mean of  $x$ , or of  $x - y$  for the paired  $t$ -test, and the structure  $stats$  containing information about the test statistic.

## Input Arguments

### **x - Sample data**

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. `ttest` performs a separate  $t$ -test along each column and returns a vector of results. If  $y$  sample data is specified,  $x$  and  $y$  must be the same size.

#### **Data Types**

single | double

### **y - Sample data**

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. If  $y$  sample data is specified,  $x$  and  $y$  must be the same size.

#### **Data Types**

single | double

### **m - Hypothesized population mean**

0 (default) | scalar value

Hypothesized population mean, specified as a scalar value.

#### **Data Types**

single | double

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

**Example:** 'Tail', 'right', 'Alpha', 0.01 conducts a right-tailed hypothesis test at the 1% significance level.

### **'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha', 0.01

### **Data Types**

single | double

### **'Dim' - Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim', 1 tests the column means, while 'Dim', 2 tests the row means.

**Example:** 'Dim', 2

### **Data Types**

single | double

### **'Tail' - Type of alternative hypothesis**

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'	Test the alternative hypothesis that the population mean is not $m$ .
'right'	Test the alternative hypothesis that the population mean is greater than $m$ .
'left'	Test the alternative hypothesis that the population mean is less than $m$ .

Example: 'Tail', 'right'

## Output Arguments

### **h - Hypothesis test result**

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p - p-value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **ci - Confidence interval**

vector

Confidence interval for the true population mean, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

### **stats - Test statistics**

structure

Test statistics, returned as a structure containing the following:

- `tstat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.
- `sd` — Estimated population standard deviation. For a paired *t*-test, this is the standard deviation of  $x - y$ .

## Examples

### Test for a Mean Equal to Zero

Load the sample data. Create a vector containing the third column of the stock returns data.

```
load stockreturns;  
x = stocks(:,3);
```

Test the null hypothesis that the sample data comes from a population with mean equal to zero.

```
[h,p,ci,stats] = ttest(x)
```

```
h =
```

```
1
```

```
p =
```

```
0.0106
```

```
ci =
```

```
-0.7357
```

```
-0.0997
```

```
stats =
```

```
tstat: -2.6065
```

```
df: 99
```

```
sd: 1.6027
```

The returned value `h = 1` indicates that `ttest` rejects the null hypothesis at the 5% significance level.



## Test Hypothesis at a Different Significance Level

Load the sample data. Create a vector containing the third column of the stock returns data.

```
load stockreturns;  
x = stocks(:,3);
```

Test the null hypothesis that the sample data are from a population with mean equal to zero at the 1% significance level.

```
h = ttest(x,0,'Alpha',0.01)
```

```
h =  
    0
```

The returned value  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the 1% significance level.

## Paired-Sample t-Test

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;  
x = grades(:,1);  
y = grades(:,2);
```

Test the null hypothesis that the pairwise difference between data vectors  $x$  and  $y$  has a mean equal to zero.

```
[h,p] = ttest(x,y)
```

```
h =  
    0  
  
p =  
    0.9805
```

The returned value of  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the default 5% significance level.

## Paired-Sample t-Test at a Different Significance Level

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;  
x = grades(:,1);  
y = grades(:,2);
```

Test the null hypothesis that the pairwise difference between data vectors  $x$  and  $y$  has a mean equal to zero at the 1% significance level.

```
[h,p] = ttest(x,y,'Alpha',0.01)
```

```
h =  
    0
```

```
p =  
    0.9805
```

The returned value of  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the 1% significance level.

## Test for a Hypothesized Mean

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Test the null hypothesis that sample data comes from a distribution with mean  $m = 75$ .

```
h = ttest(x,75)
```

```
h =
```

0

The returned value of  $h = 0$  indicates that `ttest` does not reject the null hypothesis at the 5% significance level.

### One-Sided Hypothesis Test

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Test the null hypothesis that the data comes from a population with mean equal to 65, against the alternative that the mean is greater than 65.

```
h = ttest(x,65,'Tail','right')
```

```
h =  
    1
```

The returned value of  $h = 1$  indicates that `ttest` rejects the null hypothesis at the 5% significance level, in favor of the alternate hypothesis that the data comes from a population with a mean greater than 65.

## Definitions

### One-Sample $t$ -Test

The one-sample  $t$ -test is a parametric test of the location parameter when the population standard deviation is unknown.

The test statistic is

$$t = \frac{\bar{x} - \mu}{s / \sqrt{n}},$$

where  $\bar{x}$  is the sample mean,  $\mu$  is the hypothesized population mean,  $s$  is the sample standard deviation, and  $n$  is the sample size. Under

the null hypothesis, the test statistic has Student's  $t$  distribution with  $n - 1$  degrees of freedom.

## **Multidimensional Array**

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

## **First Nonsingleton Dimension**

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

## **See Also**

`ztest` | `ttest2`

**Purpose**

Two-sample *t*-test

**Syntax**

```
h = ttest2(x,y)
h = ttest2(x,y,Name,Value)
[h,p] = ttest2( ___ )
[h,p,ci,stats] = ttest2( ___ )
```

**Description**

`h = ttest2(x,y)` returns a test decision for the null hypothesis that the data in vectors `x` and `y` comes from independent random samples from normal distributions with equal means and equal but unknown variances, using the two-sample *t*-test. The alternative hypothesis is that the data in `x` and `y` comes from populations with unequal means. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = ttest2(x,y,Name,Value)` returns a test decision for the two-sample *t*-test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct the test without assuming equal variances.

`[h,p] = ttest2( ___ )` also returns the *p*-value, `p`, of the test, using any of the input arguments in the previous syntaxes.

`[h,p,ci,stats] = ttest2( ___ )` also returns the confidence interval on the difference of the population means, `ci`, and the structure `stats` containing information about the test statistic.

**Input Arguments****x - Sample data**

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. `ttest2` treats NaN values as missing data and ignores them.

- If `x` and `y` are specified as vectors, they do not need to be the same length.

- If `x` and `y` are specified as matrices, they must have the same number of columns. `ttest2` performs a separate *t*-test along each column and returns a vector of results.
- If `x` and `y` are specified as multidimensional arrays, they must have the same size along all but the first nonsingleton dimension.

## Data Types

single | double

## `y` - Sample data

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. `ttest2` treats NaN values as missing data and ignores them.

- If `x` and `y` are specified as vectors, they do not need to be the same length.
- If `x` and `y` are specified as matrices, they must have the same number of columns. `ttest2` performs a separate *t*-test along each column and returns a vector of results.
- If `x` and `y` are specified as multidimensional arrays, they must have the same size along all but the first nonsingleton dimension. `ttest2` works along the first nonsingleton dimension.

## Data Types

single | double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'Tail','right','Alpha',0.01,'Vartype','unequal'` specifies a right-tailed test at the 1% significance level, and does not assume that `x` and `y` have equal population variances.

**'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

**Data Types**

single | double

**'Dim' - Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim',1 tests the column means, while 'Dim',2 tests the row means.

**Example:** 'Dim',2

**Data Types**

single | double

**'Tail' - Type of alternative hypothesis**

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

- |         |  |
|---------|--|
| 'both'  | Test the alternative hypothesis that the population means are not equal.                                     |
| 'right' | Test the alternative hypothesis that the population mean of $x$ is greater than the population mean of $y$ . |
| 'left'  | Test the alternative hypothesis that the population mean of $x$ is less than the population mean of $y$ .    |

**Example:** 'Tail','right'

## 'Vartype' - Variance type

'equal' (default) | 'unequal'

Variance type, specified as the comma-separated pair consisting of 'Vartype' and one of the following.

- |           |  |
|-----------|--|
| 'equal'   | Conduct test using the assumption that $x$ and $y$ are from normal distributions with unknown but equal variances.   |
| 'unequal' | Conduct test using the assumption that $x$ and $y$ are from normal distributions with unknown and unequal variances. This is called the Behrens-Fisher problem. <code>ttest2</code> uses Satterthwaite's approximation for the effective degrees of freedom. |

Vartype must be a single string, even when  $x$  is a matrix or a multidimensional array.

**Example:** 'Vartype', 'unequal'

## Output Arguments

### **h** - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### **p** - **p-value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.



**ci - Confidence interval**

vector

Confidence interval for the difference in population means of  $x$  and  $y$ , returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

**stats - Test statistics**

structure

Test statistics for the two-sample  $t$ -test, returned as a structure containing the following:

- `tstat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.
- `sd` — Pooled estimate of the population standard deviation (for the equal variance case) or a vector containing the unpooled estimates of the population standard deviations (for the unequal variance case).

**Examples****Test for Equal Means**

Load the data set. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;
x = grades(:,1);
y = grades(:,2);
```

Test the null hypothesis that the two data samples are from populations with equal means.

```
[h,p,ci,stats] = ttest2(x,y)
```

```
h =
    0
```

```
p =
    0.9867
```

```
ci =  
    -1.9438  
     1.9771  
  
stats =  
    tstat: 0.0167  
     df: 238  
     sd: 7.7084
```

The returned value of  $h = 0$  indicates that `ttest2` does not reject the null hypothesis at the default 5% significance level.

### Test for Equal Means Without Assuming Equal Variances

Load the data set. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;  
x = grades(:,1);  
y = grades(:,2);
```

Test the null hypothesis that the two data vectors are from populations with equal means, without assuming that the populations also have equal variances.

```
[h,p] = ttest2(x,y,'Vartype','unequal')
```

```
h =  
    0  
  
p =  
    0.9867
```

The returned value of  $h = 0$  indicates that `ttest2` does not reject the null hypothesis at the default 5% significance level even if equal variances are not assumed.

## Definitions

### Two-Sample *t*-test

The two-sample *t*-test is a parametric test that compares the location parameter of two independent data samples.

The test statistic is

$$t = \frac{\bar{x} - \bar{y}}{\sqrt{\frac{s_x^2}{n} + \frac{s_y^2}{m}}},$$

where  $\bar{x}$  and  $\bar{y}$  are the sample means,  $s_x$  and  $s_y$  are the sample standard deviations, and  $n$  and  $m$  are the sample sizes.

In the case where it is assumed that the two data samples are from populations with equal variances, the test statistic under the null hypothesis has Student's *t* distribution with  $n + m - 2$  degrees of freedom, and the sample standard deviations are replaced by the pooled standard deviation

$$s = \sqrt{\frac{(n-1)s_x^2 + (m-1)s_y^2}{n+m-2}}.$$

In the case where it is not assumed that the two data samples are from populations with equal variances, the test statistic under the null hypothesis has an approximate Student's *t* distribution with a number of degrees of freedom given by Satterthwaite's approximation. This test is sometimes called Welch's *t*-test.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

## ttest2

---

### See Also

ttest | ztest

<b>Purpose</b>	Tree type
<b>Syntax</b>	<code>ttype = type(t)</code>
<b>Description</b>	<code>ttype = type(t)</code> returns the type of the tree <code>t</code> . <code>ttype</code> is 'regression' for regression trees and 'classification' for classification trees.

## Examples

Create a classification tree for Fisher's iris data:

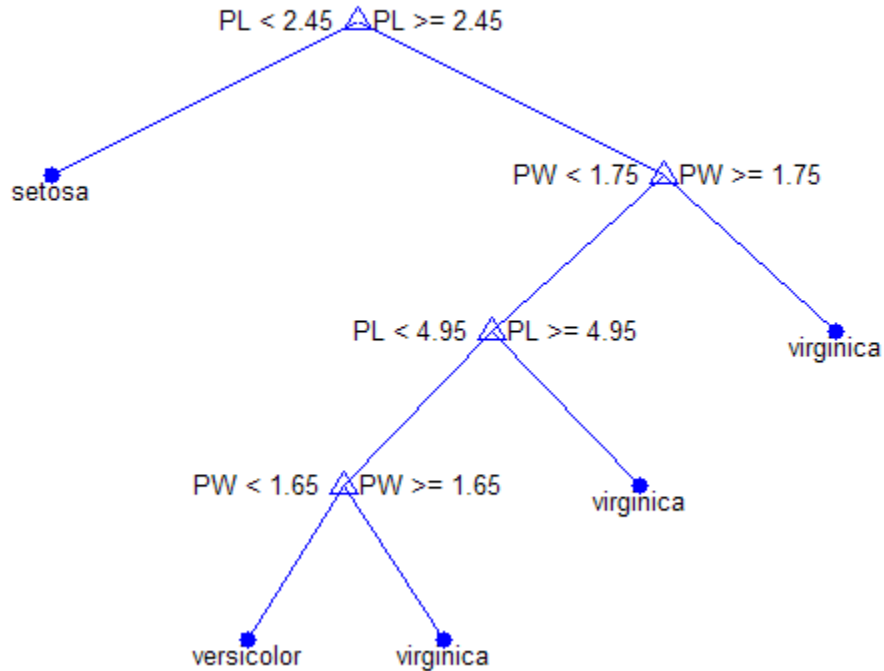
```
load fisheriris;

t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
6  if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7  class = virginica
8  class = versicolor
9  class = virginica

view(t)
```

# classregtree.type

Click to display:  Magnification:  Pruning level:



```
ttype = type(t)  
ttype =  
classification
```

## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

## See Also

classregtree

<b>Purpose</b>	Type of partition
<b>Description</b>	The type of validation partition. It is 'kfold', 'holdout', 'leaveout', or 'resubstitution'.
<b>See Also</b>	trainsize

## grandset.Type property

---

**Purpose** Name of sequence on which point set P is based

**Description** P.Type returns a string that contains the name of the sequence on which the point set P is based, for example 'Sobol'. You cannot change the Type property for a point set.



<b>Purpose</b>	Convert categorical array to unsigned 8-bit integers
<b>Syntax</b>	<code>B = uint8(A)</code>
<b>Description</b>	<code>B = uint8(A)</code> converts the categorical array <code>A</code> to unsigned 8-bit integers. Each element of <code>B</code> contains the internal categorical level code for the corresponding element of <code>A</code> . Undefined elements of <code>A</code> are assigned the value 0 in <code>B</code> . If <code>A</code> contains more than <code>intmax('uint8')</code> levels, the internal codes will saturate to <code>intmax('uint8')</code> when cast to <code>int8</code> .
<b>See Also</b>	<code>double</code>   <code>int8</code>

# categorical.uint16

---

**Purpose** Convert categorical array to unsigned 16-bit integers

**Syntax** `B = uint16(A)`

**Description** `B = uint16(A)` converts the categorical array `A` to unsigned 16-bit integers. Each element of `B` contains the internal categorical level code for the corresponding element of `A`.

Undefined elements of `A` are assigned the value 0 in `B`.

**See Also** `double` | `int16`

<b>Purpose</b>	Convert categorical array to unsigned 32-bit integers
<b>Syntax</b>	<code>B = uint32(A)</code>
<b>Description</b>	<p><code>B = uint32(A)</code> converts the categorical array <code>A</code> to unsigned 32-bit integers. Each element of <code>B</code> contains the internal categorical level code for the corresponding element of <code>A</code>.</p> <p>Undefined elements of <code>A</code> are assigned the value 0 in <code>B</code>.</p>
<b>See Also</b>	<code>double</code>   <code>int32</code>

# categorical.uint64

---

**Purpose** Convert categorical array to unsigned 64-bit integers

**Syntax** `B = uint64(A)`

**Description** `B = uint64(A)` converts the categorical array `A` to unsigned 64-bit integers. Each element of `B` contains the internal categorical level code for the corresponding element of `A`.

Undefined elements of `A` are assigned the value 0 in `B`.

**See Also** `double` | `int64`

# categorical.undeflabel property

---

**Purpose** Text label for undefined levels

**Description** Text label for undefined levels. Constant property with value '<undefined>'.

# prob.UniformDistribution

---

**Superclasses** ParametricTruncatableDistribution

**Purpose** Uniform probability distribution object

**Description** `prob.UniformDistribution` is an object consisting of parameters and a model description for a uniform probability distribution. Create a probability distribution object with specified parameters using `makedist`.

**Construction** `pd = makedist('Uniform')` creates a uniform probability distribution object using the default parameter values.

`pd = makedist('Uniform', 'Lower', lower, 'Upper', upper)` creates a uniform distribution object using the specified parameter values.

## Input Arguments

### **lower - Lower parameter**

0 (default) | scalar value

Lower limit for the uniform distribution, specified as a scalar value.

### **Data Types**

single | double

### **upper - Upper parameter**

1 (default) | scalar value

Upper parameter for the uniform distribution, specified as a scalar value greater than lower.

### **Data Types**

single | double

**Properties** **lower**

Lower parameter for the uniform distribution, stored as a scalar value.

**Data Types**

single | double

**upper**

Upper parameter for the uniform distribution, stored as a scalar value greater than lower.

**Data Types**

single | double

**DistributionName**

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

**Data Types**

char

**IsTruncated**

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated. If IsTruncated equals 1, the distribution is truncated. This property is read-only.

**Data Types**

logical

**NumParameters**

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

**Data Types**

single | double

**ParameterDescription**

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

# prob.UniformDistribution

---

## Data Types

char

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

## Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object



pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Uniform Distribution

The uniform distribution has a constant probability density function between its two parameters, lower (the minimum) and upper (the maximum). This distribution is appropriate for representing round-off errors in values tabulated to a particular number of decimal places.

The uniform distribution uses the following parameters.

Parameter	Description	Support
lower	Lower parameter	$-\infty < lower < upper$
upper	Upper parameter	$lower < upper < \infty$

The probability density function (pdf) is

# prob.UniformDistribution

---

$$f(x | lower, upper) = \begin{cases} \left( \frac{1}{upper - lower} \right) & ; \text{ } lower \leq x \leq upper \\ 0 & ; \text{ } otherwise \end{cases} .$$

and 0 otherwise.

## Examples

### Create a Uniform Distribution Object Using Default Parameters

Create a uniform distribution object using the default parameter values.

```
pd = makedist('Uniform')
```

```
pd =
```

```
UniformDistribution
```

```
Uniform distribution
```

```
Lower = 0
```

```
Upper = 1
```

### Create a Uniform Distribution Object Using Specified Parameters

Create a uniform distribution object by specifying parameter values.

```
pd = makedist('Uniform','Lower',-4,'Upper',2)
```

```
pd =
```

```
UniformDistribution
```

```
Uniform distribution
```

```
Lower = -4
```

```
Upper = 2
```

Compute the interquartile range of the distribution

```
r = iqr(pd)
```

r =

3

**See Also** `makedist`

**Concepts**

- “Uniform Distribution (Continuous)” on page B-133
- Class Attributes
- Property Attributes

# dataset.union

---

**Purpose** Set union for dataset array observations

**Syntax**

```
C = union(A,B)
C = union(A,B,vars)
C = union(A,B,vars,setOrder)
[C,iA,iB] = union( ___ )
```

**Description** `C = union(A,B)` for dataset arrays `A` and `B` returns the combined set of observations from the two arrays, with repetitions removed. The observations in the dataset array `C` are sorted.

`C = union(A,B,vars)` returns the combined set of observations from the two arrays, with repetitions of unique combinations of the variables specified in `vars` removed. The observations in the dataset array `C` are sorted by those variables.

The values for variables not specified in `vars` for each observation in `C` are taken from the corresponding observation in `A` or `B`, or from `A` if there are common observations in both `A` and `B`. If there are multiple observations in `A` or `B` that correspond to an observation in `C`, those values are taken from the first occurrence.

`C = union(A,B,vars,setOrder)` returns the observations in `C` in the order specified by `setOrder`.

`[C,iA,iB] = union( ___ )` also returns index vectors `iA` and `iB` such that `C` is a sorted combination of the values `A(iA,:)` and `B(iB,:)`. If there are common observations in `A` and `B`, then `union` returns only the index from `A`, in `iA`. If there are repeated observations in `A` or `B`, then the index of the first occurrence is returned. You can use any of the previous input arguments.

## Input Arguments

**A,B**  
Input dataset arrays.

**vars**  
Cell array of strings containing variable names or a vector of integers containing variable column numbers, indicating

the variables for which `union` removes repetitions of unique combinations of the variables.

Specify `vars` as `[]` to use its default value of all variables.

### **setOrder**

Flag indicating the sorting order for the observations in `C`. The possible values of `setOrder` are:

'sorted'	Observations in <code>C</code> are in sorted order (default).
'stable'	Observations in <code>C</code> are in the same order that they appear in <code>A</code> , then <code>B</code> .

## **Output Arguments**

### **C**

Dataset array with the combined observations of `A` and `B`, with repetitions removed. `C` is in sorted order (by default), or the order specified by `setOrder`.

### **iA**

Index vector, indicating the observations in `A` that contribute to the union. `iA` contains the index to the first occurrence of any repeated observations in `A`.

### **iB**

Index vector, indicating the observations in `B` that contribute to the union. If there are common observations in `A` and `B`, then `union` returns only the index from `A`, in `iA`. `iB` contains the index to the first occurrence of any repeated observations in `B`.

## **Examples**

### **Union of Two Dataset Arrays**

Navigate to the folder containing sample data, and load sample data.

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

# dataset.union

---

```
A = dataset('XLSFile','hospitalSmall.xlsx');  
B = dataset('XLSFile','hospitalSmall.xlsx','Sheet',2);  
[length(A) length(B)]
```

```
ans =  
  
    14     8
```

The first dataset array, A, has 14 observations. The second dataset array, B, has 8 observations.

Return the union.

```
C = union(A,B);  
length(C)
```

```
ans =  
  
    21
```

The union of the two dataset arrays has 21 observations, indicating that there was one observation replicated in A and B.

## See Also

[dataset](#) | [intersect](#) | [ismember](#) | [setdiff](#) | [setxor](#) | [sortrows](#) | [unique](#) |

## Related Examples

- “Create a Dataset Array from a File” on page 2-71
- “Merge Dataset Arrays” on page 2-101

## Concepts

- “Dataset Arrays” on page 2-135

**Purpose** Unique observations in dataset array

---

**Note** In a future release, the behavior of `dataset.unique` will change to be consistent with the MATLAB function `unique`. This behavior change is optional in R2012a. For a demonstration of using the 'R2012a' flag to preview the future behavior, or the 'legacy' flag to preserve the current behavior in your existing code, see the documentation for `unique`.

---

**Syntax**

```
C = unique(A)
[C,ia,ic] = unique(A)
C = unique(A,vars)
[C,ia,ic] = unique(A,vars)
[...] = unique(A,vars,occurrence)
[...] = unique(...,'R2012a')
[...] = unique(...,'legacy')
[...] = unique(A,vars,setOrder)
```

**Description** `C = unique(A)` returns a copy of the dataset `A` that contains only the sorted unique observations. `A` must contain only variables whose class has a `unique` method, including:

- numeric
- character
- logical
- categorical
- cell arrays of strings

For a variable with multiple columns, its class's `unique` method must support the 'rows' flag.

`[C,ia,ic] = unique(A)` also returns index vectors `ia` and `ic` such that `C = A(ia,:)` and `A = C(ic,:)`.

# dataset.unique

---

`C = unique(A,vars)` returns a dataset that contains only one observation for each unique combination of values for the variables in `A` specified in `vars`. `vars` is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. `C` includes all variables from `A`. The values in `C` for the variables not specified in `vars` are taken from the last occurrence among observations in `A` with each unique combination of values for the variables specified in `vars`.

`[C,ia,ic] = unique(A,vars)` also returns index vectors `ia` and `ic` such that `C = A(ia,:)` and `A(:,vars) = C(ic,vars)`.

`[...] = unique(A,vars,occurrence)` specifies which index is returned in `ia` in the case of repeated observations in `A`. The default value is `occurrence='last'`, which returns the index of the last occurrence of each repeated observation in `A`. `occurrence='first'` returns the index of the first occurrence of each repeated observation in `A`. The values in `C` for variables not specified in `vars` are taken from the observations `A(ia,:)`. Specify `vars` as `[]` to use the default value of all variables.

`[...] = unique(...,'R2012a')` adopts the future behavior of `unique`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `vars`, or `occurrence`.

`[...] = unique(...,'legacy')` preserves the current behavior of `unique`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `vars`, or `occurrence`.

`[...] = unique(A,vars,setOrder)` returns the observations of `C` in a specific order. `setOrder='sorted'` returns the values of `C` in sorted order. `setOrder='stable'` returns the values of `C` in the same order as `A`. If there are repeated observations in `A`, then `ia` returns the index of the first occurrence of each repeated observation. Specify `vars` as `[]` to use the default value of all variables.

## See Also

`dataset` | `set` | `subsasgn`



**Purpose** Units of variables in data set

**Description** A cell array of strings giving the units of the variables in the data set. This property may be empty, but if not empty, the number of strings must equal the number of variables. Any individual string may be empty for a variable that does not have units defined. The default is an empty cell array.

# unidcdf

---

**Purpose** Discrete uniform cumulative distribution function

**Syntax**  
`p = unidcdf(x,N)`  
`p = unidcdf(x,N,'upper')`

**Description** `p = unidcdf(x,N)` returns the discrete uniform cdf at each value in `x` using the corresponding maximum observable value in `N`. `x` and `N` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The maximum observable values in `N` must be positive integers.

`p = unidcdf(x,N,'upper')` returns the complement of the discrete uniform cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The discrete uniform cdf is

$$p = F(x | N) = \frac{\text{floor}(x)}{N} I_{(1, \dots, N)}(x)$$

The result,  $p$ , is the probability that a single observation from the discrete uniform distribution with maximum  $N$  will be a positive integer less than or equal to  $x$ . The values  $x$  do not need to be integers.

## Examples **Compute Discrete Uniform Distribution cdf**

What is the probability of drawing a number 20 or less from a hat with the numbers from 1 to 50 inside?

```
probability = unidcdf(20,50)
```

```
probability =  
    0.4000
```

**See Also** `cdf` | `unidpdf` | `unidinv` | `unidstat` | `unidrnd` | `mle`

**How To** • “Uniform Distribution (Discrete)” on page B-139

**Purpose** Discrete uniform inverse cumulative distribution function

**Syntax** `X = unidinv(P,N)`

**Description** `X = unidinv(P,N)` returns the smallest positive integer `X` such that the discrete uniform cdf evaluated at `X` is equal to or exceeds `P`. You can think of `P` as the probability of drawing a number as large as `X` out of a hat with the numbers 1 through `N` inside.

`P` and `N` can be vectors, matrices, or multidimensional arrays that have the same size, which is also the size of `X`. A scalar input for `N` or `P` is expanded to a constant array with the same dimensions as the other input. The values in `P` must lie on the interval `[0 1]` and the values in `N` must be positive integers.

**Examples** `x = unidinv(0.7,20)`

```
x =
    14
```

```
y = unidinv(0.7 + eps,20)
```

```
y =
    15
```

A small change in the first parameter produces a large jump in output. The cdf and its inverse are both step functions. The example shows what happens at a step.

**See Also** `icdf` | `unidcdf` | `unidpdf` | `unidstat` | `unidrnd`

**How To** • “Uniform Distribution (Discrete)” on page B-139

# unidpdf

---

**Purpose** Discrete uniform probability density function

**Syntax** `Y = unidpdf(X,N)`

**Description** `Y = unidpdf(X,N)` computes the discrete uniform pdf at each of the values in `X` using the corresponding maximum observable value in `N`. `X` and `N` can be vectors, matrices, or multidimensional arrays that have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `N` must be positive integers.

The discrete uniform pdf is

$$y = f(x | N) = \frac{1}{N} I_{(1, \dots, N)}(x)$$

You can think of `y` as the probability of observing any one number between 1 and `n`.

**Examples** For fixed `n`, the uniform discrete pdf is a constant.

```
y = unidpdf(1:6,10)
y =
    0.1000    0.1000    0.1000    0.1000    0.1000    0.1000
```

Now fix `x`, and vary `n`.

```
likelihood = unidpdf(5,4:9)
likelihood =
    0    0.2000    0.1667    0.1429    0.1250    0.1111
```

**See Also** `pdf` | `unidcdf` | `unidinv` | `unidstat` | `unidrnd`

**How To** • “Uniform Distribution (Discrete)” on page B-139

---

<b>Purpose</b>	Discrete uniform random numbers
<b>Syntax</b>	<pre>R = unidrnd(N) R = unidrnd(N,m,n,...) R = unidrnd(N,[m,n,...])</pre>
<b>Description</b>	<p>R = unidrnd(N) generates random numbers for the discrete uniform distribution with maximum N. The parameters in N must be positive integers. N can be a vector, a matrix, or a multidimensional array. The size of R is the size of N. The discrete uniform distribution arises from experiments equivalent to drawing a number from one to N out of a hat.</p> <p>R = unidrnd(N,m,n,...) or R = unidrnd(N,[m,n,...]) generates an m-by-n-by-... array. The N parameter can be a scalar or an array of the same size as R.</p>
<b>Examples</b>	<p>In the Massachusetts lottery, a player chooses a four-digit number. Generate random numbers for Monday through Saturday.</p> <pre>numbers = unidrnd(10000,1,6)-1 numbers =     4564  185  8214  4447  6154  7919</pre>
<b>See Also</b>	<a href="#">random</a>   <a href="#">unidpdf</a>   <a href="#">unidcdf</a>   <a href="#">unidinv</a>   <a href="#">unidstat</a>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Uniform Distribution (Discrete)” on page B-139</li></ul>

# unidstat

---

**Purpose** Discrete uniform mean and variance

**Syntax** `[M,V] = unidstat(N)`

**Description** `[M,V] = unidstat(N)` returns the mean and variance of the discrete uniform distribution with minimum value 1 and maximum value  $N$ .

The mean of the discrete uniform distribution with parameter  $N$  is  $(N + 1)/2$ . The variance is  $(N^2 - 1)/12$ .

**Examples**

```
[m,v] = unidstat(1:6)
m =
    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000
v =
    0    0.2500    0.6667    1.2500    2.0000    2.9167
```

**See Also** `unidpdf` | `unidcdf` | `unidinv` | `unidrnd`

**How To**

- “Uniform Distribution (Discrete)” on page B-139

**Purpose**

Continuous uniform cumulative distribution function

**Syntax**

```
p = unifcdf(x,a,b)
p = unifcdf(x,a,b,'upper')
```

**Description**

`p = unifcdf(x,a,b)` returns the uniform cdf at each value in `x` using the corresponding lower endpoint (minimum), `a` and upper endpoint (maximum), `b`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant matrix with the same dimensions as the other inputs.

`p = unifcdf(x,a,b,'upper')` returns the complement of the uniform cdf at each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities.

The uniform cdf is

$$p = F(x | a, b) = \frac{x - a}{b - a} I_{[a, b]}(x)$$

The standard uniform distribution has `a = 0` and `b = 1`.

**Examples****Compute Uniform Distribution cdf**

What is the probability that an observation from a standard uniform distribution will be less than 0.75?

```
probability = unifcdf(0.75)
```

```
probability =
    0.7500
```

What is the probability that an observation from a uniform distribution with `a = -1` and `b = 1` will be less than 0.75?

```
probability = unifcdf(0.75, -1, 1)
```

```
probability =
    0.8750
```

# unifcdf

---

## See Also

`cdf` | `unifpdf` | `unifinv` | `unifstat` | `unifit` | `unifrnd`

## How To

- “Uniform Distribution (Continuous)” on page B-133



<b>Purpose</b>	Continuous uniform inverse cumulative distribution function
<b>Syntax</b>	<code>X = unifinv(P,A,B)</code>
<b>Description</b>	<p><code>X = unifinv(P,A,B)</code> computes the inverse of the uniform cdf with parameters <code>A</code> and <code>B</code> (the minimum and maximum values, respectively) at the corresponding probabilities in <code>P</code>. <code>P</code>, <code>A</code>, and <code>B</code> can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs.</p> <p>The inverse of the uniform cdf is</p> $x = F^{-1}(p   a, b) = a + p(a - b)I_{[0,1]}(p)$ <p>The standard uniform distribution has <code>A = 0</code> and <code>B = 1</code>.</p>
<b>Examples</b>	<p>What is the median of the standard uniform distribution?</p> <pre>median_value = unifinv(0.5) median_value =     0.5000</pre> <p>What is the 99th percentile of the uniform distribution between -1 and 1?</p> <pre>percentile = unifinv(0.99, -1, 1) percentile =     0.9800</pre>
<b>See Also</b>	<code>icdf</code>   <code>unifcdf</code>   <code>unifpdf</code>   <code>unifstat</code>   <code>unifit</code>   <code>unifrnd</code>
<b>How To</b>	<ul style="list-style-type: none"><li>• “Uniform Distribution (Continuous)” on page B-133</li></ul>

# unifit

---

**Purpose** Continuous uniform parameter estimates

**Syntax**  
`[ahat,bhat] = unifit(data)`  
`[ahat,bhat,ACI,BCI] = unifit(data)`  
`[ahat,bhat,ACI,BCI] = unifit(data,alpha)`

**Description** `[ahat,bhat] = unifit(data)` returns the maximum likelihood estimates (MLEs) of the parameters of the uniform distribution given the data in `data`.

`[ahat,bhat,ACI,BCI] = unifit(data)` also returns 95% confidence intervals, `ACI` and `BCI`, which are matrices with two rows. The first row contains the lower bound of the interval for each column of the matrix `data`. The second row contains the upper bound of the interval.

`[ahat,bhat,ACI,BCI] = unifit(data,alpha)` enables you to control of the confidence level `alpha`. For example, if `alpha = 0.01` then `ACI` and `BCI` are 99% confidence intervals.

**Examples**

```
r = unifrnd(10,12,100,2);  
[ahat,bhat,aci,bci] = unifit(r)  
ahat =  
    10.0154    10.0060  
bhat =  
    11.9989    11.9743  
aci =  
    9.9551    9.9461  
    10.0154    10.0060  
bci =  
    11.9989    11.9743  
    12.0592    12.0341
```

**See Also** `mle` | `unifpdf` | `unifcdf` | `unifinv` | `unifstat` | `unifrnd`

**How To** • “Uniform Distribution (Continuous)” on page B-133

**Purpose** Continuous uniform probability density function

**Syntax** `Y = unifpdf(X,A,B)`

**Description** `Y = unifpdf(X,A,B)` computes the continuous uniform pdf at each of the values in `X` using the corresponding lower endpoint (minimum), `A` and upper endpoint (maximum), `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array with the same dimensions as the other inputs. The parameters in `B` must be greater than those in `A`.

The continuous uniform distribution pdf is

$$y = f(x | a, b) = \frac{1}{b - a} I_{[a, b]}(x)$$

The standard uniform distribution has `A = 0` and `B = 1`.

**Examples** For fixed `a` and `b`, the uniform pdf is constant.

```
x = 0.1:0.1:0.6;
y = unifpdf(x)
y =
    1    1    1    1    1    1
```

What if `x` is not between `a` and `b`?

```
y = unifpdf(-1,0,1)
y =
    0
```

**See Also** `pdf` | `unifcdf` | `unifinv` | `unifstat` | `unifit` | `unifrnd`

**How To** • “Uniform Distribution (Continuous)” on page B-133

# unifrnd

---

**Purpose** Continuous uniform random numbers

**Syntax**  
R = unifrnd(A,B)  
R = unifrnd(A,B,m,n,...)  
R = unifrnd(A,B,[m,n,...])

**Description** R = unifrnd(A,B) returns an array R of random numbers generated from the continuous uniform distributions with lower and upper endpoints specified by A and B, respectively. If A and B are arrays, R(i,j) is generated from the distribution specified by the corresponding elements of A and B. If either A or B is a scalar, it is expanded to the size of the other input.

R = unifrnd(A,B,m,n,...) or R = unifrnd(A,B,[m,n,...]) returns an m-by-n-by-... array. If A and B are scalars, all elements of R are generated from the same distribution. If either A or B is an array, they must be m-by-n-by-... .

**Examples** Generate one random number each from the continuous uniform distributions on the intervals (0,1), (0,2), ..., (0,5):

```
a = 0; b = 1:5;  
r1 = unifrnd(a,b)  
r1 =  
    0.8147    1.8116    0.3810    3.6535    3.1618
```

Generate five random numbers each from the same distributions:

```
B = repmat(b,5,1);  
R = unifrnd(a,B)  
R =  
    0.0975    0.3152    0.4257    2.6230    3.7887  
    0.2785    1.9412    1.2653    0.1428    3.7157  
    0.5469    1.9143    2.7472    3.3965    1.9611  
    0.9575    0.9708    2.3766    3.7360    3.2774  
    0.9649    1.6006    2.8785    2.7149    0.8559
```

Generate five random numbers from the continuous uniform distribution on (0,2):

```
r2 = unifrnd(a,b(2),1,5)
r2 =
    1.4121    0.0637    0.5538    0.0923    0.1943
```

**See Also**

[rand](#) | [random](#) | [unifpdf](#) | [unifcdf](#) | [unifinv](#) | [unifstat](#) | [unifit](#)

**How To**

- “Uniform Distribution (Continuous)” on page B-133

# unifstat

---

**Purpose** Continuous uniform mean and variance

**Syntax** `[M,V] = unifstat(A,B)`

**Description** `[M,V] = unifstat(A,B)` returns the mean of and variance for the continuous uniform distribution using the corresponding lower endpoint (minimum), A and upper endpoint (maximum), B. Vector or matrix inputs for A and B must have the same size, which is also the size of M and V. A scalar input for A or B is expanded to a constant matrix with the same dimensions as the other input.

The mean of the continuous uniform distribution with parameters  $a$  and  $b$  is  $(a + b)/2$ , and the variance is  $(a - b)^2/12$ .

## Examples

```
a = 1:6;
b = 2.*a;
[m,v] = unifstat(a,b)
m =
    1.5000    3.0000    4.5000    6.0000    7.5000    9.0000
v =
    0.0833    0.3333    0.7500    1.3333    2.0833    3.0000
```

**See Also** `unifpdf` | `unifcdf` | `unifinv` | `unifit` | `unifrnd`

**How To** • “Uniform Distribution (Continuous)” on page B-133

**Purpose** Set union for categorical arrays

---

**Note** In a future release, the behavior of `categorical.union` will change to be consistent with the MATLAB function `union`. This behavior change is optional in R2012a. For a demonstration of using the 'R2012a' flag to preview the future behavior, or the 'legacy' flag to preserve the current behavior in your existing code, see the documentation for `union`.

---

**Syntax**

```
C = union(A,B)
[C,IA,IB] = union(A,B)
[...] = union(A,B,'rows')
[...] = union(...,'R2012a')
[...] = union(...,'legacy')
[...] = union(A,B,setOrder)
[...] = union(A,B,'rows',setOrder)
```

**Description** `C = union(A,B)` for categorical vectors `A` and `B`, returns a categorical vector `C` containing the combined values from `A` and `B` with no repetitions. The result `C` is sorted. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs, as determined by their labels.

`[C,IA,IB] = union(A,B)` also returns index vectors `IA` and `IB` such that `C` is a sorted combination of the elements `A(IA)` and `B(IB)`. If there are common values in `A` and `B`, then the index is returned in `IB`. If there are repeated values in `A` or `B`, then the index of the last occurrence of each repeated value is returned.

`[...] = union(A,B,'rows')` for categorical matrices `A` and `B` with the same number of columns, returns the combined rows from the two matrices with no repetitions. The rows of the matrix `C` are sorted. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs. The optional outputs `IA` and `IB` are index vectors such that `C` is the sorted combination of the rows `C = A(IA,:)` and `C = B(IB,:)`.

## categorical.union

---

`[...] = union(..., 'R2012a')` adopts the future behavior of `union`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

`[...] = union(..., 'legacy')` preserves the current behavior of `union`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `B`, or `'rows'`.

`[...] = union(A,B,setOrder)` and `[...] = union(A,B,'rows',setOrder)` returns the observations of `C` in a specific order. `setOrder='sorted'` returns the values or rows of `C` in sorted order. `setOrder='stable'` returns the values or rows of `C` in the same order as `A`, then `B`. If `A` and `B` are row vectors, then `C` is also a row vector. Otherwise, `C` is a column vector. If `A` and `B` are column vectors, then `C` is also a column vector. If there are repeated common values in `A` or `B`, then the index of the first occurrence of each repeated value is returned.

### See Also

`intersect` | `ismember` | `setdiff` | `setxor` | `unique`



**Purpose** Unique values in categorical array

---

**Note** In a future release, the behavior of `categorical.unique` will change to be consistent with the MATLAB function `unique`. This behavior change is optional in R2012a. For a demonstration of using the 'R2012a' flag to preview the future behavior, or the 'legacy' flag to preserve the current behavior in your existing code, see the documentation for `unique`.

---

**Syntax**

```
C = unique(A)
[C,IA,IC] = unique(A)
[...] = unique(A,'rows')
[...] = unique(A,occurrence)
[...] = unique(A,'rows',occurrence)
[...] = unique(...,'R2012a')
[...] = unique(...,'legacy')
[...] = unique(A,setOrder)
[...] = unique(A,'rows',setOrder)
```

**Description** `C = unique(A)` returns a categorical array containing the same values as in `A` but with no repetitions. `C` is sorted by the order of `A`'s levels.

`[C,IA,IC] = unique(A)` also returns index vectors `IA` and `IC` such that `C = A(IA)` and `A = C(IC)`.

`[...] = unique(A,'rows')` for the categorical matrix `A` returns the unique rows of `A`. The rows of the matrix `C` are sorted by the order of `A`'s levels. The optional outputs `IA` and `IC` are index vectors such that `C = A(IA,:)` and `A = C(IC,:)`.

`[...] = unique(A,occurrence)` and `[...] = unique(A,'rows',occurrence)` specify which index is returned in `IA` in the case of repeated values or rows in `A`. The default value is `occurrence='last'`, which returns the index of the last occurrence of each repeated value or row in `A`. `occurrence='first'` returns the index of the first occurrence of each repeated value or row in `A`.

## categorical.unique

---

`[...] = unique(..., 'R2012a')` adopts the future behavior of `unique`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `'rows'`, or `occurrence`.

`[...] = unique(..., 'legacy')` preserves the current behavior of `unique`. You can specify the flag as the final argument with any previous syntax that accepts `A`, `'rows'`, or `occurrence`.

`[...] = unique(A, setOrder)` and `[...] = unique(A, 'rows', setOrder)` returns the values or rows of `C` in a specific order. `setOrder='sorted'` returns the values or rows of `C` in sorted order. `setOrder='stable'` returns the values or rows of `C` in the same order as `A`. If `A` is a row vector, then `C` is also a row vector. Otherwise, `C` is a column vector. `IA` and `IC` are column vectors. If there are repeated values in `A`, then `IA` returns the index of the first occurrence of each repeated value.

### See Also

`intersect` | `ismember` | `setdiff` | `setxor` | `union`

**Purpose**

Unstack data from single variable into multiple variables

**Syntax**

```
wide = unstack(tall, datavar, indvar)
[wide, itall] = unstack(tall, datavar, indvar)
wide = unstack(tall, datavar, indvar, 'Parameter', value)
```

**Description**

`wide = unstack(tall, datavar, indvar)` converts a dataset array `tall` to an equivalent dataset array `wide` that is in wide format, by unstacking a single variable in `tall` into multiple variables in `wide`. In general `wide` contains more variables, but fewer observations, than `tall`.

`datavar` specifies the data variable in `tall` to unstack. `indvar` specifies an indicator variable in `tall` that determines which variable in `wide` each value in `datavar` is unstacked into. `unstack` treats the remaining variables in `tall` as grouping variables. Each unique combination of their values defines a group of observations in `tall` that will be unstacked into a single observation in `wide`.

`unstack` creates `m` data variables in `wide`, where `m` is the number of group levels in `indvar`. The values in `indvar` indicate which of those `m` variables receive which values from `datavar`. The `j`-th data variable in `wide` contains the values from `datavar` that correspond to observations whose `indvar` value was the `j`-th of the `m` possible levels. Elements of those `m` variables for which no corresponding data value in `tall` exists contain a default value.

`datavar` is a positive integer, a variable name, or a logical vector containing a single true value. `indvar` is a positive integer, a variable name, or a logical vector containing a single true value.

`[wide, itall] = unstack(tall, datavar, indvar)` returns an index vector `itall` indicating the correspondence between observations in `wide` and those in `tall`. For each observation in `wide`, `itall` contains the index of the first in the corresponding group of observations in `tall`.

For more information on grouping variables, see “Grouping Variables” on page 2-51.

# dataset.unstack

---

## Input Arguments

`wide = unstack(tall, datavar, indvar, 'Parameter', value)` uses the following parameter name/value pairs to control how `unstack` converts variables in `tall` to variables in `wide`:

'GroupVars '	Grouping variables in <code>tall</code> that define groups of observations. <code>groupvars</code> is a positive integer, a vector of positive integers, a variable name, a cell array containing one or more variable names, or a logical vector. The default is all variables in <code>tall</code> not listed in <code>datavar</code> or <code>indvar</code> .
'NewDataVarNames '	A cell array of strings containing names for the data variables <code>unstack</code> should create in <code>wide</code> . Default is the group names of the grouping variable specified in <code>indvar</code> .
'AggregationFun '	A function handle that accepts a subset of values from <code>datavar</code> and returns a single value. <code>stack</code> applies this function to observations from the same group that have the same value of <code>indvar</code> . The function must aggregate the data values into a single value, and in such cases it is not possible to recover <code>tall</code> from <code>wide</code> using <code>stack</code> . The default is <code>@sum</code> for numeric data variables. For non-numeric variables, there is no default, and you must specify 'AggregationFun' if multiple observations in the same group have the same values of <code>indvar</code> .
'ConstVars '	Variables in <code>tall</code> to copy to <code>wide</code> without unstacking. The values for these variables in <code>wide</code> are taken from the first observation in each group in <code>tall</code> , so these variables should typically be constant within each group. <code>ConstVars</code> is a positive integer, a vector of positive integers, a variable name, a cell array

	containing one or more variable names, or a logical vector. The default is no variables.
--	--

You can also specify more than one data variable in `tall`, each of which becomes a set of `m` variables in `wide`. In this case, specify `datavar` as a vector of positive integers, a cell array containing variable names, or a logical vector. You may specify only one variable with `indvar`. The names of each set of data variables in `wide` are the name of the corresponding data variable in `tall` concatenated with the names specified in `'NewDataVarNames'`. The function specified in `'AggregationFun'` must return a value with a single row.

## Examples

Convert a "wide format" data set to "tall format", and then back to a different "wide format":

```
load flu

% FLU has a 'Date' variable, and 10 variables for estimated
% influenza rates (in 9 different regions, estimated from
% Google searches, plus a nationwide estimate from the
% CDC). Combine those 10 variables into a "tall" array that
% has a single data variable, 'FluRate', and an indicator
% variable, 'Region', that says which region each estimate
% is from.
[flu2,iflu] = stack(flu, 2:11, 'NewDataVarName','FluRate', ...
    'IndVarName','Region')

% The second observation in FLU is for 10/16/2005. Find the
% observations in FLU2 that correspond to that date.
flu(2,:)
flu2(iflu==2,:)

% Use the 'Date' variable from that tall array to split
% 'FluRate' into 52 separate variables, each containing the
% estimated influenza rates for each unique date. The new
% "wide" array has one observation for each region. In
% effect, this is the original array FLU "on its side".
dateNames = cellstr(datestr(flu.Date,'mmm_DD_YYYY'));
```

# dataset.unstack

---

```
[flu3,iflu2] = unstack(flu2, 'FluRate', 'Date', ...
    'NewDataVarNames',dateNames)

% Since observations in FLU3 represent regions, IFLU2
% indicates the first occurrence in FLU2 of each region.
flu2(iflu2,:)
```

## See Also

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

## How To

- “Grouping Variables” on page 2-51

**Purpose** Upper Pareto tails parameters

**Syntax** `params = upperparams(obj)`

**Description** `params = upperparams(obj)` returns the 2-element vector `params` of shape and scale parameters, respectively, of the upper tail of the Pareto tails object `obj`. `upperparams` does not return a location parameter.

**Examples** Fit Pareto tails to a  $t$  distribution at cumulative probabilities 0.1 and 0.9:

```
t = trnd(3,100,1);  
obj = paretotails(t,0.1,0.9);
```

```
lowerparams(obj)  
ans =  
    -0.1901    1.1898  
upperparams(obj)  
ans =  
    0.3646    0.5103
```

**See Also** `paretotails` | `lowerparams`

## dataset.UserData property

---

**Purpose** Variable containing additional information associated with data set

**Description** Any variable containing additional information to be associated with the data set. The default is an empty array.



<b>Purpose</b>	Variance of probability distribution object
<b>Syntax</b>	<code>v = var(pd)</code>
<b>Description</b>	<code>v = var(pd)</code> returns the variance <code>v</code> of the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Fit a probability distribution object to data using <code>fitdist</code> or the Distribution Fitting app.
<b>Output Arguments</b>	<b>v - Variance</b> nonnegative scalar value  Variance of the probability distribution, returned as a nonnegative scalar value.
<b>Examples</b>	<b>Variance of a Fitted Distribution</b>  Load the sample data. Create a vector containing the first column of students' exam grade data.  <pre>load examgrades; x = grades(:,1);</pre> Fit a kernel distribution object to the data.  <pre>pd = fitdist(x,'Kernel')</pre> <pre>pd =  KernelDistribution  Kernel = normal Bandwidth = 3.61677 Support = unbounded</pre>

# prob.KernelDistribution.var

---

Compute the variance of the fitted distribution.

```
v = var(pd)
```

```
v =
```

```
88.4893
```

## See Also

[fitdist](#) | [dfittool](#)

<b>Purpose</b>	Return variance of ProbDistUnivParam object		
<b>Syntax</b>	$V = \text{var}(PD)$		
<b>Description</b>	$V = \text{var}(PD)$ returns $V$ , the variance of the ProbDistUnivParam object $PD$ .		
<b>Input Arguments</b>	<table><tr><td><math>PD</math></td><td>An object of the class ProbDistUnivParam.</td></tr></table>	$PD$	An object of the class ProbDistUnivParam.
$PD$	An object of the class ProbDistUnivParam.		
<b>Output Arguments</b>	<table><tr><td><math>V</math></td><td>The variance of the ProbDistUnivParam object <math>PD</math>.</td></tr></table>	$V$	The variance of the ProbDistUnivParam object $PD$ .
$V$	The variance of the ProbDistUnivParam object $PD$ .		
<b>See Also</b>	var		

# prob.ParametricTruncatableDistribution.var

---

<b>Purpose</b>	Variance of probability distribution object
<b>Syntax</b>	<code>v = var(pd)</code>
<b>Description</b>	<code>v = var(pd)</code> returns the variance <code>v</code> of the probability distribution <code>pd</code> .
<b>Input Arguments</b>	<b>pd - Probability distribution</b> probability distribution object  Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using <code>makedist</code> .
<b>Output Arguments</b>	<b>v - Variance</b> nonnegative scalar value  Variance of the probability distribution, returned as a nonnegative scalar value.
<b>Examples</b>	<b>Variance of a Triangular Distribution</b>  Create a triangular distribution object.  <code>pd = makedist('Triangular','a',-3,'b',1,'c',3)</code>  <code>pd =</code>  TriangularDistribution  A = -3, B = 1, C = 3  Compute the variance of the distribution.  <code>v = var(pd)</code>  <code>v =</code>  1.5556

**See Also** `makedist`

# prob.ToolboxFittableParametricDistribution.var

---

**Purpose** Variance of probability distribution object

**Syntax** `v = var(pd)`

**Description** `v = var(pd)` returns the variance `v` of the probability distribution `pd`.

**Input Arguments** **pd - Probability distribution**  
probability distribution object

Probability distribution, specified as a probability distribution object. Create a probability distribution object with specified parameter values using `makedist`. Alternatively, create a probability distribution object by fitting it to data using `fitdist` or the Distribution Fitting app.

**Output Arguments** **v - Variance**  
nonnegative scalar value

Variance of the probability distribution, returned as a nonnegative scalar value.

**Examples** **Variance of a Fitted Distribution**

Load the sample data. Create a vector containing the first column of students' exam grade data.

```
load examgrades;  
x = grades(:,1);
```

Fit a normal distribution object to the data.

```
pd = fitdist(x,'Normal')
```

```
pd =
```

```
NormalDistribution
```

```
Normal distribution
```

```
mu = 75.0083    [73.4321, 76.5846]  
sigma = 8.7202  [7.7391, 9.98843]
```

Compute the variance of the fitted distribution.

```
v = var(pd)
```

```
v =
```

```
76.0419
```

For a normal distribution, the variance is equal to the square of the parameter `sigma`.

## Variance of a Skewed Distribution

Create a Weibull probability distribution object.

```
pd = makedist('Weibull','a',5,'b',2)
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 5
```

```
B = 2
```

Compute the variance of the distribution.

```
v = var(pd)
```

```
v =
```

```
5.3650
```

## See Also

[makedist](#) | [fitdist](#) | [dfittool](#)

# dataset.VarDescription property

---

**Purpose**

Cell array of strings giving descriptions of variables in data set

**Description**

A cell array of strings giving the descriptions of the variables in the data set. This property may be empty, but if not empty, the number of strings must equal the number of variables. Any individual string may be empty for a variable that does not have a description defined. The default is an empty cell array.



<b>Purpose</b>	Compute embedded estimates of input feature importance
<b>Syntax</b>	<code>imp = varimportance(t)</code>
<b>Description</b>	<code>imp = varimportance(t)</code> computes estimates of input feature importance for tree <code>t</code> by summing changes in the risk due to splits on every feature. The returned vector <code>imp</code> has one element for each input variable in the data used to train this tree. At each node, the risk is estimated as node impurity if impurity was used to split nodes and node error otherwise. This risk is weighted by the node probability. Variable importance associated with this split is computed as the difference between the risk for the parent node and the total risk for the two children.
<b>See Also</b>	<code>risk</code>

# dataset.VarNames property

---

**Purpose** Cell array giving names of variables in data set

**Description** A cell array of nonempty, distinct strings giving the names of the variables in the data set. The number of strings must equal the number of variables. The default is the cell array of string names for the variables used to create the data set.

# TreeBagger.VarNames property

---

**Purpose**

Variable names

**Description**

The VarNames property is a cell array containing the names of the predictor variables (features). TreeBagger takes these names from the optional 'names' parameter. The default names are 'x1', 'x2', etc.

# vartest

---

**Purpose** Chi-square variance test

**Syntax**  
`h = vartest(x,v)`  
`h = vartest(x,v,Name,Value)`  
`[h,p] = vartest( ___ )`  
`[h,p,ci,stats] = vartest( ___ )`

**Description** `h = vartest(x,v)` returns a test decision for the null hypothesis that the data in vector `x` comes from a normal distribution with variance `v`, using the chi-square variance test. The alternative hypothesis is that `x` comes from a normal distribution with a different variance. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = vartest(x,v,Name,Value)` performs the chi-square variance test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = vartest( ___ )` also returns the *p*-value of the test, `p`, using any of the input arguments in the previous syntaxes.

`[h,p,ci,stats] = vartest( ___ )` also returns the confidence interval for the true variance, `ci`, and the structure `stats` containing information about the test statistic.

## Input Arguments

**x - Sample data**  
vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array. For matrices, `vartest` performs separate tests along each column of `x`, and returns a row vector of results. For multidimensional arrays, `vartest` works along the first nonsingleton dimension of `x`.

**Data Types**  
single | double

**v - Hypothesized variance**

nonnegative scalar value

Hypothesized variance, specified as a nonnegative scalar value.

**Data Types**

single | double

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

**Example:** `'Tail', 'right', 'Alpha', 0.01` specifies a right-tailed hypothesis test at the 1% significance level.

**'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

**Example:** `'Alpha', 0.01`

**Data Types**

single | double

**'Dim' - Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix to test along, specified as the comma-separated pair consisting of `'Dim'` and a positive integer value. For example, specifying `'Dim', 1` tests the data in each column for equality to the hypothesized variance, while `'Dim', 2` tests the data in each row.

**Example:** `'Dim', 2`

## Data Types

single | double

## 'Tail' - Type of alternative hypothesis

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'	Test the alternative hypothesis that the population variance is not $v$ .
'right'	Test the alternative hypothesis that the population variance is greater than $v$ .
'left'	Test the alternative hypothesis that the population variance is less than $v$ .

Example: 'Tail', 'right'

## Output Arguments

### h - Hypothesis test result

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

### p - p-value

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### ci - Confidence interval

vector

Confidence interval for the true variance, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

### **stats - Test statistics**

structure

Test statistics for the chi-square variance test, returned as a structure containing:

- `chisqstat` — Value of the test statistic.
- `df` — Degrees of freedom of the test.

## **Examples**

### **Test for a Specified Variance**

Load the sample data. Create a vector containing the first column of the students' exam grades matrix.

```
load examgrades;
x = grades(:,1);
```

Test the null hypothesis that the data comes from a distribution with a variance of 25.

```
[h,p,ci,stats] = vartest(x,25)
```

```
h =
     1
```

```
p =
     0
```

```
ci =
    59.8936
    99.7688
```

```
stats =
```

```
chisqstat: 361.9597
df: 119
```

The returned value  $h = 1$  indicates that `vartest` rejects the null hypothesis at the default 5% significance level. `ci` shows the lower and upper boundaries of the 95% confidence interval for the true variance, and suggests that the true variance is greater than 25.

## One-Sided Hypothesis Test

Load the sample data. Create a vector containing the first column of the students' exam grades matrix.

```
load examgrades;
x = grades(:,1);
```

Test the null hypothesis that the data comes from a distribution with a variance of 25, against the alternative hypothesis that the variance is greater than 25.

```
[h,p] = vartest(x,25,'Tail','right')
```

```
h =
    1
```

```
p =
 2.4269e-26
```

The returned value of  $h = 1$  indicates that `vartest` rejects the null hypothesis at the default 5% significance level, in favor of the alternative hypothesis that the variance is greater than 25.

## Definitions

### Chi-Square Variance Test

The chi-square variance test is used to test if the variance of a population is equal to a hypothesized value.

The test statistic is



$$T = \frac{(N-1)}{\left(\frac{s}{\sigma_0}\right)^2},$$

where  $N$  is the sample size,  $s$  is the sample standard deviation, and  $\sigma_0$  is the hypothesized standard deviation. The denominator is the ratio of the sample standard deviation to the hypothesized standard deviation. The further this ratio deviates from 1, the more likely you are to reject the null hypothesis. The test statistic  $T$  has a chi-square distribution with  $N - 1$  degrees of freedom under the null hypothesis.

### **Multidimensional Array**

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### **First Nonsingleton Dimension**

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

### **See Also**

vartest2 | vartestn

# vartest2

---

**Purpose** Two-sample  $F$ -test for equal variances

**Syntax**

```
h = vartest2(x,y)
h = vartest2(x,y,Name,Value)
[h,p] = vartest2( ___ )
[h,p,ci,stats] = vartest2( ___ )
```

**Description** `h = vartest2(x,y)` returns a test decision for the null hypothesis that the data in vectors `x` and `y` comes from normal distributions with the same variance, using the two-sample  $F$ -test. The alternative hypothesis is that they come from normal distributions with different variances. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h = vartest2(x,y,Name,Value)` returns a test decision for the two-sample  $F$ -test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = vartest2( ___ )` also returns the  $p$ -value of the test, `p`, using any of the input arguments in the previous syntaxes.

`[h,p,ci,stats] = vartest2( ___ )` also returns the confidence interval for the true variance ratio, `ci`, and the structure `stats` containing information about the test statistic.

## Input Arguments

### **x - Sample data**

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array.

- If `x` and `y` are vectors, they do not need to be the same length.
- If `x` and `y` are matrices, they must have the same number of columns, but do not need to have the same number of rows. `vartest2` performs separate tests along each column and returns a vector of the results.

- If `x` and `y` are multidimensional arrays, they must have the same number of dimensions, and the same size along all but the first nonsingleton dimension.

**Data Types**`single` | `double`**y - Sample data**`vector` | `matrix` | `multidimensional array`

Sample data, specified as a vector, matrix, or multidimensional array.

- If `x` and `y` are vectors, they do not need to be the same length.
- If `x` and `y` are matrices, they must have the same number of columns, but do not need to have the same number of rows. `vartest2` performs separate tests along each column and returns a vector of the results.
- If `x` and `y` are multidimensional arrays, they must have the same number of dimensions, and the same size along all but the first nonsingleton dimension.

**Data Types**`single` | `double`**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( `' '` ). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**Example:** `'Tail', 'right', 'Alpha', 0.01` specifies a right-tailed hypothesis test at the 1% significance level.

**'Alpha' - Significance level**`0.05 (default)` | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

## **Data Types**

single | double

## **'Dim' - Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix to test along, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim',1 tests the data in each column for variance equality, while 'Dim',2 tests the data in each row.

**Example:** 'Dim',2

## **Data Types**

single | double

## **'Tail' - Type of alternative hypothesis**

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate using the  $F$ -test, specified as the comma-separated pair consisting of 'Tail' and one of the following.

- |         |   |
|---------|---|
| 'both'  | Test the alternative hypothesis that the population variances are not equal.                      |
| 'right' | Test the alternative hypothesis that the population variance of $x$ is greater than that of $y$ . |
| 'left'  | Test the alternative hypothesis that the population variance of $x$ is less than that of $y$ .    |

**Example:** 'Tail','right'

## **Output Arguments**

### **h - Hypothesis test result**

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

**p - p-value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

**ci - Confidence interval**

vector

Confidence interval for the true ratio of the population variances, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

**stats - Test statistics**

structure

Test statistics for the hypothesis test, returned as a structure containing:

- `fstat` — Value of the test statistic.
- `df1` — Numerator degrees of freedom of the test.
- `df2` — Denominator degrees of freedom of the test.

**Examples****Test for Equal Variances**

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;
x = grades(:,1);
y = grades(:,2);
```

## vartest2

---

Test the null hypothesis that the data in  $x$  and  $y$  comes from distributions with the same variance.

```
[h,p,ci,stats] = vartest2(x,y)
```

```
h =
```

```
    1
```

```
p =
```

```
    0.0019
```

```
ci =
```

```
    1.2383
```

```
    2.5494
```

```
stats =
```

```
    fstat: 1.7768
```

```
    df1: 119
```

```
    df2: 119
```

The returned result  $h = 1$  indicates that `vartest2` rejects the null hypothesis at the default 5% significance level. `ci` contains the lower and upper boundaries of the 95% confidence interval for the true variance ratio. `stats` contains the value of the test statistic for the  $F$ -test and the numerator and denominator degrees of freedom.

### One-Sided Hypothesis Test

Load the sample data. Create vectors containing the first and second columns of the data matrix to represent students' grades on two exams.

```
load examgrades;
```

```
x = grades(:,1);
```

```
y = grades(:,2);
```

Test the null hypothesis that the data in  $x$  and  $y$  comes from distributions with the same variance, against the alternative that the population variance of  $x$  is greater than that of  $y$ .

```
vartest2(x,y, 'Tail', 'right')
```

```
h =  
    1
```

```
p =  
    9.4364e-04
```

The returned result  $h = 1$  indicates that `vartest2` rejects the null hypothesis at the default 5% significance level, in favor of the alternative hypothesis that the population variance of  $x$  is greater than that of  $y$ .

## Definitions

### Two-Sample $F$ -Test

The two-sample  $F$ -test is used to test if the variances of two populations are equal.

The test statistic is

$$F = \frac{s_1^2}{s_2^2},$$

where  $s_1$  and  $s_2$  are the sample variances. The test statistic is a ratio of the two sample variances. The further this ratio deviates from 1, the more likely you are to reject the null hypothesis. Under the null hypothesis, the test statistic  $F$  has a  $F$ -distribution with numerator degrees of freedom equal to  $N_1 - 1$  and denominator degrees of freedom equal to  $N_2 - 1$ , where  $N_1$  and  $N_2$  are the sample sizes of the two data sets.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

## First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if `x` is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of `x`.

## See Also

`vartest` | `vartestn`



**Purpose**

Multiple-sample tests for equal variances

**Syntax**

```
vartestn(x)
vartestn(x,Name,Value)

vartestn(x,group)
vartestn(x,group,Name,Value)

p = vartestn( __ )
[p,stats] = vartestn( __ )
```

**Description**

`vartestn(x)` returns a summary table of statistics and a box plot for a Bartlett test of the null hypothesis that the columns of data vector `x` come from normal distributions with the same variance. The alternative hypothesis is that not all columns of data have the same variance.

`vartestn(x,Name,Value)` returns a summary table of statistics and a box plot for a test of unequal variances with additional options specified by one or more name-value pair arguments. For example, you can specify a different type of hypothesis test or change the display settings for the test results.

`vartestn(x,group)` returns a summary table of statistics and a box plot for a Bartlett test of the null hypothesis that the data in each categorical group comes from normal distributions with the same variance. The alternative hypothesis is that not all groups have the same variance.

`vartestn(x,group,Name,Value)` returns a summary table of statistics and a box plot for a test of unequal variances with additional options specified by one or more name-value pair arguments. For example, you can specify a different type of hypothesis test or change the display settings for the test results.

`p = vartestn( __ )` also returns the  $p$ -value of the test, `p`, using any of the input arguments in the previous syntaxes.

# vartestn

---

`[p,stats] = vartestn( ___ )` also returns the structure `stats` containing information about the test statistic.

## Input Arguments

### **x - Sample data**

matrix | vector

Sample data, specified as a matrix or vector. If a grouping variable `group` is specified, `x` must be a vector. If a grouping variable is not specified, `x` must be a matrix. In either case, `vartestn` treats NaN values as missing values and ignores them.

### **Data Types**

single | double

### **group - Grouping variable**

categorical array | logical or numeric vector | cell array of strings

Grouping variable, specified as a categorical array, logical or numeric vector, or cell array of strings with one row for each element of `x`. Each unique value in a grouping variable defines a group.

For example, if `Gender` is a cell array of strings with values 'Male' and 'Female', you can use `Gender` as a grouping variable to test your data by gender.

**Example:** Gender

### **Data Types**

single | double | logical | cell | char

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:**

```
'TestType','BrownForsythe','Alpha',0.01,'Display','off'
```

specifies a Brown-Forsythe test at the 1% significance level, and omits the plot of the results.

### **'Alpha' - Significance level**

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of 'Alpha' and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

### **Data Types**

single | double

### **'Dim' - Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the variances, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim',1 tests the column variances, while 'Dim',2 tests the row variances.

**Example:** 'Dim',2

### **Data Types**

single | double

### **'Display' - Display settings for test results**

'on' (default) | 'off'

Display settings for test results, specified as the comma-separated pair consisting of 'Display' and one of the following.

- |       |  |
|-------|--|
| 'on'  | Display a box plot and table of summary statistics.        |
| 'off' | Do not display a box plot and table of summary statistics. |

**Example:** 'display','off'

## 'TestType' - Type of hypothesis test

'Bartlett' (default) | 'LeveneQuadratic' | 'LeveneAbsolute' | 'BrownForsythe' | 'OBrien'

Type of hypothesis test to perform, specified as the comma-separated pair consisting of 'TestType' and one of the following.

'Bartlett'	Bartlett's test.
'LeveneQuadratic'	Levene's test computed by performing ANOVA on the squared deviations of the data values from their group means.
'LeveneAbsolute'	Levene's test computed by performing ANOVA on the absolute deviations of the data values from their group means.
'BrownForsythe'	Brown-Forsythe test computed by performing ANOVA on the absolute deviations of the data values from the group medians.
'OBrien'	O'Brien's modification of Levene's test with $W = 0.5$ .

**Example:** 'TestType', 'OBrien'

## Output Arguments

### **p** - p-value

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

### **stats** - Test statistics

structure

Test statistics for the hypothesis test, returned as a structure containing:

- **chistat:** Value of the test statistic.

- df: Degrees of freedom of the test.

## Examples

### Test Data for Equal Variances

Load the sample data.

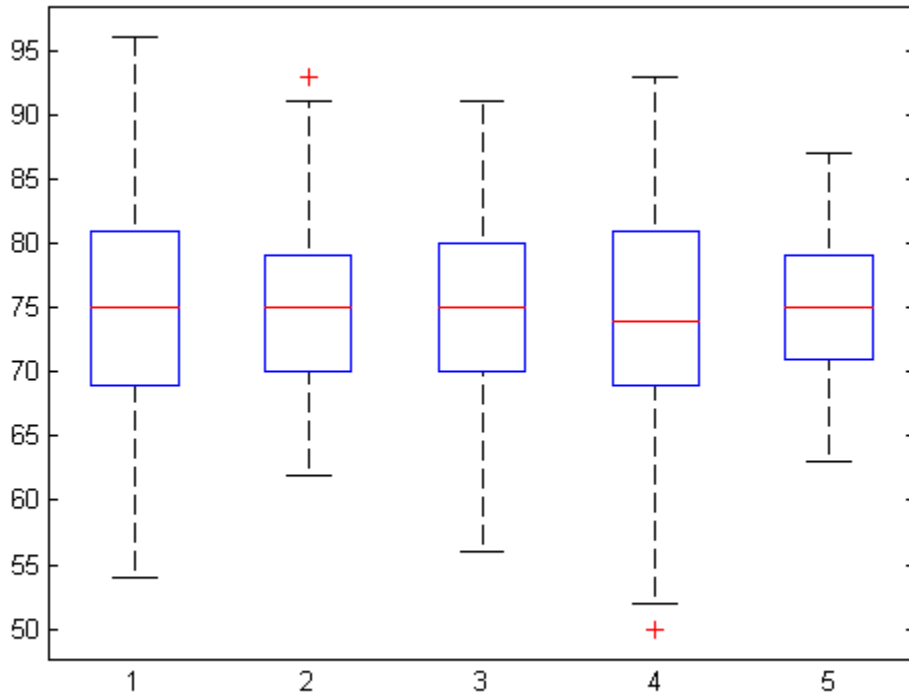
```
load examgrades;
```

Test the null hypothesis that the variances are equal across the five columns of data in the students' exam grades matrix, `grades`.

```
vartestn(grades)
```

Group Summary Table			
Group	Count	Mean	Std Dev
1	120	75.0083	8.7202
2	120	74.9917	6.54204
3	120	74.9917	7.43091
4	120	75.0333	8.60128
5	120	74.9917	5.25884
Pooled	600	75.0033	7.42558
Bartlett's statistic	38.7332		
Degrees of freedom	4		
p-value	0		

# vartestn



The low  $p$ -value,  $p = 0$ , indicates that `vartestn` rejects the null hypothesis that the variances are equal across all five columns, in favor of the alternative hypothesis that at least one column has a different variance.

## Test Grouped Data for Equal Variances

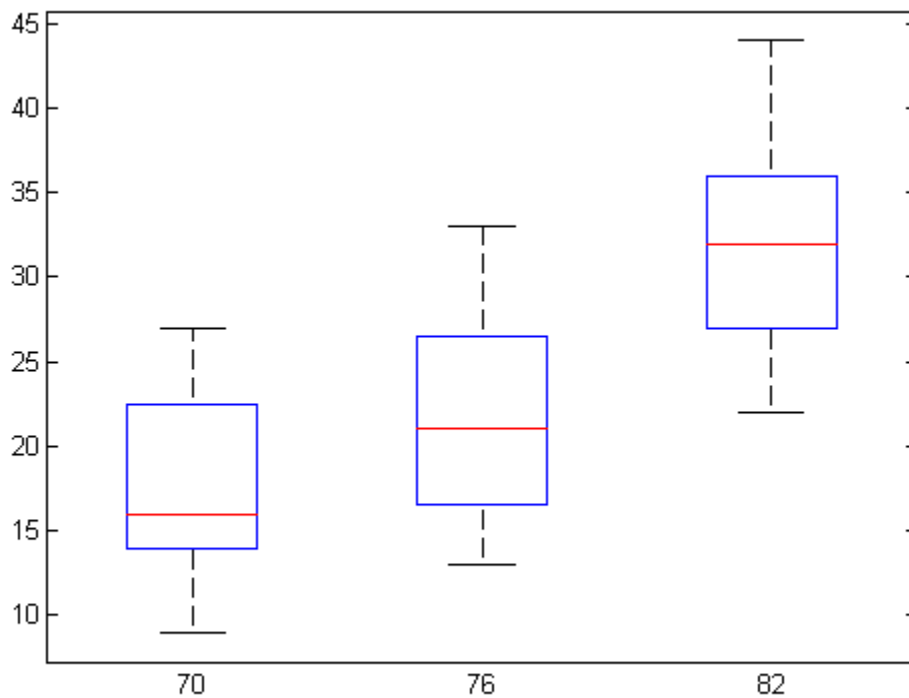
Load the sample data.

```
load carsmall;
```

Test the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

```
vartestn(MPG,Model_Year)
```

Group Summary Table			
Group	Count	Mean	Std Dev
70	29	17.6897	5.33923
76	34	21.5735	5.8893
82	31	31.7097	5.39255
Pooled	94	23.7181	5.562
Bartlett's statistic	0.36619		
Degrees of freedom	2		
p-value	0.83269		



The high  $p$ -value,  $p = 0.83269$ , indicates that `vartestn` does not reject the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

### Test for Equal Variances Using Levene's Test

Load the sample data.

```
load carsmall;
```



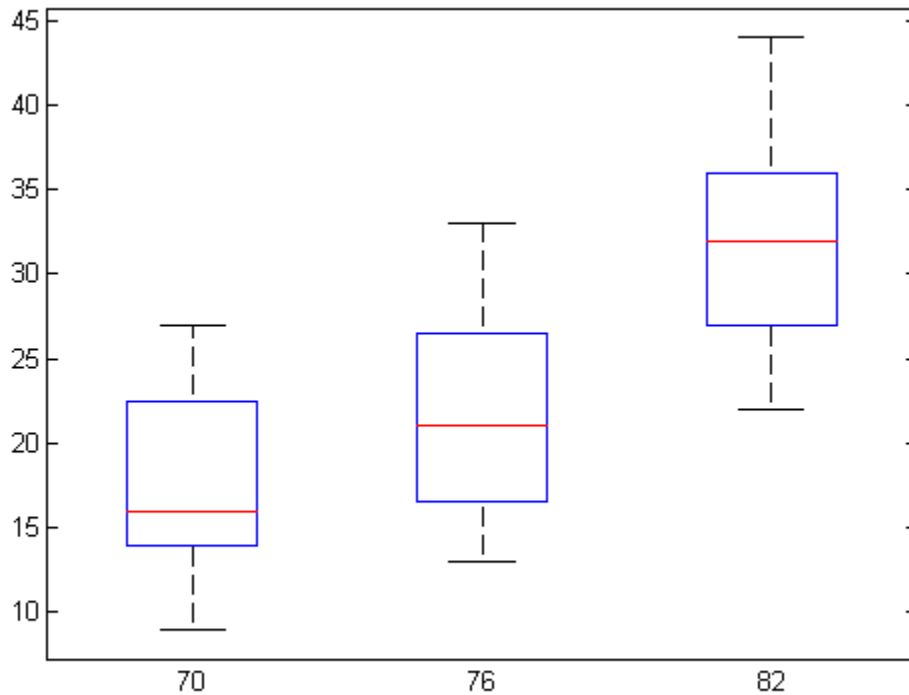
Use Levene's test to test the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

```
p = vartestn(MPG,Model_Year,'TestType','LeveneAbsolute')
```

p =

0.6320

Group Summary Table			
Group	Count	Mean	Std Dev
70	29	17.6897	5.33923
76	34	21.5735	5.8893
82	31	31.7097	5.39255
Pooled	94	23.7181	5.562
Levene's statistic (absolute)	0.46126		
Degrees of freedom	2, 91		
p-value	0.63195		



The high  $p$ -value,  $p = 0.63195$ , indicates that `vartestn` does not reject the null hypothesis that the variances in miles per gallon (MPG) are equal across different model years.

### Test for Equal Variances Using the Brown-Forsythe Test

Load the sample data.

```
load examgrades;
```

Test the null hypothesis that the variances are equal across the five columns of data in the students' exam grades matrix, `grades`, using the Brown-Forsythe test. Suppress the display of the summary table of statistics and the box plot.

```
[p,stats] = vartestn(grades,'TestType','BrownForsythe','Display','off')
```

```
p =
```

```
1.3121e-06
```

```
stats =
```

```
fstat: 8.4160
df: [4 595]
```

The low  $p$ -value,  $p = 1.3121e-06$ , indicates that `vartestn` rejects the null hypothesis that the variances are equal across all five columns, in favor of the alternative hypothesis that at least one column has a different variance.

## Definitions

### Bartlett's Test

Bartlett's test is used to test whether multiple data samples have equal variances, against the alternative that at least two of the data samples do not have equal variances.

The test statistic is

$$T = \frac{(N - k) \ln s_p^2 - \sum_{i=1}^k (N_i - 1) \ln s_i^2}{1 + (1 / (3(k - 1))) \left( \left( \sum_{i=1}^k 1 / (N_i - 1) \right) - 1 / (N - k) \right)},$$

where  $s_i^2$  is the variance of the  $i$ th group,  $N$  is the total sample size,  $N_i$  is the sample size of the  $i$ th group,  $k$  is the number of groups, and  $s_p^2$  is the pooled variance. The pooled variance is defined as

$$s_p^2 = \sum_{i=1}^k (N_i - 1) s_i^2 / (N - k).$$

The test statistic has a chi-square distribution with  $k - 1$  degrees of freedom under the null hypothesis.

Bartlett's test is sensitive to departures from normality. If your data comes from a nonnormal distribution, Levene's test could provide a more accurate result.

### **Levene, Brown-Forsythe, and O'Brien Tests**

The Levene, Brown-Forsythe, and O'Brien tests are used to test if multiple data samples have equal variances, against the alternative that at least two of the data samples do not have equal variances.

The test statistic is

$$W = \frac{(N - k) \sum_{i=1}^k N_i (\bar{Z}_i - \bar{Z}_{..})^2}{(k - 1) \sum_{i=1}^k \sum_{j=1}^{N_i} (Z_{ij} - \bar{Z}_i)^2},$$

where  $N_i$  is the sample size of the  $i$ th group, and  $k$  is the number of groups. Depending on the type of test specified with the `TestType` name-value pair arguments,  $Z_{ij}$  can have one of four definitions:

- If you specify `LeveneAbsolute`, `vartestn` uses  $Z_{ij} = |Y_{ij} - \bar{Y}_i|$ , where  $\bar{Y}_i$  is the mean of the  $i$ th subgroup.

- If you specify `LeveneQuadratic`, `vartestn` uses  $Z_{ij}^2 = (Y_{ij} - \bar{Y}_i)^2$ , where  $\bar{Y}_i$  is the mean of the  $i$ th subgroup.
- If you specify `BrownForsythe`, `vartestn` uses  $Z_{ij} = |Y_{ij} - \tilde{Y}_i|$ , where  $\tilde{Y}_i$  is the median of the  $i$ th subgroup.
- If you specify `OBrien`, `vartestn` uses

$$Z_{ij} = \frac{(0.5 + n_i - 2)n_i(y_{ij} - \bar{y}_i)^2 - 0.5(n_i - 1)\sigma_i^2}{(n_i - 1)(n_i - 2)},$$

where  $n_i$  is the size of the  $i$ th group,  $\sigma_i^2$  is its sample variance.

In all cases, the test statistic has an  $F$ -distribution with  $k - 1$  numerator degrees of freedom, and  $N - k$  denominator degrees of freedom.

The Levene, Brown-Forsythe, and O'Brien tests are less sensitive to departures from normality than Bartlett's test, so they are useful alternatives if you suspect the samples come from nonnormal distributions.

## See Also

`anova1` | `vartest` | `vartest2`

# categorical.vertcat

---

**Purpose** Vertical concatenation for categorical arrays

**Syntax** `C = vertcat(dim,A,B,...)`  
`C = vertcat(A,B)`

**Description** `C = vertcat(dim,A,B,...)` vertically concatenates the categorical arrays `A,B,...`. For matrices, all inputs must have the same number of rows. For n-D arrays, all inputs must have the same sizes except in the second dimension. The set of categorical levels for `C` is the sorted union of the sets of levels of the inputs, as determined by their labels.

`C = vertcat(A,B)` is called for the syntax `[A B]`.

**See Also** `cat` | `horzcat`

**Purpose** Vertical concatenation for dataset arrays

**Syntax** `ds = vertcat(ds1, ds2, ...)`

**Description** `ds = vertcat(ds1, ds2, ...)` vertically concatenates the dataset arrays `ds1`, `ds2`, ... . Observation names, when present, must be unique across datasets. `vertcat` fills in default observation names for the output when some of the inputs have names and some do not.

Variable names for all dataset arrays must be identical except for order. `vertcat` concatenates by matching variable names. `vertcat` assigns values for the "per-variable" properties (e.g., `Units` and `VarDescription`) in `ds` from the corresponding property values in `ds1`.

**See Also** `cat` | `horzcat`

# classregtree.view

---

**Purpose** Plot tree

**Syntax** `view(t)`  
`view(t,param1,val1,param2,val2,...)`

**Description** `view(t)` displays the decision tree `t` as computed by `classregtree` in a figure window. Each branch in the tree is labeled with its decision rule, and each terminal node is labeled with the predicted value for that node. Click any node to get more information about it. The information displayed is specified by the **Click to display** pop-up menu at the top of the figure.

`view(t,param1,val1,param2,val2,...)` specifies optional parameter name/value pairs:

- 'names' — A cell array of names for the predictor variables, in the order in which they appear in the matrix `X` from which the tree was created. (See `classregtree`.)
- 'prunelevel' — Initial pruning level to display.

For each branch node, the left child node corresponds to the points that satisfy the condition, and the right child node corresponds to the points that do not satisfy the condition.

## Examples

Create a classification tree for Fisher's iris data:

```
load fisheriris;

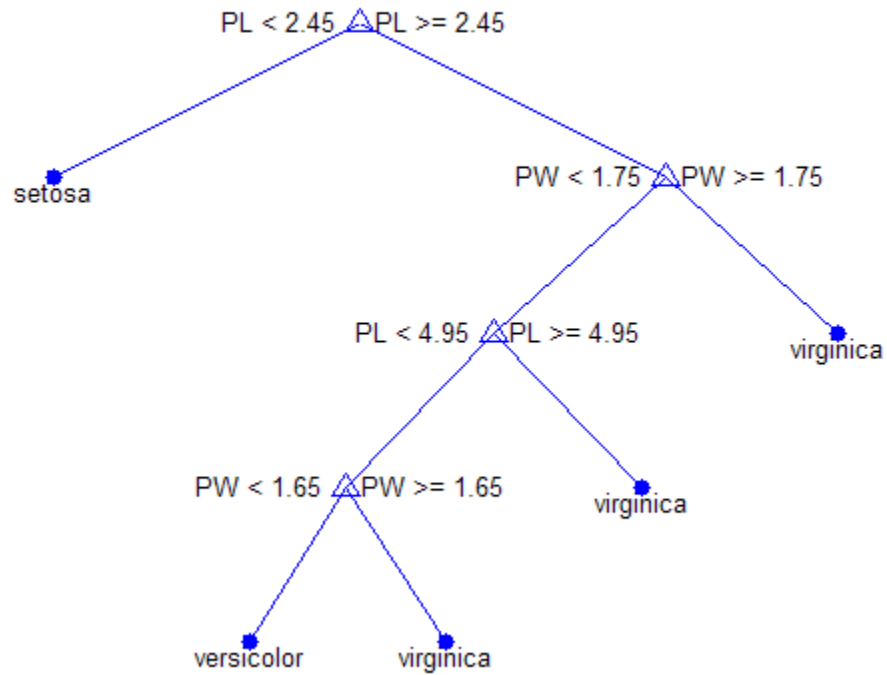
t = classregtree(meas,species,...
                'names',{'SL' 'SW' 'PL' 'PW'})
t =
Decision tree for classification
1  if PL<2.45 then node 2 elseif PL>=2.45 then node 3 else setosa
2  class = setosa
3  if PW<1.75 then node 4 elseif PW>=1.75 then node 5 else versicolor
4  if PL<4.95 then node 6 elseif PL>=4.95 then node 7 else versicolor
5  class = virginica
```



```
6 if PW<1.65 then node 8 elseif PW>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica

view(t)
```

Click to display:  Magnification:  Pruning level:



## References

[1] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.

# classregtree.view

---

## See Also

`classregtree` | `eval` | `prune` | `test`

<b>Purpose</b>	Compact clustering evaluation object
<b>Syntax</b>	<code>c = compact(eva)</code>
<b>Description</b>	<code>c = compact(eva)</code> returns a compact clustering evaluation object, which contains a subset of information about the clustering solution in <code>eva</code> . Compacting a clustering evaluation object reduces the memory requirements of the object, which is useful when clustering a large data set.
<b>Input Arguments</b>	<b>eva - Clustering evaluation data</b> clustering evaluation object  Clustering evaluation data, specified as a clustering evaluation object. Create a clustering evaluation object using <code>evalclusters</code> .
<b>Output Arguments</b>	<b>c - Compact clustering evaluation object</b> clustering evaluation object  Compact clustering evaluation object, returned as a clustering evaluation object. The compact object includes the clustering evaluation results. In the compact object, the properties for the input data <code>X</code> , optimal clustering solution <code>OptimalY</code> , and the list of excluded data <code>Missing</code> are empty.
<b>Examples</b>	<b>Create a Compact Clustering Evaluation Object</b>  Create a compact clustering evaluation object from a full clustering evaluation object.  Load the sample data.  <pre>load fisheriris;</pre>  The data contains length and width measurements from the sepals and petals of three species of iris flowers.

# clustering.evaluation.ClusterCriterion.compact

---

Create a clustering evaluation object. Cluster the data using `kmeans`, and evaluate the optimal number of clusters using the gap criterion.

```
rng('default'); % For reproducibility
eva = evalclusters(meas,'kmeans','Gap','KList',[1:6])

eva =

    GapEvaluation with properties:

        NumObservations: 150
        InspectedK: [1 2 3 4 5 6]
        CriterionValues: [0.0747 0.5906 0.8737 1.0055 1.0466 0.9848]
        OptimalK: 4
```

Create a compact clustering evaluation object from `eva`.

```
c = compact(eva)

c =

    GapEvaluation with properties:

        NumObservations: 150
        InspectedK: [1 2 3 4 5 6]
        CriterionValues: [0.0747 0.5906 0.8737 1.0055 1.0466 0.9848]
        OptimalK: 4
```

The displayed output of the compact object `c` is the same as the original object `eva`, but some properties not shown in the display are different. For example, in the compact object, the properties `x`, `OptimalY`, and `Missing` are empty.

Display the optimal clustering solution `OptimalY` for `c`.

```
c.OptimalY

ans =
```

[]

**See Also** `evalclusters`

# CompactClassificationTree.view

---

**Purpose** View tree

**Syntax** `view(tree)`  
`view(tree,Name,Value)`

**Description** `view(tree)` returns a text description of `tree`, a decision tree.  
`view(tree,Name,Value)` describes `tree` with additional options specified by one or more `Name,Value` pair arguments.

**Input Arguments** **tree**  
A classification tree or compact classification tree created by `ClassificationTree.fit` or `compact`.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### **'mode'**

String describing the display of `tree`, either 'graph' or 'text'. 'graph' opens a GUI displaying `tree`, and containing controls for querying the tree. 'text' sends output to the Command Window describing `tree`.

**Default:** 'text'

**Examples** View the classification tree for Fisher's iris model in both textual and graphical displays:

```
load fisheriris
tree = ClassificationTree.fit(meas,species);
view(tree)
```

# CompactClassificationTree.view

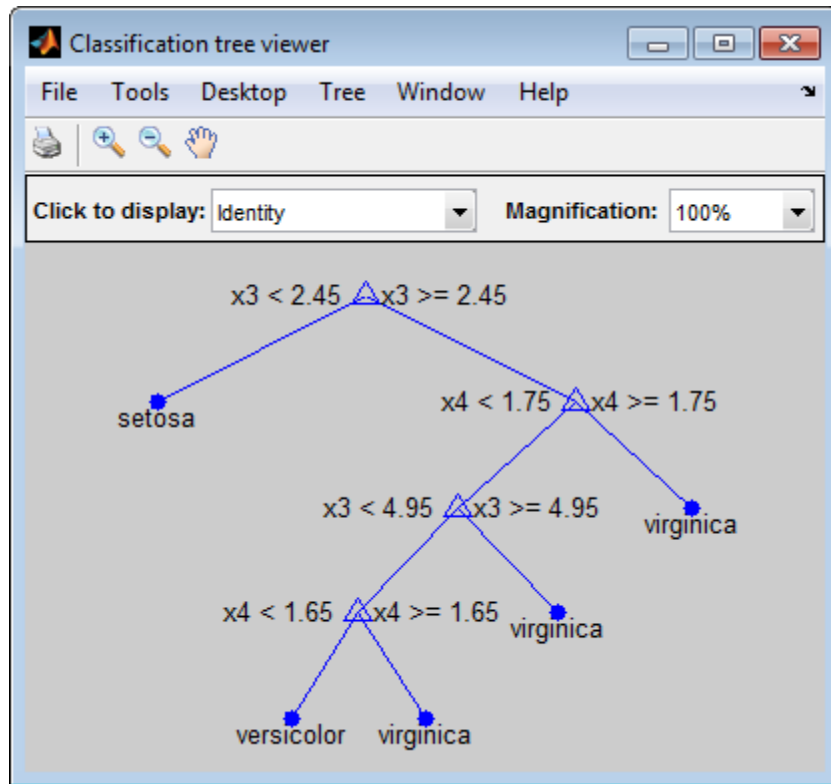
---

Decision tree for classification

```
1 if x3<2.45 then node 2 elseif x3>=2.45 then node 3 else setosa
2 class = setosa
3 if x4<1.75 then node 4 elseif x4>=1.75 then node 5 else versicolor
4 if x3<4.95 then node 6 elseif x3>=4.95 then node 7 else versicolor
5 class = virginica
6 if x4<1.65 then node 8 elseif x4>=1.65 then node 9 else versicolor
7 class = virginica
8 class = versicolor
9 class = virginica
```

```
view(tree,'mode','graph')
```

# CompactClassificationTree.view



**See Also**

ClassificationTree



**Purpose** View tree

**Syntax** `view(tree)`  
`view(tree,Name,Value)`

**Description** `view(tree)` returns a text description of `tree`, a decision tree.  
`view(tree,Name,Value)` describes `tree` with additional options specified by one or more `Name,Value` pair arguments.

**Input Arguments** **tree**  
A regression tree or compact regression tree created by `RegressionTree.fit` or `compact`.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### 'mode'

String describing the display of `tree`, either 'graph' or 'text'. 'graph' opens a GUI displaying `tree`, and containing controls for querying the tree. 'text' sends output to the Command Window describing `tree`.

**Default:** 'text'

**Examples** View a regression tree for the `carsmall` data in both textual and graphical displays:

```
load carsmall
tree = RegressionTree.fit([Weight, Cylinders],MPG,...
    'categoricalpredictors',2,'MinParent',20,...
    'PredictorNames',{'W','C'});
```

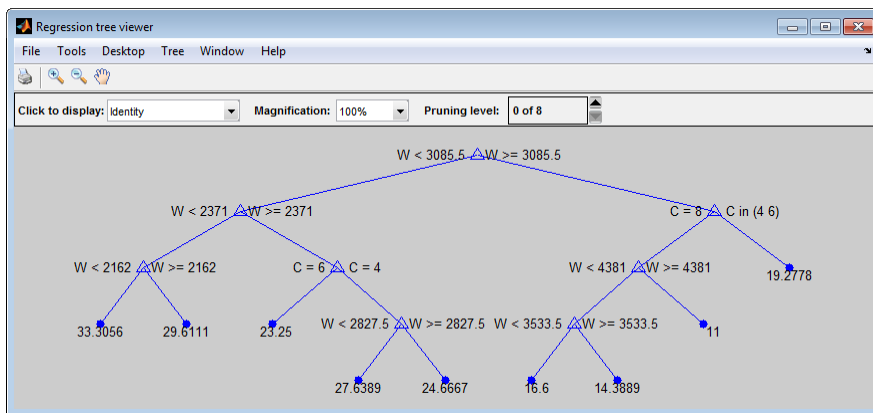
# CompactRegressionTree.view

```
view(tree)
```

Decision tree for regression

```
1 if W<3085.5 then node 2 elseif W>=3085.5 then node 3 else 23.7181
2 if W<2371 then node 4 elseif W>=2371 then node 5 else 28.7931
3 if C=8 then node 6 elseif C in {4 6} then node 7 else 15.5417
4 if W<2162 then node 8 elseif W>=2162 then node 9 else 32.0741
5 if C=6 then node 10 elseif C=4 then node 11 else 25.9355
6 if W<4381 then node 12 elseif W>=4381 then node 13 else 14.2963
7 fit = 19.2778
8 fit = 33.3056
9 fit = 29.6111
10 fit = 23.25
11 if W<2827.5 then node 14 elseif W>=2827.5 then node 15 else 27.2143
12 if W<3533.5 then node 16 elseif W>=3533.5 then node 17 else 14.8696
13 fit = 11
14 fit = 27.6389
15 fit = 24.6667
16 fit = 16.6
17 fit = 14.3889
```

```
view(tree, 'mode', 'graph')
```



## See Also

[RegressionTree](#)

# wblcdf

---

**Purpose** Weibull cumulative distribution function

**Syntax**

```
p = wblcdf(x,a,b)
[p,plo,pup] = wblcdf(x,a,b,pcov,alpha)
[p,plo,pup] = wblcdf( __ , 'upper' )
```

**Description** `p = wblcdf(x,a,b)` returns the cdf of the Weibull distribution with scale parameter `a` and shape parameter `b`, at each value in `x`. `x`, `a`, and `b` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `a` and `b` are both 1. The parameters `a` and `b` must be positive.

`[p,plo,pup] = wblcdf(x,a,b,pcov,alpha)` returns confidence bounds for `p` when the input parameters `a` and `b` are estimates. `pcov` is the 2-by-2 covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1 - `alpha`)% confidence bounds. `plo` and `pup` are arrays of the same size as `p` containing the lower and upper confidence bounds.

`[p,plo,pup] = wblcdf( __ , 'upper' )` returns the complement of the Weibull cdf for each value in `x`, using an algorithm that more accurately computes the extreme upper tail probabilities. You can use 'upper' with any of the previous syntaxes.

The function `wblcdf` computes confidence bounds for `p` using a normal approximation to the distribution of the estimate

$$\hat{b}(\log x - \log \hat{a})$$

and then transforms those bounds to the scale of the output `p`. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `pcov` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The Weibull cdf is

$$p = F(x | a, b) = \int_0^x b a^{-b} t^{b-1} e^{-\left(\frac{t}{a}\right)^b} dt = 1 - e^{-\left(\frac{x}{a}\right)^b} I_{(0, \infty)}(x)$$

**Examples****Weibull Distribution cdf**

What is the probability that a value from a Weibull distribution with parameters  $a = 0.15$  and  $b = 0.8$  is less than 0.5?

```
probability = wblcdf(0.5, 0.15, 0.8)
```

```
probability =  
0.9272
```

How sensitive is this result to small changes in the parameters?

```
[A, B] = meshgrid(0.1:0.05:0.2, 0.2:0.05:0.3);  
probability = wblcdf(0.5, A, B)
```

```
probability =  
0.7484 0.7198 0.6991  
0.7758 0.7411 0.7156  
0.8022 0.7619 0.7319
```

**See Also**

[cdf](#) | [wblpdf](#) | [wblinv](#) | [wblstat](#) | [wblfit](#) | [wbllike](#) | [wblrnd](#)

**How To**

- “Weibull Distribution” on page B-141

**Purpose** Weibull parameter estimates

**Syntax**

```
parmhat = wblfit(data)
[parmhat,parmci] = wblfit(data)
[parmhat,parmci] = wblfit(data,alpha)
[...] = wblfit(data,alpha,censoring)
[...] = wblfit(data,alpha,censoring,freq)
[...] = wblfit(...,options)
```

**Description** `parmhat = wblfit(data)` returns the maximum likelihood estimates, `parmhat`, of the parameters of the Weibull distribution given the values in the vector `data`, which must be positive. `parmhat` is a two-element row vector: `parmhat(1)` estimates the Weibull parameter  $a$ , and `parmhat(2)` estimates the Weibull parameter  $b$ , in the pdf

$$y = f(x | a, b) = ba^{-b} x^{b-1} e^{-\left(\frac{x}{a}\right)^b} I_{(0, \infty)}(x)$$

`[parmhat,parmci] = wblfit(data)` returns 95% confidence intervals for the estimates of  $a$  and  $b$  in the 2-by-2 matrix `parmci`. The first row contains the lower bounds of the confidence intervals for the parameters, and the second row contains the upper bounds of the confidence intervals.

`[[parmhat,parmci] = wblfit(data,alpha)` returns  $100(1 - \text{alpha})\%$  confidence intervals for the parameter estimates.

`[...] = wblfit(data,alpha,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = wblfit(data,alpha,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. The vector `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any non-negative values. Pass in `[]` for `alpha`, `censoring`, or `freq` to use their default values.

[...] = wblfit(...,options) accepts a structure, options, that specifies control parameters for the iterative algorithm the function uses to compute maximum likelihood estimates. The Weibull fit function accepts an options structure that can be created using the function statset. Enter statset ('wblfit') to see the names and default values of the parameters that lognfit accepts in the options structure. See the reference page for statset for more information about these options.

## Examples

```
data = wblrnd(0.5,0.8,100,1);
[parmhat, parmci] = wblfit(data)
parmhat =
    0.5861    0.8567
parmci =
    0.4606    0.7360
    0.7459    0.9973
```

## See Also

mle | wbllike | wblpdf | wblcdf | wblinv | wblstat | wblrnd

## How To

- “Weibull Distribution” on page B-141

# wblinv

---

**Purpose** Weibull inverse cumulative distribution function

**Syntax** `X = wblinv(P,A,B)`  
`[X,XLO,XUP] = wblinv(P,A,B,PCOV,alpha)`

**Description** `X = wblinv(P,A,B)` returns the inverse cumulative distribution function (cdf) for a Weibull distribution with scale parameter `A` and shape parameter `B`, evaluated at the values in `P`. `P`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The default values for `A` and `B` are both 1.

`[X,XLO,XUP] = wblinv(P,A,B,PCOV,alpha)` returns confidence bounds for `X` when the input parameters `A` and `B` are estimates. `PCOV` is a 2-by-2 matrix containing the covariance matrix of the estimated parameters. `alpha` has a default value of 0.05, and specifies 100(1 - `alpha`)% confidence bounds. `XLO` and `XUP` are arrays of the same size as `X` containing the lower and upper confidence bounds.

The function `wblinv` computes confidence bounds for `X` using a normal approximation to the distribution of the estimate

$$\log a + \frac{\log q}{b}$$

where `q` is the `P`th quantile from a Weibull distribution with scale and shape parameters both equal to 1. The computed bounds give approximately the desired confidence level when you estimate `mu`, `sigma`, and `PCOV` from large samples, but in smaller samples other methods of computing the confidence bounds might be more accurate.

The inverse of the Weibull cdf is

$$x = F^{-1}(p | a, b) = -a [\ln(1 - p)]^{1/b} I_{[0,1]}(p)$$

**Examples** The lifetimes (in hours) of a batch of light bulbs has a Weibull distribution with parameters `a = 200` and `b = 6`.



Find the median lifetime of the bulbs:

```
life = wblinv(0.5, 200, 6)
life =
    188.1486
```

---

Generate 100 random values from this distribution, and estimate the 90th percentile (with confidence bounds) from the random sample

```
x = wblrnd(200,6,100,1);
p = wblfit(x)
[nlogl,pcov] = wbllike(p,x)
[q90,q90lo,q90up] = wblinv(0.9,p(1),p(2),pcov)
p =
```

```
    204.8918    6.3920
```

```
nlogl =
```

```
    496.8915
```

```
pcov =
```

```
    11.3392    0.5233
     0.5233    0.2573
```

```
q90 =
```

```
    233.4489
```

```
q90lo =
```

226.0092

q90up =

241.1335

## **See Also**

[icdf](#) | [wblcdf](#) | [wblpdf](#) | [wblstat](#) | [wblfit](#) | [wbllike](#) | [wblrnd](#)

## **How To**

- “Weibull Distribution” on page B-141

**Purpose**

Weibull negative log-likelihood

**Syntax**

```
nlogL = wbllike(params,data)
[logL,AVAR] = wbllike(params,data)
[...] = wbllike(params,data,censoring)
[...] = wbllike(params,data,censoring,freq)
```

**Description**

`nlogL = wbllike(params,data)` returns the Weibull log-likelihood. `params(1)` is the scale parameter, `A`, and `params(2)` is the shape parameter, `B`.

`[logL,AVAR] = wbllike(params,data)` also returns `AVAR`, which is the asymptotic variance-covariance matrix of the parameter estimates if the values in `params` are the maximum likelihood estimates. `AVAR` is the inverse of Fisher's information matrix. The diagonal elements of `AVAR` are the asymptotic variances of their respective parameters.

`[...] = wbllike(params,data,censoring)` accepts a Boolean vector, `censoring`, of the same size as `data`, which is 1 for observations that are right-censored and 0 for observations that are observed exactly.

`[...] = wbllike(params,data,censoring,freq)` accepts a frequency vector, `freq`, of the same size as `data`. `freq` typically contains integer frequencies for the corresponding elements in `data`, but can contain any nonnegative values. Pass in `[]` for `censoring` to use its default value.

The Weibull negative log-likelihood for uncensored data is

$$(-\log L) = -\log \prod_{i=1}^n f(a,b | x_i) = -\sum_{i=1}^n \log f(a,b | x_i)$$

where  $f$  is the Weibull pdf.

`wbllike` is a utility function for maximum likelihood estimation.

**Examples**

This example continues the example from `wblfit`.

```
r = wblrnd(0.5,0.8,100,1);
```

```
[logL, AVAR] = wbllike(wblfit(r),r)
logL =
    47.3349
AVAR =
    0.0048    0.0014
    0.0014    0.0040
```

## References

[1] Patel, J. K., C. H. Kapadia, and D. B. Owen. *Handbook of Statistical Distributions*. New York: Marcel Dekker, 1976.

## See Also

wblfit | wblpdf | wblcdf | wblinv | wblstat | wblrnd

## How To

- “Weibull Distribution” on page B-141

**Purpose** Weibull probability density function

**Syntax** `Y = wblpdf(X,A,B)`

**Description** `Y = wblpdf(X,A,B)` computes the Weibull pdf at each of the values in `X` using the corresponding scale parameter, `A` and shape parameter, `B`. `X`, `A`, and `B` can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other inputs. The parameters in `A` and `B` must be positive.

The Weibull pdf is

$$f(x | a, b) = \frac{b}{a} \left( \frac{x}{a} \right)^{b-1} e^{-(x/a)^b}.$$

Some references refer to the Weibull distribution with a single parameter. This corresponds to `wblpdf` with `A = 1`.

**Examples** The exponential distribution is a special case of the Weibull distribution.

```
lambda = 1:6;
y = wblpdf(0.1:0.1:0.6,lambda,1)
y =
    0.9048    0.4524    0.3016    0.2262    0.1810    0.1508

y1 = exppdf(0.1:0.1:0.6,lambda)
y1 =
    0.9048    0.4524    0.3016    0.2262    0.1810    0.1508
```

**References** [1] Devroye, L. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986.

**See Also** pdf | wblcdf

## How To

- wblfit
- wblinv
- wbllike
- wblplot
- wblrnd
- wblstat
- “Weibull Distribution” on page B-141

**Purpose** Weibull probability plot

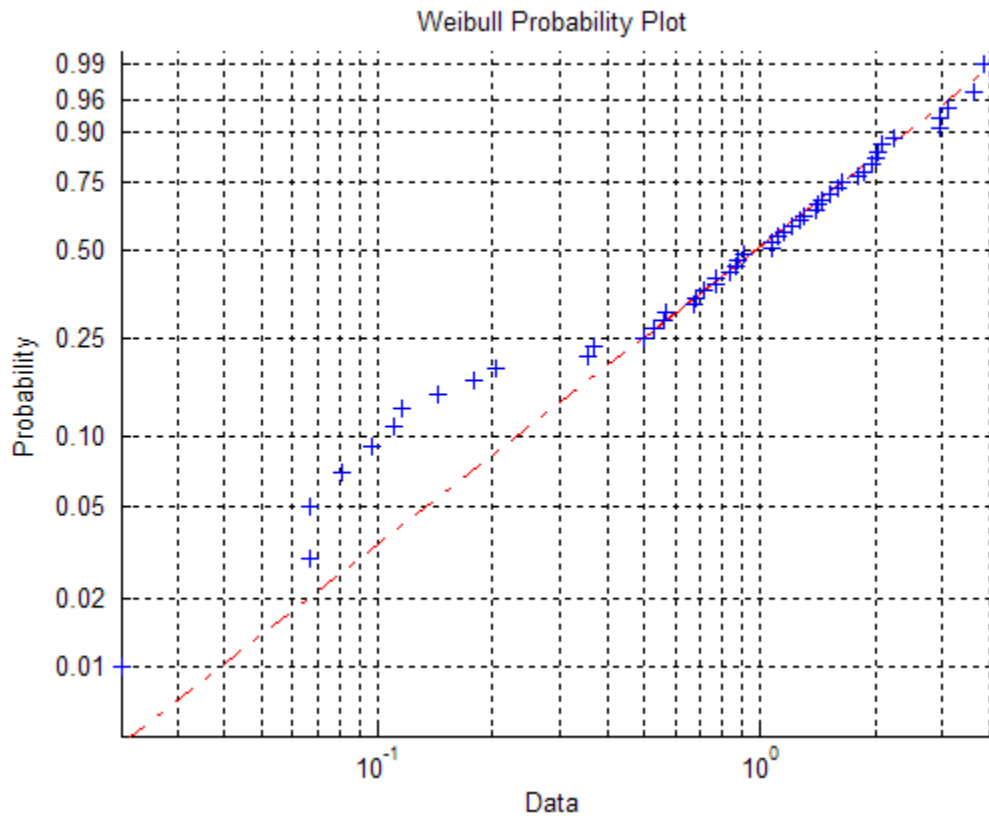
**Syntax** `wblplot(X)`  
`h = wblplot(X)`

**Description** `wblplot(X)` displays a Weibull probability plot of the data in `X`. If `X` is a matrix, `wblplot` displays a plot for each column.

`h = wblplot(X)` returns handles to the plotted lines.

The purpose of a Weibull probability plot is to graphically assess whether the data in `X` could come from a Weibull distribution. If the data are Weibull the plot will be linear. Other distribution types might introduce curvature in the plot. `wblplot` uses midpoint probability plotting positions. Use `probplot` when the data included censored observations.

**Examples** `r = wblrnd(1.2,1.5,50,1);`  
`wblplot(r)`



## See Also

[proplot](#) | [normplot](#) | [wblcdf](#)

## How To

- [wblfit](#)
- [wblinv](#)
- [wbllike](#)
- [wblpdf](#)
- [wblrnd](#)



- wblstat
- “Weibull Distribution” on page B-141

# wblrnd

---

**Purpose** Weibull random numbers

**Syntax**  
R = wblrnd(A,B)  
R = wblrnd(A,B,m,n,...)  
R = wblrnd(A,B,[m,n,...])

**Description** R = wblrnd(A,B) generates random numbers for the Weibull distribution with scale parameter, A and shape parameter, B. The input arguments A and B can be either scalars or matrices. A and B, can be vectors, matrices, or multidimensional arrays that all have the same size. A scalar input is expanded to a constant array of the same size as the other input.

R = wblrnd(A,B,m,n,...) or R = wblrnd(A,B,[m,n,...]) generates an m-by-n-by-... array. The A, B parameters can each be scalars or arrays of the same size as R.

Devroye [1] refers to the Weibull distribution with a single parameter; this is wblrnd with A = 1.

**Examples**  
n1 = wblrnd(0.5:0.5:2,0.5:0.5:2)  
n1 =  
0.0178 0.0860 2.5216 0.9124

n2 = wblrnd(1/2,1/2,[1 6])  
n2 =  
0.0046 1.7214 2.2108 0.0367 0.0531 0.0917

**References** [1] Devroye, L. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986.

**See Also** random | wblpdf | wblcdf | wblinv | wblstat | wblfit | wbllike

**How To** • “Weibull Distribution” on page B-141

**Purpose** Weibull mean and variance

**Syntax** `[M,V] = wblstat(A,B)`

**Description** `[M,V] = wblstat(A,B)` returns the mean of and variance for the Weibull distribution with scale parameter, A and shape parameter, B. Vector or matrix inputs for A and B must have the same size, which is also the size of M and V. A scalar input for A or B is expanded to a constant matrix with the same dimensions as the other input.

The mean of the Weibull distribution with parameters  $a$  and  $b$  is

$$a \left[ \Gamma(1+b^{-1}) \right]$$

and the variance is

$$a^2 \left[ \Gamma(1+2b^{-1}) - \Gamma(1+b^{-1})^2 \right]$$

### Examples

```
[m,v] = wblstat(1:4,1:4)
m =
    1.0000    1.7725    2.6789    3.6256
v =
    1.0000    0.8584    0.9480    1.0346
```

```
wblstat(0.5,0.7)
ans =
    0.6329
```

### See Also

wblpdf | wblcdf | wblinv | wblfit | wbllike | wblrnd

### How To

- “Weibull Distribution” on page B-141

# prob.WeibullDistribution

---

**Superclasses** ToolboxFittableParametricDistribution

**Purpose** Weibull probability distribution object

**Description** `prob.WeibullDistribution` is an object consisting of parameters, a model description, and sample data for a Weibull probability distribution.

Create a probability distribution object with specified parameter values using `makedist`. Alternatively, fit a distribution to data using `fitdist` or the Distribution Fitting app.

**Construction** `pd = makedist('Weibull')` creates a Weibull probability distribution object using the default parameter values.

`pd = makedist('Weibull','a',a,'b',b)` creates a Weibull probability distribution object using the specified parameter values.

## Input Arguments

### **a** - Scale parameter

1 (default) | positive scalar value

Scale parameter of the Weibull distribution, specified as a positive scalar value.

### Data Types

single | double

### **b** - Shape parameter

1 (default) | positive scalar value

Shape parameter of the Weibull distribution, specified as a positive scalar value.

### Data Types

single | double

## Properties

**a**

Scale parameter of the Weibull distribution, stored as a positive scalar value.

### Data Types

single | double

**b**

Shape parameter of the Weibull distribution, stored as a positive scalar value.

### Data Types

single | double

### DistributionName

Name of the probability distribution, stored as a valid probability distribution name string. This property is read-only.

### Data Types

char

### InputData

Data used for distribution fitting, stored as a structure containing the following:

- data: Data vector used for distribution fitting.
- cens: Censoring vector, or empty if none.
- freq: Frequency vector, or empty if none.

This property is read-only.

### Data Types

single | double

### IsTruncated

Logical flag for truncated distribution, stored as a logical value. If IsTruncated equals 0, the distribution is not truncated.

# prob.WeibullDistribution

---

If `IsTruncated` equals 1, the distribution is truncated. This property is read-only.

## Data Types

logical

## NumParameters

Number of parameters for the probability distribution, stored as a positive integer value. This property is read-only.

## Data Types

single | double

## ParameterCovariance

Covariance matrix of the parameter estimates, stored as a  $p$ -by- $p$  matrix, where  $p$  is the number of parameters in the distribution. The  $(i,j)$  element is the covariance between the estimates of the  $i$ th parameter and the  $j$ th parameter. The  $(i,i)$  element is the estimated variance of the  $i$ th parameter. If parameter  $i$  is fixed rather than estimated by fitting the distribution to data, then the  $(i,i)$  elements of the covariance matrix are 0. This property is read-only.

## Data Types

single | double

## ParameterDescription

Descriptions of distribution parameters, stored as a cell array of strings. Each cell contains a short description of one distribution parameter. This property is read-only.

## Data Types

char

## ParameterIsFixed

Logical flag for fixed parameters, stored as an array of logical values. If 0, the corresponding parameter in the `ParameterNames` array is not fixed. If 1, the corresponding parameter in the `ParameterNames` array is fixed. This property is read-only.

## Data Types

logical

## ParameterNames

Names of distribution parameters, stored as a cell array of strings. This property is read-only.

## Data Types

char

## ParameterValues

Values of distribution parameters, stored as a vector. This property is read-only.

## Data Types

single | double

## Truncation

Truncation interval for the probability distribution, stored as a vector containing the lower and upper truncation boundaries. This property is read-only.

## Data Types

single | double

## Methods

### Inherited Methods

cdf	Cumulative distribution function of probability distribution object
icdf	Inverse cumulative distribution function of probability distribution object
iqr	Interquartile range of probability distribution object
median	Median of probability distribution object

# prob.WeibullDistribution

---

pdf	Probability density function of probability distribution object
random	Generate random numbers from probability distribution object
truncate	Truncate probability distribution object
mean	Mean of probability distribution object
negloglik	Negative loglikelihood of probability distribution object
paramci	Confidence intervals for probability distribution parameters
proflik	Profile likelihood function for probability distribution object
std	Standard deviation of probability distribution object
var	Variance of probability distribution object

## Definitions

### Weibull Distribution

The Weibull distribution is used in reliability and lifetime modeling, and to model the breaking strength of materials.

The Weibull distribution uses the following parameters.



Parameter	Description	Support
a	Scale parameter	$a > 0$
b	Shape parameter	$b > 0$

The probability density function (pdf) is

$$f(x | a, b) = \frac{b}{a} \left(\frac{x}{a}\right)^{b-1} \exp\left\{-\left(\frac{x}{a}\right)^b\right\} ; x \geq 0.$$

## Examples

### Create a Weibull Distribution Object Using Default Parameters

Create a Weibull distribution object using the default parameter values.

```
pd = makedist('Weibull')
```

```
pd =
```

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 1
```

```
B = 1
```

### Create a Weibull Distribution Object Using Specified Parameter Values

Create a Weibull distribution object by specifying the parameter values.

```
pd = makedist('Weibull','a',2,'b',5)
```

```
pd =
```

# prob.WeibullDistribution

---

```
WeibullDistribution
```

```
Weibull distribution
```

```
A = 2
```

```
B = 5
```

Compute the mean of the distribution.

```
m = mean(pd)
```

```
m =
```

```
1.8363
```

## See Also

```
makedist | fitdist | dfittool
```

## Concepts

- “Weibull Distribution” on page B-141
- Class Attributes
- Property Attributes

**Purpose** Wishart random numbers

**Syntax**  
`W = wishrnd(Sigma,df)`  
`W = wishrnd(Sigma,df,D)`  
`[W,D] = wishrnd(Sigma,df)`

**Description** `W = wishrnd(Sigma,df)` generates a random matrix `W` having the Wishart distribution with covariance matrix `Sigma` and with `df` degrees of freedom. The inverse of `W` has the Inverse Wishart distribution with parameters `Tau = inv(Sigma)` and `df` degrees of freedom.

`W = wishrnd(Sigma,df,D)` expects `D` to be the Cholesky factor of `Sigma`. If you call `wishrnd` multiple times using the same value of `Sigma`, it's more efficient to supply `D` instead of computing it each time.

`[W,D] = wishrnd(Sigma,df)` returns `D` so you can provide it as input in future calls to `wishrnd`.

This function defines the parameter `Sigma` so that the mean of the output matrix is `Sigma*df`

**See Also** `iwishrnd`

**How To** • “Wishart Distribution” on page B-144

# TreeBagger.X property

---

**Purpose** X data used to create ensemble

**Description** The X property is a numeric matrix of size Nobs-by-Nvars, where Nobs is the number of observations (rows) and Nvars is the number of variables (columns) in the training data. This matrix contains the predictor (or feature) values.

**Purpose** Create dataset array from data stored in SAS XPORT format file

**Syntax**

```
data = xptread
data = xptread(filename)
[data,missing] = xptread(filename)
xptread(..., 'ReadObsNames', true)
```

**Description** `data = xptread` displays a dialog box for selecting a file, then reads data from the file into a dataset array. The file must be in the SAS XPORT format.

`data = xptread(filename)` retrieves data from a SAS XPORT format file `filename`. The XPORT format allows for 28 missing data types, represented in the file by an upper case letter, '.' or '\_'. `xptread` converts All missing data to NaN values in `data`. However, if you need the specific missing types then you can recover this information by specifying a second output.

`[data,missing] = xptread(filename)` returns a nominal array, `missing`, of the same size as `data` containing the missing data type information from the xport format file. The entries are undefined for values that are not present and are one of '.', '\_', 'A', ..., 'Z' for missing values.

`xptread(..., 'ReadObsNames', true)` treats the first variable in the file as observation names. The default value is false.

`xptread` only supports single data sets per file. `xptread` does not support compressed files.

**Examples** Read in a SAS XPORT format dataset:

```
data = xptread('sample.xpt')
```

**See Also** `dataset` | `dataset.export`

**Purpose** Convert predictor matrix to design matrix

**Syntax**

```
D = x2fx(X,model)
D = x2fx(X,model,categ)
D = x2fx(X,model,categ,catlevels)
```

**Description** `D = x2fx(X,model)` converts a matrix of predictors `X` to a design matrix `D` for regression analysis. Distinct predictor variables should appear in different columns of `X`.

The optional input `model` controls the regression model. By default, `x2fx` returns the design matrix for a linear additive model with a constant term. `model` is one of the following strings:

- 'linear' — Constant and linear terms. This is the default.
- 'interaction' — Constant, linear, and interaction terms
- 'quadratic' — Constant, linear, interaction, and squared terms
- 'purequadratic' — Constant, linear, and squared terms

If `X` has  $n$  columns, the order of the columns of `D` for a full quadratic model is:

- 1 The constant term
- 2 The linear terms (the columns of `X`, in order 1, 2, ...,  $n$ )
- 3 The interaction terms (pairwise products of the columns of `X`, in order (1, 2), (1, 3), ..., (1,  $n$ ), (2, 3), ..., ( $n-1$ ,  $n$ ))
- 4 The squared terms (in order 1, 2, ...,  $n$ )

Other models use a subset of these terms, in the same order.

Alternatively, `model` can be a matrix specifying polynomial terms of arbitrary order. In this case, `model` should have one column for each column in `X` and one row for each term in the model. The entries in any row of `model` are powers for the corresponding columns of `X`. For example, if `X` has columns `X1`, `X2`, and `X3`, then a row `[0 1 2]` in `model`

specifies the term  $(X1.^0) \cdot (X2.^1) \cdot (X3.^2)$ . A row of all zeros in *model* specifies a constant term, which can be omitted.

`D = x2fx(X,model,categ)` treats columns with numbers listed in the vector *categ* as categorical variables. Terms involving categorical variables produce dummy variable columns in *D*. Dummy variables are computed under the assumption that possible categorical levels are completely enumerated by the unique values that appear in the corresponding column of *X*.

`D = x2fx(X,model,categ,catlevels)` accepts a vector *catlevels* the same length as *categ*, specifying the number of levels in each categorical variable. In this case, values in the corresponding column of *X* must be integers in the range from 1 to the specified number of levels. Not all of the levels need to appear in *X*.

## Examples

### Example 1

The following converts 2 predictors *X1* and *X2* (the columns of *X*) into a design matrix for a full quadratic model with terms constant, *X1*, *X2*,  $X1 \cdot X2$ ,  $X1.^2$ , and  $X2.^2$ .

```
X = [1 10
     2 20
     3 10
     4 20
     5 15
     6 15];
```

```
D = x2fx(X,'quadratic')
D =
     1     1    10    10     1    100
     1     2    20    40     4    400
     1     3    10    30     9    100
     1     4    20    80    16    400
     1     5    15    75    25    225
     1     6    15    90    36    225
```

**Example 2**

The following converts 2 predictors X1 and X2 (the columns of X) into a design matrix for a quadratic model with terms constant, X1, X2, X1.\*X2, and X1.^2.

```
X = [1 10
      2 20
      3 10
      4 20
      5 15
      6 15];
model = [0 0
         1 0
         0 1
         1 1
         2 0];

D = x2fx(X,model)
D =
     1     1    10    10     1
     1     2    20    40     4
     1     3    10    30     9
     1     4    20    80    16
     1     5    15    75    25
     1     6    15    90    36
```

**See Also**

[regstats](#) | [rstool](#) | [candexch](#) | [candgen](#) | [cordexch](#) | [rowexch](#)



**Purpose** Y data used to create ensemble

**Description** The Y property is an array of true class labels for classification, or response values for regression. Y can be a numeric column vector, a character matrix, or a cell array of strings.

# zscore

---

**Purpose** Standardized  $z$ -scores

**Syntax**

```
Z = zscore(X)
Z = zscore(X, flag)
Z = zscore(X, flag, dim)
[Z, mu, sigma] = zscore( ___ )
```

**Description** `Z = zscore(X)` returns the  $z$ -score for each element of  $X$  such that columns of  $X$  are centered to have mean 0 and scaled to have standard deviation 1.  $Z$  is the same size as  $X$ .

- If  $X$  is a vector, then  $Z$  is a vector of  $z$ -scores.
- If  $X$  is a matrix, then  $Z$  is a matrix of the same size as  $X$ , and each column of  $Z$  has mean 0 and standard deviation 1.
- For multidimensional arrays,  $z$ -scores in  $Z$  are computed along the first nonsingleton dimension of  $X$ .

`Z = zscore(X, flag)` scales  $X$  using the standard deviation indicated by `flag`.

- If `flag` is 0 (default), then `zscore` scales  $X$  using the sample standard deviation, with  $n - 1$  in the denominator of the standard deviation formula. `zscore(X, 0)` is the same as `zscore(X)`.
- If `flag` is 1, then `zscore` scales  $X$  using the population standard deviation, with  $n$  in the denominator of standard deviation formula.

`Z = zscore(X, flag, dim)` standardizes  $X$  along dimension `dim`. For example, for a matrix  $X$ , if `dim = 1`, then `zscore` uses the means and standard deviations along the columns of  $X$ , if `dim = 2`, then `zscore` uses the means and standard deviations along the rows of  $X$ .

`[Z, mu, sigma] = zscore( ___ )` also returns the means and standard deviations used for centering and scaling, `mu` and `sigma`, respectively. You can use any of the input arguments in the previous syntaxes.

## Input Arguments

### X - Input data

vector | matrix | multidimensional array

Input data, specified as a vector, matrix, or multidimensional array.

### Data Types

double | single

### flag - Indicator for the standard deviation

0 (default) | 1

Indicator for the standard deviation used to compute the *z*-scores, specified as 0 or 1.

- If `flag` is 0 (default), then `zscore` scales *X* using the sample standard deviation. `zscore(X,0)` is the same as `zscore(X)`.
- If `flag` is 1, then `zscore` scales *X* using the population standard deviation.

### dim - Dimension

1 (default) | positive integer

Dimension along which to calculate the *z*-scores of *X*, specified as a positive integer. For example, for a matrix *X*, if `dim` = 1, then `zscore` uses the means and standard deviations along the columns of *X*, if `dim` = 2, then `zscore` uses the means and standard deviations along the rows of *X*.

## Output Arguments

### Z - z-scores

vector | matrix | multidimensional array

*z*-scores, returned as a vector, matrix, or multidimensional array. A vector of *z*-scores has mean 0 and variance 1.

- If *X* is a vector, then *Z* is a vector of *z*-scores.
- If *X* is an array, then `zscore` is an array, with each column or row standardized to have mean 0 and variance 1 (depending on `dim`). If `dim` is not specified, `zscore` standardizes along the first nonsingleton dimension of *X*.

## **mu - Mean**

scalar | vector

Mean of  $X$  used to compute the  $z$ -scores, returned as a scalar or vector.

- If  $X$  is a vector, then  $\mu$  is a scalar.
- If  $X$  is a matrix, then  $\mu$  is a row vector if `zscore` calculates the means along the columns of  $X$  ( $\text{dim} = 1$ ), and a column vector if `zscore` calculates the means along the rows of  $X$  ( $\text{dim} = 2$ ).

## **sigma - Standard deviation**

scalar | vector

Standard deviation of  $X$  used to compute the  $z$ -scores, returned as a scalar or vector.

- If  $X$  is a vector, then  $\sigma$  is a scalar.
- If  $X$  is a matrix, then  $\sigma$  is a row vector if `zscore` calculates the standard deviations along the columns of  $X$  ( $\text{dim} = 1$ ), and a column vector if `zscore` calculates the standard deviations along the rows of  $X$  ( $\text{dim} = 2$ ).

## **Examples**

### **Z-Scores of Two Data Vectors**

Compute and plot the  $z$ -scores of two data vectors, and then compare the results.

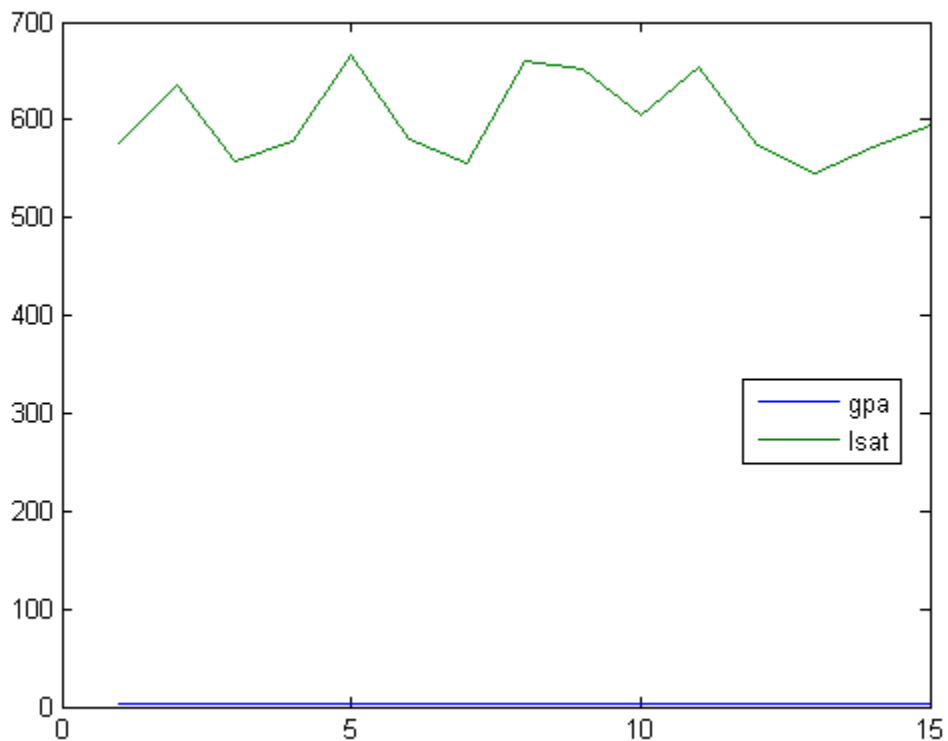
Load the sample data.

```
load('lawdata.mat')
```

Two variables load into the workspace: `gpa` and `lsat`.

Plot both variables on the same axes.

```
plot([gpa,lsat])  
legend('gpa','lsat','Location','East')
```

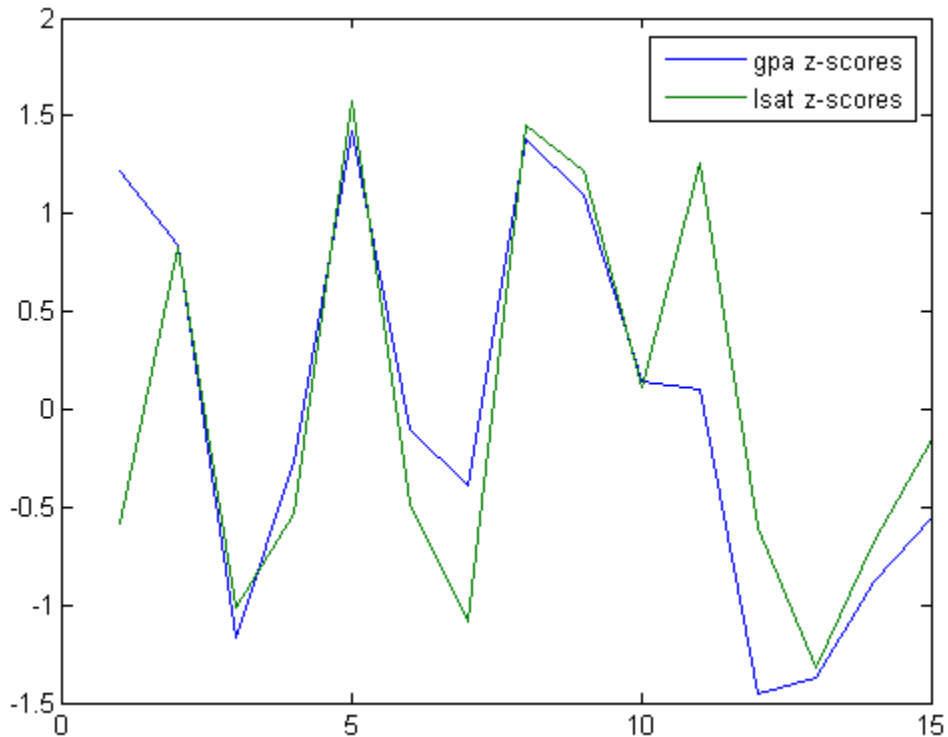


It is difficult to compare these two measures because they are on a very different scale.

Plot the z-scores of gpa and lsat on the same axes.

```
Zgpa = zscore(gpa);  
Zlsat = zscore(lsat);  
plot([Zgpa, Zlsat])  
legend('gpa z-scores', 'lsat z-scores', 'Location', 'Northeast')
```

## zscore



Now, you can see the relative performance of individuals with respect to both their `gpa` and `lsat` results. For example, the third individual's `gpa` and `lsat` results are both one standard deviation below the sample mean. The eleventh individual's `gpa` is around the sample mean but has an `lsat` score almost 1.25 standard deviations above the sample average.

Check the mean and standard deviation of the `z`-scores you created.

```
mean([Zgpa,Zlsat])
```

```
ans =
    1.0e-14 *
    -0.1088    0.0357

std([Zgpa,Zlsat])
```

```
ans =
    1    1
```

By definition,  $z$ -scores of gpa and lsat have mean 0 and standard deviation 1.

### **Z-Scores for a Population vs. Sample**

Load the sample data.

```
load('lawdata.mat')
```

Two variables load into the workspace: gpa and lsat.

Compute the  $z$ -scores of gpa using the population formula for standard deviation.

```
Z1 = zscore(gpa,1); % population formula
Z0 = zscore(gpa,0); % sample formula
disp([Z1 Z0])
```

```
    1.2554    1.2128
    0.8728    0.8432
   -1.2100   -1.1690
   -0.2749   -0.2656
    1.4679    1.4181
   -0.1049   -0.1013
   -0.4024   -0.3888
    1.4254    1.3771
    1.1279    1.0896
```

```
0.1502    0.1451
0.1077    0.1040
-1.5076   -1.4565
-1.4226   -1.3743
-0.9125   -0.8815
-0.5724   -0.5530
```

For a sample from a population, the population standard deviation formula with  $n$  in the denominator corresponds to the maximum likelihood estimate of the population standard deviation, and might be biased. The sample standard deviation formula, on the other hand, is the unbiased estimator of the population standard deviation for a sample.

### Z-Scores of a Data Matrix

Compute  $z$ -scores using the mean and standard deviation computed along the columns or rows of a data matrix.

Load the sample data.

```
load('flu.mat')
```

The dataset array `flu` is loaded in the workplace. `flu` has 52 observations on 11 variables. The first variable contains dates (in weeks). The other variables contain the flu estimates for different regions in the U.S.

Convert the dataset array to a data matrix.

```
flu2 = double(flu(:,2:end));
```

The new data matrix, `flu2`, is a 52-by-10 double data matrix. The rows correspond to the weeks and the columns correspond to the U.S. regions in the data set array `flu`.

Standardize the flu estimate for each region (the *columns* of `flu2`).

```
Z1 = zscore(flu2,[ ],1);
```



You can see the  $z$ -scores in the variable editor by double-clicking on the matrix `Z1` created in the workspace.

Standardize the flu estimate for each week (the *rows* of `flu2`).

```
Z2 = zscore(flu2,[ ],2);
```

### **Z-Scores, Mean, and Standard Deviation**

Return the mean and standard deviation used to compute the  $z$ -scores.

Load the sample data.

```
load('lawdata.mat')
```

Two variables load into the workspace: `gpa` and `lsat`.

Return the  $z$ -scores, mean, and standard deviation of `gpa`.

```
[Z,gpamean,gpastdev] = zscore(gpa)
```

```
Z =
```

```
    1.2128  
    0.8432  
   -1.1690  
   -0.2656  
    1.4181  
   -0.1013  
   -0.3888  
    1.3771  
    1.0896  
    0.1451  
    0.1040  
   -1.4565  
   -1.3743  
   -0.8815  
   -0.5530
```

gpamean =

3.0947

gpastdev =

0.2435

## Definitions

### Z-Score

For a random variable  $X$  with mean  $\mu$  and standard deviation  $\sigma$ , the  $z$ -score of a value  $x$  is

$$z = \frac{(x - \mu)}{\sigma}.$$

For sample data with mean  $\bar{X}$  and standard deviation  $S$ , the  $z$ -score of a data point  $x$  is

$$z = \frac{(x - \bar{X})}{S}.$$

$z$ -scores measure the distance of a data point from the mean in terms of the standard deviation. This is also called *standardization* of data. The standardized data set has mean 0 and standard deviation 1, and retains the shape properties of the original data set (same skewness and kurtosis).

You can use  $z$ -scores to put data on the same scale before further analysis. This lets you to compare two or more data sets with different units.

### Multidimensional Array

A *multidimensional array* is an array with more than two dimensions. For example, if  $X$  is a 1-by-3-by-4 array, then  $X$  is a three-dimensional array.

### First Nonsingleton Dimension

A *first nonsingleton dimension* is the first dimension of an array whose size is not equal to 1. For example, if  $X$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $X$ .

### Sample Standard Deviation

The *sample standard deviation*,  $S$ , is given by

$$S = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{X})^2}{n - 1}}.$$

$S$  is the square root of an unbiased estimator of the variance of the population from which  $X$  is drawn, as long as  $X$  consists of independent, identically distributed samples.

Notice that the denominator in this variance formula is  $n - 1$ .

### Population Standard Deviation

If the data is the entire population of values, then you can use the *population standard deviation*,

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \mu)^2}{n}}.$$

If  $X$  is a random sample from a population, then  $\mu$  is estimated by the sample mean, and  $\sigma$  is the biased maximum likelihood estimator of the population standard deviation.

Notice that the denominator in this variance formula is  $n$ .

## Algorithms

`zscore` returns NaNs for any sample containing NaNs.

`zscore` returns 0s for any sample that is constant (all values are the same). For example, if  $X$  is a vector of the same numeric value, then  $Z$  is a vector of 0s. If  $X$  is a matrix with a column of consisting of the same value, then that column of  $Z$  consists of 0s.

## zscore

---

### See Also

[mean](#) | [std](#)

**Purpose**

`z-test`

**Syntax**

```
h = ztest(x,m,sigma)
h= ztest(x,m,sigma,Name,Value)
[h,p] = ztest( ___ )
[h,p,ci,zval] = ztest( ___ )
```

**Description**

`h = ztest(x,m,sigma)` returns a test decision for the null hypothesis that the data in the vector `x` comes from a normal distribution with mean `m` and a standard deviation `sigma`, using the *z*-test. The alternative hypothesis is that the mean is not `m`. The result `h` is 1 if the test rejects the null hypothesis at the 5% significance level, and 0 otherwise.

`h= ztest(x,m,sigma,Name,Value)` returns a test decision for the *z*-test with additional options specified by one or more name-value pair arguments. For example, you can change the significance level or conduct a one-sided test.

`[h,p] = ztest( ___ )` also returns the *p*-value of the test, using any of the input arguments from previous syntaxes.

`[h,p,ci,zval] = ztest( ___ )` also returns the confidence interval of the population mean, `ci`, and the value of the test statistic, `zval`.

**Input Arguments****`x` - Sample data**

vector | matrix | multidimensional array

Sample data, specified as a vector, matrix, or multidimensional array.

- If `x` is specified as a vector, `ztest` returns a single value for each output argument.
- If `x` is specified as a matrix, `ztest` performs a separate *z*-test along each column of `x` and returns a vector of results.

- If `x` is specified as a multidimensional array, `ztest` works along the first nonsingleton dimension of `x`.

In all cases, `ztest` treats NaN values as missing data and ignores them.

### Data Types

single | double

### **m** - Hypothesized mean

scalar value

Hypothesized mean, specified as a scalar value.

### Data Types

single | double

### **sigma** - Population standard deviation

scalar value

Population standard deviation, specified as a scalar value.

### Data Types

single | double

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Example:** `'Tail','right','Alpha',0.01` specifies a right-tailed hypothesis test at the 1% significance level.

### **'Alpha'** - Significance level

0.05 (default) | scalar value in the range (0,1)

Significance level of the hypothesis test, specified as the comma-separated pair consisting of `'Alpha'` and a scalar value in the range (0,1).

**Example:** 'Alpha',0.01

### **Data Types**

single | double

### **'Dim' - Dimension**

first nonsingleton dimension (default) | positive integer value

Dimension of the input matrix along which to test the means, specified as the comma-separated pair consisting of 'Dim' and a positive integer value. For example, specifying 'Dim',1 tests the column means, while 'Dim',2 tests the row means.

**Example:** 'Dim',2

### **Data Types**

single | double

### **'Tail' - Type of alternative hypothesis**

'both' (default) | 'right' | 'left'

Type of alternative hypothesis to evaluate, specified as the comma-separated pair consisting of 'Tail' and one of the following.

'both'	Test the alternative hypothesis that the population mean is not equal to $m$ .
'right'	Test the alternative hypothesis that the population mean is greater than $m$ .
'left'	Test the alternative hypothesis that the population mean is less than $m$ .

**Example:** 'Tail','right'

## **Output Arguments**

### **h - Hypothesis test result**

1 | 0

Hypothesis test result, returned as a logical value.

- If  $h = 1$ , this indicates the rejection of the null hypothesis at the Alpha significance level.
- If  $h = 0$ , this indicates a failure to reject the null hypothesis at the Alpha significance level.

## **p - p-value**

scalar value in the range [0,1]

$p$ -value of the test, returned as a scalar value in the range [0,1].  $p$  is the probability of observing a test statistic as extreme as, or more extreme than, the observed value under the null hypothesis. Small values of  $p$  cast doubt on the validity of the null hypothesis.

## **ci - Confidence interval**

vector

Confidence interval for the true population mean, returned as a two-element vector containing the lower and upper boundaries of the  $100 \times (1 - \text{Alpha})\%$  confidence interval.

## **zval - Test statistic**

nonnegative scalar value

Test statistic, returned as a nonnegative scalar value.

## **Examples**

### **Test for a Hypothesized Mean**

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;  
x = grades(:,1);
```

Test the null hypothesis that the data comes from a normal distribution with mean  $m = 75$  and standard deviation  $\sigma = 10$ .

```
[h,p,ci,zval] = ztest(x,75,10)
```

```
h =
```



```
0
p =
    0.9927
ci =
    73.2191
    76.7975
zval =
    0.0091
```

The returned value of  $h = 0$  indicates that `ztest` does not reject the null hypothesis at the default 5% significance level.

### One-Sided Hypothesis Test

Load the sample data. Create a vector containing the first column of the students' exam grades data.

```
load examgrades;
x = grades(:,1);
```

Test the null hypothesis that the data comes from a normal distribution with mean  $m = 65$  and standard deviation  $\sigma = 10$ , against the alternative that the mean is greater than 65.

```
[h,p] = ztest(x,65,10,'Tail','right')
h =
    1
p =
    2.8596e-28
```

The returned value of  $h = 1$  indicates that `ztest` rejects the null hypothesis at the default 5% significance level, in favor of the alternative hypothesis that the population mean is greater than 65.

## Definitions

### z-Test

The  $z$ -test is a parametric hypothesis test used to determine whether a sample data set comes from a population with a particular mean. The test assumes the sample data comes from a population with a normal distribution and a known standard deviation.

The test statistic is

$$z = \frac{\bar{x} - \mu}{\sigma / \sqrt{n}},$$

where  $\bar{x}$  is the sample mean,  $\mu$  is the population mean,  $\sigma$  is the population standard deviation, and  $n$  is the sample size. Under the null hypothesis, the test statistic has a standard normal distribution.

### Multidimensional Array

A multidimensional array has more than two dimensions. For example, if  $x$  is a 1-by-3-by-4 array, then  $x$  is a three-dimensional array.

### First Nonsingleton Dimension

The first nonsingleton dimension is the first dimension of an array whose size is not equal to 1. For example, if  $x$  is a 1-by-2-by-3-by-4 array, then the second dimension is the first nonsingleton dimension of  $x$ .

## See Also

`ttest` | `ttest2`

# Sample Data Sets

---

Statistics Toolbox software includes the sample data sets in the following table.

To load a data set into the MATLAB workspace, type:

```
load filename
```

where *filename* is one of the files listed in the table.

Data sets contain individual data variables, description variables with references, and dataset arrays encapsulating the data set and its description, as appropriate.

<b>File</b>	<b>Description of Data Set</b>
acetylene.mat	Chemical reaction data with correlated predictors
arrhythmia.mat	Cardiac arrhythmia data from the UCI machine learning repository
carbig.mat	Measurements of cars, 1970–1982
carsmall.mat	Subset of carbig.mat. Measurements of cars, 1970, 1976, 1982
cereal.mat	Breakfast cereal ingredients
cities.mat	Quality of life ratings for U.S. metropolitan areas
discrim.mat	A version of cities.mat used for discriminant analysis
examgrades.mat	Exam grades on a scale of 0–100
fisheriris.mat	Fisher’s 1936 iris data
flu.mat	Google Flu Trends estimated ILI (influenza-like illness) percentage for various regions of the US, and CDC weighted ILI percentage based on sentinel provider reports
gas.mat	Gasoline prices around the state of Massachusetts in 1993
hald.mat	Heat of cement vs. mix of ingredients
hogg.mat	Bacteria counts in different shipments of milk

<b>File</b>	<b>Description of Data Set</b>
hospital.mat	Simulated hospital data
imports-85.mat	1985 Auto Imports Database from the UCI repository
ionosphere.mat	Ionosphere dataset from the UCI machine learning repository
kmeansdata.mat	Four-dimensional clustered data
lawdata.mat	Grade point average and LSAT scores from 15 law schools
mileage.mat	Mileage data for three car models from two factories
moore.mat	Biochemical oxygen demand on five predictors
morse.mat	Recognition of Morse code distinctions by non-coders
ovariancancer.mat	Grouped observations on 4000 predictors
parts.mat	Dimensional run-out on 36 circular parts
polydata.mat	Sample data for polynomial fitting
popcorn.mat	Popcorn yield by popper type and brand
reaction.mat	Reaction kinetics for Hougen-Watson model
sat.dat	Scholastic Aptitude Test averages by gender and test (table)
sat2.dat	Scholastic Aptitude Test averages by gender and test (csv)
spectra.mat	NIR spectra and octane numbers of 60 gasoline samples
stockreturns.mat	Simulated stock returns



# Distribution Reference

---

- “Bernoulli Distribution” on page B-3
- “Beta Distribution” on page B-5
- “Binomial Distribution” on page B-9
- “Birnbaum-Saunders Distribution” on page B-13
- “Burr Type XII Distribution” on page B-15
- “Chi-Square Distribution” on page B-28
- “Copulas” on page B-30
- “Custom Distributions” on page B-31
- “Exponential Distribution” on page B-32
- “Extreme Value Distribution” on page B-35
- “F Distribution” on page B-41
- “Gamma Distribution” on page B-43
- “Gaussian Distribution” on page B-46
- “Gaussian Mixture Distributions” on page B-47
- “Generalized Extreme Value Distribution” on page B-48
- “Generalized Pareto Distribution” on page B-53
- “Geometric Distribution” on page B-57
- “Hypergeometric Distribution” on page B-60
- “Inverse Gaussian Distribution” on page B-63
- “Inverse Wishart Distribution” on page B-64
- “Johnson System” on page B-66

- “Kernel Distribution” on page B-67
- “Logistic Distribution” on page B-76
- “Loglogistic Distribution” on page B-78
- “Lognormal Distribution” on page B-80
- “Multinomial Distribution” on page B-83
- “Multivariate Gaussian Distribution” on page B-85
- “Multivariate Normal Distribution” on page B-86
- “Multivariate  $t$  Distribution” on page B-91
- “Nakagami Distribution” on page B-97
- “Negative Binomial Distribution” on page B-98
- “Noncentral Chi-Square Distribution” on page B-104
- “Noncentral F Distribution” on page B-106
- “Noncentral  $t$  Distribution” on page B-108
- “Normal Distribution” on page B-110
- “Pareto Distribution” on page B-113
- “Pearson System” on page B-114
- “Piecewise Linear Distribution” on page B-115
- “Poisson Distribution” on page B-118
- “Rayleigh Distribution” on page B-121
- “Rician Distribution” on page B-123
- “Student’s  $t$  Distribution” on page B-124
- “ $t$  Location-Scale Distribution” on page B-126
- “Triangular Distribution” on page B-127
- “Uniform Distribution (Continuous)” on page B-133
- “Uniform Distribution (Discrete)” on page B-139
- “Weibull Distribution” on page B-141
- “Wishart Distribution” on page B-144



## Bernoulli Distribution

### In this section...

“Overview” on page B-3

“Parameters” on page B-3

“Probability Mass Function” on page B-3

“Descriptive Statistics” on page B-3

“Relationship to Other Distributions” on page B-4

### Overview

The Bernoulli distribution is a discrete probability distribution with the only two possible values for the random variable. Each instance of an event with a Bernoulli distribution is called a Bernoulli trial.

### Parameters

The Bernoulli distribution uses the following parameter.

Parameter	Description	Support
$p$	Probability of success	$0 \leq p \leq 1$

### Probability Mass Function

The probability mass function (pmf) is

$$f(x | p) = \begin{cases} 1 - p & , \quad x = 0, \\ p & , \quad x = 1 \end{cases} .$$

### Descriptive Statistics

The mean is

$$\text{mean} = p .$$

The variance is

$$\text{var} = p(1 - p).$$

### **Relationship to Other Distributions**

The Bernoulli distribution is a special case of the binomial distribution, with the number of trials  $n = 1$ . The geometric distribution models the number of Bernoulli trials before the first success (or first failure).

## Beta Distribution

### In this section...

“Definition” on page B-5  
 “Background” on page B-5  
 “Parameters” on page B-6  
 “Example” on page B-7

### Definition

The beta pdf is

$$y = f(x | a, b) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1} I_{(0,1)}(x)$$

where  $B(\cdot)$  is the Beta function. The indicator function  $I_{(0,1)}(x)$  ensures that only values of  $x$  in the range  $(0, 1)$  have nonzero probability.

### Background

The beta distribution describes a family of curves that are unique in that they are nonzero only on the interval  $(0, 1)$ . A more general version of the function assigns parameters to the endpoints of the interval.

The beta cdf is the same as the incomplete beta function.

The beta distribution has a functional relationship with the  $t$  distribution. If  $Y$  is an observation from Student's  $t$  distribution with  $\nu$  degrees of freedom, then the following transformation generates  $X$ , which is beta distributed.

$$X = \frac{1}{2} + \frac{1}{2} \frac{Y}{\sqrt{\nu + Y^2}}$$

If  $Y \sim t(\nu)$ , then  $X \sim \beta\left(\frac{\nu}{2}, \frac{\nu}{2}\right)$

This relationship is used to compute values of the  $t$  cdf and inverse function as well as generating  $t$  distributed random numbers.

## Parameters

Suppose you are collecting data that has hard lower and upper bounds of zero and one respectively. Parameter estimation is the process of determining the parameters of the beta distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the beta pdf. But for the pdf, the parameters are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. Maximum likelihood estimation (MLE) involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `betafit` returns the MLEs and confidence intervals for the parameters of the beta distribution. Here is an example using random numbers from the beta distribution with  $a = 5$  and  $b = 0.2$ .

```
rng('default'); % For reproducibility
r = betarnd(5,0.2,100,1);
[phat, pci] = betafit(r)
```

```
phat =
    7.4911    0.2135
```

```
pci =
    5.0861    0.1744
   11.0334    0.2614
```

The MLE for parameter  $a$  is 7.4911, compared to the true value of 5. The 95% confidence interval for  $a$  goes from 2.8051 to 6.2610, which does not include

the true value. While this is an unlikely result, it does sometimes happen when estimating distribution parameters.

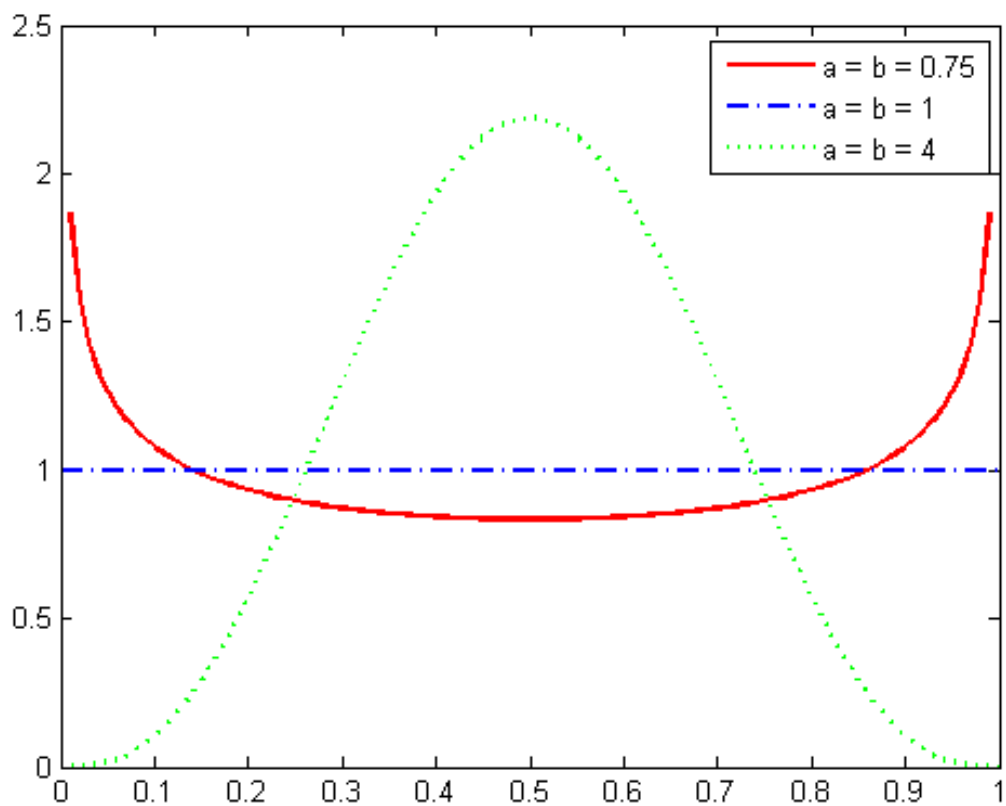
Similarly the MLE for parameter  $b$  is 0.2135, compared to the true value of 0.2. The 95% confidence interval for  $b$  goes from 0.1771 to 0.2832, which does include the true value. In this made-up example you know the “true value.” In experimentation you do not.

## Example

The shape of the beta distribution is quite variable depending on the values of the parameters, as illustrated by the plot below.

```
X = 0:.01:1;
y1 = betapdf(X,0.75,0.75);
y2 = betapdf(X,1,1);
y3 = betapdf(X,4,4);

figure;
plot(X,y1,'Color','r','LineWidth',2);
hold on;
plot(X,y2,'LineStyle','-','Color','b','LineWidth',2);
plot(X,y3,'LineStyle',':','Color','g','LineWidth',2);
legend({'a = b = 0.75','a = b = 1','a = b = 4'},'Location','NorthEast');
hold off;
```



The constant pdf (the flat line) shows that the standard uniform distribution is a special case of the beta distribution, which occurs when  $a = b = 1$ .

# Binomial Distribution

In this section...
“Overview” on page B-9
“Parameters” on page B-9
“Probability Density Function” on page B-9
“Descriptive Statistics” on page B-10
“Relationship to Other Distributions” on page B-10
“Example” on page B-10

## Overview

The binomial distribution models the total number of successes in repeated trials from an infinite population under the following conditions:

- Only two outcomes are possible on each of  $n$  trials.
- The probability of success for each trial is constant.
- All trials are independent of each other.

## Parameters

The binomial distribution uses the following parameters.

Parameter	Description	Support
$N$	Number of trials	positive integer
$p$	Probability of success	$0 \leq p \leq 1$

## Probability Density Function

The probability density function (pdf) is

$$f(x | n, p) = \binom{n}{x} p^x (1-p)^{n-x} \quad ; \quad x = 0, 1, 2, \dots, n,$$

where  $x$  is the number of successes in  $n$  trials of a Bernoulli process with probability of success  $p$ .

## **Descriptive Statistics**

The mean is

$$\text{mean} = np.$$

The variance is

$$\text{var} = np(1 - p).$$

## **Relationship to Other Distributions**

The binomial distribution is a generalization of the Bernoulli distribution, allowing for a number of trials  $n$  greater than 1. The binomial distribution generalizes to the multinomial distribution when there are more than two possible outcomes for each trial.

## **Example**

Suppose you are collecting data from a widget manufacturing process, and you record the number of widgets within specification in each batch of 100. You might be interested in the probability that an individual widget is within specification. Parameter estimation is the process of determining the parameter,  $p$ , of the binomial distribution that fits this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the binomial pdf above. But for the pdf, the parameters ( $n$  and  $p$ ) are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the value of  $p$  that give the highest likelihood given the particular set of data.



The function `binofit` returns the MLEs and confidence intervals for the parameters of the binomial distribution. Here is an example using random numbers from the binomial distribution with  $n = 100$  and  $p = 0.9$ .

```
r = binornd(100,0.9)

r =
    88

[phat, pci] = binofit(r,100)

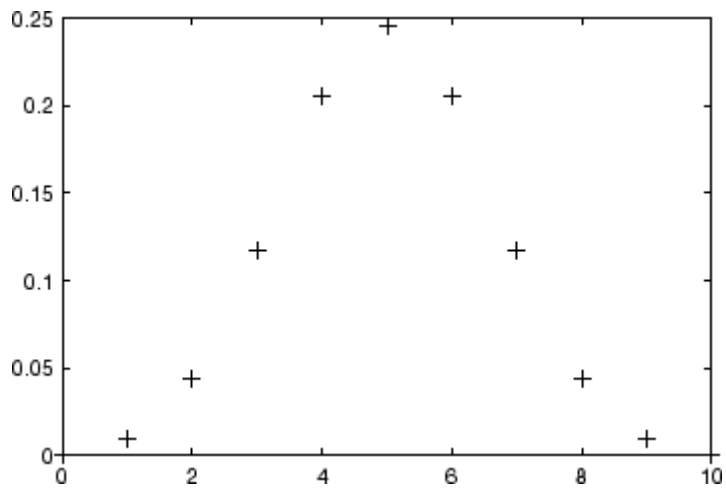
phat =
    0.8800

pci =
    0.7998
    0.9364
```

The MLE for parameter  $p$  is 0.8800, compared to the true value of 0.9. The 95% confidence interval for  $p$  goes from 0.7998 to 0.9364, which includes the true value. In this made-up example you know the “true value” of  $p$ . In experimentation you do not.

The following commands generate a plot of the binomial pdf for  $n = 10$  and  $p = 1/2$ .

```
x = 0:10;
y = binopdf(x,10,0.5);
plot(x,y, '+' )
```



## Birnbau-Saunders Distribution

### In this section...

“Definition” on page B-13

“Background” on page B-13

“Parameters” on page B-14

### Definition

The Birnbau-Saunders distribution has the density function

$$\frac{1}{\sqrt{2\pi}} \exp \left\{ - \frac{\left( \sqrt{x/\beta} - \sqrt{\beta/x} \right)^2}{2\gamma^2} \right\} \left( \frac{\sqrt{x/\beta} + \sqrt{\beta/x}}{2\gamma x} \right)$$

with scale parameter  $\beta > 0$  and shape parameter  $\gamma > 0$ , for  $x > 0$ .

If  $x$  has a Birnbau-Saunders distribution with parameters  $\beta$  and  $\gamma$ , then

$$\frac{\left( \sqrt{x/\beta} - \sqrt{\beta/x} \right)}{\gamma}$$

has a standard normal distribution.

### Background

The Birnbau-Saunders distribution was originally proposed as a lifetime model for materials subject to cyclic patterns of stress and strain, where the ultimate failure of the material comes from the growth of a prominent flaw. In materials science, Miner’s Rule suggests that the damage occurring after  $n$  cycles, at a stress level with an expected lifetime of  $N$  cycles, is proportional to  $n / N$ . Whenever Miner’s Rule applies, the Birnbau-Saunders model is a reasonable choice for a lifetime distribution model.

## **Parameters**

To estimate distribution parameters, use `mle` or the Distribution Fitting app.

## Burr Type XII Distribution

### In this section...

“Definition” on page B-15

“Background” on page B-16

“Parameters” on page B-17

“Fit a Burr Distribution and Draw the cdf” on page B-18

“Compare Lognormal and Burr pdfs” on page B-20

“Burr pdf for Various Parameters” on page B-22

“Survival and Hazard Functions of Burr Distribution” on page B-23

“Divergence of Parameter Estimates” on page B-25

### Definition

The Burr type XII distribution is a three-parameter family of distributions on the positive real line. The cumulative distribution function (cdf) of the Burr distribution is

$$F(x | \alpha, c, k) = 1 - \frac{1}{\left(1 + \left(\frac{x}{\alpha}\right)^c\right)^k}, \quad x > 0, \alpha > 0, c > 0, k > 0,$$

where  $c$  and  $k$  are the shape parameters and  $\alpha$  is the scale parameter. The probability density function (pdf) is

$$f(x | \alpha, c, k) = \frac{\frac{kc}{\alpha} \left(\frac{x}{\alpha}\right)^{c-1}}{\left(1 + \left(\frac{x}{\alpha}\right)^c\right)^{k+1}}, \quad x > 0, \alpha > 0, c > 0, k > 0.$$

The density of the Burr type XII distribution is L-shaped if  $c \leq 1$  and unimodal, otherwise.

## Background

Burr distribution was first discussed by Burr (1942) as a two-parameter family. An additional scale parameter was introduced by Tadikamalla (1980). It is a very flexible distribution family that can express a wide range of distribution shapes. The Burr distribution includes, overlaps, or has as a limiting case, many commonly used distributions such as gamma, lognormal, loglogistic, bell-shaped, and J-shaped beta distributions (but not U-shaped). Some compound distributions also correspond to the Burr distribution. For example, compounding a Weibull distribution with a gamma distribution for its scale parameter results in a Burr distribution. Similarly, compounding an exponential distribution with a gamma distribution for its rate parameter,  $1/\mu$ , also yields a Burr distribution. The Burr distribution also has two asymptotic limiting cases: Weibull and Pareto Type I.

The Burr distribution can fit a wide range of empirical data. Different values of its parameters cover a broad set of skewness and kurtosis. Hence, it is used in various fields such as finance, hydrology, and reliability to model a variety of data types. Examples of data modeled by the Burr distribution are household income, crop prices, insurance risk, travel time, flood levels, and failure data.

The survival and hazard functions of Burr type XII distribution are, respectively,

$$S(x | \alpha, c, k) = \frac{1}{\left[1 + \left(\frac{x}{\alpha}\right)^c\right]^k}$$

and

$$h(x | \alpha, c, k) = \frac{\frac{kc}{\alpha} \left(\frac{x}{\alpha}\right)^{c-1}}{1 + \left(\frac{x}{\alpha}\right)^c}.$$

If  $c > 1$ , the hazard function  $h(x)$  is non-monotonic with a mode at  $x = \alpha(c - 1)^{1/c}$ .

## Parameters

The three-parameter Burr distribution is defined by its scale parameter  $\alpha$  and shape parameters  $c$  and  $k$ . You can estimate the parameters using `mle` or `fitdist`. Both functions support censored data for Burr distribution.

Generate sample data from a Burr distribution with scale parameter 0.5 and shape parameters 2 and 5.

```
rng('default')
R = random('burr',0.5,2,5,1000,1);
```

Estimate the parameters and the confidence intervals.

```
[phat,pci] = mle(R,'distribution','burr')
```

```
phat =
```

```
    0.4154    2.1217    4.0550
```

```
pci =
```

```
    0.2985    1.9560    2.4079
    0.5782    2.3014    6.8288
```

The default 95% confidence intervals for the parameters include the true parameter values.

The three-parameter Burr distribution converges asymptotically to one of the two limiting forms as its parameters diverge:

- If  $k \rightarrow 0$ ,  $c \rightarrow \infty$ ,  $ck = \lambda$ , then the Burr distribution reduces to a two-parameter Pareto distribution with the cdf

$$F_P = 1 - \left(\frac{x}{\alpha}\right)^{-\lambda}, \quad x \geq \alpha.$$

- If  $k \rightarrow \infty$ ,  $\alpha \rightarrow \infty$ ,  $\alpha/k^{1/c} = \theta$ , then the Burr distribution reduces to a two-parameter Weibull distribution with the cdf

$$F_W(x | c, \theta) = 1 - \exp\left[\left(\frac{x}{\alpha}\right)^{-c}\right].$$

If `mle` or `fitdist` detects such divergence, it returns an error message, but informs you of the limiting distribution and corresponding parameter estimates for that distribution.

## Fit a Burr Distribution and Draw the cdf

This example shows how to fit a Burr distribution to data, draw the cdf, and construct a histogram with a Burr distribution fit.

1. Load the sample data.

```
load('arrhythmia.mat')
```

The fifth column in `X` contains a measurement obtained from electrocardiograms, called QRS duration.

2. Fit a Burr distribution to the QRS duration data, and get the parameter estimates.

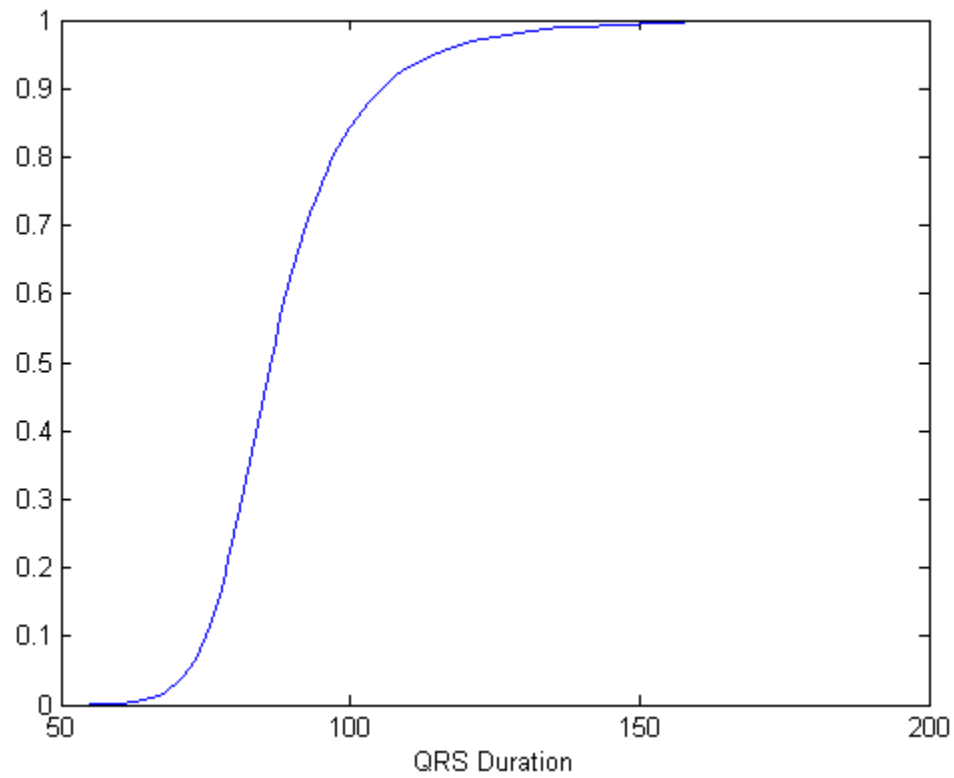
```
PD = fitdist(X(:,5), 'burr');
```

`PD` has the maximum likelihood estimates of the Burr distribution parameters in the property `Param`. The estimates are  $\alpha = 80.4515$ ,  $c = 18.9251$ ,  $k = 0.4492$ .

3. Plot the cdf of the QRS duration data.

```
QRScdf=cdf('burr', sortrows(X(:,5)), 80.4515, 18.9251, 0.4492);  
plot(sortrows(X(:,5)), QRScdf)  
xlabel('QRS Duration')
```

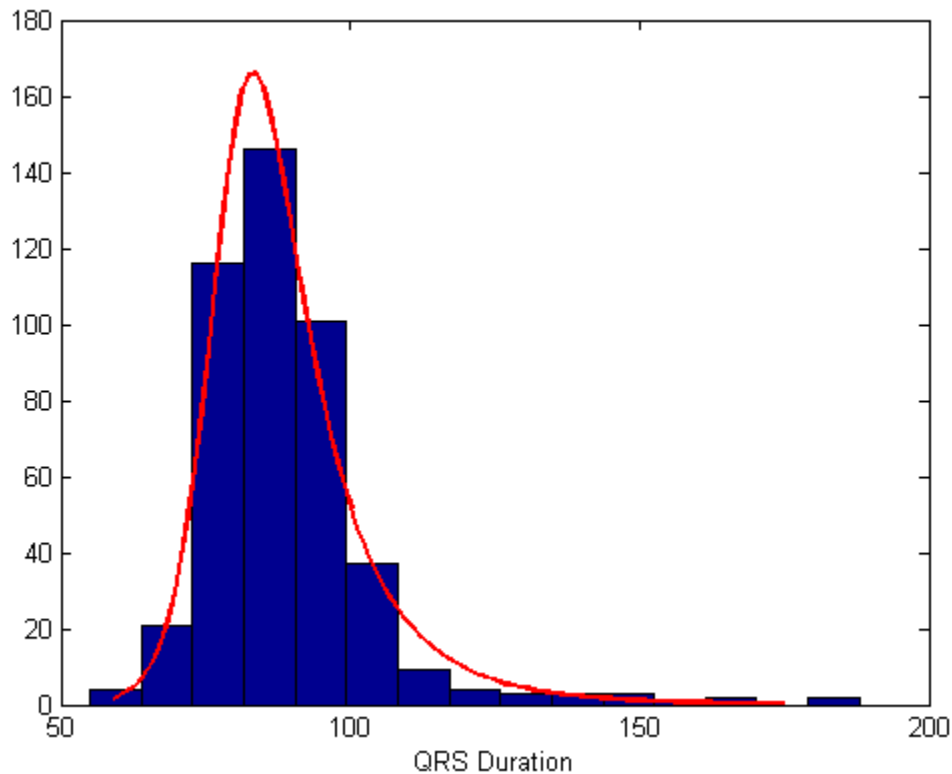




#### **cdf of QRS duration data**

4. Draw the histogram of QRS duration data with 15 bins and the pdf of the Burr distribution fit.

```
histfit(X(:,5),15,'burr')  
xlabel('QRS Duration')
```



**Histogram of QRS data with a Burr distribution fit**

## Compare Lognormal and Burr pdfs

This example shows how to compare the lognormal pdf to the Burr pdf using income data generated from a lognormal distribution.

1. Generate the income data.

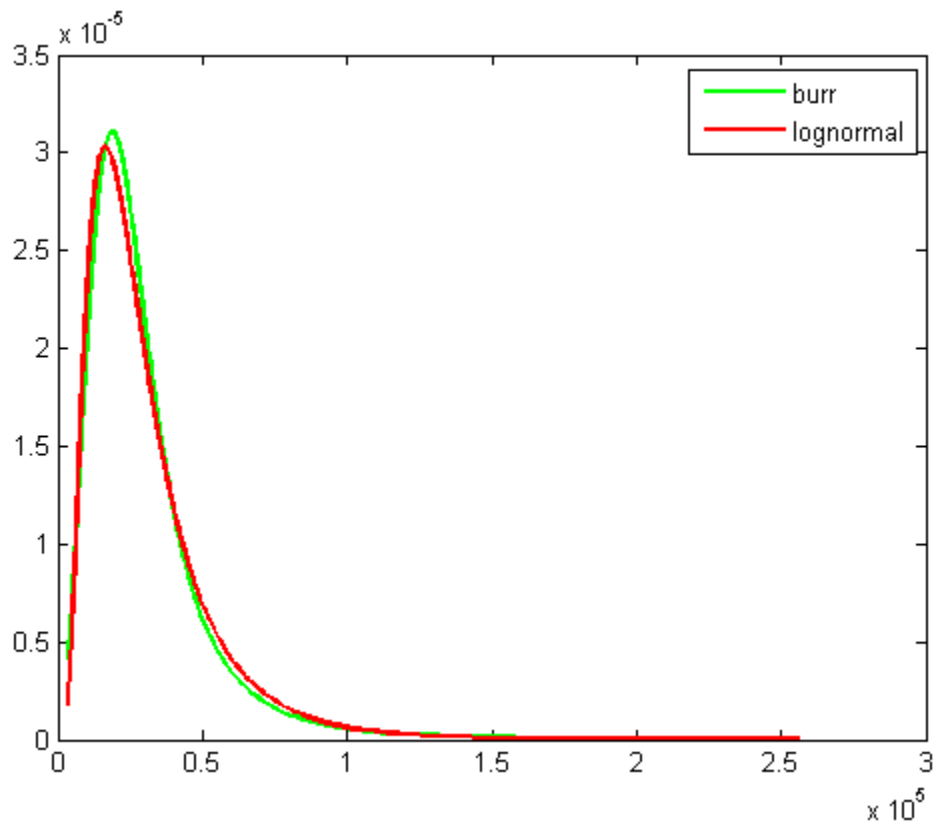
```
rng('default') % for reproducibility  
y = random('logn',log(25000),0.65,500,1);
```

2. Fit a Burr distribution.

```
pd = fitdist(y,'burr');
```

3. Plot both the Burr and lognormal pdfs of income data on the same axis.

```
p = pdf('burr',sortrows(y),26007,2.6374,1.0966);  
p2 = pdf('logn',sortrows(y),log(25000),0.65);  
plot(sortrows(y),p,'g',sortrows(y),p2,'r','LineWidth',2)  
legend('burr','lognormal')
```



**Burr and Lognormal pdfs fitted to income data**

## Burr pdf for Various Parameters

This example shows how to create a variety of shapes for probability density functions of the Burr distribution.

```
X = 0:0.01:5;
c = [0.5 0.95 2 5];
k = [0.5 0.75 2 5];
alpha = [0.5 1 2 5];
colors = ['b';'g';'r';'k']

figure
for i = 1:1:4
pdf1(i,:) = pdf('burr',X,1,c(i),0.5);
pdf2(i,:) = pdf('burr',X,1,2,k(i));
pdf3(i,:) = pdf('burr',X,alpha(i),2,0.5);

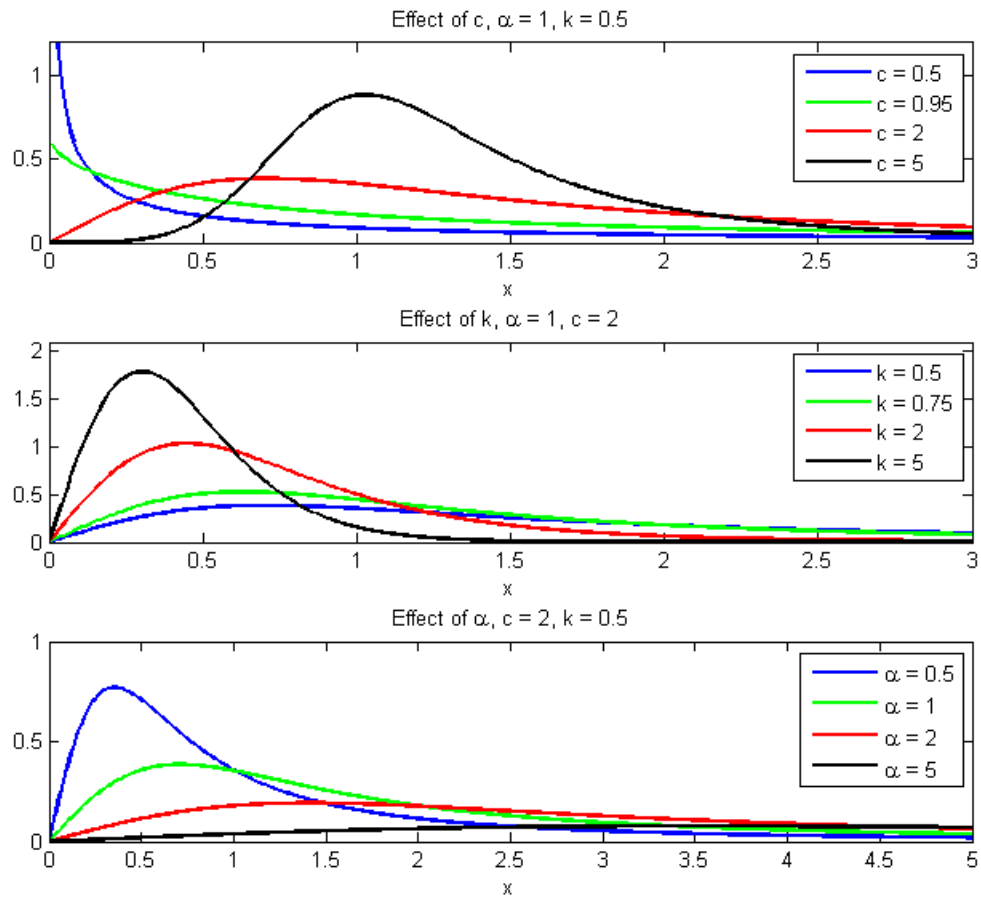
axC = subplot(3,1,1);
pC(i) = plot(X,pdf1(i,:),colors(i),'LineWidth',2);
title('Effect of c, \alpha = 1, k = 0.5'),xlabel('x')
hold on

axK = subplot(3,1,2);
pK(i) = plot(X,pdf2(i,:),colors(i),'LineWidth',2);
title('Effect of k, \alpha = 1, c = 2'),xlabel('x')
hold on

axAlpha = subplot(3,1,3);
pAlpha(i) = plot(X,pdf3(i,:),colors(i),'LineWidth',2);
title('Effect of \alpha, c = 2, k = 0.5'),xlabel('x')
hold on
end

set(axC,'XLim',[0 3],'YLim',[0 1.2]);
set(axK,'XLim',[0 3],'YLim',[0 2.1]);
set(axAlpha,'XLim',[0 5],'YLim',[0 1]);

legend(axC,'c=0.5','c=0.95','c=2','c=5');
legend(axK,'k=0.5','k=0.75','k=2','k=5');
legend(axAlpha,'\alpha=0.5','\alpha=1','\alpha=2','\alpha=5');
```



This figure illustrates how the shape and scale of the Burr distribution changes for different values of its parameters.

## Survival and Hazard Functions of Burr Distribution

This example shows how to find and plot the survival and hazard functions for a sample coming from a Burr distribution.

1. Generate the data.

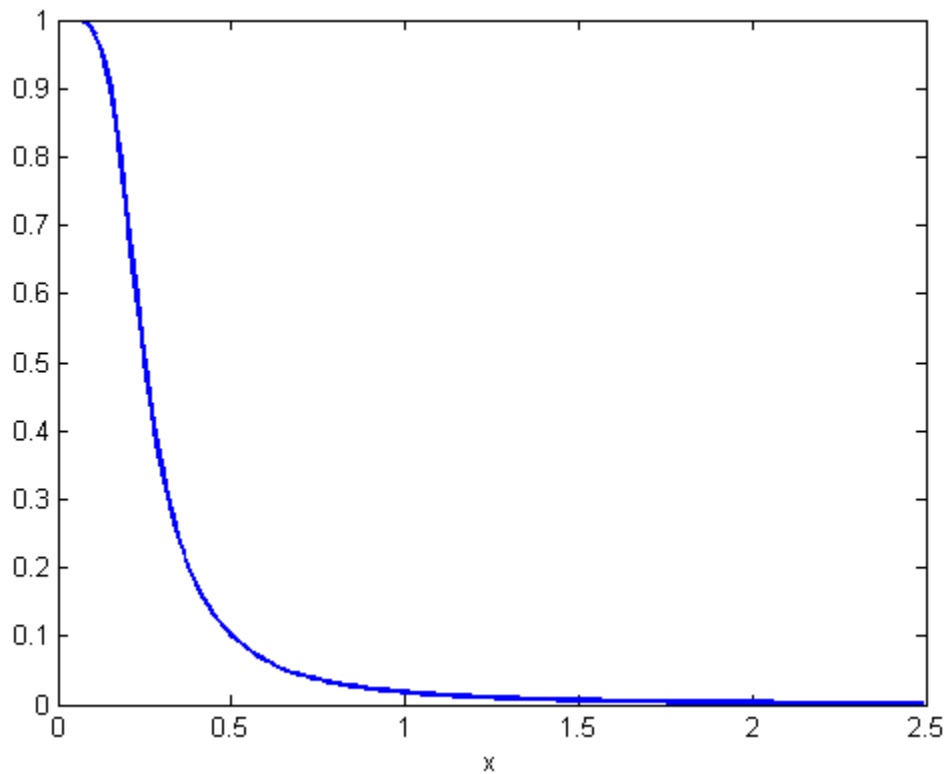
```
X = 0:0.015:2.5;
```

2. Evaluate the pdf and cdf of data in X.

```
Xpdf = pdf('burr',X,0.2,5,0.5);  
Xcdf = cdf('burr',X,0.2,5,0.5);
```

3. Evaluate and plot the survival function of data in X.

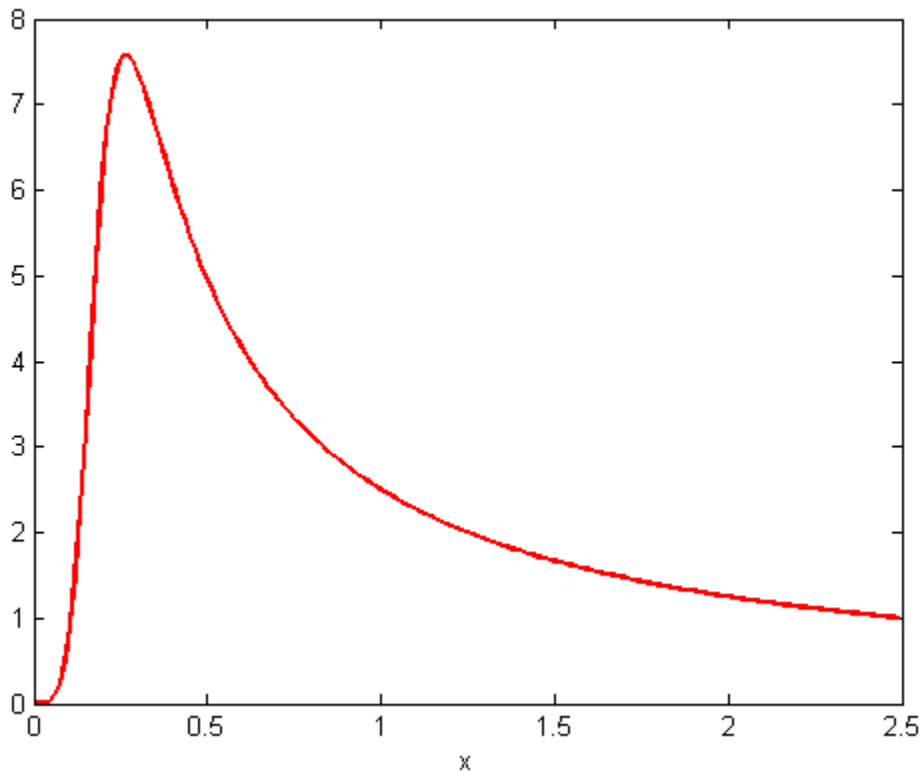
```
S = 1.-Xcdf; % survival function  
plot(X,S,'LineWidth',2)  
xlabel('x')
```



**Survival function**

4. Evaluate and plot the hazard function of data in X.

```
H = Xpdf./S; % hazard function
plot(X,H,'r','LineWidth',2)
xlabel('x')
```



**Hazard function**

### **Divergence of Parameter Estimates**

This example shows how to interpret the display when the parameter estimates diverge when fitting a Burr distribution to input data.

1. Generate sample data from the Weibull distribution with parameters 0.5 and 2.

```
rng('default') % for reproducibility
X = wblrnd(0.5,2,100,1);
```

2. Fit a Burr distribution.

```
PD = fitdist(X,'burr');
```

```
Error using addburr>burrfit (line 566)
The data are not fit by a Burr distribution with finite
parameters. The maximum likelihood fit is provided by the
k->Inf, alpha->Inf limiting form of the Burr distribution: a
Weibull distribution with the parameters below.
```

```
  a (scale): 0.476817
  b (shape): 1.96219
```

```
Error in fitdata (line 24)
      p = F(x,fixedparams{:},0.05,opts{:});
```

```
Error in ProbDistUnivParam.fit (line 94)
pd = fitdata(pd,spec,x,cens,freq,fixedparams,options);
```

```
Error in fitdist (line 124)
      pd =
      ProbDistUnivParam.fit(x,distname,'cens',cens,'freq',freq,args{:});
```

The error message tells you that the Weibull family seems to fit the data better and gives you the parameter estimates from a Weibull fit. You can use those estimates directly. If you need covariance estimates for the parameters or other information about the fit, you can refit a Weibull distribution to the data.

3. Fit a Weibull distribution to the data and find the confidence intervals for the parameter estimates.

```
PD = fitdist(X,'weibull');
paramci(PD)
```

```
ans =
```



0.4291      1.6821  
0.5298      2.2890

These are the 95% confidence intervals of the parameter estimates for the Weibull distribution fit.

## References

- [1] Burr, Irving W. “Cumulative frequency functions.” *The Annals of Mathematical Statistics*, Vol. 13, Number 2, 1942, pp. 215–232.
- [2] Tadikamalla, Pandu R. “A look at the Burr and related distributions.” *International Statistical Review*, Vol. 48, Number 3, 1980, pp. 337–344.
- [3] Rodriguez, Robert N. “A guide to the Burr type XII distributions.” *Biometrika*, Vol. 64, Number 1, 1977, pp. 129–134.
- [4] AL-Hussaini, Essam K. “A characterization of the Burr type XII distribution”. *Appl. Math. Lett.* Vol. 4, Number 1, 1991, pp. 59–61.
- [5] Grammig, Joachim and Kai-Oliver Maurer. “Non-monotonic hazard functions and the autoregressive conditional duration model.” *Econometrics Journal*, Vol. 3, 2000, pp. 16–38.

## Chi-Square Distribution

### In this section...

“Definition” on page B-28

“Background” on page B-28

“Example” on page B-29

### Definition

The  $\chi^2$  pdf is

$$y = f(x | v) = \frac{x^{(v-2)/2} e^{-x/2}}{2^{v/2} \Gamma(v/2)}$$

where  $\Gamma(\cdot)$  is the Gamma function, and  $v$  is the degrees of freedom.

### Background

The  $\chi^2$  distribution is a special case of the gamma distribution where  $b = 2$  in the equation for gamma distribution below.

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}}$$

The  $\chi^2$  distribution gets special attention because of its importance in normal sampling theory. If a set of  $n$  observations is normally distributed with variance  $\sigma^2$ , and  $s^2$  is the sample standard deviation, then

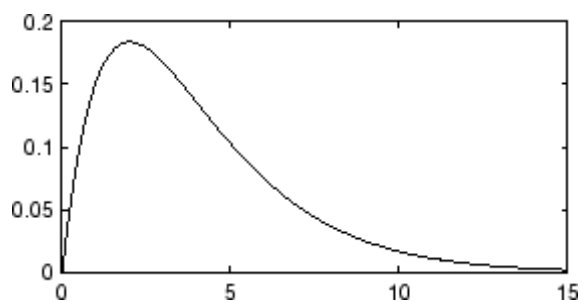
$$\frac{(n-1)s^2}{\sigma^2} \sim \chi^2(n-1)$$

This relationship is used to calculate confidence intervals for the estimate of the normal parameter  $\sigma^2$  in the function `normfit`.

## Example

The  $\chi^2$  distribution is skewed to the right especially for few degrees of freedom ( $\nu$ ). The plot shows the  $\chi^2$  distribution with four degrees of freedom.

```
x = 0:0.2:15;  
y = chi2pdf(x,4);  
plot(x,y)
```



## **Copulas**

See “Copulas: Generate Correlated Samples” on page 5-120.

## Custom Distributions

User-defined custom distributions, created using files and function handles, are supported by the Statistics Toolbox functions `pdf`, `cdf`, `icdf`, and `mle`, and the Statistics Toolbox Distribution Fitting app.

## Exponential Distribution

### In this section...

“Definition” on page B-32

“Background” on page B-32

“Parameters” on page B-32

“Example” on page B-33

### Definition

The exponential pdf is

$$y = f(x | \mu) = \frac{1}{\mu} e^{-\frac{x}{\mu}}$$

### Background

Like the chi-square distribution, the exponential distribution is a special case of the gamma distribution (obtained by setting  $\alpha = 1$ )

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}}$$

where  $\Gamma(\cdot)$  is the Gamma function.

The exponential distribution is special because of its utility in modeling events that occur randomly over time. The main application area is in studies of lifetimes.

### Parameters

Suppose you are stress testing light bulbs and collecting data on their lifetimes. You assume that these lifetimes follow an exponential distribution. You want to know how long you can expect the average light bulb to last.

Parameter estimation is the process of determining the parameters of the exponential distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the exponential pdf above. But for the pdf, the parameters are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `expfit` returns the MLEs and confidence intervals for the parameters of the exponential distribution. Here is an example using random numbers from the exponential distribution with  $\mu = 700$ .

```
lifetimes = exprnd(700,100,1);
[muhat, muc_i] = expfit(lifetimes)
```

```
muhat =
```

```
672.8207
```

```
muc_i =
```

```
547.4338
```

```
810.9437
```

The MLE for parameter  $\mu$  is 672, compared to the true value of 700. The 95% confidence interval for  $\mu$  goes from 547 to 811, which includes the true value.

In the life tests you do not know the true value of  $\mu$  so it is nice to have a confidence interval on the parameter to give a range of likely values.

## Example

For exponentially distributed lifetimes, the probability that an item will survive an extra unit of time is independent of the current age of the item. The example shows a specific case of this special property.

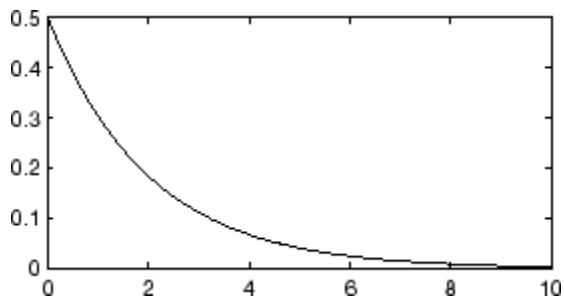
```
l = 10:10:60;
lpd = l+0.1;
```

```
deltap = (expcdf(1pd,50)-expcdf(1,50))./(1-expcdf(1,50))
```

```
deltap =  
    0.0020    0.0020    0.0020    0.0020    0.0020    0.0020
```

The following commands generate a plot of the exponential pdf with its parameter (and mean),  $\mu$ , set to 2.

```
x = 0:0.1:10;  
y = exppdf(x,2);  
plot(x,y)
```





## Extreme Value Distribution

### In this section...

“Definition” on page B-35

“Background” on page B-35

“Parameters” on page B-37

“Example” on page B-38

### Definition

The probability density function for the extreme value distribution with location parameter  $\mu$  and scale parameter  $\sigma$  is

$$y = f(x | \mu, \sigma) = \sigma^{-1} \exp\left(\frac{x - \mu}{\sigma}\right) \exp\left(-\exp\left(\frac{x - \mu}{\sigma}\right)\right)$$

This form of the probability density function is suitable for modeling the minimum value. To model the maximum value, use the negative of the original values.

If  $T$  has a Weibull distribution with parameters  $a$  and  $b$ , then  $\log T$  has an extreme value distribution with parameters  $\mu = \log a$  and  $\sigma = 1/b$ .

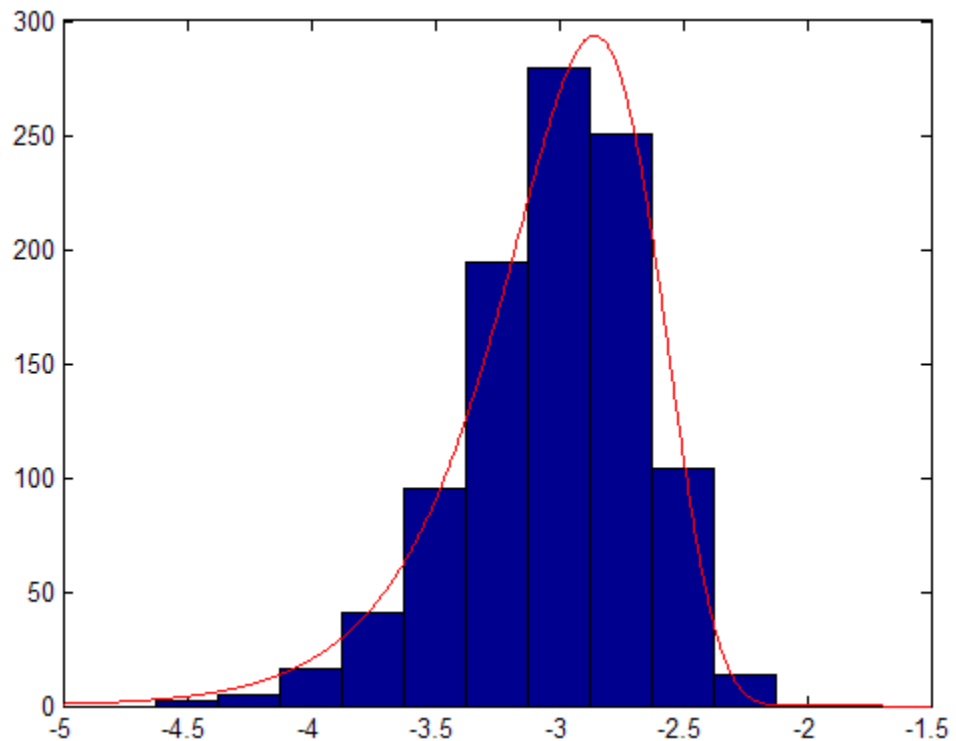
### Background

Extreme value distributions are often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. The extreme value distribution is appropriate for modeling the smallest value from a distribution whose tails decay exponentially fast, for example, the normal distribution. It can also model the largest value from a distribution, such as the normal or exponential distributions, by using the negative of the original values.

For example, the following fits an extreme value distribution to minimum values taken over 1000 sets of 500 observations from a normal distribution:

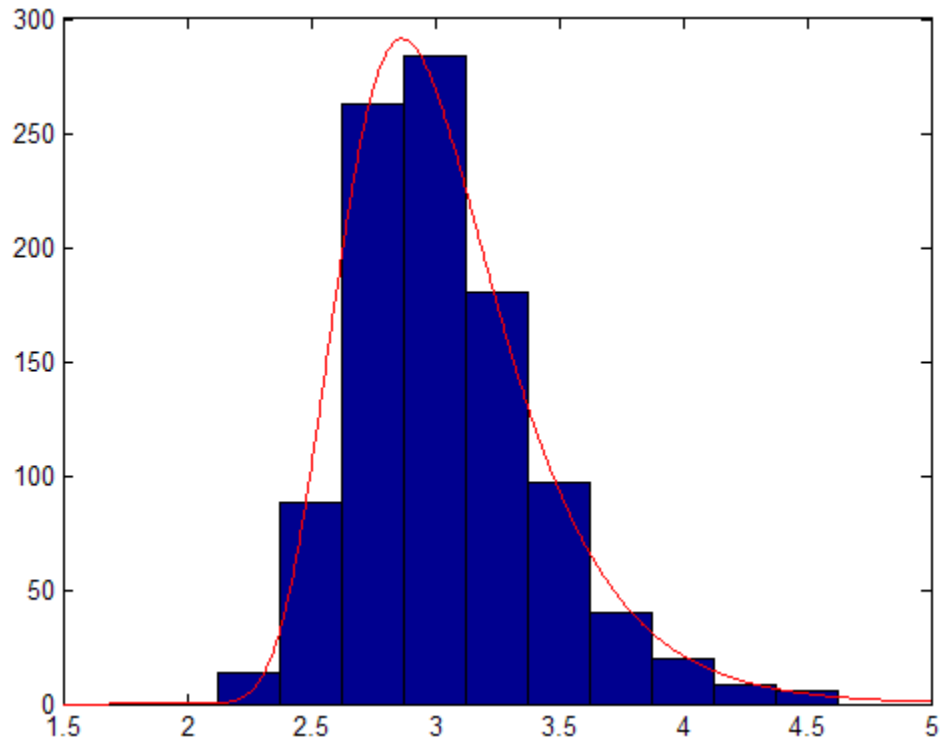
```
xMinima = min(randn(1000,500), [], 2);
```

```
paramEstsMinima = evfit(xMinima);  
y = linspace(-5,-1.5,1001);  
hist(xMinima,-4.75:.25:-1.75);  
p = evpdf(y,paramEstsMinima(1),paramEstsMinima(2));  
line(y,.25*length(xMinima)*p,'color','r')
```



The following fits an extreme value distribution to the maximum values in each set of observations:

```
xMaxima = max(randn(1000,500), [], 2);  
paramEstsMaxima = evfit(-xMaxima);  
y = linspace(1.5,5,1001);  
hist(xMaxima,1.75:.25:4.75);  
p = evpdf(-y,paramEstsMaxima(1),paramEstsMaxima(2));  
line(y,.25*length(xMaxima)*p,'color','r')
```



Although the extreme value distribution is most often used as a model for extreme values, you can also use it as a model for other types of continuous data. For example, extreme value distributions are closely related to the Weibull distribution. If  $T$  has a Weibull distribution, then  $\log(T)$  has a type 1 extreme value distribution.

## Parameters

The function `evfit` returns the maximum likelihood estimates (MLEs) and confidence intervals for the parameters of the extreme value distribution. The following example shows how to fit some sample data using `evfit`, including estimates of the mean and variance from the fitted distribution.

Suppose you want to model the size of the smallest washer in each batch of 1000 from a manufacturing process. If you believe that the sizes are

independent within and between each batch, you can fit an extreme value distribution to measurements of the minimum diameter from a series of eight experimental batches. The following code returns the MLEs of the distribution parameters as `parmhat` and the confidence intervals as the columns of `parmci`.

```
x = [19.774 20.141 19.44 20.511 21.377 19.003 19.66 18.83];  
[parmhat, parmci] = evfit(x)
```

```
parmhat =  
    20.2506    0.8223
```

```
parmci =  
    19.644 0.49861  
    20.857 1.3562
```

You can find mean and variance of the extreme value distribution with these parameters using the function `evstat`.

```
[meanfit, varfit] = evstat(parmhat(1),parmhat(2))
```

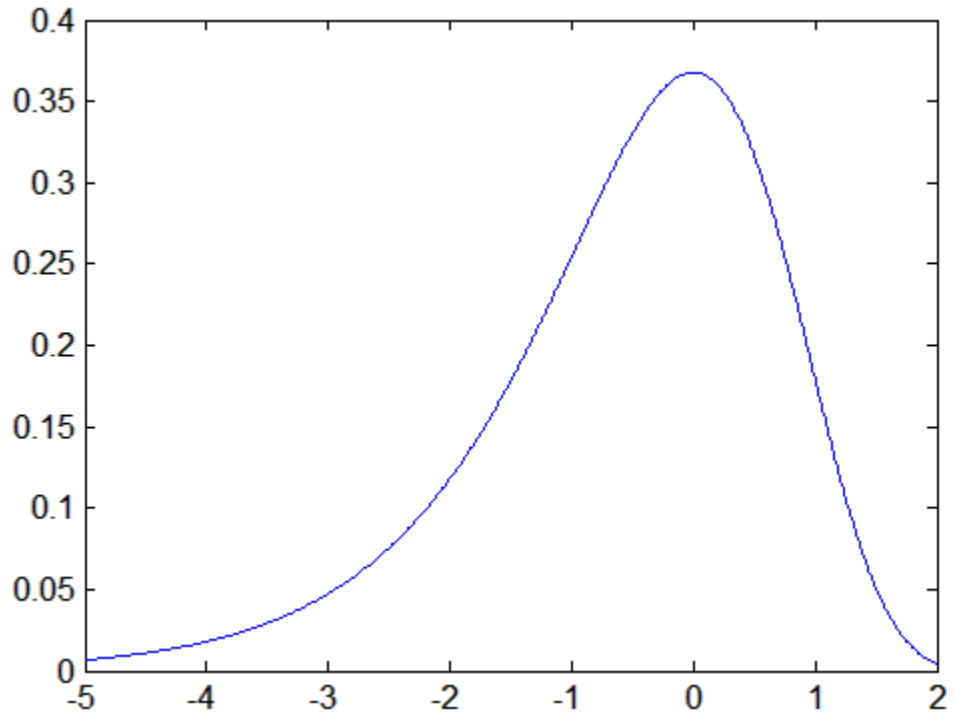
```
meanfit =  
    19.776
```

```
varfit =  
    1.1123
```

## Example

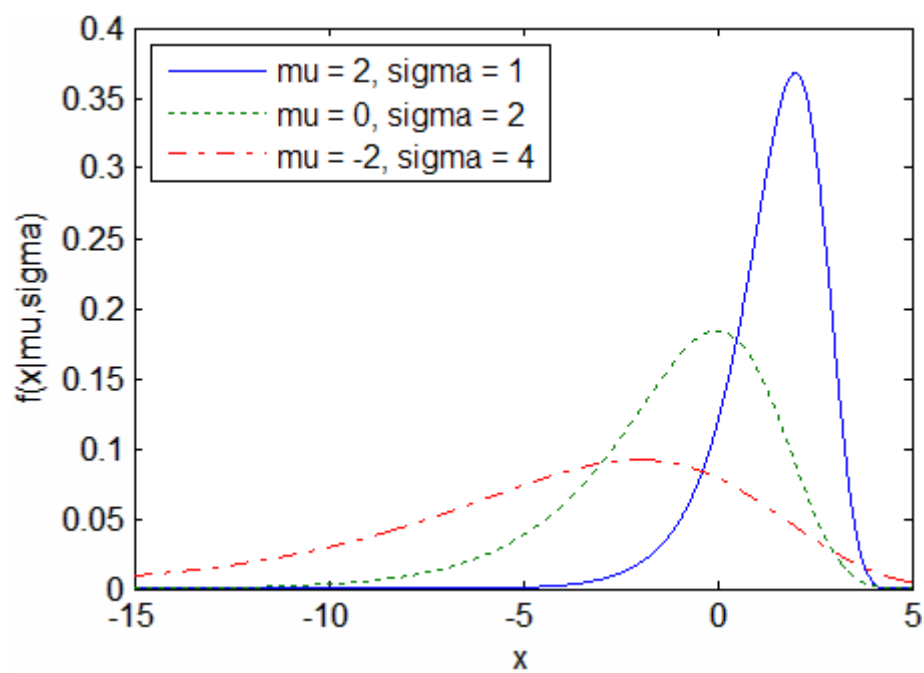
The following code generates a plot of the pdf for the extreme value distribution.

```
t = [-5:.01:2];  
y = evpdf(t);  
plot(t,y)
```



The extreme value distribution is skewed to the left, and its general shape remains the same for all parameter values. The location parameter,  $\mu$ , shifts the distribution along the real line, and the scale parameter,  $\sigma$ , expands or contracts the distribution. This example plots the probability function for different combinations of  $\mu$  and  $\sigma$ .

```
x = -15:.01:5;
plot(x,evpdf(x,2,1),'-', ...
      x,evpdf(x,0,2),':', ...
      x,evpdf(x,-2,4),'-.');
legend({'mu = 2, sigma = 1', ...
       'mu = 0, sigma = 2', ...
       'mu = -2, sigma = 4'}, ...
       'Location','NW')
xlabel('x')
ylabel('f(x|mu,sigma)')
```



## F Distribution

### In this section...

“Definition” on page B-41

“Background” on page B-41

“Example” on page B-42

### Definition

The pdf for the  $F$  distribution is

$$y = f(x | v_1, v_2) = \frac{\Gamma\left[\frac{(v_1 + v_2)}{2}\right] \left(\frac{v_1}{v_2}\right)^{\frac{v_1}{2}} x^{\frac{v_1-2}{2}}}{\Gamma\left(\frac{v_1}{2}\right) \Gamma\left(\frac{v_2}{2}\right) \left[1 + \left(\frac{v_1}{v_2}\right)x\right]^{\frac{v_1+v_2}{2}}}$$

where  $\Gamma(\cdot)$  is the Gamma function.

### Background

The  $F$  distribution has a natural relationship with the chi-square distribution. If  $\chi_1$  and  $\chi_2$  are both chi-square with  $v_1$  and  $v_2$  degrees of freedom respectively, then the statistic  $F$  below is  $F$ -distributed.

$$F(v_1, v_2) = \frac{\chi_1 / v_1}{\chi_2 / v_2}$$

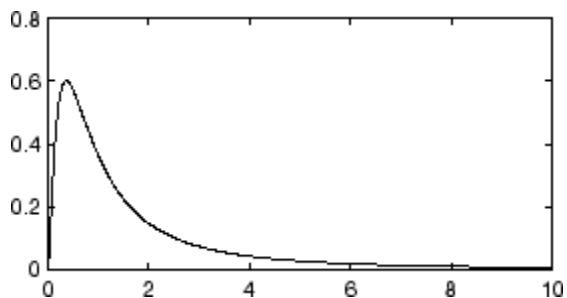
The two parameters,  $v_1$  and  $v_2$ , are the numerator and denominator degrees of freedom. That is,  $v_1$  and  $v_2$  are the number of independent pieces of information used to calculate  $\chi_1$  and  $\chi_2$ , respectively.

## Example

The most common application of the  $F$  distribution is in standard tests of hypotheses in analysis of variance and regression.

The plot shows that the  $F$  distribution exists on the positive real numbers and is skewed to the right.

```
x = 0:0.01:10;  
y = fpdf(x,5,3);  
plot(x,y)
```





## Gamma Distribution

### In this section...

“Definition” on page B-43

“Background” on page B-43

“Parameters” on page B-44

“Example” on page B-45

### Definition

The gamma pdf is

$$y = f(x | a, b) = \frac{1}{b^a \Gamma(a)} x^{a-1} e^{-\frac{x}{b}}$$

where  $\Gamma(\cdot)$  is the Gamma function.

### Background

The gamma distribution models sums of exponentially distributed random variables.

The gamma distribution family is based on two parameters. The chi-square and exponential distributions, which are children of the gamma distribution, are one-parameter distributions that fix one of the two gamma parameters.

The gamma distribution has the following relationship with the incomplete Gamma function.

$$f(x | a, b) = \text{gammainc}\left(\frac{x}{b}, a\right)$$

When  $a$  is large, the gamma distribution closely approximates a normal distribution with the advantage that the gamma distribution has density only for positive real numbers.

## Parameters

Suppose you are stress testing computer memory chips and collecting data on their lifetimes. You assume that these lifetimes follow a gamma distribution. You want to know how long you can expect the average computer memory chip to last. Parameter estimation is the process of determining the parameters of the gamma distribution that fit this data best in some sense.

One popular criterion of goodness is to maximize the likelihood function. The likelihood has the same form as the gamma pdf above. But for the pdf, the parameters are known constants and the variable is  $x$ . The likelihood function reverses the roles of the variables. Here, the sample values (the  $x$ 's) are already observed. So they are the fixed constants. The variables are the unknown parameters. MLE involves calculating the values of the parameters that give the highest likelihood given the particular set of data.

The function `gamfit` returns the MLEs and confidence intervals for the parameters of the gamma distribution. Here is an example using random numbers from the gamma distribution with  $\alpha = 10$  and  $b = 5$ .

```
lifetimes = gamrnd(10,5,100,1);  
[phat, pci] = gamfit(lifetimes)
```

```
phat =
```

```
    10.9821    4.7258
```

```
pci =
```

```
    7.4001    3.1543  
   14.5640    6.2974
```

Note  $\text{phat}(1) = \hat{a}$  and  $\text{phat}(2) = \hat{b}$ . The MLE for parameter  $a$  is 10.98, compared to the true value of 10. The 95% confidence interval for  $a$  goes from 7.4 to 14.6, which includes the true value.

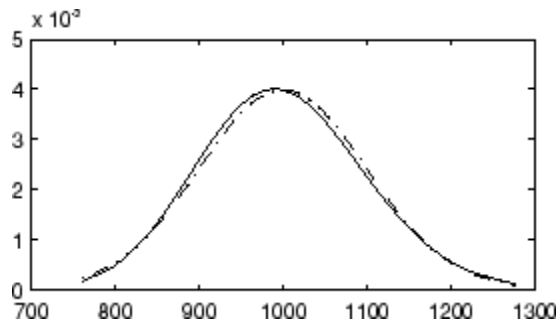
Similarly the MLE for parameter  $b$  is 4.7, compared to the true value of 5. The 95% confidence interval for  $b$  goes from 3.2 to 6.3, which also includes the true value.

In the life tests you do not know the true value of  $a$  and  $b$  so it is nice to have a confidence interval on the parameters to give a range of likely values.

### Example

In the example the gamma pdf is plotted with the solid line. The normal pdf has a dashed line type.

```
x = gaminv( (0.005:0.01:0.995), 100, 10);  
y = gampdf(x, 100, 10);  
y1 = normpdf(x, 1000, 100);  
plot(x, y, '- ', x, y1, '- -')
```



## **Gaussian Distribution**

See “Normal Distribution” on page B-110.

## **Gaussian Mixture Distributions**

See the discussion of the `gmdistribution` class in “Gaussian Mixture Models” on page 5-110.

## Generalized Extreme Value Distribution

### In this section...

“Definition” on page B-48

“Background” on page B-48

“Parameters” on page B-49

“Example” on page B-50

### Definition

The probability density function for the generalized extreme value distribution with location parameter  $\mu$ , scale parameter  $\sigma$ , and shape parameter  $k \neq 0$  is

$$y = f(x | k, \mu, \sigma) = \left(\frac{1}{\sigma}\right) \exp\left[-\left(1 + k \frac{(x - \mu)}{\sigma}\right)^{\frac{1}{k}}\right] \left(1 + k \frac{(x - \mu)}{\sigma}\right)^{-1 - \frac{1}{k}}$$

for

$$1 + k \frac{(x - \mu)}{\sigma} > 0$$

$k > 0$  corresponds to the Type II case, while  $k < 0$  corresponds to the Type III case. For  $k = 0$ , corresponding to the Type I case, the density is

$$y = f(x | 0, \mu, \sigma) = \left(\frac{1}{\sigma}\right) \exp\left(-\exp\left(-\frac{(x - \mu)}{\sigma}\right) - \frac{(x - \mu)}{\sigma}\right)$$

### Background

Like the extreme value distribution, the generalized extreme value distribution is often used to model the smallest or largest value among a large set of independent, identically distributed random values representing measurements or observations. For example, you might have batches of 1000 washers from a manufacturing process. If you record the size of the largest

washer in each batch, the data are known as block maxima (or minima if you record the smallest). You can use the generalized extreme value distribution as a model for those block maxima.

The generalized extreme value combines three simpler distributions into a single form, allowing a continuous range of possible shapes that includes all three of the simpler distributions. You can use any one of those distributions to model a particular dataset of block maxima. The generalized extreme value distribution allows you to “let the data decide” which distribution is appropriate.

The three cases covered by the generalized extreme value distribution are often referred to as the Types I, II, and III. Each type corresponds to the limiting distribution of block maxima from a different class of underlying distributions. Distributions whose tails decrease exponentially, such as the normal, lead to the Type I. Distributions whose tails decrease as a polynomial, such as Student’s  $t$ , lead to the Type II. Distributions whose tails are finite, such as the beta, lead to the Type III.

Types I, II, and III are sometimes also referred to as the Gumbel, Frechet, and Weibull types, though this terminology can be slightly confusing. The Type I (Gumbel) and Type III (Weibull) cases actually correspond to the mirror images of the usual Gumbel and Weibull distributions, for example, as computed by the functions `evcdf` and `evfit`, or `wblcdf` and `wblfit`, respectively. Finally, the Type II (Frechet) case is equivalent to taking the reciprocal of values from a standard Weibull distribution.

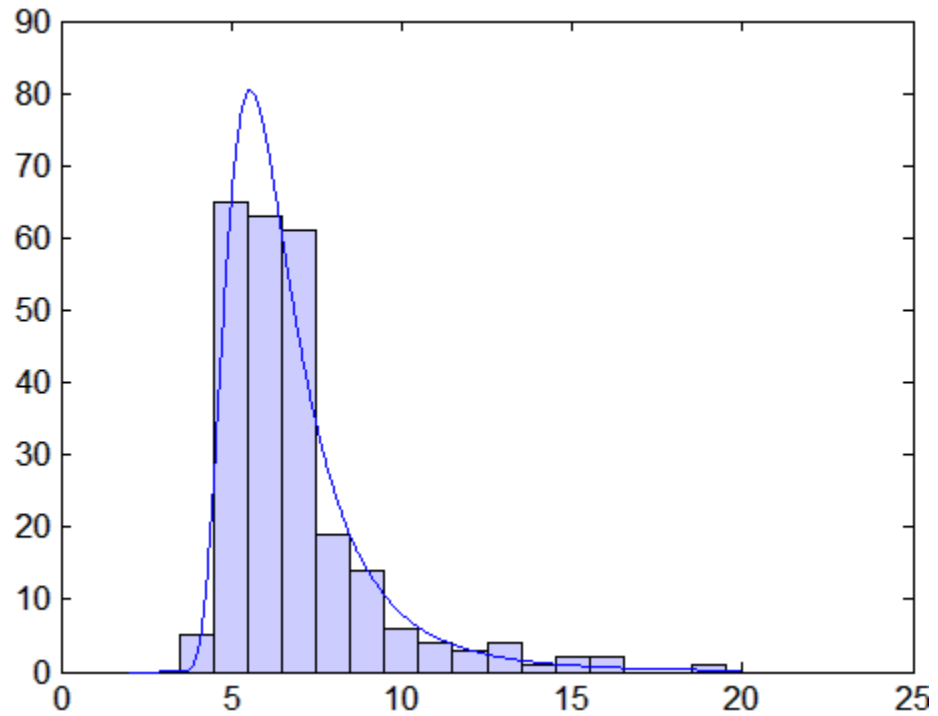
## Parameters

If you generate 250 blocks of 1000 random values drawn from Student’s  $t$  distribution with 5 degrees of freedom, and take their maxima, you can fit a generalized extreme value distribution to those maxima.

```
blocksize = 1000;
nblocks = 250;
t = trnd(5,blocksize,nblocks);
x = max(t); % 250 column maxima
paramEsts = gevfit(x)
paramEsts =
    0.2438    1.1760    5.8045
```

Notice that the shape parameter estimate (the first element) is positive, which is what you would expect based on block maxima from a Student's  $t$  distribution.

```
hist(x,2:20);  
set(get(gca,'child'),'FaceColor',[.8 .8 1])  
xgrid = linspace(2,20,1000);  
line(xgrid,nblocks*...  
      gevpdf(xgrid,paramEsts(1),paramEsts(2),paramEsts(3)));
```



### Example

The following code generates examples of probability density functions for the three basic forms of the generalized extreme value distribution.

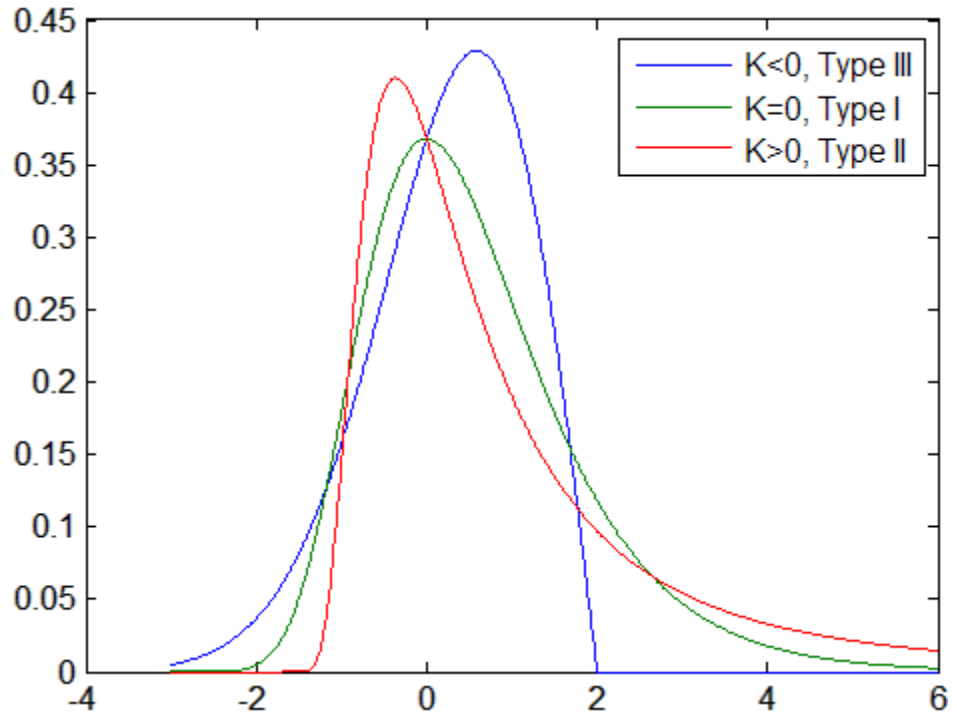
```
x = linspace(-3,6,1000);
```



```

y1 = gevpdf(x, -.5, 1, 0);
y2 = gevpdf(x, 0, 1, 0);
y3 = gevpdf(x, .5, 1, 0);
plot(x, y1, '- ', x, y2, '- ', x, y3, '- ')
legend({'K<0, Type III' 'K=0, Type I' 'K>0, Type II'});

```



Notice that for  $k > 0$ , the distribution has zero probability density for  $x$  such that

$$x < -\frac{\sigma}{k} + \mu$$

For  $k < 0$ , the distribution has zero probability density for

$$x > -\frac{\sigma}{k} + \mu$$

For  $k = 0$ , there is no upper or lower bound.

## Generalized Pareto Distribution

### In this section...

“Definition” on page B-53

“Background” on page B-53

“Parameters” on page B-54

“Example” on page B-55

### Definition

The probability density function for the generalized Pareto distribution with shape parameter  $k \neq 0$ , scale parameter  $\sigma$ , and threshold parameter  $\theta$ , is

$$y = f(x | k, \sigma, \theta) = \left( \frac{1}{\sigma} \right) \left( 1 + k \frac{(x - \theta)}{\sigma} \right)^{-1 - \frac{1}{k}}$$

for  $\theta < x$ , when  $k > 0$ , or for  $\theta < x < \theta - \sigma/k$  when  $k < 0$ .

For  $k = 0$ , the density is

$$y = f(x | 0, \sigma, \theta) = \left( \frac{1}{\sigma} \right) e^{-\frac{(x - \theta)}{\sigma}}$$

for  $\theta < x$ .

If  $k = 0$  and  $\theta = 0$ , the generalized Pareto distribution is equivalent to the exponential distribution. If  $k > 0$  and  $\theta = \sigma/k$ , the generalized Pareto distribution is equivalent to the Pareto distribution.

### Background

Like the exponential distribution, the generalized Pareto distribution is often used to model the tails of another distribution. For example, you might have washers from a manufacturing process. If random influences in the process lead to differences in the sizes of the washers, a standard probability distribution, such as the normal, could be used to model those sizes. However,

while the normal distribution might be a good model near its mode, it might not be a good fit to real data in the tails and a more complex model might be needed to describe the full range of the data. On the other hand, only recording the sizes of washers larger (or smaller) than a certain threshold means you can fit a separate model to those tail data, which are known as *exceedences*. You can use the generalized Pareto distribution in this way, to provide a good fit to extremes of complicated data.

The generalized Pareto distribution allows a continuous range of possible shapes that includes both the exponential and Pareto distributions as special cases. You can use either of those distributions to model a particular dataset of exceedences. The generalized Pareto distribution allows you to “let the data decide” which distribution is appropriate.

The generalized Pareto distribution has three basic forms, each corresponding to a limiting distribution of exceedence data from a different class of underlying distributions.

- Distributions whose tails decrease exponentially, such as the normal, lead to a generalized Pareto shape parameter of zero.
- Distributions whose tails decrease as a polynomial, such as Student’s  $t$ , lead to a positive shape parameter.
- Distributions whose tails are finite, such as the beta, lead to a negative shape parameter.

The generalized Pareto distribution is used in the tails of distribution fit objects of the `paretotails` class.

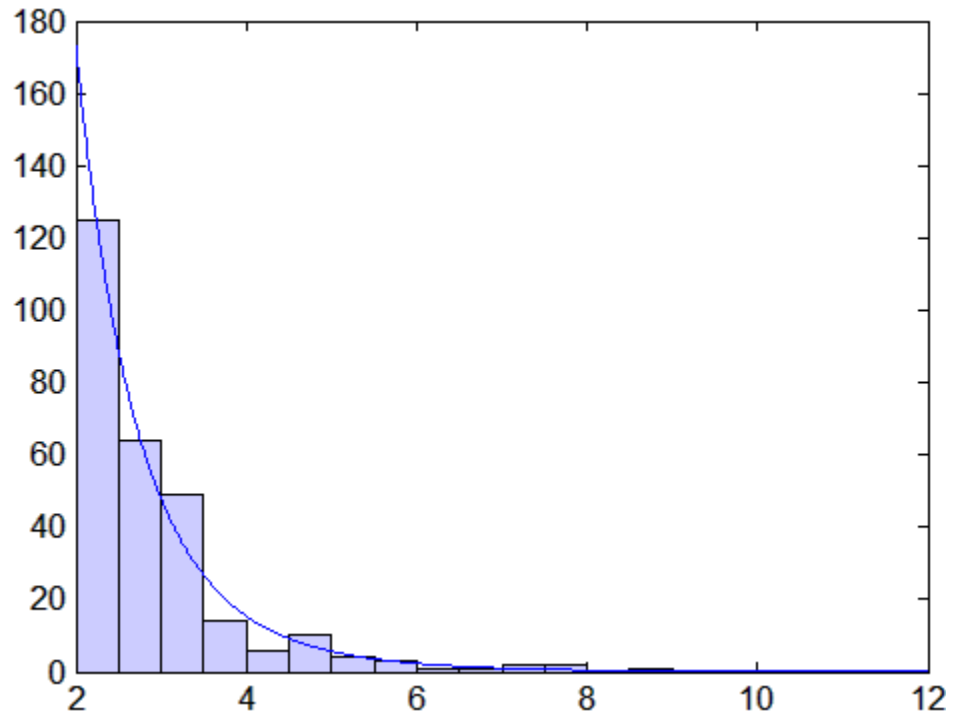
## Parameters

If you generate a large number of random values from a Student’s  $t$  distribution with 5 degrees of freedom, and then discard everything less than 2, you can fit a generalized Pareto distribution to those exceedences.

```
t = trnd(5,5000,1);
y = t(t > 2) - 2;
paramEsts = gpfity(y)
paramEsts =
    0.1267    0.8134
```

Notice that the shape parameter estimate (the first element) is positive, which is what you would expect based on exceedences from a Student's  $t$  distribution.

```
hist(y+2,2.25:.5:11.75);
set(get(gca,'child'),'FaceColor',[.8 .8 1])
xgrid = linspace(2,12,1000);
line(xgrid,.5*length(y)*...
      gppdf(xgrid,paramEsts(1),paramEsts(2),2));
```

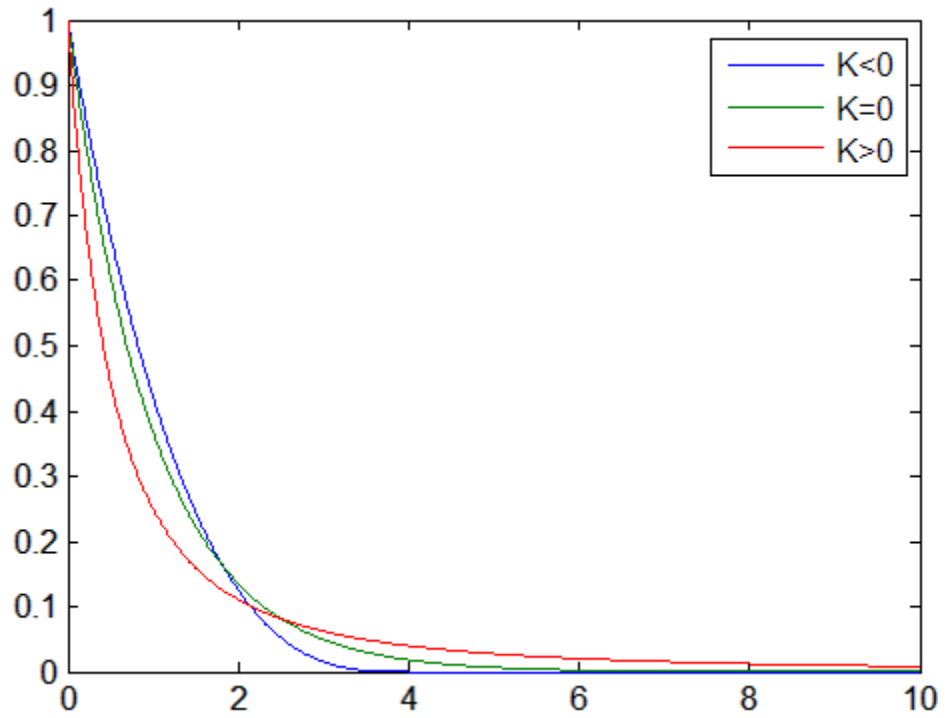


### Example

The following code generates examples of the probability density functions for the three basic forms of the generalized Pareto distribution.

```
x = linspace(0,10,1000);
y1 = gppdf(x,-.25,1,0);
y2 = gppdf(x,0,1,0);
```

```
y3 = gppdf(x,1,1,0)
plot(x,y1,'- ', x,y2,'- ', x,y3,'- ')
legend({'K<0' 'K=0' 'K>0'});
```



Notice that for  $k < 0$ , the distribution has zero probability density for  $x > -\frac{\sigma}{k}$ , while for  $k \geq 0$ , there is no upper bound.

## Geometric Distribution

### In this section...

“Overview” on page B-57

“Parameters” on page B-57

“Probability Mass Function” on page B-57

“Descriptive Statistics” on page B-58

“Relationship to Other Distributions” on page B-58

“Example” on page B-58

### Overview

The geometric distribution is discrete, existing only on the nonnegative integers. It models the number of successes before one failure in an independent succession of tests where each test results in success or failure.

### Parameters

The geometric distribution uses the following parameter.

Parameter	Description	Support
$p$	Probability of success	$0 \leq p \leq 1$

### Probability Mass Function

The probability mass function (pmf) is

$$f(x | p) = p(1 - p)^x \quad ; \quad x = 0, 1, 2, \dots, \infty.$$

## Descriptive Statistics

The mean is

$$\text{mean} = \frac{1 - p}{p}.$$

The variance is

$$\text{var} = \frac{1 - p}{p^2}.$$

## Relationship to Other Distributions

The geometric distribution is a special case of the negative binomial distribution, with  $r = 1$ .

## Example

Suppose the probability of a five-year-old battery failing in cold weather is 0.03. What is the probability of starting 25 consecutive days during a long cold snap?

```
1 - geocdf(25, 0.03)
```

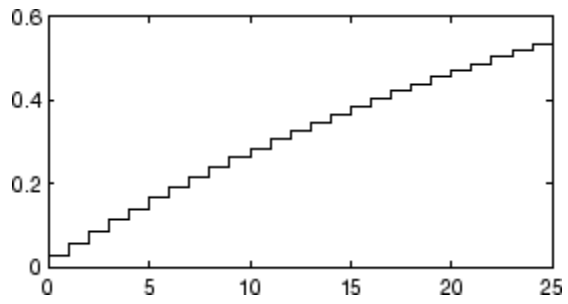
```
ans =
```

```
0.4530
```

The plot shows the cdf for this scenario.

```
x = 0:25;  
y = geocdf(x, 0.03);  
stairs(x, y)
```





## Hypergeometric Distribution

### In this section...

“Definition” on page B-60

“Background” on page B-60

“Examples” on page B-61

### Definition

The hypergeometric pdf is

$$y = f(x | M, K, n) = \frac{\binom{K}{x} \binom{M-K}{n-x}}{\binom{M}{n}}$$

### Background

The hypergeometric distribution models the total number of successes in a fixed-size sample drawn without replacement from a finite population.

The distribution is discrete, existing only for nonnegative integers less than the number of samples or the number of possible successes, whichever is greater. The hypergeometric distribution differs from the binomial only in that the population is finite and the sampling from the population is without replacement.

The hypergeometric distribution has three parameters that have direct physical interpretations.

- $M$  is the size of the population.
- $K$  is the number of items with the desired characteristic in the population.
- $n$  is the number of samples drawn.

Sampling “without replacement” means that once a particular sample is chosen, it is removed from the relevant population for all subsequent selections.

## Examples

### Compute and Plot Hypergeometric Distribution CDF

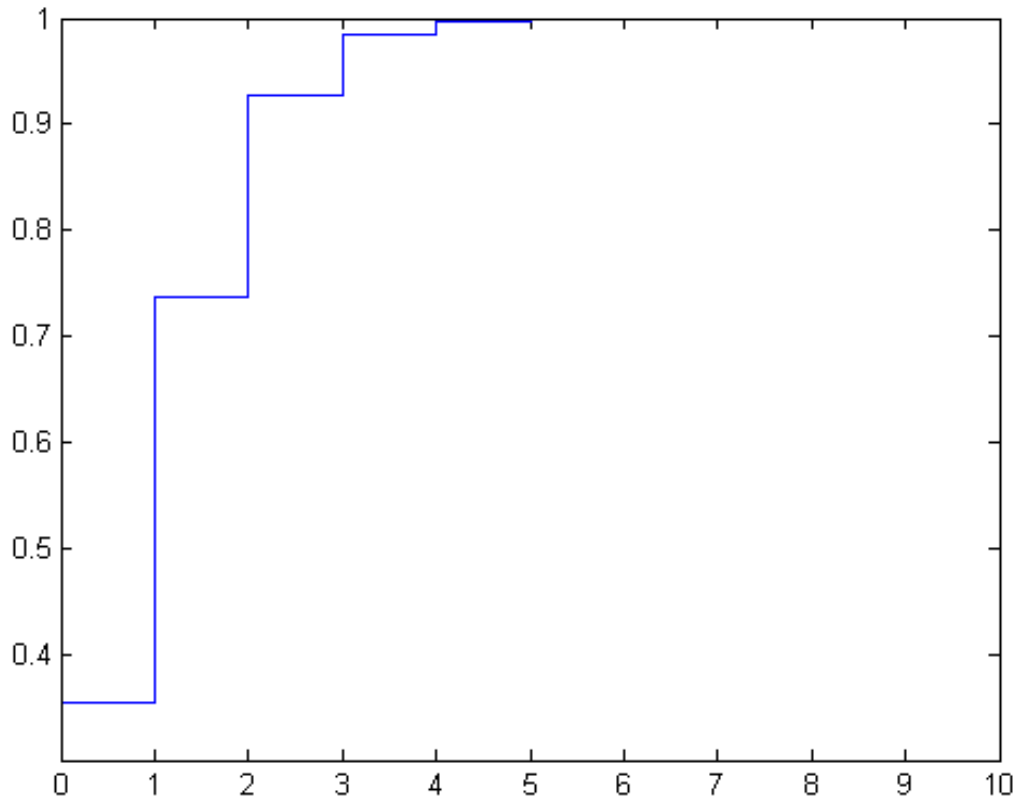
This example shows how to compute and plot the cdf of a hypergeometric distribution.

Compute the cdf of a hypergeometric distribution that draws 20 samples from a group of 1000 items, when the group contains 50 items of the desired type.

```
x = 0:10;  
y = hygecdf(x,1000,50,20);
```

Plot the cdf.

```
stairs(x,y)
```



The x-axis of the plot shows the number of items drawn that are of the desired type. The y-axis shows the corresponding cdf values.

# Inverse Gaussian Distribution

**In this section...**

“Definition” on page B-63

“Background” on page B-63

“Parameters” on page B-63

## Definition

The inverse Gaussian distribution has the density function

$$\sqrt{\frac{\lambda}{2\pi x^3}} \exp\left\{-\frac{\lambda}{2\mu^2 x}(x - \mu)^2\right\}$$

## Background

Also known as the Wald distribution, the inverse Gaussian is used to model nonnegative positively skewed data. The distribution originated in the theory of Brownian motion, but has been used to model diverse phenomena. Inverse Gaussian distributions have many similarities to standard Gaussian (normal) distributions, which lead to applications in inferential statistics.

## Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitting app.

## Inverse Wishart Distribution

### Definition

The probability density function of the  $d$ -dimensional Inverse Wishart distribution is given by

$$y = f(X, \Sigma, \nu) = \frac{|T|^{(\nu/2)} e^{\left(-\frac{1}{2}\text{trace}(TX^{-1})\right)}}{2^{(\nu d)/2} \pi^{(d(d-1))/4} |X|^{(\nu+d+1)/2} \Gamma(\nu/2) \dots \Gamma(\nu - (d-1)/2)},$$

where  $X$  and  $T$  are  $d$ -by- $d$  symmetric positive definite matrices, and  $\nu$  is a scalar greater than or equal to  $d$ . While it is possible to define the Inverse Wishart for singular  $T$ , the density cannot be written as above.

If a random matrix has a Wishart distribution with parameters  $T^{-1}$  and  $\nu$ , then the inverse of that random matrix has an inverse Wishart distribution with parameters  $T$  and  $\nu$ . The mean of the distribution is given by

$$\frac{1}{\nu - d - 1} T$$

where  $d$  is the number of rows and columns in  $T$ .

Only random matrix generation is supported for the inverse Wishart, including both singular and nonsingular  $T$ .

### Background

The inverse Wishart distribution is based on the Wishart distribution. In Bayesian statistics it is used as the conjugate prior for the covariance matrix of a multivariate normal distribution.

### Example

Notice that the sampling variability is quite large when the degrees of freedom is small.

```
Tau = [1 .5; .5 2];
df = 10; S1 = iwishrnd(Tau,df)*(df-2-1)
```

```
S1 =
    1.7959    0.64107
    0.64107    1.5496

df = 1000; S2 = iwishrnd(Tau,df)*(df-2-1)

S2 =
    0.9842    0.50158
    0.50158    2.1682
```

### **See Also**

“Wishart Distribution” on page B-144

## **Johnson System**

See “Pearson and Johnson Systems” on page 6-26.



# Kernel Distribution

**In this section...**

“Overview” on page B-67

“Kernel Density Estimator” on page B-67

“Kernel Smoothing Function” on page B-67

“Bandwidth” on page B-73

## Overview

A kernel distribution is a nonparametric representation of the probability density function (pdf) of a random variable. You can use a kernel distribution when a parametric distribution cannot properly describe the data, or when you want to avoid making assumptions about the distribution of the data. This distribution is defined by a smoothing function and a bandwidth value that controls the smoothness of the resulting density curve.

## Kernel Density Estimator

The kernel density estimator is the estimated probability density function (pdf) of the random variable. Its formula is given by the equation

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right) ; \quad -\infty < x < \infty,$$

where  $n$  is the sample size,  $K(\square)$  is the kernel smoothing function,  $h$  is the bandwidth.

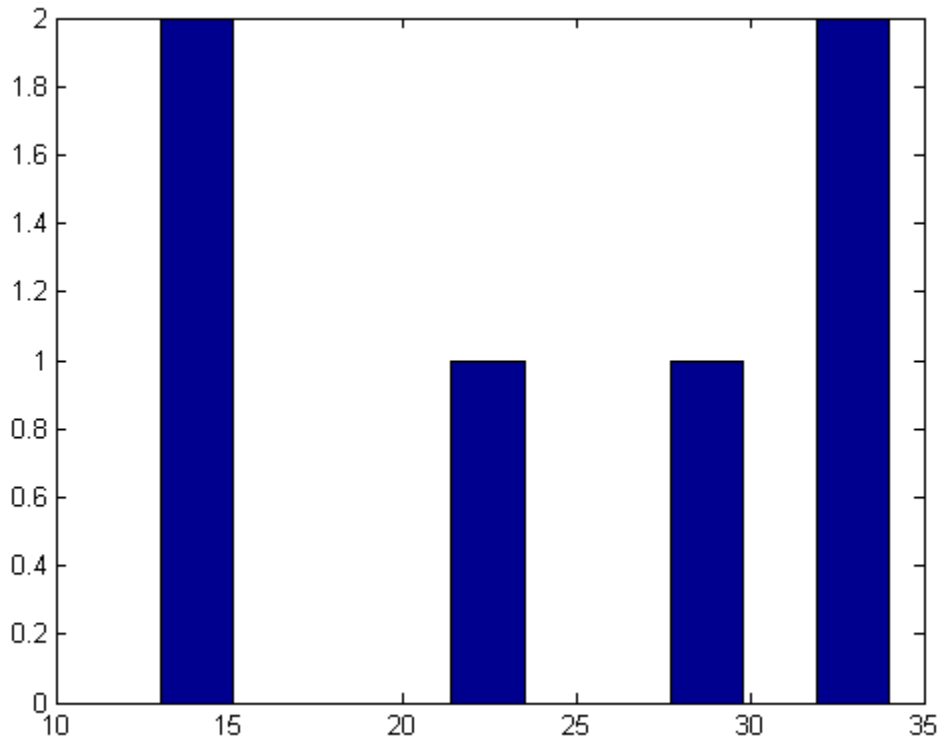
## Kernel Smoothing Function

The kernel smoothing function defines the shape of the curve used to generate the pdf. Similar to a histogram, the kernel distribution builds a function to represent the probability distribution using the sample data. But unlike a histogram, which places the values into discrete bins, a kernel distribution sums the component smoothing functions for each data value to produce a smooth, continuous probability curve. The following plots show a visual

comparison of a histogram and a kernel distribution generated from the same sample data.

A histogram represents the probability distribution by establishing bins and placing each data value in the appropriate bin.

```
SixMPG = [13;15;23;29;32;34];  
figure;  
hist(SixMPG)
```

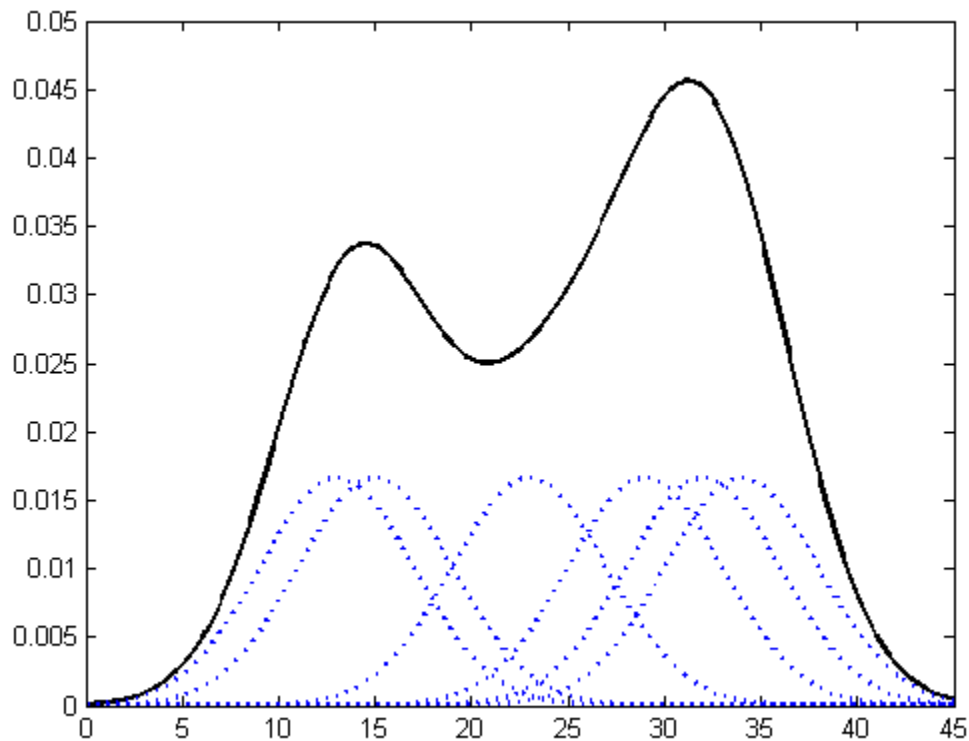


Because of this bin count approach, the histogram produces a discrete probability density function. This might be unsuitable for certain applications, such as generating random numbers from a fitted distribution.

Alternatively, the kernel distribution builds the pdf by creating an individual probability density curve for each data value, then summing the smooth curves. This approach creates one smooth, continuous probability density function for the data set.

```
figure;
pdSix = fitdist(SixMPG,'Kernel','BandWidth',4);
x = 0:.1:45;
ySix = pdf(pdSix,x);
plot(x,ySix,'k-','LineWidth',2);

% Plot each individual pdf and scale its appearance on the plot
hold on;
for i=1:6
    pd = makedist('Normal','mu',SixMPG(i),'sigma',4);
    y = pdf(pd,x);
    y = y/6;
    plot(x,y,'b:');
end
hold off;
```

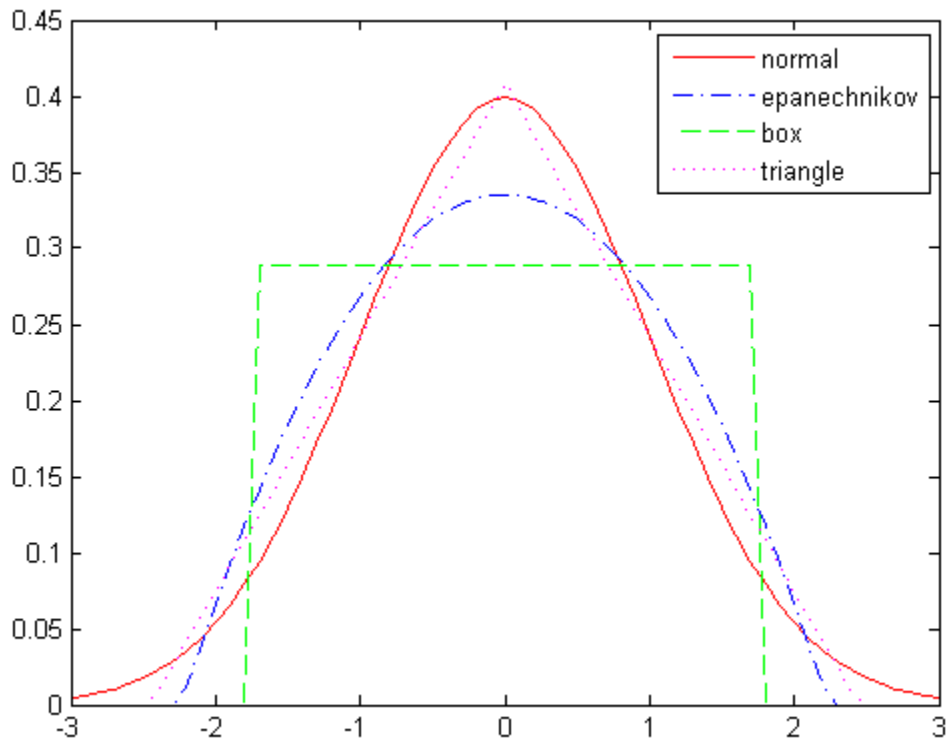


The smaller dashed curves are the probability distributions for each value in the sample data, scaled to fit the plot. The larger solid curve is the overall kernel distribution of the SixMPG data. The kernel smoothing function refers to the shape of those smaller component curves, which have a normal distribution in this example.

You can choose one of several options for the kernel smoothing function. This plot shows the shapes of the available smoothing functions.

```
% Set plot specifications
hname = {'normal' 'epanechnikov' 'box' 'triangle'};
colors = {'r' 'b' 'g' 'm'};
lines = {'-', '-.', '--', ':'};
```

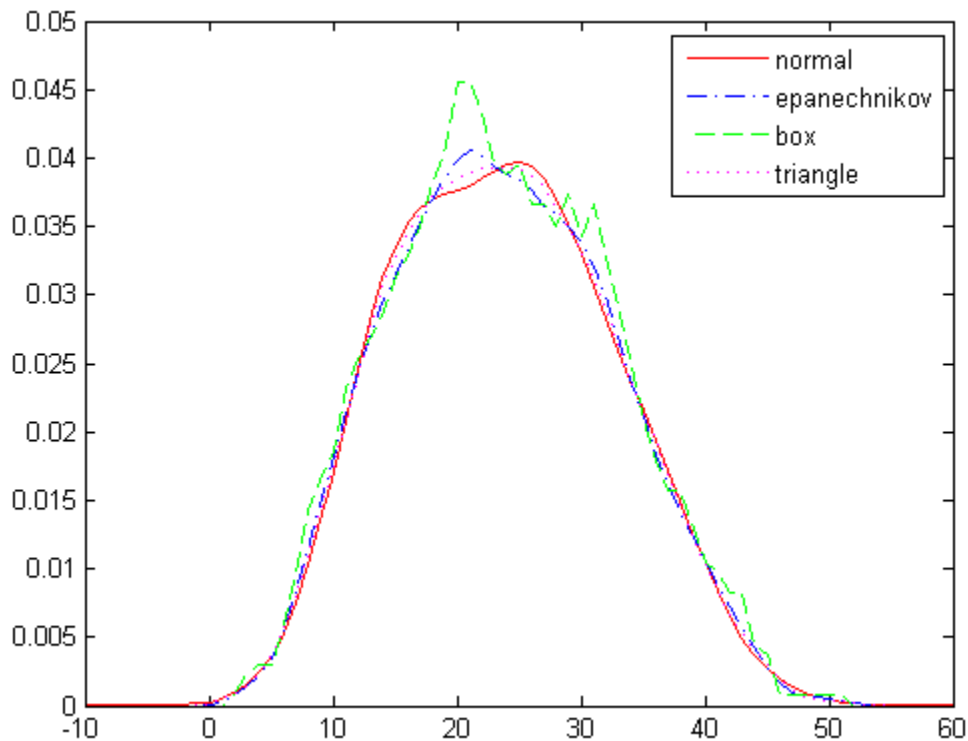
```
% Generate a sample of each kernel smoothing function and plot
data = [0];
figure;
for j=1:4
    pd = fitdist(data,'kernel','Kernel',hname{j});
    x = -3:.1:3;
    y = pdf(pd,x);
    plot(x,y,'Color',colors{j},'LineStyle',lines{j});
    hold on;
end
legend(hname{:});
hold off;
```



To understand the effect of different kernel smoothing functions on the shape of the resulting pdf estimate, compare plots of the mileage data (MPG) from `carbig.mat` using each available kernel function.

```
% Set plot specifications
hname = {'normal' 'epanechnikov' 'box' 'triangle'};
colors = {'r' 'b' 'g' 'm'};
lines = {'-', '-.', '--', ':'};

% Generate kernel distribution objects and plot
figure;
for j=1:4
    pd = fitdist(MPG, 'kernel', 'Kernel', hname{j});
    x = -10:1:60;
    y = pdf(pd, x);
    plot(x, y, 'Color', colors{j}, 'LineStyle', lines{j});
    hold on;
end
legend(hname{:});
hold off;
```



Each density curve uses the same input data, but applies a different kernel smoothing function to generate the pdf. The density estimates are roughly comparable, but the shape of each curve varies slightly. For example, the box kernel produces a density curve that is less smooth than the others.

## Bandwidth

The choice of bandwidth value controls the smoothness of the resulting probability density curve. This plot shows the density estimate for the same MPG data, using a normal kernel smoothing function with three different bandwidths.

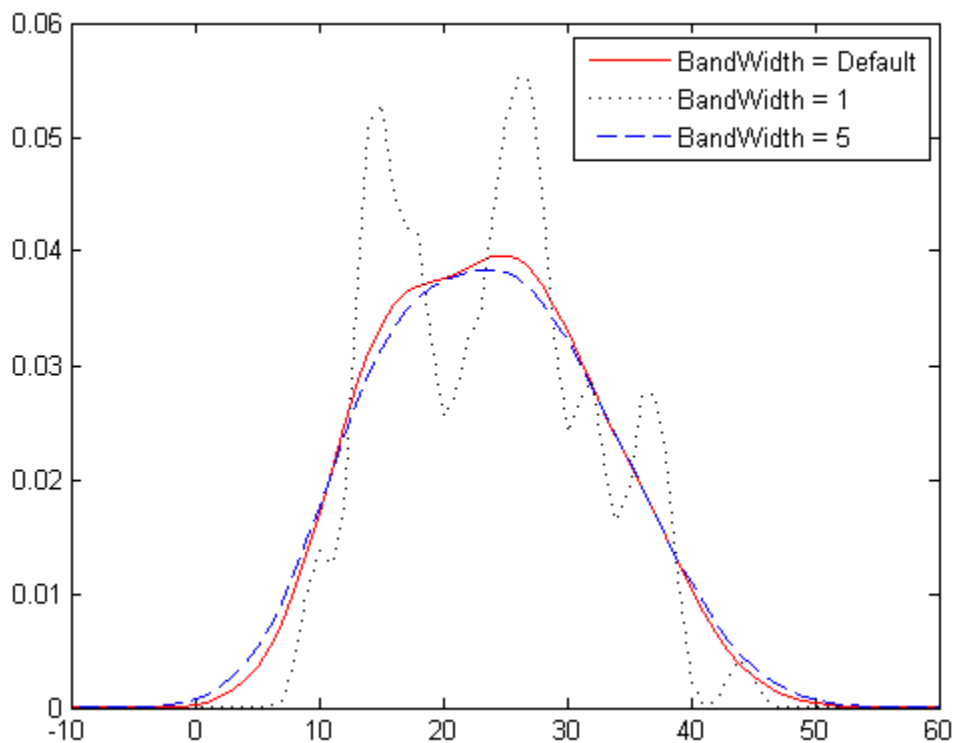
```
% Create kernel distribution objects
```

```
pd1 = fitdist(MPG,'kernel');
pd2 = fitdist(MPG,'kernel','BandWidth',1);
pd3 = fitdist(MPG,'kernel','BandWidth',5);

% Compute each pdf
x = -10:1:60;
y1 = pdf(pd1,x);
y2 = pdf(pd2,x);
y3 = pdf(pd3,x);

% Plot each pdf
plot(x,y1,'Color','r','LineStyle','-');
hold on;
plot(x,y2,'Color','k','LineStyle',':');
plot(x,y3,'Color','b','LineStyle','-');
legend({'BandWidth = Default','BandWidth = 1','BandWidth = 5'});
hold off;
```





The default bandwidth, which is theoretically optimal for estimating densities for the normal distribution, produces a reasonably smooth curve. Specifying a smaller bandwidth produces a very rough curve, but reveals that there might be two major peaks in the data. Specifying a larger bandwidth produces a curve nearly identical to the kernel function, and is so smooth that it obscures potentially important features of the data.

**See Also** `ksdensity`

### **Related Examples**

- “Fit Kernel Distribution Object to Data” on page 5-16
- “Fit Kernel Distribution Using `ksdensity`” on page 5-21
- “Fit Kernel Distributions to Grouped Data Using `ksdensity`” on page 5-23

## Logistic Distribution

In this section...
“Overview” on page B-76
“Parameters” on page B-76
“Probability Density Function” on page B-76
“Relationship to Other Distributions” on page B-77

### Overview

The logistic distribution is used for growth models and in logistic regression. It has longer tails and a higher kurtosis than the normal distribution.

### Parameters

The logistic distribution uses the following parameters.

Parameter	Description	Support
mu	Mean	$-\infty < \mu < \infty$
sigma	Scale parameter	$\sigma \geq 0$

### Probability Density Function

The probability density function (pdf) is

$$f(x | \mu, \sigma) = \frac{\exp\left\{\frac{x - \mu}{\sigma}\right\}}{\sigma \left(1 + \exp\left\{\frac{x - \mu}{\sigma}\right\}\right)^2} ; \quad -\infty < x < \infty.$$

## **Relationship to Other Distributions**

The loglogistic distribution is closely related to the logistic distribution. If  $x$  is distributed loglogistically with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed logistically with mean and standard deviation.

## Loglogistic Distribution

In this section...
“Overview” on page B-78
“Parameters” on page B-78
“Probability Density Function” on page B-78
“Relationship to Other Distributions” on page B-79

### Overview

The loglogistic distribution is a probability distribution whose logarithm has a logistic distribution. This distribution is often used in survival analysis to model events that experience an initial rate increase, followed by a rate decrease. It is also known as the Fisk distribution in economics applications.

### Parameters

The loglogistic distribution uses the following parameters.

Parameter	Description	Support
mu	Log mean	$\mu > 0$
sigma	Log scale parameter	$\sigma > 0$

### Probability Density Function

The probability density function (pdf) is

$$f(x | \mu, \sigma) = \frac{1}{\sigma} \frac{1}{x} \frac{e^z}{(1 + e^z)^2} ; \quad x \geq 0,$$

where  $z = \frac{\log(x) - \mu}{\sigma}$ .

## **Relationship to Other Distributions**

The loglogistic distribution is closely related to the logistic distribution. If  $x$  is distributed loglogistically with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed logistically with mean and standard deviation. The relationship is similar to that between the lognormal and normal distribution.

## Lognormal Distribution

In this section...
“Overview” on page B-80
“Parameters” on page B-80
“Probability Density Function” on page B-80
“Descriptive Statistics” on page B-81
“Relationship to Other Distributions” on page B-81
“Example” on page B-81

### Overview

The lognormal distribution is a probability distribution whose logarithm has a normal distribution. It is sometimes called the Galton distribution. The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(x)$  exists only when  $x$  is positive.

### Parameters

The lognormal distribution uses the following parameters.

Parameter	Description	Support
$\mu$	Log mean	$-\infty < \mu < \infty$
$\sigma$	Log standard deviation	$\sigma \geq 0$

### Probability Density Function

The probability density function (pdf) of the lognormal distribution is

$$f(x | \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left\{-\frac{(\ln x - \mu)^2}{2\sigma^2}\right\} ; \quad x > 0.$$

## Descriptive Statistics

The mean is

$$\text{mean} = \exp\left(\frac{\mu + \sigma^2}{2}\right).$$

The variance is

$$\text{var} = \exp(2\mu + \sigma^2)(\exp(\sigma^2) - 1).$$

You can compute these descriptive statistics using the `lognstat` function.

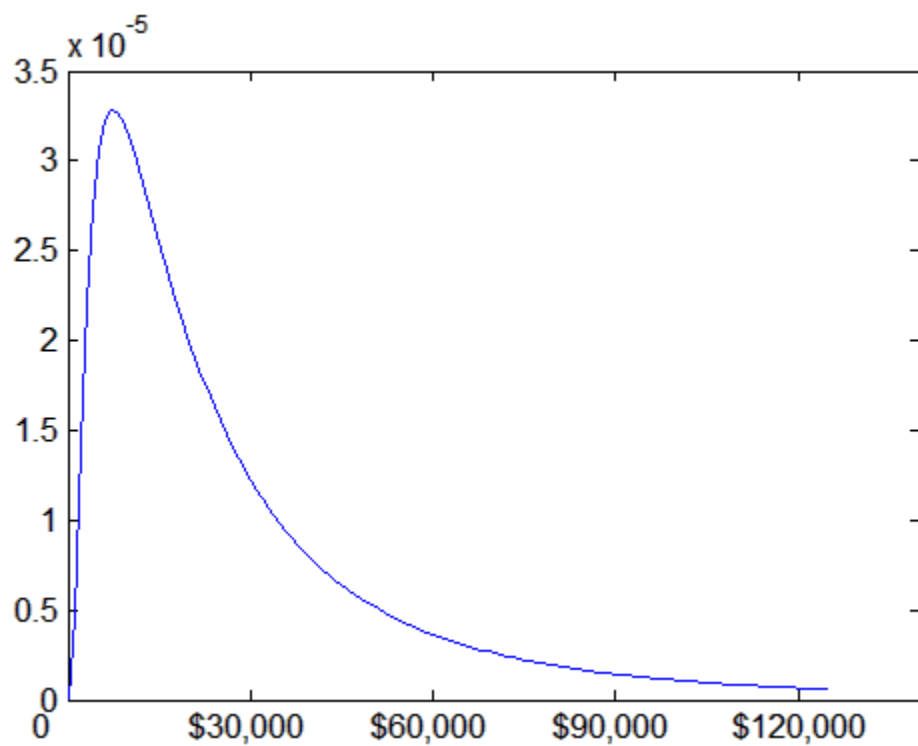
## Relationship to Other Distributions

The lognormal distribution is closely related to the normal distribution. If  $x$  is distributed lognormally with parameters  $\mu$  and  $\sigma$ , then  $\log(x)$  is distributed normally with mean  $\mu$  and standard deviation  $\sigma$ . The lognormal distribution is applicable when the quantity of interest must be positive, since  $\log(x)$  exists only when  $x$  is positive.

## Example

Suppose the income of a family of four in the United States follows a lognormal distribution with  $\mu = \log(20,000)$  and  $\sigma^2 = 1.0$ . Plot the income density.

```
x = (10:1000:125010)';
y = lognpdf(x, log(20000), 1.0);
plot(x,y)
set(gca,'xtick',[0 30000 60000 90000 120000])
set(gca,'xticklabel',{'0', '$30,000', '$60,000', ...
                    '$90,000', '$120,000'})
```





# Multinomial Distribution

## In this section...

“Overview” on page B-83

“Parameter” on page B-83

“Probability Density Function” on page B-83

“Descriptive Statistics” on page B-84

“Relationship to Other Distributions” on page B-84

## Overview

Multinomial distribution models the probability of each combination of successes in a series of independent trials. Use this distribution when there are more than two possible mutually exclusive outcomes for each trial, and each outcome has a fixed probability of success.

## Parameter

Multinomial distribution uses the following parameter.

Parameter	Description	Constraints
probabilities	Outcome probabilities	$0 \leq \text{probabilities}(i) \leq 1; \sum_{\text{all}(i)} \text{probabilities}(i) = 1$

## Probability Density Function

The multinomial pdf is

$$f(x | n, p) = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k},$$

where  $k$  is the number of possible mutually exclusive outcomes for each trial, and  $n$  is the total number of trials. The vector  $x = (x_1 \dots x_k)$  is the number of observations of each  $k$  outcome, and contains nonnegative integer components

that sum to  $n$ . The vector  $p = (p_1 \dots p_k)$  is the fixed probability of each  $k$  outcome, and contains nonnegative scalar components that sum to 1.

## **Descriptive Statistics**

The expected number of observations of outcome  $i$  in  $n$  trials is

$$E\{x_i\} = np_i,$$

where  $p_i$  is the fixed probability of outcome  $i$ .

The variance of outcome  $i$  is

$$\text{var}(x_i) = np_i(1 - p_i).$$

The covariance of outcomes  $i$  and  $j$  is

$$\text{cov}(x_i, x_j) = -np_i p_j, i \neq j.$$

## **Relationship to Other Distributions**

The multinomial distribution is a generalization of the binomial distribution. While the binomial distribution gives the probability of the number of “successes” in  $n$  independent trials of a two-outcome process, the multinomial distribution gives the probability of each combination of outcomes in  $n$  independent trials of a  $k$ -outcome process. The probability of each outcome in any one trial is given by the fixed probabilities  $p_1, \dots, p_k$ .

## **Multivariate Gaussian Distribution**

See “Multivariate Normal Distribution” on page B-86.

## Multivariate Normal Distribution

### In this section...

“Definition” on page B-86

“Background” on page B-86

“Example” on page B-87

### Definition

The probability density function of the  $d$ -dimensional multivariate normal distribution is given by

$$y = f(x, \mu, \Sigma) = \frac{1}{\sqrt{|\Sigma|} (2\pi)^d} e^{-\frac{1}{2}(x-\mu) \Sigma^{-1}(x-\mu)'}$$

where  $x$  and  $\mu$  are 1-by- $d$  vectors and  $\Sigma$  is a  $d$ -by- $d$  symmetric positive definite matrix. While it is possible to define the multivariate normal for singular  $\Sigma$ , the density cannot be written as above. Only random vector generation is supported for the singular case. Note that while most textbooks define the multivariate normal with  $x$  and  $\mu$  oriented as column vectors, for the purposes of data analysis software, it is more convenient to orient them as row vectors, and Statistics Toolbox software uses that orientation.

### Background

The multivariate normal distribution is a generalization of the univariate normal to two or more variables. It is a distribution for random vectors of correlated variables, each element of which has a univariate normal distribution. In the simplest case, there is no correlation among variables, and elements of the vectors are independent univariate normal random variables.

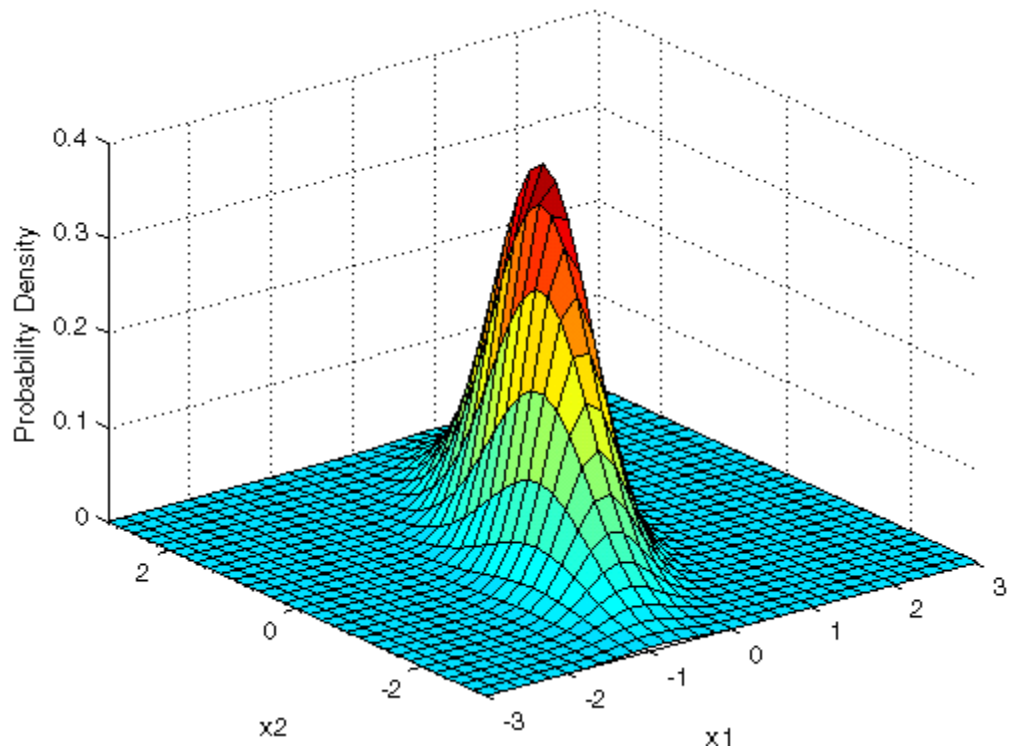
The multivariate normal distribution is parameterized with a mean vector,  $\mu$ , and a covariance matrix,  $\Sigma$ . These are analogous to the mean  $\mu$  and variance  $\sigma^2$  parameters of a univariate normal distribution. The diagonal elements of  $\Sigma$  contain the variances for each variable, while the off-diagonal elements of  $\Sigma$  contain the covariances between variables.

The multivariate normal distribution is often used as a model for multivariate data, primarily because it is one of the few multivariate distributions that is tractable to work with.

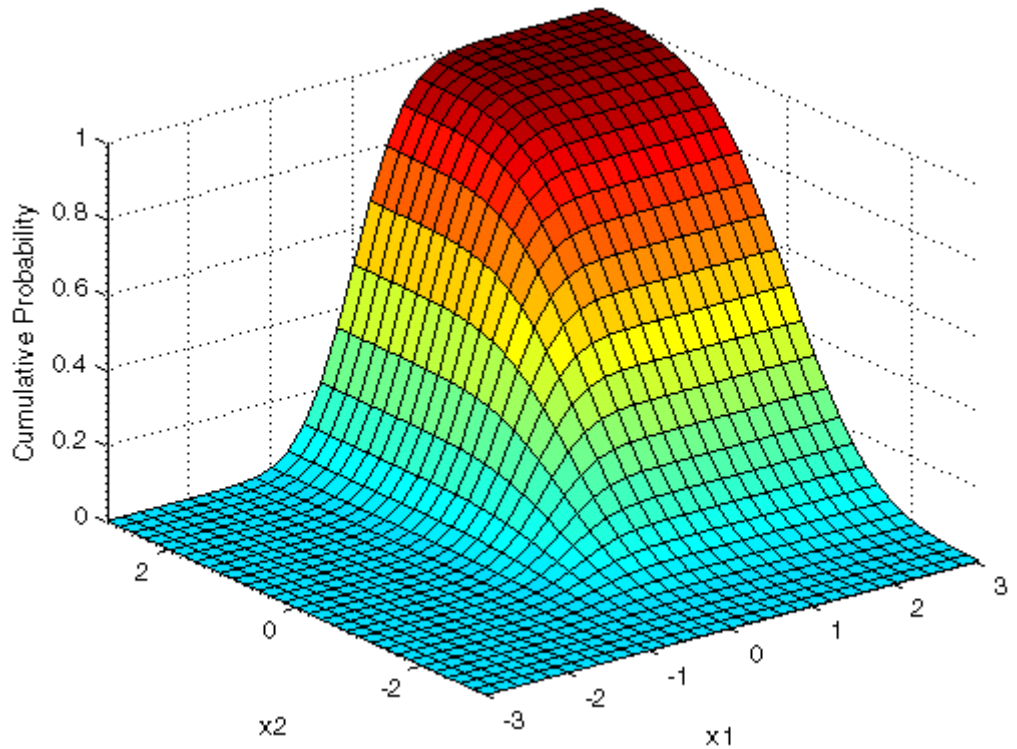
## Example

This example shows the probability density function (pdf) and cumulative distribution function (cdf) for a bivariate normal distribution with unequal standard deviations. You can use the multivariate normal distribution in a higher number of dimensions as well, although visualization is not easy.

```
mu = [0 0];
Sigma = [.25 .3; .3 1];
x1 = -3:.2:3; x2 = -3:.2:3;
[X1,X2] = meshgrid(x1,x2);
F = mvnpdf([X1(:) X2(:)],mu,Sigma);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 .4])
xlabel('x1'); ylabel('x2'); zlabel('Probability Density');
```

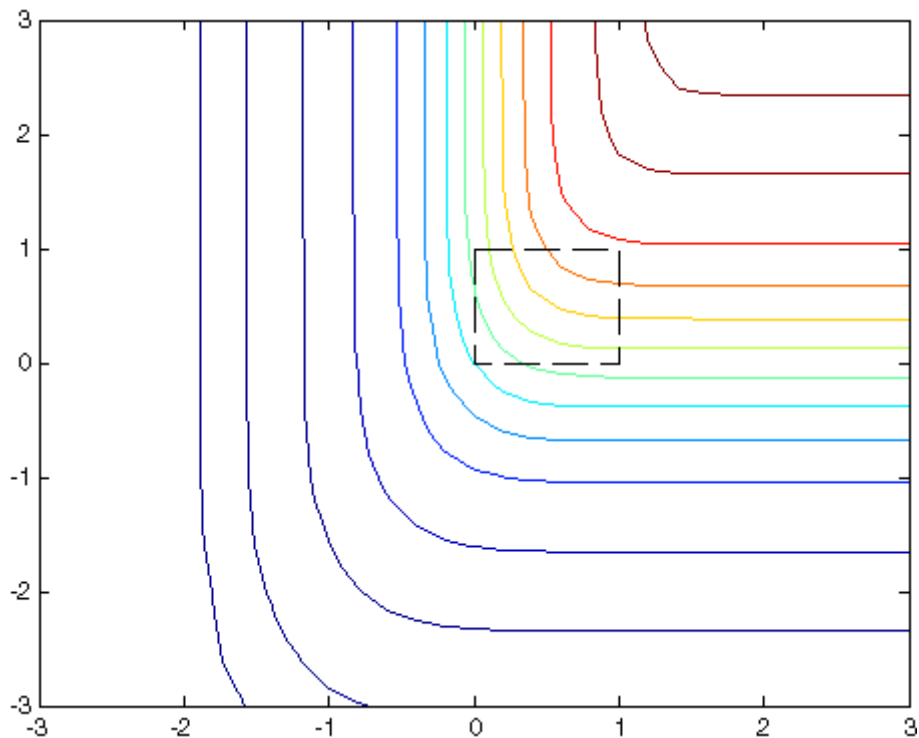


```
F = mvncdf([X1(:) X2(:)],mu,Sigma);  
F = reshape(F,length(x2),length(x1));  
surf(x1,x2,F);  
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);  
axis([-3 3 -3 3 0 1])  
xlabel('x1'); ylabel('x2'); zlabel('Cumulative Probability');
```



Since the bivariate normal distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions. For example, this contour plot illustrates the computation that follows, of the probability contained within the unit square.

```
contour(x1,x2,F,[.0001 .001 .01 .05:.1:.95 .99 .999 .9999]);
xlabel('x'); ylabel('y');
line([0 0 1 1 0],[1 0 0 1 1],'linestyle','--','color','k');
```



```
mvncdf([0 0],[1 1],mu,Sigma)
ans =
    0.20974
```

Computing a multivariate cumulative probability requires significantly more work than computing a univariate probability. By default, the `mvncdf` function computes values to less than full machine precision, and returns an estimate of the error as an optional second output:

```
[F,err] = mvncdf([0 0],[1 1],mu,Sigma)
F =
    0.20974
err =
    1e-008
```



## Multivariate $t$ Distribution

### In this section...

“Definition” on page B-91

“Background” on page B-91

“Example” on page B-92

### Definition

The probability density function of the  $d$ -dimensional multivariate Student’s  $t$  distribution is given by

$$f(x, \Sigma, \nu) = \frac{1}{|\Sigma|^{1/2}} \frac{1}{\sqrt{(\nu\pi)^d}} \frac{\Gamma((\nu + d) / 2)}{\Gamma(\nu / 2)} \left( 1 + \frac{x' \Sigma^{-1} x}{\nu} \right)^{-(\nu+d)/2}.$$

where  $x$  is a 1-by- $d$  vector,  $\Sigma$  is a  $d$ -by- $d$  symmetric, positive definite matrix, and  $\nu$  is a positive scalar. While it is possible to define the multivariate Student’s  $t$  for singular  $\Sigma$ , the density cannot be written as above. For the singular case, only random number generation is supported. Note that while most textbooks define the multivariate Student’s  $t$  with  $x$  oriented as a column vector, for the purposes of data analysis software, it is more convenient to orient  $x$  as a row vector, and Statistics Toolbox software uses that orientation.

### Background

The multivariate Student’s  $t$  distribution is a generalization of the univariate Student’s  $t$  to two or more variables. It is a distribution for random vectors of correlated variables, each element of which has a univariate Student’s  $t$  distribution. In the same way as the univariate Student’s  $t$  distribution can be constructed by dividing a standard univariate normal random variable by the square root of a univariate chi-square random variable, the multivariate Student’s  $t$  distribution can be constructed by dividing a multivariate normal random vector having zero mean and unit variances by a univariate chi-square random variable.

The multivariate Student’s  $t$  distribution is parameterized with a correlation matrix,  $\Sigma$ , and a positive scalar degrees of freedom parameter,  $\nu$ .  $\nu$  is

analogous to the degrees of freedom parameter of a univariate Student's  $t$  distribution. The off-diagonal elements of  $\Sigma$  contain the correlations between variables. Note that when  $\Sigma$  is the identity matrix, variables are uncorrelated; however, they are not independent.

The multivariate Student's  $t$  distribution is often used as a substitute for the multivariate normal distribution in situations where it is known that the marginal distributions of the individual variables have fatter tails than the normal.

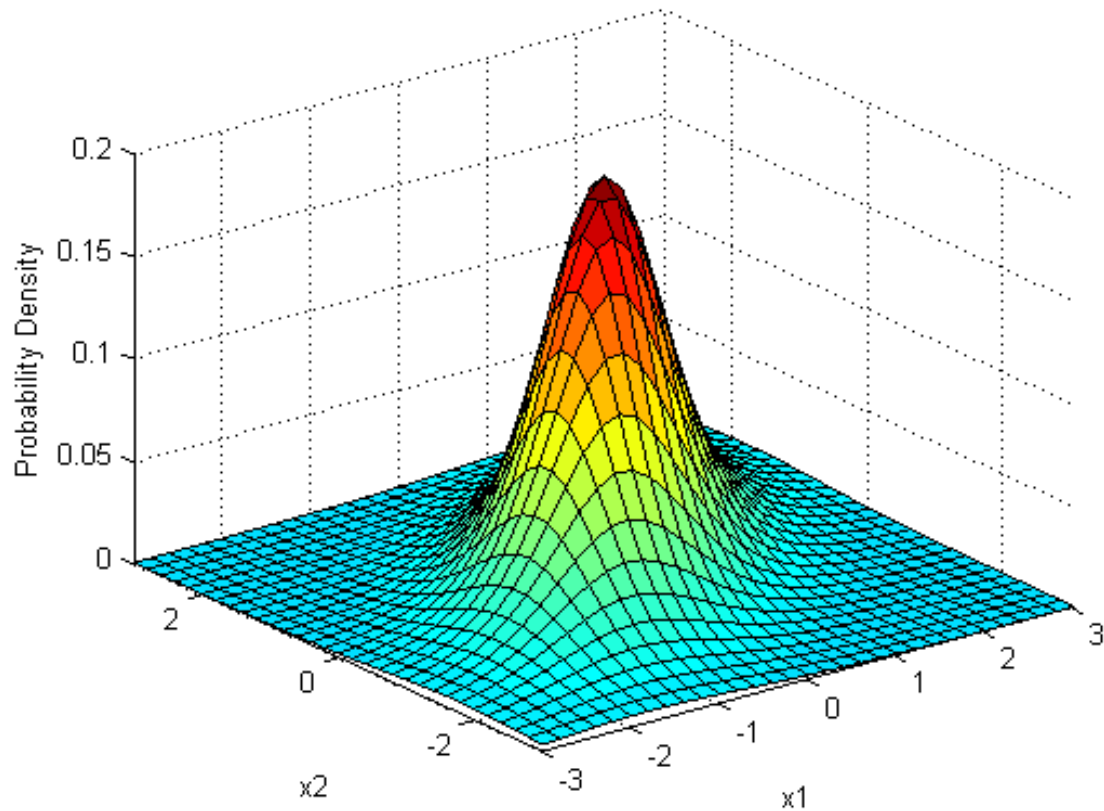
## Example

### Plot PDF and CDF of Multivariate $t$ -Distribution

This example shows the probability density function (pdf) and cumulative distribution function (cdf) for a multivariate  $t$ -distribution.

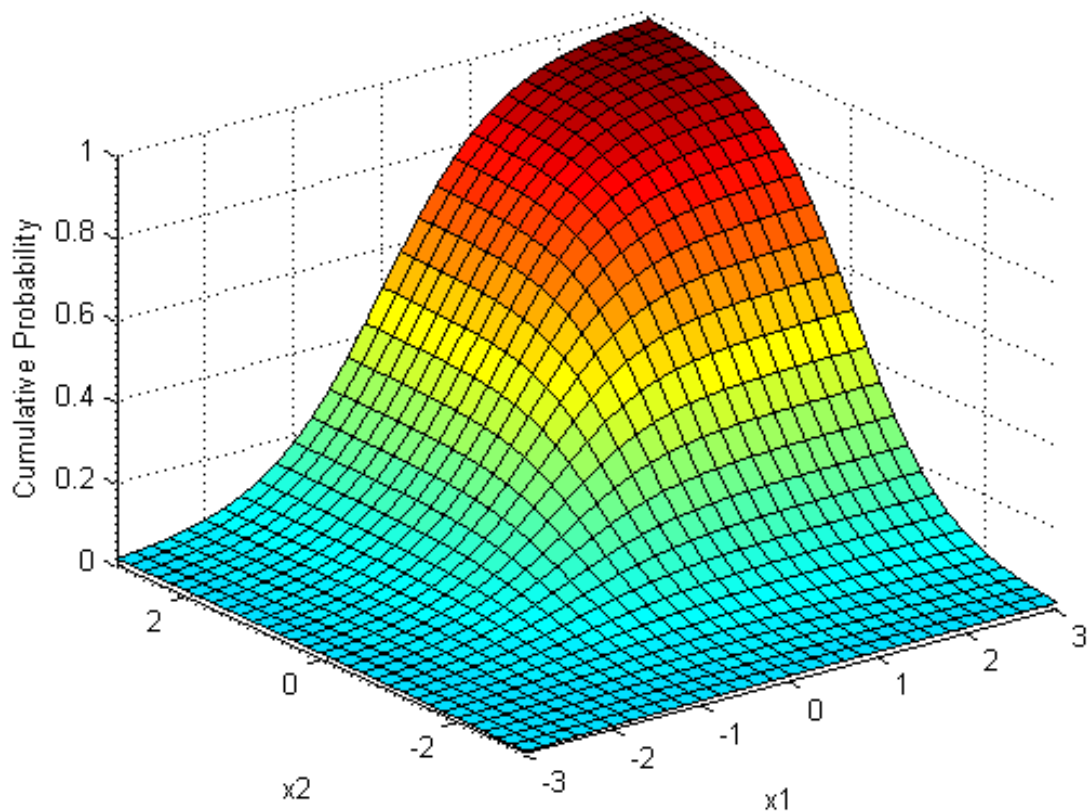
Plot the pdf of a bivariate Student's  $t$  distribution. You can use this distribution for a higher number of dimensions as well, although visualization is not easy.

```
Rho = [1 .6; .6 1];
nu = 5;
x1 = -3:.2:3; x2 = -3:.2:3;
[X1,X2] = meshgrid(x1,x2);
F = mvtpdf([X1(:) X2(:)],Rho,nu);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 .2])
xlabel('x1'); ylabel('x2'); zlabel('Probability Density');
```



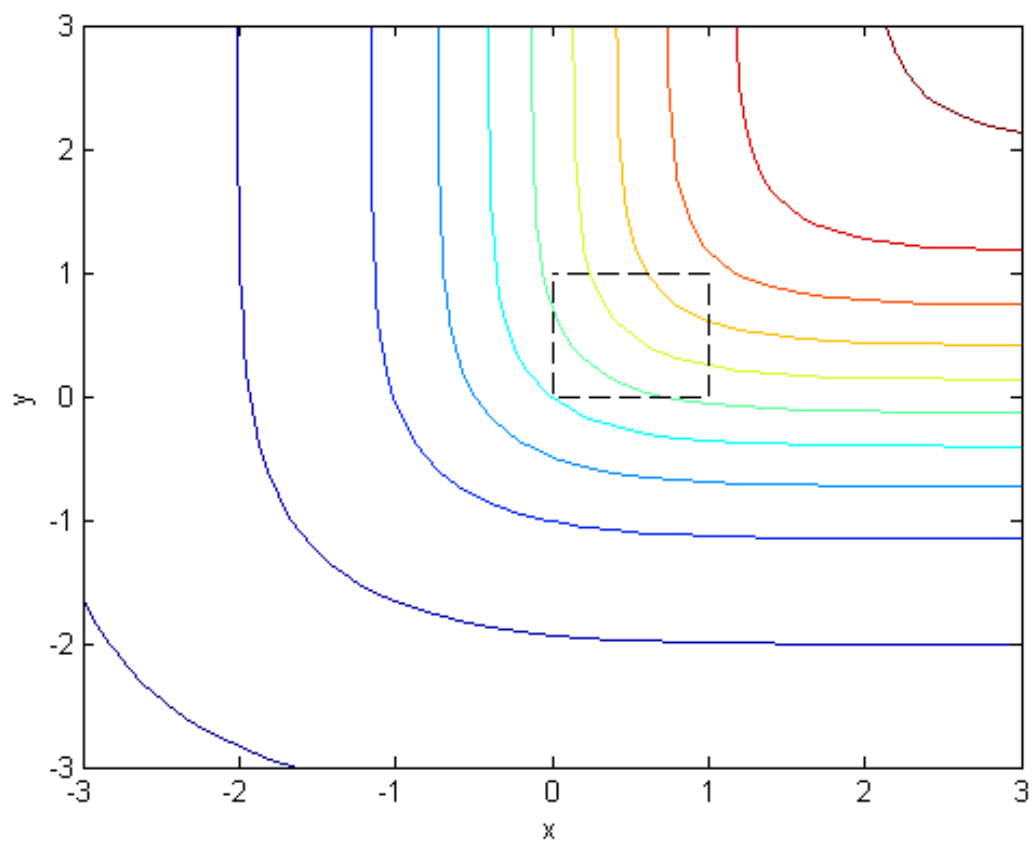
Plot the cdf of a bivariate Student's  $t$  distribution.

```
F = mvtcdf([X1(:) X2(:)],Rho,nu);
F = reshape(F,length(x2),length(x1));
surf(x1,x2,F);
caxis([min(F(:))-.5*range(F(:)),max(F(:))]);
axis([-3 3 -3 3 0 1])
xlabel('x1'); ylabel('x2'); zlabel('Cumulative Probability');
```



Since the bivariate Student's  $t$  distribution is defined on the plane, you can also compute cumulative probabilities over rectangular regions. For example, this contour plot illustrates the computation that follows, of the probability contained within the unit square shown in the figure.

```
contour(x1,x2,F,[.0001 .001 .01 .05:.1:.95 .99 .999 .9999]);
xlabel('x'); ylabel('y');
line([0 0 1 1 0],[1 0 0 1 1],'linestyle','--','color','k');
```



Compute the value of the probability contained within the unit square.

```
F = mvtcdf([0 0],[1 1],Rho,nu)
```

F =

0.1401

Computing a multivariate cumulative probability requires significantly more work than computing a univariate probability. By default, the `mvtcdf` function computes values to less than full machine precision and returns an estimate of the error, as an optional second output.

```
[F,err] = mvtcdf([0 0],[1 1],Rho,nu)
```

```
F =
```

```
0.1401
```

```
err =
```

```
1.0000e-08
```

# Nakagami Distribution

## In this section...

“Definition” on page B-97

“Background” on page B-97

“Parameters” on page B-97

## Definition

The Nakagami distribution has the density function

$$2 \left( \frac{\mu}{\omega} \right)^{\mu} \frac{1}{\Gamma(\mu)} x^{(2\mu-1)} e^{-\frac{\mu}{\omega} x^2}$$

with shape parameter  $\mu$  and scale parameter  $\omega > 0$ , for  $x > 0$ . If  $x$  has a Nakagami distribution with parameters  $\mu$  and  $\omega$ , then  $x^2$  has a gamma distribution with shape parameter  $\mu$  and scale parameter  $\omega/\mu$ .

## Background

In communications theory, Nakagami distributions, Rician distributions, and Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

## Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitting app.

## Negative Binomial Distribution

### In this section...

“Definition” on page B-98

“Background” on page B-98

“Parameters” on page B-99

“Example” on page B-101

### Definition

When the  $r$  parameter is an integer, the negative binomial pdf is

$$y = f(x | r, p) = \binom{r+x-1}{x} p^r q^x I_{(0,1,\dots)}(x)$$

where  $q = 1 - p$ . When  $r$  is not an integer, the binomial coefficient in the definition of the pdf is replaced by the equivalent expression

$$\frac{\Gamma(r+x)}{\Gamma(r)\Gamma(x+1)}$$

### Background

In its simplest form (when  $r$  is an integer), the negative binomial distribution models the number of failures  $x$  before a specified number of successes is reached in a series of independent, identical trials. Its parameters are the probability of success in a single trial,  $p$ , and the number of successes,  $r$ . A special case of the negative binomial distribution, when  $r = 1$ , is the geometric distribution, which models the number of failures before the first success.

More generally,  $r$  can take on non-integer values. This form of the negative binomial distribution has no interpretation in terms of repeated trials, but, like the Poisson distribution, it is useful in modeling count data. The negative binomial distribution is more general than the Poisson distribution because it has a variance that is greater than its mean, making it suitable for count data that do not meet the assumptions of the Poisson distribution. In the limit,



as  $r$  increases to infinity, the negative binomial distribution approaches the Poisson distribution.

## Parameters

Suppose you are collecting data on the number of auto accidents on a busy highway, and would like to be able to model the number of accidents per day. Because these are count data, and because there are a very large number of cars and a small probability of an accident for any specific car, you might think to use the Poisson distribution. However, the probability of having an accident is likely to vary from day to day as the weather and amount of traffic change, and so the assumptions needed for the Poisson distribution are not met. In particular, the variance of this type of count data sometimes exceeds the mean by a large amount. The data below exhibit this effect: most days have few or no accidents, and a few days have a large number.

```
accident = [2 3 4 2 3 1 12 8 14 31 23 1 10 7 0];  
m = mean(accident)  
v = var(accident)
```

```
m =
```

```
8.0667
```

```
v =
```

```
79.3524
```

The negative binomial distribution is more general than the Poisson, and is often suitable for count data when the Poisson is not. The function `nbinfit` returns the maximum likelihood estimates (MLEs) and confidence intervals for the parameters of the negative binomial distribution. Here are the results from fitting the accident data:

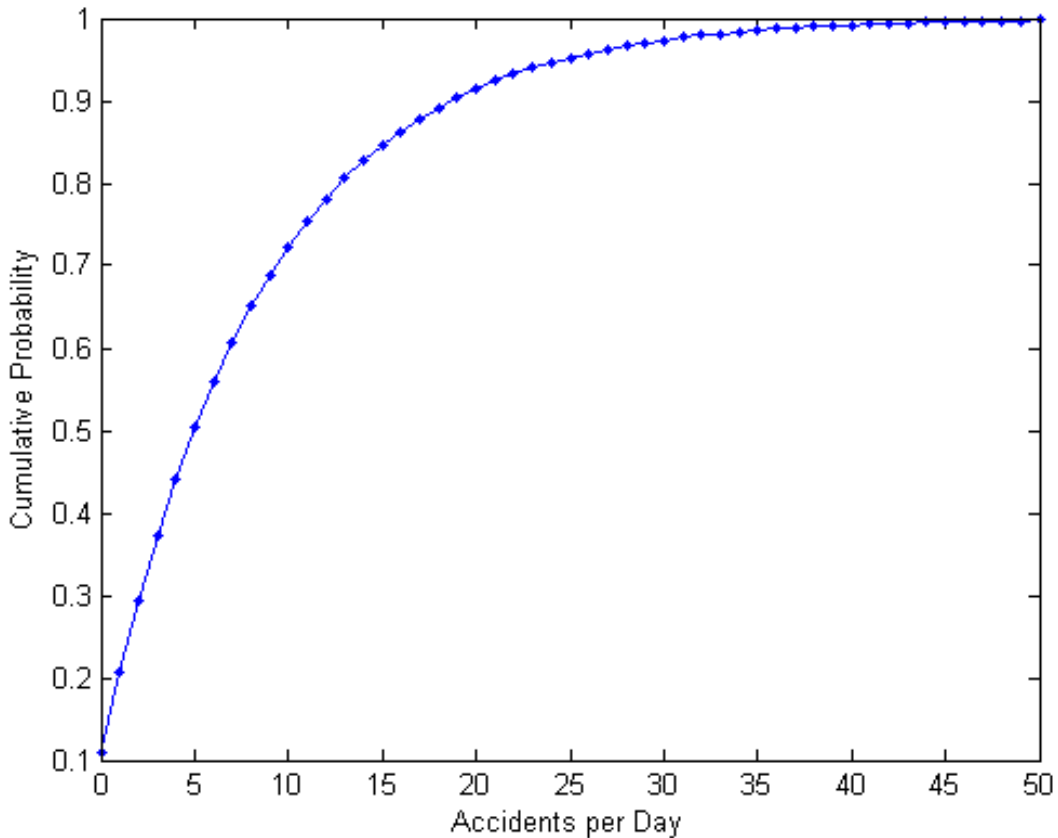
```
[phat,pci] = nbinfit(accident)
```

```
phat =  
  
    1.0060    0.1109
```

```
pci =  
  
    0.2152    0.0171  
    1.7968    0.2046
```

It is difficult to give a physical interpretation in this case to the individual parameters. However, the estimated parameters can be used in a model for the number of daily accidents. For example, a plot of the estimated cumulative probability function shows that while there is an estimated 10% chance of no accidents on a given day, there is also about a 10% chance that there will be 20 or more accidents.

```
plot(0:50,nbincdf(0:50,phat(1),phat(2)),'.-');  
xlabel('Accidents per Day')  
ylabel('Cumulative Probability')
```



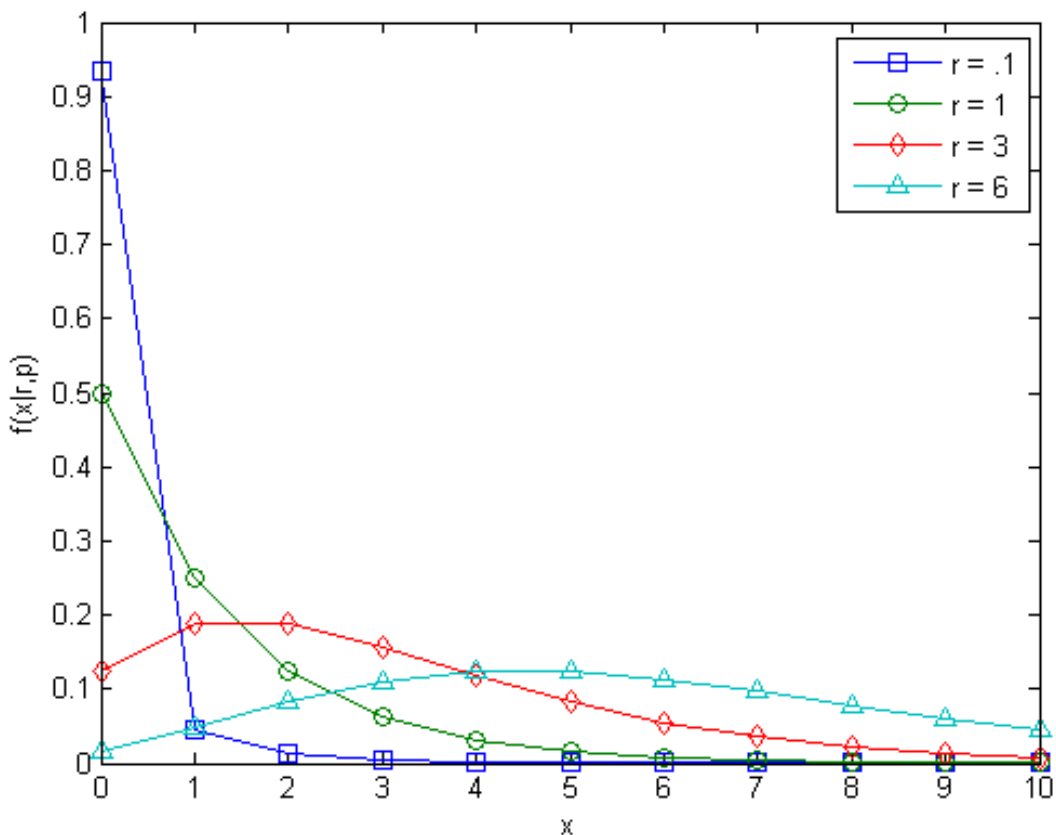
## Example

### Compute and Plot Negative Binomial Distribution PDF

This example shows how to compute and plot the PDF for a negative binomial distribution.

Compute and plot the pdf using four different values for the parameter  $r$ , the desired number of successes: .1, 1, 3, and 6. In each case, the probability of success  $p$  is .5.

```
x = 0:10;
plot(x,nbinpdf(x,.1,.5),'s-', ...
      x,nbinpdf(x,1,.5),'o-', ...
      x,nbinpdf(x,3,.5),'d-', ...
      x,nbinpdf(x,6,.5),'^-',);
legend({'r = .1' 'r = 1' 'r = 3' 'r = 6'})
xlabel('x')
ylabel('f(x|r,p)')
```



The plot shows that the negative binomial distribution can take on a variety of shapes, ranging from very skewed to nearly symmetric, depending on the value of  $r$ .

## Noncentral Chi-Square Distribution

### In this section...

“Definition” on page B-104

“Background” on page B-104

“Example” on page B-105

### Definition

There are many equivalent formulas for the noncentral chi-square distribution function. One formulation uses a modified Bessel function of the first kind. Another uses the generalized Laguerre polynomials. The cumulative distribution function is computed using a weighted sum of  $\chi^2$  probabilities with the weights equal to the probabilities of a Poisson distribution. The Poisson parameter is one-half of the noncentrality parameter of the noncentral chi-square

$$F(x | \nu, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) \Pr[\chi_{\nu+2j}^2 \leq x]$$

where  $\delta$  is the noncentrality parameter.

### Background

The  $\chi^2$  distribution is actually a simple special case of the noncentral chi-square distribution. One way to generate random numbers with a  $\chi^2$  distribution (with  $\nu$  degrees of freedom) is to sum the squares of  $\nu$  standard normal random numbers (mean equal to zero.)

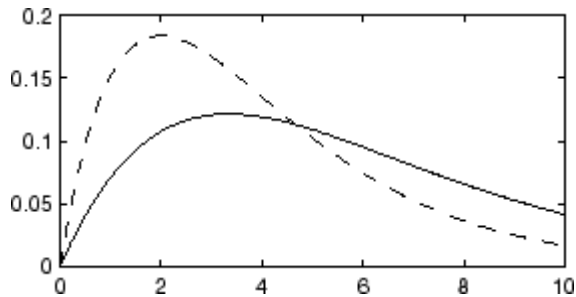
What if the normally distributed quantities have a mean other than zero? The sum of squares of these numbers yields the noncentral chi-square distribution. The noncentral chi-square distribution requires two parameters: the degrees of freedom and the noncentrality parameter. The noncentrality parameter is the sum of the squared means of the normally distributed quantities.

The noncentral chi-square has scientific application in thermodynamics and signal processing. The literature in these areas may refer to it as the “Rician Distribution” on page B-123 or generalized “Rayleigh Distribution” on page B-121.

### Example

The following commands generate a plot of the noncentral chi-square pdf.

```
x = (0:0.1:10)';  
p1 = ncx2pdf(x,4,2);  
p = chi2pdf(x,4);  
plot(x,p,'-',x,p1,'-')
```



## Noncentral F Distribution

### In this section...

“Definition” on page B-106

“Background” on page B-106

“Example” on page B-107

### Definition

Similar to the noncentral  $\chi^2$  distribution, the toolbox calculates noncentral  $F$  distribution probabilities as a weighted sum of incomplete beta functions using Poisson probabilities as the weights.

$$F(x | v_1, v_2, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta\right)^j}{j!} e^{-\frac{\delta}{2}} \right) I\left(\frac{v_1 \cdot x}{v_2 + v_1 \cdot x} \middle| \frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

$I(x | a, b)$  is the incomplete beta function with parameters  $a$  and  $b$ , and  $\delta$  is the noncentrality parameter.

### Background

As with the  $\chi^2$  distribution, the  $F$  distribution is a special case of the noncentral  $F$  distribution. The  $F$  distribution is the result of taking the ratio of  $\chi^2$  random variables each divided by its degrees of freedom.

If the numerator of the ratio is a noncentral chi-square random variable divided by its degrees of freedom, the resulting distribution is the noncentral  $F$  distribution.

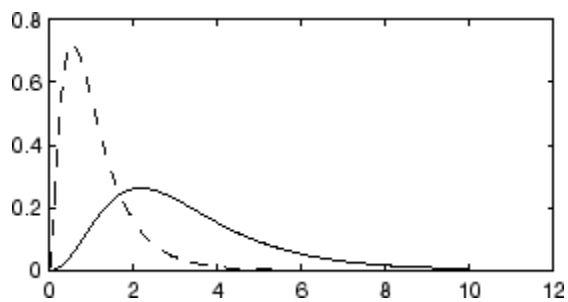
The main application of the noncentral  $F$  distribution is to calculate the power of a hypothesis test relative to a particular alternative.



## Example

The following commands generate a plot of the noncentral  $F$  pdf.

```
x = (0.01:0.1:10.01)';  
p1 = ncfpdf(x,5,20,10);  
p = fpdf(x,5,20);  
plot(x,p,'-',x,p1,'-')
```



## Noncentral $t$ Distribution

### In this section...

“Definition” on page B-108

“Background” on page B-108

“Example” on page B-109

### Definition

The most general representation of the noncentral  $t$  distribution is quite complicated. Johnson and Kotz [60] give a formula for the probability that a noncentral  $t$  variate falls in the range  $[-u, u]$ .

$$P(-u < x < u | \nu, \delta) = \sum_{j=0}^{\infty} \left( \frac{\left(\frac{1}{2}\delta^2\right)^j}{j!} e^{-\frac{\delta^2}{2}} \right) I\left(\frac{u^2}{\nu + u^2} \middle| \frac{1}{2} + j, \frac{\nu}{2}\right)$$

$I(x | \nu, \delta)$  is the incomplete beta function with parameters  $\nu$  and  $\delta$ .  $\delta$  is the noncentrality parameter, and  $\nu$  is the number of degrees of freedom.

### Background

The noncentral  $t$  distribution is a generalization of Student’s  $t$  distribution.

Student’s  $t$  distribution with  $n - 1$  degrees of freedom models the  $t$ -statistic

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

where  $\bar{x}$  is the sample mean and  $s$  is the sample standard deviation of a random sample of size  $n$  from a normal population with mean  $\mu$ . If the population mean is actually  $\mu_0$ , then the  $t$ -statistic has a noncentral  $t$  distribution with noncentrality parameter

$$\delta = \frac{\mu_0 - \mu}{\sigma / \sqrt{n}}$$

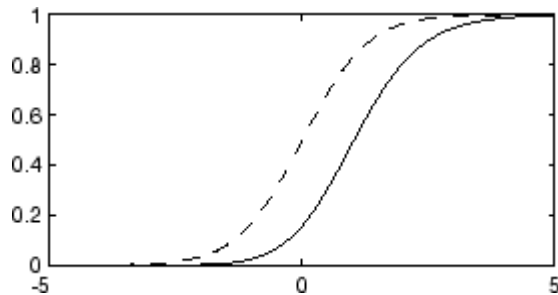
The noncentrality parameter is the normalized difference between  $\mu_0$  and  $\mu$ .

The noncentral  $t$  distribution gives the probability that a  $t$  test will correctly reject a false null hypothesis of mean  $\mu$  when the population mean is actually  $\mu_0$ ; that is, it gives the power of the  $t$  test. The power increases as the difference  $\mu_0 - \mu$  increases, and also as the sample size  $n$  increases.

### Example

The following commands generate a plot of the noncentral  $t$  pdf.

```
x = (-5:0.1:5)';  
p1 = nctcdf(x,10,1);  
p = tcdf(x,10);  
plot(x,p,'-',x,p1,'-')
```



## Normal Distribution

In this section...
“Definition” on page B-110
“Background” on page B-110
“Parameters” on page B-111
“Example” on page B-112

### Definition

The normal pdf is

$$y = f(x | \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

### Background

The normal distribution is a two-parameter family of curves. The first parameter,  $\mu$ , is the mean. The second,  $\sigma$ , is the standard deviation. The standard normal distribution (written  $\Phi(x)$ ) sets  $\mu$  to 0 and  $\sigma$  to 1.

$\Phi(x)$  is functionally related to the error function, *erf*.

$$\text{erf}(x) = 2\Phi(x\sqrt{2}) - 1$$

The first use of the normal distribution was as a continuous approximation to the binomial.

The usual justification for using the normal distribution for modeling is the Central Limit Theorem, which states (roughly) that the sum of independent samples from any distribution with finite mean and variance converges to the normal distribution as the sample size goes to infinity.

## Parameters

To use statistical parameters such as mean and standard deviation reliably, you need to have a good estimator for them. The maximum likelihood estimates (MLEs) provide one such estimator. However, an MLE might be biased, which means that its expected value of the parameter might not equal the parameter being estimated. For example, an MLE is biased for estimating the variance of a normal distribution. An unbiased estimator that is commonly used to estimate the parameters of the normal distribution is the *minimum variance unbiased estimator (MVUE)*. The MVUE has the minimum variance of all unbiased estimators of a parameter.

The MVUEs of parameters  $\mu$  and  $\sigma^2$  for the normal distribution are the sample mean and variance. The sample mean is also the MLE for  $\mu$ . The following are two common formulas for the variance.

$$s^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (\text{B-1})$$

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (\text{B-2})$$

where

$$\bar{x} = \sum_{i=1}^n \frac{x_i}{n}$$

Equation 1 is the maximum likelihood estimator for  $\sigma^2$ , and equation 2 is the MVUE.

As an example, suppose you want to estimate the mean,  $\mu$ , and the variance,  $\sigma^2$ , of the heights of all fourth grade children in the United States. The function `normfit` returns the MVUE for  $\mu$ , the square root of the MVUE for  $\sigma^2$ , and confidence intervals for  $\mu$  and  $\sigma^2$ . Here is a playful example modeling the heights in inches of a randomly chosen fourth grade class.

```
height = normrnd(50,2,30,1);           % Simulate heights.
[mu,s,muci,sci] = normfit(height)
```

```
mu =  
  50.2025
```

```
s =  
  1.7946
```

```
muci =  
  49.5210  
  50.8841
```

```
sci =  
  1.4292  
  2.4125
```

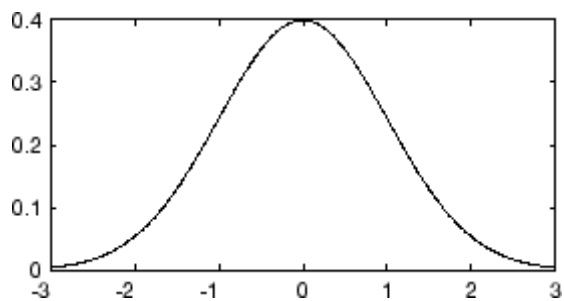
Note that  $s^2$  is the MVUE of the variance.

```
s^2
```

```
ans =  
  3.2206
```

### **Example**

The plot shows the bell curve of the standard normal pdf, with  $\mu = 0$  and  $\sigma = 1$ .



## **Pareto Distribution**

See “Generalized Pareto Distribution” on page B-53.

## **Pearson System**

See “Pearson and Johnson Systems” on page 6-26.



# Piecewise Linear Distribution

In this section...
“Overview” on page B-115
“Parameters” on page B-115
“Cumulative Distribution Function” on page B-115
“Relationship to Other Distributions” on page B-117

## Overview

The piecewise linear distribution creates a nonparametric representation of the cumulative distribution function (cdf) by linearly connecting the known cdf values from the sample data.

## Parameters

The piecewise linear distribution uses the following parameters.

Parameter	Description
x	Vector of x values at which the cdf changes slope
Fx	Vector of cdf values that correspond to each value in x

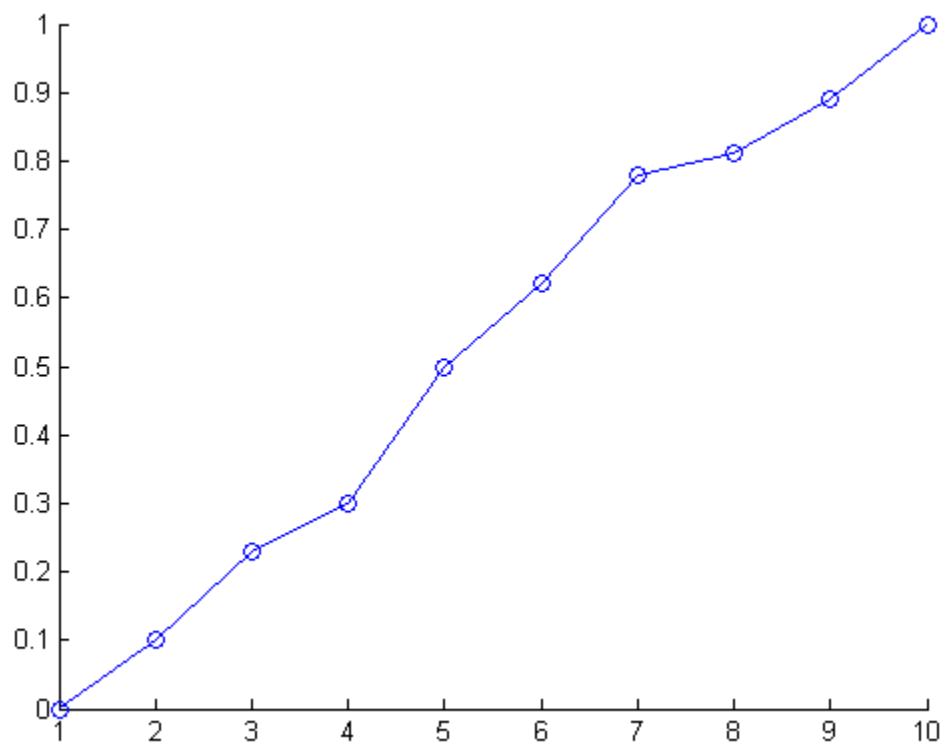
## Cumulative Distribution Function

The piecewise linear distribution constructs a continuous cumulative distribution function (cdf) by connecting the empirical cdf values such that the cdf increases linearly between  $x(j)$  and  $x(j + 1)$ .

This plot shows how the piecewise linear distribution constructs a continuous cdf from discrete values in the sample data.

```
% Create a piecewise linear distribution object
x = [1:10];
Fx = [0 .1 .23 .3 .5 .62 .78 .81 .89 1];
```

```
pd = makedist('piecelinear',x,Fx);  
  
% Compute and plot the cdf  
y = cdf(pd,x);  
figure;  
scatter(x,y);  
hold on;  
plot(x,y);  
hold off;
```



## **Relationship to Other Distributions**

The piecewise linear distribution is a continuous version of the discrete empirical cumulative distribution function (ecdf).

## Poisson Distribution

In this section...
“Definition” on page B-118
“Background” on page B-118
“Parameters” on page B-119
“Examples” on page B-119

### Definition

The Poisson pdf is

$$f(x | \lambda) = \frac{\lambda^x}{x!} e^{-\lambda}; x = 0, 1, 2, \dots, \infty.$$

### Background

The Poisson distribution is appropriate for applications that involve counting the number of times a random event occurs in a given amount of time, distance, area, etc. Sample applications that involve Poisson distributions include the number of Geiger counter clicks per second, the number of people walking into a store in an hour, and the number of flaws per 1000 feet of video tape.

The Poisson distribution is a one-parameter discrete distribution that takes nonnegative integer values. The parameter,  $\lambda$ , is both the mean and the variance of the distribution. Thus, as the size of the numbers in a particular sample of Poisson random numbers gets larger, so does the variability of the numbers.

The Poisson distribution is the limiting case of a binomial distribution where  $N$  approaches infinity and  $p$  goes to zero while  $Np = \lambda$ .

The Poisson and exponential distributions are related. If the number of counts follows the Poisson distribution, then the interval between individual counts follows the exponential distribution.

## Parameters

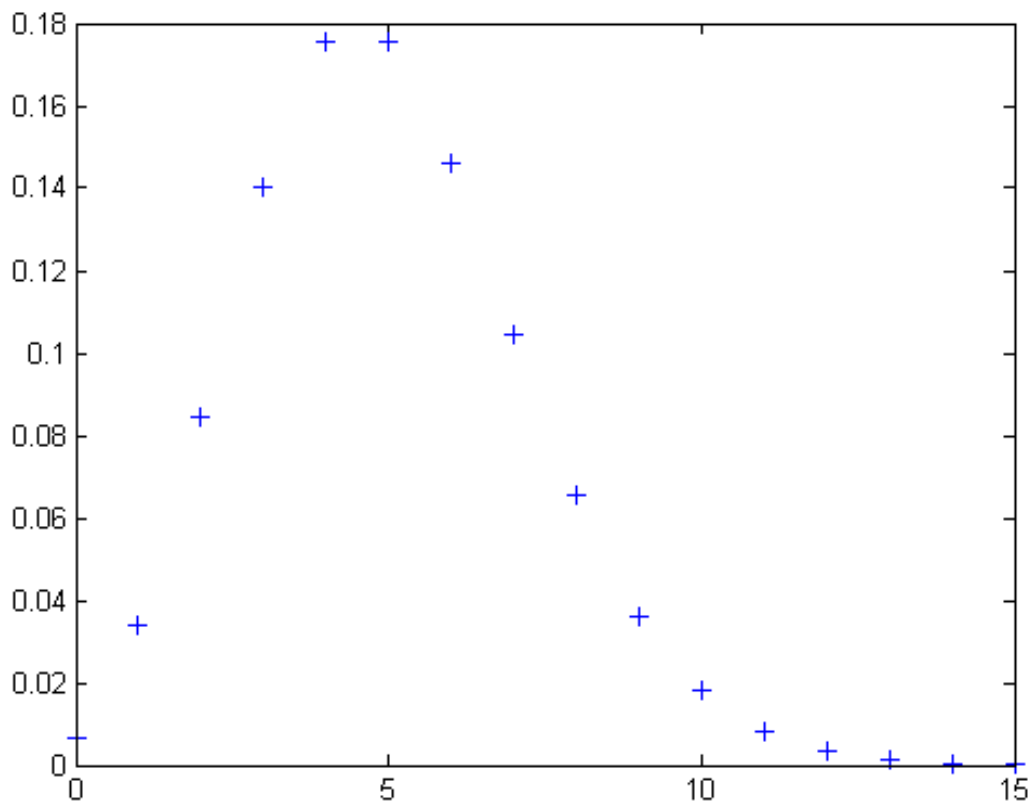
The MLE and the MVUE of the Poisson parameter,  $\lambda$ , is the sample mean. The sum of independent Poisson random variables is also Poisson distributed with the parameter equal to the sum of the individual parameters. This is used to calculate confidence intervals  $\lambda$ . As  $\lambda$  gets large the Poisson distribution can be approximated by a normal distribution with  $\mu = \lambda$  and  $\sigma^2 = \lambda$ . This approximation is used to calculate confidence intervals for values of  $\lambda$  greater than 100.

## Examples

### Compute and Plot Poisson Distribution PDF

This example shows how to compute and plot the pdf of a Poisson distribution with parameter `lambda = 5`.

```
x = 0:15;  
y = poisspdf(x,5);  
plot(x,y, '+')
```



# Rayleigh Distribution

## In this section...

“Definition” on page B-121

“Background” on page B-121

“Parameters” on page B-122

“Example” on page B-122

## Definition

The Rayleigh pdf is

$$y = f(x | b) = \frac{x}{b^2} e^{\left(\frac{-x^2}{2b^2}\right)}$$

## Background

The Rayleigh distribution is a special case of the Weibull distribution. If  $A$  and  $B$  are the parameters of the Weibull distribution, then the Rayleigh distribution with parameter  $b$  is equivalent to the Weibull distribution with parameters  $A = \sqrt{2}b$  and  $B = 2$ .

If the component velocities of a particle in the  $x$  and  $y$  directions are two independent normal random variables with zero means and equal variances, then the distance the particle travels per unit time is distributed Rayleigh.

In communications theory, Nakagami distributions, Rician distributions, and Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

## Parameters

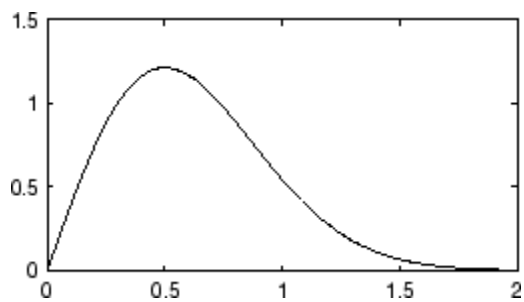
The `raylfit` function returns the MLE of the Rayleigh parameter. This estimate is

$$b = \sqrt{\frac{1}{2n} \sum_{i=1}^n x_i^2}$$

## Example

The following commands generate a plot of the Rayleigh pdf.

```
x = [0:0.01:2];  
p = raylpdf(x,0.5);  
plot(x,p)
```





## Rician Distribution

### In this section...

“Definition” on page B-123

“Background” on page B-123

“Parameters” on page B-123

### Definition

The Rician distribution has the density function

$$I_0\left(\frac{xs}{\sigma^2}\right)\frac{x}{\sigma^2}e^{-\left(\frac{x^2+s^2}{2\sigma^2}\right)}$$

with noncentrality parameter  $s \geq 0$  and scale parameter  $\sigma > 0$ , for  $x > 0$ .  $I_0$  is the zero-order modified Bessel function of the first kind. If  $x$  has a Rician distribution with parameters  $s$  and  $\sigma$ , then  $(x/\sigma)^2$  has a noncentral chi-square distribution with two degrees of freedom and noncentrality parameter  $(s/\sigma)^2$ .

### Background

In communications theory, Nakagami distributions, Rician distributions, and Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. Rayleigh and Nakagami distributions are used to model dense scatters, while Rician distributions model fading with a stronger line-of-sight. Nakagami distributions can be reduced to Rayleigh distributions, but give more control over the extent of the fading.

### Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitting app.

## Student's $t$ Distribution

**In this section...**

“Definition” on page B-124

“Background” on page B-124

“Example” on page B-125

### Definition

Student's  $t$  pdf is

$$y = f(x | \nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \frac{1}{\sqrt{\nu\pi}} \frac{1}{\left(1 + \frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

where  $\Gamma(\cdot)$  is the Gamma function.

### Background

The  $t$  distribution is a family of curves depending on a single parameter  $\nu$  (the degrees of freedom). As  $\nu$  goes to infinity, the  $t$  distribution approaches the standard normal distribution.

W. S. Gossett discovered the distribution through his work at the Guinness brewery. At the time, Guinness did not allow its staff to publish, so Gossett used the pseudonym “Student.”

If  $x$  is a random sample of size  $n$  from a normal distribution with mean  $\mu$ , then the statistic

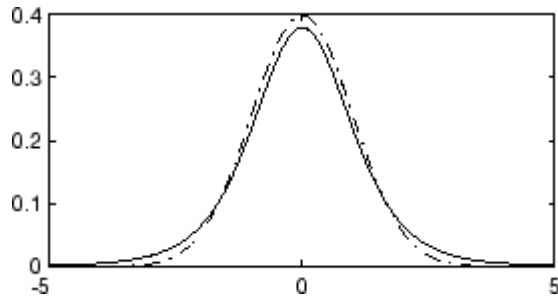
$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

where  $\bar{x}$  is the sample mean and  $s$  is the sample standard deviation, has Student's  $t$  distribution with  $n - 1$  degrees of freedom.

### Example

The plot compares the  $t$  distribution with  $\nu = 5$  (solid line) to the shorter tailed, standard normal distribution (dashed line).

```
x = -5:0.1:5;  
y = tpdf(x,5);  
z = normpdf(x,0,1);  
plot(x,y,'-',x,z,'-.-')
```



## t Location-Scale Distribution

### In this section...

“Definition” on page B-126

“Background” on page B-126

“Parameters” on page B-126

### Definition

The  $t$  location-scale distribution has the density function

$$\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sigma\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left[ \frac{\nu + \left(\frac{x-\mu}{\sigma}\right)^2}{\nu} \right]^{-\left(\frac{\nu+1}{2}\right)}$$

with location parameter  $\mu$ , scale parameter  $\sigma > 0$ , and shape parameter  $\nu > 0$ . If  $x$  has a  $t$  location-scale distribution, with parameters  $\mu$ ,  $\sigma$ , and  $\nu$ , then

$$\frac{x - \mu}{\sigma}$$

has a Student’s  $t$  distribution with  $\nu$  degrees of freedom.

### Background

The  $t$  location-scale distribution is useful for modeling data distributions with heavier tails (more prone to outliers) than the normal distribution. It approaches the normal distribution as  $\nu$  approaches infinity, and smaller values of  $\nu$  yield heavier tails.

### Parameters

To estimate distribution parameters, use `mle` or the Distribution Fitting app.

# Triangular Distribution

## In this section...

“Overview” on page B-127

“Parameters” on page B-127

“Probability Density Function” on page B-128

“Cumulative Distribution Function” on page B-129

“Descriptive Statistics” on page B-131

## Overview

The triangular distribution provides a simplistic representation of the probability distribution when limited sample data is available. Its parameters are the minimum, maximum, and mode of the data. Common applications include business and economic simulations, project management planning, natural phenomena modeling, and audio dithering.

## Parameters

The triangular distribution uses the following parameters.

Parameter	Description	Constraints
$a$	Lower limit	$a \leq b$
$b$	Peak location	$a \leq b \leq c$
$c$	Upper limit	$c \geq b$

## Parameter Estimation

Typically, you estimate triangular distribution parameters using subjectively reasonable values based on the sample data. You can estimate the lower and upper limit parameters  $a$  and  $c$  using the minimum and maximum values of the sample data, respectively. You can estimate the peak location parameter  $b$  using the sample mean, median, mode, or any other subjectively reasonable estimate of the population mode.

## Probability Density Function

The probability density function (pdf) of the triangular distribution is

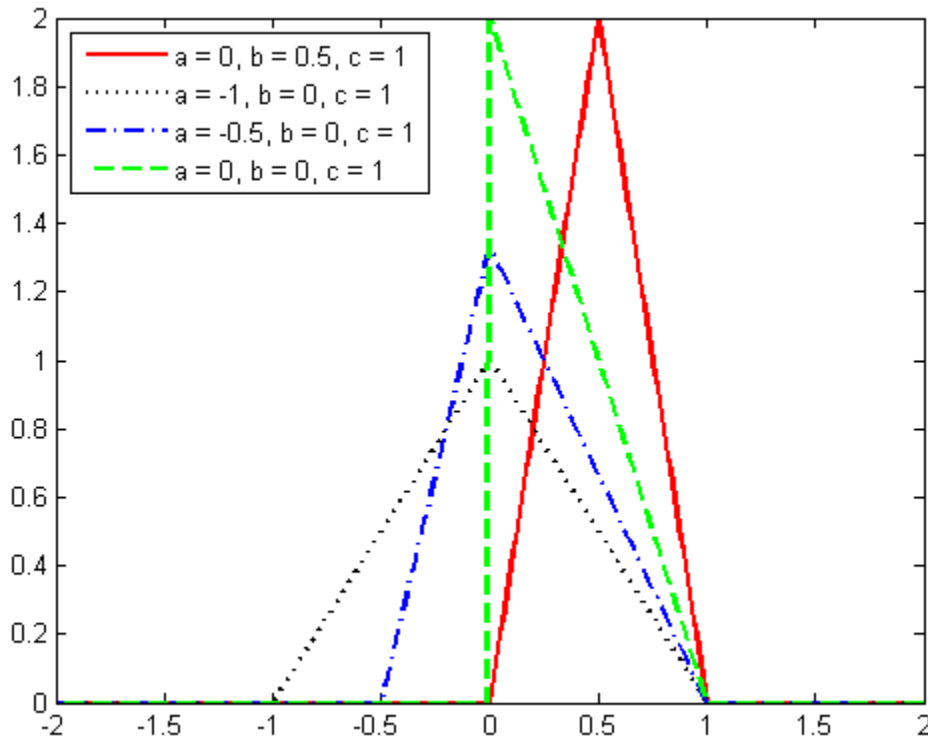
$$f(x|a,b,c) = \begin{cases} \frac{2(x-a)}{(c-a)(b-a)} & ; a \leq x \leq b \\ \frac{2(c-x)}{(c-a)(c-b)} & ; b < x \leq c \\ 0 & ; x < a, x > c \end{cases} .$$

This plot shows how changing the value of the parameters  $a$ ,  $b$ , and  $c$  alters the shape of the pdf.

```
% Create four distribution objects with different parameters
pd1 = makedist('Triangular');
pd2 = makedist('Triangular','a',-1,'b',0,'c',1);
pd3 = makedist('Triangular','a',-.5,'b',0,'c',1);
pd4 = makedist('Triangular','a',0,'b',0,'c',1);

% Compute the pdfs
x = -2:.01:2;
pdf1 = pdf(pd1,x);
pdf2 = pdf(pd2,x);
pdf3 = pdf(pd3,x);
pdf4 = pdf(pd4,x);

% Plot the pdfs
figure;
plot(x,pdf1,'r','LineWidth',2)
hold on;
plot(x,pdf2,'k','LineWidth',2);
plot(x,pdf3,'b-.','LineWidth',2);
plot(x,pdf4,'g--','LineWidth',2);
legend({'a = 0, b = 0.5, c = 1','a = -1, b = 0, c = 1',...
       'a = -0.5, b = 0, c = 1','a = 0, b = 0, c = 1'},'Location','NW');
hold off;
```



As the distance between  $a$  and  $c$  increases, the density at any particular value within the distribution boundaries decreases. Because the density function integrates to 1, the height of the pdf plot decreases as its width increases. The location of the peak parameter  $b$  determines whether the pdf skews right or left, or if it is symmetrical.

## Cumulative Distribution Function

The cumulative distribution function (cdf) of the triangular distribution is

$$F(x|a,b,c) = \begin{cases} 0 & , x < a \\ \frac{(x-a)^2}{(c-a)(b-a)} & , a \leq x \leq b \\ 1 - \frac{(c-x)^2}{(c-a)(c-b)} & , b < x \leq c \\ 1 & , x > c \end{cases} .$$

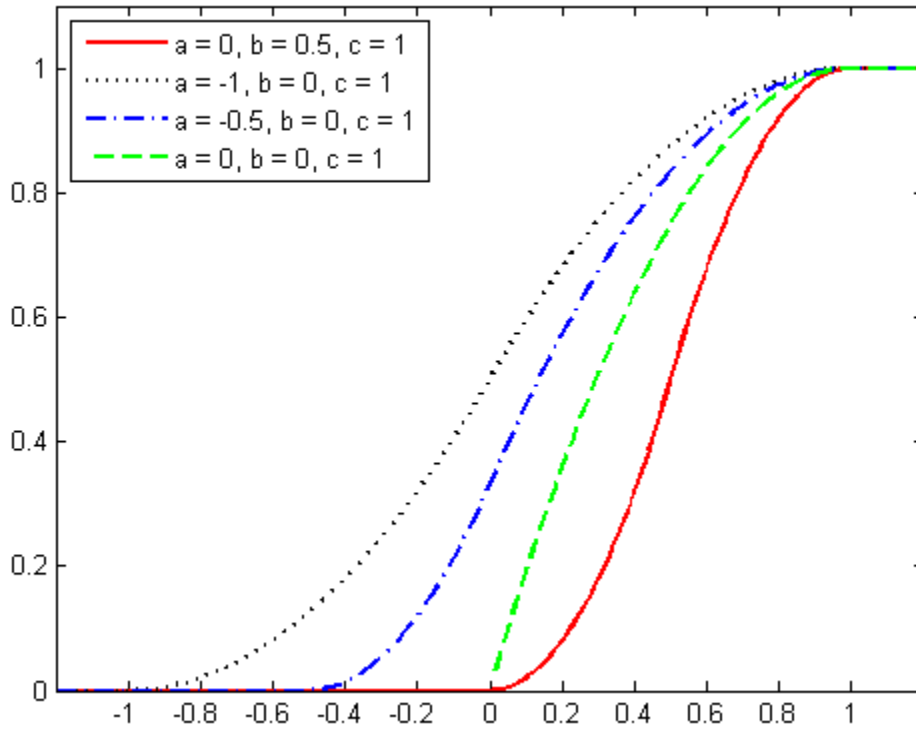
This plot shows how changing the value of the parameters  $a$ ,  $b$ , and  $c$  alters the shape of the cdf.

```
% Create four distribution objects with different parameters
pd1 = makedist('Triangular');
pd2 = makedist('Triangular','a',-1,'b',0,'c',1);
pd3 = makedist('Triangular','a',-.5,'b',0,'c',1);
pd4 = makedist('Triangular','a',0,'b',0,'c',1);

% Compute the cdfs
x = -1.2:.01:1.2;
cdf1 = cdf(pd1,x);
cdf2 = cdf(pd2,x);
cdf3 = cdf(pd3,x);
cdf4 = cdf(pd4,x);

% Plot the cdfs
figure;
plot(x,cdf1,'r','LineWidth',2)
xlim([-1.2 1.2]);
ylim([0 1.1]);hold on;
plot(x,cdf2,'k','LineWidth',2);
plot(x,cdf3,'b-.','LineWidth',2);
plot(x,cdf4,'g--','LineWidth',2);
legend({'a = 0, b = 0.5, c = 1','a = -1, b = 0, c = 1',...
       'a = -0.5, b = 0, c = 1','a = 0, b = 0, c = 1'},'Location','NW');
hold off;
```





## Descriptive Statistics

The mean and variance of the continuous uniform distribution are related to the parameters  $a$ ,  $b$ , and  $c$ .

The mean is

$$\text{mean} = \left( \frac{a + b + c}{3} \right).$$

The variance is

$$\text{var} = \left( \frac{a^2 + b^2 + c^2 - ab - ac - bc}{18} \right).$$

**Related  
Examples**

- “Generate Random Numbers Using the Triangular Distribution” on page 5-32

## Uniform Distribution (Continuous)

### In this section...

“Overview” on page B-133

“Parameters” on page B-133

“Probability Density Function” on page B-134

“Cumulative Distribution Function” on page B-135

“Descriptive Statistics” on page B-137

“Relationship to Other Distributions” on page B-138

### Overview

The uniform distribution (also called the rectangular distribution) is notable because it has a constant probability distribution function (pdf) between its two bounding parameters. It is appropriate for representing the distribution of round-off errors in values tabulated to a particular number of decimal places, and is used in random number generating techniques such as the inversion method.

### Parameters

The uniform distribution uses the following parameters.

Parameter	Description	Constraints
lower	Lower limit	$-\infty < \text{lower} < \text{upper}$
upper	Upper limit	$\text{lower} < \text{upper} < \infty$

### Parameter Estimation

The maximum likelihood estimator (MLE) for *lower* is the sample minimum. The MLE for *upper* is the sample maximum.

## Probability Density Function

The probability density function (pdf) of the continuous uniform distribution is

$$f(x | lower, upper) = \begin{cases} \left( \frac{1}{upper - lower} \right) & ; \text{ } lower \leq x \leq upper \\ 0 & ; \text{ } otherwise \end{cases} .$$

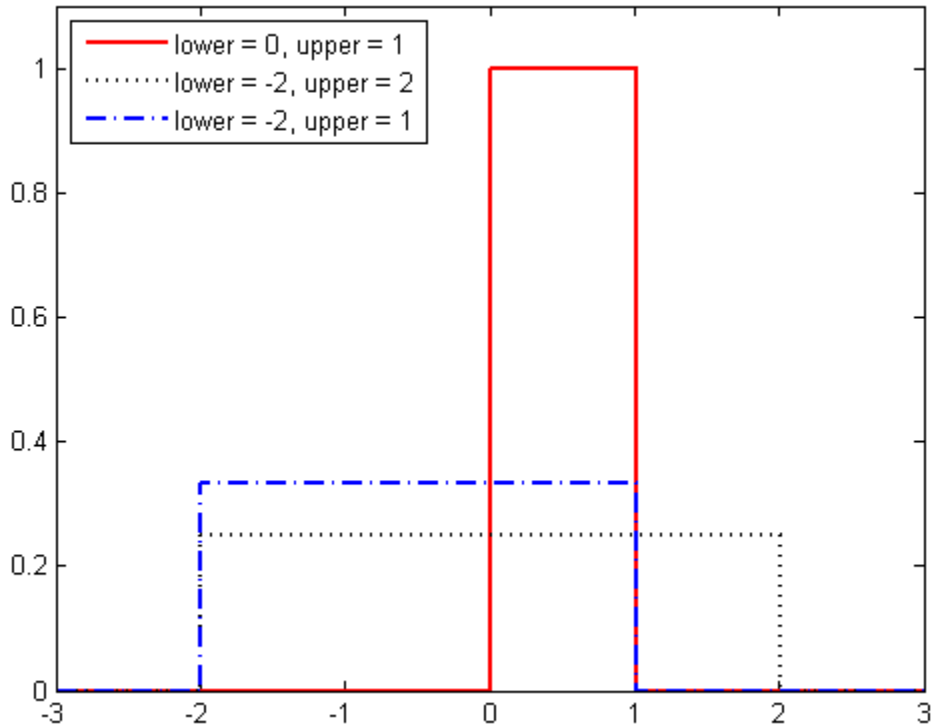
The pdf is constant between *lower* and *upper*.

This plot illustrates how changing the value of the parameters *lower* and *upper* affects the shape of the pdf.

```
% Create three distribution objects with different parameters
pd1 = makedist('Uniform');
pd2 = makedist('Uniform','lower',-2,'upper',2);
pd3 = makedist('Uniform','lower',-2,'upper',1);

% Compute the pdfs
x = -3:.01:3;
pdf1 = pdf(pd1,x);
pdf2 = pdf(pd2,x);
pdf3 = pdf(pd3,x);

% Plot the pdfs
figure;
stairs(x,pdf1,'r','LineWidth',2);
hold on;
stairs(x,pdf2,'k','LineWidth',2);
stairs(x,pdf3,'b-.','LineWidth',2);
ylim([0 1.1]);
legend({'lower = 0, upper = 1','lower = -2, upper = 2',...
       'lower = -2, upper = 1'},'Location','NW');
hold off;
```



As the distance between *lower* and *upper* increases, the density at any particular value within the distribution boundaries decreases. Because the density function integrates to 1, the height of the pdf plot decreases as its width increases.

## Cumulative Distribution Function

The cumulative distribution function (cdf) of the continuous uniform distribution is

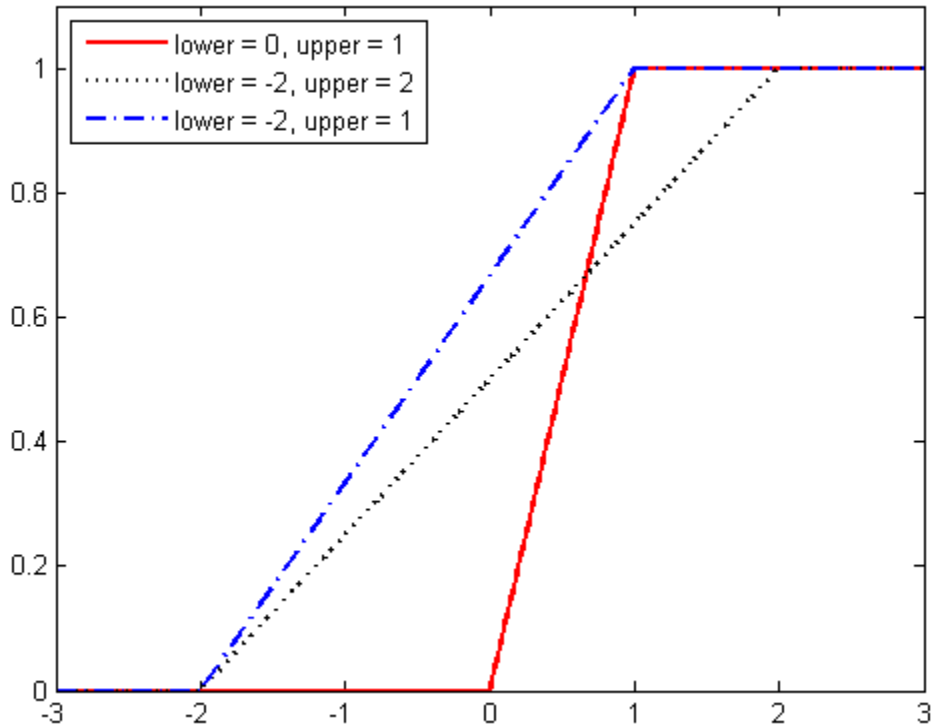
$$F(x | lower, upper) = \begin{cases} 0 & ; \quad x < lower \\ \frac{x - lower}{upper - lower} & ; \quad lower \leq x < upper . \\ 1 & ; \quad x \geq upper \end{cases}$$

This plot illustrates how changing the value of the parameters *lower* and *upper* affects the shape of the cdf.

```
% Create three distribution objects with different parameters
pd1 = makedist('Uniform');
pd2 = makedist('Uniform','lower',-2,'upper',2);
pd3 = makedist('Uniform','lower',-2,'upper',1);

% Compute the cdfs
x = -3:.01:3;
cdf1 = cdf(pd1,x);
cdf2 = cdf(pd2,x);
cdf3 = cdf(pd3,x);

% Plot the cdfs
figure;
plot(x,cdf1,'r','LineWidth',2);
hold on;
plot(x,cdf2,'k','LineWidth',2);
plot(x,cdf3,'b-.','LineWidth',2);
ylim([0 1.1]);
legend({'lower = 0, upper = 1','lower = -2, upper = 2',...
       'lower = -2, upper = 1'},'Location','NW');
hold off;
```



## Descriptive Statistics

The mean and variance of the continuous uniform distribution are related to the parameters *lower* and *upper*.

The mean is

$$\text{mean} = \frac{1}{2}(\text{lower} + \text{upper}).$$

The variance is

$$\text{var} = \frac{1}{12}(\text{upper} - \text{lower})^2 .$$

## Relationship to Other Distributions

The standard uniform distribution (lower = 0 and upper = 1) is a special case of the beta distribution obtained by setting the beta distribution parameters  $a = 1$  and  $b = 1$ .

The inversion method uses the continuous standard uniform distribution to generate random numbers for any other continuous distribution. The inversion method relies on the principle that continuous cumulative distribution functions (cdfs) range uniformly over the open interval (0,1). If  $u$  is a uniform random number on (0,1), then  $x = F^{-1}(u)$  generates a random number  $x$  from any continuous distribution with the specified cdf  $F$ .

### Related Examples

- “Generate Random Numbers Using Uniform Distribution Inversion” on page 5-101

### Concepts

- “Beta Distribution” on page B-5



## Uniform Distribution (Discrete)

### In this section...

“Definition” on page B-139

“Background” on page B-139

“Example” on page B-139

### Definition

The discrete uniform pdf is

$$y = f(x | N) = \frac{1}{N} I_{(1, \dots, N)}(x)$$

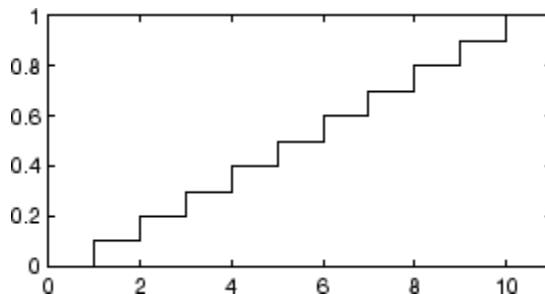
### Background

The discrete uniform distribution is a simple distribution that puts equal weight on the integers from one to  $N$ .

### Example

As for all discrete distributions, the cdf is a step function. The plot shows the discrete uniform cdf for  $N = 10$ .

```
x = 0:10;
y = unidcdf(x,10);
stairs(x,y)
set(gca, 'Xlim', [0 11])
```



Pick a random sample of 10 from a list of 553 items:

```
numbers = unidrnd(553,1,10)
```

```
numbers =
```

```
    293    372     5    213    37    231    380    326    515    468
```

## Weibull Distribution

### In this section...

“Definition” on page B-141

“Background” on page B-141

“Parameters” on page B-142

“Example” on page B-142

### Definition

The Weibull pdf is positive only for positive values of  $x$ , and is zero otherwise. For strictly positive values of the shape parameter  $b$  and scale parameter  $a$ , the density is

$$f(x | a, b) = \frac{b}{a} \left( \frac{x}{a} \right)^{b-1} e^{-(x/a)^b}.$$

### Background

Waloddi Weibull offered the distribution that bears his name as an appropriate analytical tool for modeling the breaking strength of materials. Current usage also includes reliability and lifetime modeling. The Weibull distribution is more flexible than the exponential for these purposes.

To see why, consider the hazard rate function (instantaneous failure rate). If  $f(t)$  and  $F(t)$  are the pdf and cdf of a distribution, then the hazard rate is

$$h(t) = \frac{f(t)}{1 - F(t)}$$

Substituting the pdf and cdf of the exponential distribution for  $f(t)$  and  $F(t)$  above yields a constant. The example below shows that the hazard rate for the Weibull distribution can vary.

## Parameters

Suppose you want to model the tensile strength of a thin filament using the Weibull distribution. The function `wblfit` gives maximum likelihood estimates and confidence intervals for the Weibull parameters.

```
strength = wblrnd(0.5,2,100,1);           % Simulated strengths.  
[p,ci] = wblfit(strength)
```

```
p =  
0.4715    1.9811
```

```
ci =  
  
    0.4248    1.7067  
    0.5233    2.2996
```

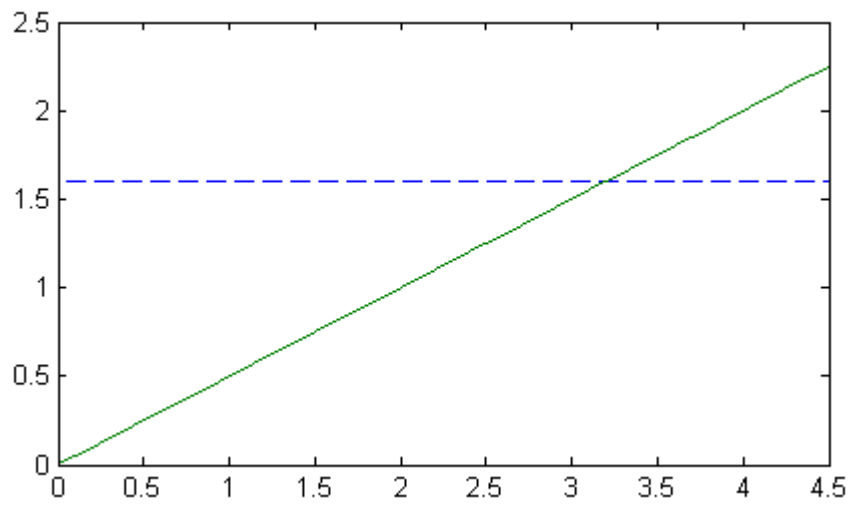
The default 95% confidence interval for each parameter contains the true value.

## Example

The exponential distribution has a constant hazard function, which is not generally the case for the Weibull distribution.

The plot shows the hazard functions for exponential (dashed line) and Weibull (solid line) distributions having the same mean life. The Weibull hazard rate here increases with age (a reasonable assumption).

```
t = 0:0.1:4.5;  
h1 = exppdf(t,0.6267) ./ (1-expcdf(t,0.6267));  
h2 = wblpdf(t,2,2) ./ (1-wblcdf(t,2,2));  
plot(t,h1,'--',t,h2,'-')
```



## Wishart Distribution

### In this section...

“Definition” on page B-144

“Background” on page B-144

“Example” on page B-145

“See Also” on page B-145

### Definition

The probability density function of the  $d$ -dimensional Wishart distribution is given by

$$y = f(X, \Sigma, \nu) = \frac{|X|^{(\nu-d-1)/2} e^{\left(-\frac{1}{2}\text{trace}(\Sigma^{-1}X)\right)}}{2^{(\nu d)/2} \pi^{(d(d-1))/4} |\Sigma|^{\nu/2} \Gamma(\nu/2) \dots \Gamma(\nu-(d-1))/2}$$

where  $X$  and  $\Sigma$  are  $d$ -by- $d$  symmetric positive definite matrices, and  $\nu$  is a scalar greater than  $d - 1$ . While it is possible to define the Wishart for singular  $\Sigma$ , the density cannot be written as above.

Only random matrix generation is supported for the Wishart distribution, including both singular and nonsingular  $\Sigma$ .

### Background

The Wishart distribution is a generalization of the univariate chi-square distribution to two or more variables. It is a distribution for symmetric positive semidefinite matrices, typically covariance matrices, the diagonal elements of which are each chi-square random variables. In the same way as the chi-square distribution can be constructed by summing the squares of independent, identically distributed, zero-mean univariate normal random variables, the Wishart distribution can be constructed by summing the inner products of independent, identically distributed, zero-mean multivariate normal random vectors.

The Wishart distribution is parameterized with a symmetric, positive semidefinite matrix,  $\Sigma$ , and a positive scalar degrees of freedom parameter,  $v$ .  $v$  is analogous to the degrees of freedom parameter of a univariate chi-square distribution, and  $\Sigma v$  is the mean of the distribution.

The Wishart distribution is often used as a model for the distribution of the sample covariance matrix for multivariate normal random data, after scaling by the sample size.

### Example

If  $x$  is a bivariate normal random vector with mean zero and covariance matrix

$$\Sigma = \begin{pmatrix} 1 & .5 \\ .5 & 2 \end{pmatrix}$$

then you can use the Wishart distribution to generate a sample covariance matrix without explicitly generating  $x$  itself. Notice how the sampling variability is quite large when the degrees of freedom is small.

```
Sigma = [1 .5; .5 2];
df = 10; S1 = wishrnd(Sigma,df)/df
```

```
S1 =
    1.7959    0.64107
    0.64107    1.5496
```

```
df = 1000; S2 = wishrnd(Sigma,df)/df
```

```
S2 =
    0.9842    0.50158
    0.50158    2.1682
```

### See Also

“Inverse Wishart Distribution” on page B-64,





# Bibliography

---

- [1] Atkinson, A. C., and A. N. Donev. *Optimum Experimental Designs*. New York: Oxford University Press, 1992.
- [2] Bates, D. M., and D. G. Watts. *Nonlinear Regression Analysis and Its Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1988.
- [3] Belsley, D. A., E. Kuh, and R. E. Welsch. *Regression Diagnostics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [4] Berry, M. W., et al. “Algorithms and Applications for Approximate Nonnegative Matrix Factorization.” *Computational Statistics and Data Analysis*. Vol. 52, No. 1, 2007, pp. 155–173.
- [5] Bookstein, Fred L. *Morphometric Tools for Landmark Data*. Cambridge, UK: Cambridge University Press, 1991.
- [6] Bouye, E., V. Durrleman, A. Nikeghbali, G. Riboulet, and T. Roncalli. “Copulas for Finance: A Reading Guide and Some Applications.” Working Paper. Groupe de Recherche Operationnelle, Credit Lyonnais, 2000.
- [7] Bowman, A. W., and A. Azzalini. *Applied Smoothing Techniques for Data Analysis*. New York: Oxford University Press, 1997.
- [8] Box, G. E. P., and N. R. Draper. *Empirical Model-Building and Response Surfaces*. Hoboken, NJ: John Wiley & Sons, Inc., 1987.
- [9] Box, G. E. P., W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. Hoboken, NJ: Wiley-Interscience, 1978.

- [10] Bratley, P., and B. L. Fox. "ALGORITHM 659 Implementing Sobol's Quasirandom Sequence Generator." *ACM Transactions on Mathematical Software*. Vol. 14, No. 1, 1988, pp. 88–100.
- [11] Breiman, L., J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Boca Raton, FL: CRC Press, 1984.
- [12] Breiman, L., et al., *Classification and Regression Trees*, Chapman & Hall, Boca Raton, 1993.
- [13] Bulmer, M. G. *Principles of Statistics*. Mineola, NY: Dover Publications, Inc., 1979.
- [14] Bury, K.. *Statistical Distributions in Engineering*. Cambridge, UK: Cambridge University Press, 1999.
- [15] Chatterjee, S., and A. S. Hadi. "Influential Observations, High Leverage Points, and Outliers in Linear Regression." *Statistical Science*. Vol. 1, 1986, pp. 379–416.
- [16] Collett, D. *Modeling Binary Data*. New York: Chapman & Hall, 2002.
- [17] Conover, W. J. *Practical Nonparametric Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1980.
- [18] Cook, R. D., and S. Weisberg. *Residuals and Influence in Regression*. New York: Chapman & Hall/CRC Press, 1983.
- [19] Cox, D. R., and D. Oakes. *Analysis of Survival Data*. London: Chapman & Hall, 1984.
- [20] Davidian, M., and D. M. Giltinan. *Nonlinear Models for Repeated Measurements Data*. New York: Chapman & Hall, 1995.
- [21] Deb, P., and M. Sefton. "The Distribution of a Lagrange Multiplier Test of Normality." *Economics Letters*. Vol. 51, 1996, pp. 123–130.
- [22] de Jong, S. "SIMPLS: An Alternative Approach to Partial Least Squares Regression." *Chemometrics and Intelligent Laboratory Systems*. Vol. 18, 1993, pp. 251–263.

- [23] Demidenko, E. *Mixed Models: Theory and Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 2004.
- [24] Delyon, B., M. Lavielle, and E. Moulines, *Convergence of a stochastic approximation version of the EM algorithm*, *Annals of Statistics*, 27, 94-128, 1999.
- [25] Dempster, A. P., N. M. Laird, and D. B. Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm." *Journal of the Royal Statistical Society*. Series B, Vol. 39, No. 1, 1977, pp. 1–37.
- [26] Devroye, L. *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986.
- [27] Dobson, A. J. *An Introduction to Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [28] Draper, N. R., and H. Smith. *Applied Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 1998.
- [29] Drezner, Z. "Computation of the Trivariate Normal Integral." *Mathematics of Computation*. Vol. 63, 1994, pp. 289–294.
- [30] Drezner, Z., and G. O. Wesolowsky. "On the Computation of the Bivariate Normal Integral." *Journal of Statistical Computation and Simulation*. Vol. 35, 1989, pp. 101–107.
- [31] DuMouchel, W. H., and F. L. O'Brien. "Integrating a Robust Option into a Multiple Regression Computing Environment." *Computer Science and Statistics: Proceedings of the 21st Symposium on the Interface*. Alexandria, VA: American Statistical Association, 1989.
- [32] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.
- [33] Efron, B., and R. J. Tibshirani. *An Introduction to the Bootstrap*. New York: Chapman & Hall, 1993.
- [34] Embrechts, P., C. Klüppelberg, and T. Mikosch. *Modelling Extremal Events for Insurance and Finance*. New York: Springer, 1997.

- [35] Evans, M., N. Hastings, and B. Peacock. *Statistical Distributions*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 1993, pp. 50–52, 73–74, 102–105, 147, 148.
- [36] Genz, A. “Numerical Computation of Rectangular Bivariate and Trivariate Normal and t Probabilities.” *Statistics and Computing*. Vol. 14, No. 3, 2004, pp. 251–260.
- [37] Genz, A., and F. Bretz. “Comparison of Methods for the Computation of Multivariate t Probabilities.” *Journal of Computational and Graphical Statistics*. Vol. 11, No. 4, 2002, pp. 950–971.
- [38] Genz, A., and F. Bretz. “Numerical Computation of Multivariate t Probabilities with Application to Power Calculation of Multiple Contrasts.” *Journal of Statistical Computation and Simulation*. Vol. 63, 1999, pp. 361–378.
- [39] Gibbons, J. D. *Nonparametric Statistical Inference*. New York: Marcel Dekker, 1985.
- [40] Goodall, C. R. “Computation Using the QR Decomposition.” *Handbook in Statistics*. Vol. 9, Amsterdam: Elsevier/North-Holland, 1993.
- [41] Hahn, Gerald J., and S. S. Shapiro. *Statistical Models in Engineering*. Hoboken, NJ: John Wiley & Sons, Inc., 1994, p. 95.
- [42] Hald, A. *Statistical Theory with Engineering Applications*. Hoboken, NJ: John Wiley & Sons, Inc., 1960.
- [43] Harman, H. H. *Modern Factor Analysis*. 3rd Ed. Chicago: University of Chicago Press, 1976.
- [44] Hastie, T., R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. New York: Springer, 2001.
- [45] Hochberg, Y., and A. C. Tamhane. *Multiple Comparison Procedures*. Hoboken, NJ: John Wiley & Sons, 1987.
- [46] Hoerl, A. E., and R. W. Kennard. “Ridge Regression: Applications to Nonorthogonal Problems.” *Technometrics*. Vol. 12, No. 1, 1970, pp. 69–82.

- [47] Hoerl, A. E., and R. W. Kennard. “Ridge Regression: Biased Estimation for Nonorthogonal Problems.” *Technometrics*. Vol. 12, No. 1, 1970, pp. 55–67.
- [48] Hogg, R. V., and J. Ledolter. *Engineering Statistics*. New York: MacMillan, 1987.
- [49] Holland, P. W., and R. E. Welsch. “Robust Regression Using Iteratively Reweighted Least-Squares.” *Communications in Statistics: Theory and Methods*, A6, 1977, pp. 813–827.
- [50] Hollander, M., and D. A. Wolfe. *Nonparametric Statistical Methods*. Hoboken, NJ: John Wiley & Sons, Inc., 1999.
- [51] Hong, H. S., and F. J. Hickernell. “ALGORITHM 823: Implementing Scrambled Digital Sequences.” *ACM Transactions on Mathematical Software*. Vol. 29, No. 2, 2003, pp. 95–109.
- [52] Huber, P. J. *Robust Statistics*. Hoboken, NJ: John Wiley & Sons, Inc., 1981.
- [53] Jackson, J. E. *A User’s Guide to Principal Components*. Hoboken, NJ: John Wiley and Sons, 1991.
- [54] Jain, A., and R. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ: Prentice-Hall, 1988.
- [55] Jarque, C. M., and A. K. Bera. “A test for normality of observations and regression residuals.” *International Statistical Review*. Vol. 55, No. 2, 1987, pp. 163–172.
- [56] Joe, S., and F. Y. Kuo. “Remark on Algorithm 659: Implementing Sobol’s Quasirandom Sequence Generator.” *ACM Transactions on Mathematical Software*. Vol. 29, No. 1, 2003, pp. 49–57.
- [57] Johnson, N., and S. Kotz. *Distributions in Statistics: Continuous Univariate Distributions-2*. Hoboken, NJ: John Wiley & Sons, Inc., 1970, pp. 130–148, 189–200, 201–219.
- [58] Johnson, N. L., N. Balakrishnan, and S. Kotz. *Continuous Multivariate Distributions*. Vol. 1. Hoboken, NJ: Wiley-Interscience, 2000.

- [59] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 1, Hoboken, NJ: Wiley-Interscience, 1993.
- [60] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*. Vol. 2, Hoboken, NJ: Wiley-Interscience, 1994.
- [61] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Discrete Multivariate Distributions*. Hoboken, NJ: Wiley-Interscience, 1997.
- [62] Johnson, N. L., S. Kotz, and A. W. Kemp. *Univariate Discrete Distributions*. Hoboken, NJ: Wiley-Interscience, 1993.
- [63] Jolliffe, I. T. *Principal Component Analysis*. 2nd ed., New York: Springer-Verlag, 2002.
- [64] Jöreskog, K. G. "Some Contributions to Maximum Likelihood Factor Analysis." *Psychometrika*. Vol. 32, 1967, pp. 443–482.
- [65] Kaufman L., and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons, Inc., 1990.
- [66] Kendall, David G. "A Survey of the Statistical Theory of Shape." *Statistical Science*. Vol. 4, No. 2, 1989, pp. 87–99.
- [67] Kocis, L., and W. J. Whiten. "Computational Investigations of Low-Discrepancy Sequences." *ACM Transactions on Mathematical Software*. Vol. 23, No. 2, 1997, pp. 266–294.
- [68] Kotz, S., and S. Nadarajah. *Extreme Value Distributions: Theory and Applications*. London: Imperial College Press, 2000.
- [69] Krzanowski, W. J. *Principles of Multivariate Analysis: A User's Perspective*. New York: Oxford University Press, 1988.
- [70] Lawless, J. F. *Statistical Models and Methods for Lifetime Data*. Hoboken, NJ: Wiley-Interscience, 2002.
- [71] Lawley, D. N., and A. E. Maxwell. *Factor Analysis as a Statistical Method*. 2nd ed. New York: American Elsevier Publishing, 1971.

- [72] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for normality with mean and variance unknown.” *Journal of the American Statistical Association*. Vol. 62, 1967, pp. 399–402.
- [73] Lilliefors, H. W. “On the Kolmogorov-Smirnov test for the exponential distribution with mean unknown.” *Journal of the American Statistical Association*. Vol. 64, 1969, pp. 387–389.
- [74] Lindstrom, M. J., and D. M. Bates. “Nonlinear mixed-effects models for repeated measures data.” *Biometrics*. Vol. 46, 1990, pp. 673–687.
- [75] Little, Roderick J. A., and Donald B. Rubin. *Statistical Analysis with Missing Data*. 2nd ed., Hoboken, NJ: John Wiley & Sons, Inc., 2002.
- [76] Mardia, K. V., J. T. Kent, and J. M. Bibby. *Multivariate Analysis*. Burlington, MA: Academic Press, 1980.
- [77] Marquardt, D.W. “Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation.” *Technometrics*. Vol. 12, No. 3, 1970, pp. 591–612.
- [78] Marquardt, D. W., and R.D. Snee. “Ridge Regression in Practice.” *The American Statistician*. Vol. 29, No. 1, 1975, pp. 3–20.
- [79] Marsaglia, G., and W. W. Tsang. “A Simple Method for Generating Gamma Variables.” *ACM Transactions on Mathematical Software*. Vol. 26, 2000, pp. 363–372.
- [80] Marsaglia, G., W. Tsang, and J. Wang. “Evaluating Kolmogorov’s Distribution.” *Journal of Statistical Software*. Vol. 8, Issue 18, 2003.
- [81] Martinez, W. L., and A. R. Martinez. *Computational Statistics with MATLAB*. New York: Chapman & Hall/CRC Press, 2002.
- [82] Massey, F. J. “The Kolmogorov-Smirnov Test for Goodness of Fit.” *Journal of the American Statistical Association*. Vol. 46, No. 253, 1951, pp. 68–78.
- [83] Matousek, J. “On the L2-Discrepancy for Anchored Boxes.” *Journal of Complexity*. Vol. 14, No. 4, 1998, pp. 527–556.

- [84] McLachlan, G., and D. Peel. *Finite Mixture Models*. Hoboken, NJ: John Wiley & Sons, Inc., 2000.
- [85] McCullagh, P., and J. A. Nelder. *Generalized Linear Models*. New York: Chapman & Hall, 1990.
- [86] McGill, R., J. W. Tukey, and W. A. Larsen. "Variations of Boxplots." *The American Statistician*. Vol. 32, No. 1, 1978, pp. 12–16.
- [87] Meeker, W. Q., and L. A. Escobar. *Statistical Methods for Reliability Data*. Hoboken, NJ: John Wiley & Sons, Inc., 1998.
- [88] Meng, Xiao-Li, and Donald B. Rubin. "Maximum Likelihood Estimation via the ECM Algorithm." *Biometrika*. Vol. 80, No. 2, 1993, pp. 267–278.
- [89] Meyers, R. H., and D.C. Montgomery. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Hoboken, NJ: John Wiley & Sons, Inc., 1995.
- [90] Miller, L. H. "Table of Percentage Points of Kolmogorov Statistics." *Journal of the American Statistical Association*. Vol. 51, No. 273, 1956, pp. 111–121.
- [91] Milliken, G. A., and D. E. Johnson. *Analysis of Messy Data, Volume 1: Designed Experiments*. Boca Raton, FL: Chapman & Hall/CRC Press, 1992.
- [92] Montgomery, D. *Introduction to Statistical Quality Control*. Hoboken, NJ: John Wiley & Sons, 1991, pp. 369–374.
- [93] Montgomery, D. C. *Design and Analysis of Experiments*. Hoboken, NJ: John Wiley & Sons, Inc., 2001.
- [94] Mood, A. M., F. A. Graybill, and D. C. Boes. *Introduction to the Theory of Statistics*. 3rd ed., New York: McGraw-Hill, 1974. pp. 540–541.
- [95] Moore, J. *Total Biochemical Oxygen Demand of Dairy Manures*. Ph.D. thesis. University of Minnesota, Department of Agricultural Engineering, 1975.



- [96] Mosteller, F., and J. Tukey. *Data Analysis and Regression*. Upper Saddle River, NJ: Addison-Wesley, 1977.
- [97] Nelson, L. S. “Evaluating Overlapping Confidence Intervals.” *Journal of Quality Technology*. Vol. 21, 1989, pp. 140–141.
- [98] Patel, J. K., C. H. Kapadia, and D. B. Owen. *Handbook of Statistical Distributions*. New York: Marcel Dekker, 1976.
- [99] Pinheiro, J. C., and D. M. Bates. “Approximations to the log-likelihood function in the nonlinear mixed-effects model.” *Journal of Computational and Graphical Statistics*. Vol. 4, 1995, pp. 12–35.
- [100] Rice, J. A. *Mathematical Statistics and Data Analysis*. Pacific Grove, CA: Duxbury Press, 1994.
- [101] Rosipal, R., and N. Kramer. “Overview and Recent Advances in Partial Least Squares.” *Subspace, Latent Structure and Feature Selection: Statistical and Optimization Perspectives Workshop (SLSFS 2005), Revised Selected Papers (Lecture Notes in Computer Science 3940)*. Berlin, Germany: Springer-Verlag, 2006, pp. 34–51.
- [102] Sachs, L. *Applied Statistics: A Handbook of Techniques*. New York: Springer-Verlag, 1984, p. 253.
- [103] Searle, S. R., F. M. Speed, and G. A. Milliken. “Population marginal means in the linear model: an alternative to least-squares means.” *American Statistician*. 1980, pp. 216–221.
- [104] Seber, G. A. F. *Linear Regression Analysis*. Hoboken, NJ: Wiley-Interscience, 2003.
- [105] Seber, G. A. F. *Multivariate Observations*. Hoboken, NJ: John Wiley & Sons, Inc., 1984.
- [106] Seber, G. A. F., and C. J. Wild. *Nonlinear Regression*. Hoboken, NJ: Wiley-Interscience, 2003.
- [107] Sexton, Joe, and A. R. Swensen. “ECM Algorithms that Converge at the Rate of EM.” *Biometrika*. Vol. 87, No. 3, 2000, pp. 651–662.

- [108] Snedecor, G. W., and W. G. Cochran. *Statistical Methods*. Ames, IA: Iowa State Press, 1989.
- [109] Spath, H. *Cluster Dissection and Analysis: Theory, FORTRAN Programs, Examples*. Translated by J. Goldschmidt. New York: Halsted Press, 1985.
- [110] Stein, M. “Large sample properties of simulations using latin hypercube sampling.” *Technometrics*. Vol. 29, No. 2, 1987, pp. 143–151. Correction, Vol. 32, p. 367.
- [111] Stephens, M. A. “Use of the Kolmogorov-Smirnov, Cramer-Von Mises and Related Statistics Without Extensive Tables.” *Journal of the Royal Statistical Society*. Series B, Vol. 32, No. 1, 1970, pp. 115–122.
- [112] Street, J. O., R. J. Carroll, and D. Ruppert. “A Note on Computing Robust Regression Estimates via Iteratively Reweighted Least Squares.” *The American Statistician*. Vol. 42, 1988, pp. 152–154.
- [113] Student. “On the Probable Error of the Mean.” *Biometrika*. Vol. 6, No. 1, 1908, pp. 1–25.
- [114] Velleman, P. F., and D. C. Hoaglin. *Application, Basics, and Computing of Exploratory Data Analysis*. Pacific Grove, CA: Duxbury Press, 1981.
- [115] Weibull, W. “A Statistical Theory of the Strength of Materials.” *Ingeniors Vetenskaps Akademiens Handlingar*. Stockholm: Royal Swedish Institute for Engineering Research, No. 151, 1939.
- [116] Zahn, C. T. “Graph-theoretical methods for detecting and describing Gestalt clusters.” *IEEE Transactions on Computers*. Vol. C-20, Issue 1, 1971, pp. 68–86.

## A

- absolute deviation 3-5
- addedvarplot 21-2
- additive effects 8-9
- adjacent value 21-148
- adjacent values 4-8
- AIC. *See* Akaike Information Criterion
- Akaike Information Criterion (AIC) 5-117 21-24
- alternative hypotheses 7-3
- analysis of variance
  - F* distribution B-42
  - multivariate 8-39
  - N-way 8-12
  - one-way 8-3
  - two-way 8-9
- andrewsplot 21-25
- anova1 21-44
- anova2 21-51
- anovan 21-55
- Ansari-Bradley test 7-14
- aocool 8-27 21-72
- average linkage 21-1740

## B

- bacteria counts 8-4
- Bartlett multiple-sample test 7-16
- Bartlett's test 7-14
- barttest 21-78
- batch updates 21-1568
- Bayes Information Criterion (BIC) 5-118 21-103
- bbdesign 21-80
- Bernoulli distribution B-3
- Bernoulli random variables 21-118
- beta distribution B-5
- betacdf 21-90
- betafit 21-92
- betainv 21-94
- betalike 21-96
- betapdf 21-98

- betarnd 21-100
- betastat 21-102
- BIC. *See* Bayes Information Criterion
- binocdf 21-111
- binofit 21-113
- binoinv 21-115
- binomial distribution B-9
  - negative B-98
- binopdf 21-116
- binornd 21-118
- binostat 21-120
- biplot 21-121
- Birnbaum-Saunders distribution B-13
- bootci 21-131
- bootstrapping 3-16
- bootstrp 21-135
- box plots 4-7
- Box-Behnken designs 18-13
  - generating 21-80
- Box-Wilson designs 18-9
- boxplot 21-142
- Burr Type XII distribution B-15

## C

- candgen 21-178
- candidate sets 18-17
- canoncorr 21-182
- Canonical Maximum Likelihood (CML) 21-498
- capability 21-185
- capability studies 19-7
- capaplot 21-188
- case names
  - reading from file 21-190
  - writing to file 21-191
- caseread 21-190
- casewrite 21-191
- CCD. *See* central composite designs
- ccdesign 21-212
- cdf 21-216

- cdfplot 21-228
  - central composite designs (CCDs)
    - generating 21-212
    - types 18-9
  - Central Limit Theorem B-110
  - centroid linkage 21-1740
  - chi-square distribution B-28
  - chi-square goodness-of-fit test 7-14
  - chi-square variance test, one-sample 7-16
  - chi2cdf 21-238
  - chi2inv 21-250
  - chi2pdf 21-252
  - chi2rnd 21-254
  - chi2stat 21-255
  - cholcov 21-260
  - circuit boards 21-116
  - city block metric 16-10 21-2444 21-2451
  - classical multidimensional scaling
    - cmdscale function 21-367
    - overview 13-50
  - Classification
    - naive bayes 15-38
    - performance curves 15-41
    - tree bagging 16-149
  - classification trees
    - example 16-49
  - classifiers 15-2
  - classify 21-329
  - cluster 21-349
  - cluster analysis
    - hierarchical clustering 14-3
    - K-means clustering 14-22
    - overview 14-2
  - cluster trees
    - constructing clusters from 21-349
    - creating 21-1734
    - creating, from data 21-358
    - inconsistency coefficient 21-1438
  - cmdscale 21-367
  - CML. *See* Canonical Maximum Likelihood
  - combnk 21-411
  - common factors 13-95
  - comparisons, multiple 8-6
  - complete linkage 21-1740
  - Computing, parallel 20-1
  - confidence intervals
    - communicating results of hypothesis tests 7-4
  - confounding effects 18-5
  - confounding patterns 18-7
  - confusionmat 21-479
  - control charts 19-3
  - controlchart 21-482
  - controlrules 21-488
  - Cook's distance 21-2910
  - cophenet 21-493
  - cophenetic correlation coefficients 14-10 21-493
  - cophenetic distance 14-10
  - copulacdf 21-495
  - copulafit 21-497
  - copulaparam 21-503
  - copulapdf 21-505
  - copularnd 21-509
  - copulas 5-121 B-30
  - copulastat 21-507
  - cordexch 21-511
  - corr 21-516
  - corrcov 21-520
  - criterion function 13-71
  - crosstab 21-553
  - crossval 21-557
  - cumulative distribution function (cdf)
    - graphing an estimate 4-15
  - curse of dimensionality 13-2
  - cut variables 21-320 21-433 21-446 21-592 21-2902
- D**
- D-optimal designs

- generating candidate set 21-178
- overview 18-15
- data
  - landmark 13-62
- data sets
  - normalizing 14-4
  - statistical examples A-2
- daugment 21-645
- dcovary 21-650
- decision trees
  - computing error rate 21-3476
  - computing response values 21-3480
  - creating 21-3469
  - creating subtrees 21-3472
  - displaying 21-3466
  - fitting 21-3469
  - pruning 21-3472
- design matrix 11-25
- design of experiments
  - basic factors 18-6
  - confounding effects 18-5
  - D-optimal designs 18-15
  - fractional factorial designs 18-5
  - full factorial designs 18-3
  - generators 18-6
  - levels 18-3
  - Plackett-Burman designs 18-5
  - resolution 18-6 21-1202
  - response surface designs 18-9
  - two-level designs 18-4
- dffitool 21-684
- dimension reduction
  - common factor analysis 21-836
  - multivariate statistical methods 13-2
  - PCA from covariance matrix 21-2409
  - PCA from raw data matrix 21-2689
  - PCA residuals 21-2411
- discrete uniform distribution B-139
- discriminant analysis 15-3
- dissimilarity matrices

- creating 14-4
- distance matrices
  - creating 14-4
- distribution fitting
  - tool 5-40
- distributions
  - custom B-31
- disttool 21-718
- dummyvar 21-723
- Durbin-Watson test 7-14

## E

- effects
  - fixed 11-23
  - random 11-23
  - statistical 11-23
- efinv 21-779
- emission matrices
  - estimating 17-9
- equal variances
  - Bartlett multiple-sample test for 7-16
  - F*-test for 7-16
- erf B-110
- error function B-110
- Euclidean distance 16-9 21-2444 21-2450
- evcdf 21-775
- evfit 21-777
- evlike 21-801
- evpdf 21-802
- evrnd 21-803
- evstat 21-804
- expcdf 21-811
- expectation maximization (EM) algorithm
  - cluster analysis 14-2
  - Gaussian mixture models 5-110 14-30
- expfit 21-813
- expinv 21-820
- explike 21-822
- exponential distribution B-32

exppdf 21-826  
exprnd 21-827  
expstat 21-828  
extrapolated 21-2754  
extreme value distribution B-35  
extreme value fit 21-777

## F

*F* distribution B-41  
*F*-test, one-sample 7-16  
factor analysis  
    maximum likelihood 21-836  
factoran 21-836  
factorial designs  
    fractional 18-5  
    full 18-3  
    generating full 21-1216  
fcdf 21-851  
feature selection  
    overview 13-71  
    sequential 13-71  
feature transformation  
    overview 13-76  
ff2n 21-862  
filter methods  
    feature selection 21-3105  
finv 21-866  
folds  
    partition 21-605  
fpdf 21-1197  
fracfactgen 21-1202  
fractional factorial designs  
    overview 18-5  
friedman 21-1205  
Friedman's test 8-37  
Friedman's test 7-14  
frnd 21-1211  
fstat 21-1212  
fsurfht 21-1213

full factorial designs  
    generating 21-1216  
    overview 18-3  
fullfact 21-1216  
furthest neighbor linkage 21-1740

## G

gagerr 21-1217  
gamcdf 21-1222  
gamfit 21-1224  
gaminv 21-1233  
gamlike 21-1235  
gamma distribution B-43  
gampdf 21-1237  
gamrnd 21-1239  
gamstat 21-1240  
Gaussian distribution B-46  
Gaussian mixture distributions B-47  
generalized extreme value distribution B-48  
generalized Pareto distribution B-53  
geocdf 21-1278  
geoinv 21-1280  
geomean 21-1281  
geometric distribution B-57  
geopdf 21-1282  
geornd 21-1283  
geostat 21-1284  
gevcdf 21-1302  
gevfit 21-1303  
gevinv 21-1305  
gevlike 21-1306  
gevpdf 21-1307  
gevrnd 21-1308  
gevstat 21-1310  
gline 21-1311  
glmfit 21-1313  
glmval 21-1320  
glyphplot 21-1325  
gname 21-1337

gpcdf 21-1339  
 gpfit 21-1340  
 gpinv 21-1342  
 gplike 21-1343  
 gplotmatrix 21-1345  
 gppdf 21-1344  
 gprnd 21-1349  
 gpstat 21-1350  
 group mean clusters, plot 8-46  
 grouped plot matrices 8-40  
 grp2idx 21-1353  
 gscatter 21-1376

## H

harmmean 21-1384  
 hidden Markov models  
   overview 17-5  
 hierarchical clustering  
   cluster analysis 14-3  
   computing inconsistency coefficient 21-1438  
   constructing clusters 21-349  
   cophenetic correlation coefficients 21-493  
   creating cluster trees 21-1734  
   creating clusters 14-16  
   creating clusters from data 21-358  
   determining proximity 21-2442 21-2448  
   evaluating cluster formation 21-493  
   grouping objects 14-6  
   inconsistency coefficient 21-1438  
   procedure 14-3  
 hist3 21-1385  
 HMM 17-5  
 HMM functions 17-7  
 hmmdecode 21-1406  
 hmmestimate 17-9 21-1408  
 hmmgenerate 21-1411  
 hmmtrain 17-10 21-1413  
 hmmviterbi 21-1416  
 holdout

  partition 21-605  
 hougen 21-1420  
 hygecdf 21-1421  
 hygeinv 21-1423  
 hygepdf 21-1424  
 hygernd 21-1425  
 hygestat 21-1426  
 hypergeometric distribution B-60  
 hypotheses B-42  
 hypothesis tests  
   assumptions 7-5  
   functions that support 7-14  
   power 7-4 21-3080

## I

icdf 21-1427  
 IFM. *See* Inference Functions for Margins method  
 incomplete beta function B-5  
 incomplete gamma function B-43  
 inconsistency coefficient 21-1438  
 inconsistent 21-1438  
 Inference Functions for Margins (IFM)  
   method 21-498  
 initial state distribution  
   changing 17-12  
 interaction effects  
   designed experiments 18-2  
   two-way ANOVA 8-9  
 interactionplot 21-1446  
 interquartile range (iqr) 3-6  
 inverse Gaussian distribution B-63  
 inverse Wishart distribution B-64  
 invpred 21-1461  
 iqr 21-1464  
 iwishrnd 21-1490

## J

jackknife 21-1491

Jarque-Bera test 7-14  
Johnson system of distributions 6-26 B-66  
johnsrnd 21-1500

## K

K-means clustering  
  cluster separation 14-23  
  local minima 14-28  
  number of clusters 14-25  
  overview 14-22  
  silhouette plot 21-3153  
kernel distribution  
  nonparametric distributions B-67  
kmeans 21-1563  
Kolmogorov-Smirnov test  
  one-sample 7-14  
  two-sample 7-15  
Kruskal-Wallis test 7-14 8-36  
kurtosis 21-1638

## L

landmark data 13-62  
latin hypercube sample 21-1680  
  normal distribution 21-1681  
least squares  
  iteratively reweighted 9-125  
leverage 21-1678  
lhsdesign 21-1680  
lhsnorm 21-1681  
likelihood function 21-98  
Lilliefors test 7-15  
linear hypothesis test 7-15  
linear regression  
  ridge 9-128  
linear transformations  
  Procrustes 21-2720  
linhpytest 21-1732  
link functions 10-13

linkage  
  average 21-1740  
  centroid 21-1740  
  complete 21-1740  
  furthest neighbor 21-1740  
  nearest neighbor 21-1740  
  single 21-1740  
  ward 21-1741  
  weighted average distance 21-1741  
loadings 13-95  
logistic distribution B-76  
logistic models 10-26  
loglogistic distribution B-78  
logncdf 21-1760  
lognfit 21-1763  
logninv 21-1765  
lognlike 21-1767  
lognormal distribution B-80  
lognormal fit 21-1763  
lognpdf 21-1776  
lognrnd 21-1778  
lognstat 21-1780  
loss  
  prediction 21-3102  
lsline 21-1820

## M

mad 21-1822  
mahal 21-1824  
Mahalanobis distance  
  computing 21-1824 21-1828  
  in cluster analysis 16-10 21-2444 21-2451  
main effects 18-2  
maineffectsplo 21-1832  
Mann-Whitney U-test 21-2840  
MANOVA 8-39  
manova1 21-1854  
manovacluster 21-1858  
Markov chains



- emission matrix 17-4
  - emissions 17-4
  - initial state 17-4
  - Monte Carlo simulations 6-13
  - overview 17-3
  - transition matrices 17-4
  - Markov models
    - hidden
      - functions for 17-7
      - generating test sequences for 17-8
      - overview 17-5
    - state diagram 17-3
  - maximum likelihood
    - factor analysis 21-836
  - MCMC 6-13
  - MDS. *See* multidimensional scaling
  - mdscale 21-1882
  - mean absolute deviation 21-1822
  - mean squares (MS) 21-44
  - measures of
    - central tendency 3-3
    - dispersion 3-5
  - median absolute deviation 21-1822
  - metric multidimensional scaling 13-50
    - See also* classical multidimensional scaling
  - mhsample 21-1920
  - Minkowski metric 16-10 21-2445 21-2451
  - missing data 3-21
  - mixed-effects models 11-24
  - mnpdf 21-1955
  - mnrnd 21-1976
  - models
    - mixed-effects 11-24
  - moment 21-1996
  - MS. *See* mean squares
  - multcompare 21-1999
  - multicollinearity 21-3041
    - addressed by ridge regression 9-128
  - multidimensional arrays
    - classical (metric) scaling 21-367
    - multidimensional scaling (MDS)
      - classical (metric) 13-50
    - multinomial distribution B-83
    - multiple comparison procedure 21-1999
    - multivariate analysis of variance
      - example 8-39
    - multivariate Gaussian distribution B-85
    - multivariate normal distribution B-86
    - multivariate statistics
      - analysis of variance 8-39
      - principal component analysis 13-80
    - multivariate *t* distribution B-91
  - multivarichart 21-2016
  - mvncdf 21-2020
  - mvnpdf 21-2024
  - mvnrnd 21-2048
  - mvregresslike 21-2046
  - mvtcdf 21-2050
  - mvtpdf 21-2054
  - mvtrnd 21-2056
- N**
- Nakagami distribution B-97
  - nancov 21-2070
  - nanmax 21-2072
  - nanmean 21-2073
  - nanmedian 21-2074
  - nanmin 21-2075
  - NaNs
    - coding missing values as 3-21
  - nanstd 21-2076
  - nansum 21-2077
  - nanvar 21-2078
  - nbincdf 21-2080
  - nbinfit 21-2082
  - nbininv 21-2083
  - nbinpdf 21-2084
  - nbinrnd 21-2086
  - nbinstat 21-2087

ncfcdf 21-2089  
ncfinv 21-2091  
ncfpdf 21-2093  
ncfrnd 21-2095  
ncfstat 21-2097  
nctcdf 21-2101  
nctinv 21-2104  
nctpdf 21-2105  
nctrnd 21-2107  
nctstat 21-2108  
ncx2cdf 21-2110  
ncx2inv 21-2113  
ncx2pdf 21-2114  
ncx2rnd 21-2116  
ncx2stat 21-2117  
nearest neighbor linkage 21-1740  
negative binomial distribution  
  confidence intervals 21-2082  
  cumulative distribution function  
    (cdf) 21-2080  
  definition B-98  
  inverse cumulative distribution function  
    (cdf) 21-2083  
  mean and variance 21-2087  
  modeling number of auto accidents B-99  
  nbincdf function 21-2080  
  nbininv function 21-2083  
  nbinpdf function 21-2084  
  parameter estimates 21-2082  
  probability density function (pdf) 21-2084  
  random matrices 21-2086  
negative binomial fit 21-2082  
Newton's method 21-1233  
nlintool 21-2161  
nlmefit 21-2163  
nlparci 21-2199  
nnmf 21-2216  
noncentral  $F$  distribution B-106  
nonlinear mixed effects 21-2163  
nonnegative matrix factorization

  dimension-reduction technique 13-77  
normal distribution B-110  
normal fit 21-2269  
normal probability plots 4-9  
normalizing  
  data sets 14-4  
normcdf 21-2267  
normfit 21-2269  
norminv 21-2271  
normlike 21-2273  
normpdf 21-2274  
normplot 21-2276  
normrnd 21-2278  
normspec 21-2279  
normstat 21-2281  
null hypotheses 7-3

## O

one-sample Kolmogorov-Smirnov test 7-14  
online updates 21-1568  
outliers  
  measures resistant to 3-3

## P

$p$  values 7-3  
parallel regression 21-1959 21-1982  
Parallel statistics 20-1  
parallelcoords 21-2340  
Pareto distribution B-113  
partial least-squares regression 9-147  
PCA. *See* principal component analysis  
pcacov 21-2409  
pcares 21-2411  
pdf 21-2426  
pdist 21-2442  
Pearson system of distributions 6-26 B-114  
pearsrnd 21-2454  
perms 21-2468

piecewise distributions B-115  
 Plackett-Burman designs 18-5  
 plsregress 21-2566  
 poisscdf 21-2574  
 poissfit 21-2576  
 poissinv 21-2577  
 Poisson distribution B-118  
 Poisson fit 21-2576  
 poisspdf 21-2584  
 poissrnd 21-2585  
 poisstat 21-2586  
 polyconf 21-2587  
 polytool 21-2593  
 posterior state probabilities  
     estimating 17-11  
 power  
     hypothesis tests 7-4  
 principal component analysis (PCA)  
     overview 13-80  
 princomp 21-2689  
 probabilities  
     posterior state, estimating 17-11  
 Probability Distribution Function Tool 5-37  
 probability distributions  
     disttool 5-37  
 probplot 21-2714  
 procrustes 21-2720  
 Procrustes analysis 13-62 21-2720  
 pseudorandom numbers  
     generating 6-2

## Q

qqplot 21-2754  
*QR* decomposition 21-2910  
 QRNG (quasi-random number generator) 6-15  
 quality assurance 21-116  
 quantile-quantile plots 4-12  
 quasi-random numbers  
     generating 6-15

sequences  
     leaping 6-17  
     point set 6-17  
     scrambling 6-17  
     skipping 6-17  
 streams 6-24  
     state 6-24

## R

randg 21-2767  
 random 21-2770  
 random number generation  
     acceptance-rejection methods 6-9  
     direct methods 6-5  
     inversion methods 6-7  
     methods 6-5  
 Random Number Generation Tool 5-80  
 random number generators (RNGs) 6-2  
 random samples  
     inverse Wishart 21-1490  
     latin hypercube 21-1680  
     latin hypercube with normal  
         distribution 21-1681  
     Wishart 21-3635  
 randsample 21-2817  
 randtool 21-2819  
 range 21-2822  
 raylcdf 21-2849  
 Rayleigh distribution B-121  
 Rayleigh fit 21-2857  
 raylfit 21-2857  
 raylinv 21-2858  
 raylpdf 21-2859  
 raylrnd 21-2860  
 raylstat 21-2861  
 rcoplot 21-2862  
 refcurve 21-2864  
 refline 21-2869  
 regress 21-2873

- regression
    - adjusted  $R$ -square statistic 21-2910
    - change in covariance 21-2910
    - change in fitted values 21-2910
    - coefficient covariance 21-2910
    - coefficients 21-2910
    - delete-1 coefficients 21-2910
    - delete-1 variance 21-2910
    - $F$  distribution B-42
    - $F$  statistic 21-2910
    - fitted values 21-2910
    - hat matrix 21-2910
    - leverage 21-2910
    - mean squared error 21-2910
    - partial least squares 9-147
    - projection matrix 21-2910
    - $R$ -square statistic 21-2910
    - residuals 21-2910
    - scaled change in coefficients 21-2910
    - scaled change in fitted values 21-2910
    - $t$  statistic 21-2910
  - regression trees
    - example 16-52
  - regstats 21-2910
  - relative efficiency 21-1464
  - resampling
    - statistics 3-16
  - residuals
    - standardized 21-2910
    - studentized 21-2910
  - response surfaces
    - designs
      - Box-Behnken 18-13
      - central composite 18-9
      - overview 18-9
  - resubstitution error 21-2223
  - Rician distribution B-123
  - ridge 21-3040
  - ridge parameters 9-128 21-3041
  - ridge regression 9-128 21-3040
  - ridge trace 21-3040
  - robust linear fit 21-2754
  - robust linear regression 21-3053
  - robustdemo 21-3049
  - robustfit 21-3053
  - rotatable designs 18-11
  - rotatefactors 21-3060
  - rowexch 21-3064
  - rsmdemo 21-3069
  - rstool 21-3073
  - runs test 7-15
  - runstest 21-3077
- ## S
- sampsizepwr 21-3080
  - SBS. *See* sequential backward selection
  - scaling arrays
    - classical multidimensional 21-367
  - scatter plots
    - grouped 8-40
  - sequential backward selection (SBS) 13-72
  - sequential feature selection
    - criterion 13-71
  - sequential forward selection (SFS) 13-72
  - sequentialfs 21-3102
  - SFS. *See* sequential forward selection
  - Shepard plots 13-59
  - sign tests 7-15
  - significance levels 7-3
  - silhouette 21-3153
  - similarity matrices
    - creating 14-4
  - single linkage 21-1740
  - skewness 21-3182
  - SPC. *See* statistical process control
  - specific variance 13-95
  - squareform 21-3196
  - SS. *See* sum of squares
  - standard normal 21-2274

standardized Euclidean distance 16-9 21-2444  
 21-2450  
 state sequences  
   estimating 17-8  
 statistical process control  
   capability studies 19-7  
   control charts 19-3  
 stepwise 21-3241  
 stepwisefit 21-3335  
 Student's  $t$  distribution B-124  
   noncentral B-108  
 sum of squares (SS) 21-44  
 surfht 21-3372

## T

$t$  location-scale distribution B-126  
 $t$ -tests  
   one-sample 7-15  
   paired-sample 7-15  
   two-sample 7-16  
 tab-delimited data  
   reading from file 21-3404  
 tabular data  
   reading from file 21-3398  
 tabulate 21-3397  
 tblread 21-3398  
 tblwrite 21-3400  
 tcdf 21-3402  
 tdfread 21-3404  
 test data 15-2  
 test sequences  
   generating, for hidden Markov model 17-8  
 test statistics 7-3  
 tiedrank 21-3434  
 tin v 21-3436  
 tpdf 21-3446  
 training data 15-2  
 transition matrices  
   estimating 17-9

treatments  
   experimental 18-3  
 treedisp 21-3466  
 treefit 21-3469  
 treeprune 21-3472  
 trees 21-3469  
   *See also* decision trees  
 treetest 21-3476  
 treeval 21-3480  
 triangular distribution B-127  
 trimmean 21-3488  
 trnd 21-3492  
 tstat 21-3499  
 two-level designs 18-4  
 two-sample Kolmogorov-Smirnov test 7-15  
 two-way ANOVA 8-9  
 type I errors 7-3  
 type II errors 7-4

## U

unidcdf 21-3538  
 unidinv 21-3539  
 unidpdf 21-3540  
 unidrnd 21-3541  
 unidstat 21-3542  
 unifcdf 21-3543  
 unifinv 21-3545  
 unifit 21-3546  
 uniform distribution B-133  
 uniformly distributed fit 21-3546  
 unifpdf 21-3547  
 unifrnd 21-3548  
 unifstat 21-3550

## W

Wald distribution B-63  
 ward linkage 21-1741  
 wblcdf 21-3612

wblfit 21-3614  
wblinv 21-3616  
wbllike 21-3619  
wblpdf 21-3621  
wblplot 21-3623  
wblrnd 21-3626  
wblstat 21-3627  
Weibull distribution B-141  
Weibull fit 21-3614  
Weibull, Waloddi B-141  
weighted average distance linkage 21-1741  
whiskers  
    on plots 4-8  
Wilcoxon rank sum test 7-15  
Wilcoxon signed rank tests 7-15

Wishart distribution B-144  
Wishart random matrix 21-3635  
    inverse 21-1490  
wishrnd 21-3635  
wrapper methods  
    feature selection 21-3105

## **X**

x2fx 21-3638  
xptread 21-3637

## **Z**

z-test, one-sample 7-16