Risk Management Toolbox™

User's Guide

MATLAB®

R2023**a**

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Risk Management Toolbox™ User's Guide

Revision History

September 2016	Online only	New for Version 1.0 (Release 2016b)
March 2017	Online only	Revised for Version 1.1 (Release 2017a)
September 2017	Online only	Revised for Version 1.2 (Release 2017b)
March 2018	Online only	Revised for Version 1.3 (Release 2018a)
September 2018	Online only	Revised for Version 1.4 (Release 2018b)
March 2019	Online only	Revised for Version 1.5 (Release 2019a)
September 2019	Online only	Revised for Version 1.6 (Release 2019b)
March 2020	Online only	Revised for Version 1.7 (Release 2020a)
September 2020	Online only	Revised for Version 1.8 (Release 2020b)
March 2021	Online only	Revised for Version 1.9 (Release 2021a)
September 2021	Online only	Revised for Version 1.10 (Release 2021b)
March 2022	Online only	Revised for Version 2.0 (Release 2022a)
September 2022	Online only	Revised for Version 2.1 (Release 2022b)
March 2023	Online only	Revised for Version 2.2 (Release 2023a)

Contents

1 Getting Started

2 Market Risk Measurements Using VaR BackTesting Tools

3 Managing Consumer Credit Risk Using the Binning Explorer for Credit Scorecards

4 Corporate Credit Risk Simulations for Portfolios

5 Model Risk Management with Modelscape

6

Functions

Getting Started

Risk Management Toolbox Product Description

Develop risk models and perform risk simulation

Risk Management Toolbox provides functions and interactive workflows for mathematical modeling and simulation of credit, insurance, and market risk. You can perform lifetime credit modeling of probabilities of default (PD), exposure at default (EAD), and loss given default (LGD), as well as expected credit loss (ECL) calculations. You can assess corporate and consumer credit risk, create credit scorecards, estimate probabilities of default, perform credit portfolio analysis, and backtest models to assess potential for financial loss. The toolbox lets you identify important scorecard variables using the predictor screening tools and use the Binning Explorer app to automatically or manually bin variables for credit scorecards. It also includes mortality and unpaid claims models to quantify and analyze insurance risk. Market risk can be assessed with backtesting and simulation tools to evaluate value-at-risk (VaR) and expected shortfall (ES).

Risk Modeling with Risk Management Toolbox

In this section...

Risk Management Toolbox provides tools for modeling seven areas of risk assessment:

- Consumer credit risk
- Corporate credit risk
- Market risk
- Insurance risk
- Lifetime models for probability of default
- Loss given default models
- Exposure at default models

Consumer Credit Risk

Consumer credit risk (also referred to as retail credit risk) is the risk of loss due to a customer's default (non-repayment) on a consumer credit product. These products can include a mortgage, unsecured personal loan, credit card, or overdraft. A common method for predicting credit risk is through a credit scorecard. The scorecard is a statistically based model for attributing a score to a customer that indicates the predicted probability that the customer will default. The data used to calculate the score can be from sources such as application forms, credit reference agencies, or products the customer already holds with the lender. Financial Toolbox™ provides tools for creating credit scorecards and performing credit portfolio analysis using scorecards. Risk Management Toolbox includes a Binning Explorer app for automatic or manual binning to streamline the binning phase of credit scorecard development. For more information, see "Overview of Binning Explorer" on page 3-2.

Corporate Credit Risk

Corporate credit risk (also referred to as wholesale credit risk) is the risk that counterparties default on their financial obligations.

At an individual counterparty level, one of the main credit risk parameters is the probability of default (PD). Risk Management Toolbox allows you to estimate probabilities of default using the following methodologies:

- Structural models: `mertonmodel` and `mertonByTimeSeries`
- Reduced-form models: `cdsbootstrap` and `bondDefaultBootstrap` using Financial Toolbox

- Historical credit ratings migrations: `transprob` using Financial Toolbox
- Statistical approaches: credit scorecards using **Binning Explorer** and the `creditscorecard` object using Financial Toolbox, and a wide selection of predictive models in Statistics and Machine Learning Toolbox™

At a credit portfolio level, on the other hand, to assess credit risk, to assess this risk, the main question to ask is, Given a current credit portfolio, how much can be lost in a given time period due to defaults? In differing circumstances, the answer to this question might mean:

- How much do you expect to lose?
- How likely is it that you will lose more than a specific amount?
- What is the most you can lose under relatively normal circumstances?
- How much can you lose if things get bad?

Mathematically, these questions all depend on estimating a distribution of losses for the credit portfolio: What are the different amounts you can lose, and how likely is it that you lose each individual amount.

Corporate credit risk is fundamentally different from market risk, which is the risk that assets lose value due to market movements. The most important difference is that markets move all the time, but defaults occur infrequently. Therefore, the sample sizes to support any modeling efforts are different. The challenge is to calibrate a distribution of credit losses, because the sample sizes are small. For credit risk, even for an individual bond that has not defaulted, you cannot collect direct data on what happens in the event of default because it has not defaulted. And once the issuer actually defaults, unless you can pool default information from similar companies, that is the only data point that you have.

For corporate credit portfolio analysis, estimating credit correlations so that you can understand the benefits of diversification is also challenging. Two companies can only default in the same time window once, so you cannot collect data on how often they default together. To collect more data, you can pool data from similar companies and under similar economic conditions.

Risk Management Toolbox provides a credit default simulation framework for credit portfolios using the `creditDefaultCopula` object, where the three main elements of credit risk for a single instrument are:

- The probability of default (PD) which is the likelihood that the issuer defaults in a given time period.
- The exposure at default (EAD) which is the amount of money that is at stake. For a traditional bond, this is the bond principal.
- The loss given default (LGD) which is the fraction of the exposure that would be lost at default. When default occurs, usually some money is recovered eventually.

The assumption is that these three quantities are fixed and known for all the companies in the credit portfolio. With this assumption, the only uncertainty is whether each company defaults, which happens with probability PD*i*.

At the credit portfolio level, however, the main question is, "What are the default correlations between issuers?" For example, for two bonds with 10MM principal each, the risk is different if you expect the companies to default together. In this scenario, you could lose 20MM minus the recovery, all at once. Alternatively, if the defaults are independent, you could lose 10MM minus recovery if one defaults, but the other company is likely still alive. Default correlations are therefore important parameters for understanding the risk at a portfolio level. These parameters are also important for understanding the diversification and concentration characteristics of the portfolio. The approach in Risk Management Toolbox is to simulate correlated variables that can be efficiently simulated and parameterized, then map the simulated values to default or nondefault states to preserve the individual default probabilities. This approach is called a copula. When normal variables are used, this approach is called a Gaussian copula. Risk Management Toolbox also provides a credit migration simulation framework for credit portfolios using the `creditMigrationCopula` object. For more information, see “Credit Rating Migration Risk” on page 1-10.

Related to the `creditDefaultCopula` and `creditMigrationCopula` objects, Risk Management Toolbox provides an analytical model known as the Asymptotic Single Risk Factor (ASRF) model. The ASRF model is useful because the Basel II documents propose this model as the standard for certain types of capital requirements. ASRF is not a Monte-Carlo model, so you can quickly compute the capital requirements for large credit portfolios. You can use the ASRF model to perform a quick sensitivity analysis and exploring "what-if" scenarios more easily than rerunning large simulations. For more information, see `asrf`.

Risk Management Toolbox also provides tools for portfolio concentration analysis, see “Concentration Indices” on page 1-15.

Market Risk

Market risk is the risk of losses in positions arising from movements in market prices. Value-at-risk is a statistical method that quantifies the risk level associated with a portfolio. VaR measures the maximum amount of loss over a specified time horizon, at a given confidence level. For example, if the one-day 95% VaR of a portfolio is 10MM, then there is a 95% chance that the portfolio loses less than 10MM the following day. In other words, only 5% of the time (or about once in 20 days) the portfolio losses exceed 10MM.

VaR Backtesting, on the other hand, measures how accurate the VaR calculations are. For many portfolios, especially trading portfolios, VaR is computed daily. At the closing of the following day, the actual profits and losses for the portfolio are known, and can be compared to the VaR estimated the day before. You can use this daily data to assess the performance of VaR models, which is the goal of VaR backtesting. As such, backtesting is a method that looks retrospectively at data and refines the VaR models. Many VaR backtesting methodologies have been proposed. As a best practice, use more than one criterion to backtest the performance of VaR models, because all tests have strengths and weaknesses.

Risk Management Toolbox provides the following VaR backtesting individual tests:

- Traffic light test (`tl`)
- Binomial test (`bin`)
- Kupiec’s tests (`pof`, `tuff`)
- Christoffersen’s tests (`cc`, `cci`)
- Haas’s tests (`tbf`, `tbfi`)

For information on the different tests, see “Overview of VaR Backtesting” on page 2-2.

Expected Shortfall (ES) Backtesting gives an estimate of the loss in those very bad days when the VaR is violated. ES is the expected loss on days when there is a VaR failure. If the VaR is 10 million and the ES is 12 million, you know that the expected loss tomorrow, if it happens to be a very bad day, is about 20% higher than the VaR.

Risk Management Toolbox provides the following table-based tests for expected shortfall based on the `esbacktest` object:

- `unconditionalNormal`
- `unconditionalT`

The following tools support expected shortfall simulation-based tests for the `esbacktestbysim` object:

- `conditional`
- `unconditional`
- `quantile`

For information on the different tests, see “Overview of Expected Shortfall Backtesting” on page 2-20.

Insurance Risk

The ability to accurately estimate unpaid claims is important to insurers. Unlike companies in other sectors, insurers might not know the exact earnings during a financial reporting period until many years later. Insurance companies take in insurance premiums on a regular basis and pay out claims when events occur. In order to maximize profits, an insurance company must accurately estimate how much will be paid out on existing claims in the future. If the estimate for unpaid claims is too low, the insurance company will become insolvent. Conversely, if the estimate is too high, then the claims reserve capital of the insurance company could have been invested elsewhere or reinvested in the business

Risk Management Toolbox supports four claims estimation methods for actuaries to use with a `developmentTriangle` for estimating unpaid claims:

- `chainLadder`
- `expectedClaims`
- `bornhuetterFerguson`
- `capeCod`

For information on the estimation methods, see “Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16.

Lifetime Models for Probability of Default

Regulatory frameworks such as IFRS 9 and CECL require institutions to estimate loss reserves based on a lifetime analysis that is conditional on macroeconomic scenarios. Earlier models were frequently designed to predict one period ahead and often with no explicit sensitivities to macroeconomic scenarios. With the IFRS 9 and CECL regulations, models must predict multiple periods ahead and the models must have an explicit dependency on macroeconomic variables.

The main output of the lifetime credit analysis is the lifetime expected credit loss (ECL). The lifetime ECL consists of the reserves that banks need to set aside for expected losses throughout the life of a loan. There are different approaches to the estimation of lifetime ECL. Some approaches use relatively simple techniques on loss data, with qualitative adjustments. Other approaches use more advanced time-series techniques or econometric models to forecast losses, with dependencies on macro variables. Another methodology uses probability of default (PD) models, loss given default (LGD) models, and exposure at default (EAD) models, and combines their outputs to estimate the ECL. The lifetime PD models in Risk Management Toolbox are in the PD-LGD-EAD category

Risk Management Toolbox provides the following lifetime PD models:

- `Logistic`
- `Probit`
- `Cox`
- `customLifetimePDModel`

For information on the different models, see “Overview of Lifetime Probability of Default Models” on page 1-25.

Loss Given Default Models

Loss given default (LGD) is the proportion of a credit that is lost in the event of default. LGD is one of the main parameters for credit risk analysis. Although there are different approaches to estimate credit loss reserves and credit capital, common methodologies require the estimation of probabilities of default (PD), loss given default (LGD), and exposure at default (EAD). The reserves and capital requirements are computed using formulas or simulations that use these parameters. For example, the loss reserves are usually estimated as the expected loss (EL), given by the following formula:

$EL = PD * LGD * EAD$

Risk Management Toolbox provides the following LGD models:

- `Regression`
- `Tobit`
- `Beta`

For information on the different models, see “Overview of Loss Given Default Models” on page 1-31.

Exposure at Default Models

EAD is seen as an estimation of the extent to which a bank may be exposed to a counterparty in the event of, and at the time of, that counterparty’s default. EAD is equal to the current amount outstanding in case of fixed exposures such as term loans. For example, the loss reserves are usually estimated as the expected loss (EL), given by the following formula:

$EL = PD * LGD * EAD$

Risk Management Toolbox provides the following EAD models:

- `Regression`
- `Tobit`
- `Beta`

For information on the different models, see “Overview of Exposure at Default Models” on page 1-34.

See Also

`varbacktest` | `esbacktest` | `esbacktestbysim` | `mertonmodel` | `mertonByTimeSeries` | `concentrationIndices` | `creditDefaultCopula` | `creditMigrationCopula` | `asrf` | `developmentTriangle` | `chainLadder` | `expectedClaims` | `bornhuetterFerguson` | `Logistic` | `Probit` | `Cox` | `Regression` | `Tobit` | `Beta` | `Regression` | `Tobit` | `Beta`

Related Examples

- “Common Binning Explorer Tasks” on page 3-4
- “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23
- “creditMigrationCopula Simulation Workflow” on page 4-10
- “creditDefaultCopula Simulation Workflow” on page 4-5
- “Modeling Correlated Defaults with Copulas” on page 4-18
- “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36
- “VaR Backtesting Workflow” on page 2-6
- “Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30
- “Expected Shortfall (ES) Backtesting Workflow Using Simulation” on page 2-34
- “Expected Shortfall Estimation and Backtesting” on page 2-44
- “Value-at-Risk Estimation and Backtesting” on page 2-10
- “Basic Lifetime PD Model Validation” on page 4-129
- “Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
- “Compare Lifetime PD Models Using Cross-Validation” on page 4-121
- “Expected Credit Loss Computation” on page 4-124
- “Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations” on page 4-195

More About

- “Credit Simulation Using Copulas” on page 4-2
- “Credit Rating Migration Risk” on page 1-10
- “Default Probability by Using the Merton Model for Structural Credit Risk” on page 1-13
- “Concentration Indices” on page 1-15
- “Traffic Light Test” on page 2-3
- “Binomial Test” on page 2-2
- “Kupiec’s POF and TUFF Tests” on page 2-3
- “Christoffersen’s Interval Forecast Tests” on page 2-4
- “Haas’s Time Between Failures or Mixed Kupiec’s Test” on page 2-4
- “Overview of Expected Shortfall Backtesting” on page 2-20
- “Overview of Lifetime Probability of Default Models” on page 1-25
- “Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

External Websites

- Introduction to Risk Management Toolbox (26 min 24 sec)
- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)
- Credit Risk Modeling with MATLAB (53 min 09 sec)
- Forecasting Corporate Default Rates with MATLAB (54 min 36 sec)

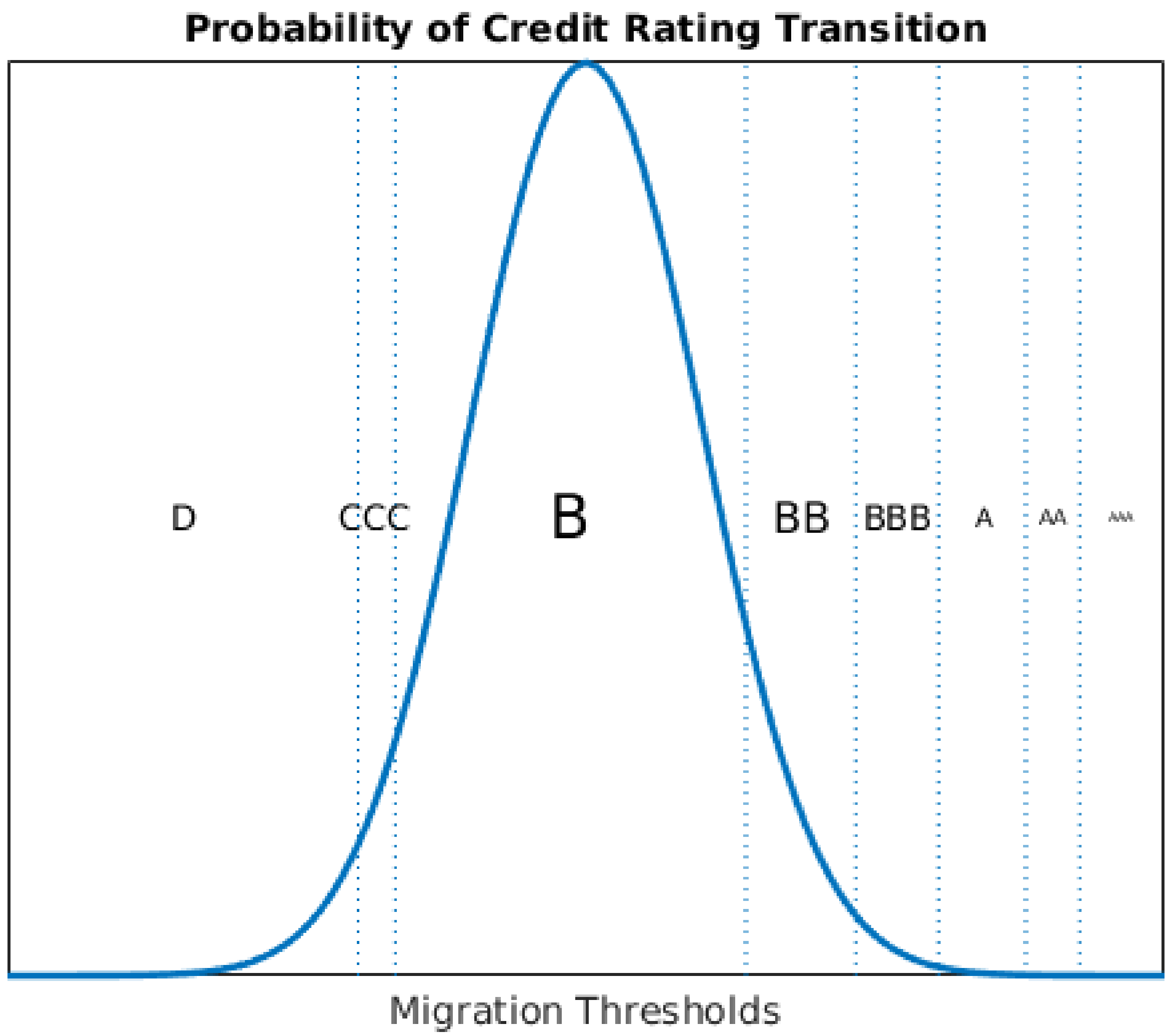
Credit Rating Migration Risk

The migration-based multi-factor copula (`creditMigrationCopula`) is similar to the `creditDefaultCopula` object. As described in “Credit Simulation Using Copulas” on page 4-2, each counterparty’s credit quality is represented by a “latent variable” which is simulated over many scenarios. The latent variable is composed of a series of correlated factors which are weighted based on the counterparty’s sensitivity to each factor. The two objects differ in how the latent variables are used for the remainder of the analysis. Instead of thinking in terms of probability of default for each obligor, the `creditMigrationCopula` object works with each obligor’s credit rating. Credit ratings are issued by several companies (S&P, Moody's, and so on). Each rating represents a level of credit quality and ratings are changed periodically as a company’s situation improves or deteriorates.

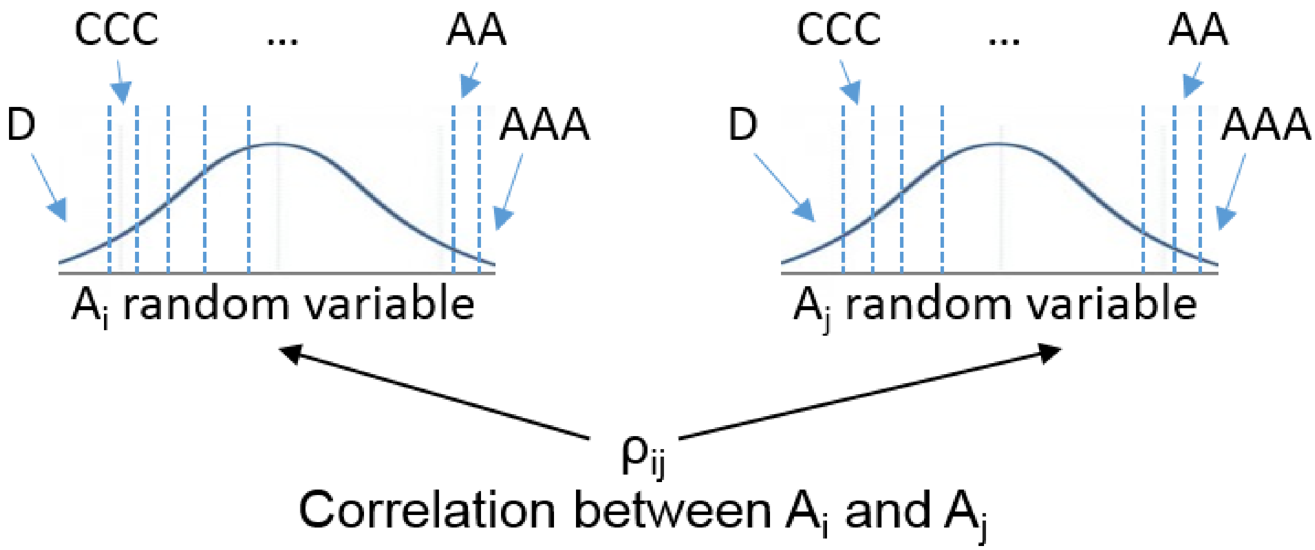
Given enough historical data, the likelihood is calculated that a company at a particular rating will migrate to a different rating over some time period. For example, this table shows the probabilities that a company with credit rating `"B"` will transition to each other rating.

New Rating	Probability (%)
AAA	0.001
AA	0.0062
A	0.1081
BBB	0.8697
BB	7.3366
B	86.7215
CCC	2.5169
Default	2.4399

While the `creditDefaultCopula` object is concerned with the 2.4% chance of default exclusively, a migration-based approach using an `creditMigrationCopula` object accounts for all possible rating states. Given these probabilities, the cut-points are calculated for the distribution of all possible latent variable values that correspond to each rating value.



For each scenario, the latent variable value determines the credit rating of the counterparty at the end of the time period based on these cut-points. The cut-points are set such that the probability of transitioning to each rating matches the probabilities in the provided transition table. You now have not just correlated defaults for each counterparty, but correlated rating changes across the entire range of credit ratings.



Each credit rating has a unique discount curve associated with it. As an obligor's credit rating falls, the obligor's bond cashflows become more deeply discounted and the total bond value drops accordingly. Conversely, if an obligor's rating improves, the cashflows are discounted less deeply, and the bond values will rise. After repricing the portfolio with all obligors' new ratings, the total portfolio value can be calculated as the sum of the new bond values. As with the `creditDefaultCopula` object, various risk measures are calculated and reported for the `creditMigrationCopula` object.

See Also

`creditMigrationCopula` | `simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios`

Related Examples

- "creditMigrationCopula Simulation Workflow" on page 4-10

Default Probability by Using the Merton Model for Structural Credit Risk

In 1974, Robert Merton proposed a model for assessing the structural credit risk of a company by modeling the company's equity as a call option on its assets. The Merton model uses the Black-Scholes-Merton option pricing methods and is structural because it provides a relationship between the default risk and the asset (capital) structure of the firm.

A company balance sheet records book values—the value of a firm's equity E, its total assets A, and its total liabilities L. The relationship between these values is defined by the equation

$$A = E + L$$

These book values for E, A, and L are all observable because they are recorded on a firm's balance sheet. However, the book values are reported infrequently. Alternatively, only the equity’s market value is observable, and is given by the firm’s stock market price times the number of outstanding shares. The market value of the firm’s assets and total liabilities are unobservable.

The Merton model relates the market values of equity, assets, and liabilities in an option pricing framework. The Merton model assumes a single liability L with maturity T, usually a period of one year or less. At time T, the firm’s value to the shareholders equals the difference $A – L$ when the asset value A is greater than the liabilities L. However, if the liabilities L exceed the asset value A, then the shareholders get nothing. The value of the equity E_T at time T is related to the value of the assets and liabilities by the following formula:

$$E_T = \max(A_T - L, 0)$$

In practice, firms have multiple maturities for their liabilities, so for a selected maturity T, a liability threshold L is chosen based on the whole liability structure of the firm. The liability threshold is also referred to as the default point. For a typical time horizon of one year, the liability threshold is commonly set to a value between the value of the short-term liabilities and the value of the total liabilities.

Assuming a lognormal distribution for the asset returns, you can use the Black-Scholes-Merton equations to relate the observable market value of equity E, and the unobservable market value of assets A, at any time prior to the maturity T:

$$E = AN(d_1) - Le^{-rT}N(d_2)$$

In this equation, r is the risk-free interest rate, N is the cumulative standard normal distribution, and d_1 and d_2 are given by

$$d_1 = \frac{\ln\left(\frac{A}{L}\right) + (r + 0.5\sigma_A^2)T}{\sigma_A\sqrt{T}}$$

$$d_2 = d_1 - \sigma_A\sqrt{T}$$

You can solve this equation using one of two approaches:

- The `mertonmodel` approach uses single-point calibration and requires values for the equity, liability, and equity volatility (σ_E).

This approach solves for (A,σ_A) using a 2-by-2 system of nonlinear equations. The first equation is the aforementioned option pricing formula. The second equation relates the unobservable volatility of assets σ_A to the given equity volatility σ_E:

$$\sigma_E = \frac{A}{E}N(d_1)\sigma_A$$

- The `mertonByTimeSeries` approach requires time series for the equity and for all other model parameters.

  If the equity time series has n data points, this approach calibrates a time series of n asset values $A_1,\ldots,A_n$ that solve the following system of equations:

  $$E_1 = A_1N(d_1) - L_1e^{-r_1T_1}N(d_2)$$

  $$\ldots$$

  $$E_n = A_nN(d_1) - L_ne^{-r_nT_n}N(d_2)$$

  The function directly computes the volatility of assets σ_A from the time series $A_1,\ldots,A_n$ as the annualized standard deviation of the log returns. This value is a single volatility value that captures the volatility of the assets during the time period spanned by the time series.

  After computing the values of A and σ_A, the function computes the distance to default (DD) is computed as the number of standard deviations between the expected asset value at maturity T and the liability threshold:

  $$DD = \frac{\log A + \left(\mu_A - \sigma_A^2/2\right)T - \log(L)}{\sigma_A\sqrt{T}}$$

  The drift parameter μ_A is the expected return for the assets, which can be equal to the risk-free interest rate, or any other value based on expectations for that firm.

  The probability of default (`PD`) is defined as the probability of the asset value falling below the liability threshold at the end of the time horizon T:

  $$PD = 1 - N(DD)$$

See Also

`mertonmodel` | `mertonByTimeSeries`

Related Examples

- "Comparison of the Merton Model Single-Point Approach to the Time-Series Approach" on page 4-54

Concentration Indices

In financial risk applications, concentration is the opposite of diversification. If all or most of your risk is in one area, it is concentrated. Higher concentration is interpreted as a risk, although for someone with a high tolerance for risk and who wants higher returns, that person might prefer concentration.

You can use concentration indices to measure and monitor concentration in a credit portfolio. Ad-hoc concentration indices are typically computed by using exposures, and therefore do not usually take into account other risk parameters such as probabilities of default. Ad-hoc concentration indices are frequently included in comprehensive concentration reports, with other concentration measures and concentration limits.

When you use the `concentrationIndices` function, Risk Management Toolbox supports the following ad-hoc concentration indices or measures:

- Concentration ratio
- Deciles of the portfolio weight distribution
- Gini coefficient
- Herfindahl-Hirschman index
- Hannah-Kay index
- Hall-Tideman index
- Theil entropy index

See Also

`concentrationIndices`

Related Examples

- “Analyze the Sensitivity of Concentration to a Given Exposure” on page 4-49
- “Compare Concentration Indices for Random Portfolios” on page 4-51

Overview of Claims Estimation Methods for Non-Life Insurance

The ability to accurately estimate unpaid claims is important to insurers. Unlike companies in other sectors, insurers might not know the exact earnings during a financial reporting period until many years later. Insurance companies take in insurance premiums on a regular basis and pay out claims when events occur. In order to maximize profits, an insurance company must accurately estimate how much will be paid out on existing claims in the future. If the estimate for unpaid claims is too low, the insurance company will become insolvent. Conversely, if the estimate is too high, then the claims reserve capital of the insurance company could have been invested elsewhere or reinvested in the business [1 on page 1-24].

Risk Management Toolbox supports four claims estimation methods for actuaries to use for estimating unpaid claims:

- `chainLadder`
- `expectedClaims`
- `bornhuetterFerguson`
- `capeCod`

Workflow to Estimate Unpaid Claims

For the different claims estimation methods, the basic workflow follows.

1. Create a development triangle with insurance claims data using `developmentTriangle`. The claims data can be for either reported claims or paid claims. You can plot reported claims using `claimsPlot`.
2. Use the development triangle to compute link ratios using `linkRatios`. You can plot link ratios using `linkRatiosPlot`.
3. Use the development triangle link ratio for reported claims or paid claims to compute the link ratio averages with `linkRatioAverages`.
4. Use `ultimateClaims` to compute the projected ultimate claims based on the link ratio averages for either reported claims or unpaid claims.
5. Using the projected ultimate claims for both the reported and paid development triangles, use any of the following to compute incurred-but-not-reported (IBNR) values and the total unpaid claims estimates:

   - Chain ladder method — Create a `chainLadder` object with development triangles for reported and paid claims, generate the IBNR values using `ibnr`, and compute the unpaid claims estimation with `unpaidClaims`.
   - Expected claims method — Create an `expectedClaims` object with development triangles for reported and paid claims as well as the earned premium. By default, the initial claims are calculated as the average of the reported ultimate claims and the paid ultimate claims. However, you can specify custom values for the initial claims. Similar to the chain ladder method, you can compute IBNR values using `ibnr` and the unpaid claims estimates with `unpaidClaims`.
   - Bornhuetter-Ferguson method — Create a `bornhuetterFerguson` object with development triangles for reported and paid claims as well as initial expected claims values, generate IBNR using `ibnr`, and compute the unpaid claims estimation with `unpaidClaims`.

- Cape Cod method — Create a `capeCod` object with development triangles for reported and paid claims as well as initial expected claims values, generate IBNR using `ibnr`, and compute the unpaid claims estimation with `unpaidClaims`.

Estimation of Ultimate Claims Using Development Triangles

One characteristic of Development Triangles is that the ultimate claims are estimated from recorded values assuming that the development of future claims resembles that in prior years — the past is indicative of the future.

The steps for development triangles are demonstrated using simulated data:

1. Use `developmentTriangle` to generate the reported claims in what is called a development triangle, where there is one row for each origin year and the columns depict how the claims develop over time.

Origin	Reported Claims as of (months)									
Year	**12**	**24**	**36**	**48**	**60**	**72**	**84**	**96**	**108**	**120**
2010	3995.71	4635.02	4866.78	4964.1	5013.74	5038.82	5058.97	5074.14	5084.29	5089.38
2011	3968.04	4682.28	4963.22	5062.49	5113.11	5138.67	5154.09	5169.56	5179.89	
2012	4217.01	5060.42	5364.04	5508.87	5558.45	5586.24	5608.59	5625.41		
2013	4374.24	5205.34	5517.67	5661.12	5740.38	5780.56	5803.68			
2014	4499.68	5309.62	5628.2	5785.79	5849.43	5878.68				
2015	4530.24	5300.38	5565.4	5715.66	5772.82					
2016	4572.63	5304.25	5569.47	5714.27						
2017	4680.56	5523.06	5854.44							
2018	4696.68	5495.11								
2019	4945.89									

2. Use `linkRatios` to calculate the age-to-age factors. These factors are also known as report-to-report factors or link ratios. The link ratios measure the change in recorded claims from one valuation date to the next. The standard naming convention is *starting month-ending month*. For example, the age-to-age factor for the 12-month period to the 24-month period is often referred to as the 12-24 factor.

   To calculate the age-to-age factors for the 12-24 period, divide the claims as of 24 months by the claims as of 12 months. Thus, the triangle of age-to-age factors has one less row and one less column than the original data triangle.

Origin	Age-to-Age Factors								
Year	**12-24**	**24-36**	**36-48**	**48-60**	**60-72**	**72-84**	**84-96**	**96-108**	**108-120**
2010	1.16	1.05	1.02	1.01	1.005	1.004	1.003	1.002	1.001
2011	1.18	1.06	1.02	1.01	1.005	1.003	1.003	1.002	
2012	1.2	1.06	1.027	1.009	1.005	1.004	1.003		
2013	1.19	1.06	1.026	1.014	1.007	1.004			
2014	1.18	1.06	1.028	1.011	1.005				
2015	1.17	1.05	1.027	1.01					
2016	1.16	1.05	1.026						
2017	1.18	1.06							
2018	1.17								
2019									

3. After calculating the age-to-age factors, use `linkRatioAverages` to calculate the averages of the age-to-age factors. Actuaries use a wide variety of averages for age-to-age factors. Some of the common ones are the simple average, medial average, geometric average, and volume-weighted average.

	Averages								
	12-24	24-36	36-48	48-60	60-72	72-84	84-96	96-108	108-120
Simple Average	1.1767	1.0563	1.0249	1.0107	1.0054	1.0038	1.0030	1.0020	1.0010
Simple Average - Latest 5	1.1720	1.0560	1.0268	1.0108	1.0054	1.0038	1.0030	1.0020	1.0010
Simple Average - Latest 3	1.1700	1.0533	1.0270	1.0117	1.0057	1.0037	1.0030	1.0020	1.0010
Medial Average - Latest 5x1	1.1733	1.0567	1.0267	1.0103	1.0050	1.0040	1.0030	1.0020	1.0010
Volume-weighted Average	1.1766	1.0563	1.0250	1.0107	1.0054	1.0038	1.0030	1.0020	1.0010
Volume-weighted Average - Latest 5	1.1720	1.0560	1.0268	1.0108	1.0054	1.0038	1.0030	1.0020	1.0010
Volume-weighted Average - Latest 3	1.1701	1.0534	1.0270	1.0117	1.0057	1.0037	1.0030	1.0020	1.0010
Geometric Average - Latest 4	1.1700	1.0550	1.0267	1.0110	1.0055	1.0037	1.0030	1.0020	1.0010

4 Use `cdfSummary` to obtain the cumulative claim development factors (CDF), which are calculated by successive multiplications beginning with the tail factor and the oldest age-to-age factor. The cumulative claim development factor projects the total growth over the remaining valuations.

	Development Factor Selection									
	12-24	24-36	36-48	48-60	60-72	72-84	84-96	96-108	108-120	Ultimate
Selected	1.1767	1.0563	1.0249	1.0107	1.0054	1.0038	1.0030	1.0020	1.0010	1.0000
CDF to Ultimate	1.3069	1.1107	1.0516	1.0261	1.0152	1.0098	1.0060	1.0030	1.0010	1.0000
Percent Reported	0.7651	0.9003	0.9510	0.9746	0.9850	0.9903	0.9940	0.9970	0.9990	1.0000

5 All of the previous steps apply to the reported claims. In order to calculate the unpaid claims estimates, you need the paid claims as well as the reported claims. Use `developmentTriangle` to generate the development triangle for paid claims.

Origin	Paid Claims as of (months)									
Year	12	24	36	48	60	72	84	96	108	120
2010	1893.92	3371.18	4079.13	4487.04	4711.39	4805.62	4853.68	4877.94	4887.71	4892.59
2011	2055.52	3638.28	4365.93	4758.87	4949.22	5048.21	5098.69	5124.18	5134.43	
2012	2242.45	3946.71	4696.58	5119.27	5324.05	5430.53	5484.83	5512.26		
2013	2373.81	4130.43	4915.22	5357.59	5571.9	5677.76	5728.85			
2014	2421.75	4189.62	4985.63	5434.34	5651.72	5759.1				
2015	2484.05	4272.56	5084.35	5541.95	5763.62					
2016	2481.74	4218.95	5020.54	5472.39						
2017	2577.88	4382.38	5171.21							
2018	2580.04	4386.07								
2019	2764.81									

Similar to the reported claims development triangle, you use the paid claims develop triangle to calculate link ratios, average link ratios, and then you can select one link ratio and calculate the cumulative development factors.

6 Use `ultimateClaims` to project the ultimate claims. The ultimate claims are equal to the product of the latest valuation of claims and the appropriate cumulative claim development factors. The projected ultimate claims are displayed for both the reported claims and the paid claims.

Origin Year	Age of Origin Year at 12/31/2019	Claims at 12/31/2019		CDF to Ultimate		Projected Ultimate Claims using Dev. Method with	
		Reported	Paid	Reported	Paid	Reported	Paid
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
2010	120	5089.38	4892.59	1.0000	1.0000	5089.38	4892.59
2011	108	5179.89	5134.43	1.0010	1.0010	5185.08	5139.56
2012	96	5625.41	5512.26	1.0030	1.0030	5642.30	5528.81
2013	84	5803.68	5728.85	1.0060	1.0080	5838.57	5774.78
2014	72	5878.68	5759.1	1.0098	1.0178	5936.20	5861.87
2015	60	5772.82	5763.62	1.0152	1.0378	5860.78	5981.45
2016	48	5714.27	5472.39	1.0261	1.0810	5863.22	5915.85
2017	36	5854.44	5171.21	1.0516	1.1799	6156.35	6101.37
2018	24	5495.11	4386.07	1.1107	1.4070	6103.54	6171.19
2019	12	4945.89	2764.81	1.3069	2.4388	6464.02	6742.81
Total		55359.57	50585.33			58139.43	58110.27

Column Notes
(3) and (4) Latest Diagonals of the Reported and Paid Claims
(5) and (6) CDF to Ultimate from the Development Factor Selection tables of Reported and Paid Claims
(7) = [(3) x (5)]
(8) = [(4) x (6)]

7 After calculating the projected ultimate claims, use a `chainLadder`, `expectedClaims`, or `bornhuetterFerguson` method for estimating the unpaid claims.

Estimation of Unpaid Claims Using Chain Ladder Method

The chain ladder method requires the Development Triangles for reported and paid claims. The chain ladder method assumes that you can predict future claims activity for a given origin year (accident year, policy year, report year, and so on) based on historical claims activity to date for that origin year. The primary assumption of this method is that the reporting and payment of future claims resembles the patterns observed in the past.

In addition, the chain ladder method requires a large volume of historical claims experience. It works best when the presence or absence of large claims does not greatly distort the data. If the volume of data is not sufficient, large claims can greatly distort the age-to-age factors, the projections of ultimate claims, and the estimate of unpaid claims.

1 After calculating the projected ultimate claims using Development Triangles, create a `chainLadder` object based on the reported and paid Development Triangles in order to compute the unpaid claim estimates with `unpaidClaims`.

2 Actuaries calculate the unpaid claims estimate as the difference between the projected ultimate claims and the actual paid claims. This value of the unpaid claim estimate represents total unpaid claims, including both the outstanding claims cases and the IBNR claims. To determine estimated IBNR values based on the chain ladder technique, subtract reported claims from the projected ultimate claims. Alternatively, you can use `ibnr` to calculate the IBNR, which is equal to the estimate of total unpaid claims less the outstanding cases.

						Unpaid Claim Estimate at 12/31/2019			
	Claims at 12/31/2019		Projected Ultimate Claims using Dev. Method with		Case Outstanding at 12/31/2019	IBNR - Based on Dev. Method with		Total - Based on Dev. Method with	
Origin Year	Reported	Paid	Reported	Paid		Reported	Paid	Reported	Paid
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
2010	5089.38	4892.59	5089.38	4892.59	196.79	0.00	-196.79	196.79	0.00
2011	5179.89	5134.43	5185.08	5139.56	45.46	5.19	-40.33	50.65	5.13
2012	5625.41	5512.26	5642.30	5528.81	113.15	16.89	-96.60	130.04	16.55
2013	5803.68	5728.85	5838.57	5774.78	74.83	34.89	-28.90	109.72	45.93
2014	5878.68	5759.1	5936.20	5861.87	119.58	57.52	-16.81	177.10	102.77
2015	5772.82	5763.62	5860.78	5981.45	9.2	87.96	208.63	97.16	217.83
2016	5714.27	5472.39	5863.22	5915.85	241.88	148.95	201.58	390.83	443.46
2017	5854.44	5171.21	6156.35	6101.37	683.23	301.91	246.93	985.14	930.16
2018	5495.11	4386.07	6103.54	6171.19	1109.04	608.43	676.08	1717.47	1785.12
2019	4945.89	2764.81	6464.02	6742.81	2181.08	1518.13	1796.92	3699.21	3978.00
Total	55359.57	50585.33	58139.42767	58110.26703	4774.24	2779.8577	2750.697029	7554.0977	7524.937

Column Notes

(2) and (3) Latest Diagonals of the Reported and Paid Claims

(4) and (5) Developed in the previous sheet

(6) = [(2) - (3)]

(7) = [(4) - (2)]

(8) = [(5) - (2)]

(9) = [(6) + (7)]

(10) = [(6) + (8)]

Estimation of Unpaid Claims Using Expected Claims Method

The key assumption of the expected claims method is that an actuary can better estimate unpaid claims based on an initial estimate rather than existing claims observed to date.

The expected claims method requires the Development Triangles for reported and paid claims as well as the earned premium. By default, the initial claims are calculated as the average of the reported ultimate claims and the paid ultimate claims. However, you can specify custom values for the initial claims. Using the initial claims, an actuary applies a claim ratio method, where ultimate claims for a development period are equal to a selected expected claim ratio multiplied by the earned premium. Using these calculated ultimate claims, the actuary can then compute the unpaid claims estimates.

1 Create an `expectedClaims` object to calculate the `ultimateClaims`.

Origin Year	Claims at 12/31/2019		CDF to Ultimate		Projected Ultimate Claims using Dev. Method with		Initial Selected Ultimate Claims	Earned	Claim Ratio		Ultimate
	Reported	Paid	Reported	Paid	Reported	Paid	Claims	Premium	Estimated	Selected	Claims
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
2010	5089.38	4892.59	1.0000	1.0000	5089.38	4892.59	4990.99	10000	49.91%	50.00%	5000
2011	5179.89	5134.43	1.0010	1.0010	5185.08	5139.56	5162.32	12000	43.02%	50.00%	6000
2012	5625.41	5512.26	1.0030	1.0030	5642.30	5528.81	5585.55	14000	39.90%	50.00%	7000
2013	5803.68	5728.85	1.0060	1.0080	5838.57	5774.78	5806.67	16000	36.29%	50.00%	8000
2014	5878.68	5759.1	1.0098	1.0178	5936.20	5861.87	5899.03	18000	32.77%	50.00%	9000
2015	5772.82	5763.62	1.0152	1.0378	5860.78	5981.45	5921.11	20000	29.61%	40.00%	8000
2016	5714.27	5472.39	1.0261	1.0810	5863.22	5915.85	5889.53	22000	26.77%	40.00%	8800
2017	5854.44	5171.21	1.0516	1.1799	6156.35	6101.37	6128.86	24000	25.54%	40.00%	9600
2018	5495.11	4386.07	1.1107	1.4070	6103.54	6171.19	6137.36	26000	23.61%	40.00%	10400
2019	4945.89	2764.81	1.3069	2.4388	6464.02	6742.81	6603.41	28000	23.58%	40.00%	11200

Column Notes
(2) and (3) Latest Diagonals of the Reported and Paid Claims
(4) and (5) CDF to Ultimate from the Development Factor Selection tables of Reported and Paid Claims
(6) = [(2) x (4)]
(7) = [(3) x (5)]
(8) = [((6) + (7)) / 2]
(9) Earned Premium from Data
(10) = [(8) / (9)]
(11) Selected judgementally based on experience in (10)
(12) = [(9) x (11)]

2 Use the `expectedClaims` object to calculate the `unpaidClaims`.

Origin Year	Claims at 12/31/2019		Ultimate	Case Outstanding at	Unpaid Claim Estimate based on Expected Claims Method	
	Reported	Paid	Claims	12/31/2019	IBNR	Total
(1)	(2)	(3)	(4)	(5)	(6)	(7)
2010	5089.38	4892.59	5000	196.79	-89.38	107.41
2011	5179.89	5134.43	6000	45.46	820.11	865.57
2012	5625.41	5512.26	7000	113.15	1374.59	1487.74
2013	5803.68	5728.85	8000	74.83	2196.32	2271.15
2014	5878.68	5759.1	9000	119.58	3121.32	3240.90
2015	5772.82	5763.62	8000	9.2	2227.18	2236.38
2016	5714.27	5472.39	8800	241.88	3085.73	3327.61
2017	5854.44	5171.21	9600	683.23	3745.56	4428.79
2018	5495.11	4386.07	10400	1109.04	4904.89	6013.93
2019	4945.89	2764.81	11200	2181.08	6254.11	8435.19
Total	55359.57	50585.33	83000	4774.24	27640.43	32414.67

Column Notes
(2) and (3) Latest Diagonals of the Reported and Paid Claims
(4) Developed in the previous table
(5) = [(2) - (3)]
(6) = [(4) - (2)]
(7) = [(4) - (3)]

Estimation of Unpaid Claims Using Bornhuetter-Ferguson Method

The Bornhuetter-Ferguson method combines the chain ladder method and the expected claims method by splitting ultimate claims into two components, actual reported (or paid) claims and expected unreported (or unpaid) claims. As the claim matures over development periods, more weight is given to the actual claims and the expected claims become gradually less important.

The Bornhuetter-Ferguson method requires the Development Triangles for reported and paid claims as well as initial expected claims values. The Bornhuetter-Ferguson method calculates its own projected ultimate claims, different from those calculated in the Development Triangle object. Using these new projected ultimate claims, the unpaid claims estimates are computed.

1 Create a `bornhuetterFerguson` object to calculate the `ultimateClaims`.

	Expected	CDF to Ultimate		Percentage		Expected Claims		Claims at 12/31/2019		Projected Ultimate Claims using B-F Method with	
Origin Year	Claims	Reported	Paid	Unreported	Unpaid	Unreported	Unpaid	Reported	Paid	Reported	Paid
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
2010	5000	1.0000	1.0000	0.00%	0.00%	0.00	0.00	5089.38	4892.59	5089.38	4892.59
2011	6000	1.0010	1.0010	0.10%	0.10%	6.00	5.98	5179.89	5134.43	5185.89	5140.41
2012	7000	1.0030	1.0030	0.30%	0.30%	20.95	20.95	5625.41	5512.26	5646.36	5533.21
2013	8000	1.0060	1.0080	0.60%	0.80%	47.80	63.62	5803.68	5728.85	5851.48	5792.47
2014	9000	1.0098	1.0178	0.97%	1.75%	87.20	157.78	5878.68	5759.1	5965.88	5916.88
2015	8000	1.0152	1.0378	1.50%	3.64%	120.06	291.34	5772.82	5763.62	5892.88	6054.96
2016	8800	1.0261	1.0810	2.54%	7.50%	223.55	659.66	5714.27	5472.39	5937.82	6132.05
2017	9600	1.0516	1.1799	4.90%	15.25%	470.79	1463.53	5854.44	5171.21	6325.23	6634.74
2018	10400	1.1107	1.4070	9.97%	28.93%	1036.72	3008.38	5495.11	4386.07	6531.83	7394.45
2019	11200	1.3069	2.4388	23.49%	59.00%	2630.42	6607.57	4945.89	2764.81	7576.31	9372.38
Total	83000					4643.50	12278.82	55359.57	50585.33	60003.07	62864.15

Column Notes
(2) Developed in the Expected Claims method
(3) and (4) CDF to Ultimate from the Development Factor Selection tables of Reported and Paid Claims
(5) = [1.00 - (1.00 / (3))]
(6) = [1.00 - (1.00 / (4))]
(7) = [(2) x (5)]
(8) = [(2) x (6)]
(9) and (10) Latest Diagonals of the Reported and Paid Claims
(11) = [(7) + (9)]
(12) = [(8) + (10)]

2 Use the `bornhuetterFerguson` object to calculate the `unpaidClaims`.

	Claims at 12/31/2019		Projected Ultimate Claims using B-F Method with		Case Outstanding at	Unpaid Claim Estimate at 12/31/2019: IBNR - Based on B-F Method with		Unpaid Claim Estimate at 12/31/2019: Total - Based on B-F Method with	
Origin Year	Reported	Paid	Reported	Paid	12/31/2019	Reported	Paid	Reported	Paid
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
2010	5089.38	4892.59	5089.38	4892.59	196.79	0.00	-196.79	196.79	0.00
2011	5179.89	5134.43	5185.89	5140.41	45.46	6.00	-39.48	51.46	5.98
2012	5625.41	5512.26	5646.36	5533.21	113.15	20.95	-92.20	134.10	20.95
2013	5803.68	5728.85	5851.48	5792.47	74.83	47.80	-11.21	122.63	63.62
2014	5878.68	5759.1	5965.88	5916.88	119.58	87.20	38.20	206.78	157.78
2015	5772.82	5763.62	5892.88	6054.96	9.2	120.06	282.14	129.26	291.34
2016	5714.27	5472.39	5937.82	6132.05	241.88	223.55	417.78	465.43	659.66
2017	5854.44	5171.21	6325.23	6634.74	683.23	470.79	780.30	1154.02	1463.53
2018	5495.11	4386.07	6531.83	7394.45	1109.04	1036.72	1899.34	2145.76	3008.38
2019	4945.89	2764.81	7576.31	9372.38	2181.08	2630.42	4426.49	4811.50	6607.57
Total	55359.57	50585.33	60003.07	62864.15	4774.24	4643.50	7504.58	9417.74	12278.82

Column Notes
(2) and (3) Latest Diagonals of the Reported and Paid Claims
(4) and (5) Developed in the previous table
(6) = [(2) - (3)]
(7) = [(4) - (2)]
(8) = [(5) - (2)]
(9) = [(6) + (7)]
(10) = [(6) + (8)]

Estimation of Unpaid Claims Using Cape Cod Method

As in the Bornhuetter-Ferguson technique, the Cape Cod technique splits ultimate claims into two components: actual reported (or paid) and expected unreported (or unpaid). As an accident year (or other time interval) matures, the actual reported claims replace the expected unreported claims and the initial expected claims assumption becomes gradually less important. The primary difference between the two methods is the derivation of the expected claim ratio. In the Cape Cod technique, the expected claim ratio is obtained from the reported claims experience instead of an independent and often judgmental selection as in the Bornhuetter-Ferguson technique.

The Cape Cod technique requires the Development Triangles for reported and paid claims as well as the earned premium. The key assumption of the Cape Cod technique is that unreported claims will develop based on expected claims, which are derived using reported (or paid) claims and earned premium. Both the Cape Cod and Bornhuetter-Ferguson methods differ from the development method where the primary assumption is that unreported claims will develop based on reported claims to date (not expected claims).

1 Create a `capeCod` object to calculate the `ultimateClaims`.

Origin Year	Earned Premium	Expected Claim Ratio	Estimated Expected Claims	Reported CDF to Ultimate	Percentage Unreported	Expected Unreported Claims	Reported Claims at 12/31/2019	Projected Ultimate Claims
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
2010	10000	31.0%	3101.86	1.0000	0.0%	0.00	5089.38	5089.38
2011	12000	31.0%	3722.23	1.0010	0.1%	3.72	5179.89	5183.61
2012	14000	31.0%	4342.60	1.0030	0.3%	13.00	5625.41	5638.41
2013	16000	31.0%	4962.97	1.0060	0.6%	29.65	5803.68	5833.33
2014	18000	31.0%	5583.35	1.0098	1.0%	54.10	5878.68	5932.78
2015	20000	31.0%	6203.72	1.0152	1.5%	93.11	5772.82	5865.93
2016	22000	31.0%	6824.09	1.0261	2.5%	173.36	5714.27	5887.63
2017	24000	31.0%	7444.46	1.0516	4.9%	365.08	5854.44	6219.52
2018	26000	31.0%	8064.83	1.1107	10.0%	803.94	5495.11	6299.05
2019	28000	31.0%	8685.21	1.3069	23.5%	2039.80	4945.89	6985.69
Total	190000		58935.33			3575.76	55359.57	58935.33

Column Notes
(2) Earned Premium from Data
(3) Based on total weighted estimated claim ratios
(4) = [(2) x (3)]
(5) CDF to Ultimate for Reported Claims
(6) = [1.00 - (1.00 / (5))]
(7) = [(4) x (6)]
(8) Latest Diagonal for Reported Claims
(9) = [(7) + (8)]

2 Use the `capeCod` object to calculate the `unpaidClaims`.

	Claims at 12/31/2019		Ultimate	Case	Unpaid Claim Estimate	
Origin Year	Reported	Paid	Claims	Outstanding	IBNR	Total
(1)	(2)	(3)	(4)	(5)	(6)	(7)
2010	5089.38	4892.59	5089.38	196.79	0.00	196.79
2011	5179.89	5134.43	5183.61	45.46	3.72	49.18
2012	5625.41	5512.26	5638.41	113.15	13.00	126.15
2013	5803.68	5728.85	5833.33	74.83	29.65	104.48
2014	5878.68	5759.1	5932.78	119.58	54.10	173.68
2015	5772.82	5763.62	5865.93	9.2	93.11	102.31
2016	5714.27	5472.39	5887.63	241.88	173.36	415.24
2017	5854.44	5171.21	6219.52	683.23	365.08	1048.31
2018	5495.11	4386.07	6299.05	1109.04	803.94	1912.98
2019	4945.89	2764.81	6985.69	2181.08	2039.80	4220.88
Total	55359.57	50585.33	58935.33	4774.24	3575.76	8350.00

Column Notes
(2) and (3) Latest Diagonals of the Reported and Paid Claims
(4) Developed in the previous table
(5) = [(2) - (3)]
(6) = [(4) - (2)]
(7) = [(5) + (6)]

References

[1] Friedland, Jacqueline. "Estimating Unpaid Claims using Basic Techniques." Arlington, VA: Casualty Actuarial Society, 2010.

[2] Wüthrich, Mario, and Michael Merz. *Stochastic Claims Reserving Methods in Insurance.* Hoboken, NJ: Wiley, 2008.

See Also

`developmentTriangle` | `chainLadder` | `expectedClaims` | `bornhuetterFerguson` | `capeCod`

Related Examples

- "Mean Square Error of Prediction for Estimated Ultimate Claims" on page 4-161
- "Bootstrap Using Chain Ladder Method" on page 4-168

Overview of Lifetime Probability of Default Models

Regulatory frameworks such as IFRS 9 and CECL require institutions to estimate loss reserves based on a lifetime analysis that is conditional on macroeconomic scenarios. Earlier models were frequently designed to predict one period ahead and often with no explicit sensitivities to macroeconomic scenarios. With the IFRS 9 and CECL regulations, models must predict multiple periods ahead and the models must have an explicit dependency on macroeconomic variables.

The main output of the lifetime credit analysis is the lifetime expected credit loss (ECL). The lifetime ECL consists of the reserves that banks need to set aside for expected losses throughout the life of a loan. There are different approaches to the estimation of lifetime ECL. Some approaches use relatively simple techniques on loss data, with qualitative adjustments. Other approaches use more advanced time-series techniques or econometric models to forecast losses, with dependencies on macro variables. Another methodology uses probability of default (PD) models, loss given default (LGD) models, and exposure at default (EAD) models, and combines their outputs to estimate the ECL. The lifetime PD models in Risk Management Toolbox are in the PD-LGD-EAD category.

Traditional PD Models Compared to Lifetime PD Models

Traditional PD models predict the probability of default for the next period (that is, next year, next quarter, and so on). These one-period ahead models include a range of methodologies, such as credit scorecards (`creditscorecard`), decision trees (`fitctree`), and transition matrices (`transprob`). These models include different types of predictors. Some of them are simple, such as customer income, and others are more complex, such as utilization rate, or some other metrics related to the financial activities of the borrower. For these models, the latest observed values of the predictors, possibly with some lagged information, are usually enough to make a prediction, and there is no need to project or forecast the values of the predictors going forward.

In contrast, the lifetime PD models require forward looking values of all predictors to make a prediction of the lifetime PD through the end of the life of the loan. Because the projected values of the predictors are needed, these models can reduce the amount and complexity of predictors and use either predictors with constant values, such as origination score, or predictors that can be projected with little effort, such as loan-to-value ratio. One predictor typically included in these models is the age of the loan. When used for regulatory purposes, macroeconomic predictors must be included in the model, and multiple macroeconomic scenarios are required for the lifetime credit analysis.

Lifetime credit analysis also requires the cumulative lifetime PD, which is a transformation of the predicted, conditional PDs. Specifically, the marginal PD, which is the increments in the cumulative lifetime PD, is used for the computation of the ECL. The survival probability is often reported as well. These alternative versions of the probability are recursive operations on the predicted, conditional PD values for a single loan. In other words, the prediction data may include rows for the same ID a few periods ahead, and the corresponding conditional PDs may show a time-dependent structure. But these conditional PD predictions are "one-period ahead" predictions where the "period" is the same time interval implicit in the training data. Conditional PD predictions are "row-by-row" predictions, where one row of the inputs predicts a conditional PD independently of all other rows. However, for the cumulative lifetime PD, the cumulative PD value for the second period depends on the conditional PDs for the first and second periods, and all subsequent periods have an explicit dependency on the previous period (a recursion). For the lifetime predictions, therefore, the software must know which rows in the inputs correspond to the same loan, so some form of loan identifier is required for the lifetime prediction. Moreover, consecutive rows in the lifetime prediction data must correspond to consecutive time periods, the recursion is defined for consecutive, one-period ahead conditional PDs, it cannot skip periods.

The following table summarizes the differences between traditional PD models and lifetime PD models.

Traditional PD Models	Lifetime PD Models
Predict one period ahead	Predict multiple periods ahead
Predict conditional PD only	Predict conditional PD, cumulative lifetime PD, marginal PD, and survival probability
Predict for each row of the data inputs, independently of all other rows	Predict for all rows of the data inputs that correspond to the same loan; this is a recursive operation that requires some form of loan identifier to know where to start the recursion
Need only most recent observed information to make PD predictions	Need the most recent information and projected, period-by-period values of predictor variables over the lifetime of the loan to make PD predictions
Can use complex predictors that result from nontrivial data processing or data transformations	Typically use simpler predictors, variables that are not hard to project and forecast
Besides loan-specific predictors, models can include macroeconomic variables or an age variable	Besides loan-specific predictors, models must include macroeconomic predictors (especially if used for regulatory purposes) and typically include an age variable

Model Development and Validation

Risk Management Toolbox supports the modeling and validation of lifetime PD models through a family of classes supporting:

- Model fitting with the `fitLifetimePDModel`
- Prediction of conditional PD with the `predict` function
- Prediction of lifetime PD (cumulative, marginal, and survival) with the `predictLifetime` function
- Model discrimination metrics with the `modelDiscrimination` function
- Plot the ROC curve with the `modelDiscriminationPlot` function
- Model calibration metrics with the `modelCalibration` function
- Plot observed default rates compared to predicted PDs on grouped data with the `modelCalibrationPlot` function

The supported model types are `Logistic`, `Probit`, `Cox`, and `customLifetimePDModel` models.

A typical modeling workflow for lifetime PD analysis includes:

1. Data preparation

   The lifetime PD models require a panel data input for fitting, prediction, and validation. The response variable must be a binary (`0` or `1`) variable, with `1` indicating default. There is a wide range of tools available to treat missing data (using `fillmissing`), handle outliers (using `filloutliers`), and perform other data preparation tasks.

2 Model fitting

   Use the `fitLifetimePDModel` function to fit a lifetime PD model. You must use the previously prepared data, select a model type, and indicate which variables correspond to loan-specific variables (such as origination score and loan-to-value ratio). Also, you can also include an age variable (such as years on books) and the macroeconomic variables (such as gross domestic product growth or unemployment rate), as well as the ID variable and response variable. You can specify a model description and also specify a model ID or tag for reporting purposes during model validation. Alternatively, you can use `customLifetimePDModel` to use a function handle to define a custom PD model.

3 Model validation

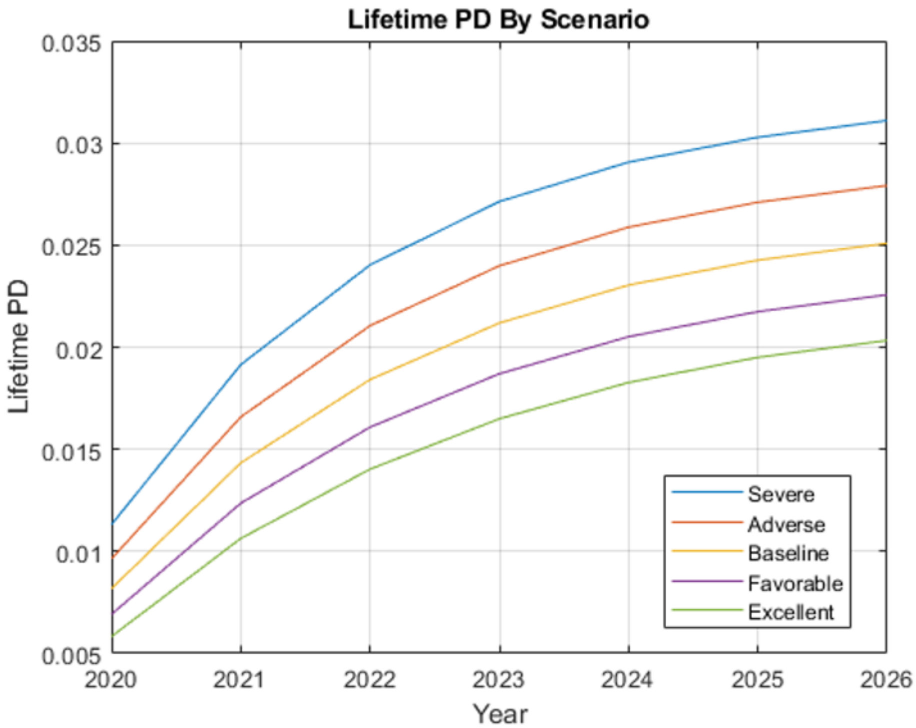
   There are multiple tasks involved in model validation, including

   - Inspect the underlying statistical model, which is stored in the `'UnderlyingModel'` property of the `Logistic`, `Probit`, or `Cox` object. For more information, see "Basic Lifetime PD Model Validation" on page 4-129.
   - Measure the model discrimination on either training or test data with the `modelDiscrimination` function. Visualizations can also be generated using the `modelDiscriminationPlot` function. Data can be segmented to measure discrimination over different segments.
   - Measure the model calibration on either training or test data with the `modelCalibration` function. Visualizations can also be generated using the `modelCalibrationPlot` function. A grouping variable is required to measure the observed default rate for each group and compare it against the average predicted conditional PD for the group.
   - Validate the model against a benchmark (for example, a champion model). For more information, see "Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113.
   - Perform a cross-validation analysis to compare alternative models. For more information, see "Compare Lifetime PD Models Using Cross-Validation" on page 4-121.
   - Perform a qualitative assessment of conditional PD predictions by using the `predict` function directly with edge cases. Note that model validation relies on the conditional PD predictions generated by the `predict` function. The `predict` function is automatically called by `modelDiscrimination` and `modelCalibration` to generate metrics.
   - Visualize the lifetime PD predictions for model validation by using the `predictLifetime` function with edge cases and then perform a qualitative assessment of the predictions.

Computation of Lifetime ECL

Once you develop and validate a lifetime PD model, you can use it for lifetime ECL analysis. The "Expected Credit Loss Computation" on page 4-124 example demonstrates the basic workflow for computing ECL.

The "Expected Credit Loss Computation" on page 4-124 example shows how to visualize the lifetime PD predictions, for different macro scenarios.



The “Expected Credit Loss Computation” on page 4-124 example also shows how to compute the ECL per scenario and how to compute the final lifetime ECL for a given loan.

```
                Severe    Adverse    Baseline    Favorable    Excellent
                ______    _______    ________    _________    _________

ECL 2020        595.58    507.16      430.44      364.11       306.97
ECL 2021        394.24    349.95      310.02      274.11        241.9
ECL 2022        235.53     215.4      196.75       179.5       163.57
ECL 2023        143.05    135.23      127.75      120.59       113.77
ECL 2024        85.219    83.517      81.816      80.118       78.429
ECL 2025        51.346    51.514      51.665      51.798       51.917
ECL 2026        33.162    33.271      33.368      33.454       33.531
ECL total       1538.1      1376      1231.8      1103.7       990.08
Probability        0.1       0.2         0.3         0.2          0.2

Lifetime ECL for company 1304 is: 1217.32
```

For more information on preparing the data for prediction (including joining loan data projections and macro forecasts) and the additional parameters and computations necessary for the estimation of the lifetime ECL, see “Expected Credit Loss Computation” on page 4-124 and `portfolioECL`.

Lifetime Credit Analysis Compared to Stress Testing

You can also use the lifetime PD models for stress testing analysis. However, lifetime credit analysis and stress testing have several differences that the following table summarizes.

Stress Testing	Lifetime Credit Analysis
Focus on negative, pessimistic scenarios	Must consider a range of scenarios, including pessimistic, neutral, and optimistic ones
Models are often biased, calibrated to produce more conservative results	Models are expected to be unbiased
Spans a few quarters ahead	Can span many years ahead
Macroeconomic forecasts for stress testing go a few quarters into the future	Macro scenarios reach far into the future and are typically expected to revert to some baseline level after a few quarters

The types of models used for both of these analyses are very similar. You can use lifetime PD models for stress testing analysis with some additional considerations to account for the differences listed in the previous table.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

See Also

`fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel` | `predict` | `predictLifetime` | `modelDiscrimination` | `modelCalibration` | `modelDiscriminationPlot` | `modelCalibrationPlot` | `portfolioECL`

Related Examples

- "Basic Lifetime PD Model Validation" on page 4-129
- "Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
- "Compare Lifetime PD Models Using Cross-Validation" on page 4-121
- "Expected Credit Loss Computation" on page 4-124
- "Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations" on page 4-195
- "Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
- "Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
- "Create Custom Lifetime PD Model for Decision Tree Model with Function Handle" on page 4-224
- "Create Custom Lifetime PD Model for Credit Scorecard Model with Function Handle" on page 3-131

- "Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations" on page 4-195

More About

- "Overview of Loss Given Default Models" on page 1-31
- "Overview of Exposure at Default Models" on page 1-34

Overview of Loss Given Default Models

Loss given default (LGD) is the proportion of a credit that is lost in the event of default. LGD is one of the main parameters for credit risk analysis. Although there are different approaches to estimate credit loss reserves and credit capital, common methodologies require the estimation of probabilities of default (PD), loss given default (LGD), and exposure at default (EAD). The reserves and capital requirements are computed using formulas or simulations that use these parameters. For example, the loss reserves are usually estimated as the expected loss (EL), given by the following formula:

$EL = PD * LGD * EAD$

With increased availability of data, there are several different types of LGD models. Risk Management Toolbox supports:

- Regression models — These are linear regression models where the response is a transformation of the LGD data. For more information on the supported transformations, see `Regression`.
- Tobit models — These are censored regression models with explicit limits on the response values to capture the fact that LGD can take values only between 0 and 1. Censoring on the left, right or both sides are supported. For more information, see `Tobit`.
- Beta models — These are beta regression models with explicit limits on the response values to capture the fact that LGD can take values only between 0 and 1. Censoring on the left, right or both sides are supported. For more information, see `Beta`.

The "Model Loss Given Default" on page 4-90 example shows these two types of models, as well as other models, are fitted using Statistics and Machine Learning Toolbox. Specifically, besides the regression and Tobit models, this example also includes a non-parametric, look-up table type of model; a Beta regression model; and a "two-stage" model where a classification model (cure-no cure) and a regression model (predicted LGD conditional on no cure) work together to make LGD predictions.

In addition, you can use the `Regression`, `Tobit`, and `Beta` models to develop LGD models that include macroeconomic predictors for stress testing or to support regulatory requirements such as IFRS 9 and CECL. For more information, see "Overview of Lifetime Probability of Default Models" on page 1-25.

Model Development and Validation

Risk Management Toolbox supports the modeling and validation of LGD models through a family of classes supporting:

- Model fitting with the `fitLGDModel`
- Prediction of LGD with the `predict` function
- Model discrimination metrics with the `modelDiscrimination` function and visualization with the `modelDiscriminationPlot` function
- Model calibration metrics with the `modelCalibration` function and visualization with the `modelCalibartionPlot` function

The supported model types are `Regression`, `Tobit`, and `Beta` models.

A typical modeling workflow for LGD analysis includes:

1 Data preparation

   Data preparation for LGD modeling requires a significant amount of work in practice. Data preparation requires consolidation of account information, pulling data from multiple data sources, accounting for recoveries, direct and indirect costs, determination of discount rates to determine the observed LGD values. There is also work regarding predictor transformations and screening. There is a wide range of tools available to treat missing data (using `fillmissing`), handle outliers (using `filloutliers`), and perform other data preparation tasks. The output of the data preparation is a training dataset with predictor columns and a response column containing the LGD values.

2 Model fitting

   Use the `fitLGDModel` function to fit an LGD model. You must use the previously prepared data and select a model type. Optional inputs allow you to indicate which variables correspond to predictor variables, or which transformation to use for a regression model, or the censoring side for a `Tobit` or `Beta` model. You can specify a model description and also specify a model ID or tag for reporting purposes during model validation.

3 Model validation

   There are multiple tasks involved in model validation, including

   - Inspect the underlying statistical model, which is stored in the `'UnderlyingModel'` property of the `Regression`, `Tobit`, or `Beta` object. For more information, see “Basic Loss Given Default Model Validation” on page 4-131.
   - Measure the model discrimination on either training or test data with the `modelDiscrimination` function. Visualizations are generated using the `modelDiscriminationPlot` function. Data can be segmented to measure discrimination over different segments.
   - Measure the model calibration on either training or test data with the `modelCalibration` function. Visualizations are generated using the `modelCalibartionPlot` function. Also, you can visualize the residuals.
   - Validate the model against a benchmark (for example, a champion model). For more information, see “Compare Tobit LGD Model to Benchmark Model” on page 4-133.
   - Perform a cross-validation analysis to compare alternative models. For more information, see “Compare Loss Given Default Models Using Cross-Validation” on page 4-140.
   - Perform a qualitative assessment of conditional PD predictions by using the `predict` function directly with edge cases. Visualize residuals using the `modelCalibartionPlot` function. There are examples of additional visualizations using histograms and box plots in the “Model Loss Given Default” on page 4-90 example.

4 Once you develop and validate a LGD model, you can use it for lifetime ECL analysis. The “Expected Credit Loss Computation” on page 4-124 example and `portfolioECL` demonstrates the basic workflow for computing ECL.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Gupton, G., and R Stein. "*Losscalc v2: Dynamic Prediction of LGD Modeling Methodology*". Moody's KMV Investor Services, 2005.

See Also

`fitLGDModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibartionPlot` | `Regression` | `Tobit` | `Beta` | `portfolioECL`

Related Examples

- "Model Loss Given Default" on page 4-90
- "Basic Loss Given Default Model Validation" on page 4-131
- "Compare Tobit LGD Model to Benchmark Model" on page 4-133
- "Compare Loss Given Default Models Using Cross-Validation" on page 4-140
- "Expected Credit Loss Computation" on page 4-124
- "Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations" on page 4-195

More About

- "Overview of Lifetime Probability of Default Models" on page 1-25
- "Overview of Exposure at Default Models" on page 1-34

Overview of Exposure at Default Models

Exposure at default (EAD) is the loss exposure (balance at the time of default) for a bank when a debtor defaults on a loan.

For example, the loss reserves are usually estimated as the expected loss (EL), given by the following formula:

$EL = PD \times LGD \times EAD$

With increased availability of data, there are several different types of EAD models. Risk Management Toolbox supports:

- Regression models — These are linear regression models where the response is a transformation of the EAD data. For more information on the supported transformations, see `Regression`.
- Tobit models — These are censored regression models with explicit limits on the response values. Censoring on the left, right or both sides are supported. For more information, see `Tobit`.
- Beta models — These are beta regression models with explicit limits on the response values. Censoring on the left, right or both sides are supported. For more information, see `Beta`.

Model Development and Validation

Risk Management Toolbox supports the modeling and validation of EAD models through a family of classes supporting:

- Model fitting with the `fitEADModel`
- Prediction of EAD with the `predict` function
- Model discrimination metrics with the `modelDiscrimination` function and visualization with the `modelDiscriminationPlot` function
- Model calibration metrics with the `modelCalibration` function and visualization with the `modelCalibrationPlot` function

The supported model types are `Regression`, `Tobit`, and `Beta` models.

A typical modeling workflow for EAD analysis includes:

1. Data preparation

   Data preparation for EAD modeling requires a significant amount of work in practice. Data preparation requires consolidation of account information, pulling data from multiple data sources, accounting for recoveries, direct and indirect costs, determination of discount rates to determine the observed EAD values. There is also work regarding predictor transformations and screening. There is a wide range of tools available to treat missing data (using `fillmissing`), handle outliers (using `filloutliers`), and perform other data preparation tasks. The output of the data preparation is a training dataset with predictor columns and a response column containing the EAD values.
2. Model fitting

   Use the `fitEADModel` function to fit an EAD model. You must use the previously prepared data and select a model type. Optional inputs allow you to indicate the limit (`LimitVar`) and drawn (`DrawnVar`) values for a `Regression`, `Tobit`, or `Beta` model. The limit value depends on the loan. If its a credit card, the limit is the credit limit, and if this is a mortgage limit it is the initial loan amount. In general, `LimitVar` is the maximum amount that can be borrowed. `DrawnVar` is the balance on the account at the time of observation, prior to default and EAD is the balance at the time of default. Also, you can specify a model description and also specify a model ID or tag for reporting purposes during model validation.

3 Model validation

   There are multiple tasks involved in model validation, including

   - Inspect the underlying statistical model, which is stored in the `'UnderlyingModel'` property of the `Regression`, `Tobit`, or `Beta` object.
   - Measure the model discrimination on either training or test data with the `modelDiscrimination` function. Visualizations are generated using the `modelDiscriminationPlot` function. Data can be segmented to measure discrimination over different segments.
   - Measure the model calibration on either training or test data with the `modelCalibration` function. Visualizations are generated using the `modelCalibrationPlot` function. Also, you can visualize the residuals.

4 Once you develop and validate an EAD model, you can use it for lifetime ECL analysis. The “Expected Credit Loss Computation” on page 4-124 example and `portfolioECL` demonstrates the basic workflow for computing ECL.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

`fitEADModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot` | `Regression` | `Tobit` | `Beta` | `portfolioECL`

Related Examples

- “Compare Results for Regression and Tobit EAD Models” on page 4-151
- “Expected Credit Loss Computation” on page 4-124
- “Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations” on page 4-195

More About

- “Exposure at Default Regression Models” on page 6-645

- "Exposure at Default Tobit Models" on page 6-656
- "Beta Regression Models" on page 6-669
- "Conversion Measure Options" on page 6-658
- "Overview of Lifetime Probability of Default Models" on page 1-25
- "Overview of Loss Given Default Models" on page 1-31

Market Risk Measurements Using VaR BackTesting Tools

Overview of VaR Backtesting

Market risk is the risk of losses in positions arising from movements in market prices. Value-at-risk (VaR) is one of the main measures of financial risk. VaR is an estimate of how much value a portfolio can lose in a given time period with a given confidence level. For example, if the one-day 95% VaR of a portfolio is 10MM, then there is a 95% chance that the portfolio loses less than 10MM the following day. In other words, only 5% of the time (or about once in 20 days) the portfolio losses exceed 10MM.

For many portfolios, especially trading portfolios, VaR is computed daily. At the closing of the following day, the actual profits and losses for the portfolio are known and can be compared to the VaR estimated the day before. You can use this daily data to assess the performance of VaR models, which is the goal of VaR backtesting. The performance of VaR models can be measured in different ways. In practice, many different metrics and statistical tests are used to identify VaR models that are performing poorly or performing better. As a best practice, use more than one criterion to backtest the performance of VaR models, because all tests have strengths and weaknesses.

Suppose that you have VaR limits and corresponding returns or profits and losses for days $t = 1,\ldots,N$. Use VaRt to denote the VaR estimate for day t (determined on day $t - 1$). Use Rt to denote the actual return or profit and loss observed on day t. Profits and losses are expressed in monetary units and represent value changes in a portfolio. The corresponding VaR limits are also given in monetary units. Returns represent the change in portfolio value as a proportion (or percentage) of its value on the previous day. The corresponding VaR limits are also given as a proportion (or percentage). The VaR limits must be produced from existing VaR models. Then, to perform a VaR backtesting analysis, provide these limits and their corresponding returns as data inputs to the VaR backtesting tools in Risk Management Toolbox.

The toolbox supports these VaR backtests:

- Binomial test
- Traffic light test
- Kupiec’s tests
- Christoffersen’s tests
- Haas’s tests

Binomial Test

The most straightforward test is to compare the observed number of exceptions, x, to the expected number of exceptions. From the properties of a binomial distribution, you can build a confidence interval for the expected number of exceptions. Using exact probabilities from the binomial distribution or a normal approximation, the `bin` function uses a normal approximation. By computing the probability of observing x exceptions, you can compute the probability of wrongly rejecting a good model when x exceptions occur. This is the p-value for the observed number of exceptions x. For a given test confidence level, a straightforward accept-or-reject result in this case is to fail the VaR model whenever x is outside the test confidence interval for the expected number of exceptions. “Outside the confidence interval” can mean too many exceptions, or too few exceptions. Too few exceptions might be a sign that the VaR model is too conservative.

The test statistic is

$$Z_{bin} = \frac{x - Np}{\sqrt{Np(1-p)}}$$

where x is the number of failures, N is the number of observations, and $p = 1$ – VaR level. The binomial test is approximately distributed as a standard normal distribution.

For more information, see “References” on page 2-5 for Jorion and `bin`.

Traffic Light Test

A variation on the binomial test proposed by the Basel Committee is the traffic light test or three zones test. For a given number of exceptions x, you can compute the probability of observing up to x exceptions. That is, any number of exceptions from 0 to x, or the cumulative probability up to x. The probability is computed using a binomial distribution. The three zones are defined as follows:

- The “red” zone starts at the number of exceptions where this probability equals or exceeds 99.99%. It is unlikely that too many exceptions come from a correct VaR model.
- The “yellow” zone covers the number of exceptions where the probability equals or exceeds 95% but is smaller than 99.99%. Even though there is a high number of violations, the violation count is not exceedingly high.
- Everything below the yellow zone is "green." If you have too few failures, they fall in the green zone. Only too many failures lead to model rejections.

For more information, see “References” on page 2-5 for Basel Committee on Banking Supervision and `tl`.

Kupiec’s POF and TUFF Tests

Kupiec (1995) introduced a variation on the binomial test called the proportion of failures (POF) test. The POF test works with the binomial distribution approach. In addition, it uses a likelihood ratio to test whether the probability of exceptions is synchronized with the probability p implied by the VaR confidence level. If the data suggests that the probability of exceptions is different than p, the VaR model is rejected. The POF test statistic is

$$LR_{POF} = -2\log\left(\frac{(1-p)^{N-x}p^x}{\left(1-\frac{x}{N}\right)^{N-x}\left(\frac{x}{N}\right)^x}\right)$$

where x is the number of failures, N the number of observations and $p = 1$ – VaR level.

This statistic is asymptotically distributed as a chi-square variable with 1 degree of freedom. The VaR model fails the test if this likelihood ratio exceeds a critical value. The critical value depends on the test confidence level.

Kupiec also proposed a second test called the time until first failure (TUFF). The TUFF test looks at when the first rejection occurred. If it happens too soon, the test fails the VaR model. Checking only the first exception leaves much information out, specifically, whatever happened after the first exception is ignored. The TBFI test extends the TUFF approach to include all the failures. See `tbfi`.

The TUFF test is also based on a likelihood ratio, but the underlying distribution is a geometric distribution. If n is the number of days until the first rejection, the test statistic is given by

$$LR_{TUFF} = -2\log\left(\frac{p(1-p)^{n-1}}{\left(\frac{1}{n}\right)\left(1-\frac{1}{n}\right)^{n-1}}\right)$$

This statistic is asymptotically distributed as a chi-square variable with 1 degree of freedom. For more information, see "References" on page 2-5 for Kupiec, `pof`, and `tuff`.

Christoffersen's Interval Forecast Tests

Christoffersen (1998) proposed a test to measure whether the probability of observing an exception on a particular day depends on whether an exception occurred. Unlike the unconditional probability of observing an exception, Christoffersen's test measures the dependency between consecutive days only. The test statistic for independence in Christoffersen's interval forecast (IF) approach is given by

$$LR_{CCI} = -2\log\left(\frac{(1-\pi)^{n00+n10}\pi^{n01+n11}}{(1-\pi_0)^{n00}\pi_0^{n01}(1-\pi_1)^{n10}\pi_1^{n11}}\right)$$

where

- $n00$ = Number of periods with no failures followed by a period with no failures.
- $n10$ = Number of periods with failures followed by a period with no failures.
- $n01$ = Number of periods with no failures followed by a period with failures.
- $n11$ = Number of periods with failures followed by a period with failures.

and

- π_0 — Probability of having a failure on period t, given that no failure occurred on period $t-1$ = $n01$ / ($n00$ + $n01$)
- π_1 — Probability of having a failure on period t, given that a failure occurred on period $t-1$ = $n11$ / ($n10$ + $n11$)
- π — Probability of having a failure on period t = ($n01$ + $n11$ / ($n00$ + $n01$ + $n10$ + $n11$)

This statistic is asymptotically distributed as a chi-square with 1 degree of freedom. You can combine this statistic with the frequency POF test to get a conditional coverage (CC) mixed test:

$LR_{CC} = LR_{POF} + LR_{CCI}$

This test is asymptotically distributed as a chi-square variable with 2 degrees of freedom.

For more information, see "References" on page 2-5 for Christoffersen, `cc`, and `cci`.

Haas's Time Between Failures or Mixed Kupiec's Test

Haas (2001) extended Kupiec's TUFF test to incorporate the time information between all the exceptions in the sample. Haas's test applies the TUFF test to each exception in the sample and aggregates the time between failures (TBF) test statistic.

$$LR_{TBFI} = -2\sum_{i=1}^{x}\log\left(\frac{p(1-p)^{n_i-1}}{\left(\frac{1}{n_i}\right)\left(1-\frac{1}{n_i}\right)^{n_i-1}}\right)$$

In this statistic, $p = 1$ – VaR level and n_i is the number of days between failures i-1 and i (or until the first exception for i = 1). This statistic is asymptotically distributed as a chi-square variable with x degrees of freedom, where x is the number of failures.

Like Christoffersen's test, you can combine this test with the frequency POF test to get a TBF mixed test, sometimes called Haas' mixed Kupiec's test:

$$LR_{TBF} = LR_{POF} + LR_{TBFI}$$

This test is asymptotically distributed as a chi-square variable with x+1 degrees of freedom. For more information, see "References" on page 2-5 for Haas, `tbf`, and `tbfi`.

References

[1] Basel Committee on Banking Supervision, *Supervisory framework for the use of "backtesting" in conjunction with the internal models approach to market risk capital requirements.* January 1996, https://www.bis.org/publ/bcbs22.htm.

[2] Christoffersen, P. "Evaluating Interval Forecasts." *International Economic Review.* Vol. 39, 1998, pp. 841–862.

[3] Cogneau, P. *"Backtesting Value-at-Risk: how good is the model?*" Intelligent Risk, PRMIA, July, 2015.

[4] Haas, M. "*New Methods in Backtesting.*" Financial Engineering, Research Center Caesar, Bonn, 2001.

[5] Jorion, P. *Financial Risk Manager Handbook. 6th Edition*, Wiley Finance, 2011.

[6] Kupiec, P. "Techniques for Verifying the Accuracy of Risk Management Models." *Journal of Derivatives.* Vol. 3, 1995, pp. 73–84.

[7] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management.* Princeton University Press, 2005.

[8] Nieppola, O. "Backtesting Value-at-Risk Models." Helsinki School of Economics, 2009.

See Also

`varbacktest` | `tl` | `bin` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Related Examples

- "Value-at-Risk Estimation and Backtesting" on page 2-10

More About

- "Risk Modeling with Risk Management Toolbox" on page 1-3

VaR Backtesting Workflow

This example shows a value-at-risk (VaR) backtesting workflow and the use of VaR backtesting tools. For a more comprehensive example of VaR backtesting, see “Value-at-Risk Estimation and Backtesting” on page 2-10.

Step 1. Load the VaR backtesting data.

Use the `VaRBacktestData.mat` file to load the VaR data into the workspace. This example works with the `EquityIndex`, `Normal95`, and `Normal99` numeric arrays. These arrays are equity returns and the corresponding VaR data at 95% and 99% confidence levels is produced with a normal distribution (a variance-covariance approach). See “Value-at-Risk Estimation and Backtesting” on page 2-10 for an example on how to generate this VaR data.

```
load('VaRBacktestData')
disp([EquityIndex(1:5) Normal95(1:5) Normal99(1:5)])
```

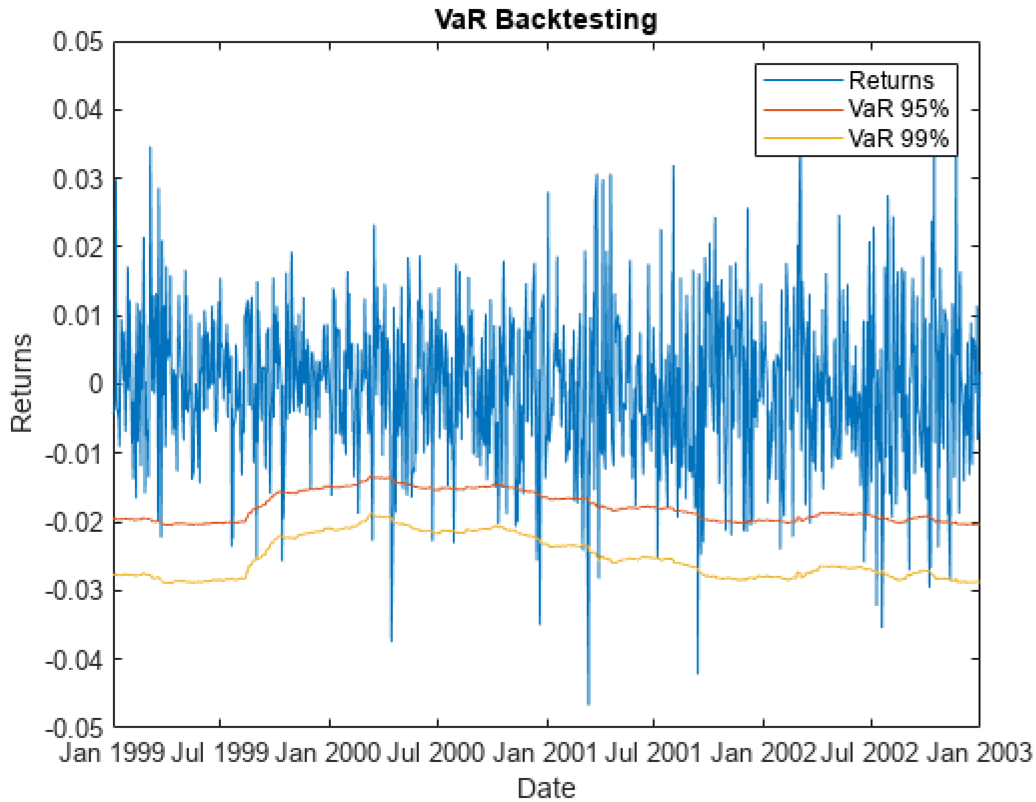
```
   -0.0043    0.0196    0.0277
   -0.0036    0.0195    0.0276
   -0.0000    0.0195    0.0275
    0.0298    0.0194    0.0275
    0.0023    0.0197    0.0278
```

The first column shows three losses in the first three days, but none of these losses exceeds the corresponding VaR (columns 2 and 3). The VaR model fails whenever the loss (negative of returns) exceeds the VaR.

Step 2. Generate a VaR backtesting plot.

Use the `plot` function to visualize the VaR backtesting data. This type of visualization is a common first step when performing a VaR backtesting analysis.

```
plot(Date,[EquityIndex -Normal95 -Normal99])
title('VaR Backtesting')
xlabel('Date')
ylabel('Returns')
legend('Returns','VaR 95%','VaR 99%')
```



Step 3. Create a `varbacktest` object.

Create a `varbacktest` object for the equity returns and the VaRs at 95% and 99% confidence levels.

```
vbt = varbacktest(EquityIndex,[Normal95 Normal99],...
   'PortfolioID','S&P', ...
   'VaRID',{'Normal95' 'Normal99'}, ...
   'VaRLevel',[0.95 0.99]);
disp(vbt)
```

```
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x2 double]
      PortfolioID: "S&P"
            VaRID: ["Normal95"    "Normal99"]
         VaRLevel: [0.9500 0.9900]
```

Step 4. Run a summary report.

Use the `summary` function to obtain a summary for the number of observations, the number of failures, and other simple metrics.

```
summary(vbt)
```

```
ans=2×10 table
    PortfolioID       VaRID        VaRLevel     ObservedLevel     Observations     Failures     Expecte
    ___________     __________     ________     _____________     ____________     ________     _______
```

```
      "S&P"       "Normal95"     0.95         0.94535          1043           57        52.15
      "S&P"       "Normal99"     0.99          0.9837          1043           17        10.43
```

Step 5. Run all tests.

Use the `runtests` function to display the final test results all at once.

```
runtests(vbt)
```

```
ans=2×11 table
    PortfolioID       VaRID        VaRLevel      TL        Bin       POF       TUFF       CC       C
    ___________    __________    ________    ______    ______    ______    ______    ______    _

      "S&P"       "Normal95"     0.95      green     accept    accept    accept    accept    ac
      "S&P"       "Normal99"     0.99      yellow    reject    accept    accept    accept    ac
```

Step 6. Run individual tests.

After running all tests, you can investigate the details of particular tests. For example, use the `tl` function to run the traffic light test.

```
tl(vbt)
```

```
ans=2×9 table
    PortfolioID       VaRID        VaRLevel      TL      Probability      TypeI      Increase     Obse
    ___________    __________    ________    ______    ___________    _______    ________    ____

      "S&P"       "Normal95"     0.95      green       0.77913      0.26396           0
      "S&P"       "Normal99"     0.99      yellow      0.97991      0.03686     0.26582
```

Step 7. Create VaR backtests for multiple portfolios.

You can create VaR backtests for different portfolios, or the same portfolio over different time windows. Run tests over two different subwindows of the original test window.

```
Ind1 = year(Date)<=2000;
Ind2 = year(Date)>2000;

vbt1 = varbacktest(EquityIndex(Ind1),[Normal95(Ind1,:) Normal99(Ind1,:)],...
   'PortfolioID','S&P, 1999-2000',...
   'VaRID',{'Normal95' 'Normal99'},...
   'VaRLevel',[0.95 0.99]);

vbt2 = varbacktest(EquityIndex(Ind2),[Normal95(Ind2,:) Normal99(Ind2,:)],...
   'PortfolioID','S&P, 2001-2002',...
   'VaRID',{'Normal95' 'Normal99'},...
   'VaRLevel',[0.95 0.99]);
```

Step 8. Display a summary report for both portfolios.

Use the `summary` function to display a summary for both portfolios.

```
Summary = [summary(vbt1); summary(vbt2)];
disp(Summary)
```

```
    PortfolioID        VaRID       VaRLevel    ObservedLevel    Observations    Failures    Ex
  ________________  ___________  ________  _____________  ____________  ________  __

  "S&P, 1999-2000"  "Normal95"     0.95       0.94626          521           28        2
  "S&P, 1999-2000"  "Normal99"     0.99       0.98464          521            8
  "S&P, 2001-2002"  "Normal95"     0.95       0.94444          522           29
  "S&P, 2001-2002"  "Normal99"     0.99       0.98276          522            9
```

Step 9. Run all tests for both portfolios.

Use the `runtests` function to display the final test result for both portfolios.

```
Results = [runtests(vbt1);runtests(vbt2)];
disp(Results)
```

```
    PortfolioID        VaRID       VaRLevel     TL       Bin      POF      TUFF      CC
  ________________  ___________  ________  ______  ______  ______  ______  ______

  "S&P, 1999-2000"  "Normal95"     0.95     green   accept  accept  accept  accept
  "S&P, 1999-2000"  "Normal99"     0.99     green   accept  accept  accept  accept
  "S&P, 2001-2002"  "Normal95"     0.95     green   accept  accept  accept  accept
  "S&P, 2001-2002"  "Normal99"     0.99     yellow  accept  accept  accept  accept
```

See Also

`varbacktest` | `tl` | `bin` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Related Examples

- “Value-at-Risk Estimation and Backtesting” on page 2-10

More About

- “Traffic Light Test” on page 2-3
- “Binomial Test” on page 2-2
- “Kupiec’s POF and TUFF Tests” on page 2-3
- “Christoffersen’s Interval Forecast Tests” on page 2-4
- “Haas’s Time Between Failures or Mixed Kupiec’s Test” on page 2-4

Value-at-Risk Estimation and Backtesting

This example shows how to estimate the value-at-risk (VaR) using three methods and perform a VaR backtesting analysis. The three methods are:

1. Normal distribution
2. Historical simulation
3. Exponential weighted moving average (EWMA)

Value-at-risk is a statistical method that quantifies the risk level associated with a portfolio. The VaR measures the maximum amount of loss over a specified time horizon and at a given confidence level.

Backtesting measures the accuracy of the VaR calculations. Using VaR methods, the loss forecast is calculated and then compared to the actual losses at the end of the next day. The degree of difference between the predicted and actual losses indicates whether the VaR model is underestimating or overestimating the risk. As such, backtesting looks retrospectively at data and helps to assess the VaR model.

The three estimation methods used in this example estimate the VaR at 95% and 99% confidence levels.

Load the Data and Define the Test Window

Load the data. The data used in this example is from a time series of returns on the S&P index from 1993 through 2003.

```
load VaRExampleData.mat
Returns = tick2ret(sp);
DateReturns = dates(2:end);
SampleSize = length(Returns);
```

Define the estimation window as 250 trading days. The test window starts on the first day in 1996 and runs through the end of the sample.

```
TestWindowStart      = find(year(DateReturns)==1996,1);
TestWindow           = TestWindowStart : SampleSize;
EstimationWindowSize = 250;
```

For a VaR confidence level of 95% and 99%, set the complement of the VaR level.

```
pVaR = [0.05 0.01];
```

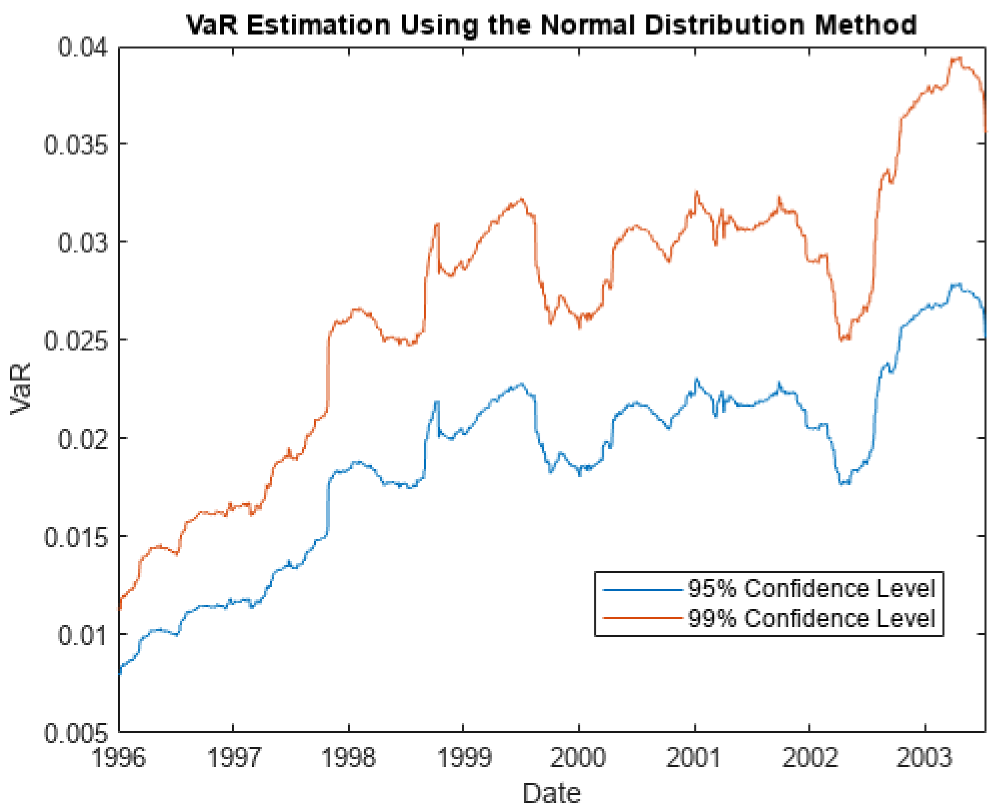
These values mean that there is at most a 5% and 1% probability, respectively, that the loss incurred will be greater than the maximum threshold (that is, greater than the VaR).

Compute the VaR Using the Normal Distribution Method

For the normal distribution method, assume that the profit and loss of the portfolio is normally distributed. Using this assumption, compute the VaR by multiplying the z-score, at each confidence level by the standard deviation of the returns. Because VaR backtesting looks retrospectively at data, the VaR "today" is computed based on values of the returns in the last $N = 250$ days leading to, but not including, "today."

```
Zscore   = norminv(pVaR);
Normal95 = zeros(length(TestWindow),1);
```

```
Normal99 = zeros(length(TestWindow),1);

for t = TestWindow
    i = t - TestWindowStart + 1;
    EstimationWindow = t-EstimationWindowSize:t-1;
    Sigma = std(Returns(EstimationWindow));
    Normal95(i) = -Zscore(1)*Sigma;
    Normal99(i) = -Zscore(2)*Sigma;
end

figure;
plot(DateReturns(TestWindow),[Normal95 Normal99])
xlabel('Date')
ylabel('VaR')
legend({'95% Confidence Level','99% Confidence Level'},'Location','Best')
title('VaR Estimation Using the Normal Distribution Method')
```

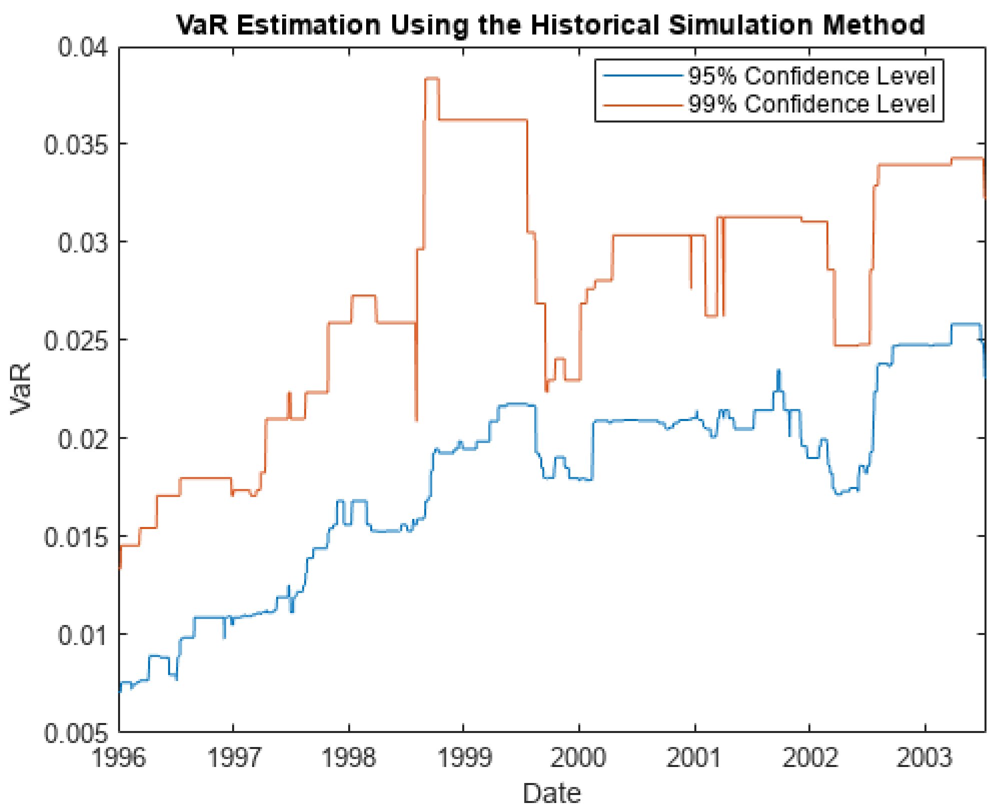


The normal distribution method is also known as parametric VaR because its estimation involves computing a parameter for the standard deviation of the returns. The advantage of the normal distribution method is its simplicity. However, the weakness of the normal distribution method is the assumption that returns are normally distributed. Another name for the normal distribution method is the variance-covariance approach.

Compute the VaR Using the Historical Simulation Method

Unlike the normal distribution method, the historical simulation (HS) is a nonparametric method. It does not assume a particular distribution of the asset returns. Historical simulation forecasts risk by assuming that past profits and losses can be used as the distribution of profits and losses for the next period of returns. The VaR "today" is computed as the *p* th-quantile of the last *N* returns prior to "today."

```
Historical95 = zeros(length(TestWindow),1);
Historical99 = zeros(length(TestWindow),1);

for t = TestWindow
    i = t - TestWindowStart + 1;
    EstimationWindow = t-EstimationWindowSize:t-1;
    X = Returns(EstimationWindow);
    Historical95(i) = -quantile(X,pVaR(1));
    Historical99(i) = -quantile(X,pVaR(2));
end

figure;
plot(DateReturns(TestWindow),[Historical95 Historical99])
ylabel('VaR')
xlabel('Date')
legend({'95% Confidence Level','99% Confidence Level'},'Location','Best')
title('VaR Estimation Using the Historical Simulation Method')
```



The preceding figure shows that the historical simulation curve has a piecewise constant profile. The reason for this is that quantiles do not change for several days until extreme events occur. Thus, the historical simulation method is slow to react to changes in volatility.

Compute the VaR Using the Exponential Weighted Moving Average Method (EWMA)

The first two VaR methods assume that all past returns carry the same weight. The exponential weighted moving average (EWMA) method assigns nonequal weights, particularly exponentially decreasing weights. The most recent returns have higher weights because they influence "today's" return more heavily than returns further in the past. The formula for the EWMA variance over an estimation window of size W_E is:

$$\hat{\sigma}_t^2 = \frac{1}{c}\sum_{i=1}^{W_E} \lambda^{i-1} y_{t-i}^2$$

where c is a normalizing constant:

$$c = \sum_{i=1}^{W_E} \lambda^{i-1} = \frac{1-\lambda^{W_E}}{1-\lambda} \quad \rightarrow \frac{1}{1-\lambda}\, as\, W_E \rightarrow \infty$$

For convenience, we assume an infinitely large estimation window to approximate the variance:

$$\hat{\sigma}_t^2 \approx (1-\lambda)(y_{t-1}^2 + \sum_{i=2}^{\infty} \lambda^{i-1} y_{t-i}^2) = (1-\lambda) y_{t-1}^2 + \lambda \hat{\sigma}_{t-1}^2$$

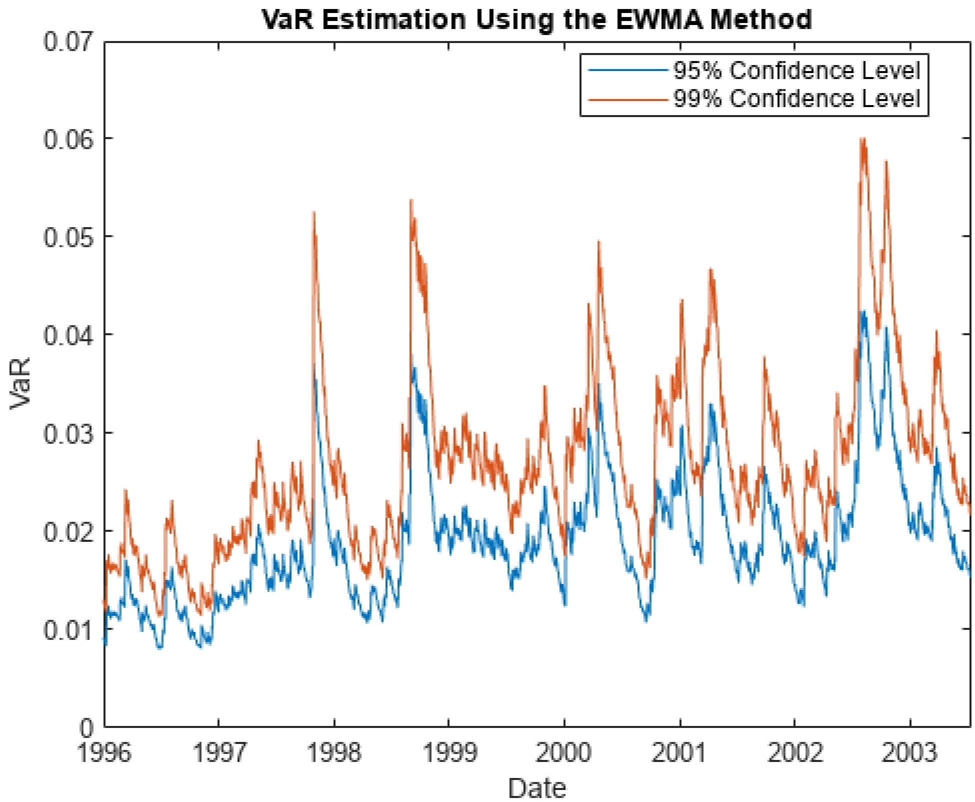
A value of the decay factor frequently used in practice is 0.94. This is the value used in this example. For more information, see References.

Initiate the EWMA using a warm-up phase to set up the standard deviation.

```
Lambda = 0.94;
Sigma2     = zeros(length(Returns),1);
Sigma2(1)  = Returns(1)^2;

for i = 2 : (TestWindowStart-1)
    Sigma2(i) = (1-Lambda) * Returns(i-1)^2 + Lambda * Sigma2(i-1);
end
```

Use the EWMA in the test window to estimate the VaR.

```
Zscore = norminv(pVaR);
EWMA95 = zeros(length(TestWindow),1);
EWMA99 = zeros(length(TestWindow),1);

for t = TestWindow
    k     = t - TestWindowStart + 1;
    Sigma2(t) = (1-Lambda) * Returns(t-1)^2 + Lambda * Sigma2(t-1);
    Sigma = sqrt(Sigma2(t));
    EWMA95(k) = -Zscore(1)*Sigma;
    EWMA99(k) = -Zscore(2)*Sigma;
end

figure;
plot(DateReturns(TestWindow),[EWMA95 EWMA99])
ylabel('VaR')
xlabel('Date')
legend({'95% Confidence Level','99% Confidence Level'},'Location','Best')
title('VaR Estimation Using the EWMA Method')
```



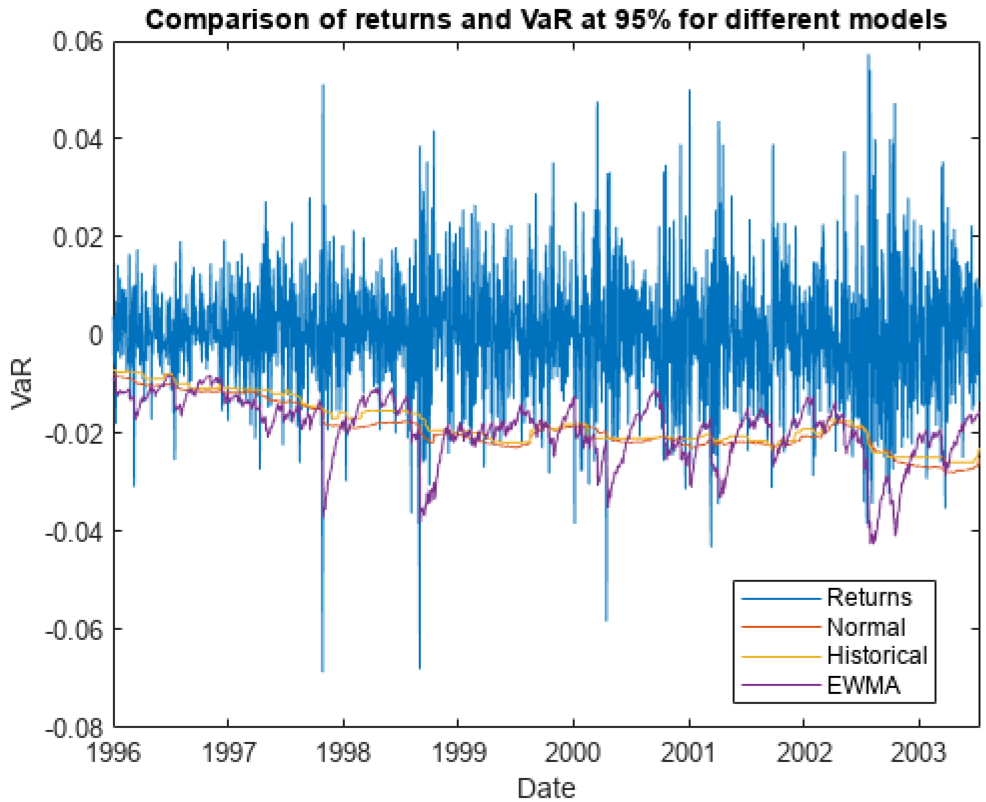
In the preceding figure, the EWMA reacts very quickly to periods of large (or small) returns.

VaR Backtesting

In the first part of this example, VaR was estimated over the test window with three different methods and at two different VaR confidence levels. The goal of VaR backtesting is to evaluate the performance of VaR models. A VaR estimate at 95% confidence is violated only about 5% of the time, and VaR failures do not cluster. Clustering of VaR failures indicates the lack of independence across time because the VaR models are slow to react to changing market conditions.

A common first step in VaR backtesting analysis is to plot the returns and the VaR estimates together. Plot all three methods at the 95% confidence level and compare them to the returns.

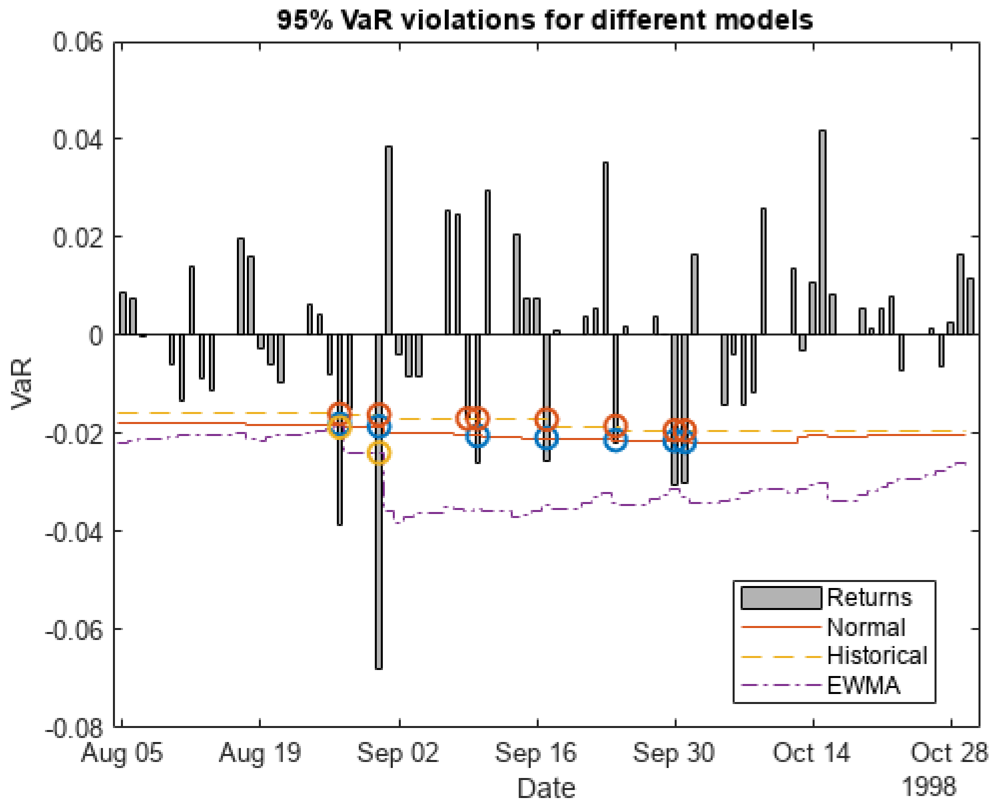
```
ReturnsTest = Returns(TestWindow);
DatesTest   = DateReturns(TestWindow);
figure;
plot(DatesTest,[ReturnsTest -Normal95 -Historical95 -EWMA95])
ylabel('VaR')
xlabel('Date')
legend({'Returns','Normal','Historical','EWMA'},'Location','Best')
title('Comparison of returns and VaR at 95% for different models')
```



To highlight how the different approaches react differently to changing market conditions, you can zoom in on the time series where there is a large and sudden change in the value of returns. For example, around August 1998:

```
ZoomInd   = (DatesTest >= datetime(1998,8,5)) & (DatesTest <= datetime(1998,10,31));
VaRData   = [-Normal95(ZoomInd) -Historical95(ZoomInd) -EWMA95(ZoomInd)];
VaRFormat = {'-','--','-.'};
D = DatesTest(ZoomInd);
R = ReturnsTest(ZoomInd);
N = Normal95(ZoomInd);
H = Historical95(ZoomInd);
E = EWMA95(ZoomInd);
IndN95    = (R < -N);
IndHS95   = (R < -H);
IndEWMA95 = (R < -E);
figure;
bar(D,R,0.5,'FaceColor',[0.7 0.7 0.7]);
hold on
for i = 1 : size(VaRData,2)
    stairs(D-0.5,VaRData(:,i),VaRFormat{i});
end
ylabel('VaR')
xlabel('Date')
legend({'Returns','Normal','Historical','EWMA'},'Location','Best','AutoUpdate','Off')
title('95% VaR violations for different models')
ax = gca;
ax.ColorOrderIndex = 1;
```

```
plot(D(IndN95),-N(IndN95),'o',D(IndHS95),-H(IndHS95),'o',...
   D(IndEWMA95),-E(IndEWMA95),'o','MarkerSize',8,'LineWidth',1.5)
xlim([D(1)-1, D(end)+1])
hold off;
```



A VaR failure or violation happens when the returns have a negative VaR. A closer look around August 27 to August 31 shows a significant dip in the returns. On the dates starting from August 27 onward, the EWMA follows the trend of the returns closely and more accurately. Consequently, EWMA has fewer VaR violations (two (2) violations, yellow diamonds) compared to the Normal Distribution approach (seven (7) violations, blue stars) or the Historical Simulation method (eight (8) violations, red squares).

Besides visual tools, you can use statistical tests for VaR backtesting. In Risk Management Toolbox™, a `varbacktest` object supports multiple statistical tests for VaR backtesting analysis. In this example, start by comparing the different test results for the normal distribution approach at the 95% and 99% VaR levels.

```
vbt = varbacktest(ReturnsTest,[Normal95 Normal99],'PortfolioID','S&P','VaRID',...
    {'Normal95','Normal99'},'VaRLevel',[0.95 0.99]);
summary(vbt)
```

```
ans=2×10 table
    PortfolioID       VaRID        VaRLevel    ObservedLevel    Observations    Failures    Expecte
    ___________    ___________    ________    _____________    ____________    ________    _______

       "S&P"       "Normal95"       0.95         0.94863           1966           101        98.3
```

```
      "S&P"       "Normal99"      0.99         0.98372          1966           32        19.66
```

The summary report shows that the observed level is close enough to the defined VaR level. The 95% and 99% VaR levels have at most `(1-VaR_level) x` *N* expected failures, where *N* is the number of observations. The failure ratio shows that the `Normal95` VaR level is within range, whereas the `Normal99` VaR Level is imprecise and under-forecasts the risk. To run all tests supported in `varbacktest`, use `runtests`.

```
runtests(vbt)
```

```
ans=2×11 table
    PortfolioID      VaRID       VaRLevel     TL       Bin       POF      TUFF       CC
    ___________    __________    ________   ______   ______   ______   ______   ______   _

      "S&P"        "Normal95"      0.95     green    accept   accept   accept   accept   re
      "S&P"        "Normal99"      0.99     yellow   reject   reject   accept   reject   ac
```

The 95% VaR passes the frequency tests, such as traffic light, binomial and proportion of failures tests (`tl`, `bin`, and `pof` columns). The 99% VaR does not pass these same tests, as indicated by the `yellow` and `reject` results. Both confidence levels got rejected in the conditional coverage independence, and time between failures independence (`cci` and `tbfi` columns). This result suggests that the VaR violations are not independent, and there are probably periods with multiple failures in a short span. Also, one failure may make it more likely that other failures will follow in subsequent days. For more information on the tests methodologies and the interpretation of results, see `varbacktest` and the individual tests.

Using a `varbacktest` object, run the same tests on the portfolio for the three approaches at both VaR confidence levels.

```
vbt = varbacktest(ReturnsTest,[Normal95 Historical95 EWMA95 Normal99 Historical99 ...
    EWMA99],'PortfolioID','S&P','VaRID',{'Normal95','Historical95','EWMA95',...
    'Normal99','Historical99','EWMA99'},'VaRLevel',[0.95 0.95 0.95 0.99 0.99 0.99]);
runtests(vbt)
```

```
ans=6×11 table
    PortfolioID        VaRID         VaRLevel     TL       Bin       POF      TUFF       CC
    ___________    ______________    ________   ______   ______   ______   ______   ______

      "S&P"        "Normal95"          0.95     green    accept   accept   accept   accept
      "S&P"        "Historical95"      0.95     yellow   accept   accept   accept   accept
      "S&P"        "EWMA95"            0.95     green    accept   accept   accept   accept
      "S&P"        "Normal99"          0.99     yellow   reject   reject   accept   reject
      "S&P"        "Historical99"      0.99     yellow   reject   reject   accept   reject
      "S&P"        "EWMA99"            0.99     red      reject   reject   accept   reject
```

The results are similar to the previous results, and at the 95% level, the frequency results are generally acceptable. However, the frequency results at the 99% level are generally rejections. Regarding independence, most tests pass the conditional coverage independence test (`cci`), which tests for independence on consecutive days. Notice that all tests fail the time between failures independence test (`tbfi`), which takes into account the times between all failures. This result suggests that all methods have issues with the independence assumption.

To better understand how these results change given market conditions, look at the years 2000 and 2002 for the 95% VaR confidence level.

```
Ind2000 = (year(DatesTest) == 2000);
vbt2000 = varbacktest(ReturnsTest(Ind2000),[Normal95(Ind2000) Historical95(Ind2000) EWMA95(Ind200
   'PortfolioID','S&P, 2000','VaRID',{'Normal','Historical','EWMA'});
runtests(vbt2000)
```

```
ans=3×11 table
    PortfolioID        VaRID         VaRLevel     TL       Bin       POF       TUFF       CC
    ___________    ____________    ________    _____    ______    ______    ______    ______    _

    "S&P, 2000"    "Normal"          0.95      green    accept    accept    accept    accept    a
    "S&P, 2000"    "Historical"      0.95      green    accept    accept    accept    accept    a
    "S&P, 2000"    "EWMA"            0.95      green    accept    accept    accept    accept    a
```

```
Ind2002 = (year(DatesTest) == 2002);
vbt2002 = varbacktest(ReturnsTest(Ind2002),[Normal95(Ind2002) Historical95(Ind2002) EWMA95(Ind200
   'PortfolioID','S&P, 2002','VaRID',{'Normal','Historical','EWMA'});
runtests(vbt2002)
```

```
ans=3×11 table
    PortfolioID        VaRID         VaRLevel      TL        Bin       POF       TUFF       CC
    ___________    ____________    ________    ______    ______    ______    ______    ______

    "S&P, 2002"    "Normal"          0.95      yellow    reject    reject    accept    reject
    "S&P, 2002"    "Historical"      0.95      yellow    reject    accept    accept    reject
    "S&P, 2002"    "EWMA"            0.95      green     accept    accept    accept    accept
```

For the year 2000, all three methods pass all the tests. However, for the year 2002, the test results are mostly rejections for all methods. The EWMA method seems to perform better in 2002, yet all methods fail the independence tests.

To get more insight into the independence tests, look into the conditional coverage independence (`cci`) and the time between failures independence (`tbfi`) test details for the year 2002. To access the test details for all tests, run the individual test functions.

```
cci(vbt2002)
```

```
ans=3×13 table
    PortfolioID        VaRID         VaRLevel     CCI      LRatioCCI    PValueCCI    Observations
    ___________    ____________    ________    ______    _________    _________    ____________

    "S&P, 2002"    "Normal"          0.95      reject     12.591      0.0003877        261
    "S&P, 2002"    "Historical"      0.95      reject     6.3051       0.012039        261
    "S&P, 2002"    "EWMA"            0.95      reject     4.6253       0.031504        261
```

In the CCI test, the probability *p* `01` of having a failure at time *t*, knowing that there was no failure at time *t*-1 is given by

$$p_{01} = \frac{N_{01}}{N_{01} + N_{00}}$$

The probability *p* `11` of having a failure at time *t*, knowing that there was failure at time *t*-1 is given by

$$p_{11} = \frac{N_{11}}{N_{11} + N_{10}}$$

From the `N00`, `N10`, `N01`, `N11` columns in the test results, the value of *p* `01` is at around 5% for the three methods, yet the values of *p* `11` are above 20%. Because there is evidence that a failure is followed by another failure much more frequently than 5% of the time, this CCI test fails.

In the time between failures independence test, look at the minimum, maximum, and quartiles of the distribution of times between failures, in the columns `TBFMin`, `TBFQ1`, `TBFQ2`, `TBFQ3`, `TBFMax`.

```
tbfi(vbt2002)
```

```
ans=3×14 table
    PortfolioID        VaRID          VaRLevel     TBFI     LRatioTBFI    PValueTBFI    Observations
    ___________    ____________       ________    ______    __________    __________    ____________

    "S&P, 2002"    "Normal"             0.95      reject      53.936      0.00010087        261
    "S&P, 2002"    "Historical"         0.95      reject      45.274       0.0010127        261
    "S&P, 2002"    "EWMA"               0.95      reject      25.756        0.027796        261
```

For a VaR level of 95%, you expect an average time between failures of 20 days, or one failure every 20 days. However, the median of the time between failures for the year 2002 ranges between 5 and 7.5 for the three methods. This result suggests that half of the time, two consecutive failures occur within 5 to 7 days, much more frequently than the 20 expected days. Consequently, more test failures occur. For the normal method, the first quartile is 1, meaning that 25% of the failures occur on consecutive days.

References

Nieppola, O. *Backtesting Value-at-Risk Models*. Helsinki School of Economics. 2009.

Danielsson, J. *Financial Risk Forecasting: The Theory and Practice of Forecasting Market Risk, with Implementation in R and MATLAB®*. Wiley Finance, 2012.

See Also

`varbacktest` | `tl` | `bin` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Related Examples

- “VaR Backtesting Workflow” on page 2-6

More About

- “Traffic Light Test” on page 2-3
- “Binomial Test” on page 2-2
- “Kupiec’s POF and TUFF Tests” on page 2-3
- “Christoffersen’s Interval Forecast Tests” on page 2-4
- “Haas’s Time Between Failures or Mixed Kupiec’s Test” on page 2-4

Overview of Expected Shortfall Backtesting

Expected Shortfall (ES) is the expected loss on days when there is a Value-at-Risk (VaR) failure. If the VaR is 10 million and the ES is 12 million, we know the expected loss tomorrow; if it happens to be a very bad day, it is 20% higher than the VaR. ES is sometimes called Conditional Value-at-Risk (CVaR), Tail Value-at-Risk (TVaR), Tail Conditional Expectation (TCE), or Conditional Tail Expectation (CTE).

There are many approaches to estimating VaR and ES, and they may lead to different VaR and ES estimates. How can one determine if models are accurately estimating the risk on a daily basis? How can one evaluate which model performs better? The `varbacktest` tools help validate the performance of VaR models with regards to estimated VaR values. The `esbacktest`, `esbacktestbysim`, and `esbacktestbyde` tools extend these capabilities to evaluate VaR models with regards to estimated ES values.

For VaR backtesting, the possibilities every day are two: either there is a VaR failure or not. If the VaR confidence level is 95%, VaR failures should happen approximately 5% of the time. To backtest VaR, you only need to know whether the VaR was exceeded (VaR failure) or not on each day of the test window and the VaR confidence level. Risk Management Toolbox VaR backtesting tools support “frequency” (assess the proportion of failures) and “independence” (assess independence across time) tests, and these tests work with the binary sequence of "failure" or "no-failure" results over the test window.

For expected shortfall (ES), the possibilities every day are infinite: The VaR may be exceeded by 1%, or by 10%, or by 150%, and so on. For example, there are three VaR failures in the following example:

Failure Data					Severity Ratio	
					Observed	Expected
Date	Return	VaR	ES		(-Return/VaR)	(ES/VaR)
26-Feb-96	-1.308	1.078	1.364		1.21	1.27
8-Mar-96	-2.051	1.110	1.404		1.85	1.27
10-Apr-96	-1.353	1.218	1.541		1.11	1.27
				Average	1.39	1.27

On failure days, the VaR is exceeded on average by 39%, but the estimated ES exceeds VaR by an average of 27%. How can you tell if 39% is significantly larger than 27%? Knowing the VaR confidence level is not enough, you must also know how likely are the different exceedances over the VaR according to the VaR model. In other words, you need some distribution information about what happens beyond the VaR according to your model assumptions. For thin-tail VaR models, 39% vs. 27% may be a large difference. However, for a heavy-tail VaR model where a severity of twice the VaR has a non-trivial probability of happening, then 39% vs. 27% over the three failure dates may not be a red flag.

A key difference between VaR backtesting and ES backtesting is that most ES backtesting methods require information about the distribution of the returns on each day, or at least the distribution of the tails beyond the VaR. One exception is the “unconditional” test (see `unconditionalNormal` and `unconditionalT`) where you can get approximate test results without providing the distribution information. This is important in practice, because the “unconditional” test is much simpler to use and can be used in principle for any VaR or ES model. The trade-off is that the approximate results may be inaccurate, especially in borderline accept, or reject cases, or for certain types of distributions.

The toolbox supports the following tests for expected shortfall backtesting for table-based tests for the unconditional Acerbi-Szekely test using the `esbacktest` object:

- `unconditionalNormal`
- `unconditionalT`

ES backtests are necessarily approximated in that they are sensitive to errors in the predicted VaR. However, the minimally biased test has only a small sensitivity to VaR errors and the sensitivity is prudential, in the sense that VaR errors lead to a more punitive ES test. See Acerbi-Szekely (2017 and 2019) for details. When distribution information is available, the minimally biased test (`minBiasRelative` or `minBiasAbsolute`) is recommended.

The toolbox supports the following Acerbi-Szekely simulation-based tests for expected shortfall backtesting using the `esbacktestbysim` object:

- `conditional`
- `unconditional`
- `quantile`
- `minBiasRelative`
- `minBiasAbsolute`

For the Acerbi-Szekely simulation-based tests, you must provide the model distribution information as part of the inputs to `esbacktestbysim`.

The toolbox also supports the following Du and Escanciano tests for expected shortfall backtesting using the `esbacktestbyde` object:

- `unconditionalDE`
- `conditionalDE`

For the Du and Escanciano simulation-based tests, you must provide the model distribution information as part of the inputs to `esbacktestbyde`.

Conditional Test by Acerbi and Szekely

The conditional test statistic by Acerbi and Szekely is based on the conditional relationship

$$ES_t = -E_t[X_t | X_t < -VaR_t]$$

where

X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period *t*.

VaR_t is the estimated VaR for period *t*.

ES_t is the estimated expected shortfall for period *t*.

The number of failures is defined as

$$NumFailures = \sum_{t=1}^{N} I_t$$

where

N is the number of periods in the test window ($t = 1,\ldots,N$).

I_t is the VaR failure indicator on period *t* with a value of 1 if X_t < -VaR, and 0 otherwise.

The conditional test statistic is defined as

$$Z_{cond} = \frac{1}{NumFailures}\sum_{t=1}^{N}\frac{X_t I_t}{ES_t} + 1$$

The conditional test has two parts. A VaR backtest must be run for the number of failures (`NumFailures`), and a standalone conditional test is performed for the conditional test statistic Z_{cond}. The conditional test accepts the model only when both the VaR test and the standalone conditional test accept the model. For more information, see `conditional`.

Unconditional Test by Acerbi and Szekely

The unconditional test statistic by Acerbi and Szekely is based on the unconditional relationship,

$$ES_t = -E_t\left[\frac{X_t I_t}{p_{VaR}}\right]$$

where

X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period *t*.

P_{VaR} is the probability of VaR failure defined as 1-VaR level.

ES_t is the estimated expected shortfall for period *t*.

I_t is the VaR failure indicator on period *t* with a value of 1 if X_t < -VaR, and 0 otherwise.

The unconditional test statistic is defined as

$$Z_{uncond} = \frac{1}{Np_{VaR}}\sum_{t=1}^{N}\frac{X_t I_t}{ES_t} + 1$$

The critical values for the unconditional test statistic are stable across a range of distributions, which is the basis for the table-based tests. The `esbacktest` class runs the unconditional test against precomputed critical values under two distributional assumptions, namely, normal distribution (thin tails, see `unconditionalNormal`), and *t* distribution with 3 degrees of freedom (heavy tails, see `unconditionalT`).

Quantile Test by Acerbi and Szekely

A sample estimator of the expected shortfall for a sample $Y_1,\ldots,Y_N$ is:

$$\widehat{ES}(Y) = -\frac{1}{\lfloor Np_{VaR}\rfloor}\sum_{i=1}^{\lfloor Np_{VaR}\rfloor} Y_{[i]}$$

where

N is the number of periods in the test window ($t = 1,\ldots,N$).

P_{VaR} is the probability of VaR failure defined as 1-VaR level.

$Y_1,\ldots,Y_N$ are the sorted sample values (from smallest to largest), and $\lfloor Np_{VaR} \rfloor$ is the largest integer less than or equal to Np_{VaR}.

To compute the quantile test statistic, a sample of size N is created at each time *t* as follows. First, convert the portfolio outcomes to X_t to ranks $U_1 = P_1(X_1), \ldots, U_N = P_N(X_N)$ using the cumulative distribution function P_t. If the distribution assumptions are correct, the rank values $U_1,\ldots,U_N$ are uniformly distributed in the interval (0,1). Then at each time *t*:

1. Invert the ranks $U = (U_1,\ldots,U_N)$ to get N quantiles $P_t^{-1}(U) = (P_t^{-1}(U_1), \ldots, P_t^{-1}(U_N))$.
2. Compute the sample estimator $\widehat{ES}(P_t^{-1}(U))$.
3. Compute the expected value of the sample estimator $E\left[\widehat{ES}(P_t^{-1}(V))\right]$

   where $V = (V_1,\ldots,V_N)$ is a sample of N independent uniform random variables in the interval (0,1). This can be computed analytically.

The quantile test statistic by Acerbi and Szekely is defined as

$$Z_{quantile} = -\frac{1}{N}\sum_{t=1}^{N}\frac{\widehat{ES}(P_t^{-1}(U))}{E[\widehat{ES}(P_t^{-1}(V))]} + 1$$

The denominator inside the sum can be computed analytically as

$$E[\widehat{ES}(P_t^{-1}(V))] = -\frac{N}{\lfloor N_{pVaR} \rfloor}\int_0^1 I_{1-p}(N - \lfloor N_{pVaR} \rfloor, \lfloor N_{pVaR} \rfloor)P_t^{-1}(p)dp$$

where $I_x(z,w)$ is the regularized incomplete beta function. For more information, see `betainc` and `quantile`.

Minimally Biased Test by Acerbi and Szekely

The *minimally biased* test statistic by Acerbi and Szekely is based on the following representation of the VaR and ES (see Acerbi and Szekely 2017 and 2019 for details and also Rockafellar and Uryasev 2002, and Acerbi and Tasche 2002):

$$ES_\alpha = \min_v E\left[v + \frac{1}{\alpha}(X + v)_-\right]$$

$$VaR_\alpha = \mathrm{argmin}_v E\left[v + \frac{1}{\alpha}(X + v)_-\right]$$

where

X is the portfolio outcome.

$(x)_-$ is the negative part function defined as $(x)_- = \max(0,-x)$.

α is 1-VaR level.

The test statistic has an absolute version and a relative version. The absolute version of the minimally biased test statistic is given by

$$Z_{minbias}^{abs} = \frac{1}{N}\sum_{t=1}^{N}(ES_t - VaR_t - \frac{1}{p_{VaR}}(X_t + VaR_t)\_)$$

where

X_t is the portfolio outcome, that is the portfolio return or portfolio profit and loss for period t.

VaR_t is the essential VaR for period t.

ES_t is the expected shortfall for period t.

p_{VaR} is the probability of Var Failure defined as 1-VaR level.

N is the number of periods in the test window (t = 1,...N).

(x)_ is the negative part function defined as $(x)\_ = \max(0,-x)$.

The relative version of the minimally biased test statistic is given by

$$Z_{minbias}^{rel} = \frac{1}{N}\sum_{t=1}^{N}\frac{1}{ES_t}(ES_t - VaR_t - \frac{1}{p_{VaR}}(X_t + VaR_t)\_)$$

ES backtests are necessarily approximated in that they are sensitive to errors in the predicted VaR. However, the minimally biased test has only a small sensitivity to VaR errors and the sensitivity is prudential, in the sense that VaR errors lead to a more punitive ES test. See Acerbi-Szekely (2017 and 2019) for details. When distribution information is available, the minimally biased test is recommended. For more information, see `minBiasRelative` and `minBiasAbsolute`.

ES Backtest Using Du-Escanciano Method

For each day, the Du-Escanciano model assumes a distribution for the returns. For example, if you have a normal distribution with a conditional variance of 1.5%, there is a corresponding cumulative distribution function P_t. By mapping the returns X_t with the distribution P_t, you get the “mapped returns” series U_t, also known as the "ranks" series, which by construction has values between 0 and 1 (see column 2 in the following table). Let α be the complement of the VaR level — for example, if the VaR level is 95%, α is 5%. If the mapped return U_t is smaller than α, then there is a VaR “violation” or VaR “failure.” This is equivalent to observing a return X_t smaller than the negative of the VaR value for that day, since, by construction, the negative of the VaR value gets mapped to α. Therefore, you can compare U_t against α without even knowing the VaR value. The series of VaR failures is denoted by h_t and it is a series of 0's and 1's stored in column 3 in the following table. Finally, column 4 in the following table contains the “cumulative violations” series, denoted by H_t. This is the severity of the mapped VaR violations on days on which the VaR is violated. For example, if the mapped return U_t is 1% and α is 5%, H_t is 4%. H_t is defined as zero if there are no VaR violations.

X_t	$U_t = P_t(X_t)$	$h_t = U_t < \alpha$	$H_t = (\alpha - U_t) * h_t$
0.00208	0.5799	0	0
-0.01073	0.1554	0	0
-0.00825	0.2159	0	0
-0.02967	0.0073	1	0.0427
0.01242	0.8745	0	0

X_t	$U_t = P_t(X_t)$	$h_t = U_t < \alpha$	$H_t = (\alpha - U_t) * h_t$
...	...	...	...

Given the violations series h_t and the cumulative violations series H_t, the Du-Escanciano (DE) tests are summarized as:

Du-Escanciano Test	VaR Test	ES Test
Unconditional	Mean of h_t	Mean of H_t
Conditional	Autocorrelation of h_t	Autocorrelation of H_t

The DE VaR tests assess the mean value and the autocorrelation of the h_t series, and the resulting tests overlap with known VaR tests. For example, the mean of h_t is expected to match α. In other words, the proportion of time the VaR is violated is expected to match the confidence level. This test is supported in the `varbacktest` class with the proportion of failures (`pof`) test (finite sample) and the binomial (`bin`) test (large-sample approximation). In turn, the conditional VaR test measures if there is a time pattern in the sequence of VaR failures (back-to-back failures, and so on). The conditional coverage independence (`cci`) test in the `varbacktest` class tests for one-lag independence. The time between failures independence (`tbfi`) test in the `varbacktest` class also assesses time independence for VaR models.

The `esbacktestbyde` class supports the DE ES tests. The DE ES tests assess the mean value and the autocorrelation of the H_t series. For the unconditional test (`unconditionalDE`), the expected value is α/2 — for example, the average value in the bottom 5% of a uniform (0,1) distribution is 2.5%. The conditional test (`conditionalDE`) assesses not only if a failure occurs but also if the failure severity is correlated to previous failure occurrences and their severities.

The test statistic for the unconditional DE ES test is

$$U_{ES} = \frac{1}{N}\sum_{t=1}^{N} H_t$$

If the number of observations is large, the test statistic is distributed as

$$U_{ES} \xrightarrow[dist]{} N\left(\frac{\alpha}{2}, \frac{\alpha(1/3 - \alpha/4)}{N}\right) = P_U$$

where $N(\mu,\sigma^2)$ is the normal distribution with mean μ and variance σ^2.

The unconditional DE ES test is a two-sided test that checks if the test statistic is close to the expected value of α/2. From the limiting distribution, a confidence level is derived. Finite-sample confidence intervals are estimated through simulation.

The test statistic for the conditional DE ES test is derived in several steps. First, define the autocovariance for lag j:

$$\gamma_j = \frac{1}{N-j}\sum_{t=j+1}^{N} (H_t - \alpha/2)(H_{t-j} - \alpha/2)$$

The autocorrelation for lag j is then

$$\rho_j = \frac{\gamma_j}{\gamma_0}$$

The test statistic for m lags is then

$$C_{ES}(m) = N\sum_{j=1}^{m}\rho_j^2$$

If the number of observations is large, the test statistic is distributed as a chi-square distribution with m degrees of freedom:

$$C_{ES}(m) \xrightarrow[dist]{} \chi_m^2$$

The conditional DE ES test is a one-sided test to determine if the conditional DE ES test statistic is much larger than zero. If so, there is evidence of autocorrelation. The limiting distribution computes large-sample critical values. Finite-sample critical values are estimated through simulation.

Comparison of ES Backtesting Methods

The backtesting tools supported by Risk Management Toolbox have the following requirements and features.

Backtesting Tool	PortfolioData Required	VarData Required	ESData Required	VaRLevel Required[a]	PortfolioID and VaRID Supported	Distribution Information Required	Supports Multiple Models[b]	Supports Multiple VaRLevels
`varbacktest`	Yes	Yes	No	Yes	Yes	No	Yes	Yes
`esbacktest`	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
`esbacktestbysim`	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
`esbacktestbyde`	Yes	No	No	Yes	Yes	Yes	No	Yes

a VaRLevel is an optional name-value pair argument with a default value of 95%. It is recommended to set the VaRLevel when creating the backtesting object.

b For example, you can backtest a normal and a t model in the same object with varbacktest, but you need two separate instances of the esbacktestbyde class to backtest them.

Risk Management Toolbox supports the following backtesting tools and their associated tests.

Test Type	Test Name	Tests for	Risk Measure	Critical Value Computation	Use Object	Use Function
Basel	Traffic light	Frequency	VaR	Exact finite-sample (binomial)	`varbacktest`	`tl`

Test Type	Test Name	Tests for	Risk Measure	Critical Value Computation	Use Object	Use Function
Various	Binomial	Frequency	VaR	Large-sample normal approximation	`varbacktest`	`bin`
Kupiec	Proportion of failures	Frequency	VaR	Exact finite-sample (log likelihood)	`varbacktest`	`pof`
Kupiec	Time until first failure	Independence	VaR	Exact finite-sample (log likelihood)	`varbacktest`	`tuff`
Christoffersen	Conditional coverage, mixed	Frequency and independence	VaR	Exact finite-sample (log likelihood)	`varbacktest`	`cc`
Christoffersen	Conditional coverage, independence	Independence	VaR	Exact finite-sample (log likelihood)	`varbacktest`	`cci`
Haas	Mixed Kupiec test	Frequency and independence	VaR	Exact finite-sample (log likelihood)	`varbacktest`	`tbf`
Haas	Independence (time between failures)	Independence	VaR	Exact finite-sample (log likelihood)	`varbacktest`	`tbfi`
Acerbi-Szekely	"Test 2" or unconditional	Severity	ES	Tables of presimulated critical values, under normal and *t* distribution	`esbacktest`	`unconditionalNormal` and `unconditionalT`
Acerbi-Szekely	"Test 1" or conditional	Severity	ES	Finite-sample simulation	`esbacktestbysim`	`conditional`
Acerbi-Szekely	"Test 2" or unconditional	Severity	ES	Finite-sample simulation	`esbacktestbysim`	`unconditional`
Acerbi-Szekely	"Test 1" or ranks (quantile)	Severity	ES	Finite-sample simulation	`esbacktestbysim`	`quantile`

Test Type	Test Name	Tests for	Risk Measure	Critical Value Computation	Use Object	Use Function
Acerbi-Szekely	Minimally Biased, relative version	Severity	ES	Finite-sample simulation	`esbacktestbysim`	`minBiasRelative`
Acerbi-Szekely	Minimally Biased, absolute version	Severity	ES	Finite-sample simulation	`esbacktestbysim`	`minBiasAbsolute`
Du-Escanciano	Unconditional	Severity	ES	Large-sample approximation and finite-sample simulation	`esbacktestbyde`	`unconditionalDE`
Du-Escanciano	Conditional	Independence	ES	Large-sample approximation and finite-sample simulation	`esbacktestbyde`	`conditionalDE`

References

[1] Basel Committee on Banking Supervision. *Supervisory Framework for the Use of “Backtesting” in Conjunction with the Internal Models Approach to Market Risk Capital Requirements.* January 1996. https://www.bis.org/publ/bcbs22.htm.

[2] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall.* MSCI Inc. December 2014.

[3] Acerbi, C., and B. Szekely. "General Properties of Backtestable Statistics. *SSRN Electronic Journal.* January, 2017.

[4] Acerbi, C., and B. Szekely. "The Minimally Biased Backtest for ES." *Risk.* September, 2019.

[5] Acerbi, C. and D. Tasche. “On the Coherence of Expected Shortfall.” *Journal of Banking and Finance.* Vol. 26, 2002, pp. 1487-1503.

[6] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science.* Vol. 63, Issue 4, April 2017.

[7] Rockafellar, R. T. and S. Uryasev. "Conditional Value-at-Risk for General Loss Distributions." *Journal of Banking and Finance.* Vol. 26, 2002, pp. 1443-1471.

See Also

`esbacktestbyde` | `esbacktest` | `esbacktestbysim` | `varbacktest`

Related Examples

- “VaR Backtesting Workflow” on page 2-6
- “Value-at-Risk Estimation and Backtesting” on page 2-10
- “Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30
- “Expected Shortfall (ES) Backtesting Workflow Using Simulation” on page 2-34
- “Expected Shortfall Estimation and Backtesting” on page 2-44
- “Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63
- “Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-72

Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information

This example shows an expected shortfall (ES) backtesting workflow and the use of ES backtesting tools. The `esbacktest` class supports two tests -- unconditional normal and unconditional *t* -- which are based on Acerbi-Szekely's unconditional test statistic (also known as the Acerbi-Szekely second test). These tests use presimulated critical values for the unconditional test statistic, with an assumption of normal distribution for the normal case and a *t* distribution with 3 degrees of freedom for the *t* case.

Step 1. Load the ES backtesting data.

Use the `ESBacktestData.mat` file to load the data into the workspace. This example works with the `Returns` numeric array. This array represents the equity returns, `VaRModel1`, `VaRModel2`, and `VaRModel3`, and the corresponding VaR data at 97.5% confidence levels, generated with three different models. The expected shortfall data is contained in `ESModel1`, `ESModel2`, and `ESModel3`. The three model distributions used to generate the expected shortfall data in this example are normal (model 1), *t* with 10 degrees of freedom (model 2), and *t* with 5 degrees of freedom (model 3). However, this distribution information is not needed in this example because the `esbacktest` object does not require it.

```
load('ESBacktestData')
whos
```

```
  Name              Size              Bytes  Class        Attributes

  Data           1966x13             223945  timetable
  Dates          1966x1               15728  datetime
  ESModel1       1966x1               15728  double
  ESModel2       1966x1               15728  double
  ESModel3       1966x1               15728  double
  Returns        1966x1               15728  double
  VaRLevel          1x1                   8  double
  VaRModel1      1966x1               15728  double
  VaRModel2      1966x1               15728  double
  VaRModel3      1966x1               15728  double
```

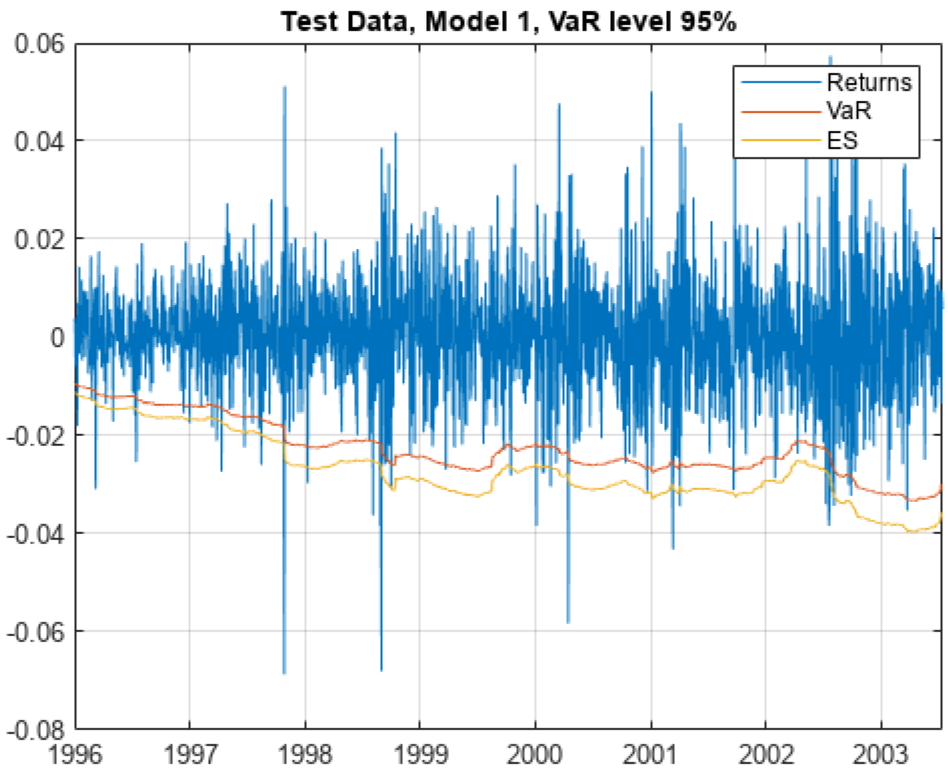
Step 2. Generate an ES backtesting plot.

Use the `plot` function to visualize the ES backtesting data. This type of visualization is a common first step when performing an ES backtesting analysis. For illustration purposes only, visualize the returns, together with VaR and ES, for a particular model.

The resulting plot shows some large violations in 1997, 1998, and 2000. The violations in 1996 look smaller in absolute terms, however relative to the volatility of that period, those violations are also significant. For the unconditional test, the magnitude of the violations and the number of violations make a difference, because the test statistic averages over the *expected* number of failures. If the expected number is small, but there are several violations, the effective severity for the test is larger. The year 2002 is an example of a year with small, but many VaR failures.

```
figure;
plot(Dates,Returns,Dates,-VaRModel1,Dates,-ESModel1)
legend('Returns','VaR','ES')
```

```
title('Test Data, Model 1, VaR level 95%')
grid on
```



Step 3. Create an `esbacktest` object.

Create an `esbacktest` object using `esbacktest`.

```
load ESBacktestData
ebt = esbacktest(Returns,[VaRModel1 VaRModel2 VaRModel3],[ESModel1 ESModel2 ESModel3],...
    'PortfolioID',"S&P",'VaRID',["Model1","Model2","Model3"],'VaRLevel',VaRLevel)
```

```
ebt = 
  esbacktest with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x3 double]
           ESData: [1966x3 double]
      PortfolioID: "S&P"
            VaRID: ["Model1"    "Model2"    "Model3"]
         VaRLevel: [0.9750 0.9750 0.9750]
```

Step 4. Generate the ES summary report.

Generate the ES summary report. The `ObservedSeverity` column shows the average ratio of loss to VaR on periods when the VaR is violated. The `ExpectedSeverity` column shows the average ratio of ES to VaR for the VaR violation periods.

```
S = summary(ebt);
disp(S)
```

```
    PortfolioID     VaRID      VaRLevel    ObservedLevel    ExpectedSeverity    ObservedSeverity
    ___________    ________    ________    _____________    ________________    ________________

       "S&P"       "Model1"     0.975         0.97101            1.1928              1.4221
       "S&P"       "Model2"     0.975         0.97202            1.2652              1.4134
       "S&P"       "Model3"     0.975         0.97202              1.37              1.4146
```

Step 5. Run a report for all tests.

Run all tests and generate a report only on the accept or reject results.

```
t = runtests(ebt);
disp(t)
```

```
    PortfolioID     VaRID      VaRLevel    UnconditionalNormal    UnconditionalT
    ___________    ________    ________    ___________________    ______________

       "S&P"       "Model1"     0.975            reject               reject
       "S&P"       "Model2"     0.975            reject               accept
       "S&P"       "Model3"     0.975            accept               accept
```

Step 6. Run the unconditional normal test.

Run the individual test for the unconditional normal test.

```
t = unconditionalNormal(ebt);
disp(t)
```

```
    PortfolioID     VaRID      VaRLevel    UnconditionalNormal      PValue      TestStatistic    C
    ___________    ________    ________    ___________________    _________    _____________    _

       "S&P"       "Model1"     0.975            reject           0.0054099       -0.38265
       "S&P"       "Model2"     0.975            reject            0.044967       -0.25011
       "S&P"       "Model3"     0.975            accept               0.149       -0.15551
```

Step 7. Run the unconditional *t* test.

Run the individual test for the unconditional *t* test.

```
t = unconditionalT(ebt);
disp(t)
```

```
    PortfolioID     VaRID      VaRLevel    UnconditionalT     PValue     TestStatistic    Critica
    ___________    ________    ________    ______________    ________    _____________    _______

       "S&P"       "Model1"     0.975          reject        0.018566       -0.38265          -0.28
       "S&P"       "Model2"     0.975          accept        0.073292       -0.25011          -0.28
       "S&P"       "Model3"     0.975          accept         0.17932       -0.15551          -0.28
```

Step 8. Run ES backtests for a particular year.

Select a particular calendar year and run the tests for that year only by creating an `esbacktest` object and passing only the data of interest.

```
Year = 1996;
Ind = year(Dates)==Year;
```

```
PortID = ['S&P, ' num2str(Year)];
PortfolioData = Returns(Ind);
VaRData = [VaRModel1(Ind) VaRModel2(Ind) VaRModel3(Ind)];
ESData = [ESModel1(Ind) ESModel2(Ind) ESModel3(Ind)];
ebt = esbacktest(PortfolioData,VaRData,ESData,...
   'PortfolioID',PortID,'VaRID',["Model1","Model2","Model3"],'VaRLevel',VaRLevel);
disp(ebt)
```

```
  esbacktest with properties:

    PortfolioData: [262x1 double]
          VaRData: [262x3 double]
           ESData: [262x3 double]
      PortfolioID: "S&P, 1996"
            VaRID: ["Model1"    "Model2"    "Model3"]
         VaRLevel: [0.9750 0.9750 0.9750]
```

```
tt = runtests(ebt);
disp(tt)
```

```
    PortfolioID     VaRID      VaRLevel    UnconditionalNormal    UnconditionalT
    ___________    ________    ________    ___________________    ______________

    "S&P, 1996"    "Model1"     0.975            reject               reject
    "S&P, 1996"    "Model2"     0.975            reject               reject
    "S&P, 1996"    "Model3"     0.975            reject               accept
```

See Also

esbacktest | summary | runtests | unconditionalNormal | unconditionalT

Related Examples

- “Expected Shortfall (ES) Backtesting Workflow Using Simulation” on page 2-34
- “Expected Shortfall Estimation and Backtesting” on page 2-44

More About

- “Overview of Expected Shortfall Backtesting” on page 2-20

Expected Shortfall (ES) Backtesting Workflow Using Simulation

This example shows an expected shortfall (ES) backtesting workflow using the `esbacktestbysim` object. The tests supported in the `esbacktestbysim` object require as inputs not only the test data (`Portfolio`, `VaR`, and `ES` data), but also the distribution information of the model being tested.

The `esbacktestbysim` class supports five tests -- conditional, unconditional, quantile, which are based on Acerbi-Szekely (2014) and minBiasAbsolute and minBiasRelative, which are based on Acerbi-Szekely (2017 ans 2019). These tests use the distributional assumptions to simulate return scenarios, assuming the distributional assumptions are correct (null hypothesis). The simulated scenarios find the distribution of typical values for the test statistics and the significance of the tests. `esbacktestbysim` supports *normal* and *t* location-scale distributions (with a fixed number of degrees of freedom throughout the test window).

Step 1. Load the ES backtesting data.

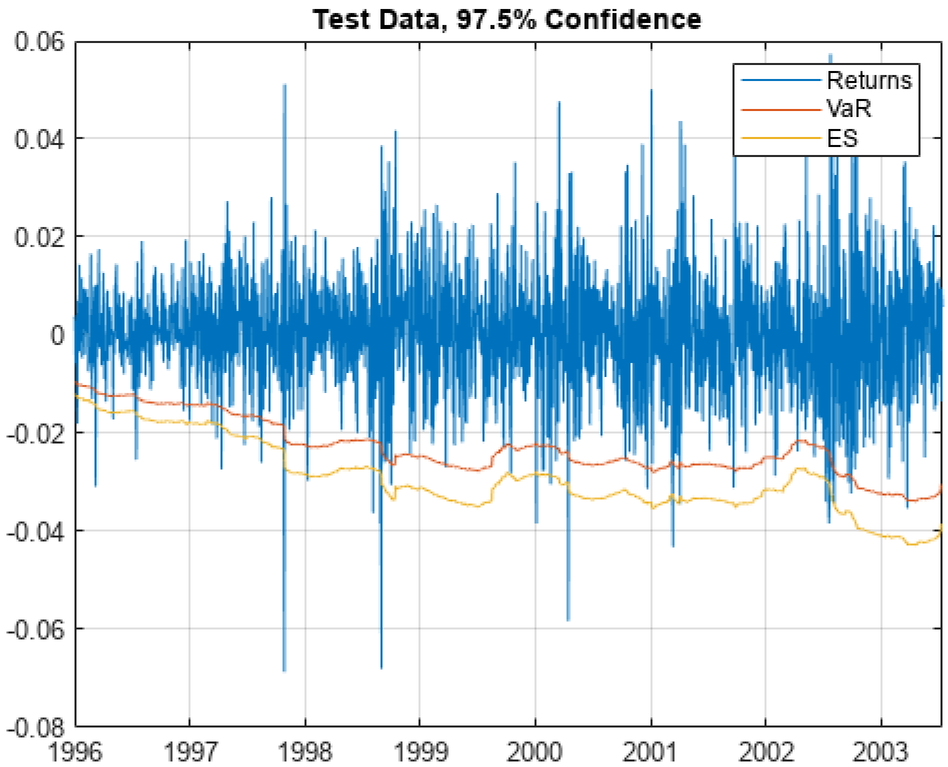
Use the `ESBacktestBySimData.mat` file to load the data into the workspace. This example works with the `Returns` numeric array. This array represents the equity returns. The corresponding VaR data and VaR confidence levels are in `VaR` and `VaRLevel`. The expected shortfall data is contained in `ES`.

```
load ESBacktestBySimData
```

Step 2. Generate an ES backtesting plot.

Use the `plot` function to visualize the ES backtesting data. This type of visualization is a common first step when performing an ES backtesting analysis. This plot displays the returns data against the VaR and ES data.

```
VaRInd = 2;
figure;
plot(Dates,Returns,Dates,-VaR(:,VaRInd),Dates,-ES(:,VaRInd))
legend('Returns','VaR','ES')
title(['Test Data, ' num2str(VaRLevel(VaRInd)*100) '% Confidence'])
grid on
```



Step 3. Create an `esbacktestbysim` object.

Create an `esbacktestbysim` object using `esbacktestbysim`. The `Distribution` information is used to simulate returns to estimate the significance of the tests. The simulation to estimate the significance is run by default when you create the `esbacktestbysim` object. Therefore, the test results are available when you create the object. You can set the optional name-value pair input argument `'Simulate'` to `false` to avoid the simulation, in which case you can use the `simulate` function before querying for test results.

```
rng('default'); % for reproducibility
IDs = ["t(dof) 95%","t(dof) 97.5%","t(dof) 99%"];
IDs = strrep(IDs,"dof",num2str(DoF));
ebts = esbacktestbysim(Returns,VaR,ES,Distribution,...
   'DegreesOfFreedom',DoF,...
   'Location',Mu,...
   'Scale',Sigma,...
   'PortfolioID',"S&P",...
   'VaRID',IDs,...
   'VaRLevel',VaRLevel);
disp(ebts)
```

```
  esbacktestbysim with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x3 double]
           ESData: [1966x3 double]
     Distribution: [1x1 struct]
```

```
      PortfolioID: "S&P"
            VaRID: ["t(10) 95%"    "t(10) 97.5%"    "t(10) 99%"]
         VaRLevel: [0.9500 0.9750 0.9900]
```

```
disp(ebts.Distribution) % distribution information stored in the 'Distribution' property
```

```
                 Name: "t"
    DegreesOfFreedom: 10
            Location: 0
               Scale: [1966x1 double]
```

Step 4. Generate the ES summary report.

The ES summary report provides information about the severity of the violations, that is, how large the loss is compared to the VaR on days when the VaR was violated. The `ObservedSeverity` (or observed average severity ratio) column is the ratio of loss to VaR over days when the VaR is violated. The `ExpectedSeverity` (or expected average severity ratio) column shows the average of the ratio of ES to VaR on the days when the VaR is violated.

```
S = summary(ebts);
disp(S)
```

```
    PortfolioID        VaRID         VaRLevel     ObservedLevel     ExpectedSeverity     ObservedSeve
    ___________    _____________    ________    _____________    ________________    ____________

       "S&P"       "t(10) 95%"        0.95        0.94812           1.3288               1.4515
       "S&P"       "t(10) 97.5%"     0.975        0.97202           1.2652               1.4134
       "S&P"       "t(10) 99%"        0.99        0.98627           1.2169               1.3947
```

Step 5. Run a report for all tests.

Run all tests and generate a report on only the accept or reject results.

```
t = runtests(ebts);
disp(t)
```

```
    PortfolioID        VaRID         VaRLevel    Conditional    Unconditional    Quantile    MinBi
    ___________    _____________    ________    ___________    _____________    ________    _____

       "S&P"       "t(10) 95%"        0.95         reject           accept          reject          a
       "S&P"       "t(10) 97.5%"     0.975         reject           reject          reject          r
       "S&P"       "t(10) 99%"        0.99         reject           reject          reject          r
```

Step 6. Run the conditional test.

Run the individual test for the conditional test (also known as the first Acerbi-Szekely test). The second output (`s`) contains simulated test statistic values, assuming the distributional assumptions are correct. Each row of the `s` output matches the `VaRID` in the corresponding row of the `t` output. Use these simulated statistics to determine the significance of the tests.

```
[t,s] = conditional(ebts);
disp(t)
```

```
    PortfolioID        VaRID         VaRLevel    Conditional    ConditionalOnly    PValue    TestS
    ___________    _____________    ________    ___________    _______________    ______    _____

       "S&P"       "t(10) 95%"        0.95         reject           reject             0       -0
       "S&P"       "t(10) 97.5%"     0.975         reject           reject         0.001        -(
       "S&P"       "t(10) 99%"        0.99         reject           reject         0.003        -(
```

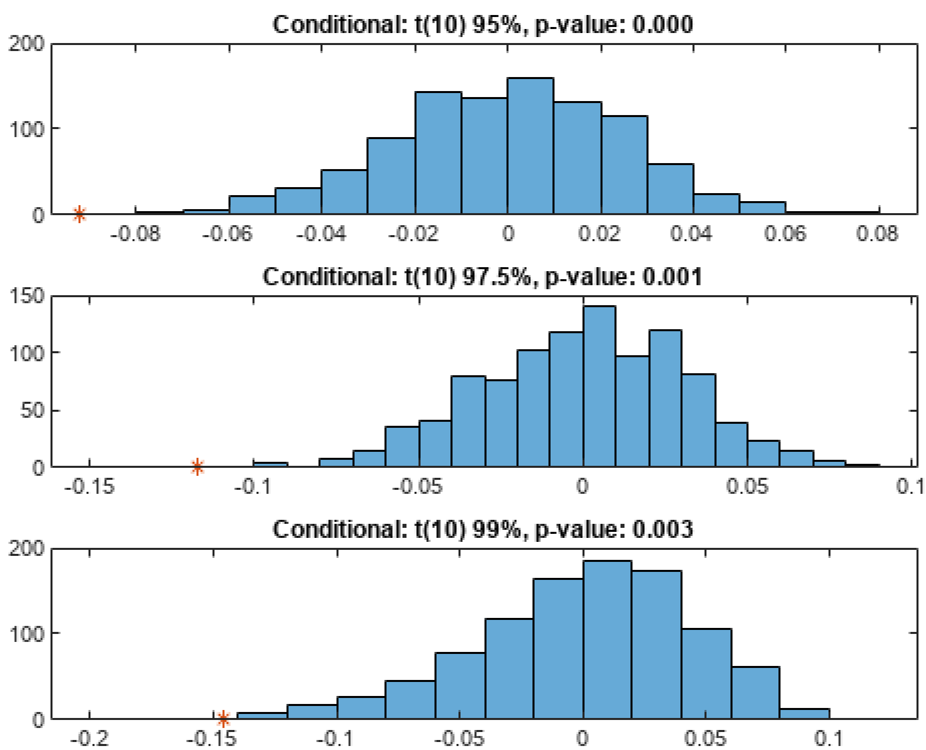
```
whos s
```

```
  Name      Size                 Bytes  Class     Attributes

  s         3x1000               24000  double
```

Step 7. Visualize the significance of the conditional test.

Visualize the significance of the conditional test using histograms to show the distribution of typical values (simulation results). In the histograms, the asterisk shows the value of the test statistic observed for the actual returns. This is a visualization of the standalone conditional test. The final conditional test result also depends on a preliminary VaR backtest, as shown in the conditional test output.

```
NumVaRs = height(t);
figure;
for VaRInd = 1:NumVaRs
   subplot(NumVaRs,1,VaRInd)
   histogram(s(VaRInd,:));
   hold on;
   plot(t.TestStatistic(VaRInd),0,'*');
   hold off;
   Title = sprintf('Conditional: %s, p-value: %4.3f',t.VaRID(VaRInd),t.PValue(VaRInd));
   title(Title)
end
```



Step 8. Run the unconditional test.

Run the individual test for the unconditional test (also known as the second Acerbi-Szekely test).

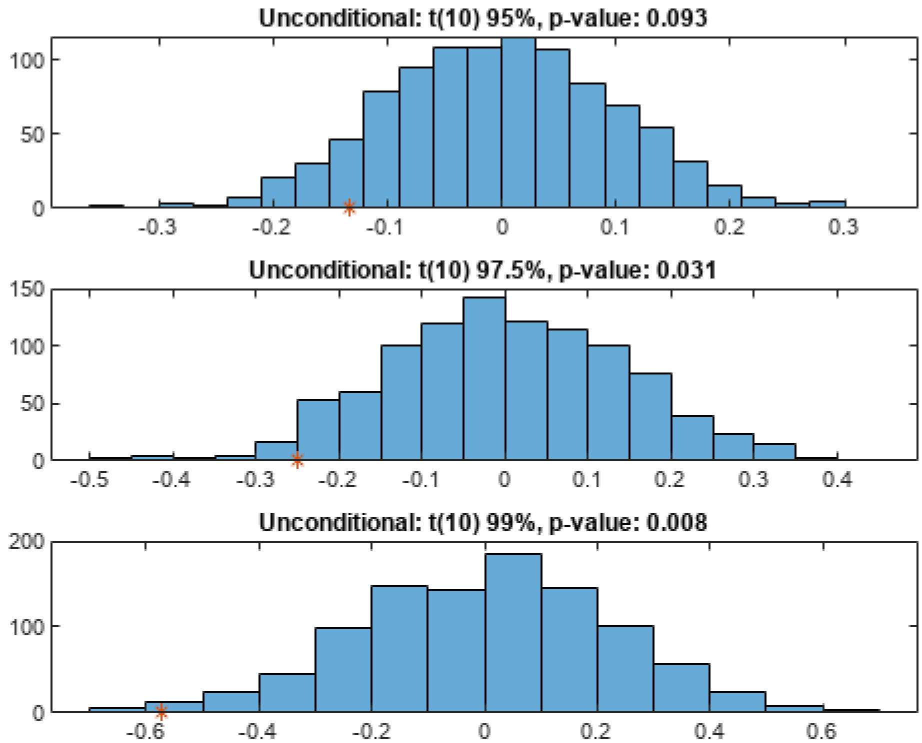
```
[t,s] = unconditional(ebts);
disp(t)
```

```
    PortfolioID        VaRID         VaRLevel    Unconditional    PValue    TestStatistic    Crit
    ___________    ______________    ________    _____________    ______    _____________    ____

       "S&P"       "t(10) 95%"         0.95         accept        0.093       -0.13342        -0
       "S&P"       "t(10) 97.5%"      0.975         reject        0.031       -0.25011         -(
       "S&P"       "t(10) 99%"         0.99         reject        0.008       -0.57396        -0
```

Step 9. Visualize the significance of the unconditional test.

Visualize the significance of the unconditional test using histograms to show the distribution of typical values (simulation results). In the histograms, the asterisk shows the value of the test statistic observed for the actual returns.

```
NumVaRs = height(t);
figure;
for VaRInd = 1:NumVaRs
   subplot(NumVaRs,1,VaRInd)
   histogram(s(VaRInd,:));
   hold on;
   plot(t.TestStatistic(VaRInd),0,'*');
   hold off;
   Title = sprintf('Unconditional: %s, p-value: %4.3f',t.VaRID(VaRInd),t.PValue(VaRInd));
   title(Title)
end
```



Step 10. Run the quantile test.

Run the individual test for the quantile test (also known as the third Acerbi-Szekely test).

```
[t,s] = quantile(ebts);
disp(t)
```

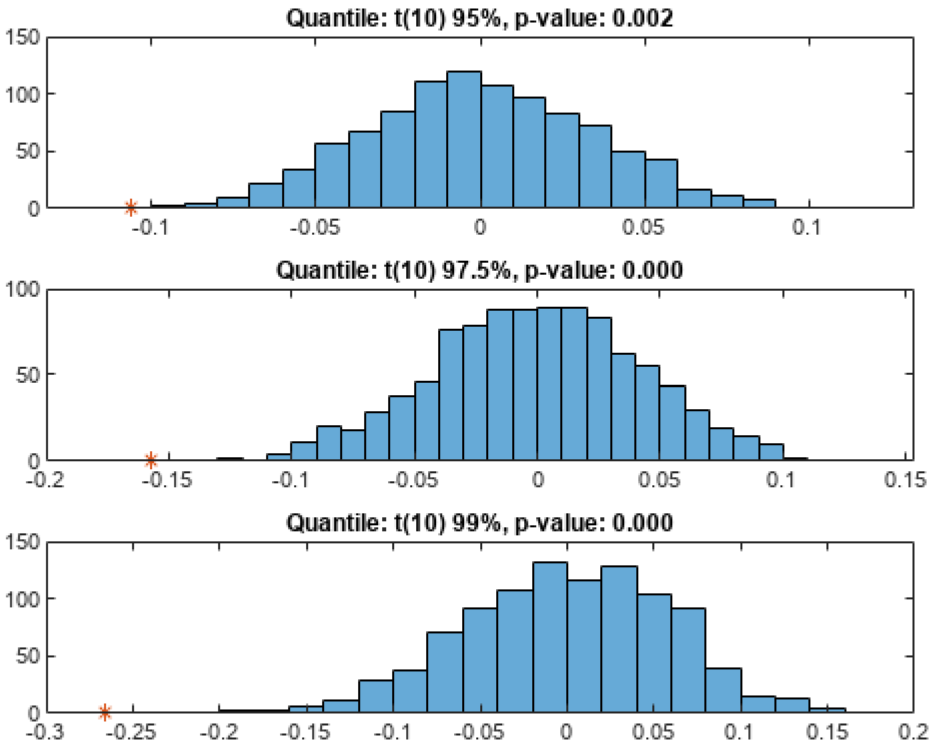
```
    PortfolioID         VaRID          VaRLevel    Quantile    PValue    TestStatistic    CriticalVa
    ___________    _____________    ________    ________    ______    _____________    __________

       "S&P"       "t(10) 95%"        0.95       reject     0.002       -0.10602        -0.0557
       "S&P"       "t(10) 97.5%"     0.975       reject         0       -0.15697        -0.0735
       "S&P"       "t(10) 99%"        0.99       reject         0       -0.26561         -0.101
```

Step 11. Visualize the significance of the quantile test.

Visualize the significance of the quantile test using histograms to show the distribution of typical values (simulation results). In the histograms, the asterisk shows the value of the test statistic observed for the actual returns.

```
NumVaRs = height(t);
figure;
for VaRInd = 1:NumVaRs
   subplot(NumVaRs,1,VaRInd)
   histogram(s(VaRInd,:));
   hold on;
   plot(t.TestStatistic(VaRInd),0,'*');
   hold off;
```

```
   Title = sprintf('Quantile: %s, p-value: %4.3f',t.VaRID(VaRInd),t.PValue(VaRInd));
   title(Title)
end
```



Step 10. Run the minBiasAbsolute test.

Run the individual test for the minBiasAbsolute test.

```
[t,s] = minBiasAbsolute(ebts);
disp(t)
```

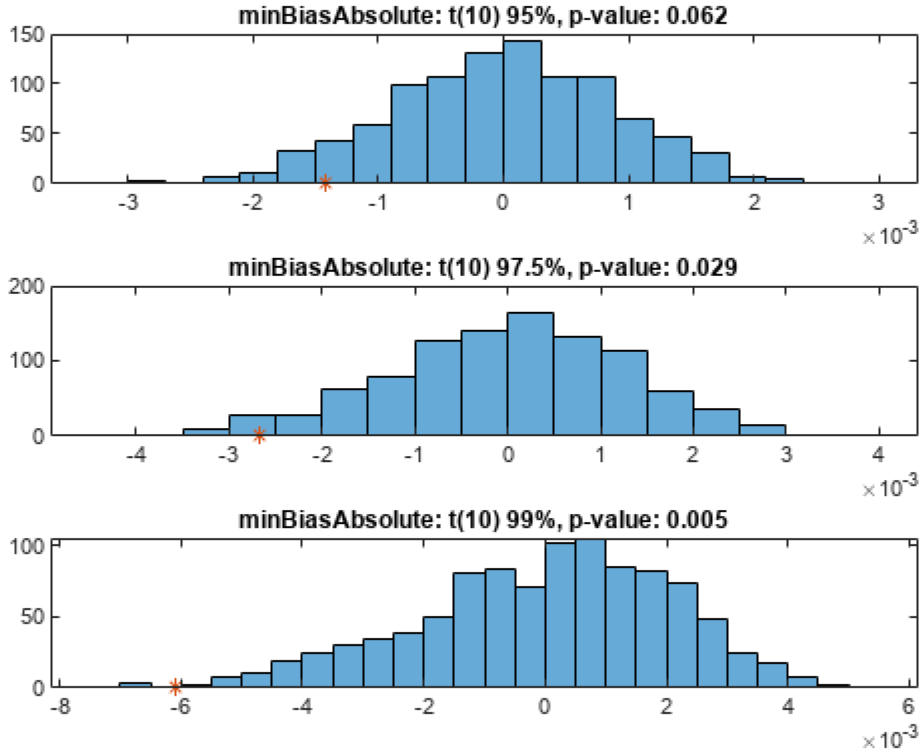
```
    PortfolioID        VaRID         VaRLevel     MinBiasAbsolute     PValue     TestStatistic     Cr
    ___________    _____________    ________    _______________    ______    _____________    __

       "S&P"       "t(10) 95%"         0.95          accept        0.062      -0.0014247       -(
       "S&P"       "t(10) 97.5%"      0.975          reject        0.029      -0.0026674       -(
       "S&P"       "t(10) 99%"         0.99          reject        0.005      -0.0060982       -(
```

Step 11. Visualize the significance of the minBiasAbsolute test.

Visualize the significance of the minBiasAbsolute test using histograms to show the distribution of typical values (simulation results). In the histograms, the asterisk shows the value of the test statistic observed for the actual returns.

```
NumVaRs = height(t);
figure;
for VaRInd = 1:NumVaRs
   subplot(NumVaRs,1,VaRInd)
```

```
    histogram(s(VaRInd,:));
    hold on;
    plot(t.TestStatistic(VaRInd),0,'*');
    hold off;
    Title = sprintf('minBiasAbsolute: %s, p-value: %4.3f',t.VaRID(VaRInd),t.PValue(VaRInd));
    title(Title)
end
```



Step 10. Run the minBiasRelative test.

Run the individual test for the minBiasRelative test.

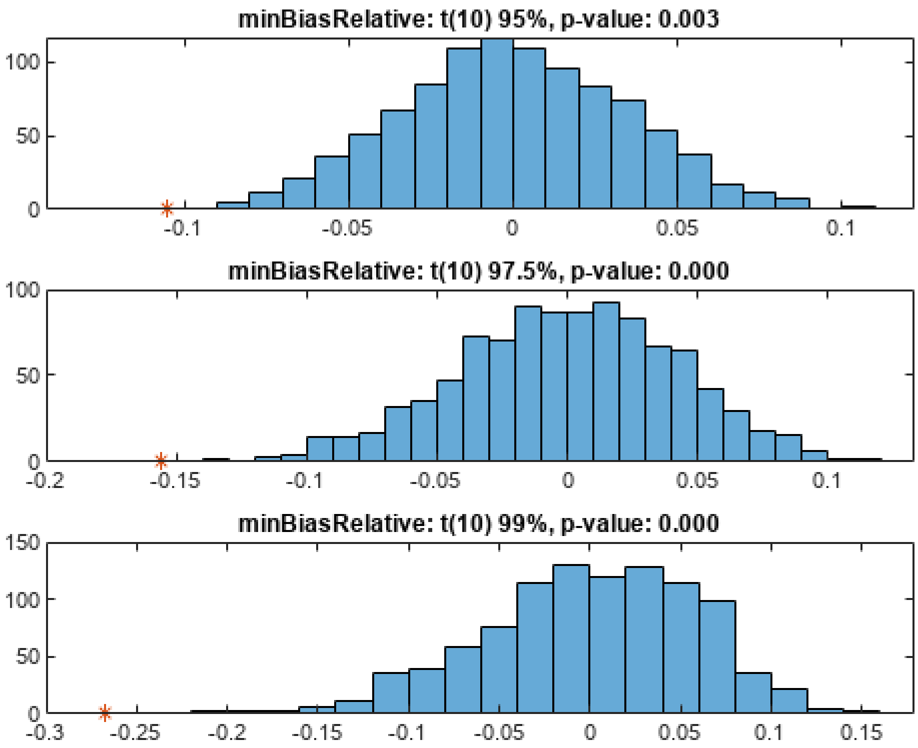
```
[t,s] = minBiasRelative(ebts);
disp(t)
```

```
    PortfolioID        VaRID         VaRLevel    MinBiasRelative    PValue    TestStatistic    Cr
    ___________    _____________    ________    _______________    ______    _____________    __

       "S&P"       "t(10) 95%"        0.95          reject          0.003       -0.10509
       "S&P"       "t(10) 97.5%"     0.975          reject              0       -0.15603
       "S&P"       "t(10) 99%"        0.99          reject              0       -0.26716
```

Step 11. Visualize the significance of the minBiasAbsolute test.

Visualize the significance of the minBiasRelative test using histograms to show the distribution of typical values (simulation results). In the histograms, the asterisk shows the value of the test statistic observed for the actual returns.

```
NumVaRs = height(t);
figure;
for VaRInd = 1:NumVaRs
   subplot(NumVaRs,1,VaRInd)
   histogram(s(VaRInd,:));
   hold on;
   plot(t.TestStatistic(VaRInd),0,'*');
   hold off;
   Title = sprintf('minBiasRelative: %s, p-value: %4.3f',t.VaRID(VaRInd),t.PValue(VaRInd));
   title(Title)
end
```



Step 12. Run a new simulation to estimate the significance of the tests.

Run the simulation again using 5000 scenarios to generate a new set of test results. If the initial test results for one of the tests are borderline, using a larger simulation can help clarify the test results.

```
ebts = simulate(ebts,'NumScenarios',5000);
t = unconditional(ebts);  % new results for unconditional test
disp(t)
```

```
    PortfolioID        VaRID        VaRLevel    Unconditional    PValue    TestStatistic    Crit
    ___________    _____________    ________    _____________    ______    _____________    ____

       "S&P"        "t(10) 95%"       0.95         accept        0.0984      -0.13342        -0
```

```
    "S&P"      "t(10) 97.5%"    0.975        reject       0.0456     -0.25011        -0
    "S&P"      "t(10) 99%"       0.99        reject       0.0104     -0.57396        -0
```

See Also

summary | runtests | conditional | unconditional | quantile | simulate | minBiasRelative | minBiasAbsolute

Related Examples

- “Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30
- “Expected Shortfall Estimation and Backtesting” on page 2-44
- “Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63

More About

- “Overview of Expected Shortfall Backtesting” on page 2-20

Expected Shortfall Estimation and Backtesting

This example shows how to perform estimation and backtesting of Expected Shortfall models.

Value-at-Risk (VaR) and Expected Shortfall (ES) must be estimated together because the ES estimate depends on the VaR estimate. Using historical data, this example estimates VaR and ES over a test window, using historical and parametric VaR approaches. The parametric VaR is calculated under the assumption of normal and *t* distributions.

This example runs the ES back tests supported in the `esbacktest`, `esbacktestbysim`, and `esbacktestbyde` functionality to assess the performance of the ES models in the test window.

The `esbacktest` object does not require any distribution information. Like the `varbacktest` object, the `esbacktest` object only takes test data as input. The inputs to `esbacktest` include portfolio data, VaR data and corresponding VaR level, and also the ES data, since this is what is back tested. Like `varbacktest`, `esbacktest` runs tests for a single portfolio, but can back test multiple models and multiple VaR levels at once. The `esbacktest` object uses precomputed tables of critical values to determine if the models should be rejected. These table-based tests can be applied as approximate tests for any VaR model. In this example, they are applied to back test historical and parametric VaR models. They could be used for other VaR approaches such as Monte-Carlo or Extreme-Value models.

In contrast, the `esbacktestbysim` and `esbacktestbyde` objects require the distribution information, namely, the distribution name (`normal` or *t*) and the distribution parameters for each day in the test window. `esbacktestbysim` and `esbacktestbyde` can only back test one model at a time because they are linked to a particular distribution, although you can still back test multiple VaR levels at once. The `esbacktestbysim` object implements simulation-based tests and it uses the provided distribution information to run simulations to determine critical values. The `esbacktestbyde` object implements tests where the critical values are derived from either a large-sample approximation or a simulation (finite sample). The `conditionalDE` test in the `esbacktestbyde` object tests for independence over time, to assess if there is evidence of autocorrelation in the series of tail losses. All other tests are severity tests to assess if the magnitude of the tail losses is consistent with the model predictions. Both the `esbacktestbysim` and `esbacktestbyde` objects support normal and *t* distributions. These tests can be used for any model where the underlying distribution of portfolio outcomes is normal or *t*, such as exponentially weighted moving average (EWMA), delta-gamma, or generalized autoregressive conditional heteroskedasticity (GARCH) models.

For additional information on the ES backtesting methodology, see `esbacktest`, `esbacktestbysim`, and `esbacktestbyde`, also see [1 on page 2-61], [2 on page 2-61], [3 on page 2-61] and [5 on page 2-61] in the References.

Estimate VaR and ES

The data set used in this example contains historical data for the S&P index spanning approximately 10 years, from the middle of 1993 through the middle of 2003. The estimation window size is defined as 250 days, so that a full year of data is used to estimate both the historical VaR, and the volatility. The test window in this example runs from the beginning of 1995 through the end of 2002.

Throughout this example, a VaR confidence level of 97.5% is used, as required by the Fundamental Review of the Trading Book (FRTB) regulation; see [4 on page 2-61].

```
load VaRExampleData.mat
Returns = tick2ret(sp);
```

```
DateReturns = dates(2:end);
SampleSize = length(Returns);

TestWindowStart = find(year(DateReturns)==1995,1);
TestWindowEnd = find(year(DateReturns)==2002,1,'last');
TestWindow = TestWindowStart:TestWindowEnd;
EstimationWindowSize = 250;

DatesTest = DateReturns(TestWindow);
ReturnsTest = Returns(TestWindow);

VaRLevel = 0.975;
```

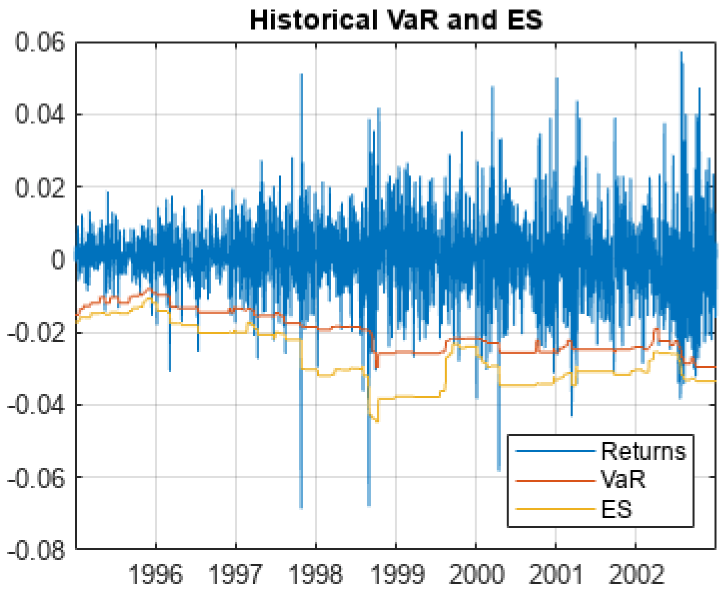
The historical VaR is a non-parametric approach to estimate the VaR and ES from historical data over an estimation window. The VaR is a percentile, and there are alternative ways to estimate the percentile of a distribution based on a finite sample. One common approach is to use the `prctile` function. An alternative approach is to sort the data and determine a cut point based on the sample size and VaR confidence level. Similarly, there are alternative approaches to estimate the ES based on a finite sample.

The `hHistoricalVaRES` local function on the bottom of this example uses a finite-sample approach for the estimation of VaR and ES following the methodology described in [7 on page 2-61]. In a finite sample, the number of observations below the VaR may not match the total tail probability corresponding to the VaR level. For example, for 100 observations and a VaR level of 97.5%, the tail observations are 2, which is 2% of the sample, however the desired tail probability is 2.5%. It could be even worse for samples with repeated observed values, for example, if the second and third sorted values were the same, both equal to the VaR, then only the smallest observed value in the sample would have a value less than the VaR, and that is 1% of the sample, not the desired 2.5%. The method implemented in `hHistoricalVaRES` makes a correction so that the tail probability is always consistent with the VaR level; see [7 on page 2-61] for details.

```
VaR_Hist = zeros(length(TestWindow),1);
ES_Hist = zeros(length(TestWindow),1);

for t = TestWindow

   i = t - TestWindowStart + 1;
   EstimationWindow = t-EstimationWindowSize:t-1;

   [VaR_Hist(i),ES_Hist(i)] = hHistoricalVaRES(Returns(EstimationWindow),VaRLevel);

end
```

The following plot shows the daily returns, and the VaR and ES estimated with the historical method.

```
figure;
plot(DatesTest,ReturnsTest,DatesTest,-VaR_Hist,DatesTest,-ES_Hist)
legend('Returns','VaR','ES','Location','southeast')
title('Historical VaR and ES')
grid on
```



For the parametric models, the volatility of the returns must be computed. Given the volatility, the VaR, and ES can be computed analytically.

A zero mean is assumed in this example, but can be estimated in a similar way.

For the normal distribution, the estimated volatility is used directly to get the VaR and ES. For the *t* location-scale distribution, the scale parameter is computed from the estimated volatility and the degrees of freedom.

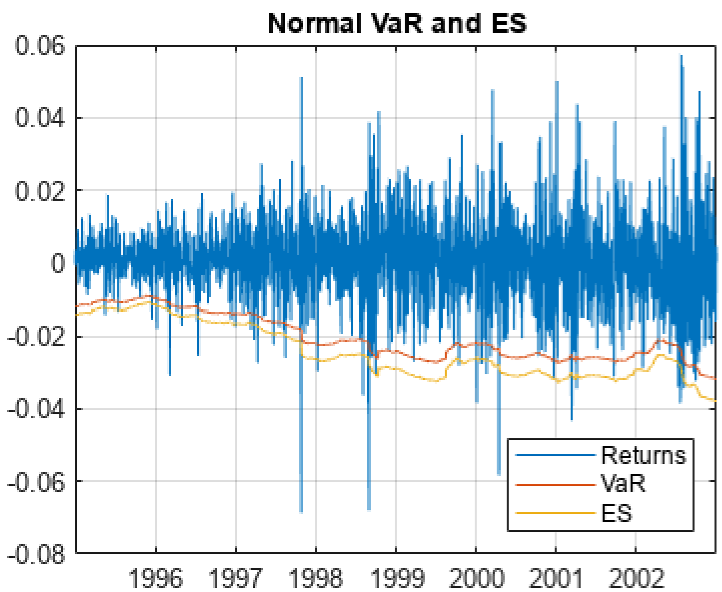
The `hNormalVaRES` and `hTVaRES` local functions take as inputs the distribution parameters (which can be passed as arrays), and return the VaR and ES. These local functions use the analytical expressions for VaR and ES for normal and *t* location-scale distributions, respectively; see [6 on page 2-61] for details.

```
% Estimate volatility over the test window
Volatility = zeros(length(TestWindow),1);

for t = TestWindow

   i = t - TestWindowStart + 1;
   EstimationWindow = t-EstimationWindowSize:t-1;

   Volatility(i) = std(Returns(EstimationWindow));

end

% Mu=0 in this example
Mu = 0;

% Sigma (standard deviation parameter) for normal distribution = Volatility
SigmaNormal = Volatility;
% Sigma (scale parameter) for t distribution = Volatility * sqrt((DoF-2)/DoF)
SigmaT10 = Volatility*sqrt((10-2)/10);
SigmaT5 = Volatility*sqrt((5-2)/5);
```

```
% Estimate VaR and ES, normal
[VaR_Normal,ES_Normal] = hNormalVaRES(Mu,SigmaNormal,VaRLevel);
% Estimate VaR and ES, t with 10 and 5 degrees of freedom
[VaR_T10,ES_T10] = hTVaRES(10,Mu,SigmaT10,VaRLevel);
[VaR_T5,ES_T5] = hTVaRES(5,Mu,SigmaT5,VaRLevel);
```

The following plot shows the daily returns, and the VaR and ES estimated with the normal method.

```
figure;
plot(DatesTest,ReturnsTest,DatesTest,-VaR_Normal,DatesTest,-ES_Normal)
legend('Returns','VaR','ES','Location','southeast')
title('Normal VaR and ES')
grid on
```



For the parametric approach, the same steps can be used to estimate the VaR and ES for alternative approaches, such as EWMA, delta-gamma approximations, and GARCH models. In all these parametric approaches, a volatility is estimated every day, either from an EWMA update, from a delta-gamma approximation, or as the conditional volatility of a GARCH model. The volatility can then be used as above to get the VaR and ES estimates for either normal or *t* location-scale distributions.

ES Backtest Without Distribution Information

The `esbacktest` object offers two back tests for ES models. Both tests use the unconditional test statistic proposed by Acerbi and Szekely in [1 on page 2-61], given by

$$Z_{\text{uncond}} = \frac{1}{\text{Np}_{\text{VaR}}}\sum_{t=1}^{N}\frac{X_tI_t}{\text{ES}_t} + 1$$

where

- N is the number of time periods in the test window.
- X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period t.

- p_{VaR} is the probability of VaR failure defined as 1-VaR level.
- ES_t is the estimated expected shortfall for period t.
- I_t is the VaR failure indicator on period t with a value of 1 if $X_t < -\mathrm{VaR}_t$, and 0 otherwise.

The expected value for this test statistic is 0, and it is negative when there is evidence of risk underestimation. To determine how negative it should be to reject the model, critical values are needed, and to determine critical values, distributional assumptions are needed for the portfolio outcomes X_t.

The unconditional test statistic turns out to be stable across a range of distributional assumptions for X_t, from thin-tailed distributions such as normal, to heavy-tailed distributions such as t with low degrees of freedom (high single digits). Only the most heavy-tailed t distributions (low single digits) lead to more noticeable differences in the critical values. See [1 on page 2-61] for details.

The `esbacktest` object takes advantage of the stability of the critical values of the unconditional test statistic and uses tables of precomputed critical values to run ES back tests. `esbacktest` has two sets of critical-value tables. The first set of critical values assumes that the portfolio outcomes X_t follow a standard normal distribution; this is the `unconditionalNormal` test. The second set of critical values uses the heaviest possible tails, it assumes that the portfolio outcomes X_t follow a t distribution with 3 degrees of freedom; this is the `unconditionalT` test.

The unconditional test statistic is sensitive to both the severity of the VaR failures relative to the ES estimate, and also to the number of VaR failures (how many times the VaR is violated). Therefore, a single but very large VaR failure relative to the ES (or only very few large losses) may cause the rejection of a model in a particular time window. A large loss on a day when the ES estimate is also large may not impact the test results as much as a large loss when the ES is smaller. And a model can also be rejected in periods with many VaR failures, even if all the VaR violations are relatively small and only slightly higher than the VaR. Both situations are illustrated in this example.

The `esbacktest` object takes as input the test data, but no distribution information is provided to `esbacktest`. Optionally, you can specify ID's for the portfolio, and for each of the VaR and ES models being backtested. Although the model ID's in this example do have distribution references (for example, `"normal"` or `"t 10"`), these are only labels used for reporting purposes. The tests do not use the fact that the first model is a historical VaR method, or that the other models are alternative parametric VaR models. The distribution parameters used to estimate the VaR and ES in the previous section are not passed to `esbacktest`, and are not used in any way in this section. These parameters, however, must be provided for the simulation-based tests supported in the `esbacktestbysim` object discussed in the Simulation-Based Tests on page 2-55 section, and for the tests supported in the `esbacktestbyde` object discussed in the Large-Sample and Simulation Tests on page 2-58 section.

```
ebt = esbacktest(ReturnsTest,[VaR_Hist VaR_Normal VaR_T10 VaR_T5],...
   [ES_Hist ES_Normal ES_T10 ES_T5],'PortfolioID',"S&P, 1995-2002",...
   'VaRID',["Historical" "Normal","T 10","T 5"],'VaRLevel',VaRLevel);
disp(ebt)
```

```
  esbacktest with properties:

    PortfolioData: [2087x1 double]
          VaRData: [2087x4 double]
           ESData: [2087x4 double]
      PortfolioID: "S&P, 1995-2002"
            VaRID: ["Historical"    "Normal"    "T 10"    "T 5"]
         VaRLevel: [0.9750 0.9750 0.9750 0.9750]
```

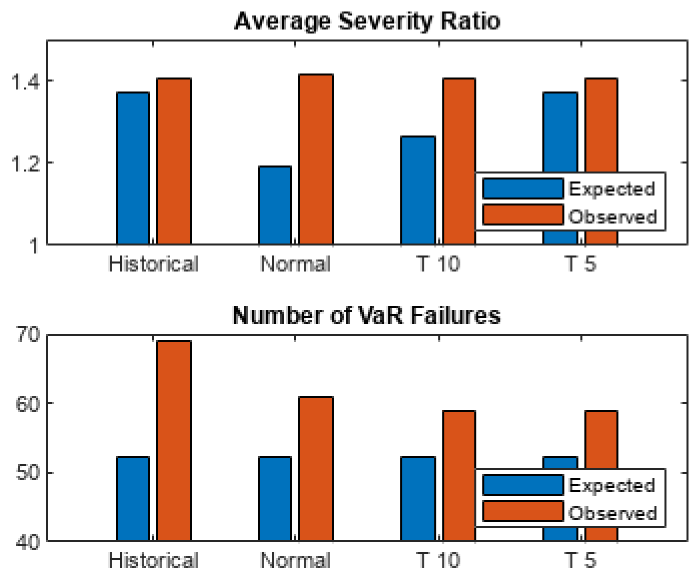
Start the analysis by running the `summary` function.

```
s = summary(ebt);
disp(s)
```

```
    PortfolioID             VaRID          VaRLevel     ObservedLevel     ExpectedSeverity     Observe
_________________      ____________      ________     _____________     ________________     _______

"S&P, 1995-2002"       "Historical"       0.975          0.96694            1.3711              1.4
"S&P, 1995-2002"       "Normal"           0.975          0.97077            1.1928              1
"S&P, 1995-2002"       "T 10"             0.975          0.97173            1.2652              1.4
"S&P, 1995-2002"       "T 5"              0.975          0.97173              1.37              1.4
```

The `ObservedSeverity` column shows the average ratio of loss to VaR on periods when the VaR was violated. The `ExpectedSeverity` column uses the average ratio of ES to VaR for the VaR violation periods. For the `"Historical"` and `"T 5"` models, the observed and expected severities are comparable. However, for the `"Historical"` method, the observed number of failures (`Failures` column) is considerably higher than the expected number of failures (`Expected` column), about 32% higher (see the `Ratio` column). Both the `"Normal"` and the `"T 10"` models have observed severities much higher than the expected severities.

```
figure;

subplot(2,1,1)
bar(categorical(s.VaRID),[s.ExpectedSeverity,s.ObservedSeverity])
ylim([1 1.5])
legend('Expected','Observed','Location','southeast')
title('Average Severity Ratio')

subplot(2,1,2)
bar(categorical(s.VaRID),[s.Expected,s.Failures])
ylim([40 70])
legend('Expected','Observed','Location','southeast')
title('Number of VaR Failures')
```



The `runtests` function runs all tests and reports only the accept or reject result. The unconditional normal test is more strict. For the 8-year test window here, two models fail both tests (`"Historical"` and `"Normal"`), one model fails the unconditional normal test, but passes the unconditional t test (`"T 10"`), and one model passes both tests (`"T 5"`).

```
t = runtests(ebt);
disp(t)
```

```
      PortfolioID           VaRID         VaRLevel     UnconditionalNormal     UnconditionalT
    ________________    _____________    ________    ___________________    ______________

    "S&P, 1995-2002"    "Historical"      0.975            reject                 reject
    "S&P, 1995-2002"    "Normal"          0.975            reject                 reject
    "S&P, 1995-2002"    "T 10"            0.975            reject                 accept
    "S&P, 1995-2002"    "T 5"             0.975            accept                 accept
```

Additional details on the tests can be obtained by calling the individual test functions. Here are the details for the `unconditionalNormal` test.

```
t = unconditionalNormal(ebt);
disp(t)
```

```
      PortfolioID           VaRID         VaRLevel     UnconditionalNormal      PValue      TestStati
    ________________    _____________    ________    ___________________    _________    _________

    "S&P, 1995-2002"    "Historical"      0.975            reject            0.0047612     -0.3791
    "S&P, 1995-2002"    "Normal"          0.975            reject            0.0043287     -0.3879
    "S&P, 1995-2002"    "T 10"            0.975            reject             0.037528      -0.256
    "S&P, 1995-2002"    "T 5"             0.975            accept              0.13069     -0.1617
```

Here are the details for the `unconditionalT` test.

```
t = unconditionalT(ebt);
disp(t)
```

```
      PortfolioID           VaRID         VaRLevel     UnconditionalT     PValue      TestStatistic
    ________________    _____________    ________    ______________    ________    _____________

    "S&P, 1995-2002"    "Historical"      0.975           reject         0.017032      -0.37917
    "S&P, 1995-2002"    "Normal"          0.975           reject         0.015375      -0.38798
    "S&P, 1995-2002"    "T 10"            0.975           accept         0.062835       -0.2569
    "S&P, 1995-2002"    "T 5"             0.975           accept          0.16414      -0.16179
```

Using the Tests for More Advanced Analyses

This section shows how to use the `esbacktest` object to run user-defined traffic-light tests, and also how to run tests over rolling test windows.

One way to define a traffic-light test is by combining the results from the unconditional normal and the unconditional *t* tests. Because the unconditional normal is more strict, one can define a traffic-light test with these levels:

- Green: The model passes both the unconditional normal and unconditional *t* tests.
- Yellow: The model fails the unconditional normal test, but passes the unconditional *t* test.
- Red: The model is rejected by both the unconditional normal and unconditional *t* tests.

```
t = runtests(ebt);
TLValue = (t.UnconditionalNormal=='reject')+(t.UnconditionalT=='reject');
```

```
t.TrafficLight = categorical(TLValue,0:2,{'green','yellow','red'},'Ordinal',true);
disp(t)
```

```
      PortfolioID           VaRID         VaRLevel     UnconditionalNormal     UnconditionalT     Tra
    __________________    ____________    ________    ___________________    ______________    ___

    "S&P, 1995-2002"    "Historical"     0.975            reject                 reject          r
    "S&P, 1995-2002"    "Normal"         0.975            reject                 reject          r
    "S&P, 1995-2002"    "T 10"           0.975            reject                 accept          y
    "S&P, 1995-2002"    "T 5"            0.975            accept                 accept          g
```

An alternative user-defined traffic-light test can use a single test, but at different test confidence levels:

- Green: The result is to `'accept'` with a test level of 95%.
- Yellow: The result is to `'reject'` at a 95% test level, but `'accept'` at 99%.
- Red: The result is `'reject'` at 99% test level.

A similar test is proposed in [1 on page 2-61] with a high test level of 99.99%.

```
t95 = runtests(ebt); % 95% is the default test level value
t99 = runtests(ebt,'TestLevel',0.99);
TLValue = (t95.UnconditionalNormal=='reject')+(t99.UnconditionalNormal=='reject');
tRolling = t95(:,1:3);
tRolling.UnconditionalNormal95 = t95.UnconditionalNormal;
tRolling.UnconditionalNormal99 = t99.UnconditionalNormal;
tRolling.TrafficLight = categorical(TLValue,0:2,{'green','yellow','red'},'Ordinal',true);
disp(tRolling)
```

```
      PortfolioID           VaRID         VaRLevel     UnconditionalNormal95     UnconditionalNormal9
    __________________    ____________    ________    _____________________    ____________________

    "S&P, 1995-2002"    "Historical"     0.975             reject                         reject
    "S&P, 1995-2002"    "Normal"         0.975             reject                         reject
    "S&P, 1995-2002"    "T 10"           0.975             reject                         accept
    "S&P, 1995-2002"    "T 5"            0.975             accept                         accept
```

The test results may be different over different test windows. Here, a one-year rolling window is used to run the ES back tests over the eight individual years spanned by the original test window. The first user-defined traffic-light described above is added to the test results table. The `summary` function is also called for each individual year to view the history of the severity and the number of VaR failures.

```
sRolling = table;
tRolling = table;
for Year = 1995:2002
   Ind = year(DatesTest)==Year;
   PortID = ['S&P, ' num2str(Year)];
   PortfolioData = ReturnsTest(Ind);
   VaRData = [VaR_Hist(Ind) VaR_Normal(Ind) VaR_T10(Ind) VaR_T5(Ind)];
   ESData = [ES_Hist(Ind) ES_Normal(Ind) ES_T10(Ind) ES_T5(Ind)];
   ebt = esbacktest(PortfolioData,VaRData,ESData,...
      'PortfolioID',PortID,'VaRID',["Historical" "Normal" "T 10" "T 5"],...
      'VaRLevel',VaRLevel);
   if Year == 1995
      sRolling = summary(ebt);
      tRolling = runtests(ebt);
   else
```

```
        sRolling = [sRolling;summary(ebt)]; %#ok<AGROW>
        tRolling = [tRolling;runtests(ebt)]; %#ok<AGROW>
    end
end

% Optional: Add the first user-defined traffic light test described above
TLValue = (tRolling.UnconditionalNormal=='reject')+(tRolling.UnconditionalT=='reject');
tRolling.TrafficLight = categorical(TLValue,0:2,{'green','yellow','red'},'Ordinal',true);
```

Display the results, one model at a time. The "T 5" model has the best performance in these tests (two "yellow"), and the the "Normal" model the worst (three "red" and one "yellow").

```
disp(tRolling(tRolling.VaRID=="Historical",:))
```

```
    PortfolioID       VaRID         VaRLevel    UnconditionalNormal    UnconditionalT    TrafficLi
    ___________    ____________    ________    ___________________    ______________    _________

    "S&P, 1995"    "Historical"     0.975            accept               accept           green
    "S&P, 1996"    "Historical"     0.975            reject               accept           yellow
    "S&P, 1997"    "Historical"     0.975            reject               reject           red
    "S&P, 1998"    "Historical"     0.975            accept               accept           green
    "S&P, 1999"    "Historical"     0.975            accept               accept           green
    "S&P, 2000"    "Historical"     0.975            accept               accept           green
    "S&P, 2001"    "Historical"     0.975            accept               accept           green
    "S&P, 2002"    "Historical"     0.975            reject               reject           red
```

```
disp(tRolling(tRolling.VaRID=="Normal",:))
```

```
    PortfolioID      VaRID      VaRLevel    UnconditionalNormal    UnconditionalT    TrafficLight
    ___________    ________    ________    ___________________    ______________    ____________

    "S&P, 1995"    "Normal"     0.975            accept               accept            green
    "S&P, 1996"    "Normal"     0.975            reject               reject            red
    "S&P, 1997"    "Normal"     0.975            reject               reject            red
    "S&P, 1998"    "Normal"     0.975            reject               accept            yellow
    "S&P, 1999"    "Normal"     0.975            accept               accept            green
    "S&P, 2000"    "Normal"     0.975            accept               accept            green
    "S&P, 2001"    "Normal"     0.975            accept               accept            green
    "S&P, 2002"    "Normal"     0.975            reject               reject            red
```

```
disp(tRolling(tRolling.VaRID=="T 10",:))
```

```
    PortfolioID     VaRID     VaRLevel    UnconditionalNormal    UnconditionalT    TrafficLight
    ___________    ______    ________    ___________________    ______________    ____________

    "S&P, 1995"    "T 10"     0.975            accept               accept            green
    "S&P, 1996"    "T 10"     0.975            reject               reject            red
    "S&P, 1997"    "T 10"     0.975            reject               accept            yellow
    "S&P, 1998"    "T 10"     0.975            accept               accept            green
    "S&P, 1999"    "T 10"     0.975            accept               accept            green
    "S&P, 2000"    "T 10"     0.975            accept               accept            green
    "S&P, 2001"    "T 10"     0.975            accept               accept            green
    "S&P, 2002"    "T 10"     0.975            reject               reject            red
```

```
disp(tRolling(tRolling.VaRID=="T 5",:))
```

```
    PortfolioID    VaRID    VaRLevel    UnconditionalNormal    UnconditionalT    TrafficLight
    ___________    _____    ________    ___________________    ______________    ____________
```

```
    "S&P, 1995"    "T 5"     0.975            accept              accept           green
    "S&P, 1996"    "T 5"     0.975            reject              accept           yellow
    "S&P, 1997"    "T 5"     0.975            accept              accept           green
    "S&P, 1998"    "T 5"     0.975            accept              accept           green
    "S&P, 1999"    "T 5"     0.975            accept              accept           green
    "S&P, 2000"    "T 5"     0.975            accept              accept           green
    "S&P, 2001"    "T 5"     0.975            accept              accept           green
    "S&P, 2002"    "T 5"     0.975            reject              accept           yellow
```

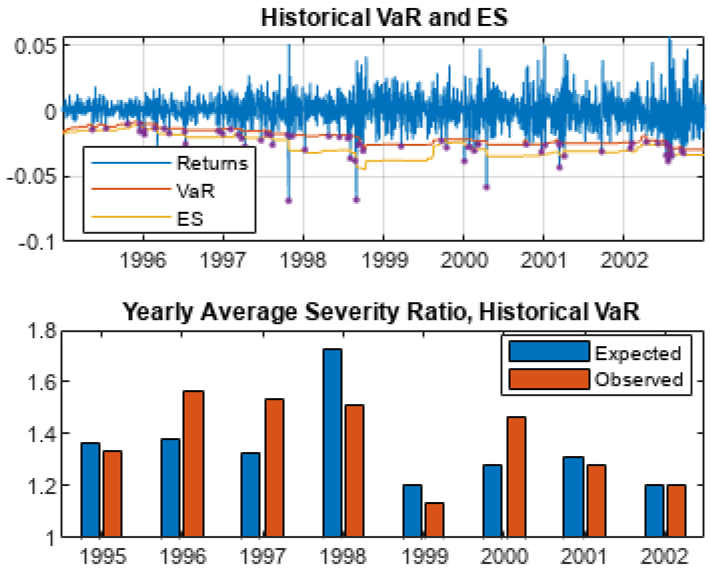
The year 2002 is an example of a year with relatively small severities, yet many VaR failures. All models perform poorly in 2002, even though the observed severities are low. However, the number of VaR failures for some models is more than twice the expected number of VaR failures.

```
disp(summary(ebt))
```

```
    PortfolioID        VaRID        VaRLevel     ObservedLevel     ExpectedSeverity     ObservedSeve
    ___________    ____________    ________    _____________    ________________    ____________

    "S&P, 2002"    "Historical"     0.975         0.94636             1.2022                 1.2
    "S&P, 2002"    "Normal"         0.975         0.94636             1.1928              1.2111
    "S&P, 2002"    "T 10"           0.975         0.95019             1.2652              1.2066
    "S&P, 2002"    "T 5"            0.975         0.95019               1.37              1.2077
```

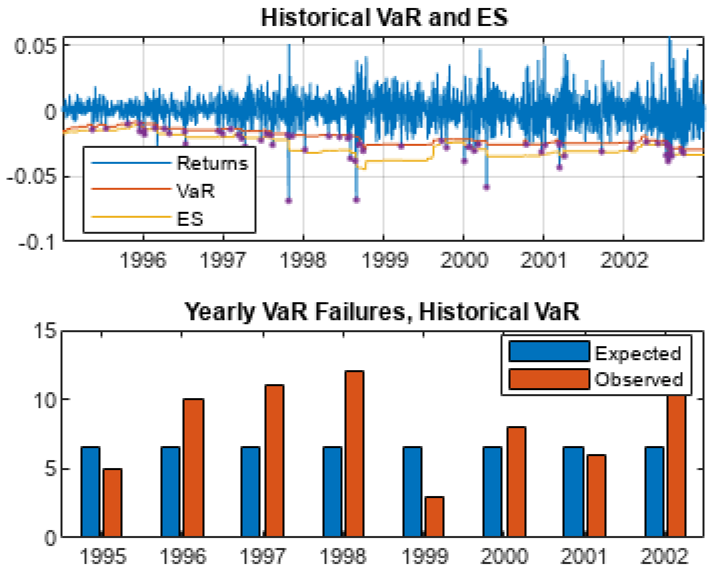
The following figure shows the data on the entire 8-year window, and severity ratio year by year (expected and observed) for the `"Historical"` model. The absolute size of the losses is not as important as the relative size compared to the ES (or equivalently, compared to the VaR). Both 1997 and 1998 have large losses, comparable in magnitude. However the expected severity in 1998 is much higher (larger ES estimates). Overall, the `"Historical"` method seems to do well with respect to severity ratios.

```
sH = sRolling(sRolling.VaRID=="Historical",:);

figure;

subplot(2,1,1)
FailureInd = ReturnsTest<-VaR_Hist;
plot(DatesTest,ReturnsTest,DatesTest,-VaR_Hist,DatesTest,-ES_Hist)
hold on
plot(DatesTest(FailureInd),ReturnsTest(FailureInd),'.')
hold off
legend('Returns','VaR','ES','Location','best')
title('Historical VaR and ES')
grid on

subplot(2,1,2)
bar(1995:2002,[sH.ExpectedSeverity,sH.ObservedSeverity])
ylim([1 1.8])
legend('Expected','Observed','Location','best')
title('Yearly Average Severity Ratio, Historical VaR')
```



However, a similar visualization with the expected against observed number of VaR failures shows that the `"Historical"` method tends to get violated many more times than expected. For example, even though in 2002 the expected average severity ratio is very close to the observed one, the number of VaR failures was more than twice the expected number. This then leads to test failures for both the unconditional normal and unconditional *t* tests.

```
figure;

subplot(2,1,1)
plot(DatesTest,ReturnsTest,DatesTest,-VaR_Hist,DatesTest,-ES_Hist)
hold on
plot(DatesTest(FailureInd),ReturnsTest(FailureInd),'.')
hold off
legend('Returns','VaR','ES','Location','best')
title('Historical VaR and ES')
grid on

subplot(2,1,2)
bar(1995:2002,[sH.Expected,sH.Failures])
legend('Expected','Observed','Location','best')
title('Yearly VaR Failures, Historical VaR')
```



Simulation-Based Tests

The `esbacktestbysim` object supports five simulation-based ES back tests. `esbacktestbysim` requires the distribution information for the portfolio outcomes, namely, the distribution name (`"normal"` or `"t"`) and the distribution parameters for each day in the test window. `esbacktestbysim` uses the provided distribution information to run simulations to determine critical values. The tests supported in `esbacktestbysim` are `conditional`, `unconditional`, `quantile`, `minBiasAbsolute`, and `minBiasRelative`. These are implementations of the tests proposed by Acerbi and Szekely in [1 on page 2-61], and [2 on page 2-61], [3 on page 2-61] for 2017 and 2019.

The `esbacktestbysim` object supports normal and *t* distributions. These tests can be used for any model where the underlying distribution of portfolio outcomes is normal or *t*, such as exponentially weighted moving average (EWMA), delta-gamma, or generalized autoregressive conditional heteroskedasticity (GARCH) models.

ES backtests are necessarily approximated in that they are sensitive to errors in the predicted VaR. However, the minimally biased test has only a small sensitivity to VaR errors and the sensitivity is prudential, in the sense that VaR errors lead to a more punitive ES test. See Acerbi-Szekely ([2 on page 2-61], [3 on page 2-61] for 2017 and 2019) for details. When distribution information is available, the minimally biased test is recommended (see `minBiasAbsolute`, `minBiasRelative`).

The `"Normal"`, `"T 10"`, and `"T 5"` models can be backtested with the simulation-based tests in `esbacktestbysim`. For illustration purposes, only `"T 5"` is backtested. The distribution name (`"t"`) and parameters (degrees of freedom, location, and scale) are provided when the `esbacktestbysim` object is created.

```
rng('default'); % for reproducibility; the esbacktestbysim constructor runs a simulation
ebts = esbacktestbysim(ReturnsTest,VaR_T5,ES_T5,"t",'DegreesOfFreedom',5,...
   'Location',Mu,'Scale',SigmaT5,...
   'PortfolioID',"S&P",'VaRID',"T 5",'VaRLevel',VaRLevel);
```

The recommended workflow is the same: first, run the `summary` function, then run the `runtests` function, and then run the individual test functions.

The `summary` function provides exactly the same information as the `summary` function from `esbacktest`.

```
s = summary(ebts);
disp(s)
```

```
    PortfolioID    VaRID    VaRLevel    ObservedLevel    ExpectedSeverity    ObservedSeverity
    ___________    _____    ________    _____________    ________________    ________________

       "S&P"       "T 5"     0.975         0.97173             1.37                1.4075
```

The `runtests` function shows the final accept or reject result.

```
t = runtests(ebts);
disp(t)
```

```
    PortfolioID    VaRID    VaRLevel    Conditional    Unconditional    Quantile    MinBiasAbsolu
    ___________    _____    ________    ___________    _____________    ________    _____________

       "S&P"       "T 5"     0.975        accept          accept         accept         accept
```

Additional details on the test results are obtained by calling the individual test functions. For example, call the `minBiasAbsolute` test. The first output, `t`, has the test results and additional details such as the p-value, test statistic, and so on. The second output, `s`, contains simulated test statistic values assuming the distributional assumptions are correct. For example, `esbacktestbysim` generated 1000 scenarios of portfolio outcomes in this case, where each scenario is a series of 2087 observations simulated from *t* random variables with 5 degrees of freedom and the given location and scale parameters. The simulated values returned in the optional `s` output show typical values of the test statistic if the distributional assumptions are correct. These are the simulated statistics used to determine the significance of the tests, that is, the reported critical values and *p*-values.

```
[t,s] = minBiasAbsolute(ebts);
disp(t)
```

```
    PortfolioID    VaRID    VaRLevel    MinBiasAbsolute    PValue    TestStatistic    CriticalVa
    ___________    _____    ________    _______________    ______    _____________    __________

       "S&P"       "T 5"     0.975          accept         0.299     -0.00080059      -0.003037
```

```
whos s
```

```
  Name      Size                 Bytes  Class     Attributes

  s         1x1000                8000  double
```

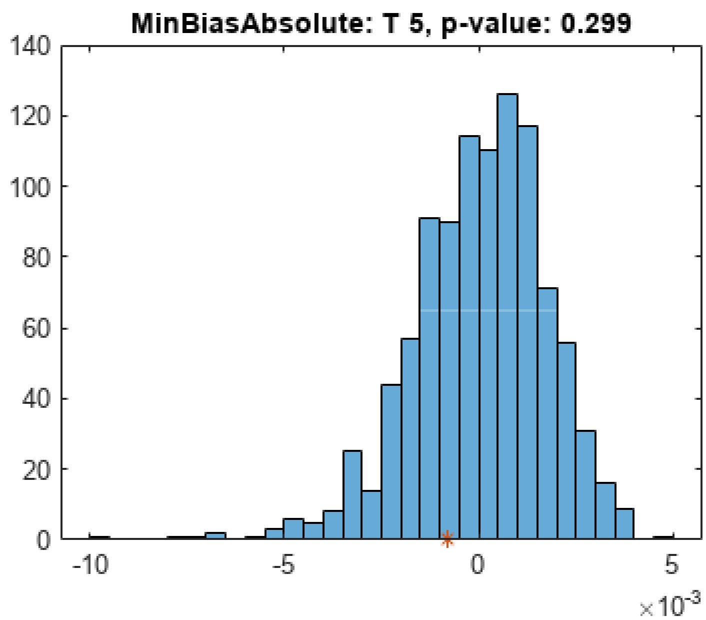
Select a test to show the test results and visualize the significance of the tests. The histogram shows the distribution of simulated test statistics, and the asterisk shows the value of the test statistic for the actual portfolio returns.

```
ESTestChoice = MinBiasAbsolute ;

switch ESTestChoice
   case 'MinBiasAbsolute'
      [t,s] = minBiasAbsolute(ebts);
   case 'MinBiasRelative'
      [t,s] = minBiasRelative(ebts);
```

```
    case 'Conditional'
        [t,s] = conditional(ebts);
    case 'Unconditional'
        [t,s] = unconditional(ebts);
    case 'Quantile'
        [t,s] = quantile(ebts);
end

disp(t)
```

```
    PortfolioID    VaRID    VaRLevel    MinBiasAbsolute    PValue    TestStatistic    CriticalVa
    ___________    _____    ________    _______________    ______    _____________    __________

       "S&P"       "T 5"     0.975          accept         0.299      -0.00080059      -0.003037
```

```
figure;
histogram(s);
hold on;
plot(t.TestStatistic,0,'*');
hold off;
Title = sprintf('%s: %s, p-value: %4.3f',ESTestChoice,t.VaRID,t.PValue);
title(Title)
```



The unconditional test statistic reported by `esbacktestbysim` is exactly the same as the unconditional test statistic reported by `esbacktest`. However the critical values reported by `esbacktestbysim` are based on a simulation using a *t* distribution with 5 degrees of freedom and the given location and scale parameters. The `esbacktest` object gives approximate test results for the `"T 5"` model, whereas the results here are specific for the distribution information provided. Also, for the `conditional` test, this is a visualization of the standalone conditional test (`ConditionalOnly` result in the table above). The final conditional test result (`Conditional` column) depends also on a preliminary VaR backtest (`VaRTestResult` column).

The `"T 5"` model is accepted by the five tests.

The `esbacktestbysim` object provides a `simulate` function to run a new simulation. For example, if there is a borderline test result where the test statistic is near the critical value, you might use the `simulate` function to simulate new scenarios. In cases where more precision is required, a larger simulation can be run.

The `esbacktestbysim` tests can be run over a rolling window, following the same approach described above for `esbacktest`. User-defined traffic-light tests can also be defined, for example, using two different test confidence levels, similar to what was done above for `esbacktest`.

Large-Sample and Simulation Tests

The `esbacktestbyde` object supports two ES back tests with critical values determined either with a large-sample approximation or a simulation (finite sample). `esbacktestbyde` requires the distribution information for the portfolio outcomes, namely, the distribution name (`"normal"` or `"t"`) and the distribution parameters for each day in the test window. It does not require the VaR of the ES data. `esbacktestbyde` uses the provided distribution information to map the portfolio outcomes into "ranks", that is, to apply the cumulative distribution function to map returns into values in the unit interval, where the test statistics are defined. `esbacktestbyde` can determine critical values by using a large-sample approximation or a finite-sample simulation.

The tests supported in `esbacktestbyde` are `conditionalDE` and `unconditionalDE`. These are implementations of the tests proposed by Du and Escanciano in [3 on page 2-61]. The `unconditionalDE` tests and all the tests previously discussed in this example are severity tests that assess if the magnitude of the tail losses is consistent with the model predictions. The `conditionalDE` test, however, is a test for independence over time that assess if there is evidence of autocorrelation in the series of tail losses.

The `esbacktestbyde` object supports normal and *t* distributions. These tests can be used for any model where the underlying distribution of portfolio outcomes is normal or *t*, such as exponentially weighted moving average (EWMA), delta-gamma, or generalized autoregressive conditional heteroskedasticity (GARCH) models.

The `"Normal"`, `"T 10"`, and `"T 5"` models can be backtested with the tests in `esbacktestbyde`. For illustration purposes, only `"T 5"` is backtested. The distribution name (`"t"`) and parameters (`DegreesOfFreedom`, `Location`, and `Scale`) are provided when the `esbacktestbyde` object is created.

```
rng('default'); % for reproducibility; the esbacktestbyde constructor runs a simulation
ebtde = esbacktestbyde(ReturnsTest,"t",'DegreesOfFreedom',5,...
   'Location',Mu,'Scale',SigmaT5,...
   'PortfolioID',"S&P",'VaRID',"T 5",'VaRLevel',VaRLevel);
```

The recommended workflow is the same: first, run the `summary` function, then run the `runtests` function, and then run the individual test functions. The `summary` function provides exactly the same information as the `summary` function from `esbacktest`.

```
s = summary(ebtde);
disp(s)
```

```
    PortfolioID    VaRID    VaRLevel    ObservedLevel    ExpectedSeverity    ObservedSeverity
    ___________    _____    ________    _____________    ________________    ________________

       "S&P"       "T 5"     0.975         0.97173             1.37               1.4075
```

The `runtests` function shows the final accept or reject result.

```
t = runtests(ebtde);
disp(t)

    PortfolioID    VaRID    VaRLevel    ConditionalDE    UnconditionalDE
    ___________    _____    ________    _____________    _______________

       "S&P"       "T 5"     0.975         reject            accept
```

Additional details on the test results are obtained by calling the individual test functions.

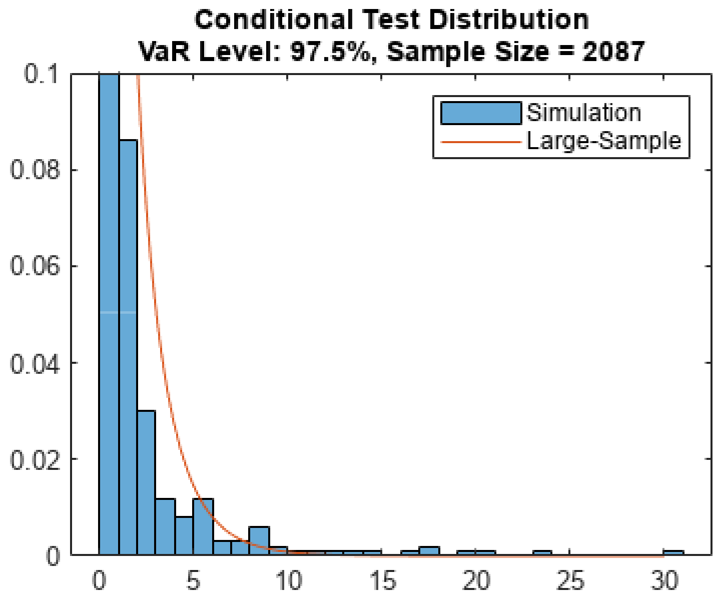
```
t = conditionalDE(ebtde);
disp(t)

    PortfolioID    VaRID    VaRLevel    ConditionalDE      PValue      TestStatistic    Critical\
    ___________    _____    ________    _____________    __________    _____________    _________

       "S&P"       "T 5"     0.975         reject        0.00034769       12.794           3.841
```

By default, the critical values are determined by a large-sample approximation. Critical values based on a finite-sample distribution estimated by using a simulation are available when using the `'CriticalValueMethod'` optional name-value pair argument.

```
[t,s] = conditionalDE(ebtde,'CriticalValueMethod','simulation');
disp(t)

    PortfolioID    VaRID    VaRLevel    ConditionalDE    PValue    TestStatistic    CriticalValue
    ___________    _____    ________    _____________    ______    _____________    _____________

       "S&P"       "T 5"     0.975         reject         0.01        12.794            3.7961
```

The second output, `s`, contains simulated test statistic values. The following visualization is useful for comparing how well the simulated finite-sample distribution matches the large-sample approximation. The plot shows that the tail of the distribution of test statistics is slightly heavier for the simulation-based (finite-sample) distribution. This means the simulation-based version of the tests are more tolerant and would not reject in some cases where the large-sample approximation results reject. How closely the large-sample and simulation distributions match depends not only on the number of observations in the test window, but also on the VaR confidence level (higher VaR levels lead to heavier tails in the finite-sample distribution).

```
xLS = 0:0.05:30;
pdfLS = chi2pdf(xLS,t.NumLags);
histogram(s,'Normalization',"pdf")
hold on
plot(xLS,pdfLS)
hold off
ylim([0 0.1])
legend({'Simulation','Large-Sample'})
Title = sprintf('Conditional Test Distribution\nVaR Level: %g%%, Sample Size = %d',VaRLevel*100,t
title(Title)
```



Similar steps can be used to see details on the `unconditionalDE` test, and to compare the large-sample and simulation based results.

The `esbacktestbyde` object provides a `simulate` function to run a new simulation. For example, if there is a borderline test result where the test statistic is near the critical value, you can use the `simulate` function to simulate new scenarios. Also, by default, the simulation stores results for up to 5 lags for the conditional test, so if simulation-based results for a larger number of lags is needed, you must use the `simulate` function.

If the large-sample approximation tests are the only tests that you need because they are reliable for a particular sample size and VaR level, you can turn off simulation when creating an `esbacktestbyde` object by using the `'Simulate'` optional input.

The `esbacktestbyde` tests can be run over a rolling window, following the same approach described above for `esbacktest`. You can also define traffic-light tests, for example, you could use two different test confidence levels, similar to what was done above for `esbacktest`.

Conclusions

To contrast the three ES backtesting objects:

- The `esbacktest` object is used for a wide range of distributional assumptions: historical VaR, parametric VaR, Monte-Carlo VaR, or extreme-value models. However, `esbacktest` offers approximate test results based on two variations of the same test: the unconditional test statistic with two different sets of precomputed critical values (`unconditionalNormal` and `unconditionalT`).
- The `esbacktestbysim` object is used for parametric methods with normal and *t* distributions (which includes EWMA, GARCH, and delta-gamma) and requires distribution parameters as inputs. `esbacktestbysim` offers five different tests (`conditional`, `unconditional`, `quantile`, `minBiasAbsolute`, and `minBiasRelative` and the critical values for these tests are simulated using the distribution information that you provide, and therefore, are more accurate. Although all ES backtests are sensitive to VaR estimation errors, the minimally biased test has only a small

sensitivity and is recommended (see Acerbi-Szekely 2017 and 2019 for details [2 on page 2-61], [3 on page 2-61]).

- The `esbacktestbyde` object is also used for parametric methods with normal and *t* distributions (which includes EWMA, GARCH, and delta-gamma) and requires distribution parameters as inputs. `esbacktestbyde` contains a severity (`unconditionalDE`) and a time-independence (`conditionalDE`) tests and it has the convenience of a large-sample, fast version of the tests. The `conditionalDE` test is the only test for independence over time for ES models among all the tests supported in these three classes.

As shown in this example, all three ES backtesting objects provide functionality to generate reports on severities, VaR failures, and test results information. The three ES backtest objects provide the flexibility to build on them. For example, you can create user-defined traffic-light tests and run the ES backtesting analysis over rolling windows.

References

[1] Acerbi, C., and B. Szekely. "Backtesting Expected Shortfall." MSCI Inc., December 2014.

[2] Acerbi, C., and B. Szekely. "General Properties of Backtestable Statistics. *SSRN Electronic Journal*. January, 2017.

[3] Acerbi, C., and B. Szekely. "The Minimally Biased Backtest for ES." *Risk*. September, 2019.

[4] Basel Committee on Banking Supervision. "Minimum Capital Requirements for Market Risk." January 2016, https://www.bis.org/bcbs/publ/d352.htm

[5] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science*. Vol 63, Issue 4, April 2017.

[6] McNeil, A., R. Frey, and P. Embrechts. *Quantitative Risk Management: Concepts, Techniques and Tools*. Princeton University Press. 2005.

[7] Rockafellar, R. T. and S. Uryasev. "Conditional Value-at-Risk for General Loss Distributions." *Journal of Banking and Finance*. Vol. 26, 2002, pp. 1443-1471.

Local Functions

```
function [VaR,ES] = hHistoricalVaRES(Sample,VaRLevel)
    % Compute historical VaR and ES
    % See [7] for technical details

    % Convert to losses
    Sample = -Sample;

    N = length(Sample);
    k = ceil(N*VaRLevel);
```

```
    z = sort(Sample);

    VaR = z(k);

    if k < N
       ES = ((k - N*VaRLevel)*z(k) + sum(z(k+1:N)))/(N*(1 - VaRLevel));
    else
       ES = z(k);
    end
end

function [VaR,ES] = hNormalVaRES(Mu,Sigma,VaRLevel)
    % Compute VaR and ES for normal distribution
    % See [6] for technical details

    VaR = -1*(Mu-Sigma*norminv(VaRLevel));
    ES = -1*(Mu-Sigma*normpdf(norminv(VaRLevel))./(1-VaRLevel));

end

function [VaR,ES] = hTVaRES(DoF,Mu,Sigma,VaRLevel)
    % Compute VaR and ES for t location-scale distribution
    % See [6] for technical details

    VaR = -1*(Mu-Sigma*tinv(VaRLevel,DoF));
    ES_StandardT = (tpdf(tinv(VaRLevel,DoF),DoF).*(DoF+tinv(VaRLevel,DoF).^2)./((1-VaRLevel).*(Do
    ES = -1*(Mu-Sigma*ES_StandardT);

end
```

See Also

Related Examples

- “Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30
- “Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63

More About

- “Overview of Expected Shortfall Backtesting” on page 2-20

Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano

This example shows the workflow for using the Du-Escanciano (DE) expected shortfall (ES) backtests and demonstrates a fixed test window for a single DE model with multiple VaR levels.

Load Data

The data in the `ESBacktestDistributionData.mat` file has returns, VaR and ES data, and distribution information for three models: normal, and *t* with 5 degrees of freedom and *t* with 10 degrees of freedom. The data spans multiple years from January 1996 to July 2003 and includes a total of 1966 observations.

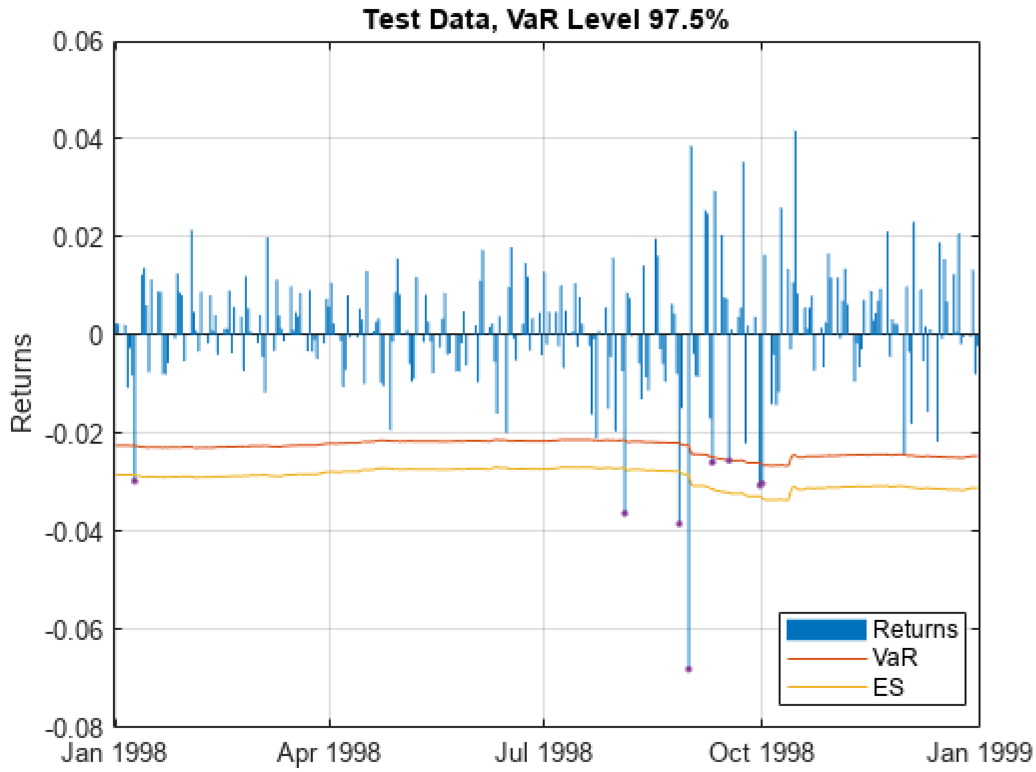
This example uses a *t* distribution with 10 degrees of freedom and focuses on one year of data to show the difference between the critical value methods for large-sample approximation and simulation supported by the `esbacktestbyde` class.

```
load ESBacktestDistributionData.mat

TargetYear = 1998; % Change to test other calendar years
Ind = year(Dates)==TargetYear;
Dates = Dates(Ind);
Returns = Returns(Ind);
VaR = T10VaR(Ind,:);
ES = T10ES(Ind,:);
Mu = 0; % Always 0 in this data set
Sigma = T10Scale(Ind);
```

Plot Data

Plot the data for a VaR level of `0.975`.

```
% Plot data
TargetVaRLevel = 0.975;
VaRInd = VaRLevel==TargetVaRLevel;

FailureInd = Returns<-VaR(:,VaRInd);

bar(Dates,Returns)
hold on
plot(Dates,-VaR(:,VaRInd),Dates,-ES(:,VaRInd))
plot(Dates(FailureInd),Returns(FailureInd),'.')
hold off
legend('Returns','VaR','ES','Location','best')
title(['Test Data, VaR Level ' num2str(TargetVaRLevel*100) '%'])
ylabel('Returns')
grid on
```



Create an esbacktestbyde Object

Create an `esbacktestbyde` object to run the DE tests. Note that VaR and ES data are not required inputs because the DE tests work on "mapped returns" or "ranks" and perform mapping by using the distribution information. However, for convenience, the `esbacktestbyde` object computes the VaR and ES data internally using the distribution information and stores the data in the `VaRData` and `ESData` properties of the `esbacktestbyde` object. The VaR and ES data is used only to estimate the severity ratios reported by the `summary` function and are not used for any of the DE tests.

By default, when you create a `esbacktestbyde` object, a simulation runs and large-sample and simulation-based critical values are available immediately. Although the simulation processing is efficient, if you verify that large-sample approximation is appropriate for the sample size and VaR level under consideration, you can turn the simulation off to increase processing speed. To turn off the simulation, when using `esbacktestbyde` to create an `esbacktestbtde` object, set the name-value pair argument `'Simulate'` to `false`.

```
rng('default'); % For reproducibility
tic;
ebtde = esbacktestbyde(Returns,"t",...
'DegreesOfFreedom',10,...
'Location',Mu,...
'Scale',Sigma,...
'VaRLevel',VaRLevel,...
'PortfolioID',"S&P",...
'VaRID',"t(10)");
toc;
```

```
Elapsed time is 0.098999 seconds.
```

```
disp(ebtde)
```

```
  esbacktestbyde with properties:

    PortfolioData: [261x1 double]
          VaRData: [261x3 double]
           ESData: [261x3 double]
     Distribution: [1x1 struct]
      PortfolioID: "S&P"
            VaRID: ["t(10)"    "t(10)"    "t(10)"]
         VaRLevel: [0.9500 0.9750 0.9900]
```

```
disp(ebtde.Distribution)
```

```
                Name: "t"
    DegreesOfFreedom: 10
            Location: 0
               Scale: [261x1 double]
```

Summary Statistics

Use `summary` to return a basic expected shortfall (ES) report on failures and severity. This is the same `summary` output as the other ES backtesting classes `esbacktest` and `esbacktestbysim`. When the `esbacktestbyde` object is created, the VaR and ES data are computed using the distribution information. This information is stored in the `VaRData` and `ESData` properties. The `summary` function uses the `VaRData` and `ESData` properties to compute the observed severity ratio.

```
disp(summary(ebtde))
```

```
    PortfolioID     VaRID      VaRLevel     ObservedLevel    ExpectedSeverity    ObservedSeverity
    ___________    _______    ________    _____________    ________________    ________________

       "S&P"       "t(10)"      0.95         0.94253            1.3288              1.5295
       "S&P"       "t(10)"     0.975         0.96935            1.2652              1.5269
       "S&P"       "t(10)"      0.99         0.98467            1.2169              1.5786
```

Run Tests

Use `runtests` to run all expected shortfall (ES) backtests for `esbacktestbyde` object. The default critical value method is `'large-sample'` or asymptotic approximation.

```
disp(runtests(ebtde))
```

```
    PortfolioID     VaRID      VaRLevel     ConditionalDE    UnconditionalDE
    ___________    _______    ________    _____________    _______________

       "S&P"       "t(10)"      0.95          accept             accept
       "S&P"       "t(10)"     0.975          accept             accept
       "S&P"       "t(10)"      0.99          accept             accept
```

Run the tests with `'simulation'` or finite-sample critical values.

```
disp(runtests(ebtde,'CriticalValueMethod','simulation'))
```

```
    PortfolioID     VaRID      VaRLevel     ConditionalDE    UnconditionalDE
    ___________    _______    ________    _____________    _______________
```

```
     "S&P"       "t(10)"      0.95         accept             accept
     "S&P"       "t(10)"     0.975         accept             accept
     "S&P"       "t(10)"      0.99         accept             accept
```

The `runtests` function accepts the name-value pair argument `'ShowDetails'` which includes extra columns in the output. Specifically, this output includes the critical value method used, number of lags, and test confidence level.

```
disp(runtests(ebtde,'CriticalValueMethod','simulation','ShowDetails',true))

    PortfolioID      VaRID      VaRLevel     ConditionalDE     UnconditionalDE     CriticalValueMetho
    ___________    _______    ________    _____________     _______________     __________________

       "S&P"       "t(10)"      0.95         accept             accept                "simulation"
       "S&P"       "t(10)"     0.975         accept             accept                "simulation"
       "S&P"       "t(10)"      0.99         accept             accept                "simulation"
```

Unconditional DE Test Details

The unconditional DE test assesses the severity of the violations based on an evaluation of the observed average tail loss and determines whether the severity is consistent with the model assumptions. All the tests supported in the related classes `esbacktest` and `esbacktestbysim` are also severity tests.

To view the unconditional DE test details, use the `unconditionalDE` function. By default, this function uses the `'large-sample'` critical value method.

```
disp(unconditionalDE(ebtde))

    PortfolioID      VaRID      VaRLevel     UnconditionalDE      PValue      TestStatistic      LowerC
    ___________    _______    ________    _______________    ________    _____________     _______

       "S&P"       "t(10)"      0.95          accept            0.31715       0.032842         0.00963
       "S&P"       "t(10)"     0.975          accept            0.32497       0.018009         0.00152
       "S&P"       "t(10)"      0.99          accept           0.076391       0.011309
```

To compare the results of `'large-sample'` to simulation-based critical values, use the name-value pair argument `'CriticalValueMethod'`. In this example, the results of both critical value methods, including the confidence interval and the *p*-values, look similar.

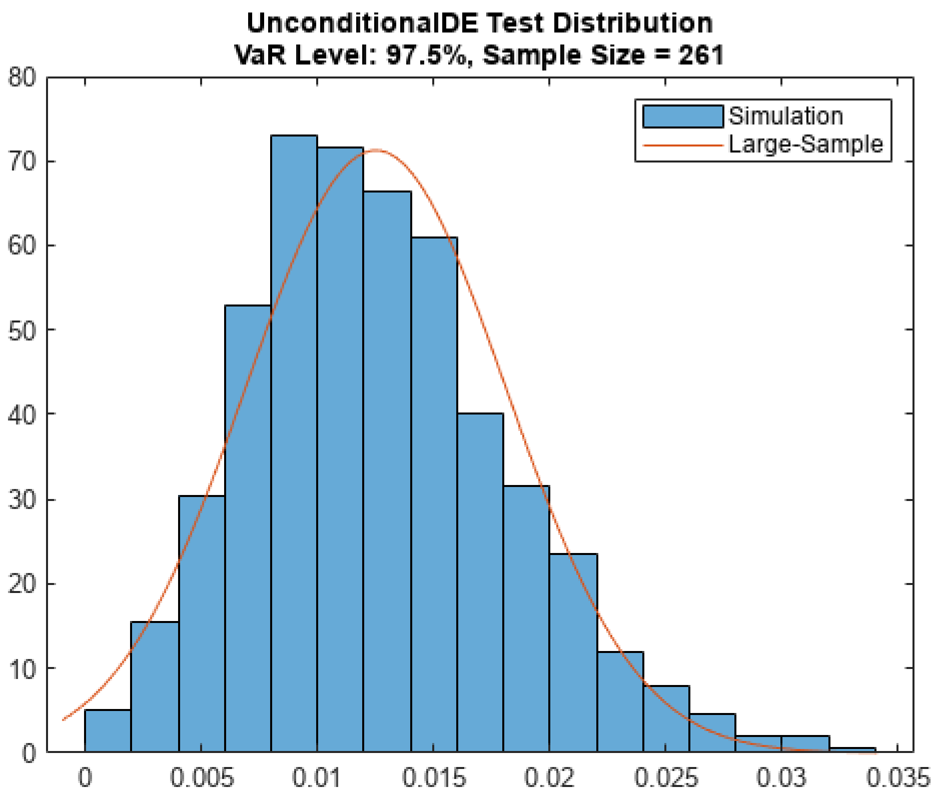
```
disp(unconditionalDE(ebtde,'CriticalValueMethod','simulation'))

    PortfolioID      VaRID      VaRLevel     UnconditionalDE     PValue     TestStatistic     LowerCI
    ___________    _______    ________    _______________    ______    _____________    ________

       "S&P"       "t(10)"      0.95          accept          0.326        0.032842        0.010859
       "S&P"       "t(10)"     0.975          accept          0.336        0.018009       0.0032440
       "S&P"       "t(10)"      0.99          accept          0.126        0.011309               0
```

You can visualize the `'simulation'` and `'large-sample'` distributions to assess whether the `'large-sample'` approximation is accurate enough for the sample size and VaR level under consideration. The `unconditionalDE` function returns the `'simulated'` test statistics as an optional output.

In this example, higher VaR levels cause a noticeable mismatch between the `'large-sample'` and `'simulation'` distributions. However, the confidence intervals and *p*-values are comparable.

```
% Choose VaR level
TargetVaRLevel = 0.975;
```

```
VaRInd = VaRLevel==TargetVaRLevel;

[~,s] = unconditionalDE(ebtde,'CriticalValueMethod','simulation');
histogram(s(VaRInd,:),'Normalization',"pdf")
hold on
t = unconditionalDE(ebtde,'CriticalValueMethod','large-sample');
Mu = t.MeanLS(VaRInd);
Sigma = t.StdLS(VaRInd);
MinValPlot = min(s(VaRInd,:))-0.001;
MaxValPlot = max(s(VaRInd,:))+0.001;
xLS = linspace(MinValPlot,MaxValPlot,101);
pdfLS = normpdf(xLS,Mu,Sigma);
plot(xLS,pdfLS)
hold off
legend({'Simulation','Large-Sample'})
Title = sprintf('UnconditionalDE Test Distribution\nVaR Level: %g%%, Sample Size = %d',VaRLevel(V
title(Title)
```



Conditional DE Test Details

The conditional DE test assesses whether there is evidence of autocorrelation in the tail losses.

Although the names are similar, the conditional DE test and the conditional test supported in `esbacktestbysim` are qualitatively different tests. The conditional Acerbi-Szekely test supported in `esbacktestbysim` tests the severity of the ES, conditional on whether the model passes a VaR test. The Acerbi-Szekely conditional test is a severity test, comparable to the tests supported in `esbacktest`, `esbacktestbysim`, and the `unconditionalDE` test.

However, the conditional DE test in `esbacktestbyde` is a test for independence across time periods.

To see the details of the conditional DE test results, use the `conditionalDE` function. By default, this function uses the `'large-sample'` critical value method and tests for one lag (correlation with the previous time period).

```
disp(conditionalDE(ebtde))
```

```
    PortfolioID      VaRID      VaRLevel     ConditionalDE     PValue      TestStatistic      CriticalVa
    ___________    ________    ________    _____________    _______    _____________    __________

       "S&P"       "t(10)"       0.95          accept       0.45361         0.5616           3.8415
       "S&P"       "t(10)"      0.975          accept       0.54189        0.37205           3.8415
       "S&P"       "t(10)"       0.99          accept       0.87949       0.022989           3.8415
```

The results of the `'large-sample'` critical value method, particularly the simulation critical values and *p*-values, differ substantially from the results of the `'simulation'` critical value method.

The critical value is similar for a 95% VaR level, but the simulation-based critical value is much larger for higher VaR levels, especially for a 99% VaR. The autocorrelation is `1` for any sample without VaR failures. Therefore, the test statistic equals the number of observations for any scenario without VaR failures. For a 99% VaR level, scenarios without VaR failures are like; consequently, there is a mass point at the number of observations which appears as a long, heavy tail in the simulated distribution of the test statistic.

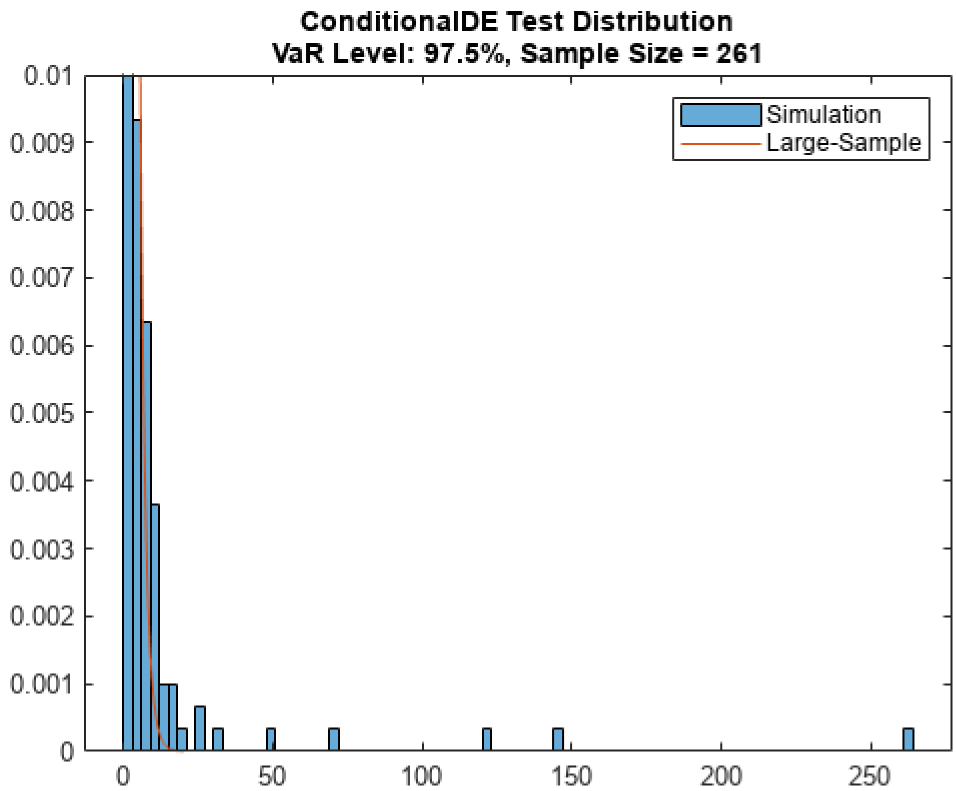
```
disp(conditionalDE(ebtde,'CriticalValueMethod','simulation'))
```

```
    PortfolioID      VaRID      VaRLevel     ConditionalDE     PValue     TestStatistic     CriticalVal
    ___________    ________    ________    _____________    ______    _____________    ___________

       "S&P"       "t(10)"       0.95          accept       0.257          0.5616           3.6876
       "S&P"       "t(10)"      0.975          accept       0.141         0.37205           5.3504
       "S&P"       "t(10)"       0.99          accept       0.502        0.022989              261
```

You can visually compare the `'large-sample'` and `'simulation'` distributions. The `conditionalDE` function also returns the simulated test statistics as an optional output.

Notice that the tail of the distribution gets heavier as the VaR level increases.

```
% Choose VaR level
TargetVaRLevel = 0.975;
VaRInd = VaRLevel==TargetVaRLevel;

[t,s] = conditionalDE(ebtde,'CriticalValueMethod','simulation');
xLS = 0:0.01:20;
pdfLS = chi2pdf(xLS,t.NumLags(1));
histogram(s(VaRInd,:),'Normalization',"pdf")
hold on
plot(xLS,pdfLS)
hold off
ylim([0 0.01])
legend({'Simulation','Large-Sample'})
Title = sprintf('ConditionalDE Test Distribution\nVaR Level: %g%%, Sample Size = %d',VaRLevel(VaR
title(Title)
```



Because the conditional DE test is based on autocorrelations, you can run the test for differing numbers of lags.

Run the conditional DE test for `2` lags. At a VaR level of 99%, the `'large-sample'` critical value method rejects the model but the `'simulation'` critical value method does not reject the model, with a *p*-value close to 10%. This shows that the `'simulation'` distribution and the `'large-sample'` approximation can lead to different results, depending on the sample size and VaR level.

```
disp(conditionalDE(ebtde,'NumLags',2,'CriticalValueMethod','large-sample'))
```

```
    PortfolioID      VaRID      VaRLevel    ConditionalDE      PValue       TestStatistic     Critica
    ___________    ________    ________    _____________    __________    _____________    _______

       "S&P"       "t(10)"       0.95         reject          0.015812         8.294             5.99
       "S&P"       "t(10)"      0.975         reject        0.00045758        15.379             5.99
       "S&P"       "t(10)"       0.99         reject        2.5771e-07        30.343             5.99
```

```
disp(conditionalDE(ebtde,'NumLags',2,'CriticalValueMethod','simulation'))
```

```
    PortfolioID      VaRID      VaRLevel    ConditionalDE    PValue    TestStatistic    CriticalVa
    ___________    ________    ________    _____________    ______    _____________    __________

       "S&P"       "t(10)"       0.95         reject          0.03         8.294            6.1397
       "S&P"       "t(10)"      0.975         reject         0.019        15.379            9.3364
       "S&P"       "t(10)"       0.99         accept         0.098        30.343               522
```

Running a New Simulation with simulate

If a *p*-value is near a rejection boundary, you can run a new simulation to request more scenarios to reduce a simulation error.

You can also run a new simulation to request a higher number of lags. By default, creating an `esbacktestbyde` object causes the simulation to run so that the simulation test results are available immediately. However, to avoid extra storage, only 5 lags are simulated. If you request more than 5 lags with the `simulate` function, the `conditionalDE` test function displays the following message:

```
No simulation results available for the number of lags requested. Call
'simulate' with the desired number of lags.
```

You first need to run a new simulation using `esbacktestbyde` and specify the number of lags to use for that simulation. Displaying the size of the `esbacktestbyde` object before and after the new simulation illustrates how simulating with more lags increases the amount of data stored in the `esbacktestbyde` object, as more simulated test statistics are stored with more lags.

```
% See bytes before new simulation, 5 lags stored
whos ebtde

  Name       Size              Bytes  Class             Attributes

  ebtde      1x1              164883  esbacktestbyde

% Simulate 6 lags
rng('default'); % for reproducibility
ebtde = simulate(ebtde,'NumLags',6);

% See bytes after new simulation, 6 lags stored
whos ebtde

  Name       Size              Bytes  Class             Attributes

  ebtde      1x1              188891  esbacktestbyde
```

After you run a new simulation with `esbacktestbyde` that increases the number of lags to 6, the test results for `conditionalDE` are available for the `'simulation'` method using 6 lags.

```
disp(conditionalDE(ebtde,'NumLags',6,'CriticalValueMethod','simulation'))

    PortfolioID    VaRID     VaRLevel    ConditionalDE    PValue    TestStatistic    CriticalVa
    ___________    _______    ________    _____________    ______    _____________    __________

       "S&P"       "t(10)"      0.95         accept        0.136        9.5173          16.412
       "S&P"       "t(10)"     0.975         accept        0.086        15.854          21.299
       "S&P"       "t(10)"      0.99         accept        0.128        30.438            1566
```

Alternatively, the `conditionalDE` test results are always available for the `'large-sample'` method for any number of lags.

```
disp(conditionalDE(ebtde,'NumLags',10,'CriticalValueMethod','large-sample'))

    PortfolioID    VaRID     VaRLevel    ConditionalDE     PValue     TestStatistic    Critica
    ___________    _______    ________    _____________    ________    _____________    _______

       "S&P"       "t(10)"      0.95         reject        0.018711       21.361          18.3
```

```
    "S&P"      "t(10)"    0.975        accept         0.088587      16.406          18.3
    "S&P"      "t(10)"     0.99        reject       0.00070234      30.526          18.3
```

See Also

esbacktestbyde | esbacktest | esbacktestbysim | varbacktest

Related Examples

- “VaR Backtesting Workflow” on page 2-6
- “Value-at-Risk Estimation and Backtesting” on page 2-10
- “Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30
- “Expected Shortfall (ES) Backtesting Workflow Using Simulation” on page 2-34
- “Expected Shortfall Estimation and Backtesting” on page 2-44
- “Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-72

Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano

This example shows the workflow for using the Du-Escanciano (DE) expected shortfall (ES) backtests for rolling window analyses and testing multiple VaR/ES models.

The rolling window workflow in this example is also used for the value-at-risk (VaR) backtests in `varbacktest` and for the Acerbi-Szekely ES backtests in the `esbacktest` and `esbacktestbysim` classes.

The multiple-model workflow in this example is also used for the `esbacktestbysim` class. For `esbacktest` and `varbacktest`, you can create a single object with multiple models and multiple VaR levels.

Rolling Window

The data in the `ESBacktestDistributionData.mat` file has returns, VaR and ES data, and distribution information for three models: normal, and *t* with 5 degrees of freedom and *t* with 10 degrees of freedom. The data spans multiple years from January 1996 to July 2003, for a total of 1966 observations.

To run the test over a rolling window, one `esbacktestbyde` object must be created for each year (or time period) of interest. In this example, each year from 1996 through 2002 is tested separately. You can test all VaR levels together, but to simplify the output, this example uses a single VaR level. You can also call any test, or the `summary` report inside the processing loop, but this example calls only the `runtests` function.

```
load ESBacktestDistributionData.mat

rng('default'); % For reproducibility

Years = 1996:2002;
TargetVaRLevel = 0.99;

t = table;
for TargetYear = Years

    Ind = year(Dates)==TargetYear;
    VaRInd = VaRLevel==TargetVaRLevel;

    ebtde = esbacktestbyde(Returns(Ind),"t",...
    'DegreesOfFreedom',10,...
    'Location',0,... % Always 0 in this data set
    'Scale',T10Scale(Ind),...
    'VaRLevel',VaRLevel(VaRInd),...
    'PortfolioID',strcat("S&P, ",string(TargetYear)),...
    'VaRID',"t(10)");

    t = [t; runtests(ebtde)];
end

disp(t)
```

```
    PortfolioID     VaRID     VaRLevel     ConditionalDE     UnconditionalDE
    ___________    _______    ________    _____________    _______________
```

```
    "S&P, 1996"    "t(10)"      0.99        reject           reject
    "S&P, 1997"    "t(10)"      0.99        accept           reject
    "S&P, 1998"    "t(10)"      0.99        accept           accept
    "S&P, 1999"    "t(10)"      0.99        reject           accept
    "S&P, 2000"    "t(10)"      0.99        accept           accept
    "S&P, 2001"    "t(10)"      0.99        accept           accept
    "S&P, 2002"    "t(10)"      0.99        reject           accept
```

For a more advanced approach, you can use arrays of `esbacktestbyde` objects and then call different functions on objects corresponding to different years as needed.

```
rng('default'); % For reproducibility

NumYears = length(Years);
ebtdeArray(NumYears) = esbacktestbyde;

TargetVaRLevel = 0.99;

for yy = 1:NumYears

    TargetYear = Years(yy);
    Ind = year(Dates)==TargetYear;
    VaRInd = VaRLevel==TargetVaRLevel;

    ebtdeArray(yy) = esbacktestbyde(Returns(Ind),"t",...
    'DegreesOfFreedom',10,...
    'Location',0,... % Always 0 in this data set
    'Scale',T10Scale(Ind),...
    'VaRLevel',VaRLevel(VaRInd),...
    'PortfolioID',strcat("S&P, ",string(TargetYear)),...
    'VaRID',"t(10)");

end

disp(ebtdeArray)
```

```
  1x7 esbacktestbyde array with properties:

    PortfolioData
    VaRData
    ESData
    Distribution
    PortfolioID
    VaRID
    VaRLevel
```

Display the summary for the year 2002.

```
disp(summary(ebtdeArray(Years==2002)))
```

```
    PortfolioID     VaRID     VaRLevel    ObservedLevel    ExpectedSeverity    ObservedSeverity
    ___________    _______    ________    _____________    ________________    ________________

    "S&P, 2002"    "t(10)"      0.99         0.98467            1.2169              1.1481
```

Concatenate the conditional tests for all years.

```
condDEResults = table;
for yy = 1:NumYears
```

```
    condDEResults = [condDEResults; conditionalDE(ebtdeArray(yy))];
end
disp(condDEResults)
```

```
    PortfolioID     VaRID     VaRLevel    ConditionalDE       PValue      TestStatistic    Critica
    ___________    _______    ________    _____________    __________    _____________    _______

    "S&P, 1996"    "t(10)"      0.99         reject         0.0084691          6.9315         3.84
    "S&P, 1997"    "t(10)"      0.99         accept           0.85691        0.032512         3.84
    "S&P, 1998"    "t(10)"      0.99         accept           0.87949        0.022989         3.84
    "S&P, 1999"    "t(10)"      0.99         reject        2.1168e-50          222.89         3.84
    "S&P, 2000"    "t(10)"      0.99         accept           0.89052        0.018948         3.84
    "S&P, 2001"    "t(10)"      0.99         accept           0.92088       0.0098664         3.84
    "S&P, 2002"    "t(10)"      0.99         reject        3.5974e-05          17.073         3.84
```

Multiple Models

Similar to the `esbacktestbysim` object, the `esbacktestbyde` object accepts only one distribution at a time. If you need to test different models side by side, then you must create different instances of the class.

In this example you run the test for a normal distribution assumption and *t* distributions with 5 and 10 degrees of freedom. You then concatenate the test results to generate a single report.

The data in the `ESBacktestDistributionData.mat` file has returns, VaR and ES data, and distribution information for three models: normal, and t with 5 and 10 degrees of freedom. The data spans multiple years from January 1996 to July 2003, for a total of 1966 observations. For simplicity, this example uses only data from 1998.

```
load ESBacktestDistributionData.mat

TargetYear = 1998;
Ind = year(Dates)==TargetYear;

rng('default'); % For reproducibility
```

Create an instance of an `esbacktestbyde` object for the normal distribution.

```
ebtdeNormal = esbacktestbyde(Returns(Ind),"normal",...
'Mean',0,...
'StandardDeviation',NormalStd(Ind),...
'VaRLevel',VaRLevel,...
'PortfolioID',strcat("S&P, ",string(TargetYear)),...
'VaRID',"normal");

disp(ebtdeNormal)
```

```
  esbacktestbyde with properties:

    PortfolioData: [261x1 double]
          VaRData: [261x3 double]
           ESData: [261x3 double]
     Distribution: [1x1 struct]
      PortfolioID: "S&P, 1998"
            VaRID: ["normal"    "normal"    "normal"]
         VaRLevel: [0.9500 0.9750 0.9900]
```

```
disp(ebtdeNormal.Distribution)
```

```
                Name: "normal"
                Mean: 0
   StandardDeviation: [261x1 double]
```

Create an instance of an `esbacktestbyde` object for the *t* distribution with 10 degrees of freedom.

```
ebtdeT10 = esbacktestbyde(Returns(Ind),"t",...
'DegreesOfFreedom',10,...
'Location',0,...
'Scale',T10Scale(Ind),...
'VaRLevel',VaRLevel,...
'PortfolioID',strcat("S&P, ",string(TargetYear)),...
'VaRID',"t(10)");

disp(ebtdeT10)

  esbacktestbyde with properties:

    PortfolioData: [261x1 double]
          VaRData: [261x3 double]
           ESData: [261x3 double]
     Distribution: [1x1 struct]
      PortfolioID: "S&P, 1998"
            VaRID: ["t(10)"    "t(10)"    "t(10)"]
         VaRLevel: [0.9500 0.9750 0.9900]

disp(ebtdeT10.Distribution)

                Name: "t"
    DegreesOfFreedom: 10
            Location: 0
               Scale: [261x1 double]
```

Create an instance of an `esbacktestbyde` object for the *t* distribution with 5 degrees of freedom.

```
ebtdeT5 = esbacktestbyde(Returns(Ind),"t",...
'DegreesOfFreedom',5,...
'Location',0,...
'Scale',T5Scale(Ind),...
'VaRLevel',VaRLevel,...
'PortfolioID',strcat("S&P, ",string(TargetYear)),...
'VaRID',"t(5)");

disp(ebtdeT5)

  esbacktestbyde with properties:

    PortfolioData: [261x1 double]
          VaRData: [261x3 double]
           ESData: [261x3 double]
     Distribution: [1x1 struct]
      PortfolioID: "S&P, 1998"
            VaRID: ["t(5)"    "t(5)"    "t(5)"]
         VaRLevel: [0.9500 0.9750 0.9900]

disp(ebtdeT5.Distribution)

                Name: "t"
    DegreesOfFreedom: 5
```

```
           Location: 0
              Scale: [261x1 double]
```

Run the tests and then concatenate the results.

```
testResults = [runtests(ebtdeNormal); runtests(ebtdeT10); runtests(ebtdeT5)];
disp(testResults)
```

```
    PortfolioID     VaRID      VaRLevel    ConditionalDE    UnconditionalDE
    ___________    ________    ________    _____________    _______________

    "S&P, 1998"    "normal"      0.95         accept             accept
    "S&P, 1998"    "normal"     0.975         accept             accept
    "S&P, 1998"    "normal"      0.99         accept             reject
    "S&P, 1998"    "t(10)"       0.95         accept             accept
    "S&P, 1998"    "t(10)"      0.975         accept             accept
    "S&P, 1998"    "t(10)"       0.99         accept             accept
    "S&P, 1998"    "t(5)"        0.95         accept             accept
    "S&P, 1998"    "t(5)"       0.975         accept             accept
    "S&P, 1998"    "t(5)"        0.99         accept             accept
```

Display the results for a VaR level of `0.99`.

```
TargetVaRLevel = 0.99;
disp(testResults(testResults.VaRLevel == TargetVaRLevel,:))
```

```
    PortfolioID     VaRID      VaRLevel    ConditionalDE    UnconditionalDE
    ___________    ________    ________    _____________    _______________

    "S&P, 1998"    "normal"      0.99         accept             reject
    "S&P, 1998"    "t(10)"       0.99         accept             accept
    "S&P, 1998"    "t(5)"        0.99         accept             accept
```

See Also

`esbacktestbyde` | `esbacktest` | `esbacktestbysim` | `varbacktest`

Related Examples

- “VaR Backtesting Workflow” on page 2-6
- “Value-at-Risk Estimation and Backtesting” on page 2-10
- “Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30
- “Expected Shortfall (ES) Backtesting Workflow Using Simulation” on page 2-34
- “Expected Shortfall Estimation and Backtesting” on page 2-44
- “Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63

Managing Consumer Credit Risk Using the Binning Explorer for Credit Scorecards

Overview of Binning Explorer

The **Binning Explorer** app enables you to interactively bin credit scorecard data. Use the **Binning Explorer** to:

- Select an automatic binning algorithm with an option to bin missing data. (For more information on algorithms for automatic binning, see `autobinning`.)
- Shift bin boundaries.
- Split bins.
- Merge bins.
- Save and export a `creditscorecard` object.

Binning Explorer complements the overall workflow for developing a credit scorecard model. Use `screenpredictors` to pare down a potentially large set of predictors to a subset that is most predictive of the credit score card response variable. You can then use this subset of predictors when using **Binning Explorer** to create the `creditscorecard` object.

Using Binning Explorer:	

1.	Open the **Binning Explorer** app.
	• MATLAB® toolstrip: On the **Apps** tab, under **Computational Finance**, click the app icon.
	• MATLAB command prompt:
	  • Enter `binningExplorer` to open the **Binning Explorer** app.
	  • Enter `binningExplorer(data)` or `binningExplorer(data,Name,Value)` to open a table in the **Binning Explorer** app by specifying a table (`data`) as input.
	  • Enter `binningExplorer(sc)` to open a `creditscorecard` object in the **Binning Explorer** app by specifying a `creditscorecard` object (`sc`) as input.

2.	Import the data into the app. You can import data into **Binning Explorer** by either starting directly from a data set or by loading an existing `creditscorecard` object from the MATLAB workspace.
3.	Use **Binning Explorer** to work interactively with the binning assignments for a scorecard.
4.	Export the scorecard to a new `creditscorecard` object. Continue the workflow from the MATLAB command line using `creditscorecard` object functions from Financial Toolbox. For more information, see `creditscorecard`.
Using `creditscorecard` Object Functions in Financial Toolbox:	
5.	Fit a logistic regression model.
6.	Review and format the credit scorecard points.
7.	Score the data.
8.	Calculate the probabilities of default for the data.
9.	Validate the quality of the credit scorecard model.

For more detailed information on this workflow, see “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23.

See Also

Apps
Binning Explorer

Classes
`creditscorecard`

Related Examples

- “Common Binning Explorer Tasks” on page 3-4
- “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23
- “Case Study for Credit Scorecard Analysis”

More About

- “Credit Scorecard Modeling Workflow”

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

Common Binning Explorer Tasks

The **Binning Explorer** app supports the following tasks:

In this section...

Import Data

Binning Explorer enables you to import data by either starting directly from the data stored in a MATLAB table or by loading an existing `creditscorecard` object.

Clean Start from Data

To start directly from data:

1. Place the credit scorecard data in your MATLAB workspace. The data must be in a MATLAB table, where each column of data can be any one of the following data types:

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

   In addition, the table must contain a binary response variable.
2. Open **Binning Explorer** from the MATLAB toolstrip: On the **Apps** tab, under **Computational Finance**, click the app icon.
3. Click **Import Data** and select the data from the **Step 1** pane of the Import Data window.
4. From the **Step 2** pane, set the **Variable Type** for each of the predictors, as needed. If the input MATLAB table contains a column for `weights`, from the **Step 2** pane, using the **Variable Type** column, click the drop-down to select **Weights**. If the data contains missing values, from the **Step 2** pane, set **Bin missing data:** to **Yes**. For more information on working with missing data, see “Credit Scorecard Modeling with Missing Values”.

5 From the **Step 3** pane, select an initial binning algorithm and click **Import Data**. The bins are plotted and displayed for each predictor. By clicking an individual predictor plot in the **Overview** pane, the details for that predictor plot display in the main pane with additional information in the **Bin Information** and **Predictor Information** panes.

Start from an Existing creditscorecard Object

To start using an existing `creditscorecard` object:

1. Place the `creditscorecard` object in your MATLAB workspace. Create the `creditscorecard` object either by using `creditscorecard` or by clicking **Export** in the **Binning Explorer** to export and save a `creditscorecard` object to the MATLAB workspace.
2. Open Binning Explorer from the MATLAB toolstrip: On the **Apps** tab, under **Computational Finance**, click the app icon.
3. Click **Import Data** and from **Step 1** pane of the Import Data window, select the `creditscorecard` object.
4. From the **Step 3** pane, select a binning algorithm. When using an existing `creditscorecard` object, it is recommended to select the **No Binning** option. To display the predictor plots, click **Import Data**.

   The bins are plotted and displayed for each predictor. By clicking an individual predictor plot in the **Overview** pane, the predictor plot displays in the main pane and associated information displays in the **Bin Information** and **Predictor Information** panes.

Start from MATLAB Command Line Using Data or an Existing creditscorecard Object

To start **Binning Explorer** from the MATLAB command line:

1. Place the credit scorecard data or existing `creditscorecard` object in your MATLAB workspace.
2. At the MATLAB command prompt:

   - Enter `binningExplorer(data)` or `binningExplorer(data,Name,Value)` to open a table in the **Binning Explorer** app by specifying a table (`data`) as input.
   - Enter `binningExplorer(sc)` to open an existing `creditscorecard` object in the **Binning Explorer** app by specifying a `creditscorecard` object (`sc`) as input.

   The bins are plotted and displayed for each predictor. By clicking an individual predictor plot in the **Overview** pane, the details for that predictor plot display in the main pane and the associated details display in the **Bin Information** and **Predictor Information** panes.

Change Predictor Type

After you import data or a `creditscorecard` object into **Binning Explorer**, you can change the predictor type.

1. Click any predictor plot. The name of the selected predictor displays on the **Binning Explorer** toolstrip under **Selected Predictor**.

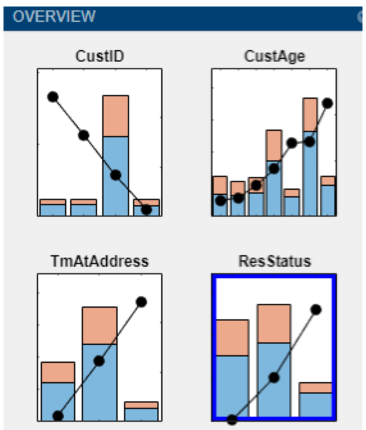
   On the **Binning Explorer** toolstrip, the predictor type for the selected predictor displays under **Predictor Type**.

2 To change the predictor type, under **Predictor Type**, select: `Numeric`, `Categorical`, or `Ordinal`. The predictor plot is updated and the details in the **Bin Information** and **Predictor Information** panes are also updated.

Change Binning Algorithm for One or More Predictors

After you import data or a `creditscorecard` object into **Binning Explorer**, you can change the binning algorithm for an individual predictor or for multiple predictors.

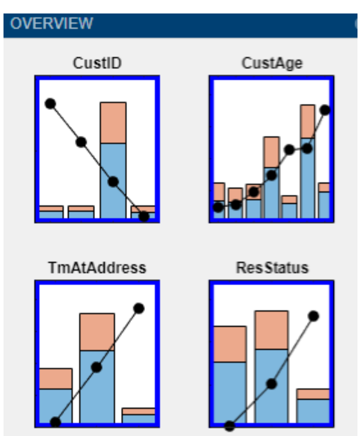
1 Click any predictor plot in the **Overview** pane. The selected predictor plot displays in the main pane.



Tip When you select a predictor plot, a status message appears above **Bin Information** that displays the last binning information for that predictor. Use this information to determine which binning algorithm is most recently applied to an individual predictor plot.

2 On the **Binning Explorer** toolstrip, click to select **Monotone**, **Split**, **Merge**, **Equal Frequency**, or **Equal Width**. The predictor plot is updated with a change of algorithm. The details in the **Bin Information** and **Predictor Information** panes are also updated.

3 To change the binning algorithm for multiple predictors, multiselect more than one predictor plot by using **Ctrl** + click or **Shift** + click to highlight each predictor plot with a blue outline.

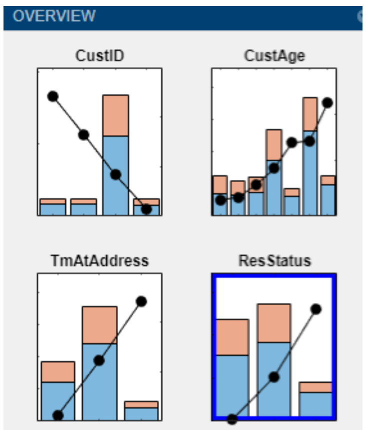


4 Click to select **Monotone**, **Split**, **Merge**, **Equal Frequency**, or **Equal Width**. All the selected predictor plots are updated for a change of algorithm.

Change Algorithm Options for Binning Algorithms

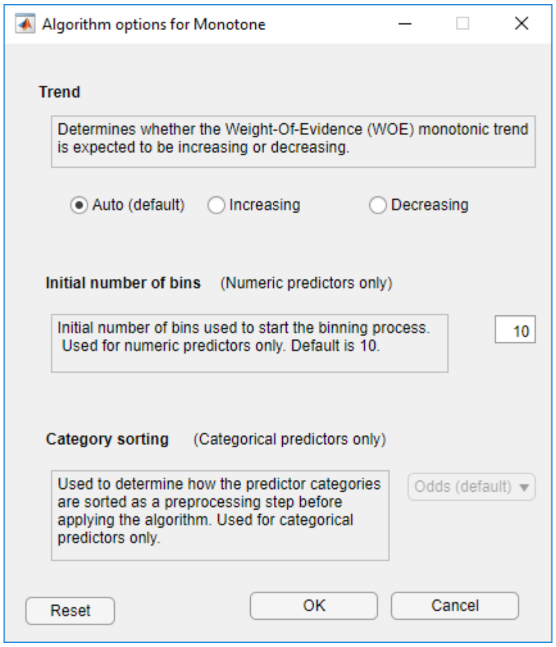
After you import data or a `creditscorecard` object into **Binning Explorer**, you can change the binning algorithm options for an individual predictor or for multiple predictors.

1 Click any predictor plot in the **Overview** pane. The predictor plot displays with a blue outline and displays in the main pane.



Tip When you select a predictor plot with the blue outline, a status message appears above **Bin Information** that displays the last binning information for that predictor. Use this information to determine which binning algorithm is most recently applied to an individual predictor plot.

2 On the **Binning Explorer** toolstrip, click **Options** to open a list of options for the **Monotone**, **Split**, **Merge**, **Equal Frequency**, and **Equal Width** algorithms. Click an option to open the associated Algorithm options dialog box. For example, clicking **Monotone Options** opens the **Algorithm options dialog box for Monotone**.



3 From the associated Algorithm options dialog box:

- **Monotone**
  - For **Trend**, select one of the following:
    - **Auto (default)** — Automatically determines if the WOE trend is increasing or decreasing.
    - **Increasing** — Looks for an increasing WOE trend.
    - **Decreasing** — Looks for a decreasing WOE trend.

    The value of **Trend** does not necessarily reflect that of the resulting WOE curve. The **Trend** option tells the algorithm to look for an increasing or decreasing trend, but the outcome might not show the desired trend. For example, the algorithm cannot find a decreasing trend when the data actually has an increasing WOE trend. For more information on the **Trend** option, see “Monotone”.
    - For **Initial number of bins**, enter an initial number of bins (default is `10`). The initial number of bins must be an integer > `2`. Used for numeric predictors only.
    - For **Category Sorting**, used for categorical predictors only, select one of the following:
      - **Odds (default)** — The categories are sorted by order of increasing values of odds, defined as the ratio of “Good” to “Bad” observations, for the given category.
      - **Goods** — The categories are sorted by order of increasing values of “Good.”
      - **Bads** — The categories are sorted by order of increasing values of “Bad.”
      - **Totals** — The categories are sorted by order of increasing values of the total number of observations (“Good” plus “Bad”).
      - **None** — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm.

      For more information, see Sort Categories

- **Split**
  - For **Measure**, select one of the following: **Gini** (default), **Chi2**, **InfoValue**, or **Entropy**.
  - For **Tolerance**, specify a tolerance value above which the gain in the information value has to be for the split to be accepted. The default is `1e-4`.
  - For **Significance**, only for the **Chi2** measure, specify a significance level threshold for the chi-square statistic, above which splitting happens. Values are in the interval `[0,1]`. Default is `0.9` (90% significance level).
  - For **Bin distribution**, specify values for
    - **MinBad** — Specifies the minimum number *n* (*n*`>=0`) of Bads per bin. The default value is `1`, to avoid pure bins.
    - **MaxBad** — Specifies the maximum number *n* (*n*`>=0`) of Bads per bin. The default value is `Inf`.
    - **MinGood** — Specifies the minimum number *n* (*n*`>=0`) of Goods per bin. The default value is `1`, to avoid pure bins.
    - **MaxGood** — Specifies the maximum number *n* (*n*`>=0`) of Goods per bin. The default value is `Inf`.
    - **MinCount** — Specifies the minimum number *n* (*n*`>=0`) of observations per bin. The default value is `1`, to avoid empty bins.
    - **MaxCount** — Specifies the maximum number *n* (*n*`>=0`) of observations per bin. The default value is `Inf`.
    - **MaxNumBins** — Specifies the maximum number *n* (*n*`>=2`) of bins resulting from the splitting. The default value is `5`.
  - For **Initial number bins**, specify an integer that determines the number (*n* >0) of bins that the predictor is initially binned into before splitting. Valid for numeric predictors only. Default is `50`.
  - For **Category sorting**, used for categorical predictors only, select a value:
    - **Goods** — The categories are sorted by order of increasing values of "Good."
    - **Bads** — The categories are sorted by order of increasing values of "Bad."
    - **Odds** — (default) The categories are sorted by order of increasing values of odds, defined as the ratio of "Good" to "Bad" observations, for the given category.
    - **Totals** — The categories are sorted by order of increasing values of total number of observations ("Good" plus "Bad").
    - **None** — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm. (The existing order of the categories can be seen in the category grouping optional output from `bininfo`.)

    For more information, see Sort Categories
- **Merge**
  - For **Measure**, select one of the following: **Chi2** (default), **Gini**, **InfoValue**, or **Entropy**.
  - For **Tolerance**, specify the minimum threshold below which merging happens for the information value and entropy statistics. Valid values are in the interval `(0.1)`. Default is `1e-3`.

- For **Significance**, specify the significance level threshold for the chi-square statistic, below which merging happens. Values are in the interval `[0,1]`. Default is `0.9` (90% significance level).
- For **Bin distribution**, specify the following:
  - **MinNumBins** — Specifies the minimum number *n* (*n*`>=2`) of bins that result from merging. The default value is `2`.
  - **MaxNumBins** — Specifies the maximum number *n* (*n*`>=2`) of bins that result from merging. The default value is `5`.
- For **Initial number of bins**, specify an integer that determines the number (*n* >0) of bins that the predictor is initially binned into before merging. Valid for numeric predictors only. Default is `50`.
- For **Category sorting**, used for categorical predictors only. Select a value:
  - **Goods** — The categories are sorted by order of increasing values of “Good.”
  - **Bads** — The categories are sorted by order of increasing values of “Bad.”
  - **Odds** — (default) The categories are sorted by order of increasing values of odds, defined as the ratio of “Good” to “Bad” observations, for the given category.
  - **Totals** — The categories are sorted by order of increasing values of total number of observations (“Good” plus “Bad”).
  - **None** — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm. (The existing order of the categories can be seen in the category grouping optional output from `bininfo`.)

  For more information, see Sort Categories

- **Equal Frequency**
  - For **Number of bins**, enter the number of bins. The default is `5`, and the number of bins must be a positive number.
  - For **Category Sorting**, select one of the following:
    - **Odds (default)** — The categories are sorted by order of increasing values of odds, defined as the ratio of “Good” to “Bad” observations, for the given category.
    - **Goods** — The categories are sorted by order of increasing values of “Good.”
    - **Bads** — The categories are sorted by order of increasing values of “Bad.”
    - **Totals** — The categories are sorted by order of increasing values of the total number of observations (“Good” plus “Bad”).
    - **None** — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm.

  Note You can use **Category Sorting** with categorical predictors only.

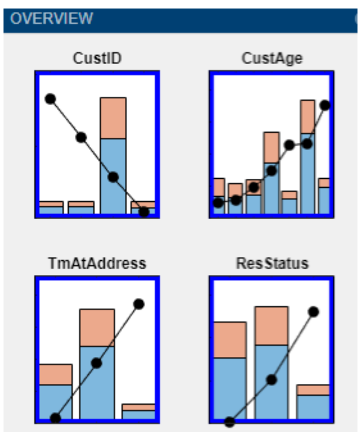
- **Equal Width**
  - For **Number of bins**, enter the number of bins. The default is `5` and the number of bins must be a positive number.
  - For **Category Sorting**, select one of the following:

- **Odds (default)** — The categories are sorted by order of increasing values of odds, defined as the ratio of "Good" to "Bad" observations, for the given category.
- **Goods** — The categories are sorted by order of increasing values of "Good."
- **Bads** — The categories are sorted by order of increasing values of "Bad."
- **Totals** — The categories are sorted by order of increasing values of the total number of observations ("Good" plus "Bad").
- **None** — No sorting is applied. The existing order of the categories is unchanged before applying the algorithm.

Note You can use **Category Sorting** with categorical predictors only.

Click **OK**. The selected predictor plot is updated with the change of algorithm options. The details in the **Bin Information** and **Predictor Information** panes are also updated. In addition, the updated algorithm options apply to any subsequent application of that algorithm to other predictors as described in "Change Binning Algorithm for One or More Predictors" on page 3-6.

4 To change the binning algorithm option for multiple predictors, multiselect more than one predictor plot by using **Ctrl+** click or **Shift** + click to highlight each predictor plot with a blue outline.

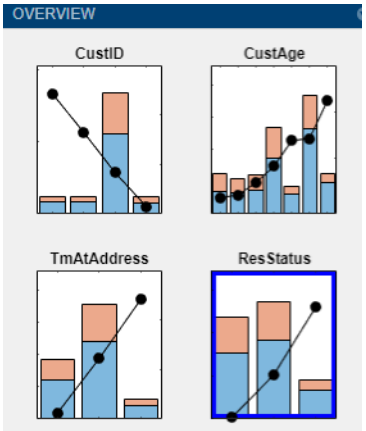


5 On the **Binning Explorer** toolstrip, click **Options** to open a list of options for the **Monotone**, **Split**, **Merge**, **Equal Frequency**, and **Equal Width** algorithms. Click an option to open the associated Algorithm options dialog box. Make your selection from the respective Algorithm Options dialog box and click **OK**. The selected predictor plots are updated for the change of algorithm.

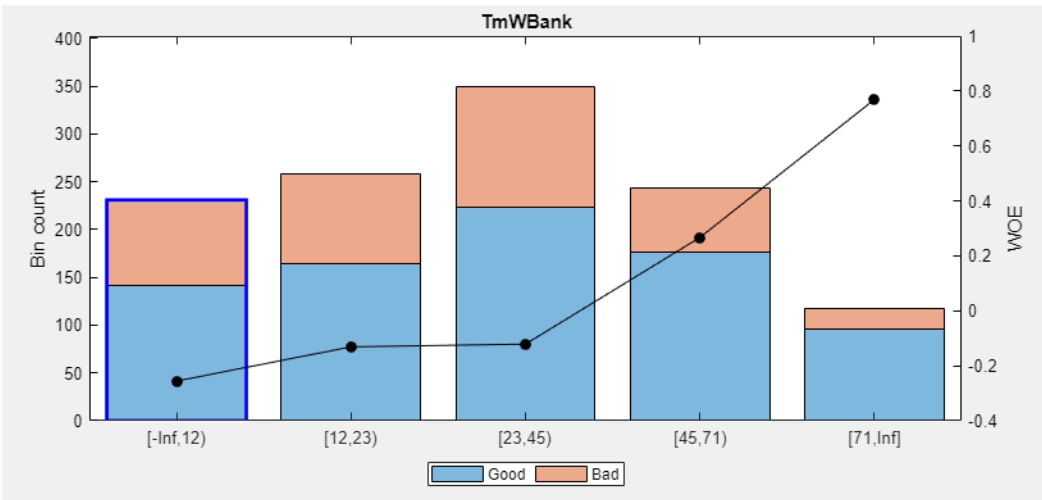
Split Bins for a Numeric Predictor

After you import data or a `creditscorecard` object into **Binning Explorer**, you can split bins for a numeric predictor.

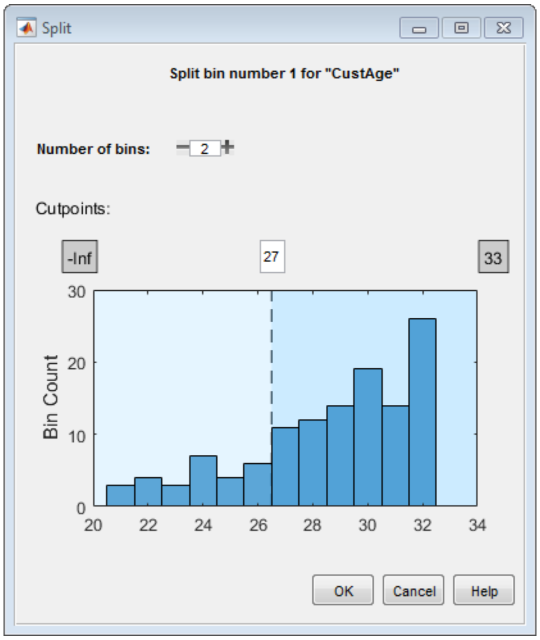
1 Click any numeric predictor plot in the **Overview** pane. The predictor plot displays in the main pane.



2 On the **Binning Explorer** toolstrip, the **Split** button is enabled. From the main pane, click a bin to apply the **Split** operation. To deselect a bin, use **Ctrl**+ click.



3 On the **Binning Explorer** toolstrip, the **Edges** text boxes display values for the edges of the selected bin. Click **Split** to open the Split dialog box.



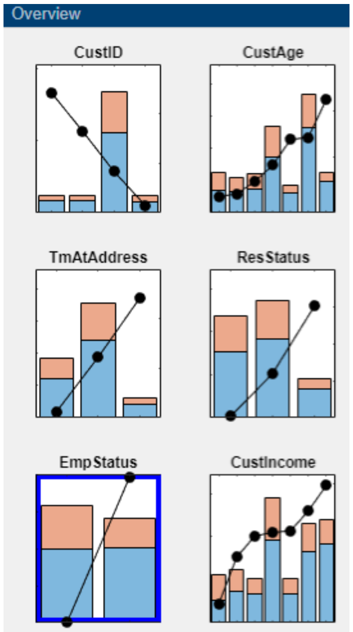
4 Use the **Number of bins** control to split the selected bin into multiple bins. Click **OK** to complete the split operation.

The plot for the selected numeric predictor is updated with the new bin information. The details in the **Bin Information** and **Predictor Information** panes are also updated.

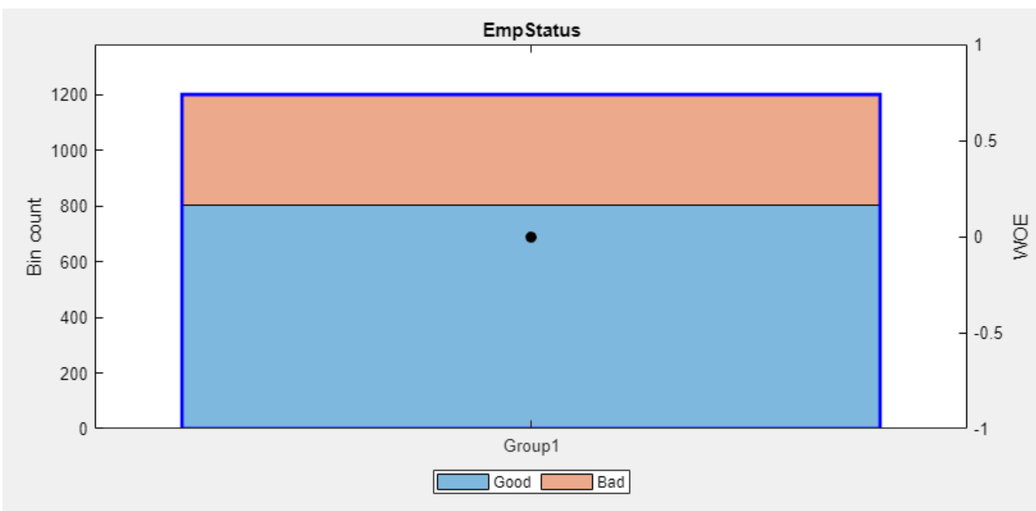
Split Bins for a Categorical Predictor

After you import data or a `creditscorecard` object into **Binning Explorer**, you can split bins for a categorical predictor.

1 Click any categorical predictor plot in the **Overview** pane. The predictor plot displays in the main pane.

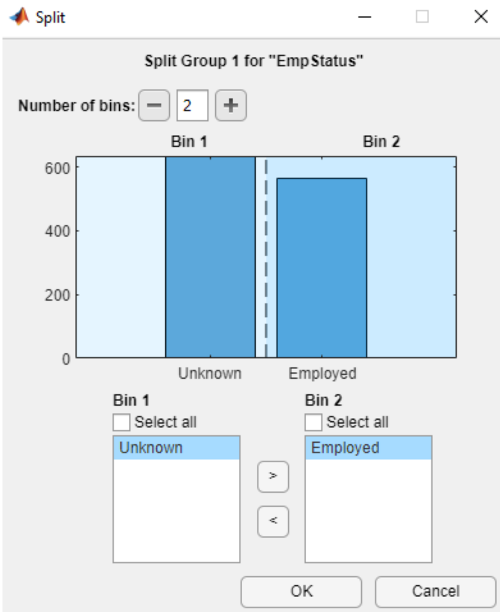


2 From the main pane, click a bin to enable the **Split** button for that bin. To deselect a bin, use **Ctrl**+ click.



On the **Binning Explorer** toolstrip, click **Split** to open the Split dialog for the selected bin.

Note The **Split** button is enabled when the selected bin has more than one unique category in it.



Use the **Number of bins** control to split the selected bin into multiple bins.

Use the arrow controls on the Split dialog box to control the contents for each of the bins that you are splitting the selected bin into.

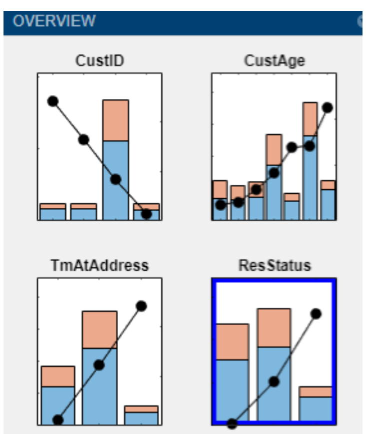
3 Click **OK** to complete the split operation.

The plot for the selected categorical predictor is updated with the new bin information. The details in the **Bin Information** and **Predictor Information** panes are also updated.

Manual Binning to Merge Bins for a Numeric or Categorical Predictor

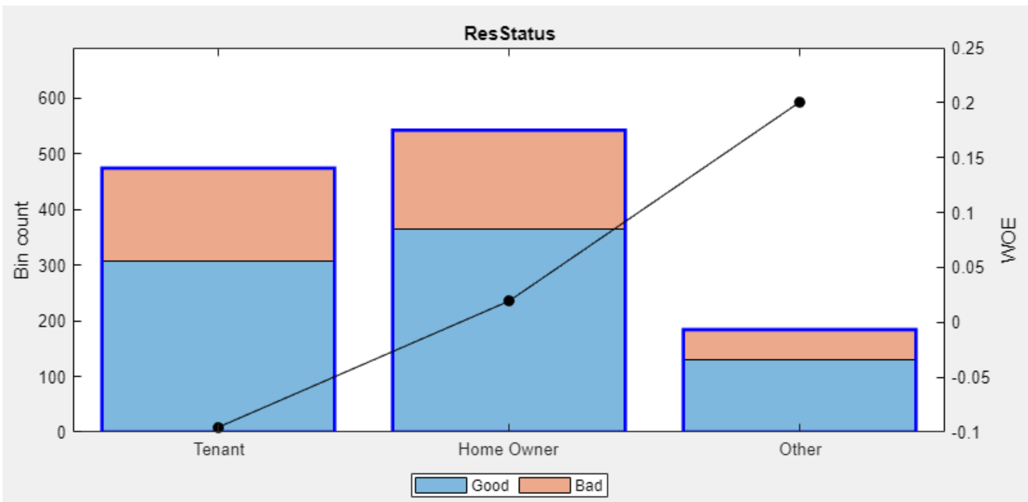
After you import data or a `creditscorecard` object into **Binning Explorer**, you can split or merge bins for a predictor.

1 Click any predictor plot in the **Overview** pane. The selected predictor plot displays in the main pane.



2 From the main pane, to merge bins, select two or more bins for merging by using **Ctrl** + click or **Shift** + click to multiselect bins to display with blue outlines. To change your bin selection, use **Ctrl**+ click to deselect a bin.

Note The **Merge** button is active only when more than one bin is selected. Only adjacent bins can be merged for numeric or ordinal predictors. Nonadjacent bins can be merged for categorical predictors.



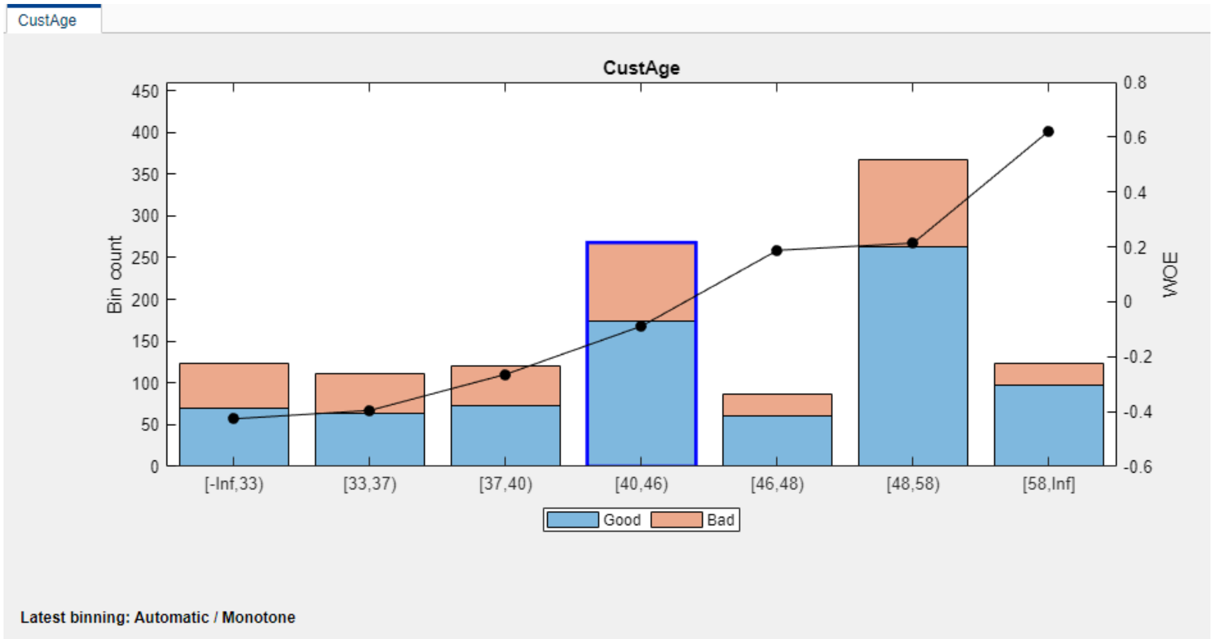
3 Click **Merge** to complete the merge operation. The plot for the selected predictor is updated with the new bin information. The details in the **Bin Information** and **Predictor Information** panes are also updated.

Change Bin Boundaries for a Single Predictor

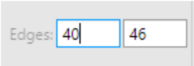
After you import data or a `creditscorecard` object into **Binning Explorer**, you can change the bin boundaries for a single predictor.

1 Click any numeric predictor plot in the **Overview** pane. The selected predictor plot displays with a blue outline and the predictor plot displays in the main pane.

2 From the main pane, click to select a specific bin where you want to change the bin dimensions. The selected bin displays with a blue outline.



3 On the **Binning Explorer** toolstrip, the **Edges** text boxes display values for the edges of the selected bin.



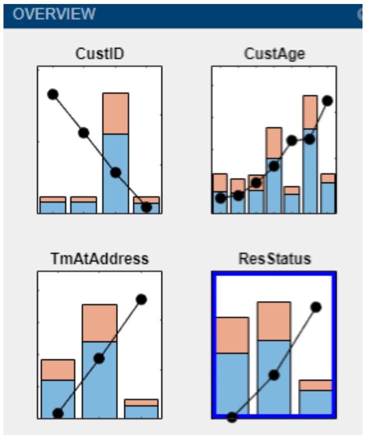
Edit the values in the **Edges** text boxes to change the selected bin's dimensions.

4 Click the main pane to complete the operation. The plot for the predictor is updated with the updated bin's dimension information. The details in the **Bin Information** and **Predictor Information** panes are also updated.

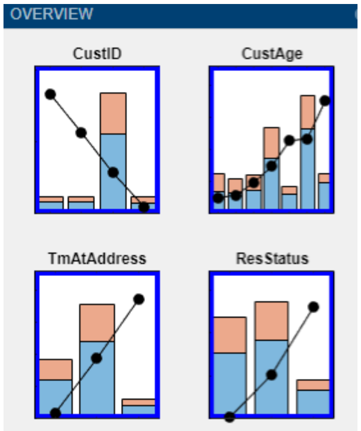
Change Bin Boundaries for Multiple Predictors

After you import data or a `creditscorecard` object into **Binning Explorer**, you can change the algorithm applied to one or more predictors and you can also redefine the number of bins.

1 From the **Overview** pane, click any predictor plot. The predictor plot displays with a blue outline.



Alternatively, select two or more predictors by using **Ctrl** + click or **Shift** + click to multiselect predictors to display with blue outlines.



2 On the **Binning Explorer** toolstrip, click **Options** to open a list of options for the **Monotone**, **Split**, **Merge**, **Equal Frequency**, and **Equal Width** algorithms. Click an option to open the associated Algorithm options dialog box. Make your selection from the respective Algorithm Options dialog box and click **OK**. The selected predictor plots are updated for the change of algorithm and the plots for the selected predictors are updated with the new bin information. The details in the **Bin Information** and **Predictor Information** panes are also updated.

Set Options for Display

Binning Explorer has options for displaying predictor plots and plot options and the associated tables displayed in **Bin Information**.

Plot Options

1 From the **Binning Explorer** toolstrip item for **Plot Options**, select any of the following predictor plot options:

- **No labels** (default)
- **Bin count**
- **% Bin level**
- **% Data level**
- **% Total count**

2 The selected label is applied to all predictor plots.

Table Options

You can set the table display options for predictor information displayed in **Bin Information**.

1 From the **Binning Explorer** toolstrip item for **Table Columns**, select any of the following options:

- **Odds**
- **WOE**

- **InfoValue**
- **Entropy**
- **Gini**
- **Chi2**
- **Members** (option is enabled for categorical predictors)

2 When selected, these options are applied to all predictors for the information displayed in **Bin Information**.

Export and Save the Binning

Binning Explorer enables you to export and save your credit scorecard binning definitions to a `creditscorecard` object.

1 Click **Export** and then click **Export Scorecard** and provide a `creditscorecard` object name. The `creditscorecard` object is saved to the MATLAB workspace.

Note If you export a previously existing `creditscorecard` object that was fit (using `fitmodel`), all fitting settings in the `creditscorecard` object are lost. You must rerun `fitmodel` on the updated `creditscorecard` object.

2 To reopen a previously saved `creditscorecard` object, click **Import Data** and select the `creditscorecard` object from the **Step 1** pane of the Import Data window.

Troubleshoot the Binning

- “Numeric Predictor Converted to Categorical Predictor Does Not Display Split Data Properly” on page 3-19
- “Predictor Plot Appears Distorted” on page 3-20

This topic shows some of the results when using **Binning Explorer** with credit scorecards that need troubleshooting. For details on the overall process of creating and developing credit scorecards, see “Overview of Binning Explorer” on page 3-2 and “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23.

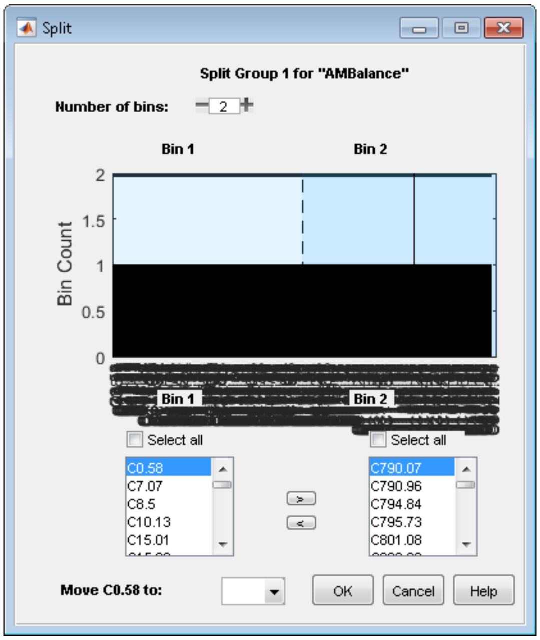
Numeric Predictor Converted to Categorical Predictor Does Not Display Split Data Properly

When you convert a numeric predictor with hundreds of values (for example, continuous data) to categorical data, the resulting data has hundreds of categories. The following example illustrates this scenario.

```
load CreditCardData
```

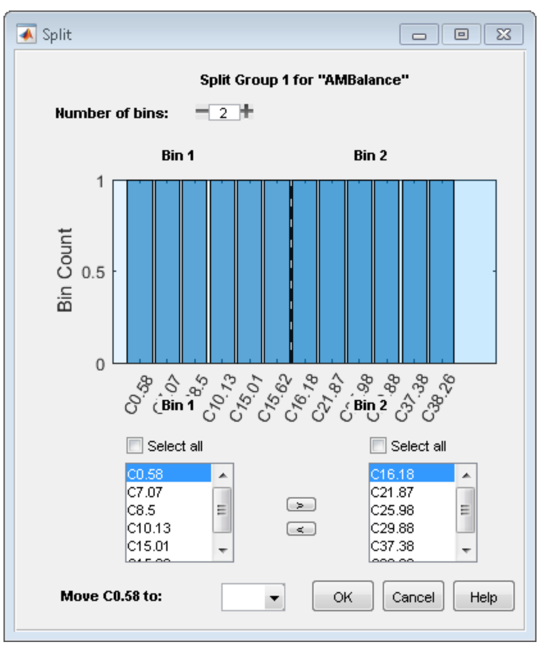
Open the **Binning Explorer** and select the numeric predictor **AMBalance** from the **Overview** pane. From the Binning Explorer toolstrip, change the predictor type to **Categorical**.

From the **Binning Explorer** toolstrip and click **Split**. The Split dialog box displays as follows:



The predictor has too many categories to display properly.

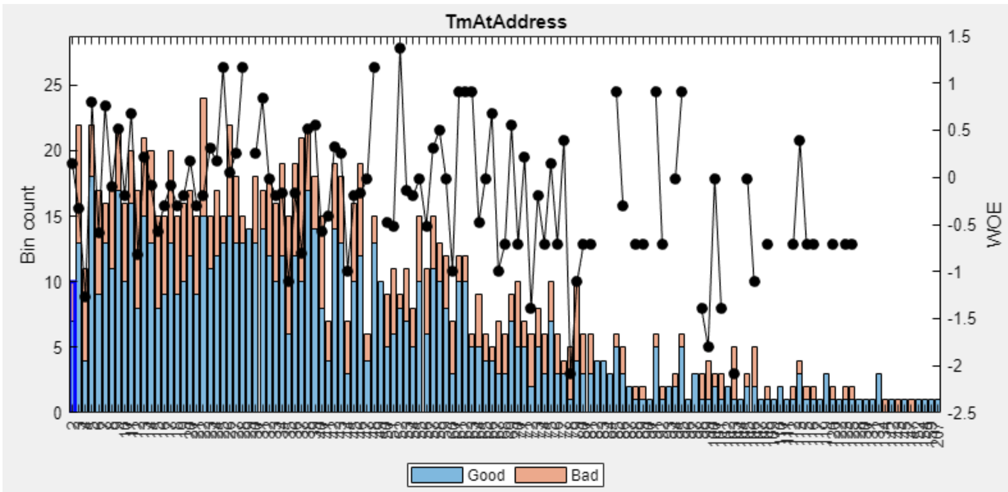
Solution: If you have a categorical predictor with a large number of categories, use the **Algorithm Options** to change the binning algorithm for that predictor to **Equal Frequency**, with the **Number of bins** set to `100` (or another smaller value). The Split dialog box then displays properly.



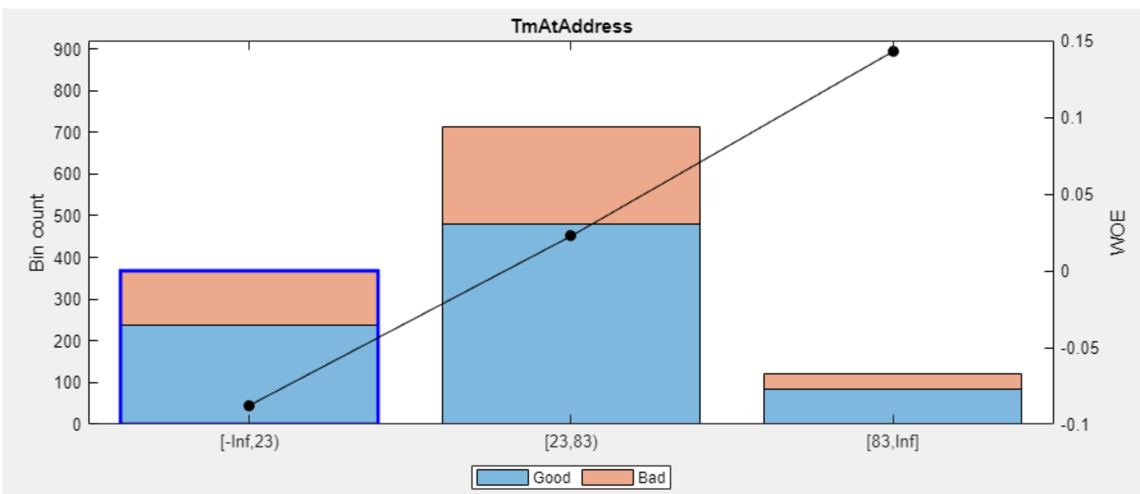
Predictor Plot Appears Distorted

When using the **Binning Explorer**, if you import data that has not been previously binned and you select **No Binning** from the Import Data window, the resulting plots might be distorted. For example, if you load the following data set into the MATLAB workspace and use **Binning Explorer** to import the data using **No Binning**, the following plot displays for the **TmAtAddress** predictor.

```
load CreditCardData
```



Solution: When you import data that has not been previously binned, select **Monotone** from the Import Data window instead. The following plot displays for the **TmAtAddress** predictor.



See Also

Apps
Binning Explorer

Classes
`creditscorecard`

Related Examples

- “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23

- “Case Study for Credit Scorecard Analysis”
- “Credit Scorecard Modeling with Missing Values”

More About

- “Overview of Binning Explorer” on page 3-2
- “Credit Scorecard Modeling Workflow”

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

Bin Data to Create Credit Scorecards Using Binning Explorer

Create a credit scorecard using the **Binning Explorer** app. Use the **Binning Explorer** to bin the data, plot the binned data information, and export a `creditscorecard` object. Then use the `creditscorecard` object with functions from Financial Toolbox to fit a logistic regression model, determine a score for the data, determine the probabilities of default, and validate the credit scorecard model using three different metrics.

Step 1. Load credit scorecard `data` into the MATLAB workspace.

Use the `CreditCardData.mat` file to load the `data` into the MATLAB workspace (using a dataset from Refaat 2011).

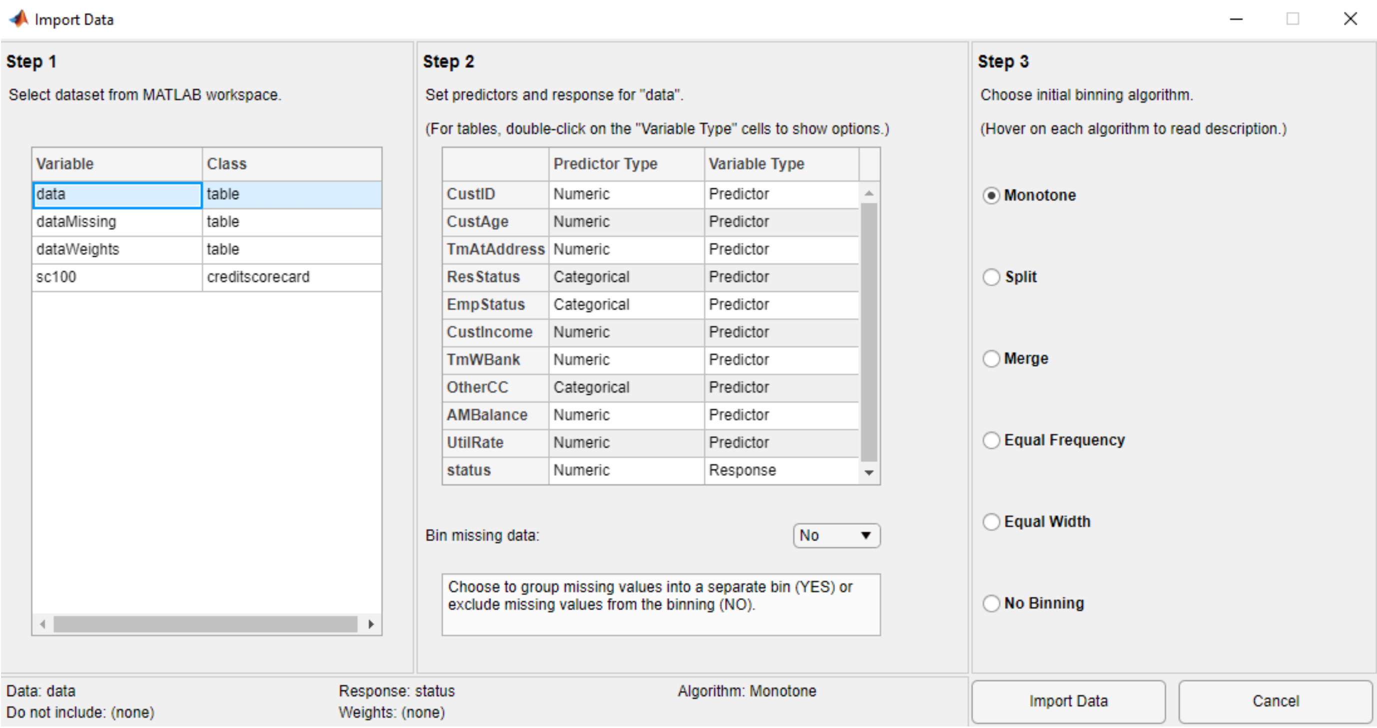
```
load CreditCardData
disp(data(1:10,:))
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance    UtilRate    status
    ______    _______    ___________    ___________    _________    __________    _______    _______    _________    ________    ______

       1        53           62         Tenant         Unknown        50000         55         Yes        1055.9       0.22        0   
       2        61           22         Home Owner     Employed       52000         25         Yes        1161.6       0.24        0   
       3        47           30         Tenant         Employed       37000         61         No         877.23       0.29        0   
       4        50           75         Home Owner     Employed       53000         20         Yes        157.37       0.08        0   
       5        68           56         Home Owner     Employed       53000         14         Yes        561.84       0.11        0   
       6        65           13         Home Owner     Employed       48000         59         Yes        968.18       0.15        0   
       7        34           32         Home Owner     Unknown        32000         26         Yes        717.82       0.02        1   
       8        50           57         Other          Employed       51000         33         No         3041.2       0.13        0   
       9        50           10         Tenant         Unknown        52000         25         Yes        115.56       0.02        1   
      10        49           30         Home Owner     Unknown        53000         23         Yes         718.5       0.17        1   
```

Step 2. Import the `data` into Binning Explorer.

Open **Binning Explorer** from the MATLAB toolstrip: On the **Apps** tab, under **Computational Finance**, click the app icon. Alternatively, you can enter `binningExplorer` on the MATLAB command line. For more information on starting the **Binning Explorer** from the command line, see “Start from MATLAB Command Line Using Data or an Existing creditscorecard Object” on page 3-5.

From the **Binning Explorer** toolstrip, select **Import Data** to open the Import Data window.



Under **Step 1**, select `data`.

Under **Step 2**, optionally set the **Variable Type** for each of the predictors. By default, the last column in the data (`'status'` in this example) is set to `'Response'`. All other variables are considered predictors. However, in this example, because `'CustID'` (customer identification number) is not a useful predictor, set the **Variable Type** column for `'CustID'` to **Do not include**.

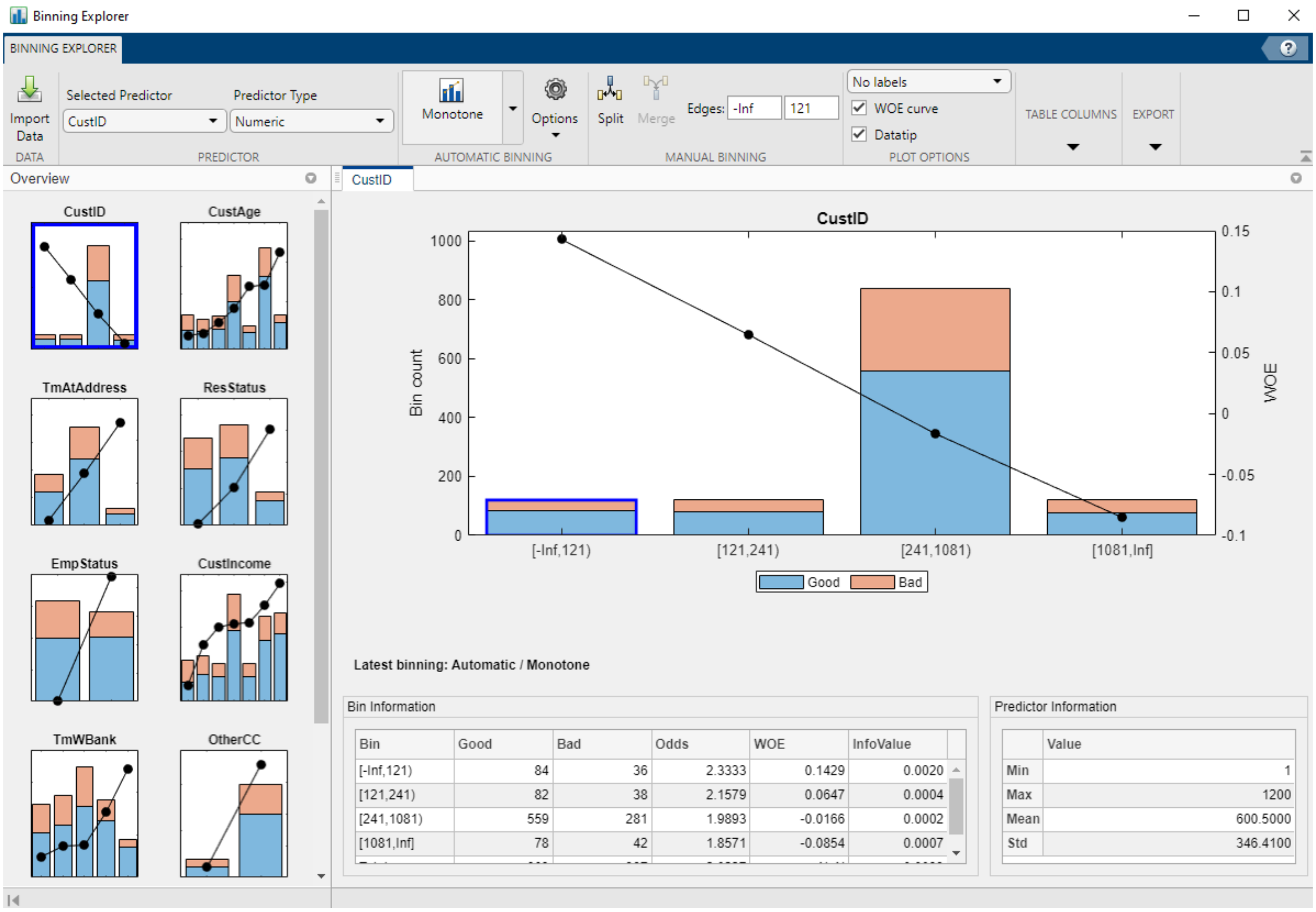
Note If the input MATLAB table contains a column for `weights`, from the **Step 2** pane, using the **Variable Type** column, click the drop-down to select **Weights**. For more information on using observation weights with a `creditscorecard` object, see "Credit Scorecard Modeling Using Observation Weights".

If the data contains missing values, from the **Step 2** pane, set **Bin missing data:** to **Yes**. For more information on working with missing data, see "Credit Scorecard Modeling with Missing Values".

Under **Step 3**, leave **Monotone** as the default initial binning algorithm.

Click **Import Data** to complete the import operation. Automatic binning using the selected algorithm is applied to all predictors as they are imported into **Binning Explorer**.

The bins are plotted and displayed for each predictor. By clicking to select an individual predictor plot from the **Overview** pane, the details for that predictor plot display in the main pane and in the **Bin Information** and **Predictor Information** panes at the bottom of the app.



Binning Explorer performs automatic binning for every predictor variable, using the default `'Monotone'` algorithm with default algorithm options. A monotonic, ideally linear trend in the Weight of Evidence (WOE) is often desirable for credit scorecards because this translates into linear points for a given predictor. WOE trends are visualized on the plots for each predictor in **Binning Explorer**.

Perform some initial data exploration. Inquire about predictor statistics for the `'ResStatus'` categorical variable.

Click the **ResStatus** plot. The **Bin Information** pane contains the “Good” and “Bad” frequencies and other bin statistics such as weight of evidence (WOE).

Bin Information

Bin	Good	Bad	Odds	WOE	InfoValue
Tenant	307	167	1.8383	-0.0956	0.0037
Home Owner	365	177	2.0621	0.0193	0.0002
Other	131	53	2.4717	0.2005	0.0059

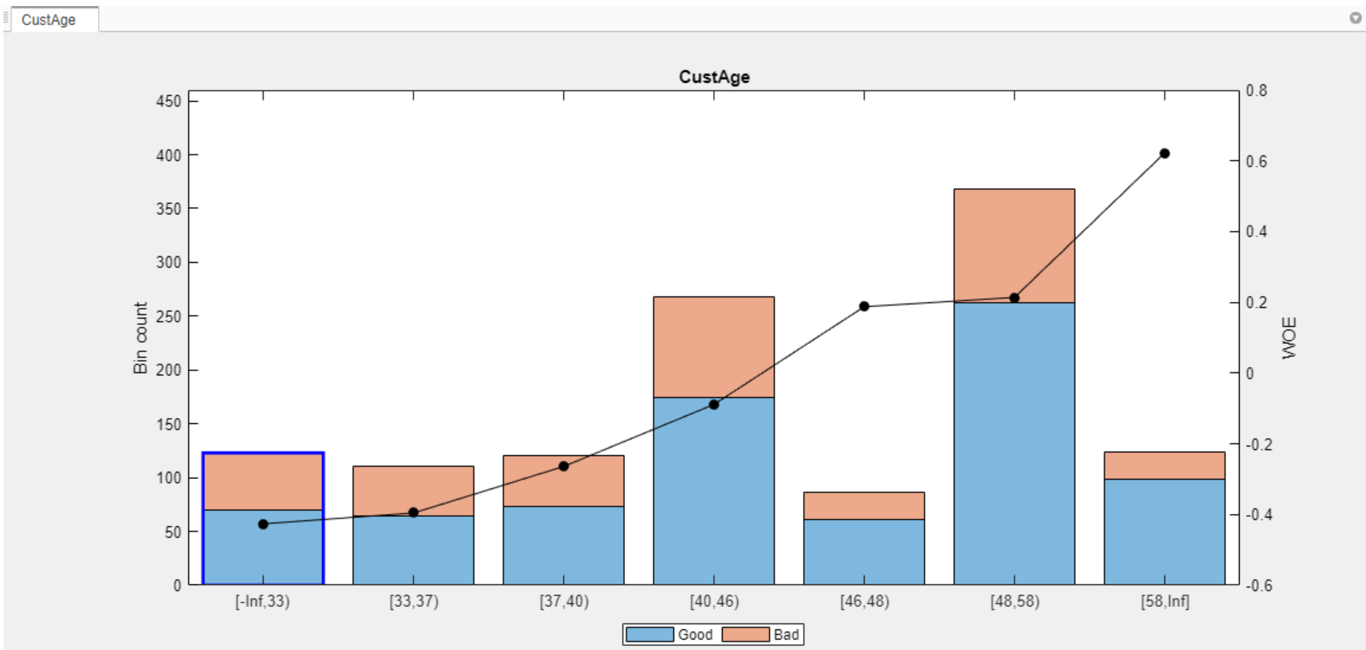
For numeric data, the same statistics are displayed. Click the **CustIncome** plot. The **Bin Information** is updated with the information about **CustIncome**.

Bin Information

Bin	Good	Bad	Odds	WOE	InfoValue
[-Inf,29000)	53	58	0.9138	-0.7946	0.0636
[29000,33000)	74	49	1.5102	-0.2922	0.0091
[33000,35000)	68	36	1.8889	-0.0684	0.0004
[35000,40000)	193	98	1.9694	-0.0267	0.0002
[40000,42000)	68	34	2.0000	-0.0113	0.0000

Step 3. Fine-tune the bins using manual binning in Binning Explorer.

Click the **CustAge** predictor plot. Notice that bins 1 and 2 have similar WOEs, as do bins 5 and 6.



Latest binning: Automatic / Monotone

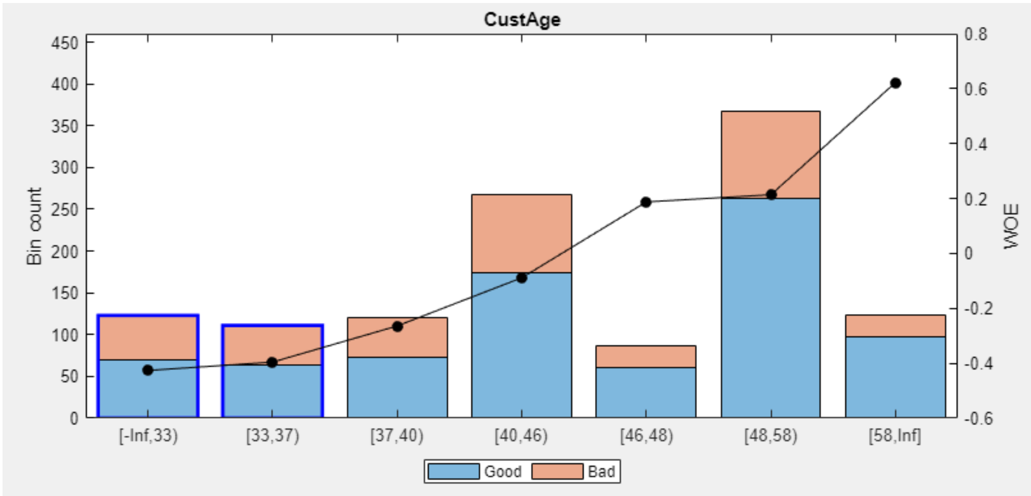
Bin Information

Bin	Good	Bad	Odds	WOE	InfoValue
[-Inf,33)	70	53	1.3208	-0.4262	0.0197
[33,37)	64	47	1.3617	-0.3957	0.0153
[37,40)	73	47	1.5532	-0.2641	0.0073
[40,46)	174	94	1.8511	-0.0887	0.0018
[46,48)	61	25	2.4400	0.1876	0.0024

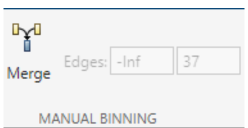
Predictor Information

	Value
Min	21.0000
Max	74.0000
Mean	45.1742
Std	9.8302

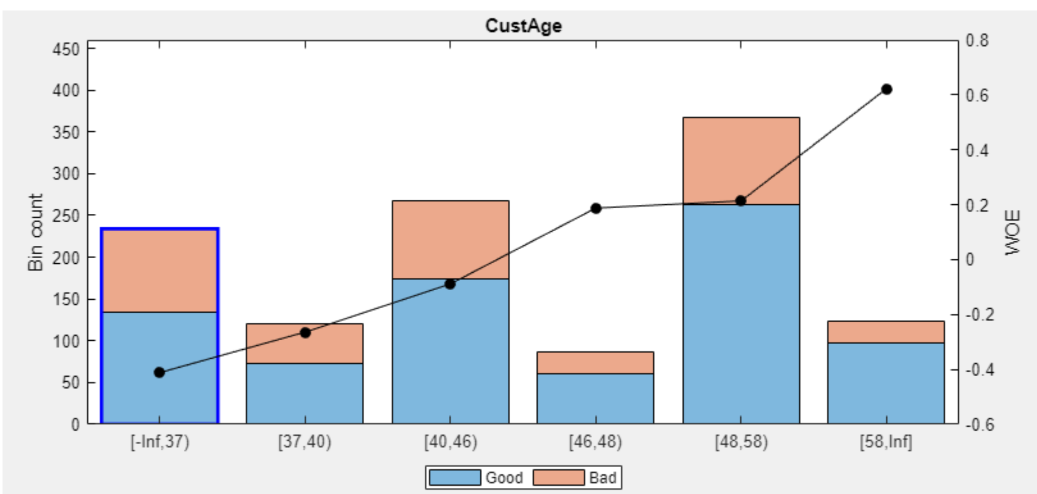
To merge bins 1 and 2, from the main pane, click **Ctrl** + click or **Shift** + click to multiselect bin 1 and 2 to display with blue outlines for merging.



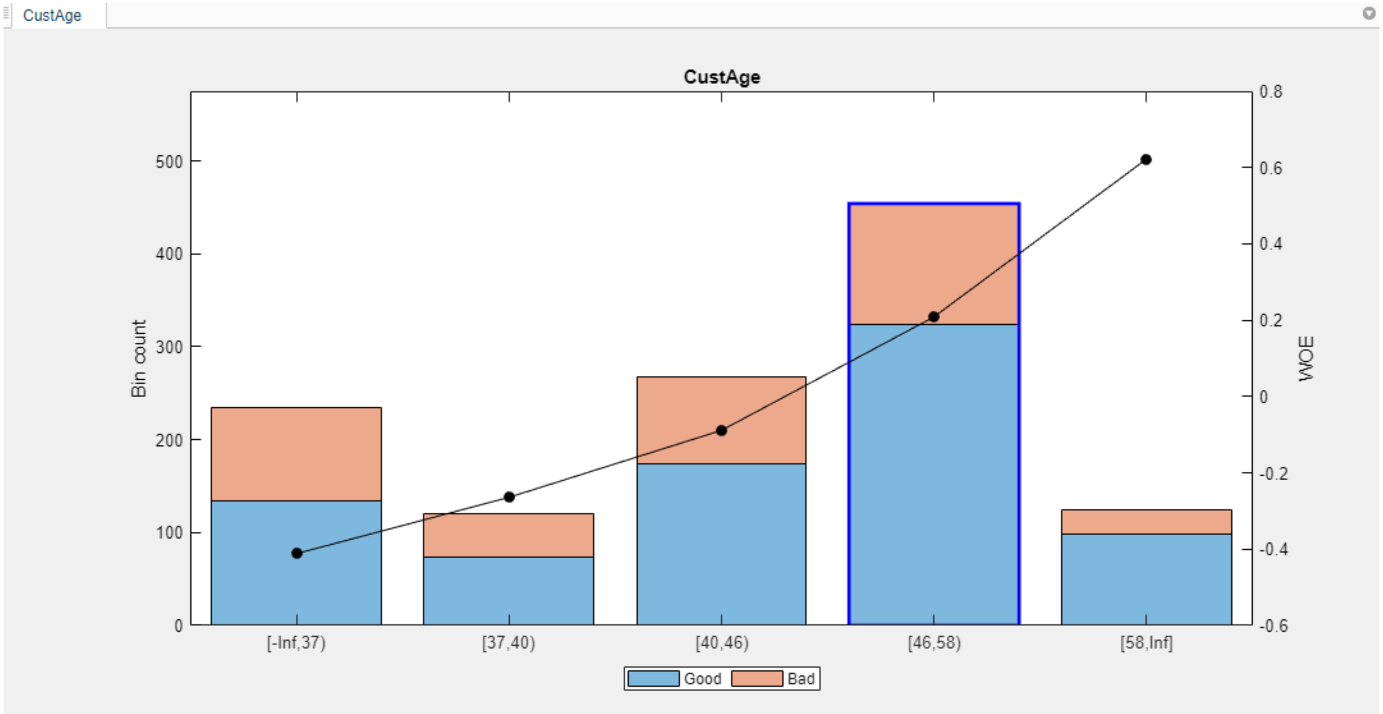
On the **Binning Explorer** toolstrip, use the read-only display for the **Edges** text boxes to verify values for the edges of the selected bins to merge.



Click **Merge** to finish merging bins 1 and 2. The **CustAge** predictor plot is updated for the new bin information and the details in the **Bin Information** and **Predictor Information** panes are also updated.



Next, merge bins 4 and 5, because they also have similar WOEs.



Latest binning: Manual

Bin Information

Bin	Good	Bad	Odds	WOE	InfoValue
[37,40)	73	47	1.5532	-0.2641	0.0073
[40,46)	174	94	1.8511	-0.0887	0.0018
[46,58)	324	130	2.4923	0.2088	0.0159
[58,Inf]	98	26	3.7692	0.6225	0.0352
Totals	803	397	2.0227	NaN	0.0951

Predictor Information

	Value
Min	21.0000
Max	74.0000
Mean	45.1742
Std	9.8302

The **CustAge** predictor plot is updated with the new bin information. The details in the **Bin Information** and **Predictor Information** panes are also updated.

Repeat this merge operation for the following bins that have similar WOEs:

- For **CustIncome**, merge bins 3, 4 and 5.
- For **TmWBank**, merge bins 2 and 3.
- For **AMBalance**, merge bins 2 and 3.

Now the bins for all predictors have close-to-linear WOE trends.

Step 4. Export the `creditscorecard` object from Binning Explorer.

After you complete your binning assignments, using **Binning Explorer**, click **Export** and then click **Export Scorecard** and provide a `creditscorecard` object name. The `creditscorecard` object (`sc`) is saved to the MATLAB workspace.

Step 5. Fit a logistic regression model.

Use the `fitmodel` function to fit a logistic regression model to the WOE data. `fitmodel` internally bins the training data, transforms it into WOE values, maps the response variable so that `'Good'` is `1`, and fits a linear logistic regression model. By default, `fitmodel` uses a stepwise procedure to determine which predictors belong in the model.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8954, Chi2Stat = 32.545914, PValue = 1.1640961e-08
2. Adding TmWBank, Deviance = 1467.3249, Chi2Stat = 23.570535, PValue = 1.2041739e-06
3. Adding AMBalance, Deviance = 1455.858, Chi2Stat = 11.466846, PValue = 0.00070848829
4. Adding EmpStatus, Deviance = 1447.6148, Chi2Stat = 8.2432677, PValue = 0.0040903428
5. Adding CustAge, Deviance = 1442.06, Chi2Stat = 5.5547849, PValue = 0.018430237
6. Adding ResStatus, Deviance = 1437.9435, Chi2Stat = 4.1164321, PValue = 0.042468555
7. Adding OtherCC, Deviance = 1433.7372, Chi2Stat = 4.2063597, PValue = 0.040272676

Generalized Linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBalance
    Distribution = Binomial

Estimated Coefficients:
                   Estimate      SE       tStat       pValue  
                   ________    _______    ______    __________

    (Intercept)     0.7024       0.064    10.975    5.0407e-28
    CustAge        0.61562     0.24783    2.4841      0.012988
    ResStatus       1.3776     0.65266    2.1107      0.034799
    EmpStatus      0.88592     0.29296     3.024     0.0024946
    CustIncome     0.69836     0.21715     3.216     0.0013001
    TmWBank          1.106     0.23266    4.7538    1.9958e-06
    OtherCC         1.0933     0.52911    2.0662      0.038806
    AMBalance       1.0437     0.32292    3.2322     0.0012285


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.42e-16
```

Step 6. Review and format scorecard points.

After fitting the logistic model, the points are unscaled by default and come directly from the combination of WOE values and model coefficients. Use the `displaypoints` function to summarize the scorecard points.

```
p1 = displaypoints(sc);
disp(p1)
```

```
    Predictors            Bin              Points  
    ____________    __________________    _________

    'CustAge'       '[-Inf,37)'            -0.15314
    'CustAge'       '[37,40)'             -0.062247
    'CustAge'       '[40,46)'              0.045763
    'CustAge'       '[46,58)'               0.22888
    'CustAge'       '[58,Inf]'              0.48354
    'ResStatus'     'Tenant'              -0.031302
    'ResStatus'     'Home Owner'            0.12697
    'ResStatus'     'Other'                 0.37652
    'EmpStatus'     'Unknown'             -0.076369
    'EmpStatus'     'Employed'              0.31456
    'CustIncome'    '[-Inf,29000)'         -0.45455
    'CustIncome'    '[29000,33000)'         -0.1037
    'CustIncome'    '[33000,42000)'        0.077768
    'CustIncome'    '[42000,47000)'         0.24406
    'CustIncome'    '[47000,Inf]'           0.43536
    'TmWBank'       '[-Inf,12)'            -0.18221
    'TmWBank'       '[12,45)'             -0.038279
    'TmWBank'       '[45,71)'               0.39569
    'TmWBank'       '[71,Inf]'              0.95074
    'OtherCC'       'No'                     -0.193
    'OtherCC'       'Yes'                   0.15868
    'AMBalance'     '[-Inf,558.88)'          0.3552
```

```
    'AMBalance'     '[558.88,1597.44)'    -0.026797
    'AMBalance'     '[1597.44,Inf]'        -0.21168
```

Use `modifybins` to give the bins more descriptive labels.

```
sc = modifybins(sc,'CustAge','BinLabels',...
{'Up to 36' '37 to 39' '40 to 45' '46 to 57' '58 and up'});

sc = modifybins(sc,'CustIncome','BinLabels',...
{'Up to 28999' '29000 to 32999' '33000 to 41999' '42000 to 46999' '47000 and up'});

sc = modifybins(sc,'TmWBank','BinLabels',...
{'Up to 11' '12 to 44' '45 to 70' '71 and up'});

sc = modifybins(sc,'AMBalance','BinLabels',...
{'Up to 558.87' '558.88 to 1597.43' '1597.44 and up'});

p1 = displaypoints(sc);
disp(p1)
```

```
     Predictors              Bin               Points  
    ____________    ____________________    _________

    'CustAge'       'Up to 36'               -0.15314
    'CustAge'       '37 to 39'              -0.062247
    'CustAge'       '40 to 45'               0.045763
    'CustAge'       '46 to 57'                0.22888
    'CustAge'       '58 and up'               0.48354
    'ResStatus'     'Tenant'                -0.031302
    'ResStatus'     'Home Owner'              0.12697
    'ResStatus'     'Other'                   0.37652
    'EmpStatus'     'Unknown'               -0.076369
    'EmpStatus'     'Employed'                0.31456
    'CustIncome'    'Up to 28999'            -0.45455
    'CustIncome'    '29000 to 32999'          -0.1037
    'CustIncome'    '33000 to 41999'         0.077768
    'CustIncome'    '42000 to 46999'          0.24406
    'CustIncome'    '47000 and up'            0.43536
    'TmWBank'       'Up to 11'               -0.18221
    'TmWBank'       '12 to 44'              -0.038279
    'TmWBank'       '45 to 70'                0.39569
    'TmWBank'       '71 and up'               0.95074
    'OtherCC'       'No'                       -0.193
    'OtherCC'       'Yes'                     0.15868
    'AMBalance'     'Up to 558.87'             0.3552
    'AMBalance'     '558.88 to 1597.43'     -0.026797
    'AMBalance'     '1597.44 and up'         -0.21168
```

Points are scaled and are also often rounded. To round and scale the points, use the `formatpoints` function. For example, you can set a target level of points corresponding to a target odds level and also set the required points-to-double-the-odds (PDO).

```
TargetPoints = 500;
TargetOdds = 2;
PDO = 50; % Points to double the odds

sc = formatpoints(sc,'PointsOddsAndPDO',[TargetPoints TargetOdds PDO]);
p2 = displaypoints(sc);
disp(p2)
```

```
    Predictors              Bin              Points
    ____________    ____________________    ______

    'CustAge'       'Up to 36'              53.239
    'CustAge'       '37 to 39'              59.796
```

```
'CustAge'       '40 to 45'              67.587
'CustAge'       '46 to 57'              80.796
'CustAge'       '58 and up'             99.166
'ResStatus'     'Tenant'                62.028
'ResStatus'     'Home Owner'            73.445
'ResStatus'     'Other'                 91.446
'EmpStatus'     'Unknown'               58.777
'EmpStatus'     'Employed'              86.976
'CustIncome'    'Up to 28999'           31.497
'CustIncome'    '29000 to 32999'        56.805
'CustIncome'    '33000 to 41999'        69.896
'CustIncome'    '42000 to 46999'        81.891
'CustIncome'    '47000 and up'           95.69
'TmWBank'       'Up to 11'              51.142
'TmWBank'       '12 to 44'              61.524
'TmWBank'       '45 to 70'              92.829
'TmWBank'       '71 and up'             132.87
'OtherCC'       'No'                    50.364
'OtherCC'       'Yes'                   75.732
'AMBalance'     'Up to 558.87'          89.908
'AMBalance'     '558.88 to 1597.43'     62.353
'AMBalance'     '1597.44 and up'        49.016
```

Step 7. Score the data.

Use the `score` function to compute the scores for the training data. You can also pass an optional `data` input to `score`, for example, validation data. The points per predictor for each customer are provided as an optional output.

```
[Scores,Points] = score(sc);
disp(Scores(1:10))
disp(Points(1:10,:))
```

```
  528.2044
  554.8861
  505.2406
  564.0717
  554.8861
  586.1904
  441.8755
  515.8125
  524.4553
  508.3169

    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBalance
    _______    _________    _________    __________    _______    _______    _________

    80.796     62.028       58.777        95.69        92.829     75.732     62.353   
    99.166     73.445       86.976        95.69        61.524     75.732     62.353   
    80.796     62.028       86.976       69.896        92.829     50.364     62.353   
    80.796     73.445       86.976        95.69        61.524     75.732     89.908   
    99.166     73.445       86.976        95.69        61.524     75.732     62.353   
    99.166     73.445       86.976        95.69        92.829     75.732     62.353   
    53.239     73.445       58.777       56.805        61.524     75.732     62.353   
    80.796     91.446       86.976        95.69        61.524     50.364     49.016   
    80.796     62.028       58.777        95.69        61.524     75.732     89.908   
    80.796     73.445       58.777        95.69        61.524     75.732     62.353   
```

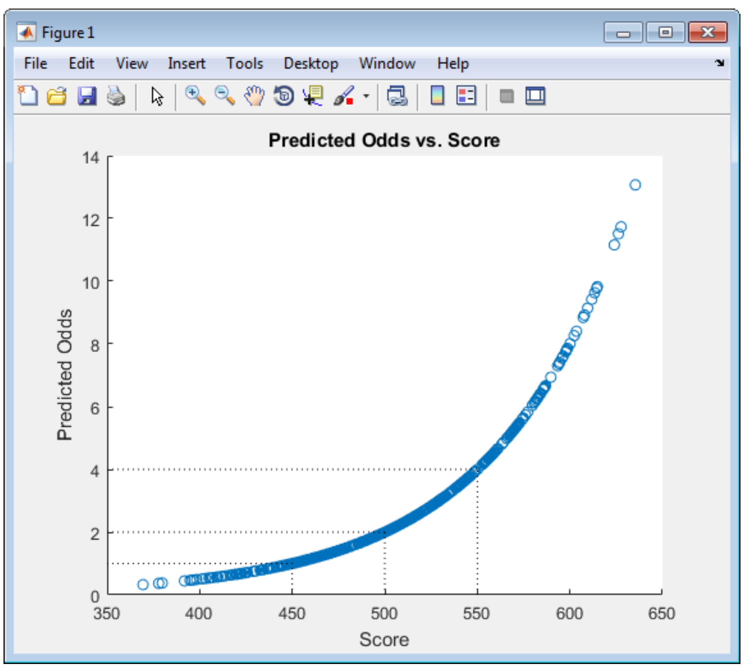
Step 8. Calculate the probability of default.

To calculate the probability of default, use the `probdefault` function.

```
pd = probdefault(sc);
```

Define the probability of being “Good” and plot the predicted odds versus the formatted scores. Visually analyze that the target points and target odds match and that the points-to-double-the-odds (PDO) relationship holds.

```
ProbGood = 1-pd;
PredictedOdds = ProbGood./pd;

figure
scatter(Scores,PredictedOdds)
title('Predicted Odds vs. Score')
xlabel('Score')
ylabel('Predicted Odds')

hold on

xLimits = xlim;
yLimits = ylim;

% Target points and odds
plot([TargetPoints TargetPoints],[yLimits(1) TargetOdds],'k:')
plot([xLimits(1) TargetPoints],[TargetOdds TargetOdds],'k:')

% Target points plus PDO
plot([TargetPoints+PDO TargetPoints+PDO],[yLimits(1) 2*TargetOdds],'k:')
plot([xLimits(1) TargetPoints+PDO],[2*TargetOdds 2*TargetOdds],'k:')

% Target points minus PDO
plot([TargetPoints-PDO TargetPoints-PDO],[yLimits(1) TargetOdds/2],'k:')
plot([xLimits(1) TargetPoints-PDO],[TargetOdds/2 TargetOdds/2],'k:')

hold off
```



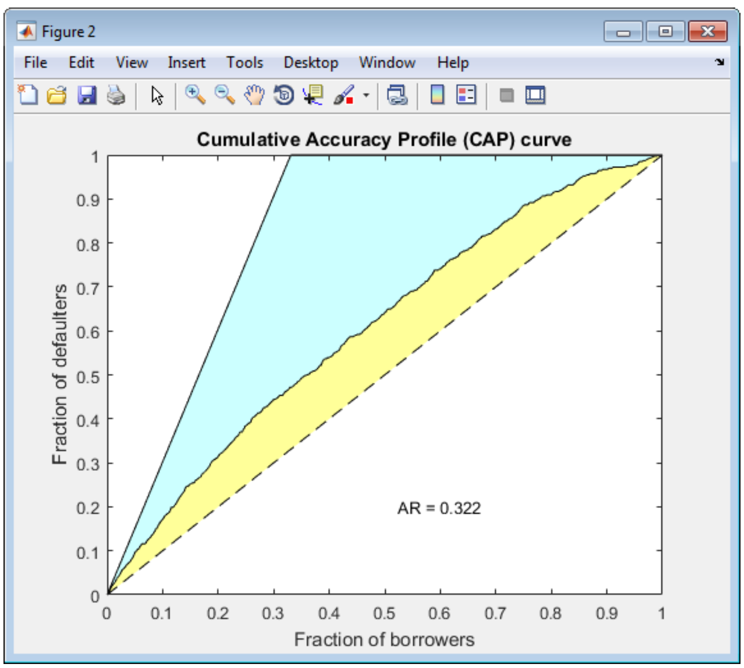
Step 9. Validate the credit scorecard model using the CAP, ROC, and Kolmogorov-Smirnov statistic

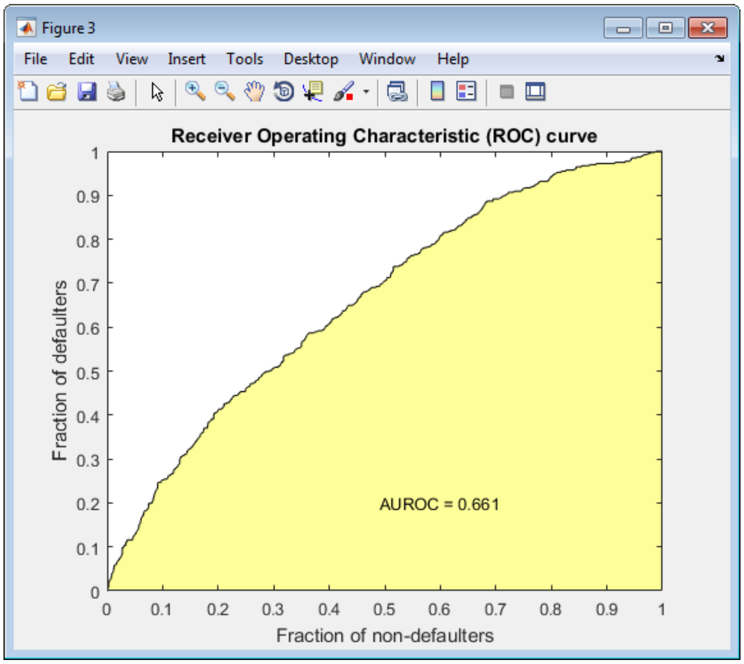
The `creditscorecard` object supports three validation methods, the Cumulative Accuracy Profile (CAP), the Receiver Operating Characteristic (ROC), and the Kolmogorov-Smirnov (KS) statistic. For more information on CAP, ROC, and KS, see `validatemodel`.

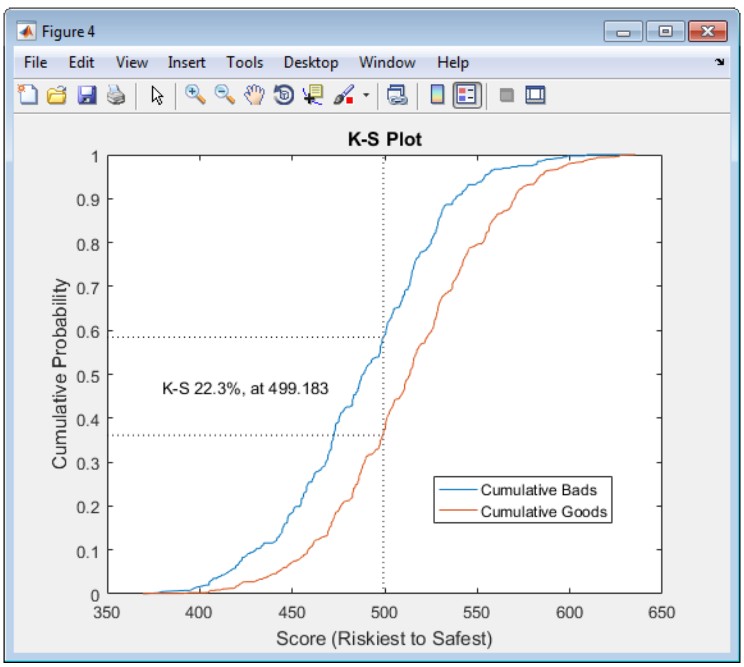
```
[Stats,T] = validatemodel(sc,'Plot',{'CAP','ROC','KS'});
disp(Stats)
disp(T(1:15,:))
```

Measure	Value
'Accuracy Ratio'	0.32225
'Area under ROC curve'	0.66113
'KS statistic'	0.22324
'KS score'	499.18

Scores	ProbDefault	TrueBads	FalseBads	TrueGoods	FalseGoods	Sensitivity	FalseAlarm	PctObs
369.4	0.7535	0	1	802	397	0	0.0012453	0.00083333
377.86	0.73107	1	1	802	396	0.0025189	0.0012453	0.0016667
379.78	0.7258	2	1	802	395	0.0050378	0.0012453	0.0025
391.81	0.69139	3	1	802	394	0.0075567	0.0012453	0.0033333
394.77	0.68259	3	2	801	394	0.0075567	0.0024907	0.0041667
395.78	0.67954	4	2	801	393	0.010076	0.0024907	0.005
396.95	0.67598	5	2	801	392	0.012594	0.0024907	0.0058333
398.37	0.67167	6	2	801	391	0.015113	0.0024907	0.0066667
401.26	0.66276	7	2	801	390	0.017632	0.0024907	0.0075
403.23	0.65664	8	2	801	389	0.020151	0.0024907	0.0083333
405.09	0.65081	8	3	800	389	0.020151	0.003736	0.0091667
405.15	0.65062	11	5	798	386	0.027708	0.0062267	0.013333
405.37	0.64991	11	6	797	386	0.027708	0.007472	0.014167
406.18	0.64735	12	6	797	385	0.030227	0.007472	0.015
407.14	0.64433	13	6	797	384	0.032746	0.007472	0.015833







See Also

`creditscorecard` | `screenpredictors` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel` | `compactCreditScorecard`

Related Examples

- "Common Binning Explorer Tasks" on page 3-4
- "Credit Scorecard Modeling with Missing Values"
- "Feature Screening with screenpredictors" on page 3-64
- "Troubleshooting Credit Scorecard Results"
- "Credit Rating by Bagging Decision Trees"
- "Stress Testing of Consumer Credit Default Probabilities Using Panel Data" on page 3-36

More About

- "Overview of Binning Explorer" on page 3-2
- "About Credit Scorecards"
- "Credit Scorecard Modeling Workflow"
- Monotone Adjacent Pooling Algorithm (MAPA)
- "Credit Scorecard Modeling Using Observation Weights"

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

Stress Testing of Consumer Credit Default Probabilities Using Panel Data

This example shows how to work with consumer (retail) credit panel data to visualize observed default rates at different levels. It also shows how to fit a model to predict probabilities of default (PD) and lifetime PD values, and perform a stress-testing analysis.

The panel data set of consumer loans enables you to identify default rate patterns for loans of different ages, or years on books. You can use information about a score group to distinguish default rates for different score levels. In addition, you can use macroeconomic information to assess how the state of the economy affects consumer loan default rates.

A standard logistic regression model, a type of generalized linear model, is fitted to the retail credit panel data with and without macroeconomic predictors, using `fitLifetimePDModel` from Risk Management Toolbox™. Although the same model can be fitted using the `fitglm` function from Statistics and Machine Learning Toolbox™, the lifetime probability of default (PD) version of the model is designed for credit applications, and supports lifetime PD prediction and model validation tools, including the discrimination and accuracy plots shown in this example. The example also describes how to fit a more advanced model to account for panel data effects, a generalized linear mixed effects model. However, the panel effects are negligible for the data set in this example and the standard logistic model is preferred for efficiency.

The logistic regression model predicts probabilities of default for all score levels, years on books, and macroeconomic variable scenarios. There is a brief discussion on how to predict lifetime PD values, with pointers to additional functionality. The example shows model discrimination and model accuracy tools to validate and compare models. In the last section of this example, the logistic model is used for a stress-testing analysis, the model predicts probabilities of default for a given baseline, as well as default probabilities for adverse and severely adverse macroeconomic scenarios.

For additional information, refer to “Overview of Lifetime Probability of Default Models” on page 1-25. See also the example “Modeling Probabilities of Default with Cox Proportional Hazards” on page 4-28, which follows the same workflow but uses Cox regression instead of logistic regression and also has information on the computation of lifetime PD and lifetime Expected Credit Loss (ECL).

Panel Data Description

The main data set (`data`) contains the following variables:

- `ID`: Loan identifier.
- `ScoreGroup`: Credit score at the beginning of the loan, discretized into three groups: `High Risk`, `Medium Risk`, and `Low Risk`.
- `YOB`: Years on books.
- `Default`: Default indicator. This is the response variable.
- `Year`: Calendar year.

There is also a small data set (`dataMacro`) with macroeconomic data for the corresponding calendar years:

- `Year`: Calendar year.
- `GDP`: Gross domestic product growth (year over year).

- `Market`: Market return (year over year).

The variables `YOB`, `Year`, `GDP`, and `Market` are observed at the end of the corresponding calendar year. The score group is a discretization of the original credit score when the loan started. A value of `1` for `Default` means that the loan defaulted in the corresponding calendar year.

There is also a third data set (`dataMacroStress`) with baseline, adverse, and severely adverse scenarios for the macroeconomic variables. This table is used for the stress-testing analysis.

This example uses simulated data, but the same approach has been successfully applied to real data sets.

Load the Panel Data

Load the data and view the first 10 and last 10 rows of the table. The panel data is stacked, in the sense that observations for the same ID are stored in contiguous rows, creating a tall, thin table. The panel is unbalanced, because not all IDs have the same number of observations.

```
load RetailCreditPanelData.mat
```

```
fprintf('\nFirst ten rows:\n')
```

```
First ten rows:
```

```
disp(data(1:10,:))
```

```
    ID     ScoreGroup     YOB    Default    Year
    __    ___________    ___    _______    ____

    1      Low Risk        1       0        1997
    1      Low Risk        2       0        1998
    1      Low Risk        3       0        1999
    1      Low Risk        4       0        2000
    1      Low Risk        5       0        2001
    1      Low Risk        6       0        2002
    1      Low Risk        7       0        2003
    1      Low Risk        8       0        2004
    2      Medium Risk     1       0        1997
    2      Medium Risk     2       0        1998
```

```
fprintf('Last ten rows:\n')
```

```
Last ten rows:
```

```
disp(data(end-9:end,:))
```

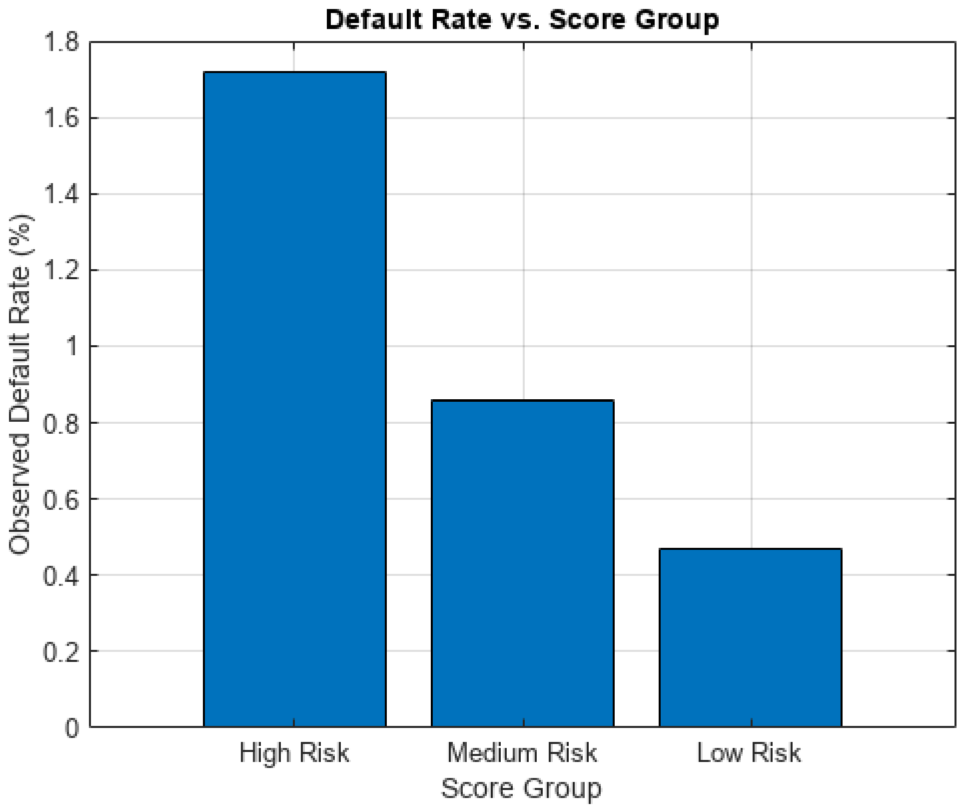
```
      ID       ScoreGroup     YOB    Default    Year
    _____    ___________    ___    _______    ____

    96819    High Risk       6        0       2003
    96819    High Risk       7        0       2004
    96820    Medium Risk     1        0       1997
    96820    Medium Risk     2        0       1998
    96820    Medium Risk     3        0       1999
    96820    Medium Risk     4        0       2000
    96820    Medium Risk     5        0       2001
    96820    Medium Risk     6        0       2002
    96820    Medium Risk     7        0       2003
    96820    Medium Risk     8        0       2004
```

```
nRows = height(data);
UniqueIDs = unique(data.ID);
nIDs = length(UniqueIDs);

fprintf('Total number of IDs: %d\n',nIDs)

Total number of IDs: 96820

fprintf('Total number of rows: %d\n',nRows)

Total number of rows: 646724
```

Default Rates by Score Groups and Years on Books

Use the credit score group as a grouping variable to compute the observed default rate for each score group. For this, use the `groupsummary` function to compute the mean of the `Default` variable, grouping by the `ScoreGroup` variable. Plot the results on a bar chart. As expected, the default rate goes down as the credit quality improves.

```
DefRateByScore = groupsummary(data,'ScoreGroup','mean','Default');
NumScoreGroups = height(DefRateByScore);

disp(DefRateByScore)

    ScoreGroup     GroupCount    mean_Default
    ___________    __________    ____________

    High Risk      2.0999e+05      0.017167
    Medium Risk    2.1743e+05     0.0086006
    Low Risk        2.193e+05     0.0046784

bar(DefRateByScore.ScoreGroup,DefRateByScore.mean_Default*100)
title('Default Rate vs. Score Group')
xlabel('Score Group')
ylabel('Observed Default Rate (%)')
grid on
```



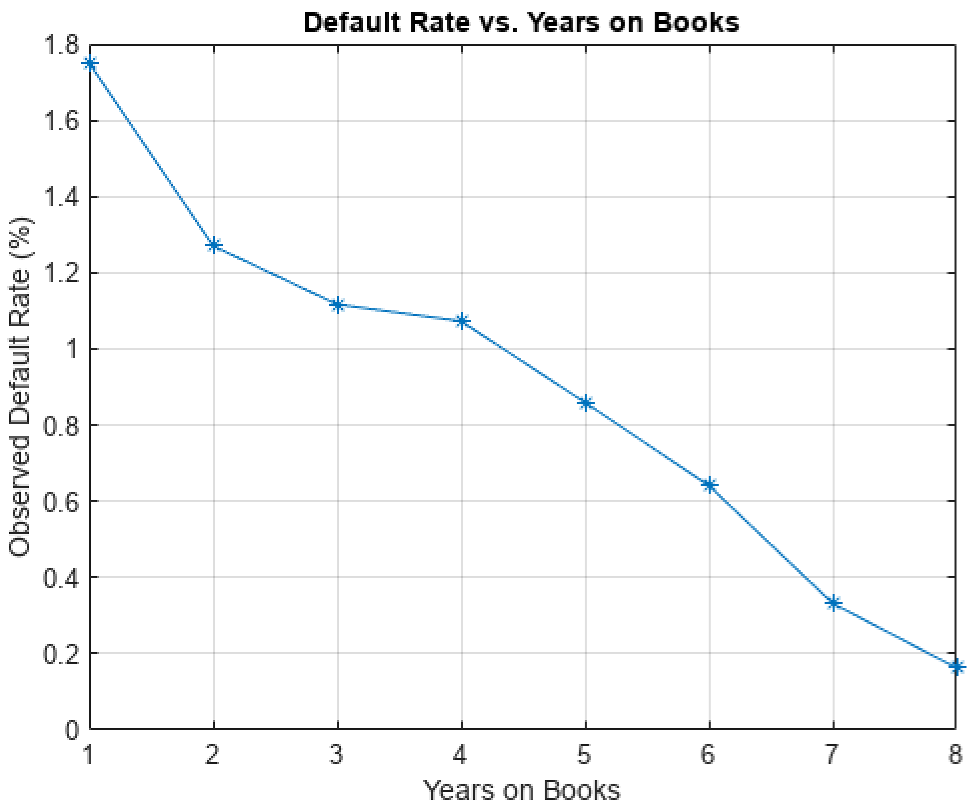
Next, compute default rates grouping by years on books (represented by the `YOB` variable). The resulting rates are conditional one-year default rates. For example, the default rate for the third year on books is the proportion of loans defaulting in the third year, relative to the number of loans that are in the portfolio past the second year. In other words, the default rate for the third year is the number of rows with `YOB = 3` and `Default` = 1, divided by the number of rows with `YOB = 3`.

Plot the results. There is a clear downward trend, with default rates going down as the number of years on books increases. Years three and four have similar default rates. However, it is unclear from this plot whether this is a characteristic of the loan product or an effect of the macroeconomic environment.

```
DefRateByYOB = groupsummary(data,'YOB','mean','Default');
NumYOB = height(DefRateByYOB);

disp(DefRateByYOB)
```

```
    YOB    GroupCount    mean_Default
    ___    __________    ____________

     1       96820         0.017507  
     2       94535         0.012704  
     3       92497         0.011168  
     4       91068         0.010728  
     5       89588        0.0085949  
     6       88570         0.006413  
     7       61689        0.0033231  
     8       31957        0.0016272  
```

```
plot(double(DefRateByYOB.YOB),DefRateByYOB.mean_Default*100,'-*')
title('Default Rate vs. Years on Books')
xlabel('Years on Books')
ylabel('Observed Default Rate (%)')
grid on
```

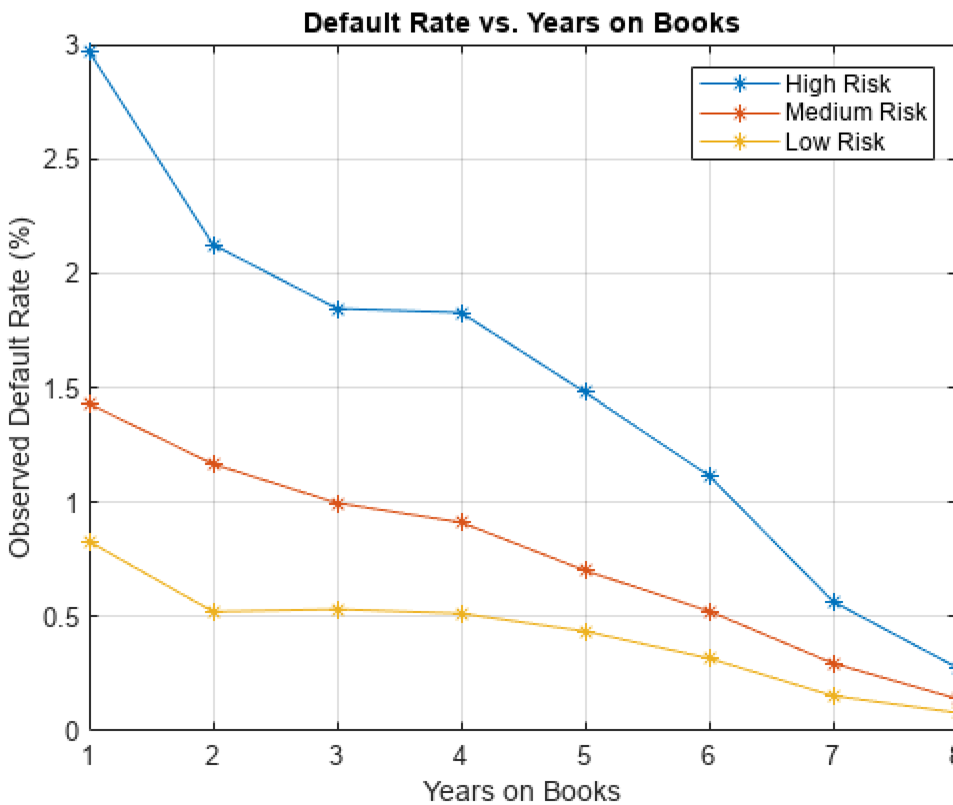


Now, group both by the score group and number of years on books and then plot the results. The plot shows that all score groups behave similarly as time progresses, with a general downward trend. Years three and four are an exception to the downward trend: the rates flatten for the `High Risk` group, and go up in year three for the `Low Risk` group.

```
DefRateByScoreYOB = groupsummary(data,{'ScoreGroup','YOB'},'mean','Default');

% Display output table to show the way it is structured
% Display only the first 10 rows, for brevity
disp(DefRateByScoreYOB(1:10,:))
```

```
    ScoreGroup     YOB    GroupCount    mean_Default
    ___________    ___    __________    ____________

    High Risk       1       32601         0.029692  
    High Risk       2       31338         0.021252  
    High Risk       3       30138         0.018448  
    High Risk       4       29438         0.018276  
    High Risk       5       28661         0.014794  
    High Risk       6       28117         0.011168  
    High Risk       7       19606        0.0056615  
    High Risk       8       10094        0.0027739  
```

```
    Medium Risk     1       32373          0.014302
    Medium Risk     2       31775          0.011676

DefRateByScoreYOB2 = reshape(DefRateByScoreYOB.mean_Default,...
   NumYOB,NumScoreGroups);

plot(DefRateByScoreYOB2*100,'-*')
title('Default Rate vs. Years on Books')
xlabel('Years on Books')
ylabel('Observed Default Rate (%)')
legend(categories(data.ScoreGroup))
grid on
```



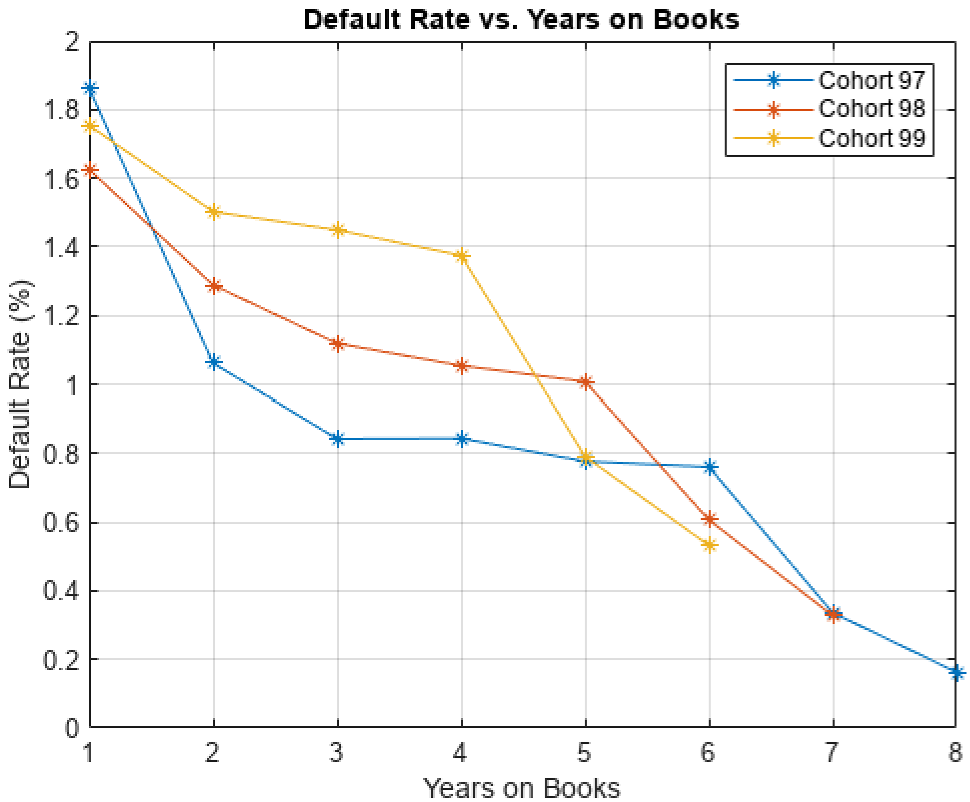
Years on Books Versus Calendar Years

The data contains three cohorts, or vintages: loans started in 1997, 1998, and 1999. No loan in the panel data started after 1999.

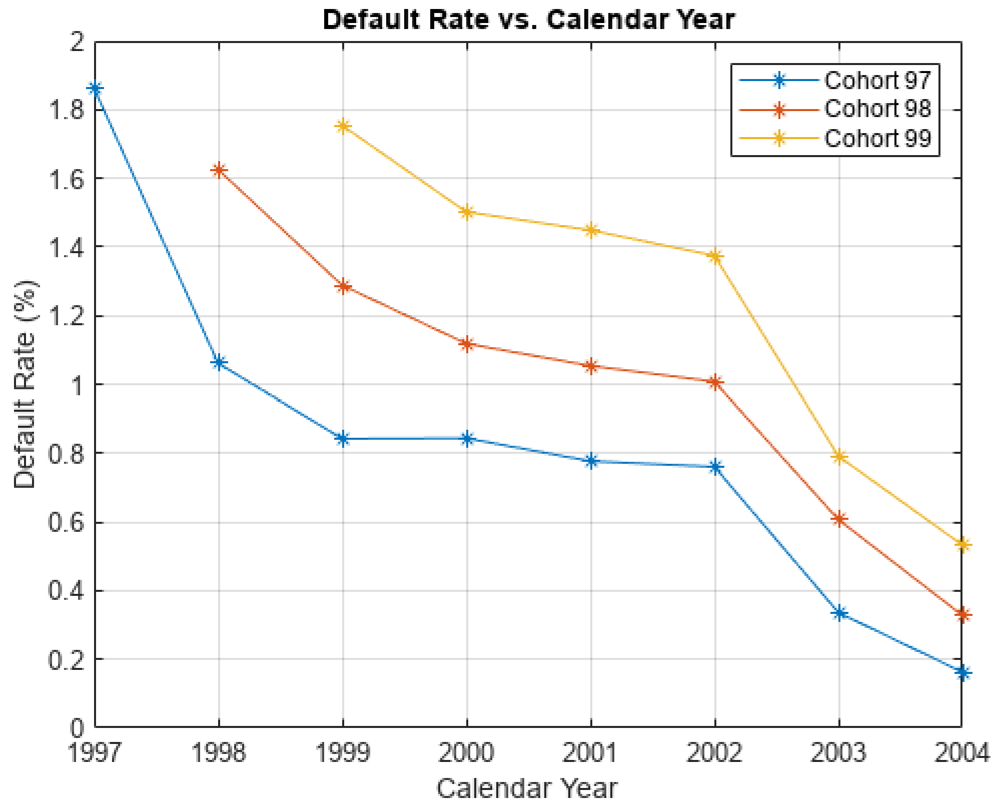
This section shows how to visualize the default rate for each cohort separately. The default rates for all cohorts are plotted, both against the number of years on books and against the calendar year. Patterns in the years on books suggest the loan product characteristics. Patterns in the calendar years suggest the influence of the macroeconomic environment.

From years two through four on books, the curves show different patterns for the three cohorts. When plotted against the calendar year, however, the three cohorts show similar behavior from 2000 through 2002. The curves flatten during that period.

```
% Get IDs of 1997, 1998, and 1999 cohorts
IDs1997 = data.ID(data.YOB==1&data.Year==1997);
IDs1998 = data.ID(data.YOB==1&data.Year==1998);
IDs1999 = data.ID(data.YOB==1&data.Year==1999);
% IDs2000AndUp is unused, it is only computed to show that this is empty,
% no loans started after 1999
IDs2000AndUp = data.ID(data.YOB==1&data.Year>1999);

% Get default rates for each cohort separately
ObsDefRate1997 = groupsummary(data(ismember(data.ID,IDs1997),:),...
    'YOB','mean','Default');

ObsDefRate1998 = groupsummary(data(ismember(data.ID,IDs1998),:),...
    'YOB','mean','Default');

ObsDefRate1999 = groupsummary(data(ismember(data.ID,IDs1999),:),...
    'YOB','mean','Default');

% Plot against the years on books
plot(ObsDefRate1997.YOB,ObsDefRate1997.mean_Default*100,'-*')
hold on
plot(ObsDefRate1998.YOB,ObsDefRate1998.mean_Default*100,'-*')
plot(ObsDefRate1999.YOB,ObsDefRate1999.mean_Default*100,'-*')
hold off
title('Default Rate vs. Years on Books')
xlabel('Years on Books')
ylabel('Default Rate (%)')
legend('Cohort 97','Cohort 98','Cohort 99')
grid on
```




```
% Plot against the calendar year
Year = unique(data.Year);

plot(Year,ObsDefRate1997.mean_Default*100,'-*')
hold on
plot(Year(2:end),ObsDefRate1998.mean_Default*100,'-*')
plot(Year(3:end),ObsDefRate1999.mean_Default*100,'-*')
hold off
title('Default Rate vs. Calendar Year')
xlabel('Calendar Year')
ylabel('Default Rate (%)')
legend('Cohort 97','Cohort 98','Cohort 99')
grid on
```



Model of Default Rates Using Score Group and Years on Books

After you visualize the data, you can build predictive models for the default rates.

Split the panel data into training and testing sets, defining these sets based on ID numbers.

```
NumTraining = floor(0.6*nIDs);

rng('default'); % for reproducibility
TrainIDInd = randsample(nIDs,NumTraining);
TrainDataInd = ismember(data.ID,UniqueIDs(TrainIDInd));
TestDataInd = ~TrainDataInd;
```

The first model uses only score group and number of years on books as predictors of the default rate *p*. The odds of defaulting are defined as *p/(1-p)*. The logistic model relates the logarithm of the odds, or *log odds*, to the predictors as follows:

$$\log\left(\frac{p}{1-p}\right) = a_H + a_M 1_M + a_L 1_L + b_{YOB} YOB + \epsilon$$

1M is an indicator with a value `1` for `Medium Risk` loans and `0` otherwise, and similarly for *1L* for `Low Risk` loans. This is a standard way of handling a categorical predictor such as `ScoreGroup`. There is effectively a different constant for each risk level: *aH* for `High Risk`, *aH+aM* for `Medium Risk`, and *aH+aL* for `Low Risk`.

```
ModelNoMacro = fitLifetimePDModel(data(TrainDataInd,:),'logistic',...
   'ModelID','No Macro','Description','Logistic model with YOB and score group, but no macro var
```

```
   'IDVar','ID','LoanVars','ScoreGroup','AgeVar','YOB','ResponseVar','Default');
disp(ModelNoMacro.UnderlyingModel)
```

```
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB
    Distribution = Binomial

Estimated Coefficients:
                             Estimate       SE         tStat       pValue
                             ________    ________    _______    ___________

    (Intercept)               -3.2453    0.033768    -96.106              0
    ScoreGroup_Medium Risk    -0.7058    0.037103    -19.023     1.1014e-80
    ScoreGroup_Low Risk       -1.2893    0.045635    -28.253    1.3076e-175
    YOB                      -0.22693    0.008437    -26.897    2.3578e-159


388018 observations, 388014 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.83e+03, p-value = 0
```

For any row in the data, the value of *p* is not observed, only a `0` or `1` default indicator is observed. The calibration finds model coefficients, and the predicted values of *p* for individual rows can be recovered with the `predict` function.

The `Intercept` coefficient is the constant for the `High Risk` level (the *aH* term), and the `ScoreGroup_Medium Risk` and `ScoreGroup_Low Risk` coefficients are the adjustments for `Medium Risk` and `Low Risk` levels (the *aM* and *aL* terms).

The default probability *p* and the log odds (the left side of the model) move in the same direction when the predictors change. Therefore, because the adjustments for `Medium Risk` and `Low Risk` are negative, the default rates are lower for better risk levels, as expected. The coefficient for number of years on books is also negative, consistent with the overall downward trend for number of years on books observed in the data.

An alternative way to fit the model is using the `fitglm` function from Statistics and Machine Learning Toolbox™. The formula above is expressed as

```
Default ~ 1 + ScoreGroup + YOB
```

The `1 + ScoreGroup` terms account for the baseline constant and the adjustments for risk level. Set the optional argument `Distribution` to `binomial` to indicate that a logistic model is desired (that is, a model with log odds on the left side), as follows:

```
ModelNoMacro = fitglm(data(TrainDataInd,:), 'Default ~ 1 + ScoreGroup +
YOB','Distribution','binomial');
```

As mentioned in the introduction, the advantage of the lifetime PD version of the model fitted with `fitLifetimePDModel` is that it is designed for credit applications, and it can predict lifetime PD and supports model validation tools, including the discrimination and accuracy plots. For more information, see “Overview of Lifetime Probability of Default Models” on page 1-25.

To account for panel data effects, a more advanced model using mixed effects can be fitted using the `fitglme` function from Statistics and Machine Learning Toolbox™. Although this model is not fitted in this example, the code is very similar:

```
ModelNoMacro = fitglme(data(TrainDataInd,:),'Default ~ 1 + ScoreGroup + YOB + 
(1|ID)','Distribution','binomial');
```

The `(1|ID)` term in the formula adds a *random effect* to the model. This effect is a predictor whose values are not given in the data, but fitted together with the model coefficients. A random value is fit for each ID. This additional fitting requirement substantially increases the computational time to fit the model in this case, because of the very large number of IDs. For the panel data set in this example, the random term has a negligible effect. The variance of the random effects is very small and the model coefficients barely change when the random effect is introduced. The simpler logistic regression model is preferred, because it is faster to fit and to predict, and the default rates predicted with both models are essentially the same.

Predict the probability of default for training and testing data. The `predict` function predicts conditional PD values, row by row. We store the data to compare the predictions against the macro model in the next section.

```
data.PDNoMacro = zeros(height(data),1);

% Predict in-sample
data.PDNoMacro(TrainDataInd) = predict(ModelNoMacro,data(TrainDataInd,:));
% Predict out-of-sample
data.PDNoMacro(TestDataInd) = predict(ModelNoMacro,data(TestDataInd,:));
```

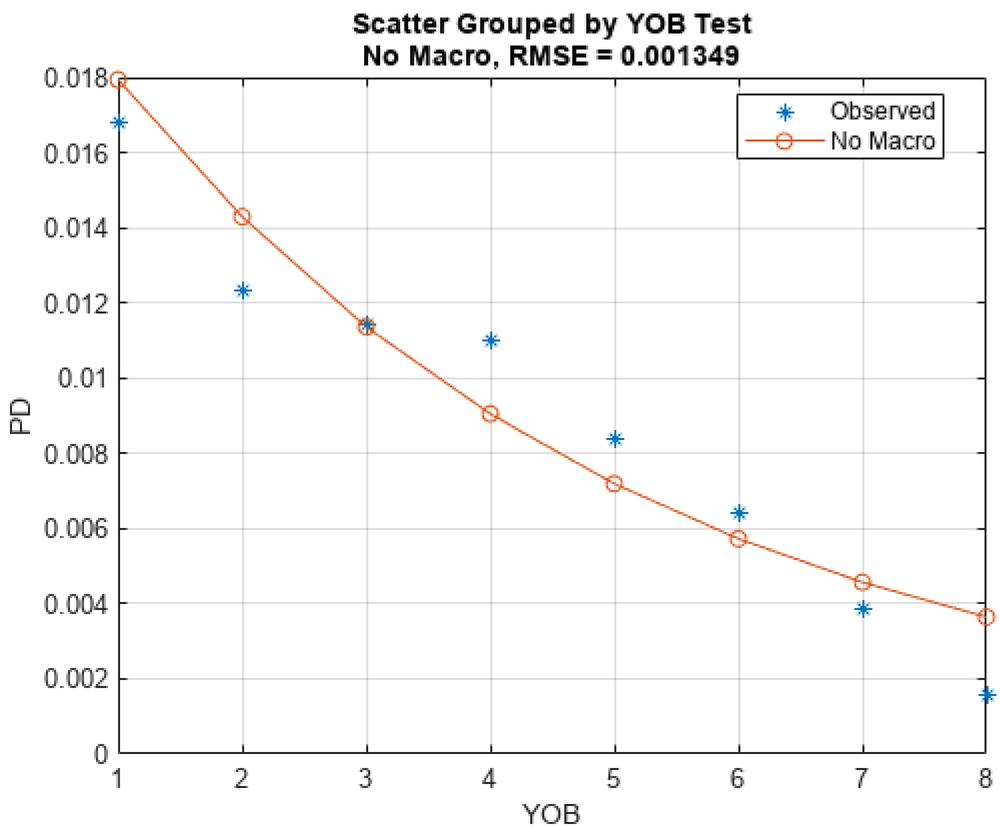
To make lifetime PD predictions, use the `predictLifetime` function. For lifetime predictions, projected values of the predictors are required for each ID value in the prediction data set. For example, predict the survival probability for the first two IDs in the dataset. See how the conditional PD (`PDNoMacro` column) and the lifetime PD (`LifetimePD` column) match for the first year of each ID. After that year, the lifetime PD increases because it is a cumulative probability. For more information, see `predictLifetime`. See also the “Expected Credit Loss Computation” on page 4-124 example.

```
data1 = data(1:16,:);
data1.LifetimePD = predictLifetime(ModelNoMacro,data1);
disp(data1)
```

ID	ScoreGroup	YOB	Default	Year	PDNoMacro	LifetimePD
1	Low Risk	1	0	1997	0.0084797	0.0084797
1	Low Risk	2	0	1998	0.0067697	0.015192
1	Low Risk	3	0	1999	0.0054027	0.020513
1	Low Risk	4	0	2000	0.0043105	0.024735
1	Low Risk	5	0	2001	0.0034384	0.028088
1	Low Risk	6	0	2002	0.0027422	0.030753
1	Low Risk	7	0	2003	0.0021867	0.032873
1	Low Risk	8	0	2004	0.0017435	0.034559
2	Medium Risk	1	0	1997	0.015097	0.015097
2	Medium Risk	2	0	1998	0.012069	0.026984
2	Medium Risk	3	0	1999	0.0096422	0.036366
2	Medium Risk	4	0	2000	0.0076996	0.043785
2	Medium Risk	5	0	2001	0.006146	0.049662
2	Medium Risk	6	0	2002	0.0049043	0.054323
2	Medium Risk	7	0	2003	0.0039125	0.058023
2	Medium Risk	8	0	2004	0.0031207	0.060962

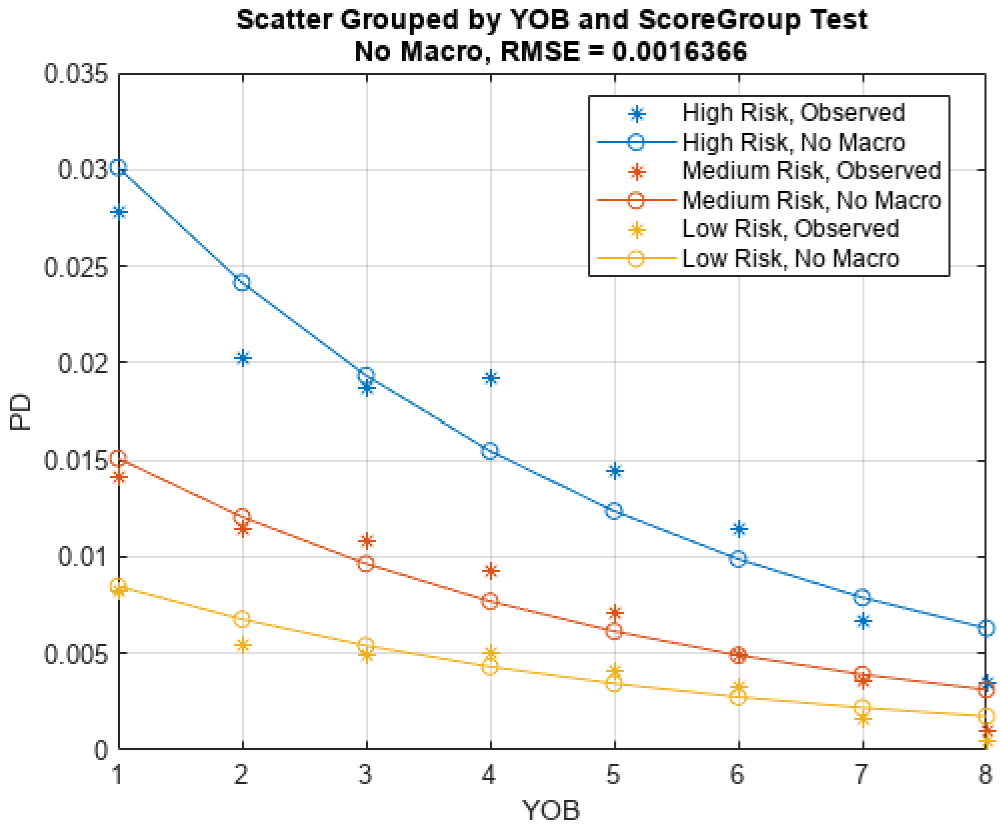
Visualize the in-sample (training) or out-of-sample (test) fit using `modelCalibrationPlot`. It requires a grouping variable to compute default rates and average predicted PD values for each group. Use the years on books as grouping variable here.

```
DataSetChoice = Test ;
if DataSetChoice=="Training"
   Ind = TrainDataInd;
else
   Ind = TestDataInd;
end
modelCalibrationPlot(ModelNoMacro,data(Ind,:),'YOB','DataID',DataSetChoice)
```



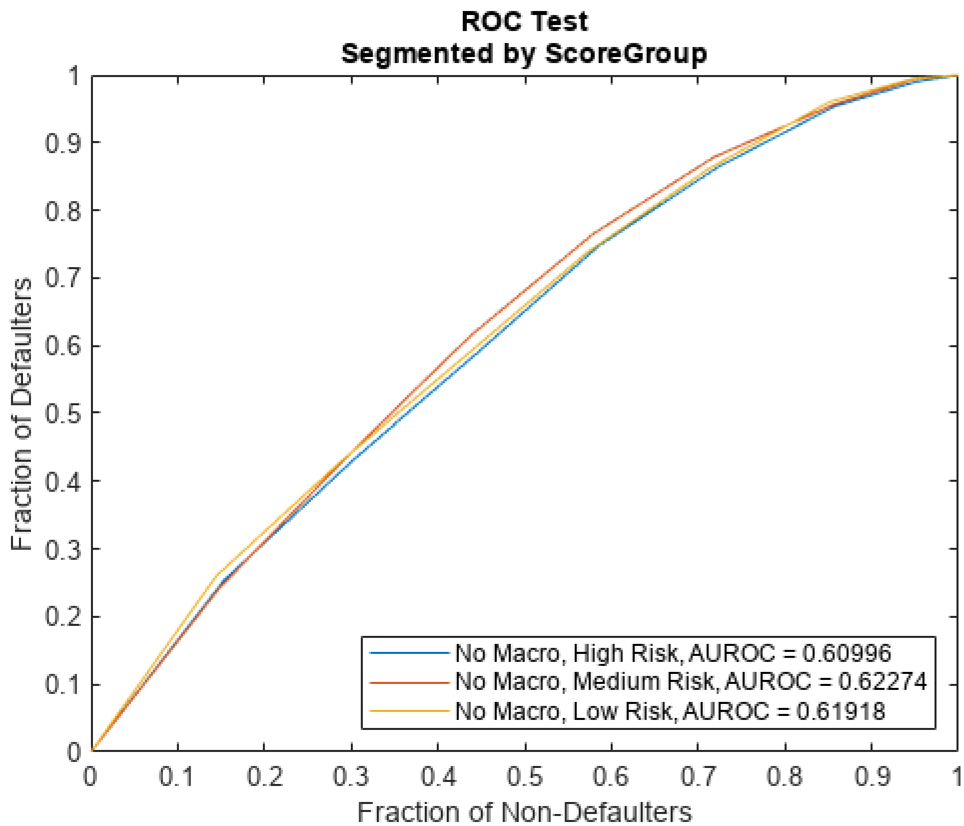
The score group can be input as a second grouping variable to visualize the fit by score groups.

```
modelCalibrationPlot(ModelNoMacro,data(Ind,:),{'YOB' 'ScoreGroup'},'DataID',DataSetChoice)
```



Lifetime PD models also support validation tools for model discrimination. In particular, the `modelDiscriminationPlot` function creates the receiver operating characteristic (ROC) curve plot. Here a separate ROC curve is requested for each score group. For more information, see `modelDiscriminationPlot`.

```
modelDiscriminationPlot(ModelNoMacro,data(Ind,:),'SegmentBy','ScoreGroup','DataID',DataSetChoice)
```



Model of Default Rates Including Macroeconomic Variables

The trend predicted with the previous model, as a function of years on books, has a very regular decreasing pattern. The data, however, shows some deviations from that trend. To try to account for those deviations, add the gross domestic product annual growth (represented by the `GDP` variable) and stock market annual returns (represented by the `Market` variable) to the model.

$$\log\left(\frac{p}{1-p}\right) = a_H + a_M 1_M + a_L 1_L + b_{YOB} YOB + b_{GDP} GDP + b_{Market} Market + \epsilon$$

Expand the data set to add one column for `GDP` and one for `Market`, using the data from the `dataMacro` table.

```
data = join(data,dataMacro);
disp(data(1:10,:))
```

```
    ID    ScoreGroup    YOB    Default    Year    PDNoMacro     GDP     Market
    __    __________    ___    _______    ____    _________    _____    ______

    1     Low Risk      1         0       1997    0.0084797     2.72     7.61
    1     Low Risk      2         0       1998    0.0067697     3.57    26.24
    1     Low Risk      3         0       1999    0.0054027     2.86     18.1
    1     Low Risk      4         0       2000    0.0043105     2.43     3.19
    1     Low Risk      5         0       2001    0.0034384     1.26   -10.51
    1     Low Risk      6         0       2002    0.0027422    -0.59   -22.95
    1     Low Risk      7         0       2003    0.0021867     0.63     2.78
    1     Low Risk      8         0       2004    0.0017435     1.85     9.48
```

```
    2     Medium Risk     1        0        1997     0.015097     2.72       7.61
    2     Medium Risk     2        0        1998     0.012069     3.57      26.24
```

Fit the model with the macroeconomic variables, or macro model, by expanding the model formula to include the `GDP` and the `Market` variables.

```
ModelMacro = fitLifetimePDModel(data(TrainDataInd,:),'logistic',...
   'ModelID','Macro','Description','Logistic model with YOB, score group and macro variables',...
   'IDVar','ID','LoanVars','ScoreGroup','AgeVar','YOB',...
   'MacroVars',{'GDP','Market'},'ResponseVar','Default');
disp(ModelMacro.UnderlyingModel)
```

```
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                              Estimate         SE          tStat       pValue   
                             __________    _________    _______    ___________

    (Intercept)                  -2.667      0.10146    -26.287    2.6919e-152
    ScoreGroup_Medium Risk     -0.70751     0.037108    -19.066     4.8223e-81
    ScoreGroup_Low Risk         -1.2895     0.045639    -28.253    1.2892e-175
    YOB                        -0.32082     0.013636    -23.528    2.0867e-122
    GDP                        -0.12295     0.039725     -3.095      0.0019681
    Market                   -0.0071812    0.0028298    -2.5377       0.011159


388018 observations, 388012 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.97e+03, p-value = 0
```

Both macroeconomic variables show a negative coefficient, consistent with the intuition that higher economic growth reduces default rates.

Use the `predict` function to predict the conditional PD. For illustration, here is how to predict the conditional PD on training and testing data using the macro model. The results are stored as a new column in the `data` table. Lifetime PD prediction is also supported with the `predictLifetime` function, as shown in the Model of Default Rates Using Score Group and Years on Books on page 3-44 section.

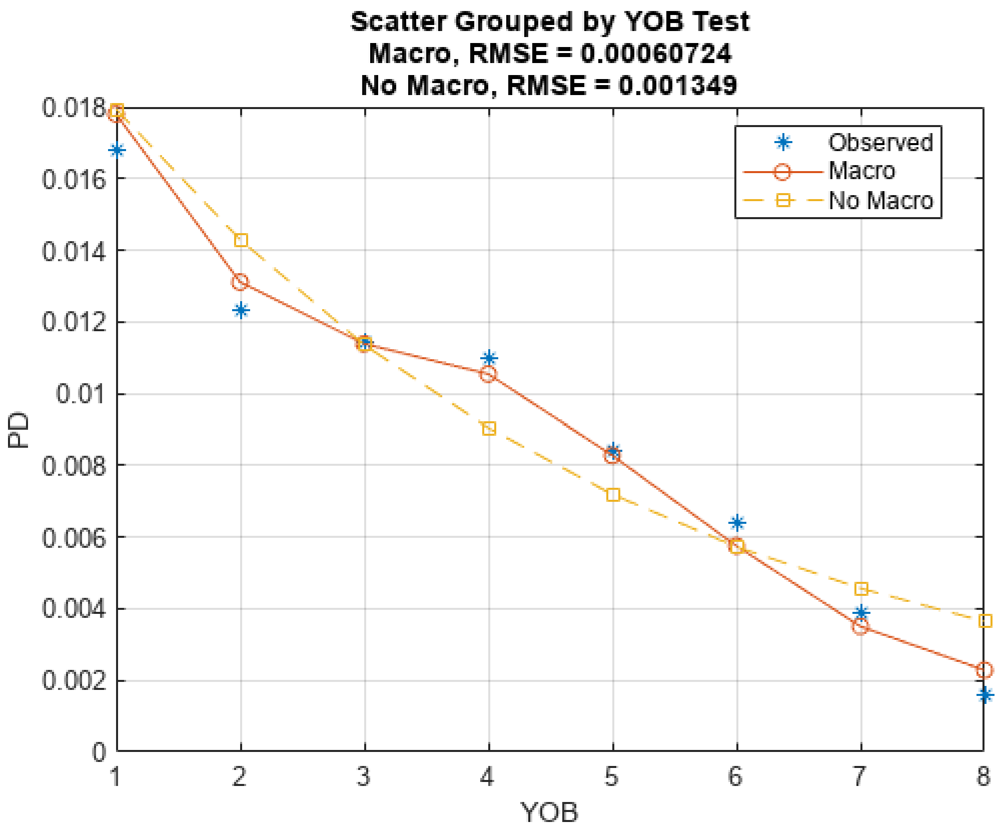
```
data.PDMacro = zeros(height(data),1);

% Predict in-sample
data.PDMacro(TrainDataInd) = predict(ModelMacro,data(TrainDataInd,:));
% Predict out-of-sample
data.PDMacro(TestDataInd) = predict(ModelMacro,data(TestDataInd,:));
```

The model accuracy and discrimination plots offer readily available comparison tools for the models.

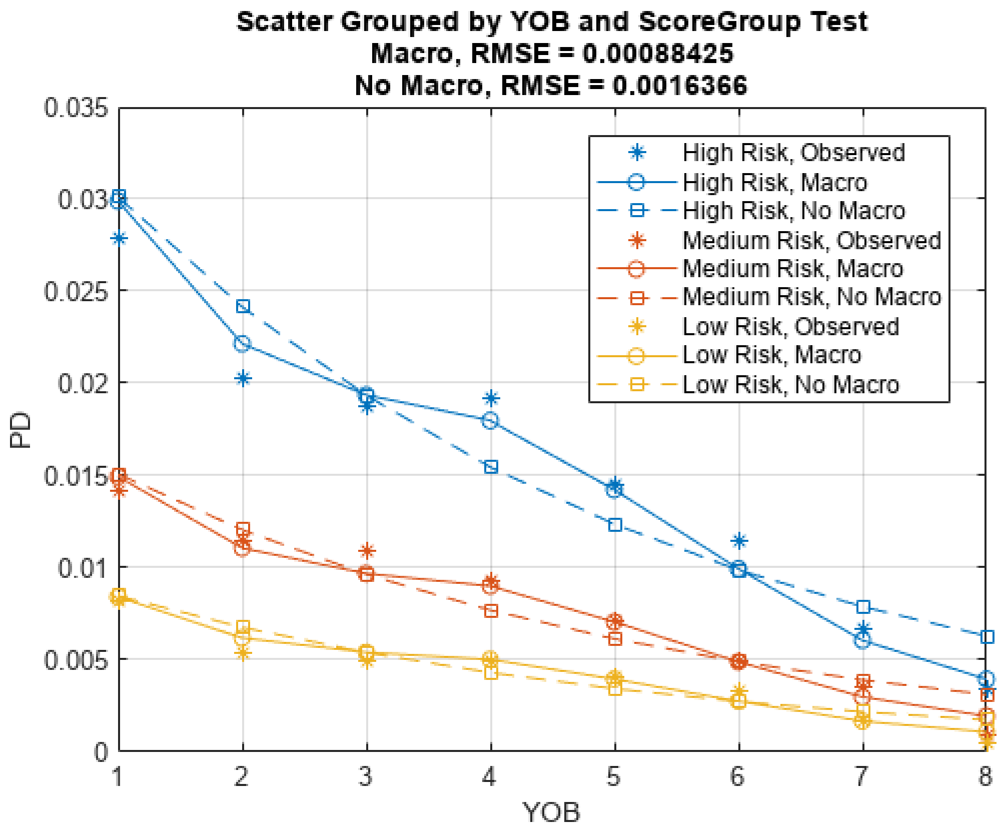
Visualize the in-sample or out of sample fit using `modelCalibrationPlot`. Pass the predictions from the model without macroeconomic variables as a reference model. Plot both using years on books as the single grouping variable first, and then using score group as a second grouping variable.

```
DataSetChoice = Test ;
if DataSetChoice=="Training"
   Ind = TrainDataInd;
```

```
else
   Ind = TestDataInd;
end
modelCalibrationPlot(ModelMacro,data(Ind,:),'YOB','ReferencePD',data.PDNoMacro(Ind),'ReferenceID
```



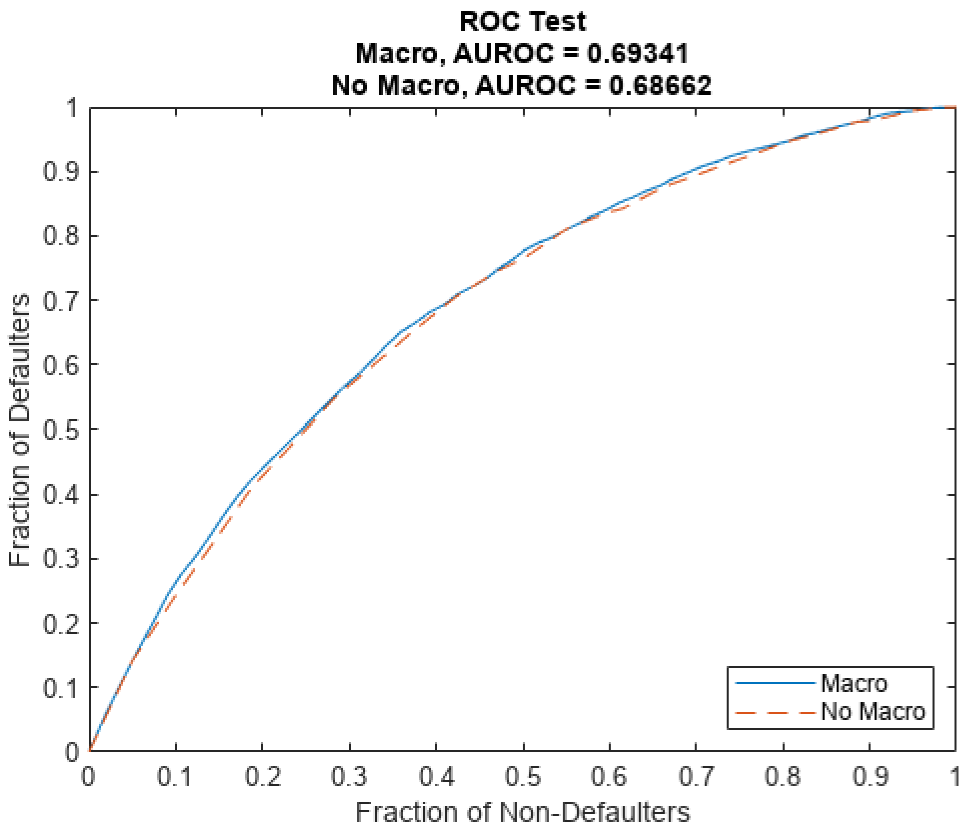

```
modelCalibrationPlot(ModelMacro,data(Ind,:),{'YOB','ScoreGroup'},'ReferencePD',data.PDNoMacro(Inc
```



The accuracy of the predictions significantly improves compared to the model with no macroeconomic variables. The predicted conditional PD values more closely follow the pattern of the observed default rates and the root mean square error (RMSE) reported is significantly smaller when the macroeconomic variables are included in the model.

Plot the ROC curve of the macro model and the model without macroeconomic variables to compare their performance with regards to model discrimination.

```
modelDiscriminationPlot(ModelMacro,data(Ind,:),'ReferencePD',data.PDNoMacro(Ind),'ReferenceID',Mo
```



Discrimination measures the ranking of customers by risk. Both models perform similarly, with only a slight improvement when the macroeconomic variables are added to the model. This means both models do a similar job separating low risk, medium risk and high risk customers by assigning higher PD values to customers with higher risk.

Although the discrimination performance of both models is similar, the predicted PD values are more accurate for the macro model. Using both discrimination and accuracy tools is important for model validation and model comparison.

Stress Testing of Probability of Default

Use the fitted macro model to stress-test the predicted probabilities of default.

Assume the following are stress scenarios for the macroeconomic variables provided, for example, by a regulator.

```
disp(dataMacroStress)
```

```
                 GDP     Market
                _____    ______

    Baseline     2.27    15.02
    Adverse      1.31     4.56
    Severe      -0.22    -5.64
```

Set up a basic data table for predicting the probabilities of default. This is a dummy data table, with one row for each combination of score group and number of years on books.

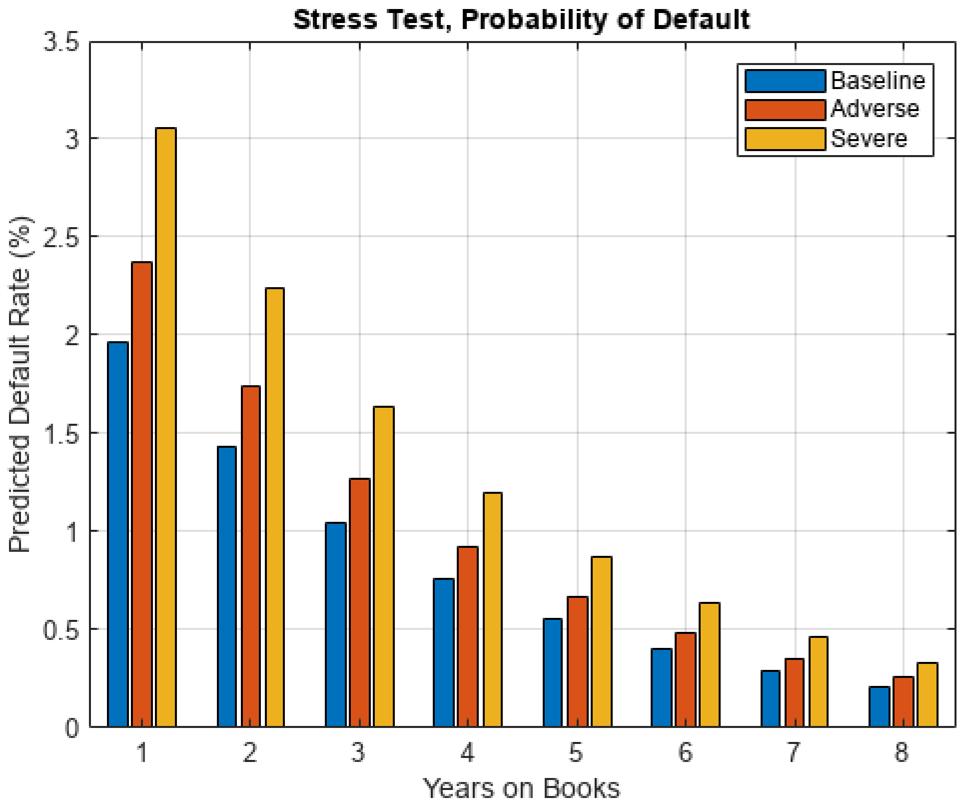
```
dataBaseline = table;
[ScoreGroup,YOB]=meshgrid(1:NumScoreGroups,1:NumYOB);
dataBaseline.ScoreGroup = categorical(ScoreGroup(:),1:NumScoreGroups,...
   categories(data.ScoreGroup),'Ordinal',true);
dataBaseline.YOB = YOB(:);
dataBaseline.ID = ones(height(dataBaseline),1);
dataBaseline.GDP = zeros(height(dataBaseline),1);
dataBaseline.Market = zeros(height(dataBaseline),1);
```

To make the predictions, set the same macroeconomic conditions (baseline, adverse, or severely adverse) for all combinations of score groups and number of years on books.

```
% Predict baseline the probabilities of default
dataBaseline.GDP(:) = dataMacroStress.GDP('Baseline');
dataBaseline.Market(:) = dataMacroStress.Market('Baseline');
dataBaseline.PD = predict(ModelMacro,dataBaseline);

% Predict the probabilities of default in the adverse scenario
dataAdverse = dataBaseline;
dataAdverse.GDP(:) = dataMacroStress.GDP('Adverse');
dataAdverse.Market(:) = dataMacroStress.Market('Adverse');
dataAdverse.PD = predict(ModelMacro,dataAdverse);

% Predict the probabilities of default in the severely adverse scenario
dataSevere = dataBaseline;
dataSevere.GDP(:) = dataMacroStress.GDP('Severe');
dataSevere.Market(:) = dataMacroStress.Market('Severe');
dataSevere.PD = predict(ModelMacro,dataSevere);
```

Visualize the average predicted probability of default across score groups under the three alternative regulatory scenarios. Here, all score groups are implicitly weighted equally. However, predictions can also be made at a loan level for any given portfolio to make the predicted default rates consistent with the actual distribution of loans in the portfolio. The same visualization can be produced for each score group separately.

```
PredPDYOB = zeros(NumYOB,3);
PredPDYOB(:,1) = mean(reshape(dataBaseline.PD,NumYOB,NumScoreGroups),2);
PredPDYOB(:,2) = mean(reshape(dataAdverse.PD,NumYOB,NumScoreGroups),2);
PredPDYOB(:,3) = mean(reshape(dataSevere.PD,NumYOB,NumScoreGroups),2);

figure;
bar(PredPDYOB*100);
xlabel('Years on Books')
ylabel('Predicted Default Rate (%)')
legend('Baseline','Adverse','Severe')
title('Stress Test, Probability of Default')
grid on
```



References

1. Generalized Linear Models documentation, see "Generalized Linear Models".
2. Generalized Linear Mixed Effects Models documentation, see "Generalized Linear Mixed-Effects Models".
3. Federal Reserve, Comprehensive Capital Analysis and Review (CCAR): https://www.federalreserve.gov/bankinforeg/ccar.htm
4. Bank of England, Stress Testing: https://www.bankofengland.co.uk/financial-stability
5. European Banking Authority, EU-Wide Stress Testing: https://www.eba.europa.eu/risk-analysis-and-data/eu-wide-stress-testing

See Also

`fitglm` | `fitglme` | `fitLifetimePDModel` | `predict` | `predictLifetime` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot` | `Logistic` | `Probit`

Related Examples

- "Credit Rating by Bagging Decision Trees"
- "Credit Scorecard Modeling with Missing Values"
- "Basic Lifetime PD Model Validation" on page 4-129
- "Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113

- "Compare Lifetime PD Models Using Cross-Validation" on page 4-121
- "Expected Credit Loss Computation" on page 4-124

More About

- "Overview of Lifetime Probability of Default Models" on page 1-25

compactCreditScorecard Object Workflow

This example shows a workflow for creating a `compactCreditScorecard` object from a `creditscorecard` object.

Step 1. Create a `creditscorecard` object

To create a `compactCreditScorecard` object, you must first create a `creditscorecard` object. Create a `creditscorecard` object with the `CreditCardData.mat` file, and set the name-value pair argument `'BinMissingData'` to `true` because the `dataMissing` data set contains missing data.

```
load CreditCardData.mat
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);
sc = modifybins(sc,'CustAge','MinValue',0);
sc = modifybins(sc,'CustIncome','MinValue',0);
```

Step 2. Fit a logistic regression model for the `creditscorecard` object

Use `fitmodel` to fit a logistic regression model using the Weight of Evidence (WOE) data.

```
[sc, mdl] = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue   
                   ________    ________    ______    __________

    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

Step 3. Create a new data set for scoring the `creditscorecard` object

Create a new data set that is used for scoring based on the previously created `creditscorecard` object.

```
tdata = data(1:10, mdl.PredictorNames);
tdata.CustAge(2) = NaN;
tdata.CustAge(5) = -5;
tdata.ResStatus(1) = '<undefined>';
tdata.ResStatus(3) = 'Landlord';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
tdata.EmpStatus(7) = 'Freelancer';
tdata.CustIncome(8) = -1;
tdata.CustIncome(4) = NaN;
disp(tdata);
```

```
    CustAge     ResStatus       EmpStatus      CustIncome    TmWBank    OtherCC    AMBalance
    _______    ___________    ___________    __________    _______    _______    _________

      53       <undefined>    Unknown          50000         55         Yes        1055.9
     NaN       Home Owner     Employed         52000         25         Yes        1161.6
      47       Landlord       <undefined>      37000         61         No         877.23
      50       Home Owner     Employed           NaN         20         Yes        157.37
      -5       Home Owner     Employed         53000         14         Yes        561.84
      65       Home Owner     Employed         48000         59         Yes        968.18
      34       Home Owner     Freelancer       32000         26         Yes        717.82
      50       Other          Employed            -1         33         No         3041.2
      50       Tenant         Unknown          52000         25         Yes        115.56
      49       Home Owner     Unknown          53000         23         Yes         718.5
```

Use `displaypoints` to display the points per predictor. Use `score` to compute the credit scores using the new data (`tdata`). Then use `probdefault` with the new data (`tdata`) to calculate probability of default. When using `formatpoints`, the `'Missing'` name-value pair argument is set to `'minpoints'` because `tdata` contains missing data.

```
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors            Bin            Points  
    _____________    ______________    _________

    {'CustAge'  }    {'[0,33)'    }     -0.14173
    {'CustAge'  }    {'[33,37)'   }     -0.11095
    {'CustAge'  }    {'[37,40)'   }    -0.059244
    {'CustAge'  }    {'[40,46)'   }     0.074167
    {'CustAge'  }    {'[46,48)'   }       0.1889
    {'CustAge'  }    {'[48,51)'   }      0.20204
    {'CustAge'  }    {'[51,58)'   }      0.22935
    {'CustAge'  }    {'[58,Inf]'  }      0.45019
    {'CustAge'  }    {'<missing>' }    0.0096749
    {'ResStatus'}    {'Tenant'    }    -0.029778
    {'ResStatus'}    {'Home Owner'}      0.12425
    {'ResStatus'}    {'Other'     }      0.36796
    {'ResStatus'}    {'<missing>' }       0.1364
    {'EmpStatus'}    {'Unknown'   }    -0.075948
    {'EmpStatus'}    {'Employed'  }      0.31401
    {'EmpStatus'}    {'<missing>' }          NaN
          ⋮
```

```
[Scores, Points] = score(sc, tdata)
```

```
Scores = 10×1

    1.2784
    1.0071
       NaN
       NaN
    0.9960
    1.8771
       NaN
       NaN
    1.0283
    0.8095
```

```
Points=10×7 table
     CustAge      ResStatus     EmpStatus     CustIncome      TmWBank      OtherCC      AMBalance 
    _________    _________    _________    __________    _________    ________    _________

      0.22935       0.1364    -0.075948       0.45309       0.3958     0.15715    -0.017438
    0.0096749      0.12425      0.31401       0.45309    -0.033652     0.15715    -0.017438
       0.1889       0.1364          NaN      0.080697       0.3958    -0.18537    -0.017438
      0.20204      0.12425      0.31401           NaN    -0.044701     0.15715      0.35539
    0.0096749      0.12425      0.31401       0.45309    -0.044701     0.15715    -0.017438
      0.45019      0.12425      0.31401       0.45309       0.3958     0.15715    -0.017438
     -0.11095      0.12425          NaN      -0.11452    -0.033652     0.15715    -0.017438
      0.20204      0.36796      0.31401           NaN    -0.033652    -0.18537     -0.21195
      0.20204    -0.029778    -0.075948       0.45309    -0.033652     0.15715      0.35539
      0.20204      0.12425    -0.075948       0.45309    -0.033652     0.15715    -0.017438
```

```
pd = probdefault(sc, tdata)
```

```
pd = 10×1

    0.2178
    0.2676
       NaN
       NaN
    0.2697
    0.1327
       NaN
       NaN
    0.2634
    0.3080
```

```
sc = formatpoints(sc,'BasePoints',true,'Missing','minpoints','Round','finalscore','PointsOddsAndI
PointsInfo1 = displaypoints(sc)
```

```
PointsInfo1=39×3 table
      Predictors           Bin          Points 
    ______________    ______________    _______

    {'BasePoints'}    {'BasePoints'}     500.66
    {'CustAge'   }    {'[0,33)'    }    -17.461
    {'CustAge'   }    {'[33,37)'   }     -15.24
    {'CustAge'   }    {'[37,40)'   }    -11.511
    {'CustAge'   }    {'[40,46)'   }    -1.8871
    {'CustAge'   }    {'[46,48)'   }     6.3888
```

```
    {'CustAge'   }    {'[48,51)'   }     7.3367
    {'CustAge'   }    {'[51,58)'   }     9.3068
    {'CustAge'   }    {'[58,Inf]'  }     25.238
    {'CustAge'   }    {'<missing>' }    -6.5392
    {'ResStatus' }    {'Tenant'    }    -9.3852
    {'ResStatus' }    {'Home Owner'}     1.7253
    {'ResStatus' }    {'Other'     }     19.305
    {'ResStatus' }    {'<missing>' }     2.6022
    {'EmpStatus' }    {'Unknown'   }    -12.716
    {'EmpStatus' }    {'Employed'  }     15.414
      ⋮
```

```
[Scores1, Points1] = score(sc, tdata)
```

```
Scores1 = 10×1

   542
   523
   488
   495
   522
   585
   445
   448
   524
   508
```

```
Points1=10×8 table
    BasePoints    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBala
    __________    _______    _________    _________    __________    _______    _______    ______

      500.66       9.3068      2.6022      -12.716       25.446       21.314     4.0988     -8.4
      500.66      -6.5392      1.7253       15.414       25.446      -9.6646     4.0988     -8.4
      500.66       6.3888      2.6022      -12.716      -1.4161       21.314    -20.609     -8.4
      500.66       7.3367      1.7253       15.414      -42.148      -10.462     4.0988     18.3
      500.66      -6.5392      1.7253       15.414       25.446      -10.462     4.0988     -8.4
      500.66       25.238      1.7253       15.414       25.446       21.314     4.0988     -8.4
      500.66       -15.24      1.7253      -12.716      -15.498      -9.6646     4.0988     -8.4
      500.66       7.3367      19.305       15.414      -42.148      -9.6646    -20.609     -22.5
      500.66       7.3367     -9.3852      -12.716       25.446      -9.6646     4.0988     18.3
      500.66       7.3367      1.7253      -12.716       25.446      -9.6646     4.0988     -8.4
```

```
pd1 = probdefault(sc, tdata)
```

```
pd1 = 10×1

    0.2178
    0.2676
    0.3721
    0.3488
    0.2697
    0.1327
    0.5178
    0.5077
    0.2634
```

```
    0.3080
```

Step 4. Create a `compactCreditScorecard` object from the `creditscorecard` object

Create a `compactCreditScorecard` object using the `creditscorecard` object as the input. Alternatively, you can create the `compactCreditScorecard` object using the `compact` function in Financial Toolbox™.

```
csc = compactCreditScorecard(sc)
```

```
csc = 
  compactCreditScorecard with properties:

              Description: ''
                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
        NumericPredictors: {'CustAge'  'CustIncome'  'TmWBank'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
            PredictorVars: {'CustAge'  'ResStatus'  'EmpStatus'  'CustIncome'  'TmWBank'  'OtherC
```

Step 5. Use associated functions to analyze the `compactCreditScorecard` object

You can analyze the compactCreditScorecard object with `displaypoints`, `score`, and `probdefault` from Risk Management Toolbox™.

```
PointsInfo2 = displaypoints(csc)
```

```
PointsInfo2=39×3 table
      Predictors           Bin          Points 
    _______________    _______________    _______

    {'BasePoints'}    {'BasePoints'}     500.66
    {'CustAge'   }    {'[0,33)'    }    -17.461
    {'CustAge'   }    {'[33,37)'   }     -15.24
    {'CustAge'   }    {'[37,40)'   }    -11.511
    {'CustAge'   }    {'[40,46)'   }    -1.8871
    {'CustAge'   }    {'[46,48)'   }     6.3888
    {'CustAge'   }    {'[48,51)'   }     7.3367
    {'CustAge'   }    {'[51,58)'   }     9.3068
    {'CustAge'   }    {'[58,Inf]'  }     25.238
    {'CustAge'   }    {'<missing>' }    -6.5392
    {'ResStatus' }    {'Tenant'    }    -9.3852
    {'ResStatus' }    {'Home Owner'}     1.7253
    {'ResStatus' }    {'Other'     }     19.305
    {'ResStatus' }    {'<missing>' }     2.6022
    {'EmpStatus' }    {'Unknown'   }    -12.716
    {'EmpStatus' }    {'Employed'  }     15.414
      ⋮
```

```
[Scores2, Points2] = score(csc, tdata)
```

```
Scores2 = 10×1

   542
   523
```

```
    488
    495
    522
    585
    445
    448
    524
    508

Points2=10×8 table
    BasePoints    CustAge    ResStatus    EmpStatus    CustIncome    TmWBank    OtherCC    AMBala
    __________    _______    _________    _________    __________    _______    _______    ______

      500.66       9.3068      2.6022      -12.716       25.446       21.314     4.0988      -8.4
      500.66      -6.5392      1.7253       15.414       25.446      -9.6646     4.0988      -8.4
      500.66       6.3888      2.6022      -12.716      -1.4161       21.314    -20.609      -8.4
      500.66       7.3367      1.7253       15.414      -42.148      -10.462     4.0988      18.3
      500.66      -6.5392      1.7253       15.414       25.446      -10.462     4.0988      -8.4
      500.66       25.238      1.7253       15.414       25.446       21.314     4.0988      -8.4
      500.66       -15.24      1.7253      -12.716      -15.498      -9.6646     4.0988      -8.4
      500.66       7.3367      19.305       15.414      -42.148      -9.6646    -20.609     -22.5
      500.66       7.3367     -9.3852      -12.716       25.446      -9.6646     4.0988      18.3
      500.66       7.3367      1.7253      -12.716       25.446      -9.6646     4.0988      -8.4
```

```
pd2 = probdefault(csc, tdata)
```

```
pd2 = 10×1

    0.2178
    0.2676
    0.3721
    0.3488
    0.2697
    0.1327
    0.5178
    0.5077
    0.2634
    0.3080
```

Compare the size of the `creditscorecard` and `compactCreditScorecard` objects.

```
whos('dataMissing','sc','csc')
```

```
  Name                Size              Bytes  Class                         Attributes

  csc                 1x1               39598  compactCreditScorecard
  dataMissing      1200x11              84603  table
  sc                  1x1              166575  creditscorecard
```

The size of the `compactCreditScorecard` object is lightweight compared to the `creditscorecard` object. However, the `compactCreditScorecard` object cannot be directly modified. If you need to change a `compactCreditScorecard` object, you must change the starting creditscorecard object, and then reconvert that object to create the compactCreditScorecard object again.

See Also

creditscorecard | screenpredictors | autobinning | bininfo | predictorinfo | modifypredictor | modifybins | bindata | plotbins | fitmodel | displaypoints | formatpoints | score | setmodel | probdefault | validatemodel

Related Examples

- “Common Binning Explorer Tasks” on page 3-4
- “Credit Scorecard Modeling with Missing Values”
- “Feature Screening with screenpredictors” on page 3-64
- “Troubleshooting Credit Scorecard Results”
- “Credit Rating by Bagging Decision Trees”
- “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36

More About

- “Overview of Binning Explorer” on page 3-2
- “About Credit Scorecards”
- “Credit Scorecard Modeling Workflow”
- Monotone Adjacent Pooling Algorithm (MAPA)
- “Credit Scorecard Modeling Using Observation Weights”

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

Feature Screening with screenpredictors

This example shows how to perform predictor screening using screenpredictors and then set predictor thresholds using the Threshold Predictors live task. Predictor screening using `screenpredictors` is a type of univariate analysis performed as an early step in the "Credit Scorecard Modeling Workflow". Predictor screening is an important preprocessing step when you work with credit scorecards, as data sets can be prohibitively large and have dozens or hundreds of potential predictors.

The goal of screening predictors is to pare down the set of predictors to a subset that is more useful in predicting the response variable based on the calculated metrics. You can set predictor thresholds using the Threshold Predictors live task to select the top predictors as ranked by a given metric to train your credit scorecards.

Load Data

The credit card data table contains a customer ID (`CustID`), nine predictors, and the response variable (`status`). Some of the risk factors are more useful in predicting the probability of a loan default, whereas others are less useful. The screening process helps you select the best subset of predictors.

Although the data set in this example contains only a few predictors, in practice, credit scorecard data sets can be very large. The predictor screening process is important as data sets grow to contain dozens or hundreds of predictors.

```
% Load credit card data.
load CreditCardData.mat

% Use the dataMissing data set, which contains some missing values.
data = dataMissing;

% Identify the ID and response variables.
idvar = 'CustID';
responsevar = 'status';

% Examine the structure of the table.
disp(head(data));
```

```
    CustID    CustAge    TmAtAddress     ResStatus     EmpStatus    CustIncome    TmWBank    Othe
    ______    _______    ___________    ___________    _________    __________    _______    ____

      1          53          62         <undefined>    Unknown        50000         55         Ye
      2          61          22         Home Owner     Employed       52000         25         Ye
      3          47          30         Tenant         Employed       37000         61         No
      4         NaN          75         Home Owner     Employed       53000         20         Ye
      5          68          56         Home Owner     Employed       53000         14         Ye
      6          65          13         Home Owner     Employed       48000         59         Ye
      7          34          32         Home Owner     Unknown        32000         26         Ye
      8          50          57         Other          Employed       51000         33         No
```

Add Additional Derived Predictors

Often, derivative predictors can capture additional information or produce better metrics results; for example, the ratio of two predictors or a predictor transformation for predictor x, such as x^2 or log(x). To demonstrate this, create two derived predictors and add them to the data set.

```
data.BalanceUtilRatio = data.AMBalance ./ data.UtilRate;
data.BalanceIncomeRatio = data.AMBalance ./ data.CustIncome;
```

Compute Metrics

Use `screenpredictors` to compute several measures of risk factor predictiveness. The columns of the output table contain the metrics values for the predictors. The table is sorted by the information value (`InfoValue`).

```
T = screenpredictors(data,'IDVar',idvar,'ResponseVar',responsevar)
```

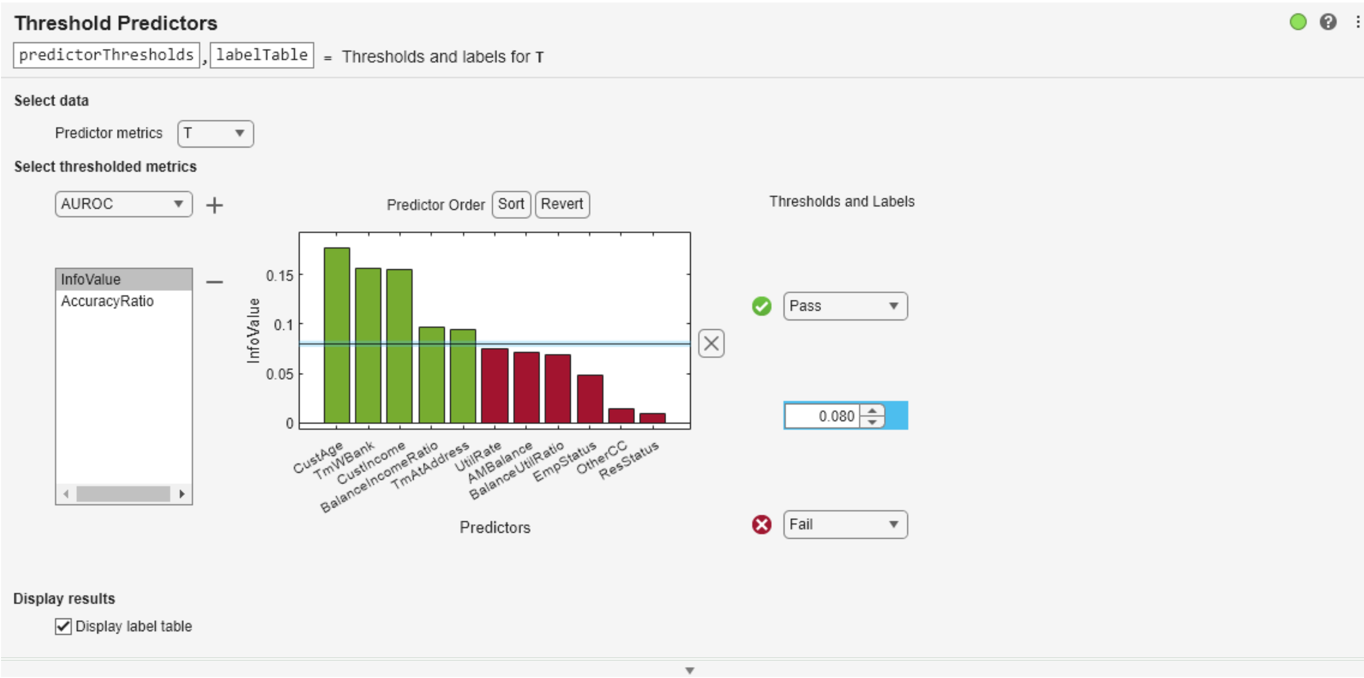
T=11×7 table

	InfoValue	AccuracyRatio	AUROC	Entropy	Gini	Chi2PVal
CustAge	0.17698	0.1672	0.5836	0.88795	0.42645	0.00205
TmWBank	0.15719	0.13612	0.56806	0.89167	0.42864	0.00545
CustIncome	0.15572	0.17758	0.58879	0.891	0.42731	0.00184
BalanceIncomeRatio	0.097073	0.1278	0.5639	0.90024	0.43303	0.119
TmAtAddress	0.094574	0.010421	0.50521	0.90089	0.43377	0.1
UtilRate	0.075086	0.035914	0.51796	0.90405	0.43575	0.455
AMBalance	0.07159	0.087142	0.54357	0.90446	0.43592	0.485
BalanceUtilRatio	0.068955	0.026538	0.51327	0.90486	0.43614	0.525
EmpStatus	0.048038	0.10886	0.55443	0.90814	0.4381	0.000378
OtherCC	0.014301	0.044459	0.52223	0.91347	0.44132	0.0476
ResStatus	0.0095558	0.049855	0.52493	0.91446	0.44198	0.298

Set Threshold Metrics

Set thresholds for the predictors based on one or more metrics. Use the Threshold Predictors live task to interactively select thresholds for one or more predictors. In the plot displayed for **Predictors**, green bars indicate predictors that pass the threshold and red bars indicate predictors that do not pass the threshold. You can omit predictors that do not "pass" the threshold from the final data set.

Use the Threshold Predictors live task to select predictors based on their information value (`InfoValue`) and accuracy ratio (`AccuracyRatio`). Additional thresholds can be set by adding the desired metric using the **Select threshold metrics** drop-down control.



labelTable=*11×2 table*

	InfoValue	AccuracyRatio
CustAge	Pass	Pass
TmWBank	Pass	Pass
CustIncome	Pass	Pass
BalanceIncomeRatio	Pass	Pass
TmAtAddress	Pass	Fail
UtilRate	Fail	Fail
AMBalance	Fail	Pass
BalanceUtilRatio	Fail	Fail
EmpStatus	Fail	Pass
OtherCC	Fail	Fail
ResStatus	Fail	Fail

Screening Summary

Summarize the thresholding results in table form. The `lableTable` output from the live task indicates which of the predictors passed each of the threshold tests.

```
disp(labelTable)
```

	InfoValue	AccuracyRatio
CustAge	Pass	Pass
TmWBank	Pass	Pass
CustIncome	Pass	Pass
BalanceIncomeRatio	Pass	Pass
TmAtAddress	Pass	Fail
UtilRate	Fail	Fail
AMBalance	Fail	Pass

```
    BalanceUtilRatio        Fail           Fail
    EmpStatus               Fail           Pass
    OtherCC                 Fail           Fail
    ResStatus               Fail           Fail
```

Reduce Table

Create a reduced table that contains only the passing predictors. Select only the predictors that pass both of the threshold tests and create a reduced data set.

```
% Select predictors that pass at least 2 metric threshold tests.
all_passes = labelTable.Variables == "Pass";
pass_both_idx = 2 <= sum(all_passes,2);
selected_predictors = T.Row(pass_both_idx);

% Trim the data table to contain only the ID, passing predictors, and
% response.
top_predictor_table = data(:,[idvar; selected_predictors; responsevar]);
```

Use `creditscorecard` to create a `creditscorecard` object using the reduced data set.

```
% Create the credit scorecard using the screened predictors.
sc = creditscorecard(top_predictor_table,'IDVar',idvar,'ResponseVar',responsevar,...
    'BinMissingData', true)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmWBank'  'CustIncome'  'BalanceIncomeRatio'
        NumericPredictors: {'CustAge'  'TmWBank'  'CustIncome'  'BalanceIncomeRatio'}
    CategoricalPredictors: {1x0 cell}
           BinMissingData: 1
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmWBank'  'CustIncome'  'BalanceIncomeRatio'}
                     Data: [1200x6 table]
```

For more information on developing credit scorecards, see “Create Credit Scorecards”.

See Also

`creditscorecard` | `screenpredictors` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Related Examples

- “Common Binning Explorer Tasks” on page 3-4
- “Credit Scorecard Modeling with Missing Values”
- “Troubleshooting Credit Scorecard Results”
- “Credit Rating by Bagging Decision Trees”
- “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36

More About

- “Overview of Binning Explorer” on page 3-2
- “About Credit Scorecards”
- “Credit Scorecard Modeling Workflow”
- Monotone Adjacent Pooling Algorithm (MAPA)
- “Credit Scorecard Modeling Using Observation Weights”

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

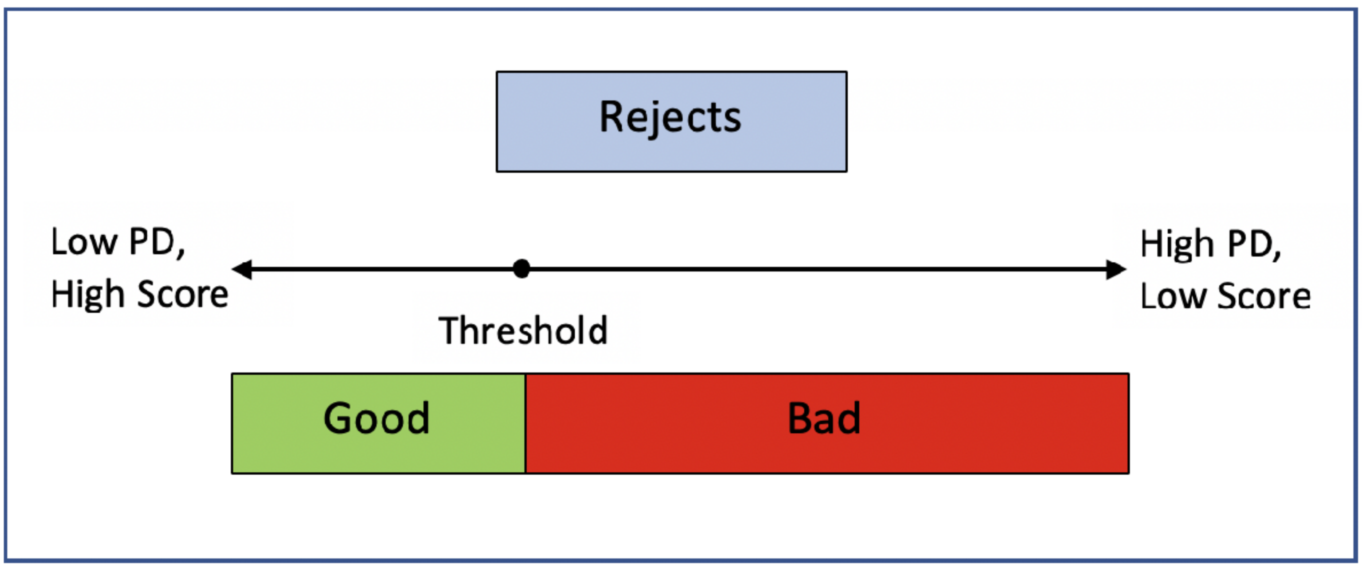
Use Reject Inference Techniques with Credit Scorecards

This example demonstrates the hard-cutoff and fuzzy augmentation approaches to reject inference.

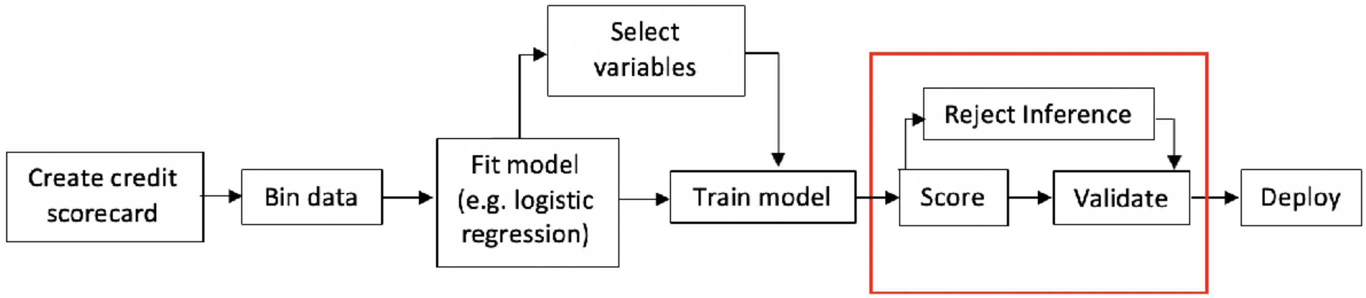
Reject inference is a method for improving the quality of a credit scorecard by incorporating data from rejected loan applications. Bias can result if a credit scorecard model is built only on accepts and does not account for applications rejected because of past denials for credit or unknown nondefault status. By using the reject inference method, you can infer the performance of rejects and include them in your credit scorecard model to remedy this bias.

To develop a credit scorecard, you must identify each borrower as either "good" or "bad". For rejected applications, information to identify borrowers as "good" or "bad" is not available. You cannot tell for sure to which group a borrower would have belonged had they been granted a loan. The reject inference method allows you to infer whether a borrower would likely be "good" or "bad" enabling you to incorporate the rejected application data into the data set that you use to build a credit scorecard.

As the diagram shows, reject inference requires that you determine the threshold (cutoff point) below which rejects are considered as "bad." This example demonstrates the hard-cutoff and the fuzzy augmentation approaches to calculate this threshold.



The following diagram shows the typical process for building a scorecard model. The red box represents the reject inference process, where the performance of the previously rejected applications is estimated and then used to re-train the credit scorecard model.



The workflow for the reject inference process is:

1. Build a logistic regression model based on the accepts.
2. Infer the class of rejects using one of the reject inference techniques.
3. Combine the accepts and rejects into a single data set.
4. Create a new scorecard, bin the expanded data set, and build a new logistic model.
5. Validate the final model.

There are two types of reject inference:

- *Simple assignment* does not use a reject inference process and either ignores rejects or assigns all rejects to the "bad" class.
- *Augmentation* uses a reject inference process to handle rejects based on a scoring model by combining the original data set with the rejects data.

This example focuses on augmentation techniques. The most popular techniques for augmentation are:

- **Simple augmentation** — Using a cutoff value, this method assigns rejects with scores below and above the value to the "bad" or "good" class, respectively. The cutoff value must reflect that the rate of bads in the rejects is higher than in the accepted population. After the class ("good" or "bad") is assigned to the rejects, the entire population of accepts and rejects are fitted in the credit scorecard model and then scored. This approach is also called the *hard-cutoff* technique.
- **Fuzzy augmentation** — This method scores the rejects by using a credit scorecard model based on the accepts. These rejects are duplicated into two observations, where each is assigned a probability of being "good" or "bad," and then aggregated to the accepts. A new credit scorecard model is then estimated on the new data set.

In this example, the following workflows are presented:

- Hard-cutoff on page 3-71
- Fuzzy augmentation on page 3-78

Both of these approaches use the binning rules preserved from the original scorecard and apply them to the new scorecard that is based on the combined data set.

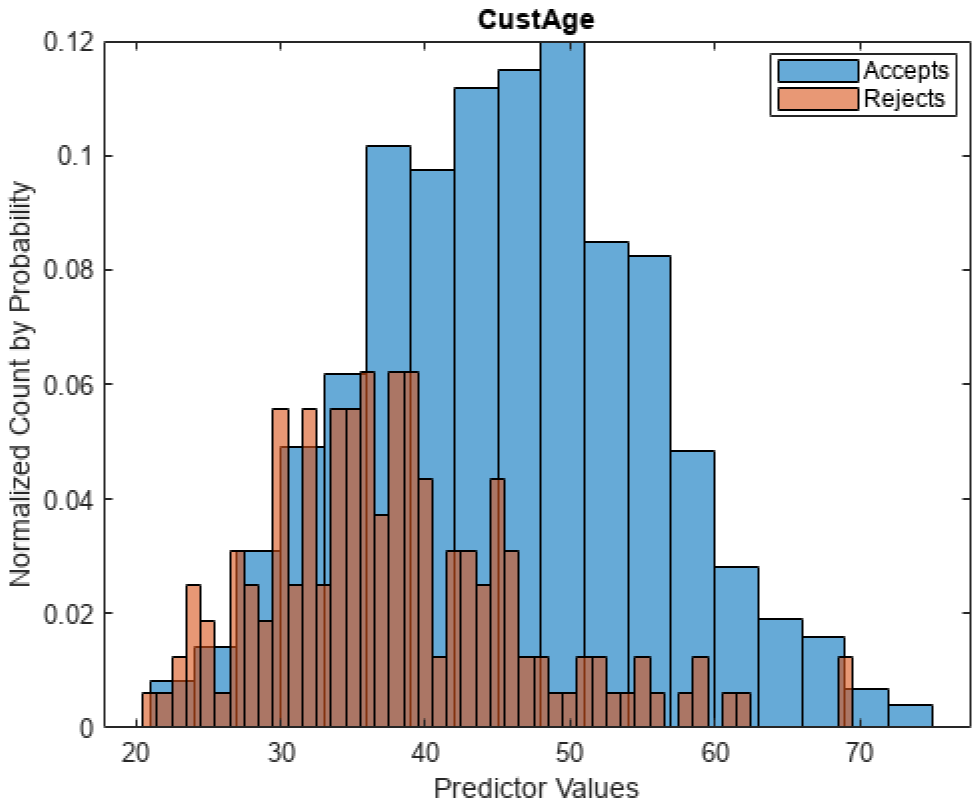
Note: The data sets in this example are technically through-the-door (TTD) observations. That is, accepts and rejects are lumped together and differentiated based on their accept or reject decision. A rejects data set is then created from the TTD observations.

Hard-Cutoff Technique Workflow

The *hard-cutoff* technique uses a predefined cutoff value and assigns rejects below the cutoff as "bad" and above the cutoff as "good." The cutoff value must reflect that the rate of "bads" in the rejects is higher than in the accepts. After each reject is assigned a class ("good" or "bad"), the entire population of accepts and rejects is fitted in a credit scorecard model, and then that model is scored and validated. This approach is also called the *simple augmentation* technique. The main challenge in this approach is choosing the cutoff value.

First, visualize the data for accepts and rejects for a selected predictor.

```
% Load the data
load CreditCardData.mat
load RejectsCreditCardData.mat

Predictor = CustAge;
figure;
h1 = histogram(data.(Predictor));
hold on
h2 = histogram(Rejects.(Predictor));
h1.Normalization = 'probability';
h2.Normalization = 'probability';
title(Predictor)
xlabel('Predictor Values')
ylabel('Normalized Count by Probability')
hold off
legend({'Accepts','Rejects'},'Location','best');
```



Create a `creditscorecard` Object for the Accepts and Score the Data

Use `creditscorecard` to create a `creditscorecard` object that you can use to bin, fit, and then score the accepts.

```
scHC = creditscorecard(data,'IDVar','CustID');
scHC = autobinning(scHC);
scHC = fitmodel(scHC);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
```

```
    EmpStatus      0.88565       0.293    3.0227     0.0025055
    CustIncome     0.70164     0.21844    3.2121     0.0013179
    TmWBank         1.1074     0.23271    4.7589    1.9464e-06
    OtherCC         1.0883     0.52912    2.0569      0.039696
    AMBalance        1.045     0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

```
ScoreRange = [300 850];
scHC = formatpoints(scHC,'WorstAndBestScores',ScoreRange);
ScoresAccepts = score(scHC);
```

Choose a Bad Rate and Score the Rejects

A reject is "good" or "bad" based on the specified bad rate (BR) value. In general, the credit scoring industry assumes that rejects have a BR of 75%. This is a subjective evaluation that is usually based on an unknown value. In this example, you can adjust the value of `BR`.

The `CreditCardData.mat` input data has `'status'` as response. Assume that `GoodLabel` (which means a nondefault) is the class that has a higher count in the response. In this example, `GoodLabel` is `0`, which means that default only happens when the response is equal to `1`.

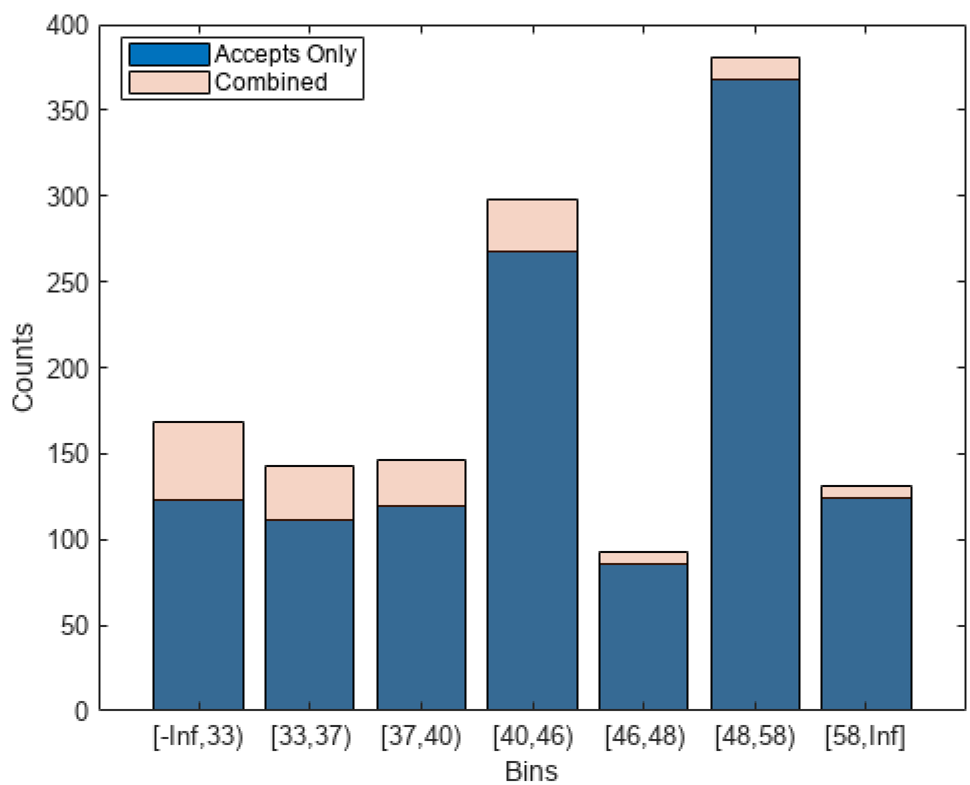
```
% Define the BR
BR = 0.75 ;

% Sort rejects by ascending CustID order
N = height(Rejects);
Rejects = sortrows(Rejects);
ScoresRejects = score(scHC,Rejects);

% Find the lowest quantile based on the BR and set the corresponding observations to bad
BadLabel   = setdiff(unique(scHC.Data.(scHC.ResponseVar)),scHC.GoodLabel);
ScoreThres = quantile(ScoresRejects,BR);
ResponseRejects = zeros(N,1);
ResponseRejects(ScoresRejects < ScoreThres)  = BadLabel;
ResponseRejects(ScoresRejects >= ScoreThres) = scHC.GoodLabel;

% Create the rejects table
RejectsTable    = [Rejects table(ResponseRejects,'VariableNames',{scHC.ResponseVar})];
```

Combine Accepts and Rejects Into a New Data Set, Score, and Validate

To draw a more accurate comparison between the accepts and the combined data set, use the same binning rules from the initial accepts credit scorecard and copy them to the `creditscorecard` object built on the combined dataset. This ensures that the binning assignment does not affect the later comparison of the two credit scorecard models. Also, you can visualize how the rejects are spread out in the data range of each predictor.

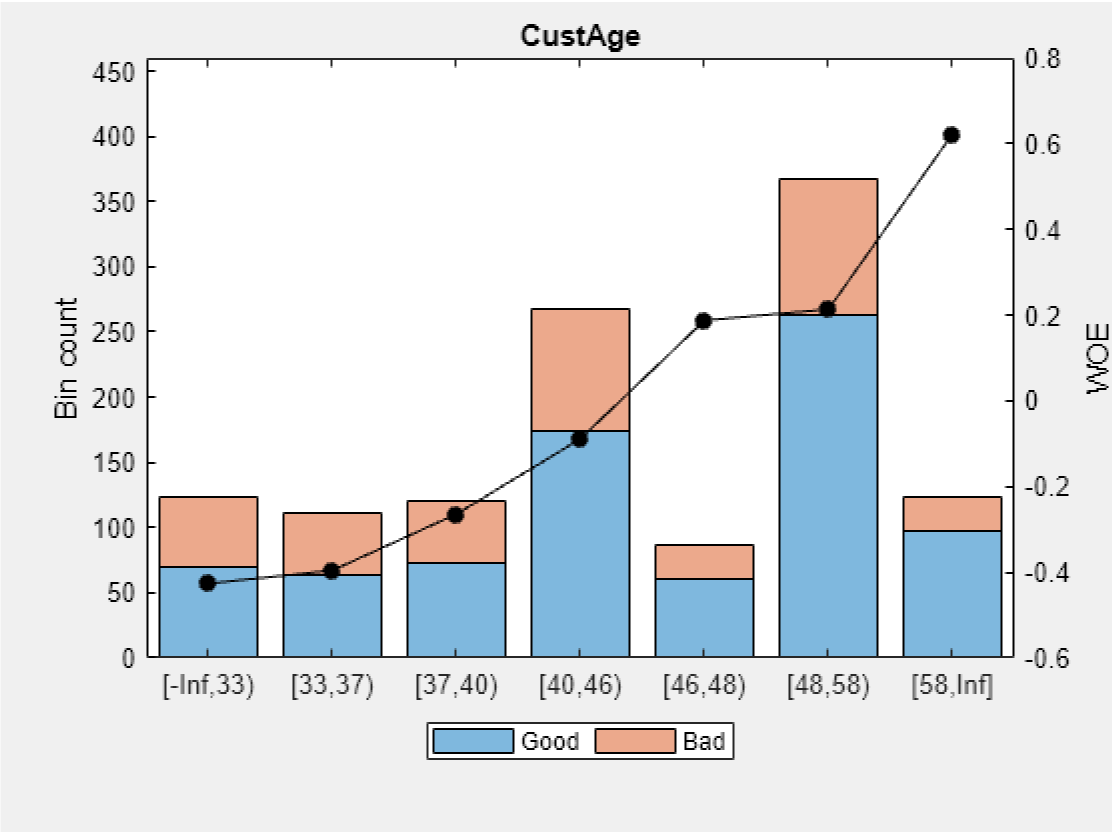
```
% Create the final combined scorecard
CombinedData = [data(:,2:end);RejectsTable(:,2:end)];
scNewHC = creditscorecard(CombinedData,'GoodLabel',0);

% Bin using the same binning rules as the base scorecard
Predictors = scHC.PredictorVars;
```

```
Edges = struct();
for i = 1 : length(Predictors)
    Pred = Predictors{i};
    [bi,cp] = bininfo(scHC,Pred);
    if ismember(Pred,scHC.NumericPredictors)
        scNewHC = modifybins(scNewHC,Pred,'CutPoints',cp);
    else
        scNewHC = modifybins(scNewHC,Pred,'CatGrouping',cp);
    end
    Edges.(Pred) = bi.Bin(1:end-1);
end

% Visualize the rejects distribution in each bin
bd1 = bindata(scHC,data);
bd2 = bindata(scHC,CombinedData);

Predictor = CustAge ;
figure;
bar(categorical(Edges.(Predictor)),histcounts(bd1.(Predictor)))
hold on
bar(categorical(Edges.(Predictor)),histcounts(bd2.(Predictor)),'FaceAlpha',0.25)
hold off
xlabel('Bins')
ylabel('Counts')
legend({'Accepts Only','Combined'},'Location','best')
```

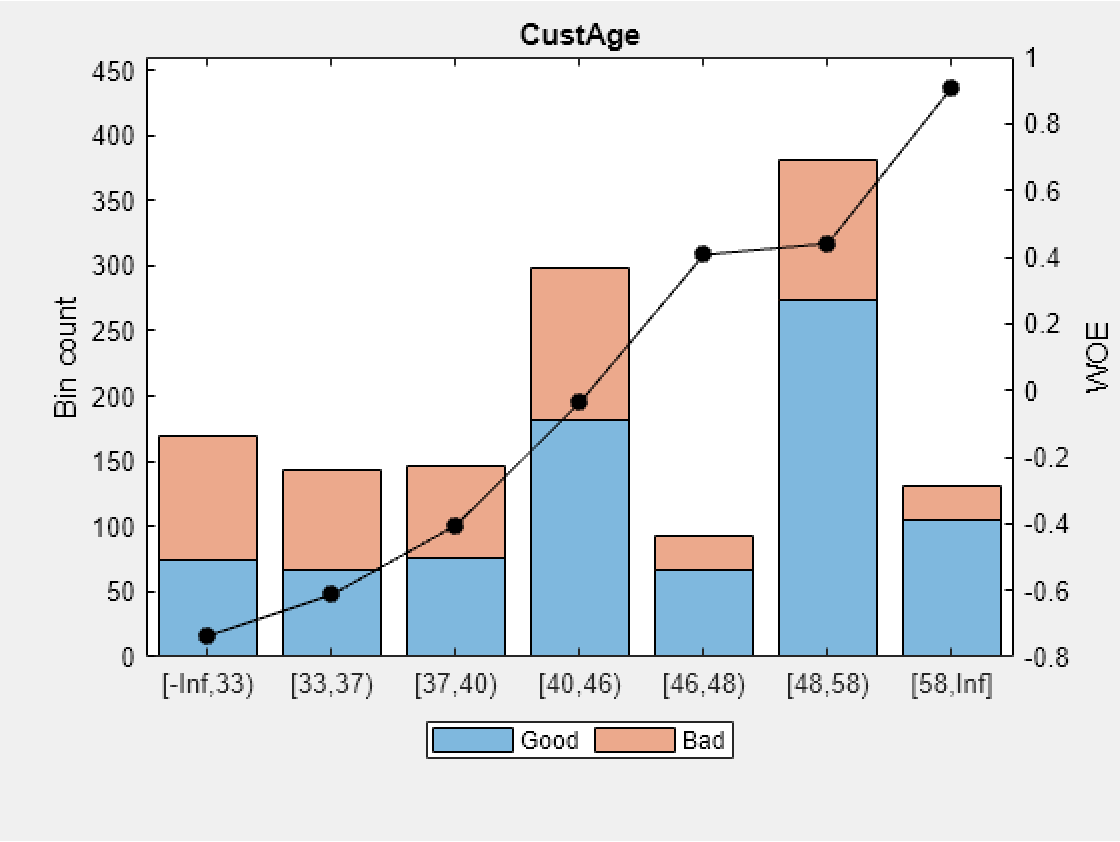


Compare the initial `creditscorecard` object (`scHC`) to the new `creditscorecard` object (`scNewHC`) for the distribution of "goods" and "bads" for the selected predictor.

```
plotbins(scHC,Predictor);
```




```
plotbins(scNewHC,Predictor);
```



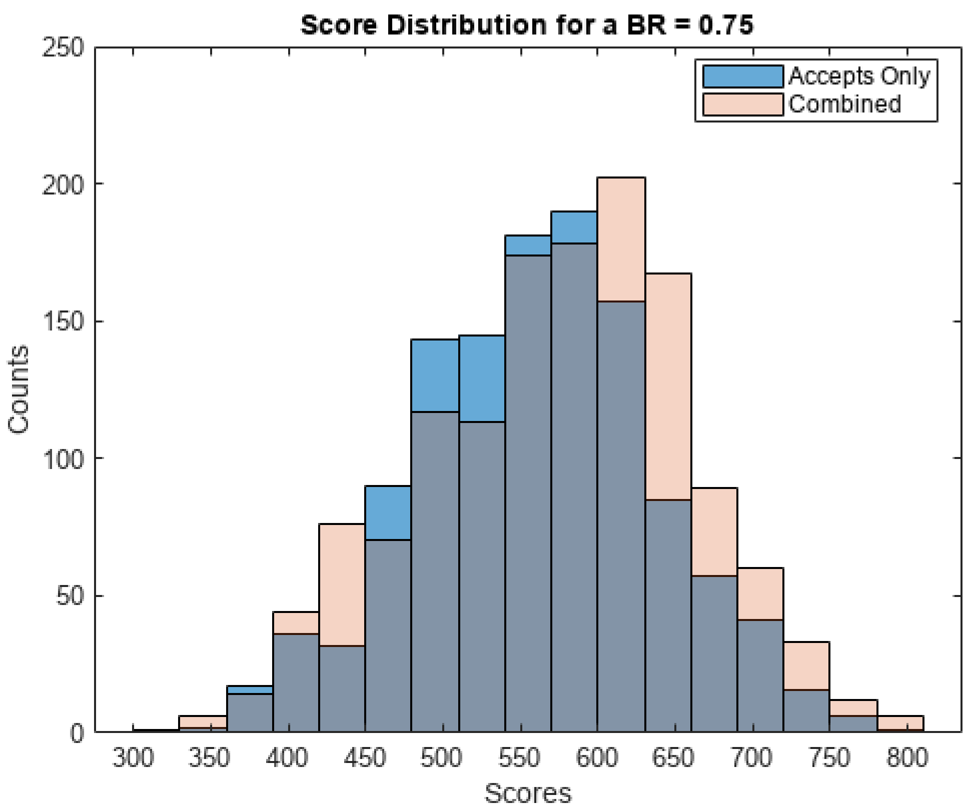
Fit a logistic regression model for the `creditscorecard` object `scNewHC` and then score `scNewHC`.

```
scNewHC = fitmodel(scNewHC);
```

```
1. Adding CustIncome, Deviance = 1693.9882, Chi2Stat = 114.39516, PValue = 1.0676416e-26
2. Adding TmWBank, Deviance = 1650.6615, Chi2Stat = 43.326628, PValue = 4.6323638e-11
3. Adding AMBalance, Deviance = 1623.0668, Chi2Stat = 27.594773, PValue = 1.4958244e-07
4. Adding EmpStatus, Deviance = 1603.603, Chi2Stat = 19.463733, PValue = 1.0252802e-05
5. Adding CustAge, Deviance = 1592.3467, Chi2Stat = 11.256272, PValue = 0.00079354409
6. Adding ResStatus, Deviance = 1582.0086, Chi2Stat = 10.338134, PValue = 0.0013030966
7. Adding OtherCC, Deviance = 1572.1, Chi2Stat = 9.9086387, PValue = 0.0016450476

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.48115     0.061301     7.849    4.1925e-15
    CustAge        0.50857      0.14449    3.5197    0.00043207
    ResStatus        1.151      0.34773    3.3101    0.00093262
    EmpStatus      0.78527      0.17826    4.4051    1.0572e-05
    CustIncome     0.68743      0.12372    5.5563    2.7555e-08
    TmWBank         1.0001      0.16731    5.9779    2.2607e-09
    OtherCC        0.97659      0.30956    3.1548     0.0016062
    AMBalance      0.91563      0.19073    4.8006    1.5819e-06
```

```
1361 observations, 1353 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 236, p-value = 2.29e-47
```

```
scNewHC  = formatpoints(scNewHC,'WorstAndBestScores',ScoreRange);
Scores = score(scNewHC);

% Visualize the score distribution
histogram(ScoresAccepts)
hold on
histogram(Scores,'FaceAlpha',0.25)
hold off
ylabel('Counts')
xlabel('Scores')
title(sprintf('Score Distribution for a BR = %.2f',BR))
legend({'Accepts Only','Combined'},'Location','best')
```



Validate the Model on the Combined Data Set

Before validation, you must adjust the data set. To adjust the data set, you can either:

- Validate the accepts for both scorecards
- Validate the combined data set for both scorecards

```
% Get statistics for the accepts
StatsA1 = validatemodel(scHC);
```

```
StatsA2 = validatemodel(scNewHC,data);

% Get the statistics for the combined data set
StatsC1 = validatemodel(scHC,CombinedData);
StatsC2 = validatemodel(scNewHC);

s1 = table(StatsA1.Value,StatsA2.Value,'VariableNames',{'BaseScorecard','CombinedScorecard'});
s2 = table(StatsC1.Value,StatsC2.Value,'VariableNames',{'BaseScorecard','CombinedScorecard'});
Stats = table(StatsA1.Measure,s1,s2,'VariableNames',{'Measure','Accepts','Combined'});
disp(Stats)
```

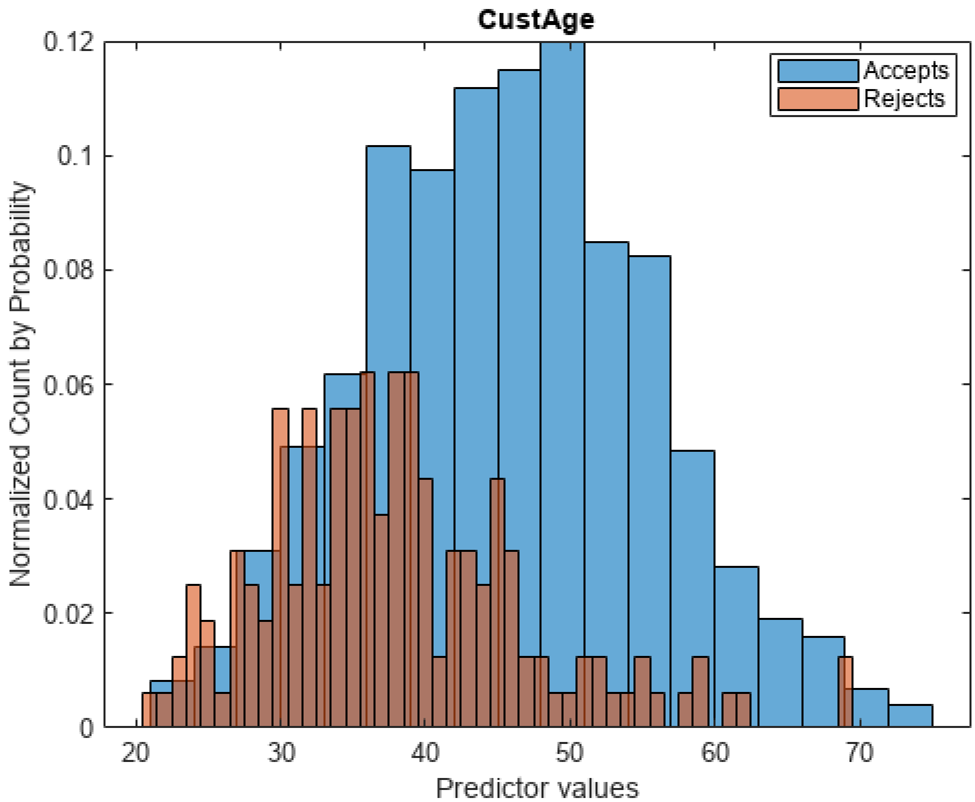
```
           Measure                          Accepts                              Combined
                                BaseScorecard    CombinedScorecard    BaseScorecard    CombinedSc
    ________________________    _________________________________    _____________________________

    {'Accuracy Ratio'      }       0.32258            0.31695            0.47022            0.465
    {'Area under ROC curve'}       0.66129            0.65848            0.73511            0.732
    {'KS statistic'        }        0.2246            0.22946            0.34528            0.336
    {'KS score'            }        550.72             576.57             512.44            542
```

Fuzzy Augmentation Technique Workflow

The *Fuzzy augmentation* technique starts by building a scorecard using the accepts only and then this scorecard model is used to score the rejects. Unlike the hard-cutoff technique, the fuzzy augmentation approach does not assign "good" or "bad" classes. Rather, each reject is duplicated into two observations and assigned a weighted "good" or "bad" value, based on a probability of being "good" or "bad." The weighted rejects are then added to the accepts data set and the combined data set is used to create a scorecard that is then fit and validated.

First, visualize the data for accepts and rejects for a selected predictor.

```
% Load the data
matFileName = fullfile(matlabroot,'toolbox','finance','findemos','CreditCardData');
load(matFileName)
load RejectsCreditCardData.mat

Predictor = CustAge ;
figure;
h1 = histogram(data.(Predictor));
hold on
h2 = histogram(Rejects.(Predictor));
h1.Normalization = 'probability';
h2.Normalization = 'probability';
title(Predictor)
xlabel('Predictor values')
ylabel('Normalized Count by Probability')
hold off
legend({'Accepts','Rejects'},'Location','best');
```



Create a creditscorecard Object for the Accepts and Score the Data

Use `creditscorecard` to create a `creditscorecard` object for the accepts, which you can bin, fit, and then score.

```
scFA = creditscorecard(data,'IDVar','CustID');
scFA = autobinning(scFA);
scFA = fitmodel(scFA);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)     0.70239    0.064001    10.975    5.0538e-28
    CustAge         0.60833     0.24932      2.44      0.014687
    ResStatus         1.377     0.65272    2.1097      0.034888
```

```
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

```
ScoreRange = [300 850];
scFA = formatpoints(scFA,'WorstAndBestScores',ScoreRange);
ScoresAccepts = score(scFA);
```

Score the Rejects and Create the Combined Data Set

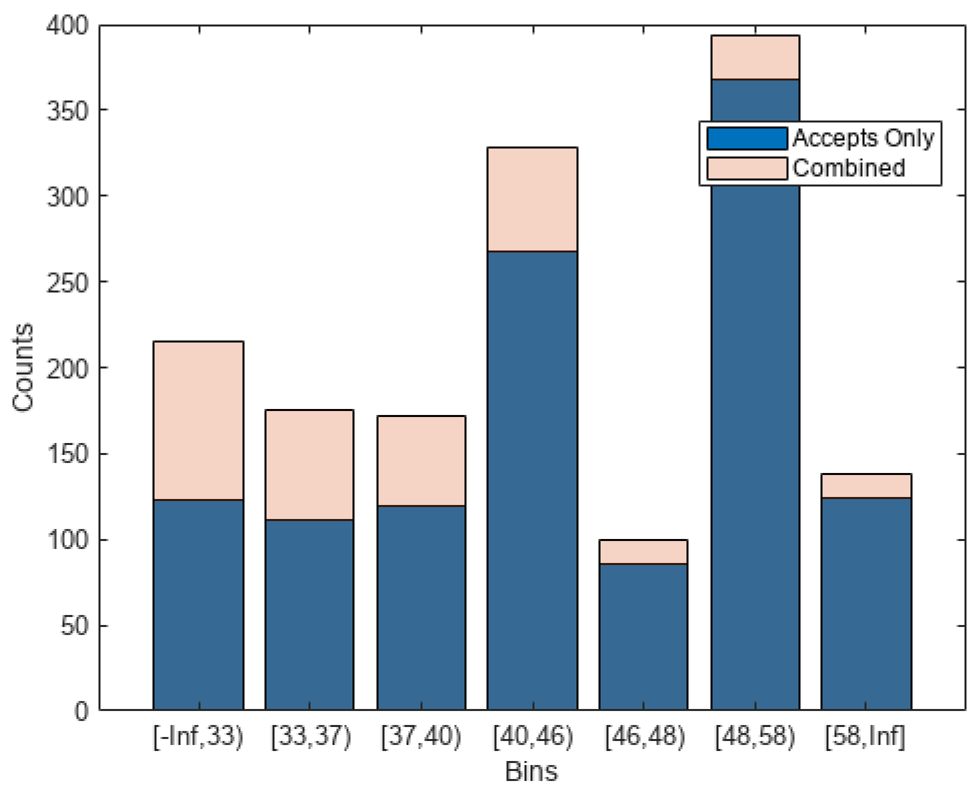
```
% Load the rejects dataset and score the observations
load RejectsCreditCardData.mat

ScoresRejects = score(scFA,Rejects);

% Compute the probabilities of default and use as weights
pdRejects = probdefault(scFA,Rejects);

% Assign bad status to pd (probability of default) and good status to 1-pd weights
BadLabel = setdiff(unique(scFA.Data.(scFA.ResponseVar)),scFA.GoodLabel);
Weights  = zeros(2*length(pdRejects),1);
Response = zeros(2*length(pdRejects),1);
Weights(1:2:end)  = pdRejects;
Response(1:2:end) = BadLabel;
Weights(2:2:end)  = 1-pdRejects;
Response(2:2:end) = scFA.GoodLabel;

% Rearrange the response so that each two rows correspond to the same
% observation from rejects
RejectsTable = repelem(Rejects(:,2:end),2,1);
RejectsTable = addvars(RejectsTable,Weights,Response,'NewVariableNames',...
    {'Weights',scFA.ResponseVar});

% Combine accepts and rejects
AcceptsData = addvars(data,ones(height(data),1),'Before',scFA.ResponseVar,...
    'NewVariableNames','Weights');
CombinedData = [AcceptsData(:,2:end);RejectsTable];
```

Combine Accepts and Rejects into a New Data Set, Score, and Validate

To draw a more accurate comparison between the accepts and the combined data set, use the same binning rules from the initial accepts credit scorecard and copy them to the `creditscorecard` object built on the combined dataset. This ensures that the binning assignments does not affect the later comparison of the two credit scorecard models. Also, you can visualize how the rejects are spread out in the data range of each predictor.

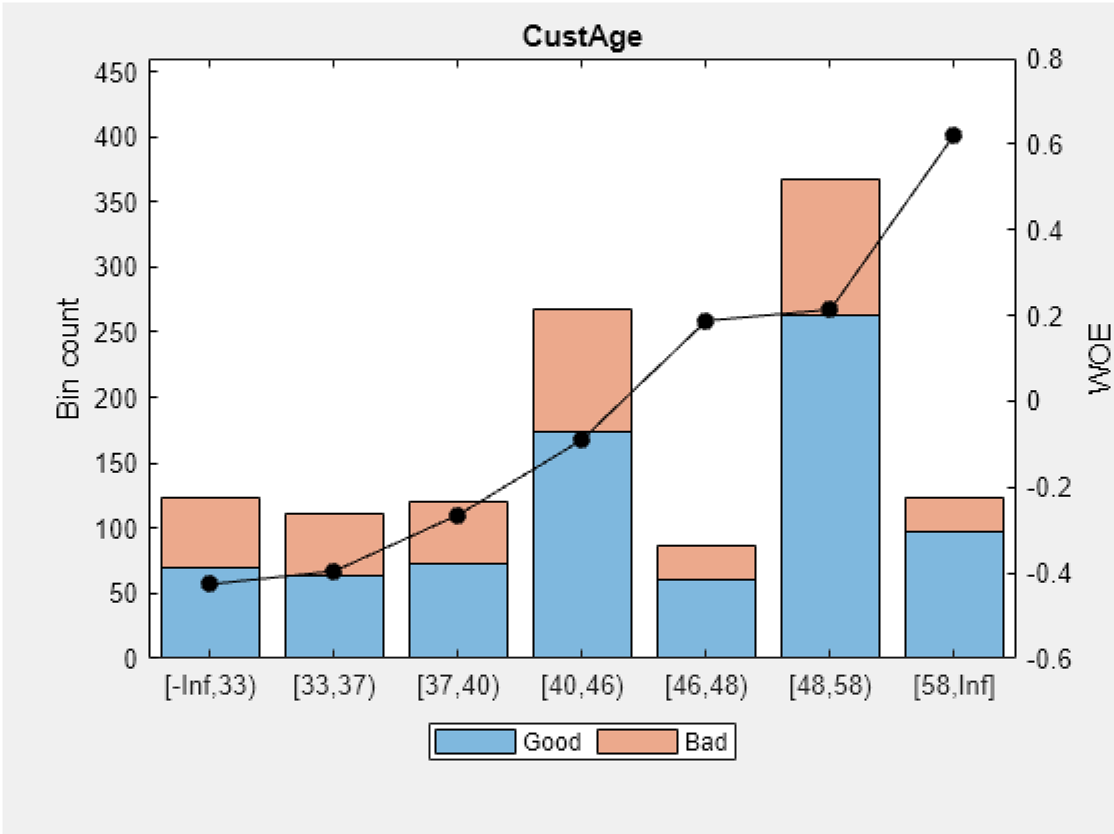
```
scNewFA = creditscorecard(CombinedData,'GoodLabel',0,'WeightsVar','Weights');

% Bin using the same binning rules as the base scorecard
Predictors = scFA.PredictorVars;
Edges = struct();
```

```
for i = 1 : length(Predictors)
    Pred = Predictors{i};
    [bi,cp] = bininfo(scFA,Pred);
    if ismember(Pred,scFA.NumericPredictors)
        scNewFA = modifybins(scNewFA,Pred,'CutPoints',cp);
    else
        scNewFA = modifybins(scNewFA,Pred,'CatGrouping',cp);
    end
    Edges.(Pred) = bi.Bin(1:end-1);
end

% Visualize the rejects distribution in each bin
bd1 = bindata(scFA,data);
bd2 = bindata(scFA,CombinedData);

Predictor = CustAge ;
figure;
bar(categorical(Edges.(Predictor)),histcounts(bd1.(Predictor)))
hold on
bar(categorical(Edges.(Predictor)),histcounts(bd2.(Predictor)),'FaceAlpha',0.25)
hold off
xlabel('Bins')
ylabel('Counts')
legend({'Accepts Only','Combined'},'Location','best')
```

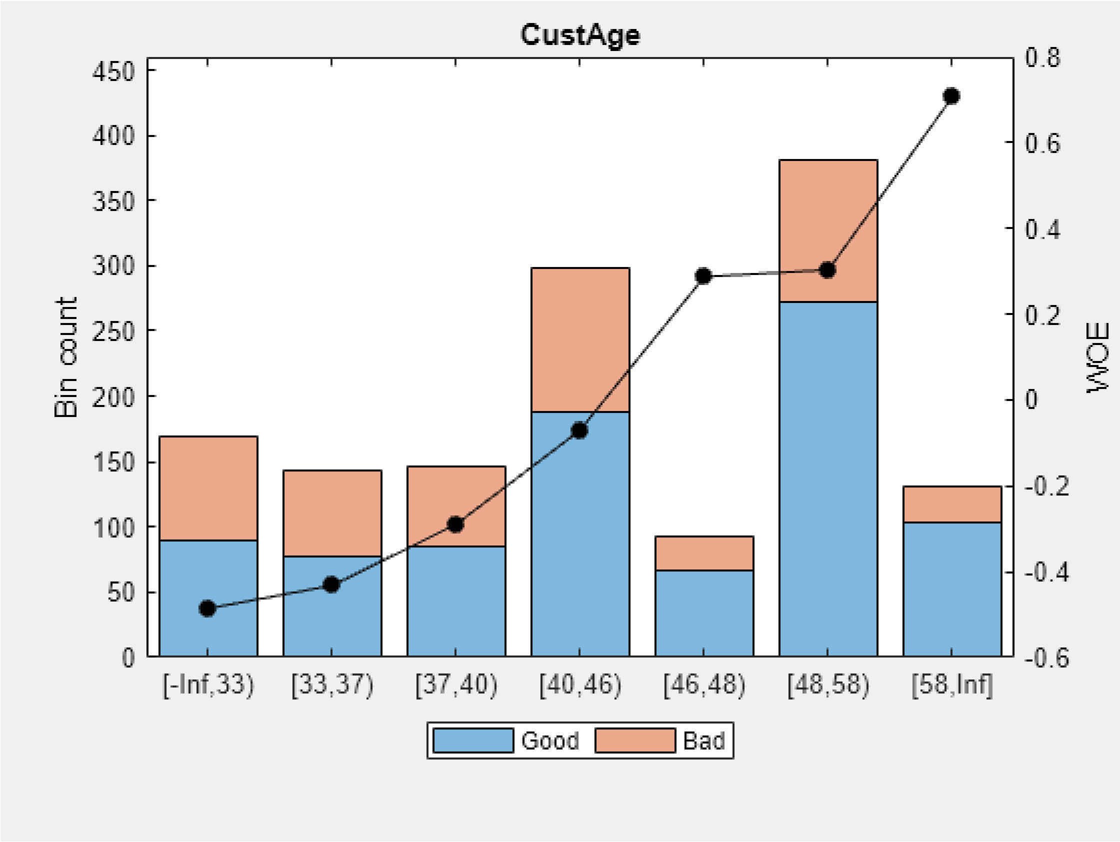


Compare the initial `creditscorecard` object (`scFA`) to the new `creditscorecard` object (`scNewFA`) for the distribution of "goods" and "bads" for the selected predictor.

```
plotbins(scFA,Predictor);
```




```
plotbins(scNewFA,Predictor);
```

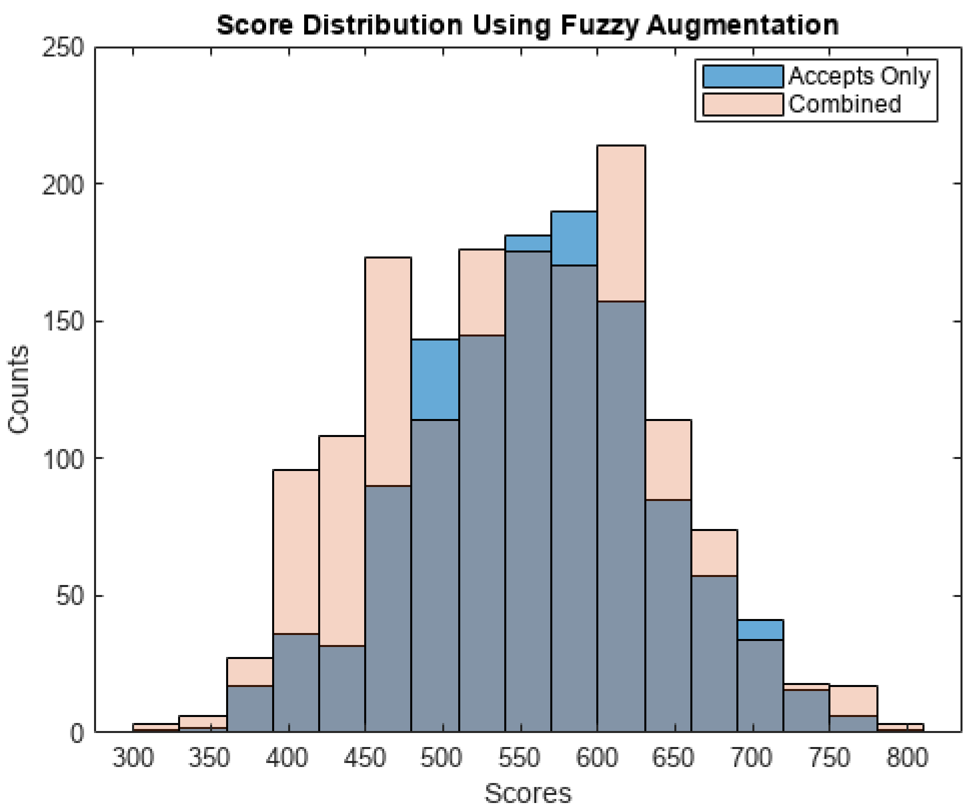


Fit a logistic regression model for the `creditscorecard` object `scNewFA` and then score `scNewFA`.

```
scNewFA = fitmodel(scNewFA);
```

```
1. Adding CustIncome, Deviance = 1711.3102, Chi2Stat = 54.160619, PValue = 1.8475277e-13
2. Adding TmWBank, Deviance = 1682.5353, Chi2Stat = 28.774866, PValue = 8.1299351e-08
3. Adding AMBalance, Deviance = 1668.2956, Chi2Stat = 14.239727, PValue = 0.00016093686
4. Adding EmpStatus, Deviance = 1658.2944, Chi2Stat = 10.001236, PValue = 0.001564352
5. Adding CustAge, Deviance = 1652.3976, Chi2Stat = 5.8967925, PValue = 0.015168483
6. Adding OtherCC, Deviance = 1647.7632, Chi2Stat = 4.6344022, PValue = 0.031337059
7. Adding ResStatus, Deviance = 1642.8332, Chi2Stat = 4.9299914, PValue = 0.026394448

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.60838     0.059654    10.198    2.0142e-24
    CustAge        0.50755      0.20092    2.5262      0.011532
    ResStatus        1.082      0.48919    2.2119      0.026971
    EmpStatus      0.74776      0.23526    3.1784     0.0014809
    CustIncome      0.6372      0.17519    3.6371    0.00027567
    TmWBank        0.96561      0.19664    4.9106    9.0815e-07
    OtherCC        0.90699      0.40476    2.2408      0.025039
    AMBalance      0.87642      0.25404    3.4499    0.00056077
```

```
1522 observations, 1514 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 123, p-value = 2.16e-23

scNewFA  = formatpoints(scNewFA,'WorstAndBestScores',ScoreRange);
Scores = score(scNewFA);
pd = probdefault(scNewFA);

% Visualize the score distribution
histogram(ScoresAccepts)
hold on
histogram(Scores,'FaceAlpha',0.25)
hold off
ylabel('Counts')
xlabel('Scores')
title('Score Distribution Using Fuzzy Augmentation')
legend({'Accepts Only','Combined'},'Location','best')
```



Validate the Model on the Combined Data Set

Before validation, you must adjust the data set. To adjust the data set, you can either:

- Validate the accepts for both scorecards
- Validate the combined data set for both scorecards

```
% Get statistics for the accepts
data.Weights = ones(height(data),1);
StatsA1 = validatemodel(scFA);
StatsA2 = validatemodel(scNewFA,data);
% Get the statistics for the combined data set
StatsC1 = validatemodel(scFA,CombinedData);
StatsC2 = validatemodel(scNewFA);

s1 = table(StatsA1.Value,StatsA2.Value,'VariableNames',{'BaseScorecard','CombinedScorecard'});
s2 = table(StatsC1.Value,StatsC2.Value,'VariableNames',{'BaseScorecard','CombinedScorecard'});
Stats = table(StatsA1.Measure,s1,s2,'VariableNames',{'Measure','Accepts','Combined'});
disp(Stats)
```

```
           Measure                             Accepts                                 Combined
                                  BaseScorecard    CombinedScorecard    BaseScorecard    CombinedSc
    ________________________     ____________________________________    ___________________________

    {'Accuracy Ratio'      }        0.32258            0.32088             0.29419            0.351
    {'Area under ROC curve'}        0.66129            0.66044             0.64709            0.675
    {'KS statistic'        }         0.2246            0.22799             0.22596            0.256
    {'KS score'            }         550.72             554.84              512.44              520
```

Summary

This example demonstrates how to use a reject inference process within the framework of the credit scorecard workflow. The Hard-Cutoff and the Fuzzy Augmentation techniques show how you can bin the data, fit a model, integrate the rejects with the accepts into a new credit scorecard model, and then validate the new credit scorecard model.

There is no clear-cut conclusion for which of these reject inference approaches is the best. This example is intended to illustrate how to use the features of `creditscorecard` to implement two different reject inference approaches.

References

1. Baesesn, B., D. Rösch, and H. Scheule. *Credit Risk Analytics: Measurement Techniques, Applications and Examples in SAS*. Wiley and SAS Business Series, 2016.
2. Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`creditscorecard` | `screenpredictors` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Related Examples

- “Common Binning Explorer Tasks” on page 3-4
- “Credit Scorecard Modeling with Missing Values”
- “Feature Screening with screenpredictors” on page 3-64
- “Troubleshooting Credit Scorecard Results”
- “Credit Rating by Bagging Decision Trees”
- “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36

More About

- “Overview of Binning Explorer” on page 3-2
- “About Credit Scorecards”
- “Credit Scorecard Modeling Workflow”
- Monotone Adjacent Pooling Algorithm (MAPA)
- “Credit Scorecard Modeling Using Observation Weights”

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

Credit Scoring Using Logistic Regression and Decision Trees

Create and compare two credit scoring models, one based on logistic regression and the other based on decision trees.

Credit rating agencies and banks use challenger models to test the credibility and goodness of a credit scoring model. In this example, the base model is a logistic regression model and the challenger model is a decision tree model.

Logistic regression links the score and probability of default (PD) through the logistic regression function, and is the default fitting and scoring model when you work with `creditscorecard` objects. However, decision trees have gained popularity in credit scoring and are now commonly used to fit data and predict default. The algorithms in decision trees follow a top-down approach where, at each step, the variable that splits the dataset "best" is chosen. "Best" can be defined by any one of several metrics, including the Gini index, information value, or entropy. For more information, see “Decision Trees”.

In this example, you:

- Use both a logistic regression model and a decision tree model to extract PDs.
- Validate the challenger model by comparing the values of key metrics between the challenger model and the base model.

Compute Probabilities of Default Using Logistic Regression

First, create the base model by using a `creditscorecard` object and the default logistic regression function `fitmodel`. Fit the `creditscorecard` object by using the full model, which includes all predictors for the generalized linear regression model fitting algorithm. Then, compute the PDs using `probdefault`. For a detailed description of this workflow, see “Case Study for Credit Scorecard Analysis”.

```
% Create a creditscorecard object, bin data, and fit a logistic regression model
load CreditCardData.mat
scl = creditscorecard(data,'IDVar','CustID');
scl = autobinning(scl);
scl = fitmodel(scl,'VariableSelection','fullmodel');
```

```
Generalized linear regression model:
    logit(status) ~ 1 + CustAge + TmAtAddress + ResStatus + EmpStatus + CustIncome + TmWBank + O
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue  
                   ________    ________    ________    __________

    (Intercept)     0.70246    0.064039      10.969    5.3719e-28
    CustAge          0.6057     0.24934      2.4292      0.015131
    TmAtAddress      1.0381     0.94042      1.1039       0.26963
    ResStatus        1.3794      0.6526      2.1137      0.034538
    EmpStatus       0.89648     0.29339      3.0556     0.0022458
    CustIncome      0.70179     0.21866      3.2095     0.0013295
    TmWBank          1.1132     0.23346      4.7683    1.8579e-06
    OtherCC          1.0598     0.53005      1.9994      0.045568
    AMBalance        1.0572     0.36601      2.8884     0.0038718
```

```
    UtilRate       -0.047597     0.61133    -0.077858       0.93794


1200 observations, 1190 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 91, p-value = 1.05e-15
```

```
% Compute the corresponding probabilities of default
pdL = probdefault(scl);
```

Compute Probabilities of Default Using Decision Trees

Next, create the challenger model. Use the Statistics and Machine Learning Toolbox™ method `fitctree` to fit a Decision Tree (DT) to the data. By default, the splitting criterion is Gini's diversity index. In this example, the model is an input argument to the function, and the response `'status'` comprises all predictors when the algorithm starts. For this example, see the name-value pairs in `fitctree` to the maximum number of splits to avoid overfitting and specify the predictors as categorical.

```
% Create and view classification tree
CategoricalPreds = {'ResStatus','EmpStatus','OtherCC'};
dt = fitctree(data,'status~CustAge+TmAtAddress+ResStatus+EmpStatus+CustIncome+TmWBank+OtherCC+Uti
    'MaxNumSplits',30,'CategoricalPredictors',CategoricalPreds);
disp(dt)
```

```
  ClassificationTree
           PredictorNames: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
             ResponseName: 'status'
    CategoricalPredictors: [3 4 7]
               ClassNames: [0 1]
           ScoreTransform: 'none'
          NumObservations: 1200
```

The decision tree is shown below. You can also use the `view` function with the name-value pair argument `'mode'` set to `'graph'` to visualize the tree as a graph.

```
view(dt)
```

```
Decision tree for classification
 1  if CustIncome<30500 then node 2 elseif CustIncome>=30500 then node 3 else 0
 2  if TmWBank<60 then node 4 elseif TmWBank>=60 then node 5 else 1
 3  if TmWBank<32.5 then node 6 elseif TmWBank>=32.5 then node 7 else 0
 4  if TmAtAddress<13.5 then node 8 elseif TmAtAddress>=13.5 then node 9 else 1
 5  if UtilRate<0.255 then node 10 elseif UtilRate>=0.255 then node 11 else 0
 6  if CustAge<60.5 then node 12 elseif CustAge>=60.5 then node 13 else 0
 7  if CustAge<46.5 then node 14 elseif CustAge>=46.5 then node 15 else 0
 8  if CustIncome<24500 then node 16 elseif CustIncome>=24500 then node 17 else 1
 9  if TmWBank<56.5 then node 18 elseif TmWBank>=56.5 then node 19 else 1
10  if CustAge<21.5 then node 20 elseif CustAge>=21.5 then node 21 else 0
11  class = 1
12  if EmpStatus=Employed then node 22 elseif EmpStatus=Unknown then node 23 else 0
13  if TmAtAddress<131 then node 24 elseif TmAtAddress>=131 then node 25 else 0
14  if TmAtAddress<97.5 then node 26 elseif TmAtAddress>=97.5 then node 27 else 0
15  class = 0
16  class = 0
17  if ResStatus in {Home Owner Tenant} then node 28 elseif ResStatus=Other then node 29 else 1
18  if TmWBank<52.5 then node 30 elseif TmWBank>=52.5 then node 31 else 0
19  class = 1
```

```
20  class = 1
21  class = 0
22  if UtilRate<0.375 then node 32 elseif UtilRate>=0.375 then node 33 else 0
23  if UtilRate<0.005 then node 34 elseif UtilRate>=0.005 then node 35 else 0
24  if CustIncome<39500 then node 36 elseif CustIncome>=39500 then node 37 else 0
25  class = 1
26  if UtilRate<0.595 then node 38 elseif UtilRate>=0.595 then node 39 else 0
27  class = 1
28  class = 1
29  class = 0
30  class = 1
31  class = 0
32  class = 0
33  if UtilRate<0.635 then node 40 elseif UtilRate>=0.635 then node 41 else 0
34  if CustAge<49 then node 42 elseif CustAge>=49 then node 43 else 1
35  if CustIncome<57000 then node 44 elseif CustIncome>=57000 then node 45 else 0
36  class = 1
37  class = 0
38  class = 0
39  if CustIncome<34500 then node 46 elseif CustIncome>=34500 then node 47 else 1
40  class = 1
41  class = 0
42  class = 1
43  class = 0
44  class = 0
45  class = 1
46  class = 0
47  class = 1
```

When you use `fitctree`, you can adjust the “Name-Value Pair Arguments” depending on your use case. For example, you can set a small minimum leaf size, which yields a better accuracy ratio (see Model Validation on page 3-94) but can result in an overfitted model.

The decision tree has a `predict` function that, when used with a second and third output argument, gives valuable information.

```
% Extract probabilities of default
[~,ObservationClassProb,Node] = predict(dt,data);
pdDT = ObservationClassProb(:,2);
```

This syntax has the following outputs:

- `ObservationClassProb` returns a `NumObs`-by-2 array with class probability at all observations. The order of the classes is the same as in `dt.ClassName`. In this example, the class names are `[0 1]` and the good label, by choice, based on which class has the highest count in the raw data, is `0`. Therefore, the first column corresponds to nondefaults and the second column to the actual PDs. The PDs are needed later in the workflow for scoring or validation.
- `Node` returns a `NumObs`-by-`1` vector containing the node numbers corresponding to the given observations.

Predictor Importance

In predictor (or variable) selection, the goal is to select as few predictors as possible while retaining as much information (predictive accuracy) about the data as possible. In the `creditscorecard` class, the `fitmodel` function internally selects predictors and returns *p*-values for each predictor. The analyst can then, outside the `creditscorecard` workflow, set a threshold for these *p*-values and choose the predictors worth keeping and the predictors to discard. This step is useful when the number of predictors is large.

Typically, training datasets are used to perform predictor selection. The key objective is to find the best set of predictors for ranking customers based on their likelihood of default and estimating their PDs.

Using Logistic Regression for Predictor Importance

Predictor importance is related to the notion of predictor weights, since the weight of a predictor determines how important it is in the assignment of the final score, and therefore, in the PD. Computing predictor weights is a back-of-the-envelope technique whereby the weights are determined by dividing the range of points for each predictor by the total range of points for the entire `creditscorecard` object. For more information on this workflow, see “Case Study for Credit Scorecard Analysis”.

For this example, use `formatpoints` with the option `PointsOddsandPDO` for scaling. This is not a necessary step, but it helps ensure that all points fall within a desired range (that is, nonnegative points). The `PointsOddsandPDO` scaling means that for a given value of `TargetPoints` and `TargetOdds` (usually `2`), the odds are "double", and then `formatpoints` solves for the scaling parameters such that `PDO` points are needed to double the odds.

```
% Choose target points, target odds, and PDO values
TargetPoints = 500;
TargetOdds = 2;
PDO = 50;

% Format points and compute points range
scl = formatpoints(scl,'PointsOddsAndPDO',[TargetPoints TargetOdds PDO]);
[PointsTable,MinPts,MaxPts] = displaypoints(scl);
PtsRange = MaxPts - MinPts;
disp(PointsTable(1:10,:))
```

```
      Predictors            Bin         Points
    _______________    ______________   ______

    {'CustAge'    }    {'[-Inf,33)'}    37.008
    {'CustAge'    }    {'[33,37)'  }    38.342
    {'CustAge'    }    {'[37,40)'  }    44.091
    {'CustAge'    }    {'[40,46)'  }    51.757
    {'CustAge'    }    {'[46,48)'  }    63.826
    {'CustAge'    }    {'[48,58)'  }     64.97
    {'CustAge'    }    {'[58,Inf]' }    82.826
    {'CustAge'    }    {'<missing>'}       NaN
    {'TmAtAddress'}    {'[-Inf,23)'}    49.058
    {'TmAtAddress'}    {'[23,83)'  }    57.325
```

```
fprintf('Minimum points: %g, Maximum points: %g\n',MinPts,MaxPts)
```

```
Minimum points: 348.705, Maximum points: 683.668
```

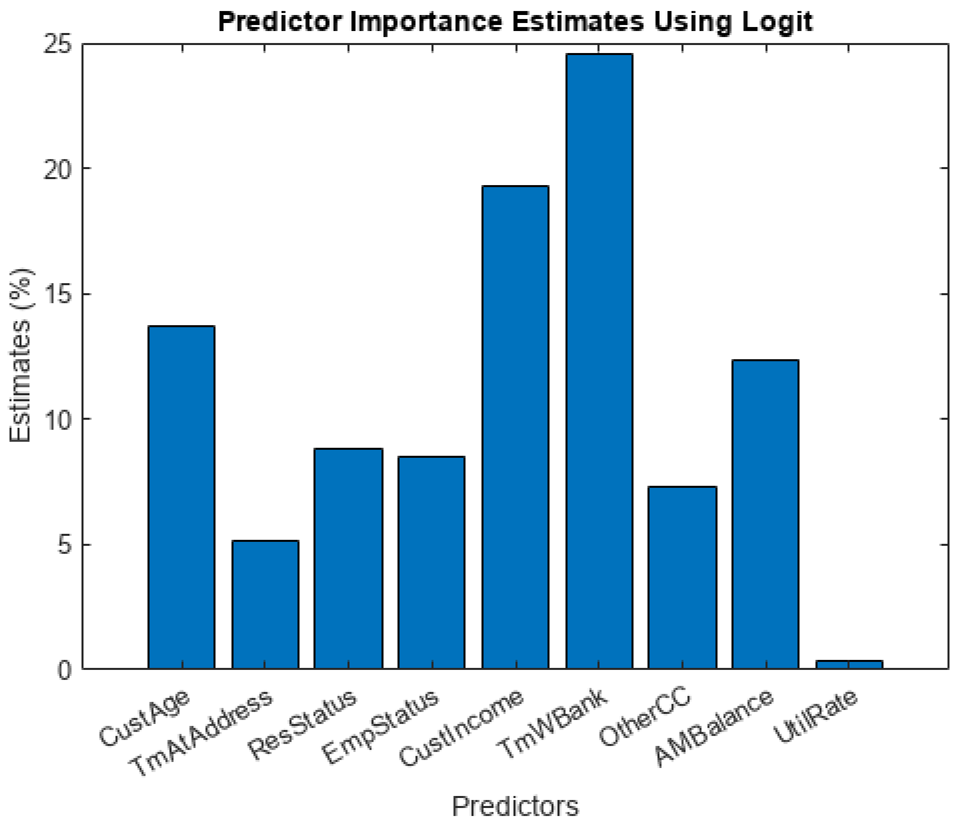
The weights are defined as the range of points, for any given predictor, divided by the range of points for the entire scorecard.

```
Predictor = unique(PointsTable.Predictors,'stable');
NumPred = length(Predictor);
```

```
Weight  = zeros(NumPred,1);

for ii = 1 : NumPred
   Ind = strcmpi(Predictor{ii},PointsTable.Predictors);
   MaxPtsPred = max(PointsTable.Points(Ind));
   MinPtsPred = min(PointsTable.Points(Ind));
   Weight(ii) = 100*(MaxPtsPred-MinPtsPred)/PtsRange;
end

PredictorWeights = table(Predictor,Weight);
PredictorWeights(end+1,:) = PredictorWeights(end,:);
PredictorWeights.Predictor{end} = 'Total';
PredictorWeights.Weight(end) = sum(Weight);
disp(PredictorWeights)
```

```
       Predictor        Weight 
    _______________    _______

    {'CustAge'    }     13.679
    {'TmAtAddress'}     5.1564
    {'ResStatus'  }     8.7945
    {'EmpStatus'  }      8.519
    {'CustIncome' }     19.259
    {'TmWBank'    }     24.557
    {'OtherCC'    }     7.3414
    {'AMBalance'  }     12.365
    {'UtilRate'   }    0.32919
    {'Total'      }        100
```

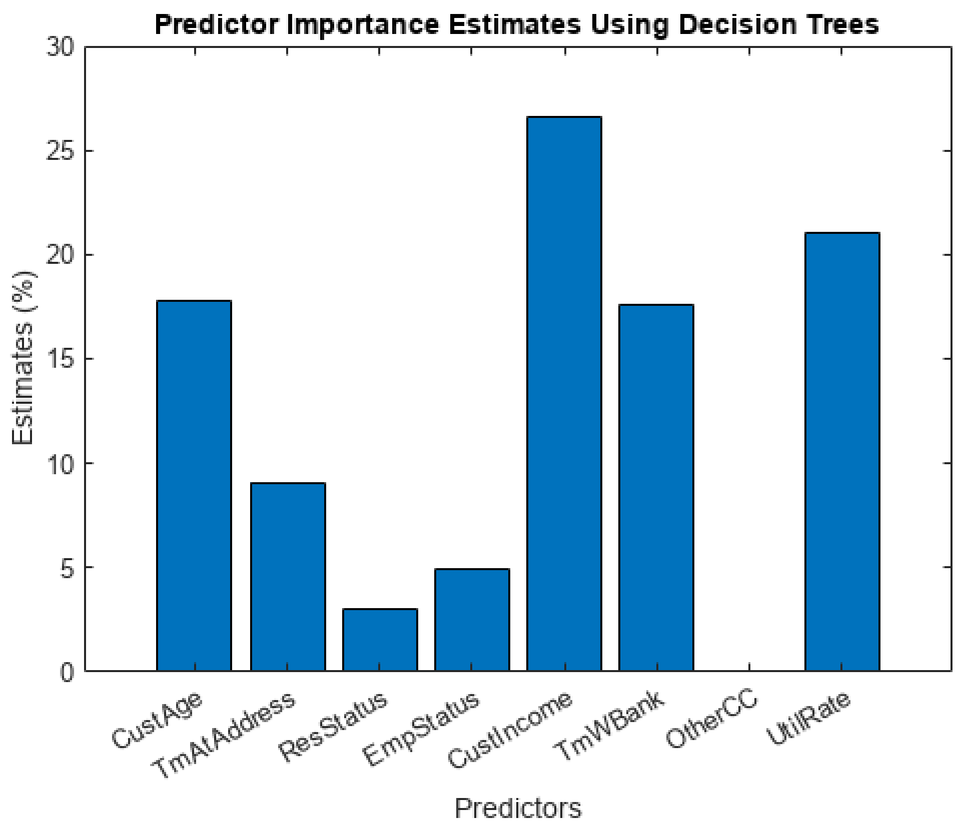
```
% Plot a histogram of the weights
figure
bar(PredictorWeights.Weight(1:end-1))
title('Predictor Importance Estimates Using Logit');
ylabel('Estimates (%)');
xlabel('Predictors');
xticklabels(PredictorWeights.Predictor(1:end-1));
```



Using Decision Trees for Predictor Importance

When you use decision trees, you can investigate predictor importance using the `predictorImportance` function. On every predictor, the function sums and normalizes changes in the risks due to splits by using the number of branch nodes. A high value in the output array indicates a strong predictor.

```
imp = predictorImportance(dt);

figure;
bar(100*imp/sum(imp)); % to normalize on a 0-100% scale
title('Predictor Importance Estimates Using Decision Trees');
ylabel('Estimates (%)');
xlabel('Predictors');
xticklabels(dt.PredictorNames);
```

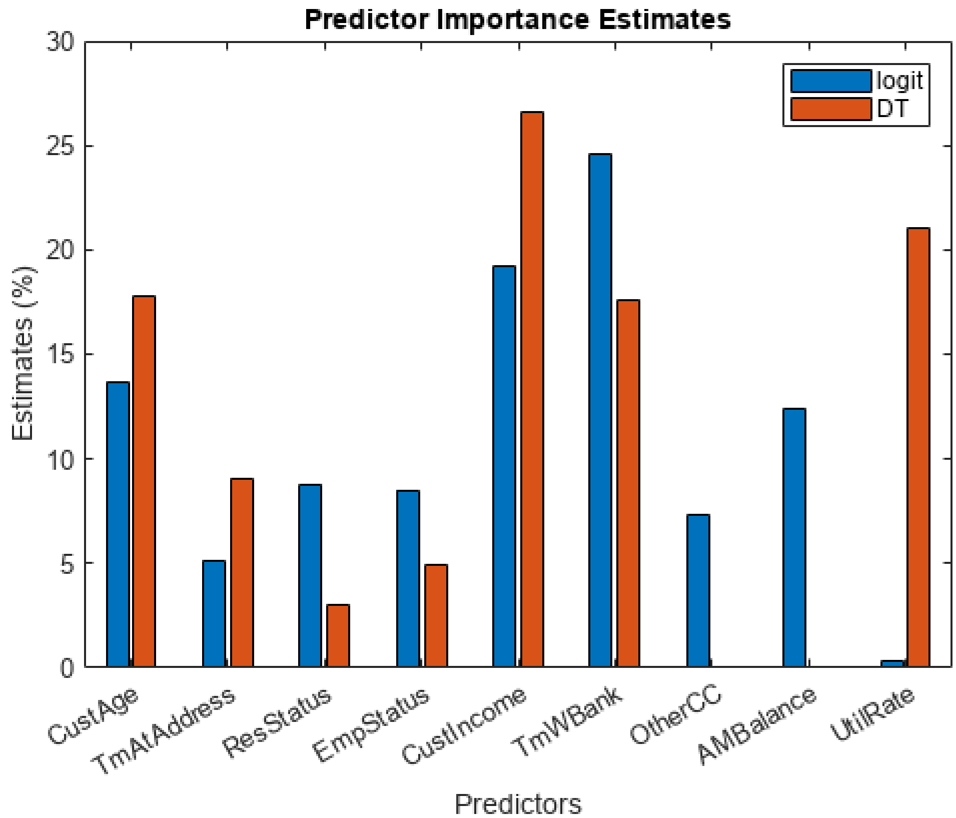


In this case, `'CustIncome'` (parent node) is the most important predictor, followed by `'UtilRate'`, where the second split happens, and so on. The predictor importance step can help in predictor screening for datasets with a large number of predictors.

Notice that not only are the weights across models different, but the selected predictors in each model also diverge. The predictors `'AMBalance'` and `'OtherCC'` are missing from the decision tree model, and `'UtilRate'` is missing from the logistic regression model.

Normalize the predictor importance for decision trees using a percent from 0 through 100%, then compare the two models in a combined histogram.

```
Ind = ismember(Predictor,dt.PredictorNames);
w = zeros(size(Weight));
w(Ind) = 100*imp'/sum(imp);
figure
bar([Weight,w]);
title('Predictor Importance Estimates');
ylabel('Estimates (%)');
xlabel('Predictors');
h = gca;
xticklabels(Predictor)
legend({'logit','DT'})
```



Note that these results depend on the binning algorithm you choose for the `creditscorecard` object and the parameters used in `fitctree` to build the decision tree.

Model Validation

The `creditscorecard` function `validatemodel` attempts to compute scores based on internally computed points. When you use decision trees, you cannot directly run a validation because the model coefficients are unknown and cannot be mapped from the PDs.

To validate the `creditscorecard` object using logistic regression, use the `validatemodel` function.

```
% Model validation for the creditscorecard
[StatsL,tL] = validatemodel(scl);
```

To validate decision trees, you can directly compute the statistics needed for validation.

```
% Compute the Area under the ROC
[x,y,t,AUC] = perfcurve(data.status,pdDT,1);
KSValue = max(y - x);
AR = 2 * AUC - 1;

% Create Stats table output
Measure = {'Accuracy Ratio','Area Under ROC Curve','KS Statistic'}';
Value  = [AR;AUC;KSValue];

StatsDT = table(Measure,Value);
```

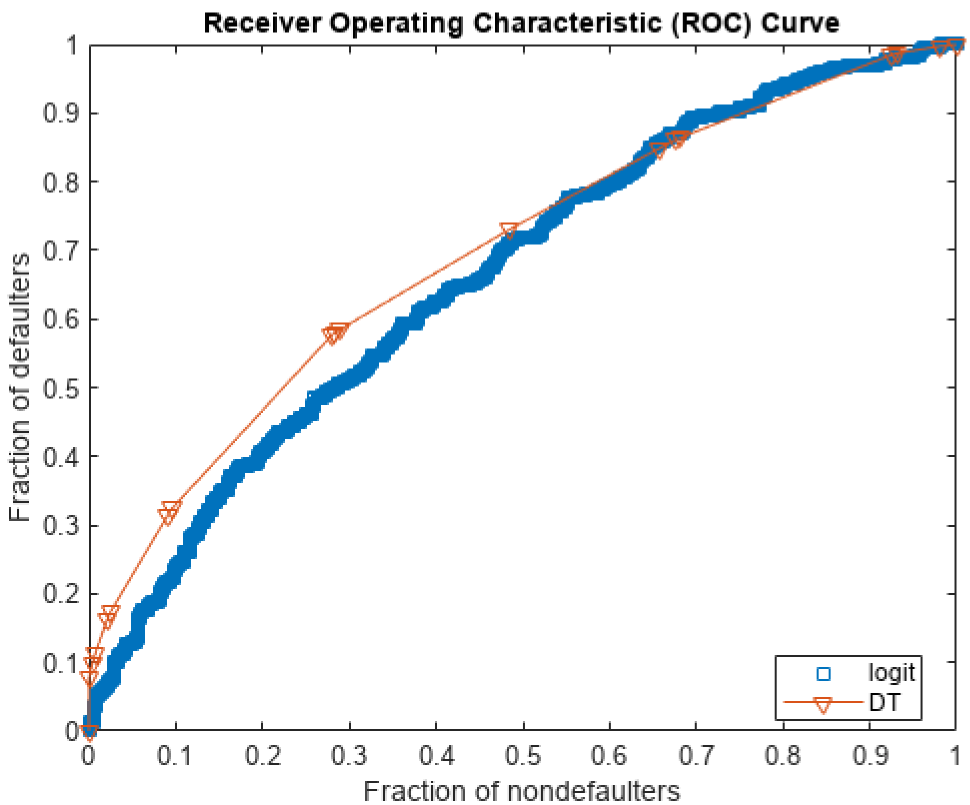
ROC Curve

The area under the receiver operating characteristic (AUROC) curve is a performance metric for classification problems. AUROC measures the degree of separability — that is, how much the model can distinguish between classes. In this example, the classes to distinguish are defaulters and nondefaulters. A high AUROC indicates good predictive capability.

The ROC curve is plotted with the true positive rate (also known as the sensitivity or recall) plotted against the false positive rate (also known as the fallout or specificity). When `AUROC = 0.7`, the model has a 70% chance of correctly distinguishing between the classes. When `AUROC = 0.5`, the model has no discrimination power.

This plot compares the ROC curves for both models using the same dataset.

```
figure
plot([0;tL.FalseAlarm],[0;tL.Sensitivity],'s')
hold on
plot(x,y,'-v')
xlabel('Fraction of nondefaulters')
ylabel('Fraction of defaulters')
legend({'logit','DT'},'Location','best')
title('Receiver Operating Characteristic (ROC) Curve')
```




```
tValidation = table(Measure,StatsL.Value(1:end-1),StatsDT.Value,'VariableNames',...
    {'Measure','logit','DT'});

disp(tValidation)
```

```
         Measure             logit       DT   
    ________________________    _______    _______

    {'Accuracy Ratio'      }    0.32515    0.38903
    {'Area Under ROC Curve'}    0.66258    0.69451
    {'KS Statistic'        }    0.23204    0.29666
```

As the AUROC values show, given the dataset and selected binning algorithm for the `creditscorecard` object, the decision tree model has better predictive power than the logistic regression model.

Summary

This example compares the logistic regression and decision tree scoring models using the `CreditCardData.mat` dataset. A workflow is presented to compute and compare PDs using decision trees. The decision tree model is validated and contrasted with the logistic regression model.

When reviewing the results, remember that these results depend on the choice of the dataset and the default binning algorithm (monotone adjacent pooling algorithm) in the logistic regression workflow.

- Whether a logistic regression or decision tree model is a better scoring model depends on the dataset and the choice of binning algorithm. Although the decision tree model in this example is a better scoring model, the logistic regression model produces higher accuracy ratio (`0.42`), AUROC (`0.71`), and KS statistic (`0.30`) values if the binning algorithm for the creditscorecard object is set as `'Split'` with `Gini` as the split criterion.
- The `validatemodel` function requires scaled scores to compute validation metrics and values. If you use a decision tree model, scaled scores are unavailable and you must perform the computations outside the `creditscorecard` object.
- To demonstrate the workflow, this example uses the same dataset for training the models and for testing. However, to validate a model, using a separate testing dataset is ideal.
- Scaling options for decision trees are unavailable. To use scaling, choose a model other than decision trees.

See Also

`creditscorecard` | `screenpredictors` | `autobinning` | `bininfo` | `predictorinfo` | `modifypredictor` | `modifybins` | `bindata` | `plotbins` | `fitmodel` | `displaypoints` | `formatpoints` | `score` | `setmodel` | `probdefault` | `validatemodel`

Related Examples

- “Common Binning Explorer Tasks” on page 3-4
- “Credit Scorecard Modeling with Missing Values”
- “Feature Screening with screenpredictors” on page 3-64
- “Troubleshooting Credit Scorecard Results”
- “Credit Rating by Bagging Decision Trees”
- “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36

More About

- “Overview of Binning Explorer” on page 3-2

- “About Credit Scorecards”
- “Credit Scorecard Modeling Workflow”
- Monotone Adjacent Pooling Algorithm (MAPA)
- “Credit Scorecard Modeling Using Observation Weights”

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

Explore Fairness Metrics for Credit Scoring Model

This example shows how to calculate and display fairness metrics for two sensitive attributes. You can use these metrics to test data and the model for fairness and then determine the thresholds to apply for your situation. You can also use the metrics to understand the biases in your model, the levels of disparity between groups, and how to assess the fairness of the model. This example uses the `fairnessMetrics` class in the Statistics and Machine Learning Toolbox™ to compute, display, and plot the various fairness metrics.

Fairness Metrics Calculations

Fairness metrics are a set of measures that enable you to detect the presence of bias in your data or model. Bias refers to the preference of one group over another group, implicitly or explicitly. When you detect bias in your data or model, you can decide to take action to mitigate the bias. Bias detection is a set of measures that enable you to see the presence of unfairness toward one group or another. Bias mitigation is a set of tools to reduce the amount of bias that occurs in the data or model for the current analysis.

A set of metrics exists for the data and a set of metrics also exists for the model. Group metrics measure information within the group, whereas bias metrics measure differences across groups. The example calculates two bias metrics (Statistical Parity Difference (SPD) and Disparate Impact (DI)) and a group metric (group count) at the data level. In this example, you calculate four bias metrics and 17 group metrics at the model level.

Bias metrics:

- Statistical Parity Difference (SPD) measures the difference that the majority and protected classes receive a favorable outcome. This measure must be equal to 0 to be fair.

  $$\text{SPD} = P\left(\widehat{Y} = 1 \mid A = \text{minority}\right) - P\left(\widehat{Y} = 1 \mid A = \text{majority}\right),$$
  $$\text{where } \widehat{Y} \text{ are the model predictions and } A \text{ is the group of the sensitive attribute}.$$

- Disparate Impact (DI) compares the proportion of individuals that receive a favorable outcome for two groups, a majority group and a minority group. This measure must be equal to 1 to be fair.

  $$\text{DI} = P\left(\widehat{Y} = 1 \mid A = \text{minority}\right) / P\left(\widehat{Y} = 1 \mid A = \text{majority}\right),$$
  $$\text{where } \widehat{Y} \text{ are the model predictions and } A \text{ is the group of the sensitive attribute}.$$

- Equal Opportunity Difference (EOD) measures the deviation from the equality of opportunity, which means that the same proportion of each population receives the favorable outcome. This measure must be equal to 0 to be fair.

  $$\text{EOD} = P\left(\widehat{Y} = 1 \mid A = \text{minority}, Y = 1\right) - P\left(\widehat{Y} = 1 \mid A = \text{majority}, Y = 1\right),$$
  $$\text{where } \widehat{Y} \text{ are the model predictions, } A \text{ is the group of the sensitive attribute,}$$
  $$\text{and } Y \text{ are the true labels}.$$

- Average Absolute Odds Difference (AAOD) measures bias by using the false positive rate and true positive rate. This measure must be equal to 0 to be fair.

$$\text{AAOD} = \frac{1}{2}\left[\left|FPR_{A\,=\,minority} - FPR_{A\,=\,majority}\right| + \left|TPR_{A\,=\,minority} - TPR_{A\,=\,majority}\right|\right],$$

where A is the group of the sensitive attribute .

Group metrics:

- True Positives (TP) is the total number of outcomes where the model correctly predicts the positive class.
- True Negatives (TN) is the total number of outcomes where the model correctly predicts the negative class.
- False Positives (FP) is the total number of outcomes where the model incorrectly predicts the positive class.
- False Negatives (FN) is the total number of outcomes where the model incorrectly predicts the negative class.
- True Positive Rate (TPR) is the sensitivity.

$$\text{TPR} = \frac{\text{TP}}{(\text{TP} + \text{FN})}$$

- True Negative Rate (TNR) is the specificity or selectivity.

$$\text{TNR} = \frac{\text{TN}}{(\text{TN} + \text{FP})}$$

- False Positive Rate (FPR) is the Type-I error.

$$\text{FPR} = \frac{\text{FP}}{(\text{FP} + \text{TN})}$$

- False Negative Rate (FNR) is the Type-II error.

$$\text{FNR} = \frac{\text{FN}}{(\text{FN} + \text{TP})}$$

- False Discovery Rate (FDR) is the ratio of the number of false positive results to the number of total positive test results.

$$\text{FDR} = \frac{\text{FP}}{(\text{FP} + \text{TP})}$$

- False Omission Rate (FOR) is the ratio of the number of individuals with a negative predicted value for which the true label is positive.

$$\text{FOR} = \frac{\text{FN}}{(\text{FN} + \text{TN})}$$

- Positive Predictive Value (PPV) is the ratio of the number of true positives to the number of true positives and false positives.

$$\text{PPV} = \frac{\text{TP}}{(\text{TP} + \text{FP})}$$

- Negative Predictive Value (NPV) is the ratio of the number of true negatives to the number of true positives and false positives.

$$\text{NPV} = \frac{\text{TN}}{(\text{TN} + \text{FN})}$$

- Rate of Positive Predictions (RPP) or Acceptance Rate is the ratio of the number of false and true positives to the total observations.

$$\text{RPP} = \frac{(\text{FP} + \text{TP})}{(\text{TN} + \text{TP} + \text{FN} + \text{FP})}$$

- Rate of Negative Predictions (RNP) is the ratio of the number of false and true negatives to the total observations.

$$\text{RNP} = \frac{(\text{FN} + \text{TN})}{(\text{TN} + \text{TP} + \text{FN} + \text{FP})}$$

- Accuracy (ACC) is the ratio of the number of true negatives and true positives to the total observations.

$$\text{ACC} = \frac{(\text{TN} + \text{TP})}{(\text{TN} + \text{TP} + \text{FN} + \text{FP})}$$

- Group Count is the number of individuals in the group.
- Group Size Ratio is the ratio of the number of individuals in that group to the total number of individuals.

The example focuses on bias detection in credit card data and explores bias metrics and group metrics based on the sensitive attributes of customer age (`CustAge`) and residential status (`ResStatus`). The data contains the residential status as a categorical variable and the customer age as a numeric variable. To create predictions and analyze the data for fairness, you group the customer age variable into bins.

Visualize Sensitive Attributes in Credit Card Data

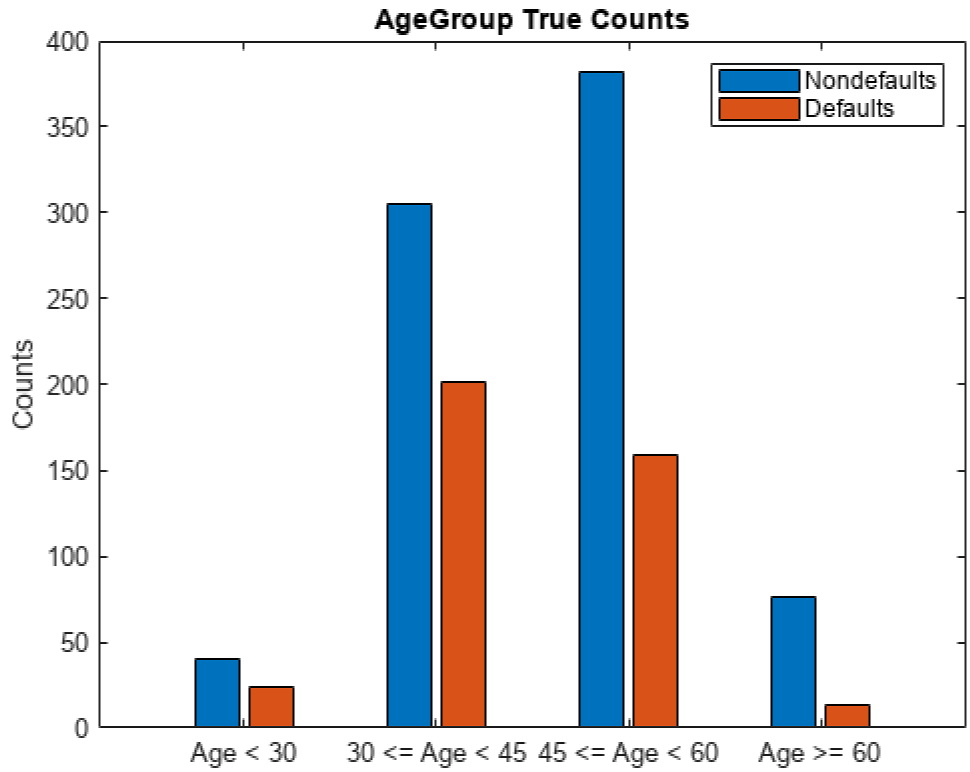
Load the credit card data set. Group the customer age into bins. Use the `discretize` function for a numeric variable to create groups that identify age groups of interest for comparison on fairness. Retrieve the counts for both sensitive attributes of customer age and residential status.

```
load CreditCardData.mat

AgeGroup = discretize(data.CustAge,[min(data.CustAge) 30 45 60 max(data.CustAge)], ...
    'categorical',{'Age < 30','30 <= Age < 45','45 <= Age < 60','Age >= 60'});
data = addvars(data,AgeGroup,'After','CustAge');

gs_data_ResStatus = groupsummary(data,{'ResStatus','status'});
gs_data_AgeGroup = groupsummary(data,{'AgeGroup','status'});
```

Plot the count of customers who have defaulted on their credit card payments and who have not defaulted by age.

```
Attribute = AgeGroup;
figure
bar(unique(data.(Attribute)), ...
    [eval("gs_data_"+Attribute+".GroupCount(1:2:end)"), ...
    eval("gs_data_"+Attribute+".GroupCount(2:2:end)")]');
title(Attribute +" True Counts");
```

```
ylabel('Counts')
legend({'Nondefaults','Defaults'})
```



Calculate Fairness Metrics for Data

Calculate fairness metrics for the residential status and customer age data. The `fairnessMetrics` class returns a `fairnessMetrics` object, whichis then passed into the `report` method to obtain a table with bias metrics and group metrics. Bias metrics take into account two classes (the majority and minority) at a time, while group metrics are within the individual group. In the data set, if you use residential status as the sensitive attribute, then the `Home Owner` group is the majority class because this class contains the largest number of individuals. Based on the SPD and DI metrics, the data set does not show a significant presence of bias for residential status. For the customer age data, the age group between 45 and 60 is the majority class because this class contains the largest number of individuals. Compared to the residential status, based on the SPD and DI metrics, the age group that is greater than 60 shows a slightly larger presence of bias.

```
dataMetricsObj = fairnessMetrics(data, 'status', 'SensitiveAttributeNames',{'ResStatus','AgeGrou
```

```
dataMetricsObj = 
  fairnessMetrics with properties:

    SensitiveAttributeNames: {'ResStatus'  'AgeGroup'}
             ReferenceGroup: {'Home Owner'  '45 <= Age < 60'}
               ResponseName: 'status'
              PositiveClass: 1
                BiasMetrics: [7x4 table]
               GroupMetrics: [7x4 table]
```

```
  Properties, Methods
```

```
dataMetricsTable = report(dataMetricsObj,'GroupMetrics','GroupCount')
```

```
dataMetricsTable=7×5 table
    SensitiveAttributeNames        Groups        StatisticalParityDifference    DisparateImpact
    _______________________    _______________    ___________________________    _______________

           ResStatus           Home Owner                        0                        1
           ResStatus           Tenant                     0.025752                   1.0789
           ResStatus           Other                     -0.038525                  0.88203
           AgeGroup            Age < 30                     0.0811                   1.2759
           AgeGroup            30 <= Age < 45              0.10333                   1.3516
           AgeGroup            45 <= Age < 60                    0                        1
           AgeGroup            Age >= 60                  -0.14783                    0.497
```

Create Credit Scorecard Model and Generate Predictions

Create a credit scorecard model using the `creditscorecard` function. Perform automatic binning of the predictors using the `autobinning` function. Fit a logistic regression model to the Weight of Evidence (WOE) data using the `fitmodel` function. Store the predictor names and corresponding coefficients in the credit scorecard model.

```
PredictorVars = setdiff(data.Properties.VariableNames, ...
    {'AgeGroup','CustID','status'});
sc = creditscorecard(data,'IDVar','CustID', ...
    'PredictorVars',PredictorVars);
sc = autobinning(sc);
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792
```

```
1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Display unscaled points for predictors retained in the model using the `displaypoints` function.

```
pointsinfo = displaypoints(sc)
```

```
pointsinfo=37×3 table
      Predictors              Bin              Points
    _______________    _________________    _________

    {'CustAge'    }    {'[-Inf,33)'    }     -0.15894
    {'CustAge'    }    {'[33,37)'      }     -0.14036
    {'CustAge'    }    {'[37,40)'      }    -0.060323
    {'CustAge'    }    {'[40,46)'      }     0.046408
    {'CustAge'    }    {'[46,48)'      }      0.21445
    {'CustAge'    }    {'[48,58)'      }      0.23039
    {'CustAge'    }    {'[58,Inf]'     }        0.479
    {'CustAge'    }    {'<missing>'    }          NaN
    {'ResStatus'  }    {'Tenant'       }    -0.031252
    {'ResStatus'  }    {'Home Owner'   }      0.12696
    {'ResStatus'  }    {'Other'        }      0.37641
    {'ResStatus'  }    {'<missing>'    }          NaN
    {'EmpStatus'  }    {'Unknown'      }    -0.076317
    {'EmpStatus'  }    {'Employed'     }      0.31449
    {'EmpStatus'  }    {'<missing>'    }          NaN
    {'CustIncome'}     {'[-Inf,29000)'}      -0.45716
      ⋮
```

For details about creating a more in depth credit scoring model, see the “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23.

Calculate the probability of default for the credit scorecard model using the `probdefault` function. Define the threshold for the probability of default as 0.35. Create an array of predictions where each value is greater than the threshold.

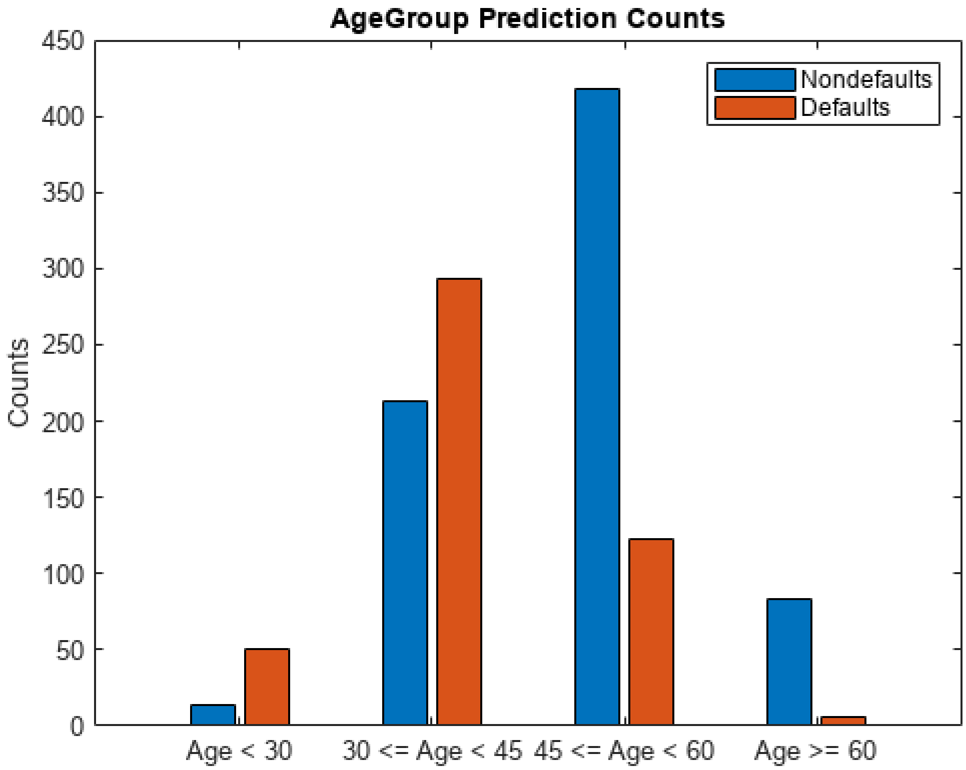
```
pd = probdefault(sc);
threshold = 0.35;
predictions = double(pd>threshold);
```

Add the resulting predictions to the `data` output table. To calculate bias metrics, you can set aside a set of validation data. Retrieve the counts for the residential status and customer age predictions. Plot the customer age predictions.

```
data = addvars(data,predictions,'After','status');
gs_predictions_ResStatus = groupsummary(data,{'ResStatus','predictions'}, ...
    'IncludeEmptyGroups',true);
gs_predictions_AgeGroup = groupsummary(data,{'AgeGroup','predictions'}, ...
    'IncludeEmptyGroups',true);
```

```
Attribute = AgeGroup ;
figure
bar(unique(data.(Attribute)), ...
```

```
    [eval("gs_predictions_"+Attribute+".GroupCount(1:2:end)"), ...
    eval("gs_predictions_"+Attribute+".GroupCount(2:2:end)")]');
title(Attribute +" Prediction Counts");
ylabel('Counts')
legend({'Nondefaults','Defaults'})
```



Calculate and Visualize Fairness Metrics for Credit Scorecard Model

Calculate model bias and group metrics for residential status and customer age. For the DI model metric, the commonly used range to assess fairness is between 0.8 and 1.25 [3 on page 3-109]. A value of less than 0.8 indicates the presence of bias. However, a value greater than 1.25 indicates that something is incorrect and additional investigation might be required. The model bias metrics in this example show a greater effect on fairness than the data bias metrics. After the model has been fitted, the negative SPD and EOD values mean that the `Other` group shows a slight presence of bias. In the group metrics, the FPR group metric of 39.7% is higher for tenants than home owners, which means that tenants are more likely to be falsely labeled as defaults. The FDR, FOR, PPV, and NPV group metrics show a very minimal presence of bias.

Looking at the model bias metrics SPD, DI, EOD, and AAOD for customer age, the 30 and under group has the greatest variance from the majority class and might require further investigation. Further, the age group over 60 shows the presence of bias based on the negative SPD and EOD values and the very low DI value. Also, based on the DI metrics, additional model bias mitigation might be required.

In the group metrics, the FPR group metric of 80% is much higher for the 30 and under group than the majority class, which means that those individuals whose age is 30 and under are more likely to be falsely labeled as defaults. The FDR group metric of 83.3% is much higher for the over 60 group than the majority class, which means that 83.3% of individuals whose age is over 60 and identified as defaults by the model are false positives. The `Accuracy` metric shows the highest accuracy for the over 60 group at 80.9%.

```
modelMetricsObj = fairnessMetrics(data, 'status', 'SensitiveAttributeNames',{'ResStatus','AgeGrou
```

```
modelMetricsObj = 
  fairnessMetrics with properties:

    SensitiveAttributeNames: {'ResStatus'  'AgeGroup'}
             ReferenceGroup: {'Home Owner'  '45 <= Age < 60'}
               ResponseName: 'status'
              PositiveClass: 1
                BiasMetrics: [7x7 table]
               GroupMetrics: [7x20 table]
                 ModelNames: 'Model1'


  Properties, Methods
```

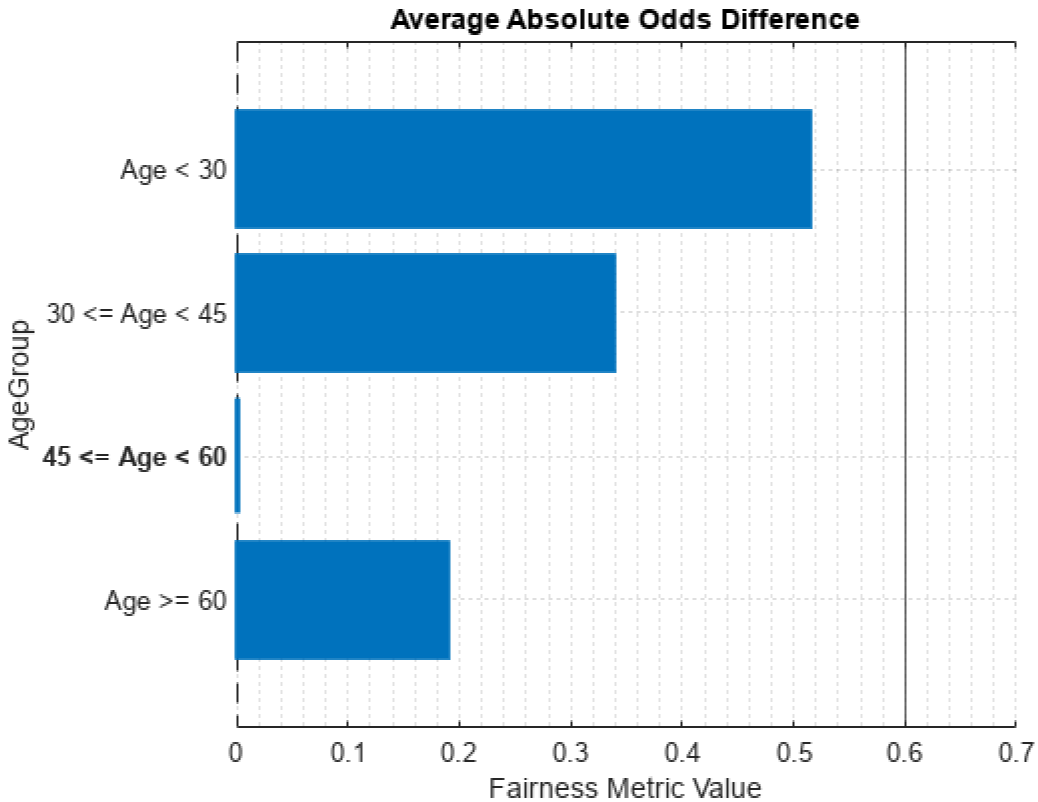
```
modelMetricsTable = report(modelMetricsObj,'GroupMetrics','all')
```

modelMetricsTable=*7×24 table*

ModelNames	SensitiveAttributeNames	Groups	StatisticalParityDifference	Di
Model1	ResStatus	Home Owner	0	
Model1	ResStatus	Tenant	0.10173	
Model1	ResStatus	Other	-0.11541	
Model1	AgeGroup	Age < 30	0.55389	
Model1	AgeGroup	30 <= Age < 45	0.35169	
Model1	AgeGroup	45 <= Age < 60	0	
Model1	AgeGroup	Age >= 60	-0.15994	

Choose the bias metric and sensitive attribute and plot it. This code selects `AAOD` and `AgeGroup` by default.

```
BiasMetric = AverageAbsoluteO... ;
SensitiveAttribute = AgeGroup ;
plot(modelMetricsObj, BiasMetric, "SensitiveAttributeNames", SensitiveAttribute);
```

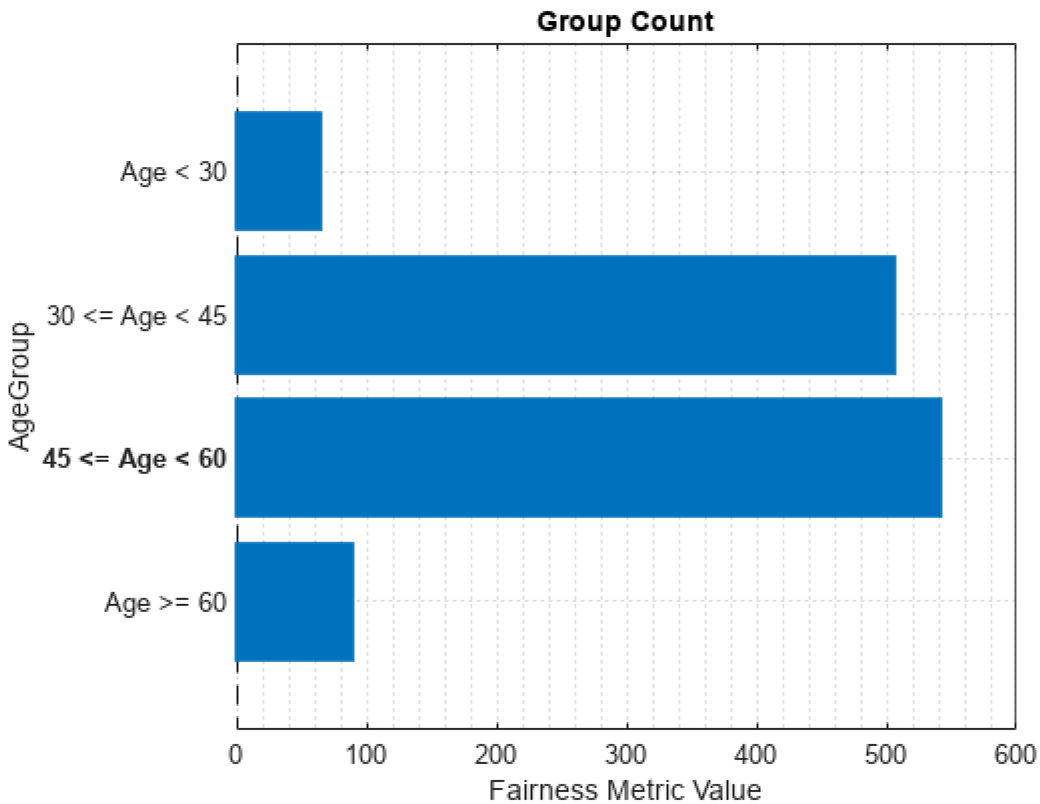


For the same sensitive attribute, choose the group metric and plot it. This code selects the group count by default. The resulting plots show the metric values for the selected sensitive attribute.

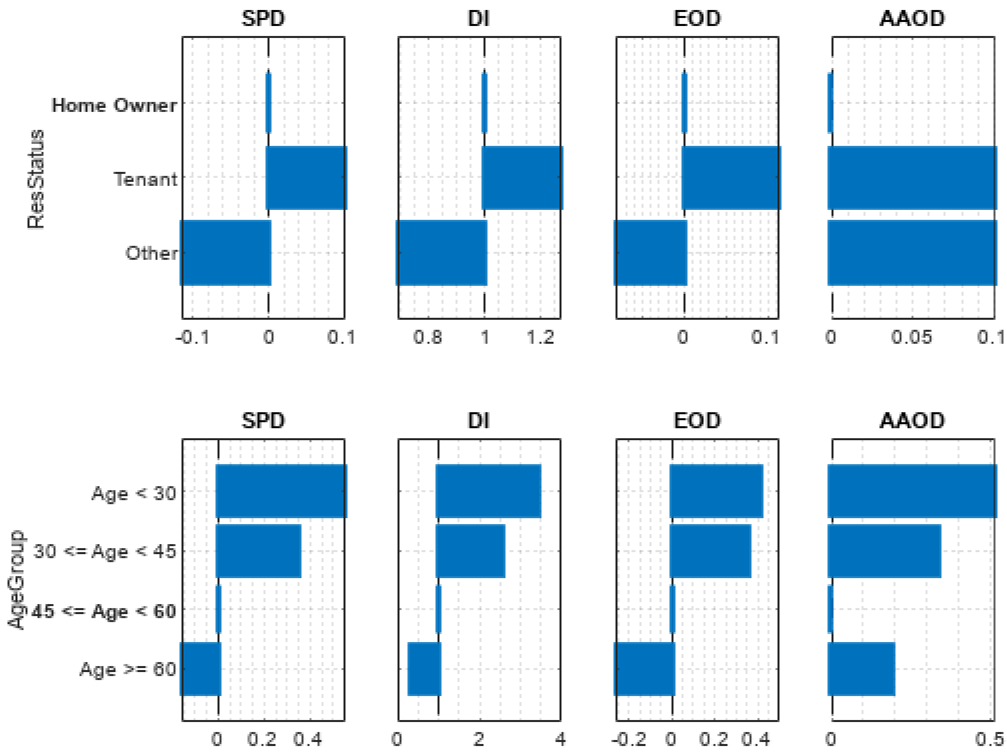
```
GroupMetric = GroupCount ;
plot(modelMetricsObj, GroupMetric, "SensitiveAttributeNames", SensitiveAttribute);
```

Plot the SPD, DI, EOD, and AAOD bias metrics for the two sensitive attributes.

```
MetricsShort = ["spd" "di" "eod" "aaod"];
tiledlayout(2,4)
```




```
for sa = string(modelMetricsObj.SensitiveAttributeNames)
   for m = MetricsShort
      nexttile
      h = plot(modelMetricsObj,m,'SensitiveAttributeNames',sa);
      title(h.Parent,upper(m));
      h.Parent.XLabel = [];
      if m~=MetricsShort(1)
         h.Parent.YTickLabel = '';
         h.Parent.YLabel = [];
      end
   end
end
```



Bias preserving metrics seek to keep the historic performance in the outputs of a target model with equivalent error rates for each group as shown in the training data. These metrics do not alter the status quo that exists in society. A fairness metric is classified as bias preserving when a perfect classifier exactly satisfies the metric. In contrast, bias transforming metrics require the explicit decision regarding which biases the system should exhibit. These metrics do not accept the status quo and acknowledge that protected groups start from different points that are not equal. The main difference between these two types of metrics is that most bias transforming metrics are satisfied by matching decision rates between groups, whereas bias preserving metrics require matching error rates instead. To assess the fairness of a decision-making system, use both bias preserving and transforming metrics to create the broadest possible view of the bias in the system.

Evaluating whether a metric is bias preserving is straightforward with a perfect classifier. In the absence of a perfect classifier, you can substitute the predictions with the classifier response and observe if the formula is trivially true. EOD and AAOD are bias preserving metrics because they have no variance; however, SPD and DI are bias transforming metrics as they show a variance from the majority classes.

```
biasMetrics_ResStatus1Obj = fairnessMetrics(data, 'status', 'SensitiveAttributeNames' ,'ResStatu
report(biasMetrics_ResStatus1Obj)
```

```
ans=3×7 table
    ModelNames     SensitiveAttributeNames      Groups       StatisticalParityDifference     Dispara
    __________     _______________________    __________     ___________________________     _______

      status              ResStatus           Home Owner                   0
      status              ResStatus           Tenant                0.025752                       1
```

```
    status            ResStatus           Other                  -0.038525               0.8
```

```
biasMetrics_AgeGroup1Obj = fairnessMetrics(data, 'status', 'SensitiveAttributeNames', 'AgeGroup',
report(biasMetrics_AgeGroup1Obj)
```

```
ans=4×7 table
    ModelNames    SensitiveAttributeNames       Groups        StatisticalParityDifference    Di
    __________    _______________________    _____________    ___________________________    __

      status             AgeGroup            Age < 30                    0.0811
      status             AgeGroup            30 <= Age < 45             0.10333
      status             AgeGroup            45 <= Age < 60                   0
      status             AgeGroup            Age >= 60                 -0.14783
```

References

1 Schmidt, Nicolas, Sue Shay, Steve Dickerson, Patrick Haggerty, Arjun R. Kannan, Kostas Kotsiopoulos, Raghu Kulkarni, Alexey Miroshnikov, Kate Prochaska, Melanie Wiwczaroski, Benjamin Cox, Patrick Hall, and Josephine Wang. *Machine Learning: Considerations for Fairly and Transparently Expanding Access to Credit.* Mountain View, CA: H2O.ai, Inc., July 2020.

2 Mehrabi, Ninareh, et al. “A Survey on Bias and Fairness in Machine Learning.” *ArXiv:1908.09635 [Cs]*, Sept. 2019. *arXiv.org*, https://arxiv.org/abs/1908.09635.

3 Saleiro, Pedro, et al. “Aequitas: A Bias and Fairness Audit Toolkit.” *ArXiv:1811.05577 [Cs]*, Apr. 2019. *arXiv.org*, https://arxiv.org/abs/1811.05577.

4 Wachter, Sandra, et al. *Bias Preservation in Machine Learning: The Legality of Fairness Metrics Under EU Non-Discrimination Law*. SSRN Scholarly Paper, ID 3792772, Social Science Research Network, 15 Jan. 2021. *papers.ssrn.com*, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3792772.

See Also

`creditscorecard` | `autobinning` | `fitmodel` | `displaypoints` | `probdefault`

Related Examples

- “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23
- “Bias Mitigation in Credit Scoring by Reweighting” on page 3-110
- “Bias Mitigation in Credit Scoring by Disparate Impact Removal” on page 3-119

Bias Mitigation in Credit Scoring by Reweighting

Bias mitigation is the process of removing bias from a data set or a model in order to make it fair. Bias mitigation usually follows a bias detection step, where a series of metrics are computed based on a data set or model predictions. Bias mitigation has three stages: *pre-processing*, *in-processing*, and *post-processing*. This example demonstates a pre-processing method to mitigate bias in a credit scoring workflow. The example uses bias detection and bias mitigation functionality from the Statistics and Machine Learning Toolbox™. For a detailed example on bias detection, see the following example: “Explore Fairness Metrics for Credit Scoring Model” on page 3-98.

The bias mitigation method in this example is *Reweighting* which essentially reweights observations within a data set to guarantee fairness between different subgroups within a sensitive attribute. As a result of reweighting, the Statistical Parity Difference (SPD) of all subgroups goes to `0` and the Disparate Impact metric becomes `1`. This example demonstrates how reweighting works in a credit scoring workflow.

Load Data

Load the `CreditCardData` data set and discretize the `'CustAge'` predictor.

```
load CreditCardData.mat

AgeGroup = discretize(data.CustAge,[min(data.CustAge) 30 45 60 max(data.CustAge)], ...
    'categorical',{'Age < 30','30 <= Age < 45','45 <= Age < 60','Age >= 60'});

data = addvars(data,AgeGroup,'After','CustAge');
head(data)
```

```
    CustID    CustAge       AgeGroup        TmAtAddress     ResStatus     EmpStatus    CustIncome
    ______    _______    ______________    ___________    __________    _________    __________

      1         53       45 <= Age < 60        62         Tenant        Unknown        50000
      2         61       Age >= 60             22         Home Owner    Employed       52000
      3         47       45 <= Age < 60        30         Tenant        Employed       37000
      4         50       45 <= Age < 60        75         Home Owner    Employed       53000
      5         68       Age >= 60             56         Home Owner    Employed       53000
      6         65       Age >= 60             13         Home Owner    Employed       48000
      7         34       30 <= Age < 45        32         Home Owner    Unknown        32000
      8         50       45 <= Age < 60        57         Other         Employed       51000
```

Split the data set into training and testing data. Use the training data to fit the model and the testing data to predict from the model.

```
rng('default');
c = cvpartition(size(data,1),'HoldOut',0.3);
data_Train = data(c.training(),:);
data_Test = data(c.test(),:);
```

Compute Fairness Metrics at Predictor and Model Level

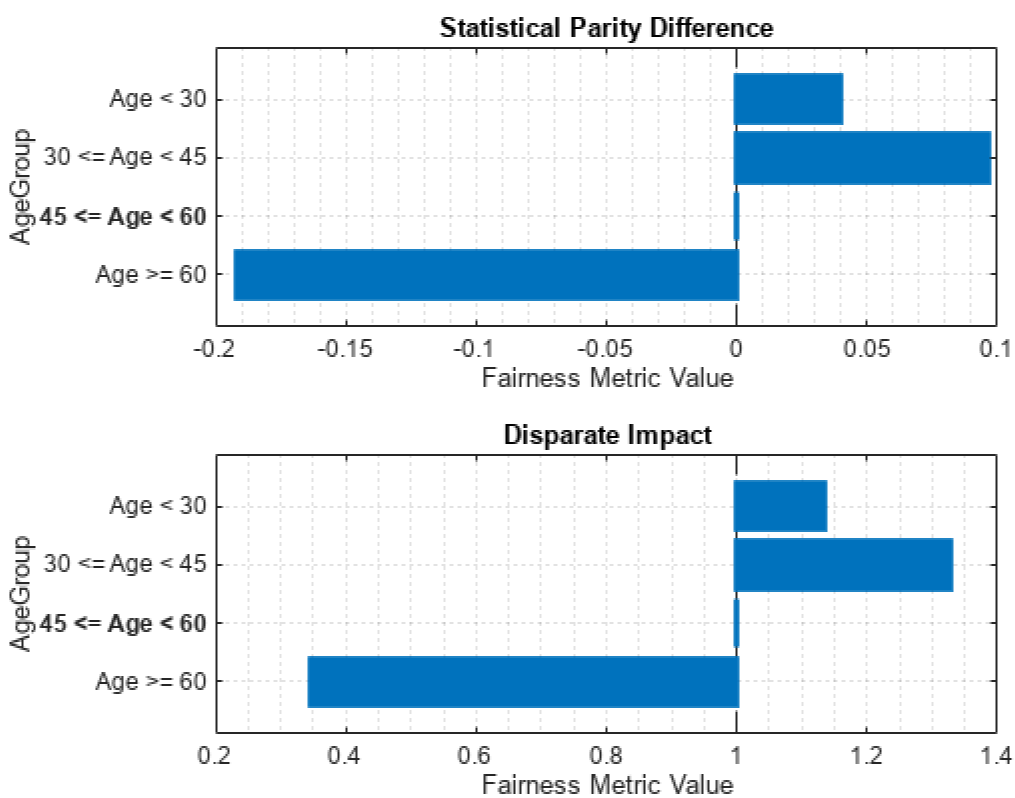
Compute the fairness metrics for the training data by creating a `fairnessMetrics` object and then generating a metrics report using `report`. Since you are only working with data and there is no fitted model, only two bias metrics are computed for `StatisticalParityDifference` and `DisparateImpact`. The two group metrics computed are `GroupCount` and `GroupSizeRatio`. The fairness metrics are computed for two sensitive attributes, Age (`'AgeGroup'`) and Residential Status (`'ResStatus'`).

```
trainingDataMetrics = fairnessMetrics(data_Train, 'status', 'SensitiveAttributeNames',{'AgeGroup
tdmReport = report(trainingDataMetrics)
```

tdmReport=*7×4 table*

SensitiveAttributeNames	Groups	StatisticalParityDifference	DisparateImpact
AgeGroup	Age < 30	0.039827	1.1357
AgeGroup	30 <= Age < 45	0.096324	1.3282
AgeGroup	45 <= Age < 60	0	1
AgeGroup	Age >= 60	-0.19181	0.34648
ResStatus	Home Owner	0	1
ResStatus	Tenant	0.01689	1.0529
ResStatus	Other	-0.02108	0.93404

```
figure
tiledlayout(2,1)
nexttile
plot(trainingDataMetrics,'spd')
nexttile
plot(trainingDataMetrics,'di')
```



Looking at the `DisparateImpact` bias metric for both `AgeGroup` and `ResStatus`, you can see that there is a much larger variance in the `AgeGroup` predictor as compared to the `ResStatus` predictor.

This suggests that users are treated more unfairly when it comes to their age as compared to their residential status. This example focuses on the `AgeGroup` predictor and attempts to reduce bias among its subgroups.

To begin, fit a credit scoring model and compute the model-level bias metrics. This provides a baseline for comparison.

Since `CustAge` and `AgeGroup` are essentially the same predictor and this is a sensitive attribute, you can exclude it from the model. Additionally, you can use `'status'` as the response variable and `'CustID'` as the `ID` variable.

```
PredictorVars = setdiff(data_Train.Properties.VariableNames, ...
    {'CustAge','AgeGroup','CustID','FairWeights','status'});
sc1 = creditscorecard(data_Train,'IDVar','CustID', ...
    'PredictorVars',PredictorVars,'ResponseVar','status');
sc1 = autobinning(sc1);
sc1 = fitmodel(sc1,'VariableSelection','fullmodel');
```

```
Generalized linear regression model:
    logit(status) ~ 1 + TmAtAddress + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AM
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat        pValue
                   ________    ________    ________    __________

    (Intercept)     0.73924    0.077237      9.5711     1.058e-21
    TmAtAddress      1.2577     0.99118      1.2689       0.20448
    ResStatus         1.755       1.295      1.3552       0.17535
    EmpStatus       0.88652     0.32232      2.7504     0.0059516
    CustIncome      0.95991     0.19645      4.8862    1.0281e-06
    TmWBank           1.132      0.3157      3.5856    0.00033637
    OtherCC         0.85227      2.1198     0.40204       0.68765
    AMBalance        1.0773     0.31969      3.3698    0.00075232
    UtilRate       -0.19784     0.59565    -0.33214       0.73978


840 observations, 831 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 66.5, p-value = 2.44e-11
```

```
pointsinfo1 = displaypoints(sc1)
```

```
pointsinfo1=38×3 table
      Predictors              Bin              Points
    _______________    _________________    _________

    {'TmAtAddress'}    {'[-Inf,9)'     }     -0.17538
    {'TmAtAddress'}    {'[9,16)'       }      0.05434
    {'TmAtAddress'}    {'[16,23)'      }     0.096897
    {'TmAtAddress'}    {'[23,Inf]'     }      0.13984
    {'TmAtAddress'}    {'<missing>'    }          NaN
    {'ResStatus'  }    {'Tenant'       }    -0.017688
    {'ResStatus'  }    {'Home Owner'   }      0.11681
    {'ResStatus'  }    {'Other'        }      0.29011
    {'ResStatus'  }    {'<missing>'    }          NaN
    {'EmpStatus'  }    {'Unknown'      }    -0.097582
    {'EmpStatus'  }    {'Employed'     }      0.33162
```

```
    {'EmpStatus'  }    {'<missing>'    }          NaN
    {'CustIncome' }    {'[-Inf,30000)' }     -0.61962
    {'CustIncome' }    {'[30000,36000)'}     -0.10695
    {'CustIncome' }    {'[36000,40000)'}    0.0010845
    {'CustIncome' }    {'[40000,42000)'}     0.065532
      ⋮
```

```
pd1 = probdefault(sc1,data_Test);
```

Set the `threshold` value that controls the allocation of "goods" and "bads."

```
threshold = 0.35 ;
predictions1 = double(pd1>threshold);
```

Create a `fairnessMetrics` object to compute fairness metrics at the model level and then generate a metrics report using `report`.

```
modelMetrics1 = fairnessMetrics(data_Test, 'status', 'Predictions', predictions1, 'SensitiveAttr
mmReport1 = report(modelMetrics1)
```

mmReport1=*4×7 table*

ModelNames	SensitiveAttributeNames	Groups	StatisticalParityDifference	Di
Model1	AgeGroup	Age < 30	0.54312	
Model1	AgeGroup	30 <= Age < 45	0.19922	
Model1	AgeGroup	45 <= Age < 60	0	
Model1	AgeGroup	Age >= 60	-0.15385	

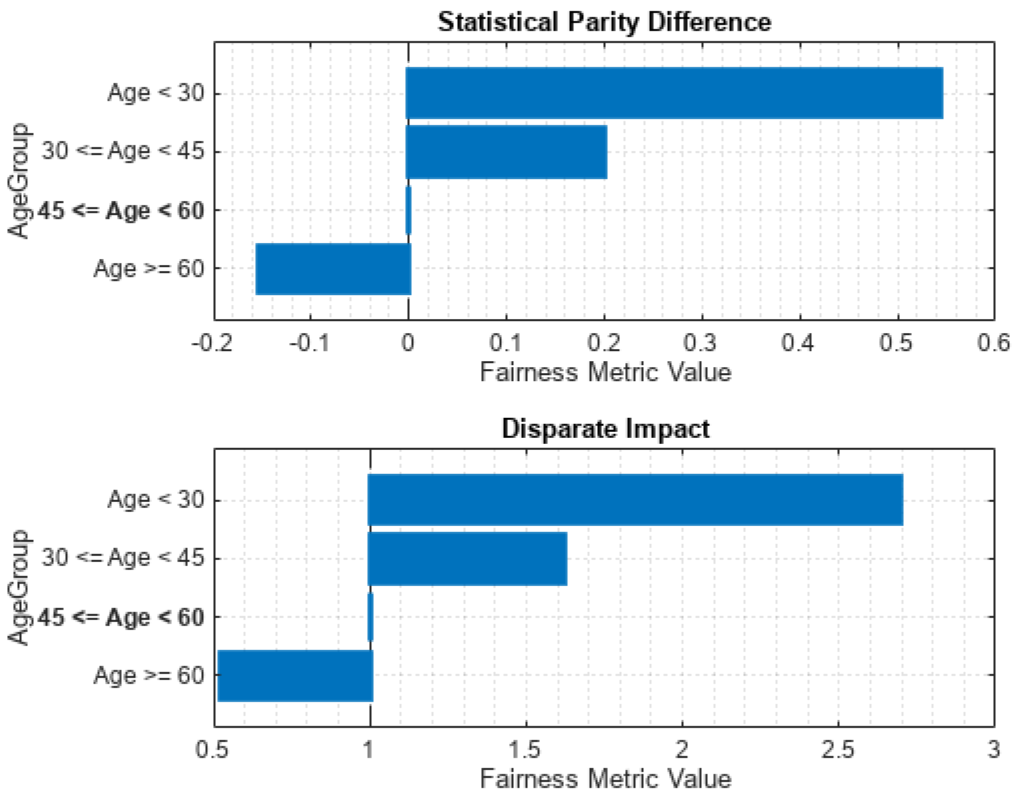
Measure accuracy of model using `validatemodel`.

```
validatemodel(sc1)
```

ans=*4×2 table*

Measure	Value
{'Accuracy Ratio' }	0.33751
{'Area under ROC curve'}	0.66876
{'KS statistic' }	0.26418
{'KS score' }	1.0403

```
figure
tiledlayout(2,1)
nexttile
plot(modelMetrics1,'spd')
nexttile
plot(modelMetrics1,'di')
```



Reweight Data at Predictor and Model Level

Use `fairnessWeights` to reweight the training data to remove bias for the sensitive attribute `'AgeGroup'`.

```
fairWeights = fairnessWeights(data_Train, 'AgeGroup', 'status');
data_Train.FairWeights = fairWeights;
head(data_Train)
```

CustID	CustAge	AgeGroup	TmAtAddress	ResStatus	EmpStatus	CustIncome
1	53	45 <= Age < 60	62	Tenant	Unknown	50000
2	61	Age >= 60	22	Home Owner	Employed	52000
3	47	45 <= Age < 60	30	Tenant	Employed	37000
4	50	45 <= Age < 60	75	Home Owner	Employed	53000
7	34	30 <= Age < 45	32	Home Owner	Unknown	32000
8	50	45 <= Age < 60	57	Other	Employed	51000
9	50	45 <= Age < 60	10	Tenant	Unknown	52000
10	49	45 <= Age < 60	30	Home Owner	Unknown	53000

Use `fairnessMetrics` to compute fairness metrics for the training data after reweighting and use `report` to generate a fairness metrics report..

```
trainingDataMetrics_AfterReweighting = fairnessMetrics(data_Train, 'status', 'SensitiveAttributeN
tdmrReport = report(trainingDataMetrics_AfterReweighting)
```

```
tdmrReport=4×4 table
    SensitiveAttributeNames        Groups        StatisticalParityDifference    DisparateImpact
    _______________________    ______________    ___________________________    _______________

           AgeGroup            Age < 30                  -2.9976e-15                   1
           AgeGroup            30 <= Age < 45            -5.5511e-16                   1
           AgeGroup            45 <= Age < 60                      0                   1
           AgeGroup            Age >= 60                 -2.9421e-15                   1
```

By applying the reweighting algorithm to the `AgeGroup` predictor, you can completely remove the disparate impact for `AgeGroup`. Then use this debiased data to fit a model to produce predictions with an overall reduced disparate impact at the model level.

Use `creditscorecard` to fit a new credit scoring model with the new fair weights and compute model-level bias metrics.

```
sc2 = creditscorecard(data_Train,'IDVar','CustID', ...
     'PredictorVars',PredictorVars,'WeightsVar','FairWeights','ResponseVar','status');
sc2 = autobinning(sc2);
sc2 = fitmodel(sc2,'VariableSelection','fullmodel');
```

```
Generalized linear regression model:
    logit(status) ~ 1 + TmAtAddress + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AM
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat        pValue  
                   ________    ________    ________    __________

    (Intercept)     0.74055    0.076222      9.7158    2.5817e-22
    TmAtAddress      1.3416      0.9108       1.473       0.14075
    ResStatus        2.0467      1.7669      1.1584       0.24672
    EmpStatus       0.91879     0.32197      2.8536     0.0043222
    CustIncome      0.91038     0.33216      2.7407       0.00613
    TmWBank          1.1067     0.30826      3.5901     0.0003305
    OtherCC         0.42264      3.5078     0.12049        0.9041
    AMBalance        1.1347      0.3447      3.2919    0.00099504
    UtilRate       -0.39861     0.77284    -0.51577       0.60601


840 observations, 831 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 46.6, p-value = 1.85e-07
```

```
pointsinfo2 = displaypoints(sc2)
```

```
pointsinfo2=34×3 table
      Predictors              Bin             Points 
    _______________    _________________    ________

    {'TmAtAddress'}    {'[-Inf,9)'     }    -0.21328
    {'TmAtAddress'}    {'[9,23)'       }     0.07168
    {'TmAtAddress'}    {'[23,Inf]'     }     0.14763
    {'TmAtAddress'}    {'<missing>'    }         NaN
    {'ResStatus'  }    {'Tenant'       }    0.016048
    {'ResStatus'  }    {'Home Owner'   }    0.091092
    {'ResStatus'  }    {'Other'        }     0.28326
```

```
    {'ResStatus'  }    {'<missing>'    }         NaN
    {'EmpStatus'  }    {'Unknown'      }    -0.10352
    {'EmpStatus'  }    {'Employed'     }     0.33653
    {'EmpStatus'  }    {'<missing>'    }         NaN
    {'CustIncome' }    {'[-Inf,30000)' }    -0.37618
    {'CustIncome' }    {'[30000,40000)'}    0.047483
    {'CustIncome' }    {'[40000,42000)'}     0.10244
    {'CustIncome' }    {'[42000,47000)'}     0.14652
    {'CustIncome' }    {'[47000,Inf]'  }     0.40015
      ⋮
```

```
pd2 = probdefault(sc2,data_Test);
predictions2 = double(pd2>threshold);
```

Use `fairnessMetrics` to compute fairness metrics at the model level and `report` to generate a fairness metrics report.

```
modelMetrics2 = fairnessMetrics(data_Test, 'status', 'Predictions', predictions2, 'SensitiveAttri
mmReport2 = report(modelMetrics2)
```

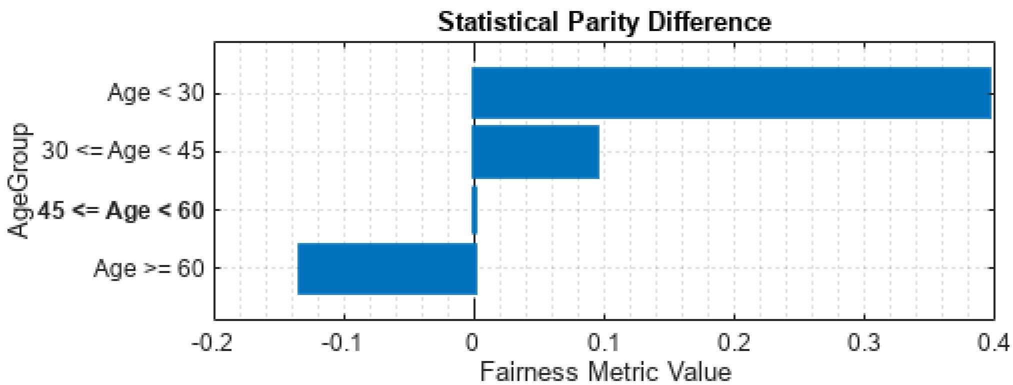
```
mmReport2=4×7 table
    ModelNames    SensitiveAttributeNames       Groups        StatisticalParityDifference    Di
    __________    _______________________    ______________    ___________________________    __

      Model1             AgeGroup            Age < 30                    0.39394
      Model1             AgeGroup            30 <= Age < 45             0.094298
      Model1             AgeGroup            45 <= Age < 60                    0
      Model1             AgeGroup            Age >= 60                  -0.13333
```

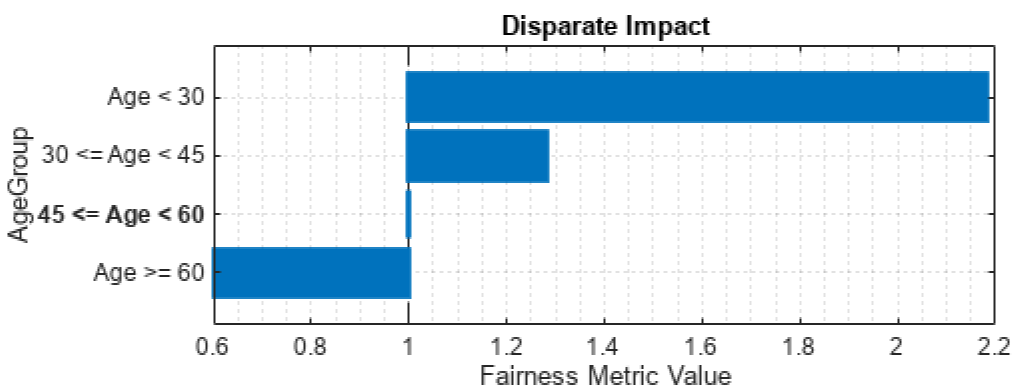
Measure accuracy of model using `validatemodel`.

```
validatemodel(sc2)
```

```
ans=4×2 table
            Measure              Value
    ________________________    _______

    {'Accuracy Ratio'      }    0.27735
    {'Area under ROC curve'}    0.63868
    {'KS statistic'        }    0.22702
    {'KS score'            }    0.90741
```

```
figure
tiledlayout(2,1)
nexttile
plot(modelMetrics2,'spd')
nexttile
plot(modelMetrics2,'di')
```





The process of reweighting removed all the bias from the training data. When you use the new data to fit a model, the overall bias in the model is reduced when compared to a model trained with biased data. As a consequence of this reduction in bias, there is a drop in model accuracy. You can choose to make tradeoff to improve fairness.

References

[1] Nielsen, Aileen. "Chapter 4. Fairness PreProcessing." *Practical Fairness*. O'Reilly Media, Inc., Dec. 2020.

[2] Mehrabi, Ninareh, et al. “A Survey on Bias and Fairness in Machine Learning.” *ArXiv:1908.09635 [Cs]*, Sept. 2019. *arXiv.org*, https://arxiv.org/abs/1908.09635.

[3] Wachter, Sandra, et al. *Bias Preservation in Machine Learning: The Legality of Fairness Metrics Under EU Non-Discrimination Law*. SSRN Scholarly Paper, ID 3792772, Social Science Research Network, 15 Jan. 2021. *papers.ssrn.com*, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3792772.

See Also
`creditscorecard` | `autobinning` | `fitmodel` | `displaypoints` | `probdefault`

Related Examples
- “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23

- “Explore Fairness Metrics for Credit Scoring Model” on page 3-98
- “Bias Mitigation in Credit Scoring by Disparate Impact Removal” on page 3-119

Bias Mitigation in Credit Scoring by Disparate Impact Removal

Disparate impact removal is a pre-processing technique in bias mitigation. Using this technique, you modify the original credit score data to increase group fairness, while still preserving rank ordering within groups. Using a disparate impact removal technique reduces the bias introduced by the credit scoring model more than if you use the original data to train the credit scoring model. You perform the disparate impact removal technique using the `disparateImpactRemover` class from the Statistics and Machine Learning Toolbox™. This class returns a `remover` object along with a table containing the new predictor values. However, you need to use the `transform` method with the `remover` object on the test data before you can predict using the fitted credit scoring model.

The disparate impact removal technique works only with the distribution of data within a numeric predictor for each subgroup of a sensitive attribute. The `disparateImpactRemover` class has no knowledge of, or relation to, the response variable. In this example, you treat all the numeric predictors, time at address (`TmAtAddress`), customer income (`CustIncome`), time with Bank (`TmWBank`), average monthly balance (`AMBalance`), and utilization rate (`UtilRate`), with respect to the sensitive attribute, customer age (`AgeGroup`).

Original Credit Scoring Model

This example uses a credit scoring workflow. Load the `CreditCardData.mat` and use the `'data'` data set.

```
load CreditCardData.mat

AgeGroup = discretize(data.CustAge,[min(data.CustAge) 30 45 60 max(data.CustAge)], ...
    'categorical',{'Age < 30','30 <= Age < 45','45 <= Age < 60','Age >= 60'});
data = addvars(data,AgeGroup,'After','CustAge');
head(data)
```

```
    CustID    CustAge       AgeGroup        TmAtAddress     ResStatus     EmpStatus    CustIncome
    ______    _______    ______________    ___________    __________    _________    __________

      1         53       45 <= Age < 60        62         Tenant        Unknown        50000
      2         61       Age >= 60             22         Home Owner    Employed       52000
      3         47       45 <= Age < 60        30         Tenant        Employed       37000
      4         50       45 <= Age < 60        75         Home Owner    Employed       53000
      5         68       Age >= 60             56         Home Owner    Employed       53000
      6         65       Age >= 60             13         Home Owner    Employed       48000
      7         34       30 <= Age < 45        32         Home Owner    Unknown        32000
      8         50       45 <= Age < 60        57         Other         Employed       51000
```

```
rng('default')
```

Split the data set into training and testing data.

```
c = cvpartition(size(data,1),'HoldOut',0.3);
data_Train = data(c.training(),:);
data_Test = data(c.test(),:);
head(data_Train)
```

```
    CustID    CustAge       AgeGroup        TmAtAddress     ResStatus     EmpStatus    CustIncome
    ______    _______    ______________    ___________    __________    _________    __________

       1        53       45 <= Age < 60        62         Tenant        Unknown        50000
```

```
         2         61        Age >= 60             22          Home Owner    Employed       52000
         3         47        45 <= Age < 60        30          Tenant        Employed       37000
         4         50        45 <= Age < 60        75          Home Owner    Employed       53000
         7         34        30 <= Age < 45        32          Home Owner    Unknown        32000
         8         50        45 <= Age < 60        57          Other         Employed       51000
         9         50        45 <= Age < 60        10          Tenant        Unknown        52000
        10         49        45 <= Age < 60        30          Home Owner    Unknown        53000
```

Use `creditscorecard` to create a `creditscorecard` object and use `fitmodel` to fit a credit scoring model with the the training data (`data_Train`).

```
PredictorVars = setdiff(data_Train.Properties.VariableNames, ...
    {'CustAge','AgeGroup','CustID','status'});
sc1 = creditscorecard(data_Train,'IDVar','CustID', ...
    'PredictorVars',PredictorVars,'ResponseVar','status');
sc1 = autobinning(sc1);
sc1 = fitmodel(sc1,'VariableSelection','fullmodel');
```

```
Generalized linear regression model:
    logit(status) ~ 1 + TmAtAddress + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AM
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE         tStat        pValue   
                   ________    ________    ________    __________

    (Intercept)     0.73924    0.077237      9.5711     1.058e-21
    TmAtAddress      1.2577     0.99118      1.2689       0.20448
    ResStatus         1.755       1.295      1.3552       0.17535
    EmpStatus       0.88652     0.32232      2.7504     0.0059516
    CustIncome      0.95991     0.19645      4.8862    1.0281e-06
    TmWBank           1.132      0.3157      3.5856    0.00033637
    OtherCC         0.85227      2.1198     0.40204       0.68765
    AMBalance        1.0773     0.31969      3.3698    0.00075232
    UtilRate       -0.19784     0.59565    -0.33214       0.73978


840 observations, 831 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 66.5, p-value = 2.44e-11
```

Use `displaypoints` to compute the points per predictor per bin for the `creditscorecard` model (`sc1`).

```
pointsinfo1 = displaypoints(sc1)
```

```
pointsinfo1=38×3 table
      Predictors              Bin              Points  
    _______________    _________________    _________

    {'TmAtAddress'}    {'[-Inf,9)'     }     -0.17538
    {'TmAtAddress'}    {'[9,16)'       }      0.05434
    {'TmAtAddress'}    {'[16,23)'      }     0.096897
    {'TmAtAddress'}    {'[23,Inf]'     }      0.13984
    {'TmAtAddress'}    {'<missing>'    }          NaN
    {'ResStatus'  }    {'Tenant'       }    -0.017688
    {'ResStatus'  }    {'Home Owner'   }      0.11681
    {'ResStatus'  }    {'Other'        }      0.29011
```

```
    {'ResStatus'  }    {'<missing>'    }          NaN
    {'EmpStatus'  }    {'Unknown'      }    -0.097582
    {'EmpStatus'  }    {'Employed'     }      0.33162
    {'EmpStatus'  }    {'<missing>'    }          NaN
    {'CustIncome' }    {'[-Inf,30000)' }     -0.61962
    {'CustIncome' }    {'[30000,36000)'}     -0.10695
    {'CustIncome' }    {'[36000,40000)'}    0.0010845
    {'CustIncome' }    {'[40000,42000)'}     0.065532
      ⋮
```

Use `probdefault` to determine the likelihood of default for the `data_Test` data set and the `creditscorecard` model (`sc1`).

```
pd1 = probdefault(sc1,data_Test);
threshold = 0.35;
predictions1 = double(pd1>threshold);
```

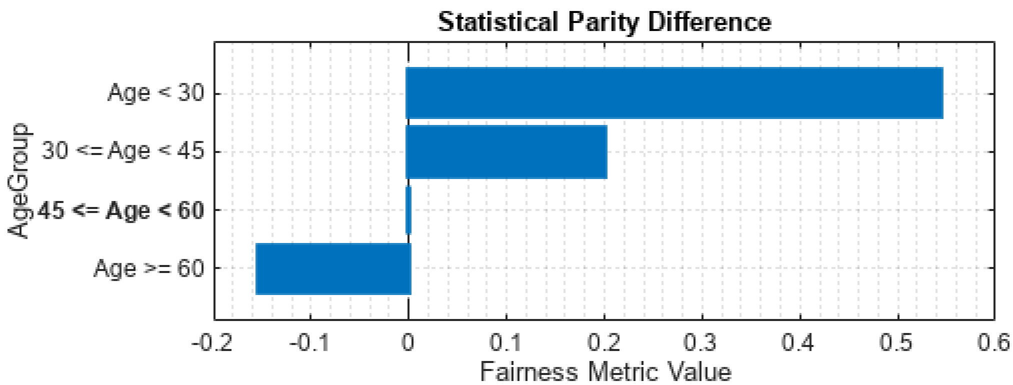
Use `fairnessMetrics` to compute fairness metrics at the model level as a baseline. Use `report` to generate the fairness metrics report.

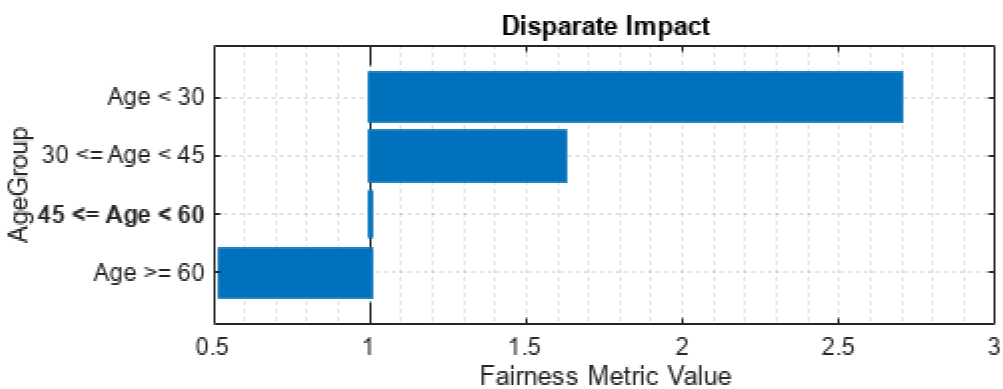
```
modelMetrics1 = fairnessMetrics(data_Test,'status','Predictions',predictions1,'SensitiveAttribute
mmReport1 = report(modelMetrics1,'GroupMetrics','GroupCount')
```

```
mmReport1=4×8 table
    ModelNames    SensitiveAttributeNames        Groups        StatisticalParityDifference    Di
    __________    _______________________    ______________    ___________________________    __

      Model1             AgeGroup            Age < 30                    0.54312
      Model1             AgeGroup            30 <= Age < 45              0.19922
      Model1             AgeGroup            45 <= Age < 60                    0
      Model1             AgeGroup            Age >= 60                  -0.15385
```

Use `plot` to visualize the statistical parity difference (`'spd'`) and disparate impact (`'di'`) bias metrics.

```
figure
tiledlayout(2,1)
nexttile
plot(modelMetrics1,'spd')
nexttile
plot(modelMetrics1,'di')
```





Bias Mitigation by Disparate Impact Removal

For each of the five continuous predictors, `'TmAtAddress'`, `'CustIncome'`, `'TmWBank'`, `'AMBalance'`, and `'UtilRate'` plot the original distributions of data within each age group.

Choose a numeric predictor to plot.

```
predictor = CustIncome ;
[f1, xi1] = ksdensity(data_Train.(predictor)(data_Train.AgeGroup=='Age < 30'));
[f2, xi2] = ksdensity(data_Train.(predictor)(data_Train.AgeGroup=='30 <= Age < 45'));
[f3, xi3] = ksdensity(data_Train.(predictor)(data_Train.AgeGroup=='45 <= Age < 60'));
[f4, xi4] = ksdensity(data_Train.(predictor)(data_Train.AgeGroup=='Age >= 60'));
```

Create a `disparateImpactRemover` object and return the `newTrainTbl` table with the new predictor values.

```
[remover, newTrainTbl] = disparateImpactRemover(data_Train, 'AgeGroup' , 'PredictorNames', {'TmA
```

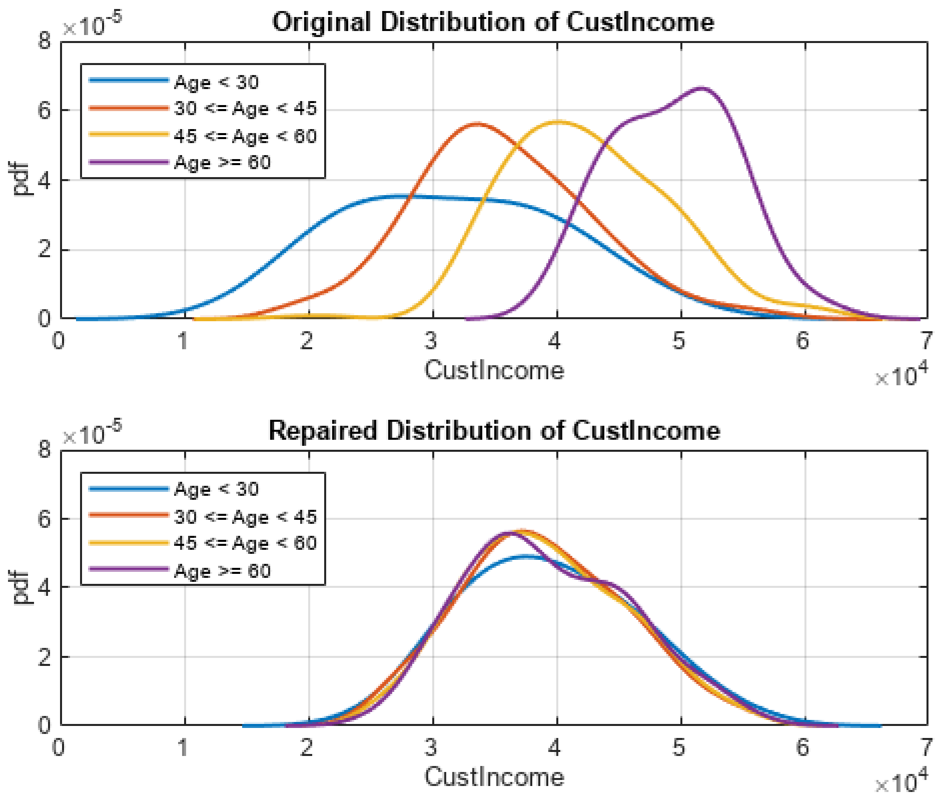
```
remover = 
  disparateImpactRemover with properties:

        RepairFraction: 1
        PredictorNames: {'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'UtilRate'}
    SensitiveAttribute: 'AgeGroup'


newTrainTbl=840×12 table
    CustID    CustAge       AgeGroup        TmAtAddress    ResStatus     EmpStatus    CustIncome
```

```
    ______      _______      ______________      ___________      __________      _________      __________

       1          53         45 <= Age < 60        58.599         Tenant          Unknown          47000
       2          61         Age >= 60                 24         Home Owner      Employed         41500
       3          47         45 <= Age < 60          30.5         Tenant          Employed         33500
       4          50         45 <= Age < 60        68.622         Home Owner      Employed         49401
       7          34         30 <= Age < 45          30.5         Home Owner      Unknown          35500
       8          50         45 <= Age < 60         53.39         Other           Employed         47000
       9          50         45 <= Age < 60             9         Tenant          Unknown          49401
      10          49         45 <= Age < 60          30.5         Home Owner      Unknown          49401
      11          52         45 <= Age < 60            24         Tenant          Unknown          30500
      12          48         45 <= Age < 60        77.291         Other           Unknown          40500
      14          44         30 <= Age < 45        68.622         Other           Unknown          44500
      17          39         30 <= Age < 45             9         Tenant          Employed         37500
      20          52         45 <= Age < 60        51.442         Other           Unknown          38500
      21          37         30 <= Age < 45        10.343         Tenant          Unknown          36500
      22          51         45 <= Age < 60        12.087         Home Owner      Employed         31500
      24          43         30 <= Age < 45            40         Tenant          Employed         33500
      ⋮
```

```
[nf1, nxi1] = ksdensity(newTrainTbl.(predictor)(newTrainTbl.AgeGroup=='Age < 30'));
[nf2, nxi2] = ksdensity(newTrainTbl.(predictor)(newTrainTbl.AgeGroup=='30 <= Age < 45'));
[nf3, nxi3] = ksdensity(newTrainTbl.(predictor)(newTrainTbl.AgeGroup=='45 <= Age < 60'));
[nf4, nxi4] = ksdensity(newTrainTbl.(predictor)(newTrainTbl.AgeGroup=='Age >= 60'));
```

Plot the original and the repaired distributions.

```
figure;
tiledlayout(2,1)
ax1 = nexttile;
plot(xi1, f1, 'LineWidth', 1.5)
hold on
plot(xi2, f2, 'LineWidth', 1.5)
plot(xi3, f3, 'LineWidth', 1.5)
plot(xi4, f4, 'LineWidth', 1.5)
legend(["Age < 30"; "30 <= Age < 45"; "45 <= Age < 60"; "Age >= 60"],'Location','northwest')
ax1.Title.String = "Original Distribution of " + predictor;
xlabel(predictor)
ylabel('pdf')
grid on
ax2 = nexttile;
plot(nxi1, nf1, 'LineWidth', 1.5)
hold on
plot(nxi2, nf2, 'LineWidth', 1.5)
plot(nxi3, nf3, 'LineWidth', 1.5)
plot(nxi4, nf4, 'LineWidth', 1.5)
legend(["Age < 30"; "30 <= Age < 45"; "45 <= Age < 60"; "Age >= 60"],'Location','northwest')
ax2.Title.String = "Repaired Distribution of " + predictor;
xlabel(predictor)
ylabel('pdf')
grid on
linkaxes([ax1, ax2], 'xy')
```



This plot demonstrates that the initial distributions of `CustIncome` of each group within the `AgeGroup` predictor are different. Younger people seem to have a lower income on average and more variation than older people. This difference introduces bias, which the fitted model then reflects. The `disparateImpactRemover` function modifies the data so that the distributions of all the subgroups are more similar. You see this distribution in the second subplot **Repaired Distribution of CustIncome**. Using this new data, you can fit a logistic regression model and then measure the model-level metrics and compare these with the model-level metrics from the original `creditscorecard` model (`sc1`).

New Credit Scoring Model

Use `creditscorecard` to create a `creditscorecard` object and use `fitmodel` to fit a credit scoring model with the the new data (`newTrainTbl`). Then, you can compute model-level bias metrics using `fairnessMetrics`.

```
sc2 = creditscorecard(newTrainTbl,'IDVar','CustID', ...
      'PredictorVars',PredictorVars,'ResponseVar','status');
sc2 = autobinning(sc2);
sc2 = fitmodel(sc2,'VariableSelection','fullmodel');
```

```
Generalized linear regression model:
    logit(status) ~ 1 + TmAtAddress + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AM
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE         tStat       pValue
                   ________    ________    ________    __________
```

```
    (Intercept)      0.74041     0.07641     9.6899     3.327e-22
    TmAtAddress       1.1658     0.87564     1.3313       0.18308
    ResStatus         1.8719      1.2848     1.4569       0.14513
    EmpStatus        0.88699     0.31991     2.7727       0.00556
    CustIncome       0.98269     0.28725      3.421    0.00062396
    TmWBank           1.1392     0.30677     3.7135    0.00020442
    OtherCC          0.55005      2.0969    0.26231       0.79308
    AMBalance         1.0478      0.3607     2.9049     0.0036734
    UtilRate       -0.071972     0.58704    -0.1226       0.90242


840 observations, 831 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 50.4, p-value = 3.36e-08
```

Use `displaypoints` to compute the points per predictor per bin for the `creditscorecard` model (`sc2`).

```
pointsinfo2 = displaypoints(sc2)
```

```
pointsinfo2=35×3 table
      Predictors              Bin             Points
    _______________    _________________    _________

    {'TmAtAddress'}    {'[-Inf,9)'     }     -0.11003
    {'TmAtAddress'}    {'[9,15.1453)'  }    -0.091424
    {'TmAtAddress'}    {'[15.1453,Inf]'}      0.14546
    {'TmAtAddress'}    {'<missing>'    }          NaN
    {'ResStatus'  }    {'Tenant'       }    -0.024878
    {'ResStatus'  }    {'Home Owner'   }      0.11858
    {'ResStatus'  }    {'Other'        }      0.30343
    {'ResStatus'  }    {'<missing>'    }          NaN
    {'EmpStatus'  }    {'Unknown'      }    -0.097539
    {'EmpStatus'  }    {'Employed'     }       0.3319
    {'EmpStatus'  }    {'<missing>'    }          NaN
    {'CustIncome' }    {'[-Inf,31500)' }     -0.30942
    {'CustIncome' }    {'[31500,38500)'}     -0.09789
    {'CustIncome' }    {'[38500,45000)'}      0.21233
    {'CustIncome' }    {'[45000,Inf]'  }        0.494
    {'CustIncome' }    {'<missing>'    }          NaN
      ⋮
```

Before computing probabilities of default with the test data, you need to transform the test data using the same transformation as for the training data. To make this transformation, use the `transform` method of the `remover` object and pass it in the `data_Test` data set. Then, use `probdefault` to compute the likelihood of default of the `data_Test` data set.

```
newTestTbl = transform(remover,data_Test);
pd2 = probdefault(sc2,newTestTbl);
predictions2 = double(pd2>threshold);
```

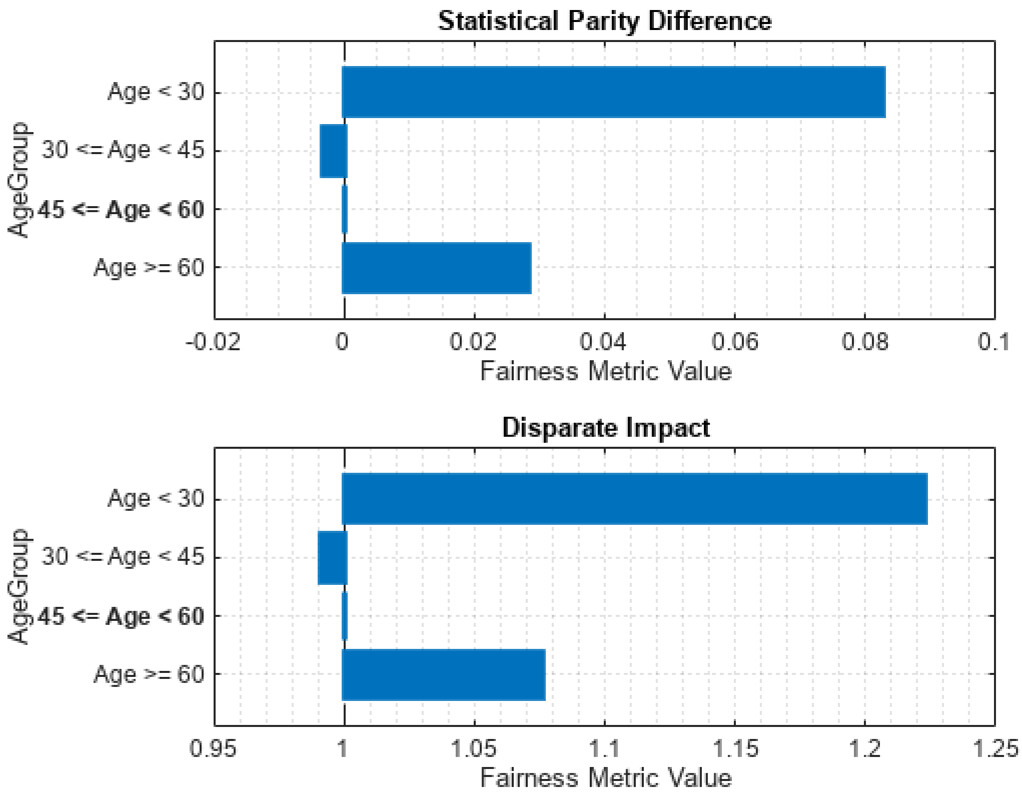
Use `fairnessMetrics` to compute fairness metrics at the model level and use `report` to generate a fairness metrics report.

```
modelMetrics2 = fairnessMetrics(newTestTbl,'status','Predictions',predictions2,'SensitiveAttribu
mmReport2 = report(modelMetrics2,'GroupMetrics','GroupCount')
```

```
mmReport2=4×8 table
    ModelNames    SensitiveAttributeNames       Groups        StatisticalParityDifference    Di
    __________    _______________________    ______________    ___________________________    __

      Model1             AgeGroup            Age < 30                    0.082751
      Model1             AgeGroup            30 <= Age < 45            -0.0033738
      Model1             AgeGroup            45 <= Age < 60                     0
      Model1             AgeGroup            Age >= 60                   0.028205
```

Use `plot` to visualize the statistical parity difference (`'spd'`) and disparate impact (`'di'`) bias metrics.

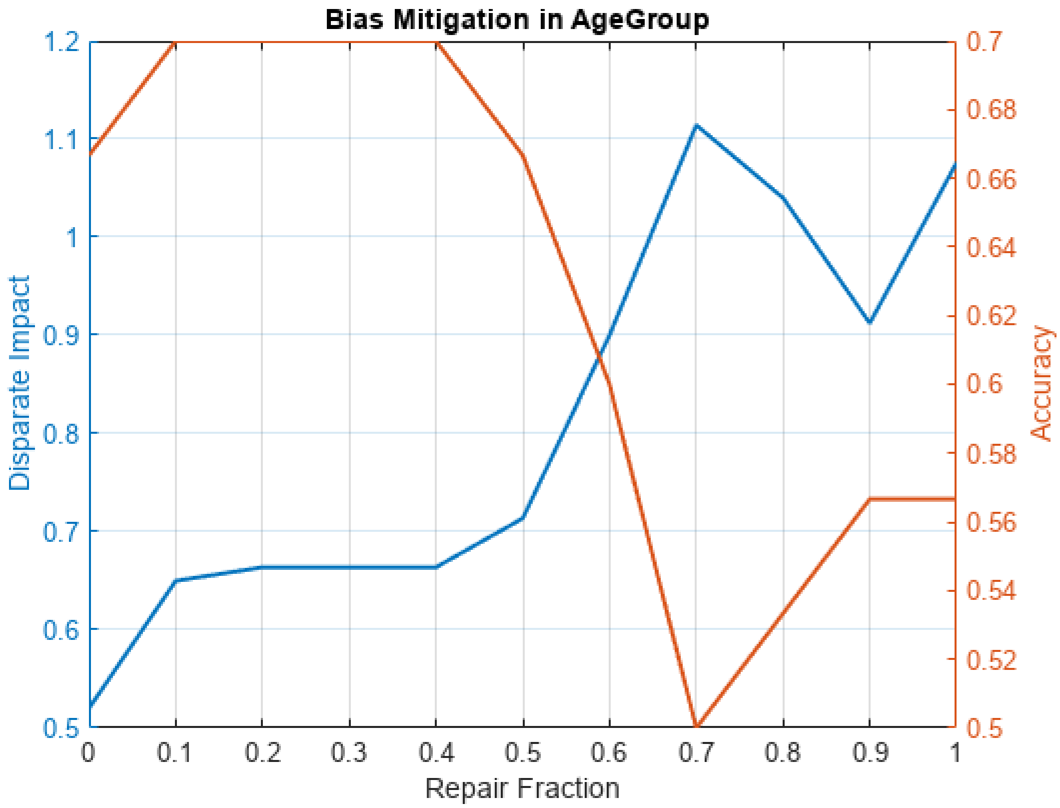
```
figure
tiledlayout(2,1)
nexttile
plot(modelMetrics2,'spd')
nexttile
plot(modelMetrics2,'di')
```



Plot Disparate Impact and Accuracy for Different Repair Fraction Values

In this example, the bias mitigation process uses `disparateImpactRemover` to set `RepairFraction = 1` in order to mitigate bias. However, it is useful to see how the disparate impact and accuracy varies with a change in the `RepairFraction` value. For example, use the `AgeGroup` predictor and plot the disparate impact and accuracy of the different subgroups for different values of `RepairFraction`.

```
subgroup = Age >= 60 ;
r = 0:0.1:1;
Accuracy = zeros(11,1);
di = zeros(11,1);

for i = 0:1:10
    [rmvr, trainTbl] = disparateImpactRemover(data_Train, 'AgeGroup' , ...
           'PredictorNames', {'TmAtAddress','CustIncome','TmWBank','AMBalance','UtilRate'},'Repai
    testTbl = transform(rmvr, data_Test);
    sc = creditscorecard(trainTbl,'IDVar','CustID', ...
        'PredictorVars',PredictorVars,'ResponseVar','status');
    sc = autobinning(sc);
    sc = fitmodel(sc,'VariableSelection','fullmodel','Display','off');
    pd = probdefault(sc,testTbl);
    predictions = double(pd>threshold);
    modelMetrics = fairnessMetrics(newTestTbl, 'status', 'Predictions', predictions, 'SensitiveAt
    mmReport = report(modelMetrics,'BiasMetrics','di','GroupMetrics','accuracy');
    di(i+1) = mmReport.DisparateImpact(subgroup);
    Accuracy(i+1) = mmReport.Accuracy(subgroup);
end

figure
yyaxis left
plot(r, di,'LineWidth', 1.5)
title('Bias Mitigation in AgeGroup ')
xlabel('Repair Fraction')
ylabel('Disparate Impact')
yyaxis right
plot(r, Accuracy,'LineWidth', 1.5)
ylabel('Accuracy')
grid on
```

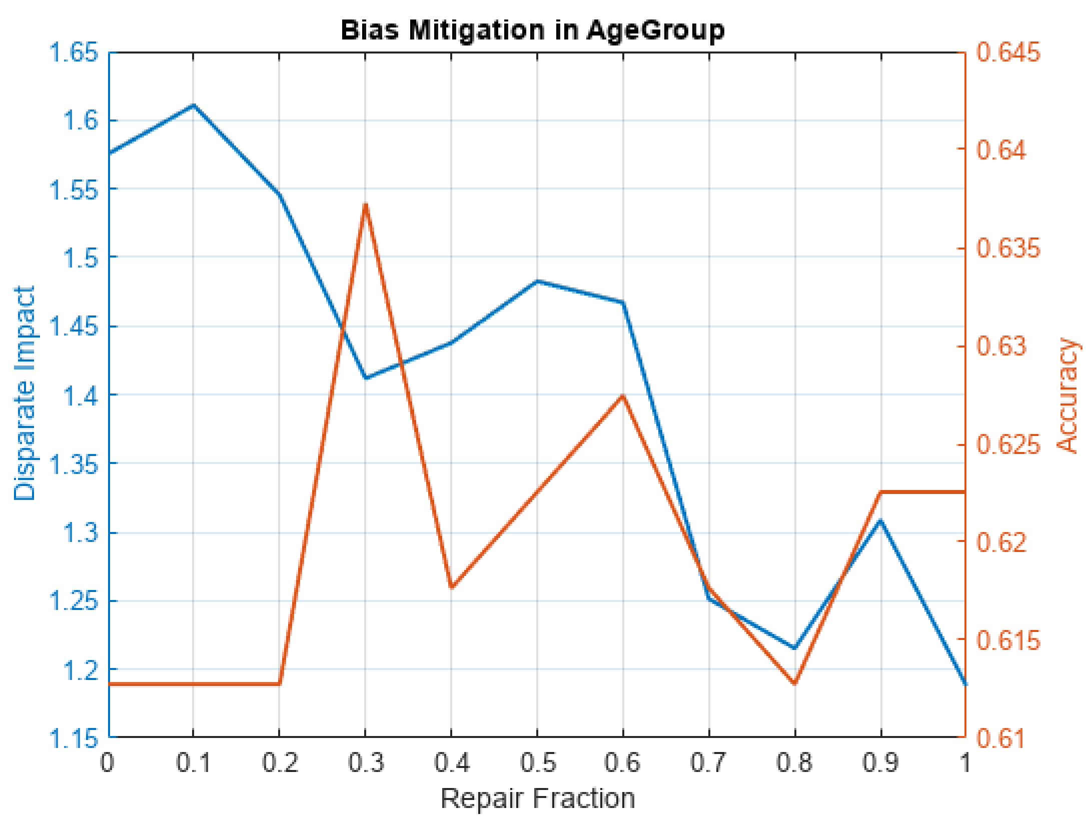


If you select the subgroup `'Age < 30'` from this plot, you can see that the accuracy increases as the `RepairFraction` value increases. Although this seems counterintuitive, looking further at the `GroupCount` of that age group in the `mmReport2` table, this group has only 22 observations. This small number of observations explains the anomaly in this plot.

One way to mitigate this issue of not having enough data for a subgroup is to combine all unprivileged groups and compare them as one group against the privileged group. The following code shows you how by setting the majority group (`45 <= Age < 60`) as the privileged group and then by combining every other group into one and setting that group as the unprivliged group.

```
privilegedGroup = '45 <= Age < 60';
twoAgeGroups_TrainTbl = data_Train;
twoAgeGroups_TrainTbl.AgeGroup = addcats(twoAgeGroups_TrainTbl.AgeGroup,'Other','After','Age >= 6
twoAgeGroups_TrainTbl.AgeGroup(twoAgeGroups_TrainTbl.AgeGroup ~= privilegedGroup) = 'Other';
twoAgeGroups_TestTbl = data_Test;
twoAgeGroups_TestTbl.AgeGroup = addcats(twoAgeGroups_TestTbl.AgeGroup,'Other','After','Age >= 60
twoAgeGroups_TestTbl.AgeGroup(twoAgeGroups_TestTbl.AgeGroup ~= privilegedGroup) = 'Other';

r = 0:0.1:1;
Accuracy = zeros(11,1);
di = zeros(11,1);

for i = 0:1:10
  [rmvr, trainTbl] = disparateImpactRemover(twoAgeGroups_TrainTbl, 'AgeGroup' , ...
          'PredictorNames', {'TmAtAddress','CustIncome','TmWBank','AMBalance','UtilRate'},'Repai
   testTbl = transform(rmvr, twoAgeGroups_TestTbl);
   sc = creditscorecard(trainTbl,'IDVar','CustID', ...
```

```
        'PredictorVars',PredictorVars,'ResponseVar','status');
    sc = autobinning(sc);
    sc = fitmodel(sc,'VariableSelection','fullmodel','Display','off');
    pd = probdefault(sc,testTbl);
    predictions = double(pd>threshold);
    modelMetrics = fairnessMetrics(twoAgeGroups_TestTbl, 'status', 'Predictions', predictions, ...
        'SensitiveAttributeNames','AgeGroup','ReferenceGroup','45 <= Age < 60');
     mmReport = report(modelMetrics,'BiasMetrics','di','GroupMetrics','accuracy');
     di(i+1) = mmReport.DisparateImpact(2);
     Accuracy(i+1) = mmReport.Accuracy(2);
end

figure
yyaxis left
plot(r, di,'LineWidth', 1.5)
title('Bias Mitigation in AgeGroup ')
xlabel('Repair Fraction')
ylabel('Disparate Impact')
yyaxis right
plot(r, Accuracy,'LineWidth', 1.5)
ylabel('Accuracy')
grid on
```



You can use this privileged group and unprivliged group method if the goal is not to measure the bias of each individual group against the privileged group, but rather to measure the overall fairness of all customers who are not part of the privileged group.

References

[1] Nielsen, Aileen. "Chapter 4. Fairness PreProcessing." *Practical Fairness*. O'Reilly Media, Inc., Dec. 2020.

[2] Mehrabi, Ninareh, et al. “A Survey on Bias and Fairness in Machine Learning.” *ArXiv:1908.09635 [Cs]*, Sept. 2019. *arXiv.org*, https://arxiv.org/abs/1908.09635.

[3] Wachter, Sandra, et al. *Bias Preservation in Machine Learning: The Legality of Fairness Metrics Under EU Non-Discrimination Law*. SSRN Scholarly Paper, ID 3792772, Social Science Research Network, 15 Jan. 2021. *papers.ssrn.com*, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3792772.

See Also

`creditscorecard` | `autobinning` | `fitmodel` | `displaypoints` | `probdefault`

Related Examples

- “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23
- “Explore Fairness Metrics for Credit Scoring Model” on page 3-98
- “Bias Mitigation in Credit Scoring by Reweighting” on page 3-110

Create Custom Lifetime PD Model for Credit Scorecard Model with Function Handle

This example shows how to use `customLifetimePDModel` to create a lifetime model for the probability of default. Using a retail credit data in panel format, you can create a credit scorecard model and then use a function handle with `customLifetimePDModel` to create a lifetime PD model.

Fit Credit Scorecard Model

Load the data set.

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);

nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Use `creditscorecard` to create a `creditscorecard` object and then use `autobinning` to bin the data. Alternativeliy, you can bin the data using the Binning Explorer. You can fit the model using `fitmodel`.

```
sc = creditscorecard(data(TrainDataInd,:),'IDVar','ID','PredictorVars',{'ScoreGroup' 'YOB' 'GDP'
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'Default'
               WeightsVar: ''
                 VarNames: {'ID'  'ScoreGroup'  'YOB'  'Default'  'Year'  'GDP'  'Market'}
        NumericPredictors: {'YOB'  'GDP'  'Market'}
    CategoricalPredictors: {'ScoreGroup'}
           BinMissingData: 0
                    IDVar: 'ID'
            PredictorVars: {'ScoreGroup'  'YOB'  'GDP'  'Market'}
                     Data: [388097x7 table]
```

```
sc = autobinning(sc);
sc = autobinning(sc,'YOB','Algorithm','Split');
sc = fitmodel(sc);
```

```
1. Adding ScoreGroup, Deviance = 42417.8562, Chi2Stat = 986.130141, PValue = 1.85820778e-216
2. Adding YOB, Deviance = 41644.7594, Chi2Stat = 773.096796, PValue = 3.81440566e-170
3. Adding Market, Deviance = 41616.8646, Chi2Stat = 27.8948108, PValue = 1.28092837e-07
4. Adding GDP, Deviance = 41612.2361, Chi2Stat = 4.62852205, PValue = 0.0314446396
```

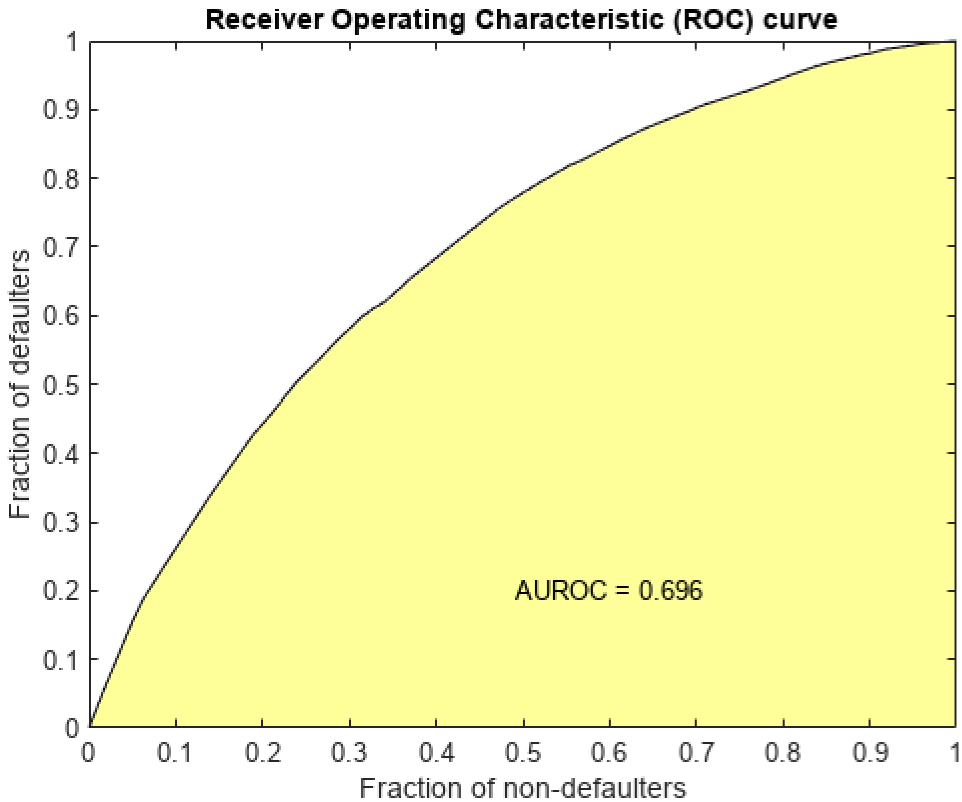
```
Generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue   
                   ________    ________    ______    ___________

    (Intercept)     4.6006     0.017273    266.35              0
    ScoreGroup     0.98953     0.033117     29.88    3.5837e-196
    YOB             1.0439      0.04216     24.76    2.4054e-135
    GDP             4.5496       2.1012    2.1652        0.03037
    Market          1.6892      0.44761    3.7738     0.00016076


388097 observations, 388092 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.79e+03, p-value = 0
```

```
displaypoints(sc)
```

```
ans=16×3 table
      Predictors            Bin           Points 
    ______________    ________________    _______

    {'ScoreGroup'}    {'High Risk'   }    0.61946
    {'ScoreGroup'}    {'Medium Risk'}      1.3073
    {'ScoreGroup'}    {'Low Risk'    }     1.8816
    {'ScoreGroup'}    {'<missing>'   }        NaN
    {'YOB'       }    {'[-Inf,2)'    }    0.56097
    {'YOB'       }    {'[2,5)'       }     1.0021
    {'YOB'       }    {'[5,7)'       }     1.4673
    {'YOB'       }    {'[7,Inf]'     }     2.4996
    {'YOB'       }    {'<missing>'   }        NaN
    {'GDP'       }    {'[-Inf,0.63)'}       1.051
    {'GDP'       }    {'[0.63,Inf]' }      1.1664
    {'GDP'       }    {'<missing>'   }        NaN
    {'Market'    }    {'[-Inf,2.78)'}      1.0661
    {'Market'    }    {'[2.78,9.48)'}      1.1262
    {'Market'    }    {'[9.48,Inf]' }      1.2358
    {'Market'    }    {'<missing>'   }        NaN
```

Validate the `creditscorecard` model using `validatemodel`.

```
figure;
s = validatemodel(sc,data(TestDataInd,:),'Plot','roc');
```




```
disp(s)
```

```
             Measure               Value
    _________________________    _______

    {'Accuracy Ratio'       }    0.39124
    {'Area under ROC curve'}     0.69562
    {'KS statistic'         }    0.28409
    {'KS score'             }     4.6019
```

Wrap Credit Scorecard Model as Lifetime PD Model

Create a function handle for the `probdefault` function of the `creditscorecard` object. The only variable in the function handle (`predictFcnHandle`) is the data. The `creditscorecard` object (`sc`) is a fixed parameter of the `probdefault` function.

Use `customLifetimePDModel` to create an instance of a custom lifetime PD model using the function handle `predictFcnHandle`. Also, set up variable names for the model. The base class `LifetimePDModel` uses those variable names for different validations and computations.

```
predictFcnHandle = @(data)probdefault(sc,data);
pdModel = customLifetimePDModel(predictFcnHandle,'ModelID','MyScorecardModel','IDVar','ID','AgeVa
```

```
pdModel = 
  CustomLifetimePD with properties:

            ModelID: "MyScorecardModel"
        Description: ""
```

```
    UnderlyingModel: @(data)probdefault(sc,data)
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

```
pdModel.UnderlyingModel
```

```
ans = function_handle with value:
    @(data)probdefault(sc,data)
```

Predict and Validate Scores Using Custom Lifetime PD Model

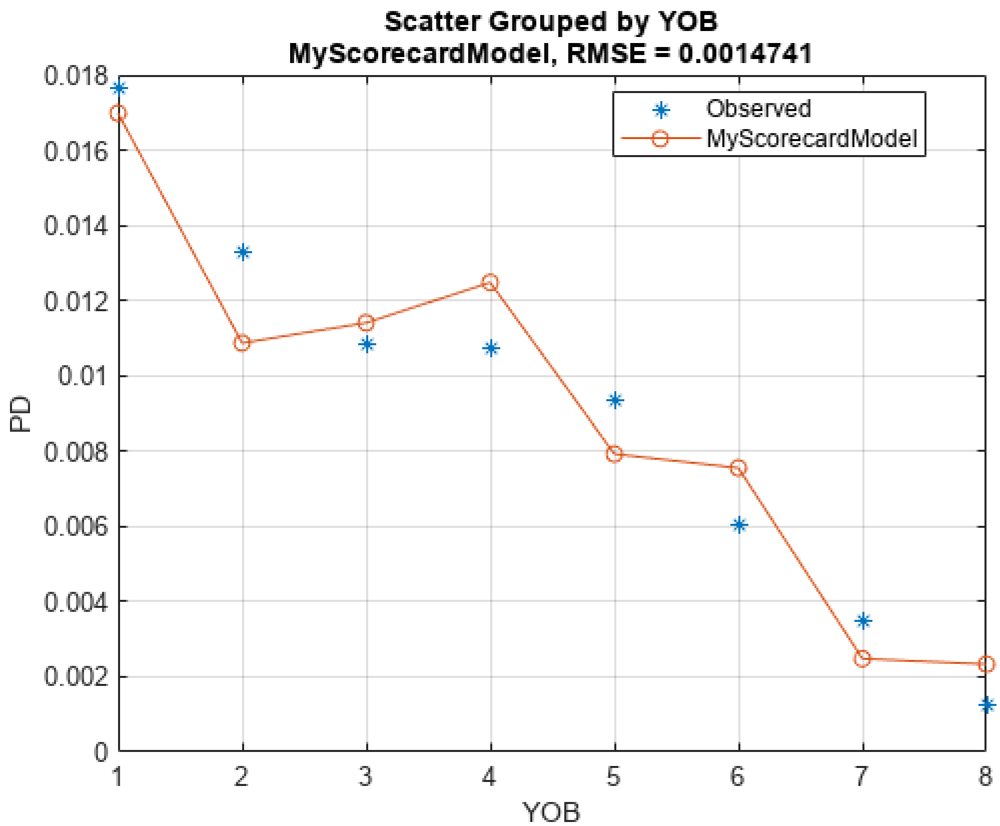
You can use `pdModel` like any other lifetime PD model. The training and test data sets are in panel data format and can be passed to either `predict` or `predictLifetime`. The `predict` function returns the conditional PD, the same prediction as the `probdefault` function for the credit scorecard. The `predictLifetime` function returns the cumulative probability of default for each ID. Here, the first ID in the test data set spans the first eight rows. The conditional PD can go up or down, but the cumulative PD always increases from one period to the next.

```
CondPD = predict(pdModel,data(TestDataInd,:));
LifetimePD = predictLifetime(pdModel,data(TestDataInd,:));
disp([CondPD(1:8) LifetimePD(1:8)])
```

```
    0.0154    0.0154
    0.0089    0.0241
    0.0089    0.0328
    0.0099    0.0424
    0.0066    0.0488
    0.0075    0.0559
    0.0022    0.0580
    0.0020    0.0599
```

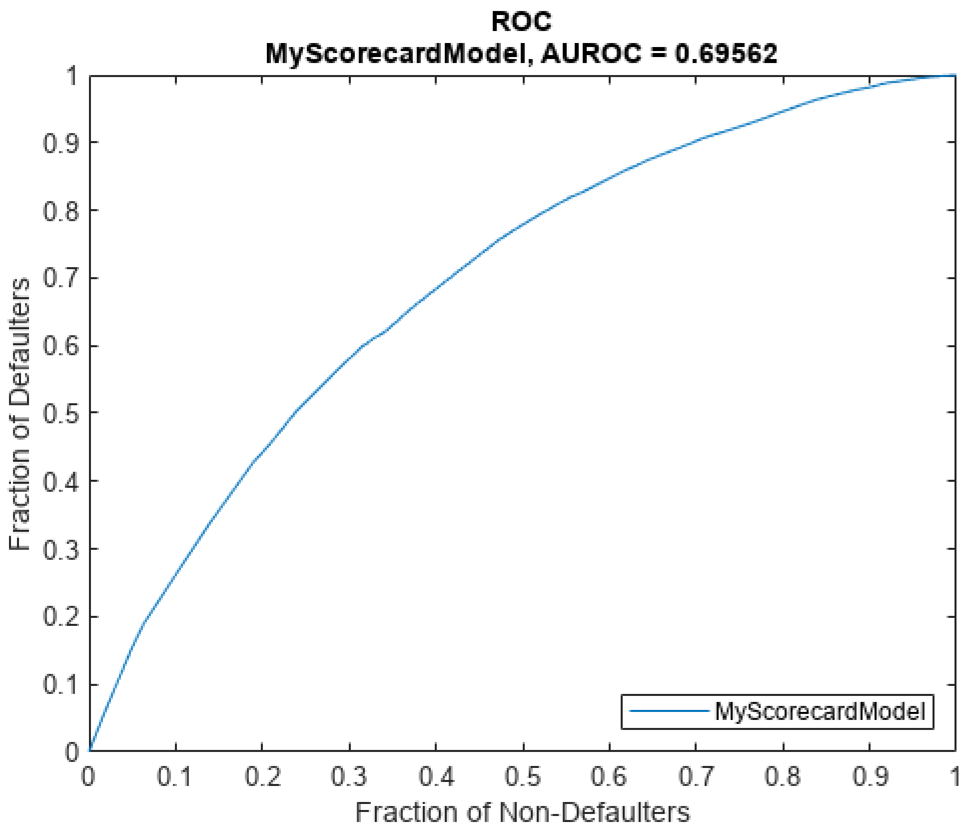
By wrapping the credit scorecard as a lifetime PD model object (`pdModel`), you can use all the validation capabilities of lifetime PD models are available. Use `modelCalibrationPlot` to plot observed default rates compared to the predicted PDs on grouped data.

```
figure;
modelCalibrationPlot(pdModel,data(TestDataInd,:),'YOB')
```



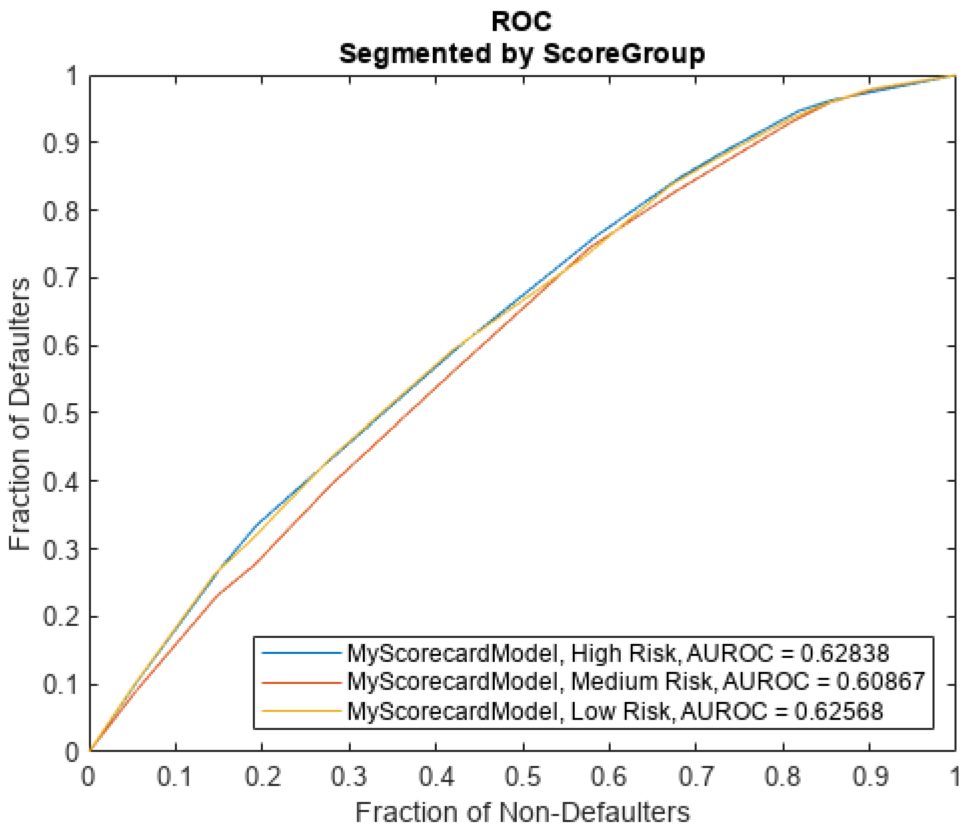
Use `modelDiscriminationPlot` to plot the ROC curve.

```
figure;
modelDiscriminationPlot(pdModel,data(TestDataInd,:))
```



Use `modelDiscriminationPlot` to plot the ROC curve and segment the data by `ScoreGroup`.

```
figure;
modelDiscriminationPlot(pdModel,data(TestDataInd,:),'SegmentBy','ScoreGroup')
```



See Also

`customLifetimePDModel` | `fitLifetimePDModel` | `predict` | `predictLifetime` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot`

Related Examples

- “Create Custom Lifetime PD Model for Decision Tree Model with Function Handle” on page 4-224
- “Credit Scorecard Modeling with Missing Values”

More About

- “Overview of Binning Explorer” on page 3-2
- “About Credit Scorecards”
- “Credit Scorecard Modeling Workflow”

External Websites

- Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

Corporate Credit Risk Simulations for Portfolios

Credit Simulation Using Copulas

In this section...

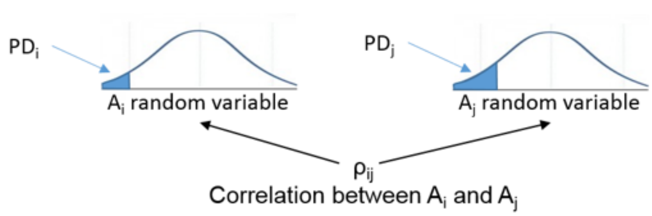
Predicting the credit losses for a counterparty depends on three main elements:

- Probability of default (PD)
- Exposure at default (EAD), the value of the instrument at some future time
- Loss given default (LGD), which is defined as 1 − *Recovery*

If these quantities are known at future time t, then the expected loss is PD × EAD × LGD. In this case, you can model the expected loss for a single counterparty by using a binomial distribution. The difficulty arises when you model a portfolio of these counterparties and you want to simulate them with some default correlation.

To simulate correlated defaults, the copula model associates each counterparty with a random variable, called a "latent" variable. These latent variables are correlated using some proxy for their credit worthiness, for example, their stock price. These latent variables are then mapped to default or nondefault outcomes such that the default occurs with probability PD.

This figure summarizes the copula simulation approach.



The random variable A_i associated to the ith counterparty falls in the default shaded region with probability PDi. If the simulated value falls in that region, it is interpreted as a default. The jth counterparty follows a similar pattern. If the A_i and A_j random variables are highly correlated, they tend to both have high values (no default), or both have low values (fall in the default region). Therefore, there is a default correlation.

Factor Models

For M issuers, $M(M-1)/2$ correlation parameters are required. For $M = 1000$, this is about half a million correlations. One practical variation of the approach is the one-factor model, which makes all the latent variables dependent on a single factor. This factor Z represents the underlying systemic credit quality in the economy. This model also includes a random idiosyncratic error.

$$A_i = w_i Z + \sqrt{1 - w_i^2}\varepsilon_i$$

This significantly reduces the input-data requirements, because now you need only the M sensitivities, that is, the weights w1,...,wM. If Z and ε_i are standard normal variables, then Ai is also a standard normal.

An extension of the one-factor model is a multifactor model.

$$A_i = w_{i1}Z_1 + \ldots + w_{iK}Z_K + w_{i\varepsilon}\varepsilon_i$$

This model has several factors, each one associated with some underlying credit driver. For example, you can have factors for different regions or countries, or for different industries. Each latent variable is now a combination of several random variables plus the idiosyncratic error (epsilon) again.

When the latent variables *Ai* are normally distributed, there is a Gaussian copula. A common alternative is to let the latent variables follow a *t* distribution, which leads to a *t* copula. *t* copulas result in heavier tails than Gaussian copulas. Implied credit correlations are also larger with *t* copulas. Switching between these two copula approaches can provide important information on model risk.

Supported Simulations

Risk Management Toolbox supports simulations for counterparty credit defaults and counterparty credit rating migrations.

Credit Default Simulation

The `creditDefaultCopula` object is used to simulate and analyze multifactor credit default simulations. These simulations assume that you calculated the main inputs to this model on your own. The main inputs to this model are:

- `PD` — Probability of default
- `EAD` — Exposure at default
- `LGD` — Loss given default (1 − *Recovery*)
- `Weights` — Factor and idiosyncratic weights
- `FactorCorrelation` — An optional factor correlation matrix for multifactor models

The `creditDefaultCopula` object enables you to simulate defaults using the multifactor copula and return the results as a distribution of losses on a portfolio and counterparty level. You can also use the `creditDefaultCopula` object to calculate several risk measures at the portfolio level and the risk contributions from individual obligors. The outputs of the `creditDefaultCopula` model and the associated functions are:

- The full simulated distribution of portfolio losses across scenarios and the losses on each counterparty across scenarios. For more information, see `creditDefaultCopula` object properties and `simulate`.
- Risk measures (`VaR`, `CVaR`, `EL`, `Std`) with confidence intervals. See `portfolioRisk`.
- Risk contributions per counterparty (for `EL` and `CVaR`). See `riskContribution`.
- Risk measures and associated confidence bands. See `confidenceBands`.
- Counterparty scenario details for individual losses for each counterparty. See `getScenarios`.

Credit Rating Migration Simulation

The `creditMigrationCopula` object enables you to simulate changes in credit rating for each counterparty.

The `creditMigrationCopula` object is used to simulate counterparty credit migrations. These simulations assume that you calculated the main inputs to this model on your own. The main inputs to this model are:

- `migrationValues` — Values of the counterparty positions for each credit rating.
- `ratings` — Current credit rating for each counterparty.
- `transitionMatrix` — Matrix of credit rating transition probabilities.
- `LGD` — Loss given default (1 − *Recovery*)
- `Weights` — Factor and idiosyncratic model weights

You can also use the `creditMigrationCopula` object to calculate several risk measures at the portfolio level and the risk contributions from individual obligors. The outputs of the `creditMigrationCopula` model and the associated functions are:

- The full simulated distribution of portfolio values. For more information, see `creditMigrationCopula` object properties and `simulate`.
- Risk measures (`VaR`, `CVaR`, `EL`, `Std`) with confidence intervals. See `portfolioRisk`.
- Risk contributions per counterparty (for `EL` and `CVaR`). See `riskContribution`.
- Risk measures and associated confidence bands. See `confidenceBands`.
- Counterparty scenario details for each counterparty. See `getScenarios`.

See Also

`creditDefaultCopula` | `creditMigrationCopula` | `asrf`

Related Examples

- “creditDefaultCopula Simulation Workflow” on page 4-5
- “creditMigrationCopula Simulation Workflow” on page 4-10
- “Modeling Correlated Defaults with Copulas” on page 4-18
- “One-Factor Model Calibration” on page 4-64

More About

- “Corporate Credit Risk” on page 1-3
- “Credit Rating Migration Risk” on page 1-10

creditDefaultCopula Simulation Workflow

This example shows a common workflow for using a `creditDefaultCopula` object for a portfolio of credit instruments.

For an example of an advanced workflow using the `creditDefaultCopula` object, see “Modeling Correlated Defaults with Copulas” on page 4-18.

Step 1. Create a creditDefaultCopula object with a two-factor model.

Load the saved portfolio data. Create a `creditDefaultCopula` object with a two-factor model using with the values `EAD`, `PD`, `LGD`, and `Weights2F`.

```
load CreditPortfolioData.mat;
cdc = creditDefaultCopula(EAD, PD, LGD,Weights2F,'FactorCorrelation',FactorCorr2F);
disp(cdc)
```

```
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioLosses: []
```

```
disp(cdc.Portfolio(1:10:100,:))
```

```
    ID     EAD          PD         LGD              Weights          
    __    ______    __________    ____    _________________________

     1    21.627     0.0050092    0.35      0.35         0      0.65
    11    29.338     0.0050092    0.55      0.35         0      0.65
    21    3.8275     0.0020125    0.25    0.1125    0.3375      0.55
    31    26.286     0.0020125    0.55    0.1125    0.0375      0.85
    41    42.868     0.0050092    0.55      0.25         0      0.75
    51    7.1259    0.00099791    0.25         0      0.25      0.75
    61    10.678     0.0020125    0.35         0      0.15      0.85
    71     2.395    0.00099791    0.55         0      0.15      0.85
    81    26.445      0.060185    0.55         0      0.45      0.55
    91    7.1637       0.11015    0.25      0.35         0      0.65
```

Step 2. Set the VaRLevel to 99%.

Set the `VarLevel` property for the `creditDefaultCopula` object to 99% (the default is 95%).

```
cdc.VaRLevel = 0.99;
```

Step 3. Run a simulation.

Use the `simulate` function to run a simulation on the `creditDefaultCopula` object for 100,000 scenarios.

```
 cdc = simulate(cdc,1e5)
```

```
cdc = 
  creditDefaultCopula with properties:
```

```
            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioLosses: [30.1008 3.6910 3.2895 19.2151 7.5761 44.5088 19.5419 1.7909 72.1443 12.69
```

Step 4. Generate a report for the portfolio risk.

Use the `portfolioRisk` function to obtain a report for risk measures and confidence intervals for `EL`, `Std`, `VaR`, and `CVaR`.

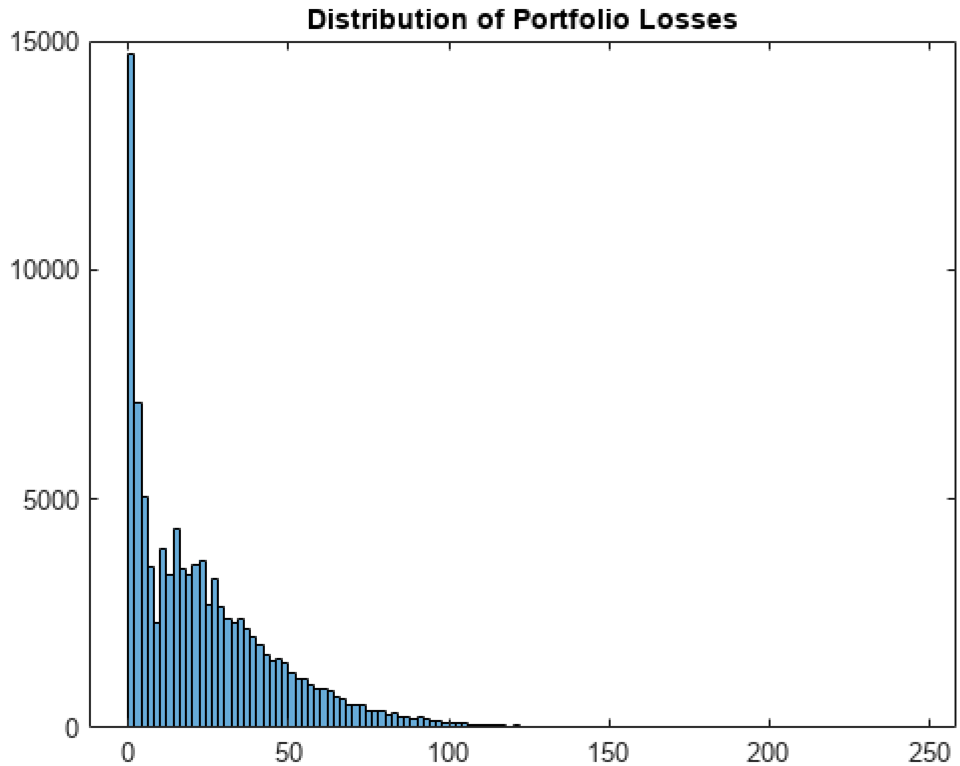
```
[portRisk,RiskConfidenceInterval] = portfolioRisk(cdc)
```

```
portRisk=1×4 table
      EL        Std       VaR      CVaR 
    ______    ______    _____    ______

    24.876    23.778    102.4    121.28


RiskConfidenceInterval=1×4 table
           EL                  Std                  VaR                  CVaR      
    ________________    ________________    ________________    ________________

    24.729    25.023    23.674    23.883    101.19     103.5    120.13    122.42
```

Step 5. Visualize the distribution.

Use the `histogram` function to display the distribution for `EL`, `VaR`, and `CVaR`.

```
histogram(cdc.PortfolioLosses);
title('Distribution of Portfolio Losses');
```



Step 6. Generate a risk contributions report.

Use the `riskContribution` function to display the risk contribution. The risk contributions, `EL` and `CVaR`, are *additive*. If you sum each of these two metrics over all the counterparties, you get the values reported for the entire portfolio in the `portfolioRisk` table.

```
rc = riskContribution(cdc);

disp(rc(1:10,:))
```

ID	EL	Std	VaR	CVaR
1	0.036031	0.022762	0.083828	0.13625
2	0.068357	0.039295	0.23373	0.24984
3	1.2228	0.60699	2.3184	2.3775
4	0.002877	0.00079014	0.0024248	0.0013137
5	0.12127	0.037144	0.18474	0.24622
6	0.12638	0.078506	0.39779	0.48334
7	0.84284	0.3541	1.6221	1.8183
8	0.00090088	0.00011379	0.0016463	0.00089197
9	0.93117	0.87638	3.3868	3.9936
10	0.26054	0.37918	1.7399	2.3042

Step 7. Simulate the risk exposure with a t copula.

Use the `simulate` function with optional input arguments for `Copula` and `t`. Save the results to a new `creditDefaultCopula` object (`cct`).

```
cdct = simulate(cdc,1e5,'Copula','t','DegreesOfFreedom',10)
```

```
cdct = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioLosses: [3.6910 1.9775 128.4550 2.1852 4.8512 0 26.2682 0 54.8980 24.3618 16.8483
```

Step 8. Compare confidence bands for different copulas.

Use the `confidenceBands` function to compare confidence bands for the two different copulas.

```
confidenceBands(cdc,'RiskMeasure','Std','ConfidenceIntervalLevel',0.90,'NumPoints',10)
```

```
ans=10×4 table
    NumScenarios    Lower      Std      Upper 
    ____________    ______    ______    ______

       10000        23.525    23.799    24.079
       20000        23.564    23.758    23.955
       30000        23.543    23.701    23.861
       40000        23.621    23.758    23.897
       50000        23.565    23.687    23.811
       60000        23.604    23.716    23.829
       70000        23.688    23.792    23.897
       80000        23.663     23.76    23.858
       90000        23.639     23.73    23.823
       1e+05        23.691    23.778    23.866
```

```
confidenceBands(cdct,'RiskMeasure','Std','ConfidenceIntervalLevel',0.90,'NumPoints',10)
```

```
ans=10×4 table
    NumScenarios    Lower      Std      Upper 
    ____________    ______    ______    ______

       10000        31.923    32.294    32.675
       20000        31.775    32.036    32.302
       30000        31.759    31.972    32.188
       40000        31.922    32.107    32.295
       50000        32.012    32.179    32.347
       60000        31.911    32.062    32.216
       70000        31.879    32.019    32.161
       80000        31.909     32.04    32.173
       90000        31.866     31.99    32.114
       1e+05        31.933     32.05    32.169
```

See Also

`creditDefaultCopula` | `simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios` | `asrf`

Related Examples

- “Credit Simulation Using Copulas” on page 4-2
- “creditMigrationCopula Simulation Workflow” on page 4-10
- “Modeling Correlated Defaults with Copulas” on page 4-18
- “One-Factor Model Calibration” on page 4-64

More About

- “Risk Modeling with Risk Management Toolbox” on page 1-3

creditMigrationCopula Simulation Workflow

This example shows a common workflow for using a `creditMigrationCopula` object for a portfolio of counterparty credit ratings.

Step 1. Create a creditMigrationCopula object with a 4-factor model

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a `creditMigrationCopula` object with a 4-factor model using `creditMigrationCopula`.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioValues: []
```

Step 2. Set the VaRLevel to 99%.

Set the `VarLevel` property for the `creditMigrationCopula` object to 99% (the default is 95%).

```
cmc.VaRLevel = 0.99;
```

Step 3. Display the `Portfolio` property for information about migration values, ratings, LGDs, and weights.

Display the `Portfolio` property containing information about migration values, ratings, LGDs, and weights. The columns in the migration values are in the same order of the ratings, with the default rating in the last column.

```
head(cmc.Portfolio)
```

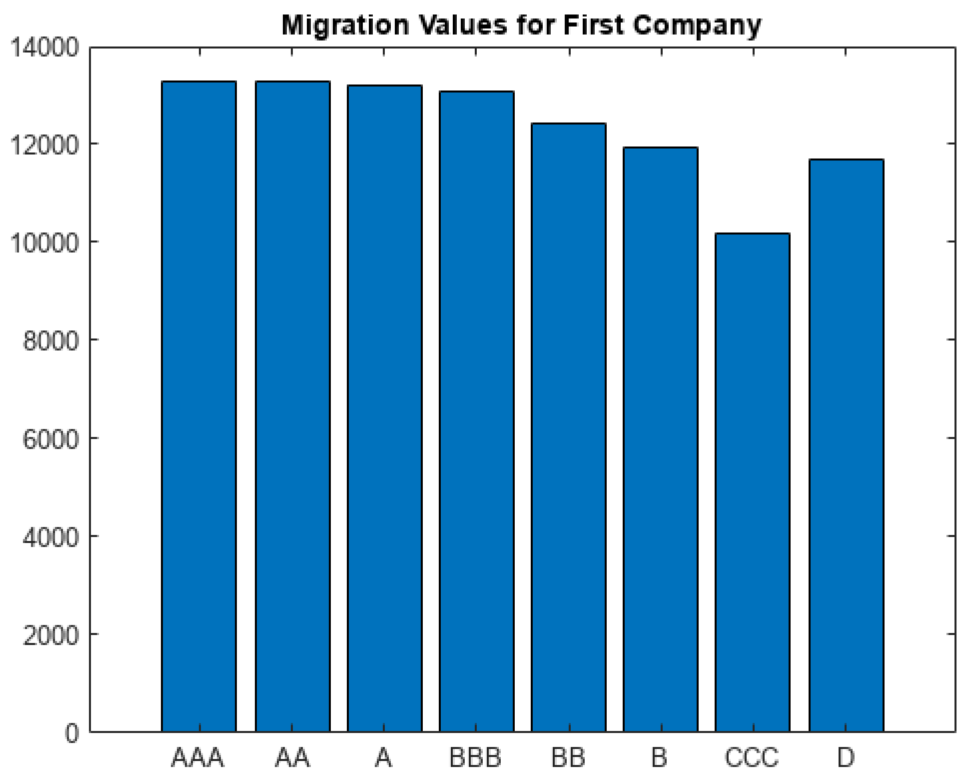
```
    ID    MigrationValues     Rating      LGD                      Weights
    __    _______________    _______    ______    ___________________________________

    1       1x8 double       "A"        0.6509    0        0         0      0.5     0.5
    2       1x8 double       "BBB"      0.8283    0     0.55         0        0    0.45
    3       1x8 double       "AA"       0.6041    0      0.7         0        0     0.3
    4       1x8 double       "BB"       0.6509    0     0.55         0        0    0.45
    5       1x8 double       "BBB"      0.4966    0        0      0.75        0    0.25
    6       1x8 double       "BB"       0.8283    0        0         0     0.65    0.35
```

```
    7       1x8 double       "BB"       0.6041      0        0        0    0.65    0.35
    8       1x8 double       "BB"       0.4873    0.5        0        0       0     0.5
```

Step 4. Display migration values for a counterparty.

For example, you can display the migration values for the first counterparty. Note that the value for default is higher than some of the non-default ratings. This is because the migration value for the default rating is a reference value (for example, face value, forward value at current rating, or other) that is multiplied by the recovery rate during the simulation to get the value of the asset in the event of default. The recovery rate is 1-`LGD` when the `LGD` input to `creditMigrationCopula` is a constant `LGD` value (the `LGD` input has one column). The recovery rate is a random quantity when the `LGD` input to `creditMigrationCopula` is specified as a mean and standard deviation for a beta distribution (the `LGD` input has two columns).

```
bar(cmc.Portfolio.MigrationValues(1,:))
xticklabels(cmc.RatingLabels)
title('Migration Values for First Company')
```



Step 5. Run a simulation.

Use the `simulate` function to simulate 100,000 scenarios.

```
cmc = simulate(cmc,1e5)
```

```
cmc = 
  creditMigrationCopula with properties:
```

```
            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioValues: [2.0082e+06 1.9950e+06 1.9933e+06 2.0009e+06 1.9819e+06 1.9955e+06 1.9962e
```

Step 6. Generate a report for the portfolio risk.

Use the `portfolioRisk` function to obtain a report for risk measures and confidence intervals for `EL`, `Std`, `VaR`, and `CVaR`.

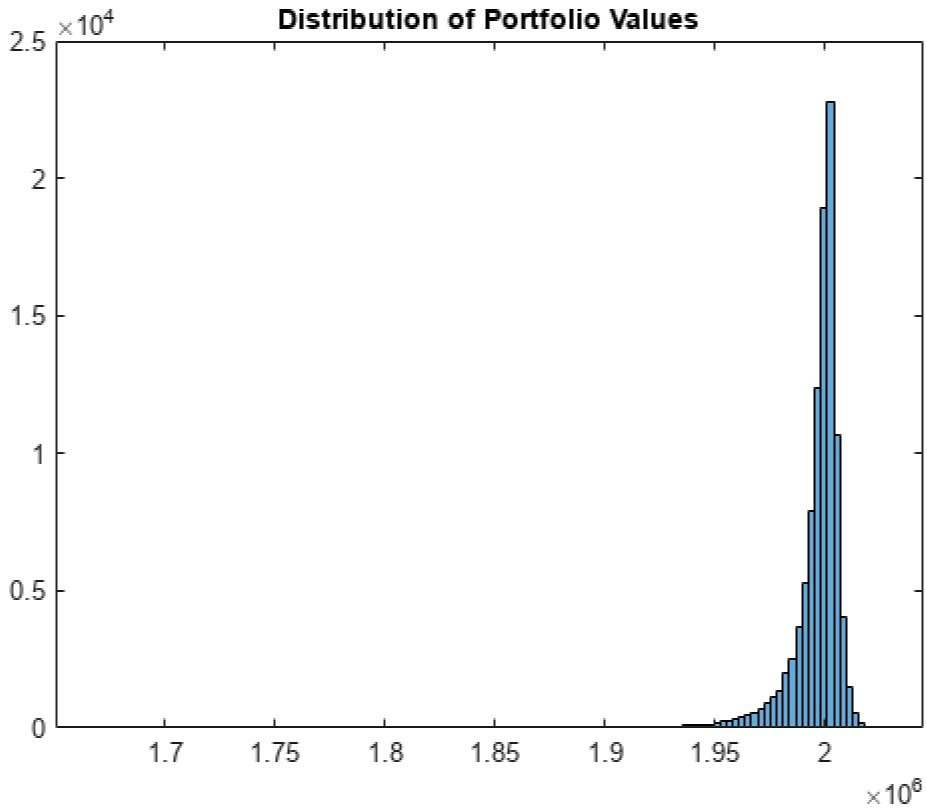
```
[portRisk,RiskConfidenceInterval] = portfolioRisk(cmc)
```

```
portRisk=1×4 table
      EL        Std       VaR      CVaR 
    ______     _____     _____     _____

    4515.9     12963     57176     83975

RiskConfidenceInterval=1×4 table
           EL                 Std                 VaR                CVaR      
    ________________    ______________    ______________    ______________

    4435.6    4596.3    12907    13021    55739    58541    82137    85812
```

Step 7. Visualize the distribution.

View a histogram of the portfolio values.

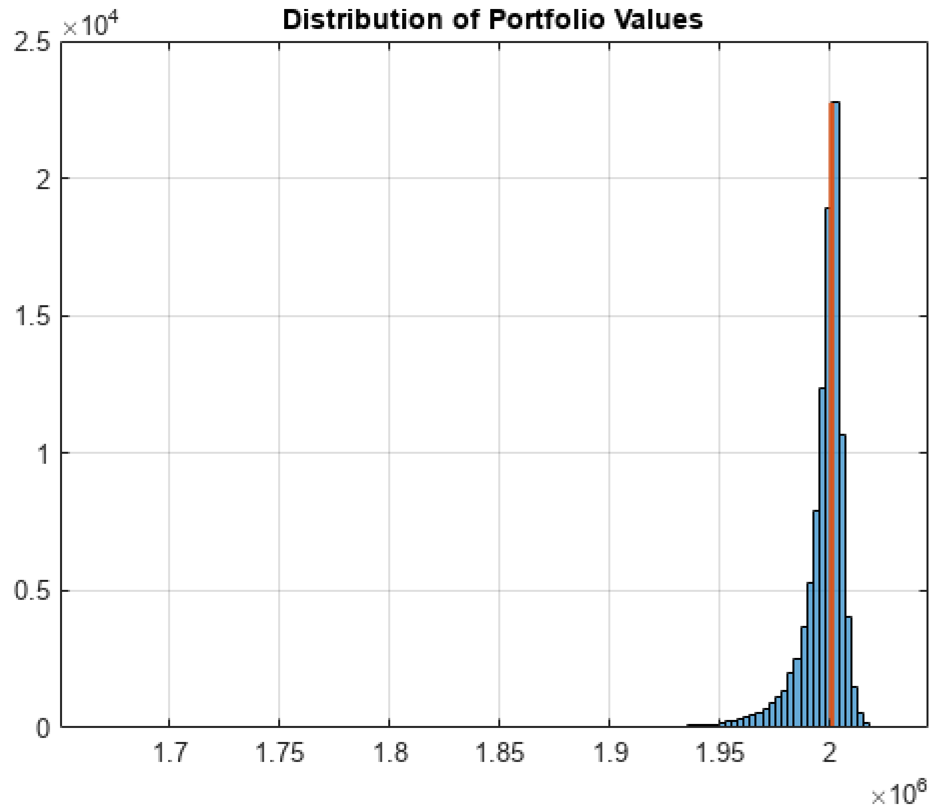
```
figure
h = histogram(cmc.PortfolioValues,125);
title('Distribution of Portfolio Values');
```



Step 8. Overlay the value if all counterparties maintain current credit ratings.

Overlay the value that the portfolio object (`cmc`) takes if all counterparties maintain their current credit ratings.

```
CurrentRatingValue = portRisk.EL + mean(cmc.PortfolioValues);

hold on
plot([CurrentRatingValue CurrentRatingValue],[0 max(h.Values)],'LineWidth',2);
grid on
```



Step 9. Generate a risk contributions report.

Use the riskContribution function to display the risk contribution. The risk contributions, EL and CVaR, are additive. If you sum each of these two metrics over all the counterparties, you get the values reported for the entire portfolio in the portfolioRisk table.

```
rc = riskContribution(cmc);
disp(rc(1:10,:))
```

ID	EL	Std	VaR	CVaR
1	15.521	41.153	238.72	279.18
2	8.49	18.838	92.074	122.19
3	6.0937	20.069	113.22	181.53
4	6.6964	55.885	272.23	313.25
5	23.583	73.905	360.32	573.39
6	10.722	114.97	445.94	728.38
7	1.8393	84.754	262.32	490.39
8	11.711	39.768	175.84	253.29
9	2.2154	4.4038	22.797	31.039
10	1.7453	2.5545	9.8801	17.603

Step 10. Simulate the risk exposure with a t copula.

To use a *t* copula with 10 degrees of freedom, use the simulate function with optional input arguments. Save the results to a new creditMigrationCopula object (cmct).

```
cmct = simulate(cmc,1e5,'Copula','t','DegreesOfFreedom',10)
```

```
cmct = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioValues: [2.0021e+06 2.0007e+06 1.9834e+06 2.0025e+06 2.0002e+06 2.0021e+06 2.0039e
```

Step 11. Generate a report for the portfolio risk for the t copula.

Use the `portfolioRisk` function to obtain a report for risk measures and confidence intervals for `EL`, `Std`, `VaR`, and `CVaR`.

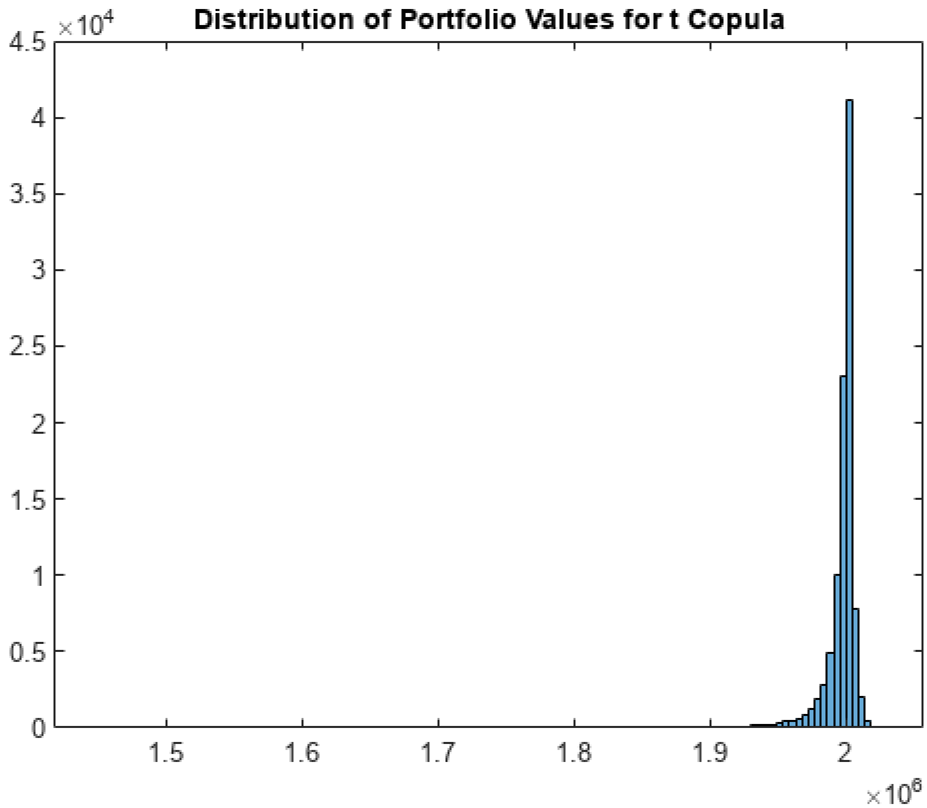
```
[portRisk2,RiskConfidenceInterval2] = portfolioRisk(cmct)
```

```
portRisk2=1×4 table
     EL      Std       VaR         CVaR   
    ____    _____    _____    __________

    4544    17034    72270    1.2391e+05


RiskConfidenceInterval2=1×4 table
           EL                  Std                 VaR                    CVaR           
    ________________    ______________    ______________    ________________________

    4438.5    4649.6    16960    17109    69769    75382    1.1991e+05    1.2791e+05
```

Step 12. Visualize the distribution for the t copula.

View a histogram of the portfolio values.

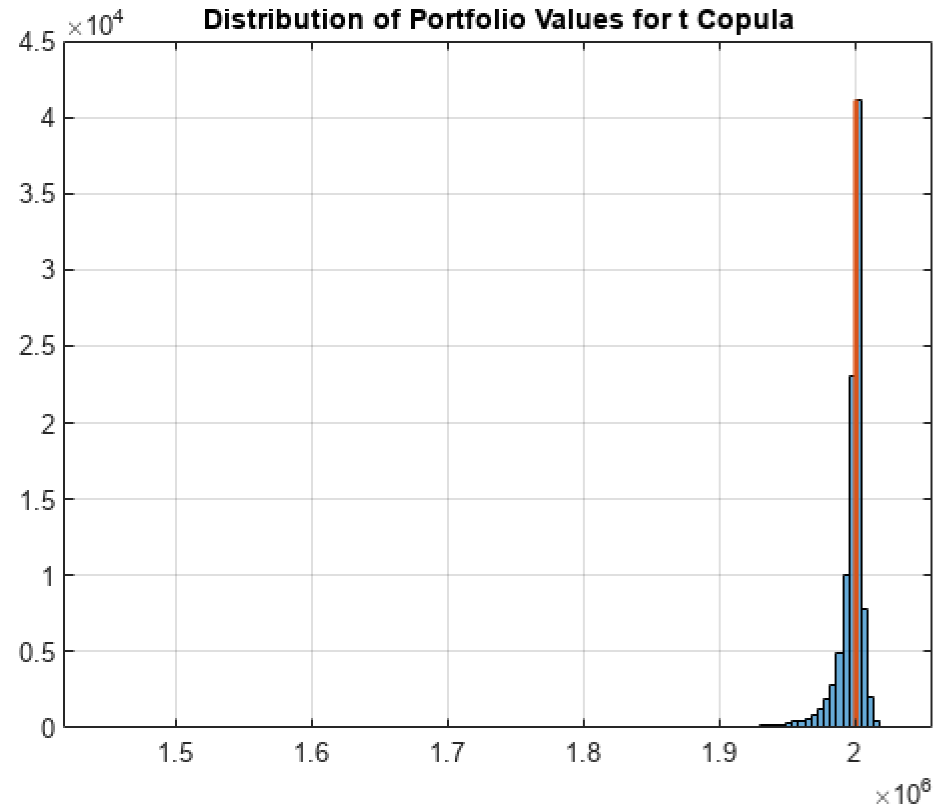
```
figure
h = histogram(cmct.PortfolioValues,125);
title('Distribution of Portfolio Values for t Copula');
```



Step 13. Overlay the value if all counterparties maintain current credit ratings for t copula.

Overlay the value that the portfolio object (`cmct`) takes if all counterparties maintain their current credit ratings.

```
CurrentRatingValue2 = portRisk2.EL + mean(cmct.PortfolioValues);

hold on
plot([CurrentRatingValue2 CurrentRatingValue2],[0 max(h.Values)],'LineWidth',2);
grid on
```



See Also

creditMigrationCopula | simulate | portfolioRisk | riskContribution | confidenceBands | getScenarios | asrf

Related Examples

- “Credit Simulation Using Copulas” on page 4-2
- “creditDefaultCopula Simulation Workflow” on page 4-5
- “Modeling Correlated Defaults with Copulas” on page 4-18
- “One-Factor Model Calibration” on page 4-64

More About

- “Credit Rating Migration Risk” on page 1-10

Modeling Correlated Defaults with Copulas

This example explores how to simulate correlated counterparty defaults using a multifactor copula model.

Potential losses are estimated for a portfolio of counterparties, given their exposure at default, default probability, and loss given default information. A `creditDefaultCopula` object is used to model each obligor's credit worthiness with latent variables. Latent variables are composed of a series of weighted underlying credit factors, as well as, each obligor's idiosyncratic credit factor. The latent variables are mapped to an obligor's default or nondefault state for each scenario based on their probability of default. Portfolio risk measures, risk contributions at a counterparty level, and simulation convergence information are supported in the `creditDefaultCopula` object.

This example also explores the sensitivity of the risk measures to the type of copula (Gaussian copula versus *t* copula) used for the simulation.

Load and Examine Portfolio Data

The portfolio contains 100 counterparties and their associated credit exposures at default (`EAD`), probability of default (`PD`), and loss given default (`LGD`). Using a `creditDefaultCopula` object, you can simulate defaults and losses over some fixed time period (for example, one year). The `EAD`, `PD`, and `LGD` inputs must be specific to a particular time horizon.

In this example, each counterparty is mapped onto two underlying credit factors with a set of weights. The `Weights2F` variable is a `NumCounterparties-by-3` matrix, where each row contains the weights for a single counterparty. The first two columns are the weights for the two credit factors and the last column is the idiosyncratic weights for each counterparty. A correlation matrix for the two underlying factors is also provided in this example (`FactorCorr2F`).

```
load CreditPortfolioData.mat
whos EAD PD LGD Weights2F FactorCorr2F
```

```
  Name                Size            Bytes  Class     Attributes

  EAD               100x1               800  double              
  FactorCorr2F        2x2                32  double              
  LGD               100x1               800  double              
  PD                100x1               800  double              
  Weights2F         100x3              2400  double              
```

Initialize the `creditDefaultCopula` object with the portfolio information and the factor correlation.

```
rng('default');
cc = creditDefaultCopula(EAD,PD,LGD,Weights2F,'FactorCorrelation',FactorCorr2F);

% Change the VaR level to 99%.
cc.VaRLevel = 0.99;

disp(cc)
```

```
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9900
```

```
         UseParallel: 0
     PortfolioLosses: []
```

```
cc.Portfolio(1:5,:)
```

```
ans=5×5 table
    ID     EAD         PD        LGD          Weights
    __    ______    _________    ____    __________________

    1     21.627    0.0050092    0.35    0.35       0    0.65
    2     3.2595     0.060185    0.35       0    0.45    0.55
    3     20.391      0.11015    0.55    0.15       0    0.85
    4     3.7534    0.0020125    0.35    0.25       0    0.75
    5     5.7193     0.060185    0.35    0.35       0    0.65
```

Simulate the Model and Plot Potential Losses

Simulate the multifactor model using the `simulate` function. By default, a Gaussian copula is used. This function internally maps realized latent variables to default states and computes the corresponding losses. After the simulation, the `creditDefaultCopula` object populates the `PortfolioLosses` and `CounterpartyLosses` properties with the simulation results.

```
cc = simulate(cc,1e5);
disp(cc)
```

```
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioLosses: [30.1008 3.6910 3.2895 19.2151 7.5761 44.5088 19.5419 1.7909 72.1443 12.69
```

The `portfolioRisk` function returns risk measures for the total portfolio loss distribution, and optionally, their respective confidence intervals. The value-at-risk (VaR) and conditional value-at-risk (CVaR) are reported at the level set in the `VaRLevel` property for the `creditDefaultCopula` object.

```
[pr,pr_ci] = portfolioRisk(cc);
```

```
fprintf('Portfolio risk measures:\n');
```

```
Portfolio risk measures:
```

```
disp(pr)
```

```
      EL        Std       VaR      CVaR
    ______    ______    _____    ______

    24.876    23.778    102.4    121.28
```

```
fprintf('\n\nConfidence intervals for the risk measures:\n');
```

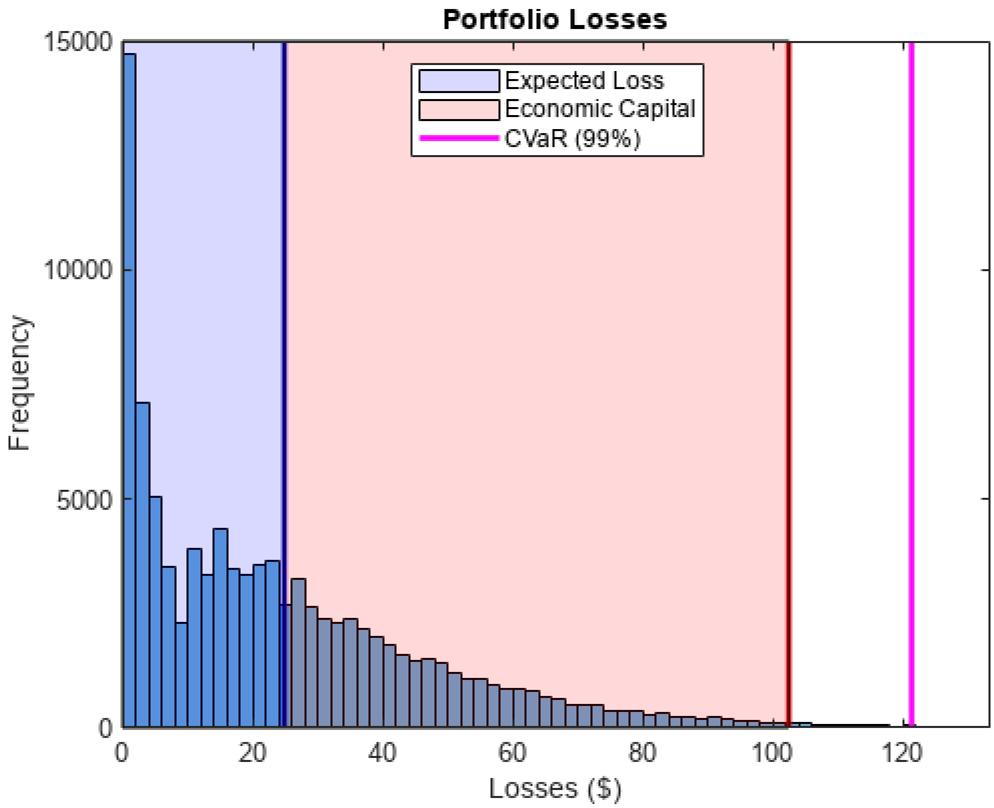
```
Confidence intervals for the risk measures:
```

```
disp(pr_ci)
```

```
        EL                 Std                VaR                CVaR
________________    ________________    ________________    ________________

24.729    25.023    23.674    23.883    101.19     103.5    120.13    122.42
```

Look at the distribution of portfolio losses. The expected loss (EL), VaR, and CVaR are marked as the vertical lines. The economic capital, given by the difference between the VaR and the EL, is shown as the shaded area between the EL and the VaR.

```
histogram(cc.PortfolioLosses)
title('Portfolio Losses');
xlabel('Losses ($)')
ylabel('Frequency')
hold on

% Overlay the risk measures on the histogram.
xlim([0 1.1 * pr.CVaR])
plotline = @(x,color) plot([x x],ylim,'LineWidth',2,'Color',color);
plotline(pr.EL,'b');
plotline(pr.VaR,'r');
cvarline = plotline(pr.CVaR,'m');

% Shade the areas of expected loss and economic capital.
plotband = @(x,color) patch([x fliplr(x)],[0 0 repmat(max(ylim),1,2)],...
    color,'FaceAlpha',0.15);
elband = plotband([0 pr.EL],'blue');
ulband = plotband([pr.EL pr.VaR],'red');
legend([elband,ulband,cvarline],...
    {'Expected Loss','Economic Capital','CVaR (99%)'},...
    'Location','north');
```



Find Concentration Risk for Counterparties

Find the concentration risk in the portfolio using the `riskContribution` function. `riskContribution` returns the contribution of each counterparty to the portfolio EL and CVaR. These additive contributions sum to the corresponding total portfolio risk measure.

```
rc = riskContribution(cc);

% Risk contributions are reported for EL and CVaR.
rc(1:5,:)
```

```
ans=5×5 table
    ID       EL            Std           VaR          CVaR
    __    ________    __________    _________    _________

    1     0.036031      0.022762     0.083828      0.13625
    2     0.068357      0.039295      0.23373      0.24984
    3       1.2228       0.60699       2.3184       2.3775
    4     0.002877    0.00079014    0.0024248    0.0013137
    5      0.12127      0.037144      0.18474      0.24622
```

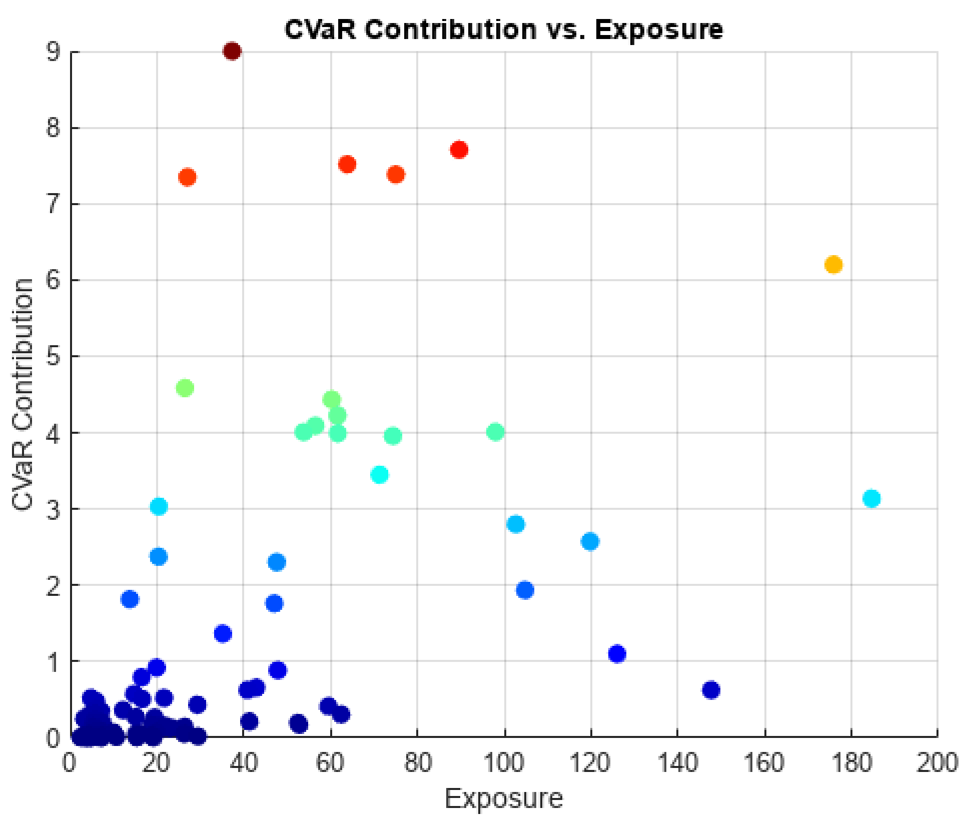
Find the riskiest counterparties by their CVaR contributions.

```
[rc_sorted,idx] = sortrows(rc,'CVaR','descend');
rc_sorted(1:5,:)
```

```
ans=5×5 table
    ID      EL         Std        VaR        CVaR
    __    _______    ______    ______    ______

    89     2.2647    2.2063    8.2676    8.9997
    96     1.3515    1.6514    6.6157    7.7062
    66    0.90459     1.474    6.4168    7.5149
    22     1.5745    1.8663    6.0121    7.3814
    16     1.6352    1.5288    6.3404    7.3462
```

Plot the counterparty exposures and CVaR contributions. The counterparties with the highest CVaR contributions are plotted in red and orange.

```
figure;
pointSize = 50;
colorVector = rc_sorted.CVaR;
scatter(cc.Portfolio(idx,:).EAD, rc_sorted.CVaR,...
    pointSize,colorVector,'filled')
colormap('jet')
title('CVaR Contribution vs. Exposure')
xlabel('Exposure')
ylabel('CVaR Contribution')
grid on
```



Investigate Simulation Convergence with Confidence Bands

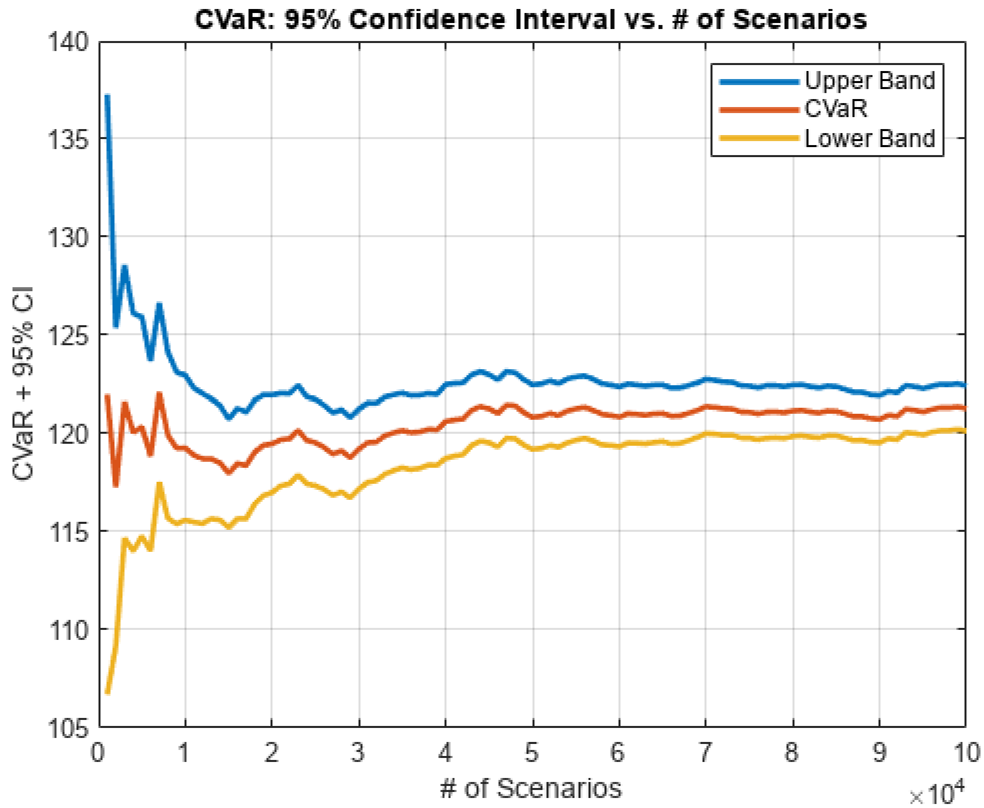
Use the `confidenceBands` function to investigate the convergence of the simulation. By default, the CVaR confidence bands are reported, but confidence bands for all risk measures are supported using the optional `RiskMeasure` argument.

```
cb = confidenceBands(cc);

% The confidence bands are stored in a table.
cb(1:5,:)
```

```
ans=5×4 table
    NumScenarios    Lower      CVaR      Upper
    ____________    ______    ______    ______

        1000         106.7    121.99    137.28
        2000        109.18    117.28    125.38
        3000        114.68    121.63    128.58
        4000        114.02    120.06    126.11
        5000        114.77    120.36    125.94
```

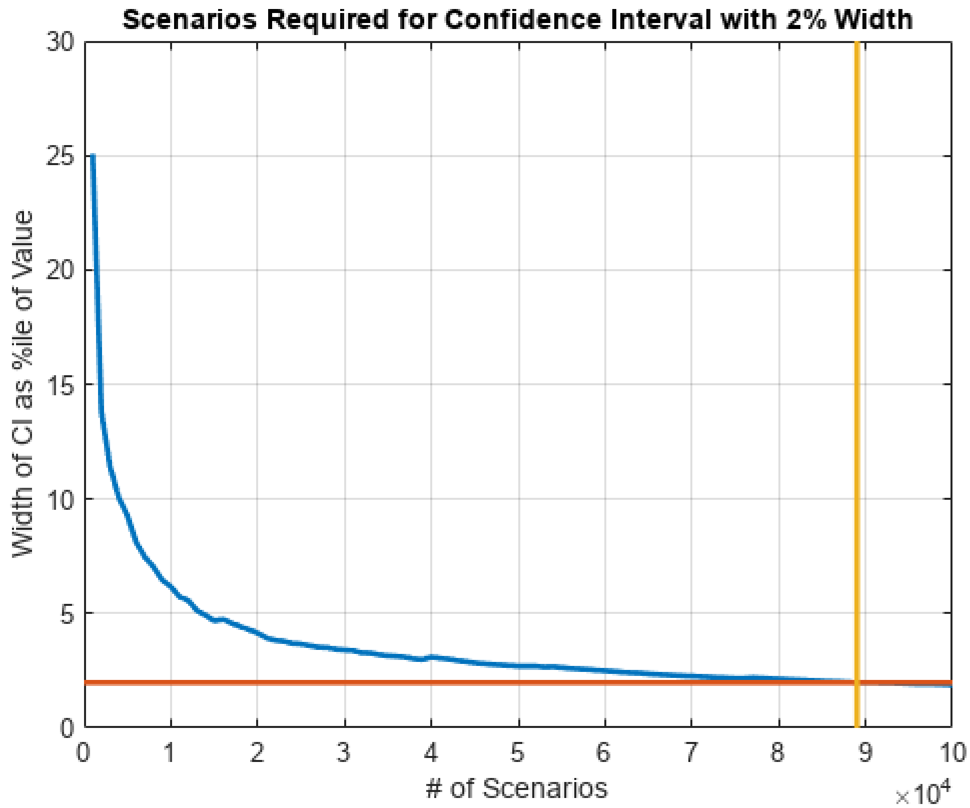
Plot the confidence bands to see how quickly the estimates converge.

```
figure;
plot(...
    cb.NumScenarios,...
    cb{:,{'Upper' 'CVaR' 'Lower'}},...
    'LineWidth',2);

title('CVaR: 95% Confidence Interval vs. # of Scenarios');
xlabel('# of Scenarios');
ylabel('CVaR + 95% CI')
legend('Upper Band','CVaR','Lower Band');
grid on
```



Find the necessary number of scenarios to achieve a particular width of the confidence bands.

```
width = (cb.Upper - cb.Lower) ./ cb.CVaR;

figure;
plot(cb.NumScenarios,width * 100,'LineWidth',2);
title('CVaR: 95% Confidence Interval Width vs. # of Scenarios');
xlabel('# of Scenarios');
ylabel('Width of CI as %ile of Value')
grid on

% Find point at which the confidence bands are within 1% (two sided) of the
% CVaR.
thresh = 0.02;

scenIdx = find(width <= thresh,1,'first');
scenValue = cb.NumScenarios(scenIdx);
widthValue = width(scenIdx);
hold on
plot(xlim,100 * [widthValue widthValue],...
    [scenValue scenValue], ylim,...
    'LineWidth',2);
title('Scenarios Required for Confidence Interval with 2% Width');
```



Compare Tail Risk for Gaussian and *t* Copulas

Switching to a *t* copula increases the default correlation between counterparties. This results in a fatter tail distribution of portfolio losses, and in higher potential losses in stressed scenarios.

Rerun the simulation using a *t* copula and compute the new portfolio risk measures. The default degrees of freedom (dof) for the *t* copula is five.

```
cc_t = simulate(cc,1e5,'Copula','t');
pr_t = portfolioRisk(cc_t);
```

See how the portfolio risk changes with the *t* copula.

```
fprintf('Portfolio risk with Gaussian copula:\n');
```

```
Portfolio risk with Gaussian copula:
```

```
disp(pr)
```

```
      EL        Std       VaR      CVaR 
    ______    ______    _____    ______

    24.876    23.778    102.4    121.28
```

```
fprintf('\n\nPortfolio risk with t copula (dof = 5):\n');
```

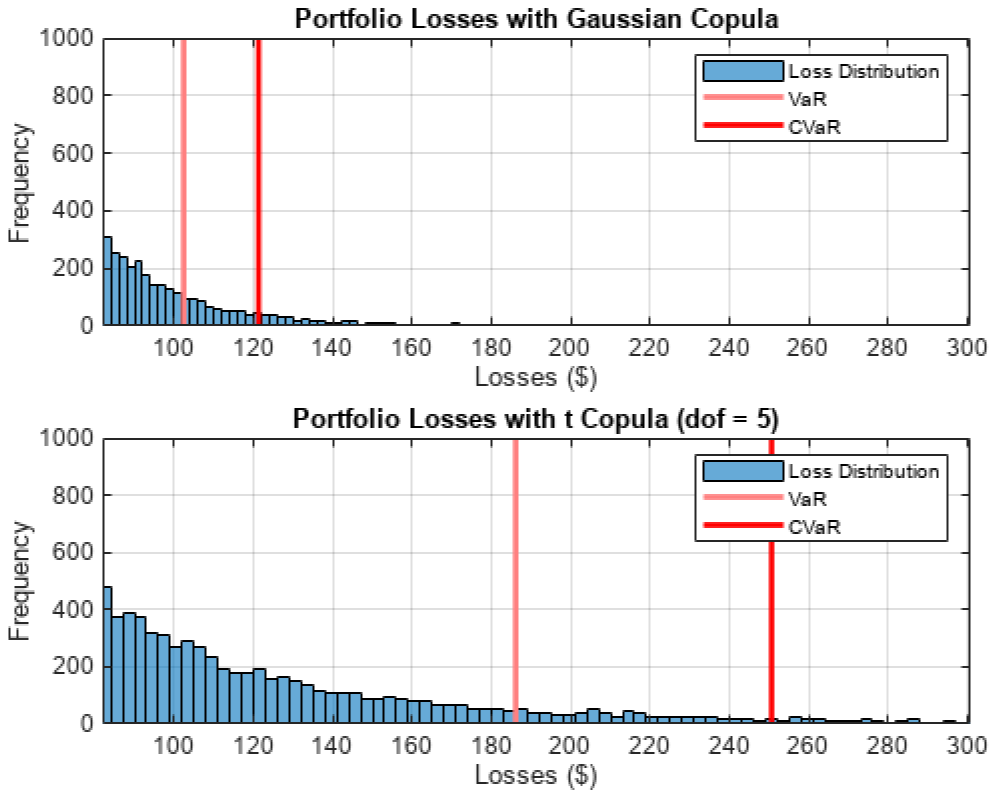
```
Portfolio risk with t copula (dof = 5):
```

```
disp(pr_t)
```

```
      EL        Std        VaR       CVaR
    ______    ______    ______    ______

    24.808    38.749    186.08    250.59
```

Compare the tail losses of each model.

```
% Plot the Gaussian copula tail.
figure;
subplot(2,1,1)
p1 = histogram(cc.PortfolioLosses);
hold on
plotline(pr.VaR,[1 0.5 0.5])
plotline(pr.CVaR,[1 0 0])
xlim([0.8 * pr.VaR  1.2 * pr_t.CVaR]);
ylim([0 1000]);
grid on
legend('Loss Distribution','VaR','CVaR')
title('Portfolio Losses with Gaussian Copula');
xlabel('Losses ($)');
ylabel('Frequency');

% Plot the t copula tail.
subplot(2,1,2)
p2 = histogram(cc_t.PortfolioLosses);
hold on
plotline(pr_t.VaR,[1 0.5 0.5])
plotline(pr_t.CVaR,[1 0 0])
xlim([0.8 * pr.VaR  1.2 * pr_t.CVaR]);
ylim([0 1000]);
grid on
legend('Loss Distribution','VaR','CVaR');
title('Portfolio Losses with t Copula (dof = 5)');
xlabel('Losses ($)');
ylabel('Frequency');
```



The tail risk measures VaR and CVaR are significantly higher using the *t* copula with five degrees of freedom. The default correlations are higher with *t* copulas, therefore there are more scenarios where multiple counterparties default. The number of degrees of freedom plays a significant role. For very high degrees of freedom, the results with the *t* copula are similar to the results with the Gaussian copula. Five is a very low number of degrees of freedom and, consequentially, the results show striking differences. Furthermore, these results highlight that the potential for extreme losses are very sensitive to the choice of copula and the number of degrees of freedom.

See Also

`creditDefaultCopula` | `simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios`

Related Examples

- “Credit Simulation Using Copulas” on page 4-2
- “creditDefaultCopula Simulation Workflow” on page 4-5
- “One-Factor Model Calibration” on page 4-64

More About

- “Risk Modeling with Risk Management Toolbox” on page 1-3

Modeling Probabilities of Default with Cox Proportional Hazards

This example shows how to work with consumer (retail) credit panel data to visualize observed probabilities of default (PDs) at different levels. It also shows how to fit a Cox proportional hazards (PH) model, also known as Cox regression, to predict PDs. In addition, it shows how to perform a stress-testing analysis, how to model lifetime PDs, and how to calculate the lifetime expected credit loss (ECL) value using `portfolioECL`.

This example uses `fitLifetimePDModel` from Risk Management Toolbox™ to fit the `Cox` PH model. Although the same model can be fitted using `fitcox`, the lifetime probability of default (PD) version of the `Cox` model is designed for credit applications, and supports conditional PD prediction, lifetime PD prediction, and model validation tools, including the discrimination and accuracy plots.

A similar example, “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36, follows the same workflow, but it uses a `Logistic` regression model instead of a `Cox` model. The main differences in the two approaches are:

- *Model fit* — The `Cox` PH model has a nonparametric baseline hazard rate that can match patterns in the PDs more closely than the fully parametric `Logistic` model.
- *Extrapolating beyond the observed ages in the data* — The `Cox` PH model, because it is built on top of a nonparametric baseline hazard rate, needs additional rules or assumptions to extrapolate to loan ages that are not observed in the data set. For an example, see “Use Cox Lifetime PD Model to Predict Conditional PD” on page 6-329. Conversely, the `Logistic` model treats the age of the loan as a continuous variable; therefore, a `Logistic` model can seamlessly extrapolate to predict PDs for ages not observed in the data set.

Data Exploration with Survival Analysis Tools

Start with some data visualizations, mainly the visualization of PDs as a function of age, which in this data set is the same as years-on-books (YOB). Because `Cox` PH is a survival analysis model, this example discusses some survival analysis tools and concepts and uses the empirical cumulative distribution function (`ecdf`) functionality for some of these computations and visualizations.

The main data set (`data`) contains the following variables:

- `ID`: Loan identifier.
- `ScoreGroup`: Credit score at the beginning of the loan, discretized into three groups, `High Risk`, `Medium Risk`, and `Low Risk`.
- `YOB`: Years on books.
- `Default`: Default indicator. This is the response variable.
- `Year`: Calendar year.

There is also a small data set (`dataMacro`) with macroeconomic data for the corresponding calendar years that contains the following variables:

- `Year`: Calendar year.
- `GDP`: Gross domestic product growth (year over year).
- `Market`: Market return (year over year).

The variables `YOB`, `Year`, `GDP`, and `Market` are observed at the end of the corresponding calendar year. The `ScoreGroup` is a discretization of the original credit score when the loan started. A value of `1` for `Default` means that the loan defaulted in the corresponding calendar year.

A third data set (`dataMacroStress`) contains baseline, adverse, and severely adverse scenarios for the macroeconomic variables. The stress-testing analysis on page 4-41 in this example uses this table.

Load the simulated data.

```
load RetailCreditPanelData.mat
disp(head(data,10))
```

```
    ID     ScoreGroup     YOB    Default    Year
    __    ___________    ___    _______    ____

    1     Low Risk        1        0       1997
    1     Low Risk        2        0       1998
    1     Low Risk        3        0       1999
    1     Low Risk        4        0       2000
    1     Low Risk        5        0       2001
    1     Low Risk        6        0       2002
    1     Low Risk        7        0       2003
    1     Low Risk        8        0       2004
    2     Medium Risk     1        0       1997
    2     Medium Risk     2        0       1998
```

Preprocess the panel data to put it in the format expected by some of the survival analysis tools.

```
% Use groupsummary to reduce data to one ID per row, and keep track of
% whether the loan defaulted or not.
dataSurvival = groupsummary(data,'ID','sum','Default');
disp(head(dataSurvival,10))
```

```
    ID    GroupCount    sum_Default
    __    __________    ___________

     1        8              0
     2        8              0
     3        8              0
     4        6              0
     5        7              0
     6        7              0
     7        8              0
     8        6              0
     9        7              0
    10        8              0
```

```
% You can also get years observed from YOB, though in this example, the YOB always
% starts from 1 in the data, so the GroupCount equals the final YOB.
dataSurvival.Properties.VariableNames{2} = 'YearsObserved';
dataSurvival.Properties.VariableNames{3} = 'Default';
% If there is no default, it is a censored observation.
dataSurvival.Censored = ~dataSurvival.Default;
disp(head(dataSurvival,10))
```

```
    ID    YearsObserved    Default    Censored
    __    _____________    _______    ________
```

```
     1          8            0          true
     2          8            0          true
     3          8            0          true
     4          6            0          true
     5          7            0          true
     6          7            0          true
     7          8            0          true
     8          6            0          true
     9          7            0          true
    10          8            0          true
```

The main variable is the amount of time each loan was observed (`YearsObserved`), which is the final value of the years-on-books (`YOB`) variable. This years observed is the number of years until default, or until the end of the observation period (eight years), or until the loan is removed from the sample due to prepayment. In this data set, the `YOB` information is the same as the age of the loan because all loans start with a `YOB` of 1. For other data sets, this case might true. For example, in a trading portfolio, the `YOB` and age may be different because a loan purchased in the third year of its life would have an age of `3`, but a `YOB` value of 1.

The second required variable is the censoring variable (`Censored`). In this analysis, the event of interest is the loan default. If a loan is observed until default, you have all of the information about the time until default. Therefore, the lifetime information is uncensored or complete. Alternatively, the information is considered censored, or incomplete, if at the end of the observation period the loan has not defaulted. The loan could not default because it was prepaid or the loan had not defaulted by the end of the eight-year observation period in the sample.

Add the `ScoreGroup` and `Vintage` information to the data. The value of these variables remains constant throughout the life of the loan. The score given at origination determines the `ScoreGroup` and the origination year determines the `Vintage` or cohort.

```
% You can get ScoreGroup from YOB==1 because, in this data set,
% YOB always starts at 1 and the ID's order is the same in data and
% dataSurvival.
dataSurvival.ScoreGroup = data.ScoreGroup(data.YOB==1);
% Define vintages based on the year the loan started. All loans
% in this data set start in year 1 of their life.
dataSurvival.Vintage = data.Year(data.YOB==1);
disp(head(dataSurvival,10))
```

```
    ID    YearsObserved    Default    Censored    ScoreGroup     Vintage
    __    _____________    _______    ________    ___________    _______

     1          8             0         true      Low Risk        1997  
     2          8             0         true      Medium Risk     1997  
     3          8             0         true      Medium Risk     1997  
     4          6             0         true      Medium Risk     1999  
     5          7             0         true      Medium Risk     1998  
     6          7             0         true      Medium Risk     1998  
     7          8             0         true      Medium Risk     1997  
     8          6             0         true      Medium Risk     1999  
     9          7             0         true      Low Risk        1998  
    10          8             0         true      Low Risk        1997  
```

Compare the number of rows in the original data set (in panel data format) and the aggregated data set (in the more traditional survival format).

```
fprintf('Number of rows original data: %d\n',height(data));
```

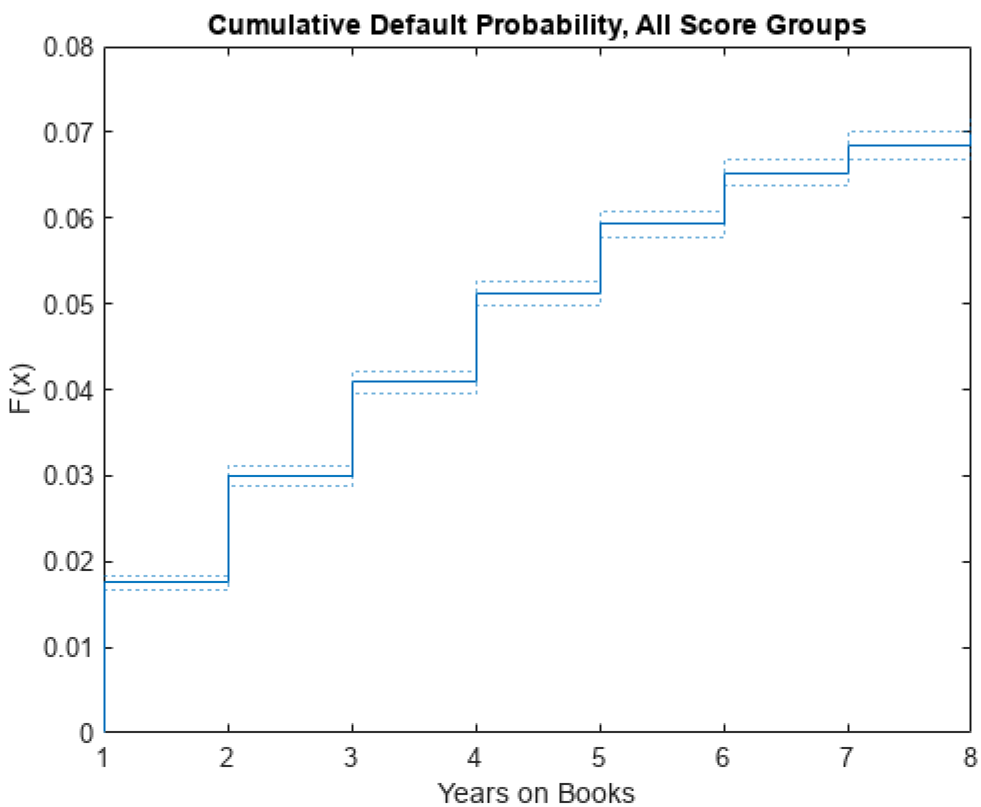
```
Number of rows original data: 646724
```

```
fprintf('Number of rows survival data: %d\n',height(dataSurvival));
```

```
Number of rows survival data: 96820
```

Plot the cumulative default probability against YOB for the entire portfolio (all score groups and vintages) using the empirical cumulative distribution function (`ecdf`).

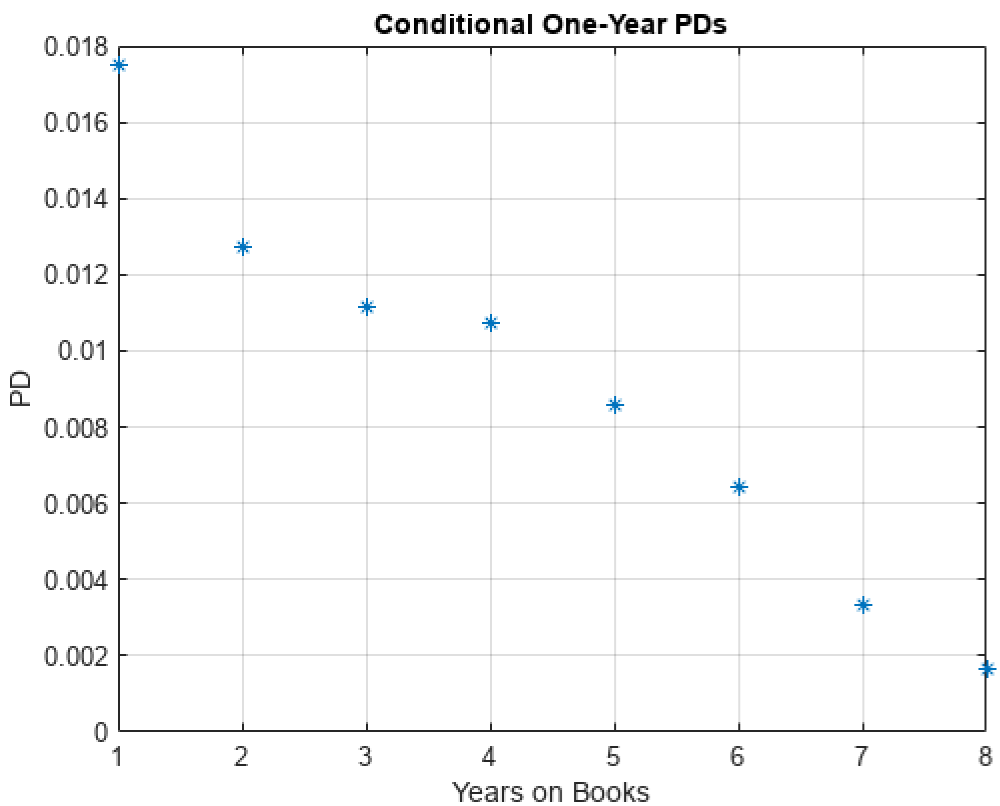
```
ecdf(dataSurvival.YearsObserved,'Censoring',dataSurvival.Censored,'Bounds','on')
title('Cumulative Default Probability, All Score Groups')
xlabel('Years on Books')
```



Plot conditional one-year PDs against YOB. For example, the conditional one-year PD for a `YOB` of `3` is the conditional one-year PD for loans that are in their third year of life. In survival analysis, this value coincides with the discrete hazard rate, denoted by *h*, since the number of defaults in a particular year is the number of "failures," and the number of loans still on books at the beginning of that same year is the same as the "number at risk." To compute *h*, get the cumulative hazard function output, denoted by *H*, and transform it to the hazard function *h*. For more information, see "Kaplan-Meier Method".

```
[H,x] = ecdf(dataSurvival.YearsObserved,'Censoring',dataSurvival.Censored, ...
    'Function','cumulative hazard');
% Take the diff of H to get the hazard h.
h = diff(H);
x(1) = [];
```

```
% In this example, the times observed (stored in variable x) do not change for
% different score groups, or for training vs. test sets. For other data sets,
% you may need to check the x and h variables after every call to the ecdf function before
% plotting or concatenating results. (For example, if data set has no defaults in a
% particular year for the test data.)

plot(x,h,'*')
grid on
title('Conditional One-Year PDs')
ylabel('PD')
xlabel('Years on Books')
```



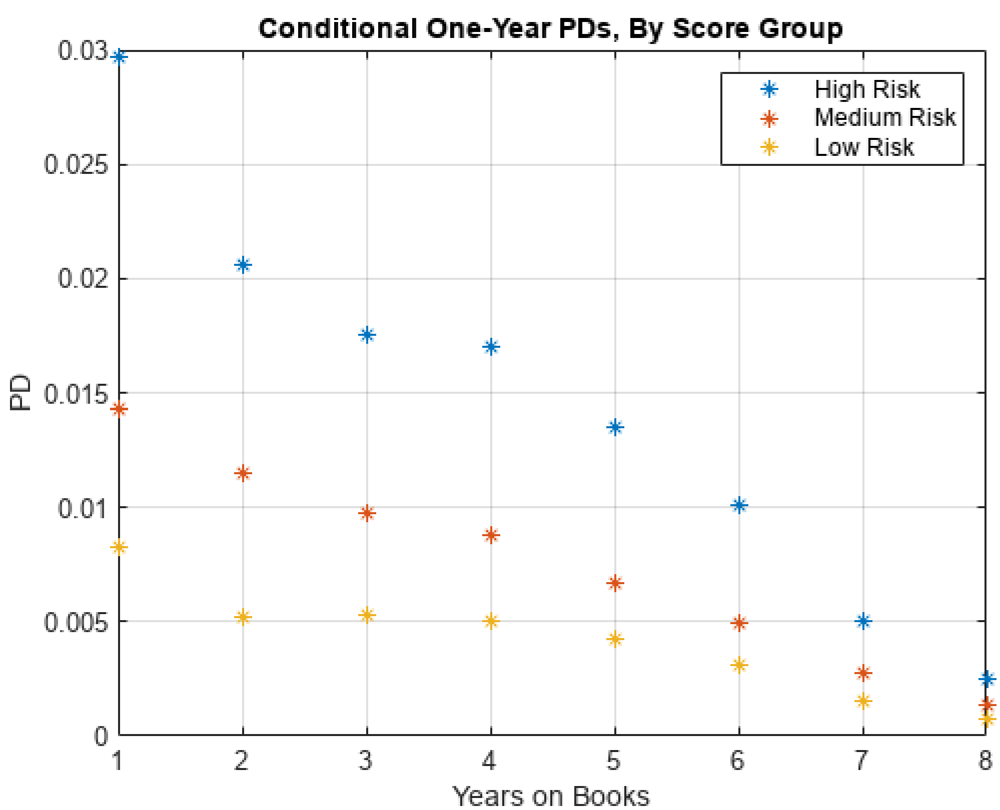
You can also compute these probabilities directly with `groupsummary` using the original panel data format. For more information, see the companion example, “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36. Alternatively, you can compute these probabilities with `grpstats` using the original panel data format. Either of these approaches gives the same conditional one-year PDs.

```
PDvsYOBByGroupsummary = groupsummary(data,'YOB','mean','Default');

PDvsYOBByGrpstats = grpstats(data.Default,data.YOB);

PDvsYOB = table((1:8)',h,PDvsYOBByGroupsummary.mean_Default,PDvsYOBByGrpstats, ...
    'VariableNames',{'YOB','ECDF','Groupsummary','Grpstats'});
disp(PDvsYOB)
```

```
    YOB      ECDF        Groupsummary    Grpstats
    ___    _________    ____________    _________

     1      0.017507      0.017507       0.017507
     2      0.012704      0.012704       0.012704
     3      0.011168      0.011168       0.011168
     4      0.010728      0.010728       0.010728
     5     0.0085949     0.0085949      0.0085949
     6      0.006413      0.006413       0.006413
     7     0.0033231     0.0033231      0.0033231
     8     0.0016272     0.0016272      0.0016272
```

Segment the data by `ScoreGroup` to get the PDs disaggregated by `ScoreGroup`.

```
ScoreGroupLabels = categories(dataSurvival.ScoreGroup);
NumScoreGroups = length(ScoreGroupLabels);
hSG = zeros(length(h),NumScoreGroups);
for ii=1:NumScoreGroups
    Ind = dataSurvival.ScoreGroup==ScoreGroupLabels{ii};
    H = ecdf(dataSurvival.YearsObserved(Ind),'Censoring',dataSurvival.Censored(Ind));
    hSG(:,ii) = diff(H);
end
plot(x,hSG,'*')
grid on
title('Conditional One-Year PDs, By Score Group')
xlabel('Years on Books')
ylabel('PD')
legend(ScoreGroupLabels)
```



You can also disaggregate PDs by `Vintage` information and segment the data in a similar way. You can plot these PDs against `YOB` or against calendar year. To see these visualizations, refer to “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36.

Cox PH Model Without Macro Effects

This section shows how to fit a `Cox` PH model without macro information. The model includes only the time-independent predictor `ScoreGroup` at the origination of the loans. Time-independent predictors contain information that remains constant throughout the life of the loan. This example uses only `ScoreGroup`, but other time-independent predictors could be added to the model (for example, `Vintage` information).

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. The Cox PH model is:

$$h(X_i, t) = h_0(t)\exp\left(\sum_{j=1}^{p} x_{\mathrm{ij}} b_j\right)$$

where:

- $X_i = (x_{\mathrm{i1}}, \ldots, x_{\mathrm{ip}})$ are the predictor variables for the ith subject.
- b_j is the coefficient of the jth predictor variable.
- $h(X_i, t)$ is the hazard rate at time t for X_i.
- $h_0(t)$ is the baseline hazard rate function.

For more details, see the `Cox` and `fitcox` or “Cox Proportional Hazards Model” and the references therein.

The basic `Cox` PH model assumes that the predictor values do not change throughout the life of the loans. In this example, `ScoreGroup` does not change because it is the score given to borrowers at the beginning of the loan. `Vintage` is also constant throughout the life of the loan.

A `Cox` model could use time-dependent scores. For example, if the credit score information is updated every year, you model a time-dependent predictor in the `Cox` model similar to the way the macro variables are added to the model later in the Cox PH Model with Macro Effects on page 4-38 section.

To fit a `Cox` lifetime PD model using `fitLifetimePDModel`, use the original `data` table in panel data format. Although the survival data format in the `dataSurvival` table can be used with other survival functions such as `ecdf` or `fitcox`, the `fitLifetimePDModel` function always works with the panel data format. This simplifies the switch between models with, or without time-dependent models, and the same panel data format is used for the validation functions such as `modelCalibrationPlot`. When fitting Cox models, the `fitLifetimePDModel` function treats the age variable (`'AgeVar'` argument) as the time to event and it uses the response variable (`'ResponseVar'` argument) binary values to identify the censored observations.

In the fitted model that follows, the only predictor is the `ScoreGroup` variable. The `fitLifetimePDModel` function checks the periodicity of the data (the most common age increments) and stores it in the `'TimeInterval'` property of the `Cox` lifetime PD model. The `'TimeInterval'` information is important for the prediction of conditional PD using `predict`.

Split the data into training and testing subsets and then fit the model using the training data.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % For reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));

pdModel = fitLifetimePDModel(data(TrainDataInd,:),'cox', ...
   'IDVar','ID','AgeVar','YOB','LoanVars','ScoreGroup','ResponseVar','Default');
disp(pdModel)
```

```
  Cox with properties:

            TimeInterval: 1
     ExtrapolationFactor: 1
                 ModelID: "Cox"
             Description: ""
         UnderlyingModel: [1x1 CoxModel]
                   IDVar: "ID"
                  AgeVar: "YOB"
                LoanVars: "ScoreGroup"
               MacroVars: ""
             ResponseVar: "Default"
```

```
pdModel.UnderlyingModel
```

```
ans = 
Cox Proportional Hazards regression model

                                Beta          SE        zStat        pValue   
                              ________    ________    _______    ___________

    ScoreGroup_Medium Risk    -0.67831    0.037029    -18.319     5.8806e-75
    ScoreGroup_Low Risk        -1.2453    0.045243    -27.525    8.8419e-167


Log-likelihood: -41783.047
```

To predict the conditional PDs, use `predict`. For example, predict the PD for the first `ID` in the data.

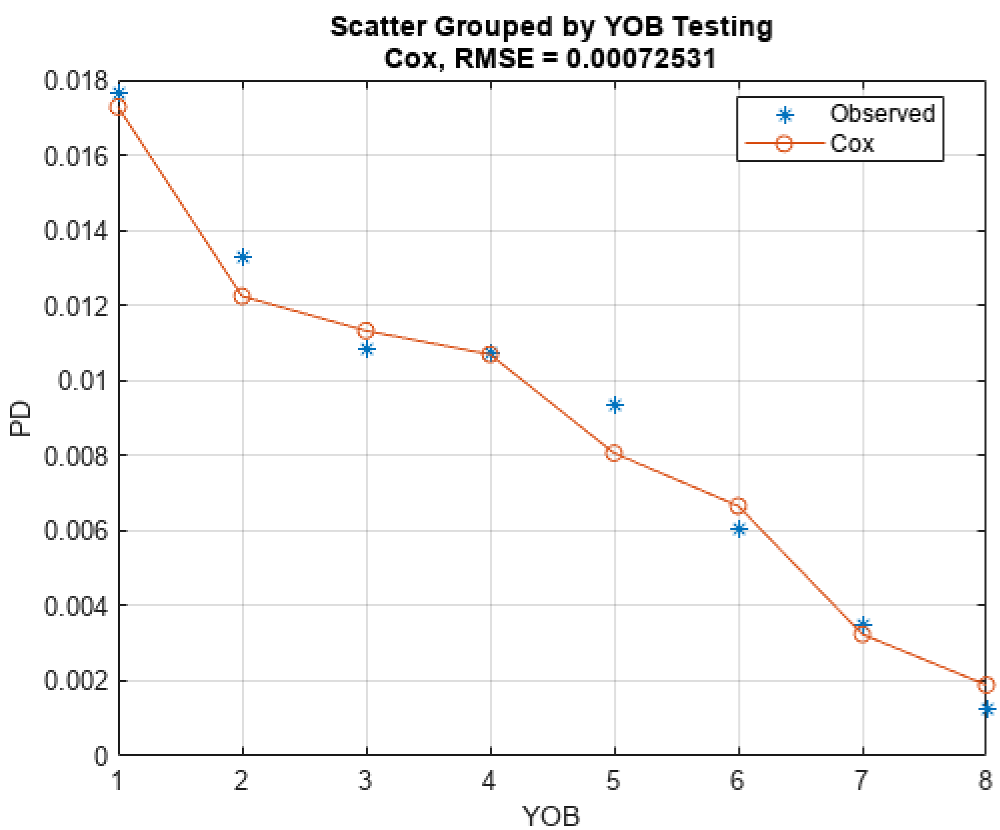
```
PD_ID1 = predict(pdModel,data(1:8,:))
```

```
PD_ID1 = 8×1

    0.0083
    0.0059
    0.0055
    0.0052
```

```
    0.0039
    0.0033
    0.0016
    0.0009
```

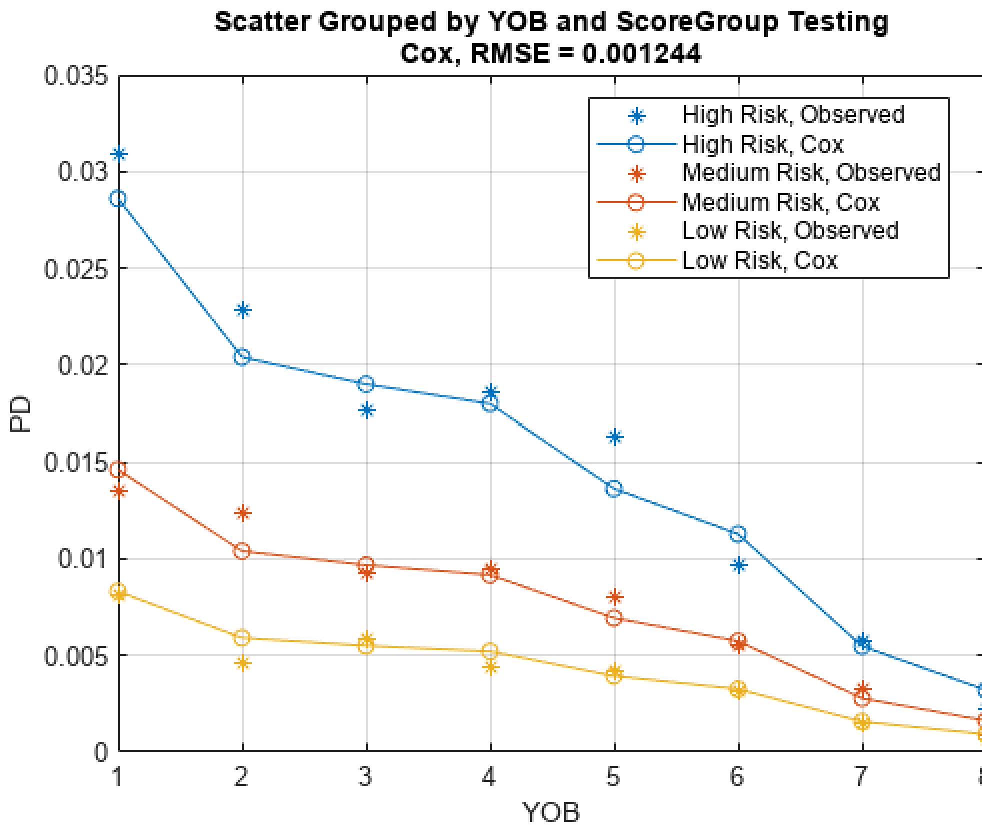
To compare the predicted PDs against the observed default rates in the training or test data, use `modelCalibrationPlot`. This plot is a visualization of the calibration of the predicted PD values (also known as predictive ability). A grouping variable is required for the PD model accuracy. By using `YOB` as the grouping variable, the observed default rates are the same as the default rates discussed in the Data Exploration with Survival Analysis Tools on page 4-28 section.

```
DataSetChoice = Testing ;
if DataSetChoice=="Training"
    Ind = TrainDataInd;
else
    Ind = TestDataInd;
end

modelCalibrationPlot(pdModel,data(Ind,:),'YOB','DataID',DataSetChoice)
```



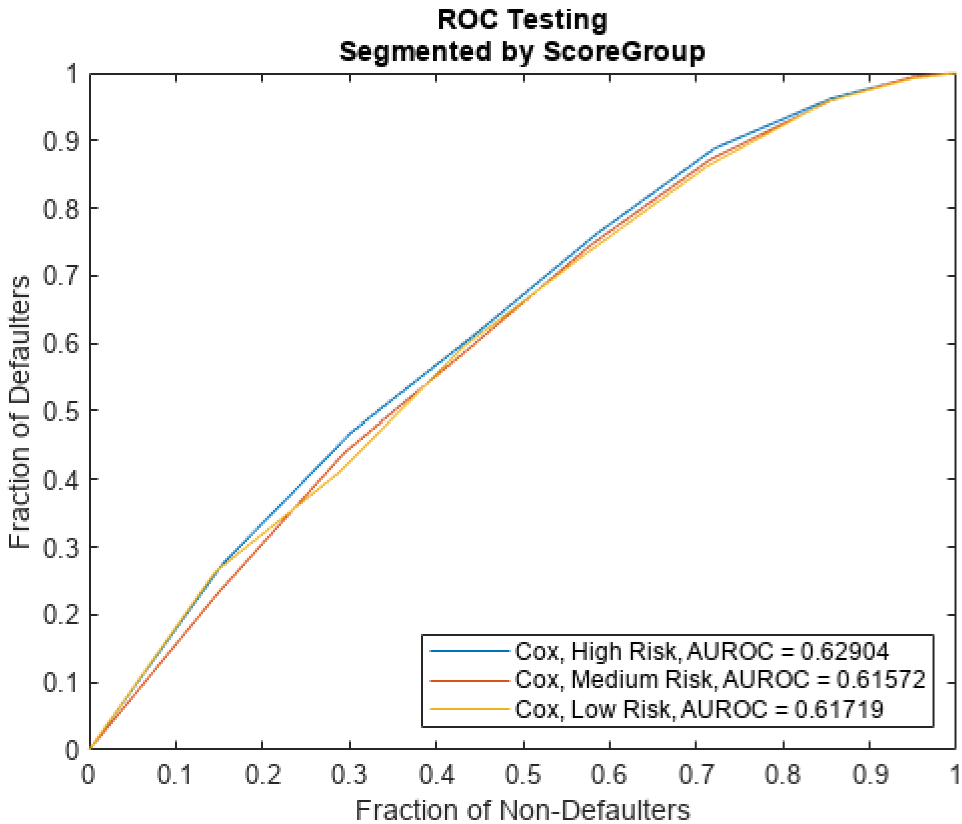
The calibration plot accepts a second grouping variable. For example, use `ScoreGroup` as a second grouping variable to visualize the PD predictions per `ScoreGroup`, against the `YOB`.

```
modelCalibrationPlot(pdModel,data(Ind,:),{'YOB','ScoreGroup'},'DataID',DataSetChoice)
```



The `modelDiscriminationPlot` returns the ROC curve. Use the optional `'SegmentBy'` argument to visualize the ROC for each `ScoreGroup`.

```
modelDiscriminationPlot(pdModel,data(Ind,:),'DataID',DataSetChoice,'SegmentBy','ScoreGroup')
```



The nonparametric part of the Cox model allows it to closely match the training data pattern, even though only `ScoreGroup` is included as a predictor in this model. The results on test data show larger errors than on the training data, but this result is still a good fit.

The addition of macro information is important because both the stress testing and the lifetime PD projections require an explicit dependency on macro information.

Cox PH Model with Macro Effects

This section shows how to fit a `Cox` PH model that includes macro information, specifically, gross domestic product (GDP) growth, and stock market growth. The value of the macro variables changes every year, so the predictors are time dependent.

The extension of the Cox proportional hazards model to account for time-dependent variables is:

$$h(X_i, t) = h_0(t)\exp\left(\sum_{j=1}^{\mathrm{p1}} x_{\mathrm{ij}} b_j + \sum_{k=1}^{\mathrm{p2}} x_{\mathrm{ik}}(t) c_k\right)$$

where:

- x_{ij} is the predictor variable value for the ith subject and the jth time-independent predictor.
- $x_{\mathrm{ik}}(t)$ is the predictor variable value for the ith subject and the kth time-dependent predictor at time t.
- b_j is the coefficient of the jth time-independent predictor variable.

- c_k is the coefficient of the kth time-dependent predictor variable.
- $h(X_i(t), t)$ is the hazard rate at time t for $X_i(t)$.
- $h_0(t)$ is the baseline hazard rate function.

For more details, see `Cox`, `fitcox`, or "Cox Proportional Hazards Model" and the references therein.

Macro variables are treated as time-dependent variables. If the time-independent information, such as the initial `ScoreGroup`, provides a baseline level of risk through the life of the loan, it is reasonable to expect that changing macro conditions may increase or decrease the risk around that baseline level. Also, if the macro conditions change, you can expect that these variations in risk will be different from one year to the next. For example, years with low economic growth should make all loans more risky, independently of their initial `ScoreGroup`.

The data input for the `Cox` lifetime PD model with time-dependent predictors uses the original panel data with the addition of the macro information.

As mentioned earlier, when fitting Cox models, the `fitLifetimePDModel` function treats the age variable (`'AgeVar'` argument) as the time to event and it uses the response variable (`'ResponseVar'` argument) binary values to identify the censored observations. In the fitted model that follows, the predictors are `ScoreGroup`, `GDP`, and `Market`. The `fitLifetimePDModel` checks the periodicity of the data (the most common age increments) and stores it in the `'TimeInterval'` property of the `Cox` lifetime PD model. For time-dependent models, the `'TimeInterval'` value is used to define age intervals for each row where the predictor values are constant. For more information, see "Time Interval for Cox Models" on page 6-551. The `'TimeInterval'` information is also important for the prediction of conditional PD when using `predict`.

Internally, the `fitLifetimePDModel` function uses `fitcox`. Using `fitLifetimePDModel` for credit models offers some advantages over `fitcox`. For example, when you work directly with `fitcox`, you need a survival version of the data for time-independent models and a "counting process" version of the data (similar to the panel data form, but with additional information) is needed for time-dependent models. The `fitLifetimePDModel` function always takes the panel data form as input and peforms the data preprocessing before calling `fitcox`. Also, with the lifetime PD version of the `Cox` model, you have access to credit-specific prediction and validation functionality not directly supported in the underlying Cox model.

```
data = join(data,dataMacro);
head(data)
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

```
pdModelMacro = fitLifetimePDModel(data(TrainDataInd,:),'cox', ...
   'IDVar','ID','AgeVar','YOB','LoanVars','ScoreGroup', ...
   'MacroVars',{'GDP','Market'},'ResponseVar','Default');
disp(pdModelMacro)
```

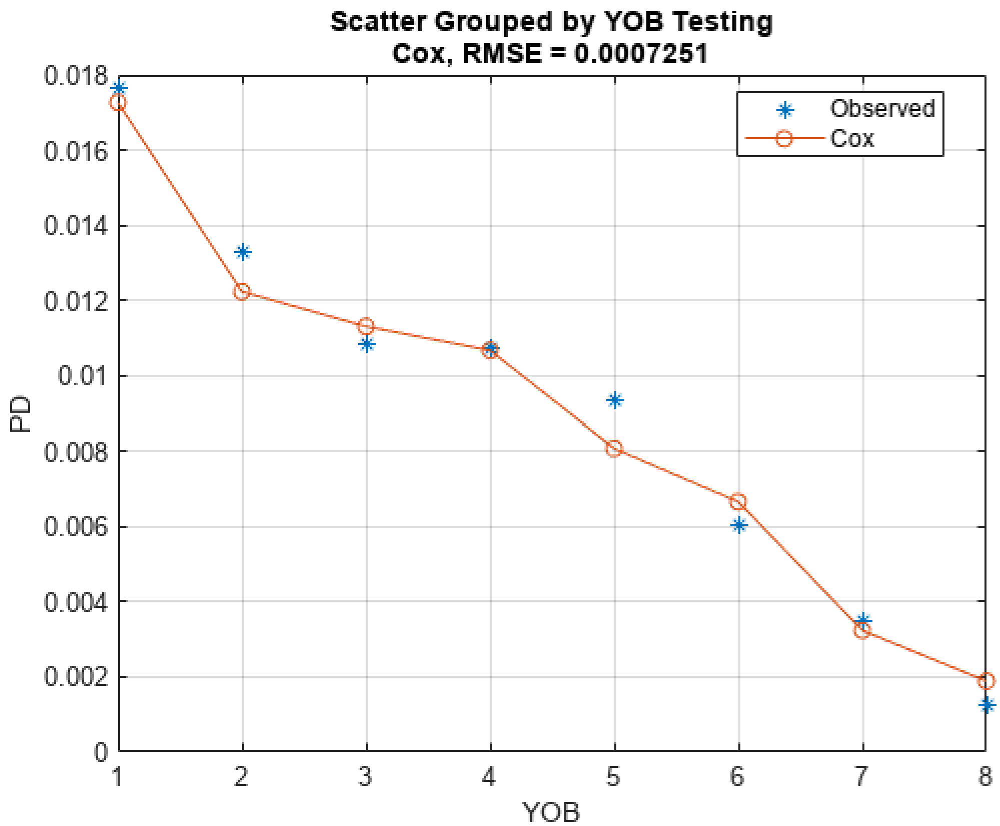
```
  Cox with properties:

            TimeInterval: 1
     ExtrapolationFactor: 1
                 ModelID: "Cox"
             Description: ""
         UnderlyingModel: [1x1 CoxModel]
                   IDVar: "ID"
                  AgeVar: "YOB"
                LoanVars: "ScoreGroup"
               MacroVars: ["GDP"    "Market"]
             ResponseVar: "Default"
```

```
disp(pdModelMacro.UnderlyingModel)
```

```
Cox Proportional Hazards regression model

                               Beta          SE         zStat        pValue   
                            __________    _________    _______    ____________

    ScoreGroup_Medium Risk     -0.6794     0.037029    -18.348      3.4442e-75
    ScoreGroup_Low Risk        -1.2442     0.045244    -27.501     1.7116e-166
    GDP                      -0.084533     0.043687     -1.935        0.052995
    Market                  -0.0084411    0.0032221    -2.6198       0.0087991


Log-likelihood: -41742.871
```

Visualize the model calibration (also known as predictive ability) of the predicted PD values using `modelCalibrationPlot`.

```
DataSetChoice = Testing;
if DataSetChoice=="Training"
    Ind = TrainDataInd;
else
    Ind = TestDataInd;
end

modelCalibrationPlot(pdModelMacro,data(Ind,:),'YOB','DataID',DataSetChoice)
```



The macro effects help the model match the observed default rates even closer and the match to the training data looks like an interpolation for the macro model.

The accuracy plot by `ScoreGroup` and the ROC curve is created the same way as for the Cox model without macro variables.

Stress Testing

This section shows how to perform a stress-testing analysis of PDs using the `Cox` macro model.

Assume that a regulator has provided the following stress scenarios for the macroeconomic variables `GDP` and `Market`.

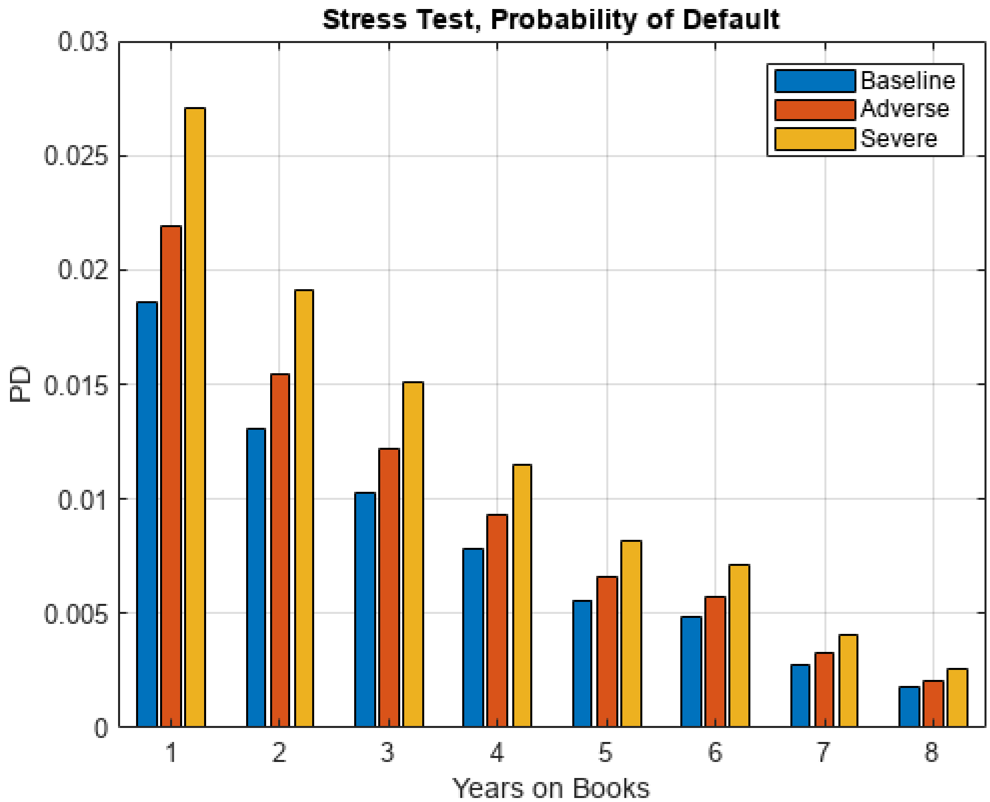
```
disp(dataMacroStress)
```

```
                 GDP     Market
                _____    ______

    Baseline     2.27    15.02
    Adverse      1.31     4.56
    Severe      -0.22    -5.64
```

The following code predicts PDs for each `ScoreGroup` and each macro scenario. For the visualization of each macro scenario, take the average over the `ScoreGroups` to aggregate the data into a single PD by `YOB`.

```
dataStress = table;
dataStress.YOB = repmat((1:8)',3,1);
```

```
dataStress.ScoreGroup = repmat("",size(dataStress.YOB));
dataStress.ScoreGroup(1:8) = ScoreGroupLabels{1};
dataStress.ScoreGroup(9:16) = ScoreGroupLabels{2};
dataStress.ScoreGroup(17:24) = ScoreGroupLabels{3};
dataStress.GDP = zeros(size(dataStress.YOB));
dataStress.Market = zeros(size(dataStress.YOB));

ScenarioLabels = dataMacroStress.Properties.RowNames;
NumScenarios = length(ScenarioLabels);

PDScenarios = zeros(length(x),NumScenarios);

for jj=1:NumScenarios

   Scenario = ScenarioLabels{jj};

   dataStress.GDP(:) = dataMacroStress.GDP(Scenario);
   dataStress.Market(:) = dataMacroStress.Market(Scenario);

   % Predict PD for each ScoreGroup for the current scenario.
   dataStress.PD = predict(pdModelMacro,dataStress);

   % Average PD over ScoreGroups, by age, to visualize in a single plot.
   PDAvgTable = groupsummary(dataStress,"YOB","mean","PD");
   PDScenarios(:,jj) = PDAvgTable.mean_PD;

end

figure;
bar(x,PDScenarios)
title('Stress Test, Probability of Default')
xlabel('Years on Books')
ylabel('PD')
legend('Baseline','Adverse','Severe')
grid on
```



Lifetime PD and ECL

This section shows how to compute lifetime PDs using the `Cox` macro model and how to compute lifetime expected credit losses (ECL).

For lifetime modeling, the PD model is the same, but it is used differently. You need the predicted PDs not just one period ahead, but for each year throughout the life of each particular loan. You also need macro scenarios throughout the life of the loans. This example sets up alternative long-term macro scenarios, computes lifetime PDs under each scenario, and computes the corresponding one-year PDs, marginal PDs, and survival probabilities. The lifetime and marginal PDs are visualized for each year, under each macro scenario. The ECL is then computed for each scenario and the weighted average lifetime ECL.

For concreteness, this example looks into an eight-year loan at the beginning of its third year and predicts the one-year PD from years `3` through `8` of the life of this loan. This example also computes the survival probability over the remaining life of the loan. The relationship between the survival probability $S(t)$ and the one-year conditional PDs or hazard rates $h(t)$, sometimes also called the *forward* PDs, is:

$$S(0) = 1,$$
$$S(1) = (1 - \mathrm{PD}(1)),$$
$$\ldots$$
$$S(t) = S(t-1)(1 - \mathrm{PD}(t)) = (1 - \mathrm{PD}(1))\cdots(1 - \mathrm{PD}(t))$$

The lifetime PD (LPD) is the cumulative PD over the life of the loan, given by the complement of the survival probability:

$$\mathrm{LPD}(t) = 1 - S(t)$$

Another quantity of interest is the marginal PD (MPD), which is the increase in the lifetime PD between two consecutive periods:

$$\mathrm{MPD}(t+1) = \mathrm{LPD}(t+1) - \mathrm{LPD}(t)$$

It follows that the marginal PD is also the decrease in survival probability between consecutive periods, and also the hazard rate multiplied by the survival probability:

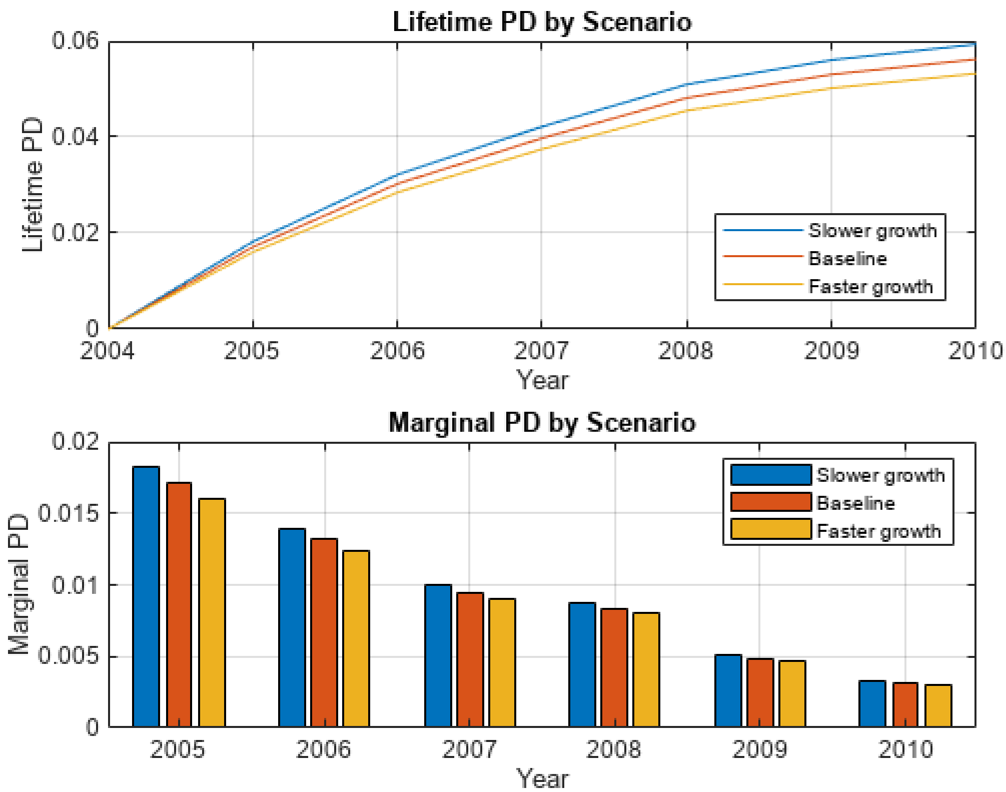
$$\mathrm{MPD}(t+1) = S(t) - S(t+1) = \mathrm{PD}(t+1)S(t)$$

For more information, see `predictLifetime` and “Kaplan-Meier Method”. The `predictLifetime` function supports lifetime PD, marginal PD, and survival probability formats.

Specify three macroeconomic scenarios, one baseline projection, and two simple shifts of 20% higher or 20% lower values for the baseline growth, which are called *faster growth* and *slower growth*, respectively. The scenarios in this example, and the corresponding probabilities, are simple scenarios for illustration purposes only. A more thorough set of scenarios can be constructed with more powerful models using Econometrics Toolbox™ or Statistics and Machine Learning Toolbox™; see, for example, “Model the United States Economy” (Econometrics Toolbox). Automated methods can usually simulate large numbers of scenarios. In practice, only a small number of scenarios are required and these scenarios, and their corresponding probabilities, are selected combining quantitative tools and expert judgment. Also, see the “Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations” on page 4-195 example that shows a detailed workflow for ECL calculations, including the determination of macro scenarios.

```
CurrentAge = 3; % Currently starting third year of loan
Maturity = 8; % Loan ends at end of year 8
YOBLifetime = (CurrentAge:Maturity)';
NumYearsRemaining = length(YOBLifetime);

dataLifetime = table;
dataLifetime.ID = ones(NumYearsRemaining,1);
dataLifetime.YOB = YOBLifetime;
dataLifetime.ScoreGroup = repmat("High Risk",size(dataLifetime.YOB)); % High risk
dataLifetime.GDP = zeros(size(dataLifetime.YOB));
dataLifetime.Market = zeros(size(dataLifetime.YOB));

% Macro scenarios for lifetime analysis
GDPPredict = [2.3; 2.2; 2.1; 2.0; 1.9; 1.8];
GDPPredict = [0.8*GDPPredict GDPPredict 1.2*GDPPredict];

MarketPredict = [15; 13; 11; 9; 7; 5];
MarketPredict = [0.8*MarketPredict MarketPredict 1.2*MarketPredict];

ScenLabels = ["Slower growth" "Baseline" "Faster growth"];
NumMacroScen = size(GDPPredict,2);

% Scenario probabilities for the computation of lifetime ECL
PScenario = [0.2; 0.5; 0.3];

PDLifetime = zeros(size(GDPPredict));
```

```
PDMarginal = zeros(size(GDPPredict));
for ii = 1:NumMacroScen
    dataLifetime.GDP = GDPPredict(:,ii);
    dataLifetime.Market = MarketPredict(:,ii);
    PDLifetime(:,ii) = predictLifetime(pdModelMacro,dataLifetime); % Returns lifetime PD by defau
    PDMarginal(:,ii) = predictLifetime(pdModelMacro,dataLifetime,'ProbabilityType','marginal');
end

% Start lifetime PD at last year with value of 0 for visualization
% purposes.
tLifetime0 = (dataMacro.Year(end):dataMacro.Year(end)+NumYearsRemaining)';
PDLifetime = [zeros(1,NumMacroScen);PDLifetime];
tLifetime = tLifetime0(2:end);

figure;
subplot(2,1,1)
plot(tLifetime0,PDLifetime)
xticks(tLifetime0)
grid on
xlabel('Year')
ylabel('Lifetime PD')
title('Lifetime PD by Scenario')
legend(ScenLabels,'Location','best')
subplot(2,1,2)
bar(tLifetime,PDMarginal)
grid on
xlabel('Year')
ylabel('Marginal PD')
title('Marginal PD by Scenario')
legend(ScenLabels)
```



These lifetime PDs, by scenario, are one of the inputs for the computation of lifetime expected credit losses (ECL). ECL also requires lifetime values for loss given default (LGD) and exposure at default (EAD), for each scenario, and the scenario probabilities. For simplicity, this example assumes a constant LGD and EAD value, but these parameters for LGD and EAD models could vary by scenario and by time period. For more information, see `fitLGDModel` and `fitEADModel`. Use `portfolioECL` to compute the lifetime ECL.

The computation of lifetime ECL also requires the effective interest rate (EIR) for discounting purposes. In this example, the discount factors are computed at the end of the time periods, but other discount times may be used. For example, you might use the midpoint in between the time periods; that is, discount first-year amounts with a 6-month discount factor, discount second-year amounts with a 1.5-year discount factor, and so on).

With these inputs, the expected credit loss at time t for scenario s is defined as:

$$\mathrm{ECL}(t;s) = \mathrm{MPD}(t;s)\,\mathrm{LGD}(t;s)\,\mathrm{EAD}(t;s)\,\mathrm{Disc}(t)$$

where t denotes a time period, s denotes a scenario, and $\mathrm{Disc}(t) = \frac{1}{(1+\mathrm{EIR})^t}$.

For each scenario, a lifetime ECL is computed by adding ECLs across time, from the fist time period in the analysis, to the expected life of the product denoted by T. In this example, it is five years (this loan is a simple loan with five years remaining to maturity):

$$\mathrm{ECL}(s) = \sum_{t=1}^{T} \mathrm{ECL}(t;s)$$

Finally, compute the weighed average of these expected credit losses, across all scenarios, to get a single lifetime ECL value, where $P(s)$ denotes the scenario probabilities:

$$\mathrm{ECL} = \sum_{s=1}^{\mathrm{NumScenarios}} \mathrm{ECL}(s)P(s)$$

These computations are supported with the `portfolioECL` function.

```
LGD = 0.55; % Loss given default
EAD = 100; % Exposure at default
EIR = 0.045; % Effective interest rate

PDMarginalTable = table(dataLifetime.ID, PDMarginal(:,1), PDMarginal(:,2), PDMarginal(:,3),'Varia
LGDTable = table(dataLifetime.ID(1), LGD,'VariableNames',["ID","LGD"]);
EADTable = table(dataLifetime.ID(1), EAD,'VariableNames',["ID","EAD"]);

[totalECL, ECLByID, ECLByPeriod] = portfolioECL(PDMarginalTable, LGDTable, EADTable, 'InterestRat
    'ScenarioNames',ScenLabels, 'ScenarioProbabilities',PScenario, 'IDVar','ID','Periodicity','an
disp(ECLByID);
```

```
    ID     ECL  
    __    ______

    1     2.7441
```

```
disp(ECLByPeriod);
```

```
    ID    TimePeriod    Slower growth    Baseline    Faster growth
    __    __________    _____________    ________    _____________

    1         1            0.95927       0.90012          0.8446  
    1         2              0.703       0.66366         0.62646  
    1         3            0.48217       0.45781         0.43463  
    1         4            0.40518       0.38686         0.36931  
    1         5            0.22384       0.21488         0.20624  
    1         6            0.13866       0.13381          0.1291  
```

```
fprintf('Lifetime ECL: %g\n',totalECL)
```

```
Lifetime ECL: 2.7441
```

When the LGD and EAD do not depend on the scenarios (even if they change with time), the weighted average of the lifetime PD curves is taken to get a single, average lifetime PD curve.

```
MarginalPDLifetimeWeightedAvg = PDMarginal*PScenario;
MarginalPDLifetimeWeightedAvgTable = table(dataLifetime.ID, MarginalPDLifetimeWeightedAvg,'Varial
totalECLByWeightedPD = portfolioECL(MarginalPDLifetimeWeightedAvgTable, LGDTable, EADTable, 'Inte
    'IDVar','ID','Periodicity','annual');
fprintf('Lifetime ECL, using weighted lifetime PD: %g, same result because of constant LGD and EA
    totalECLByWeightedPD)
```

```
Lifetime ECL, using weighted lifetime PD: 2.7441, same result because of constant LGD and EAD.
```

However, when the LGD and EAD values change with the scenarios, pass scenario-dependent inputs (the `PDMarginalTable` input) to the `portfolioECL` function to first compute the ECL values at scenario level. Then you can find the weighted average of the ECL values. For example, see “Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations” on page 4-195

where all inputs (marginal PD, LGD and EAD) change period-by-period and are sensitive to the macroeconomic scenarios.

Conclusion

This example showed how to fit a Cox model for PDs, how to perform stress testing of the PDs, and how to compute lifetime PDs and ECL. A similar example, "Stress Testing of Consumer Credit Default Probabilities Using Panel Data" on page 3-36, follows the same workflow but uses logistic regression, instead of Cox regression. The `fitLifetimePDModel` function supports `Cox`, `Logistic`, and `Probit` models. The computation of lifetime PDs and ECL at the end of this example can also be performed with logistic or probit models. For an example, see "Expected Credit Loss Computation" on page 4-124.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Federal Reserve, Comprehensive Capital Analysis and Review (CCAR): https://www.federalreserve.gov/bankinforeg/ccar.htm

[4] Bank of England, Stress Testing: https://www.bankofengland.co.uk/financial-stability

[5] European Banking Authority, EU-Wide Stress Testing: https://www.eba.europa.eu/risk-analysis-and-data/eu-wide-stress-testing

See Also

`fitLifetimePDModel` | `predict` | `predictLifetime` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot` | `Logistic` | `Probit` | `Cox`

Related Examples

- "Basic Lifetime PD Model Validation" on page 4-129
- "Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
- "Compare Lifetime PD Models Using Cross-Validation" on page 4-121
- "Expected Credit Loss Computation" on page 4-124
- "Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144

More About

- "Overview of Lifetime Probability of Default Models" on page 1-25

Analyze the Sensitivity of Concentration to a Given Exposure

This example shows how to sweep through a range of values for an existing exposure from `0` to double the current value and plot the corresponding values. This could be used as one criterion (among others) for assessing portfolio limits.

Load credit portfolio data and use exposure at default (`EAD`) as the portfolio values. Compute current values of concentration indices.

```
load CreditPortfolioData.mat
P = EAD;
CurrentConcentration = concentrationIndices(P)
```

```
CurrentConcentration=1×8 table
         ID              CR           Deciles        Gini          HH           HK           HT           TI
    ___________    ________    ____________    _______    ________    ________    ________    ____

    "Portfolio"    0.058745    1x11 double     0.55751    0.023919    0.013363    0.022599    0.53
```

Choose an index of interest. For instance, select a loan with maximum exposure.

```
[~,IndMax] = max(P);
CurrentExposure = P(IndMax);
```

Sweep through a range of multipliers for the selected exposure and get the corresponding concentration measures.

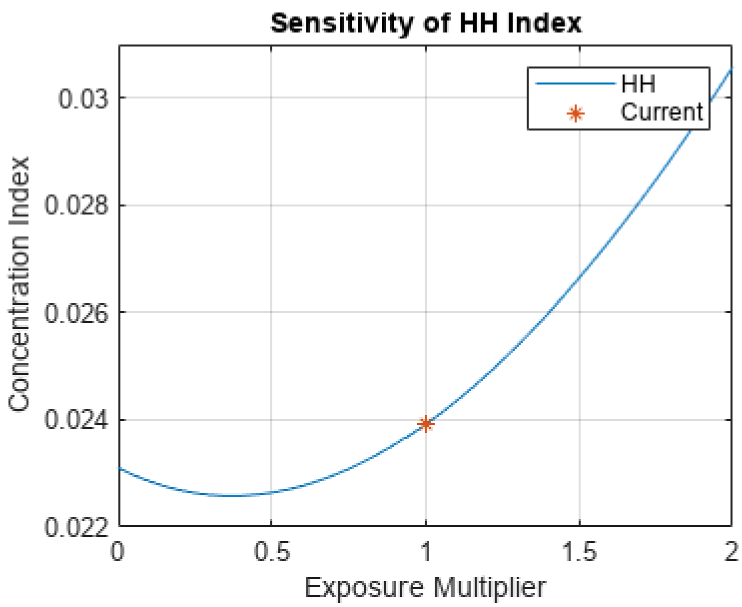
```
Multiplier = 0.0:0.05:2;
% Compute concentration with selected exposure removed from portfolio
P(IndMax) = 0;
ciSensitivity = concentrationIndices(P,'ID','Multiplier 0.0');
ciSensitivity = repmat(ciSensitivity,length(Multiplier),1);
for ii=2:length(Multiplier)
   P(IndMax) = CurrentExposure*Multiplier(ii);
   ci = concentrationIndices(P,'ID',['Multiplier ' num2str(Multiplier(ii))]);
   ciSensitivity(ii,:) = ci;
end
% Display first five rows
disp(ciSensitivity(1:5,:))
```

```
            ID                CR           Deciles        Gini          HH           HK           HT
    _________________    ________    ____________    _______    ________    ________    ________

    "Multiplier 0.0"     0.059442    1x11 double     0.55051    0.023102    0.013314    0.022248
    "Multiplier 0.05"    0.059257    1x11 double      0.5467    0.022968    0.013185    0.022061
    "Multiplier 0.1"     0.059074    1x11 double     0.54456    0.022855    0.013156    0.021957
    "Multiplier 0.15"    0.058891    1x11 double     0.54355    0.022762    0.013143    0.021908
    "Multiplier 0.2"     0.058709    1x11 double     0.54313    0.022688    0.013139    0.021888
```

Plot the sensitivity to changes in exposure for a particular index.

```
IndexID = 'HH';
figure;
plot(Multiplier',ciSensitivity.(IndexID))
hold on
```

```
plot(1,CurrentConcentration.(IndexID),'*')
hold off
title(['Sensitivity of ' IndexID ' Index'])
xlabel('Exposure Multiplier')
ylabel('Concentration Index')
legend(IndexID,'Current')
grid on
```



See Also

`concentrationIndices`

Related Examples

- "Compare Concentration Indices for Random Portfolios" on page 4-51

More About

- "Concentration Indices" on page 1-15

Compare Concentration Indices for Random Portfolios

This example shows how to simulate random portfolios with different distributions and compare their concentration indices. For illustration purposes, a lognormal and Weibull distribution are used. The distribution parameters are chosen arbitrarily to get a similar range of values for both random portfolios.

Generate random portfolios with different distributions.

```
rng('default'); % for reproducibility
PLgn = lognrnd(1,1,1,300);
PWbl = wblrnd(2,0.5,1,300);
```

Display largest simulated loan value.

```
fprintf('\nLargest loan Lognormal: %g\n',max(PLgn));
```

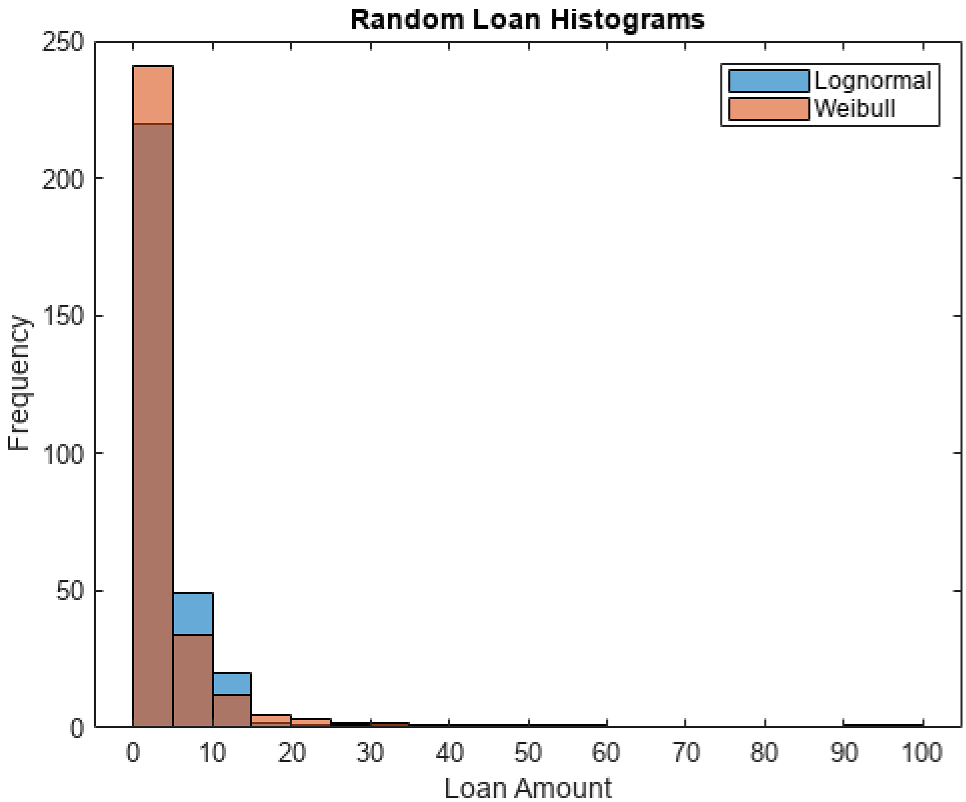
```
Largest loan Lognormal: 97.3582
```

```
fprintf('Largest loan Weibull: %g\n',max(PWbl));
```

```
Largest loan Weibull: 91.5866
```

Plot the portfolio histograms.

```
figure;
histogram(PLgn,0:5:100)
hold on
histogram(PWbl,0:5:100)
hold off
title('Random Loan Histograms')
xlabel('Loan Amount')
ylabel('Frequency')
legend('Lognormal','Weibull')
```

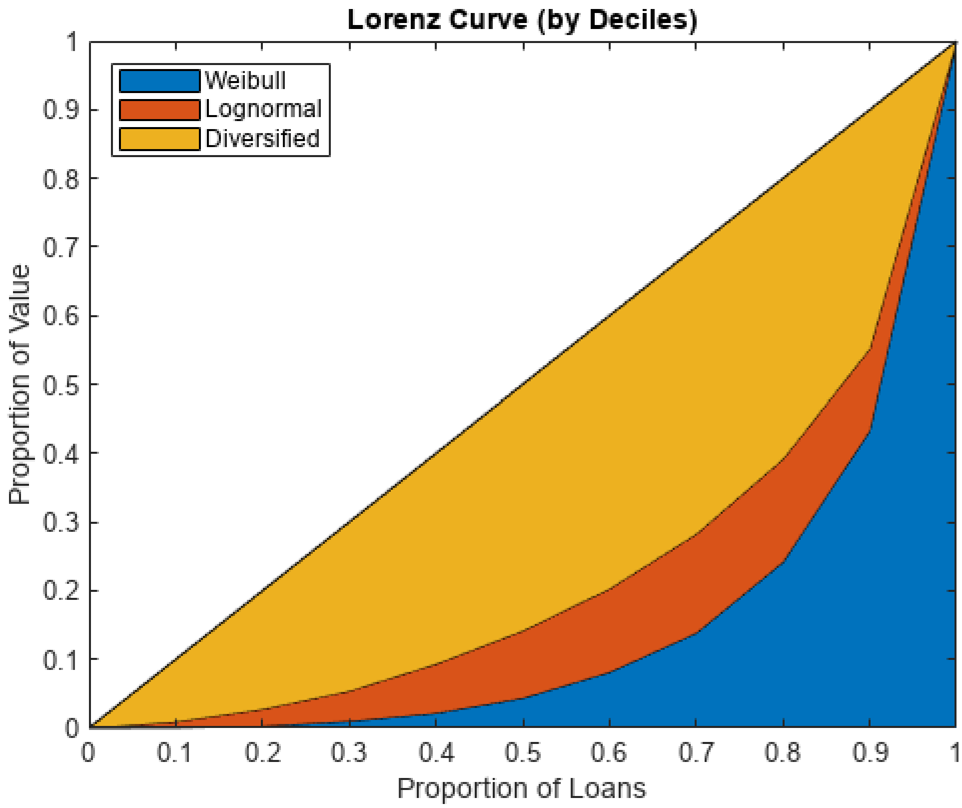


Compute and display the concentration measures.

```
ciLgn = concentrationIndices(PLgn,'ID','Lognormal');
ciWbl = concentrationIndices(PWbl,'ID','Weibull');
disp([ciLgn;ciWbl])
```

```
        ID            CR          Deciles        Gini        HH          HK           HT
    ___________    ________    ____________    ______    ________    _________    _________    _

    "Lognormal"    0.066363    1x11 double     0.5686    0.013298    0.0045765    0.0077267    0
    "Weibull"      0.090152    1x11 double    0.72876    0.020197    0.0062594     0.012289    1
```

```
ProportionLoans = 0:0.1:1;
figure;
area(ProportionLoans',[ciWbl.Deciles; ciLgn.Deciles-ciWbl.Deciles; ProportionLoans-ciLgn.Deciles
axis([0 1 0 1])
legend('Weibull','Lognormal','Diversified','Location','NorthWest')
title('Lorenz Curve (by Deciles)')
xlabel('Proportion of Loans')
ylabel('Proportion of Value')
```



See Also

`concentrationIndices`

Related Examples

- "Analyze the Sensitivity of Concentration to a Given Exposure" on page 4-49

More About

- "Concentration Indices" on page 1-15

Comparison of the Merton Model Single-Point Approach to the Time-Series Approach

This example shows how to compare the Merton model approach, where equity volatility is provided, to the time series approach.

Load the data from `MertonData.mat`.

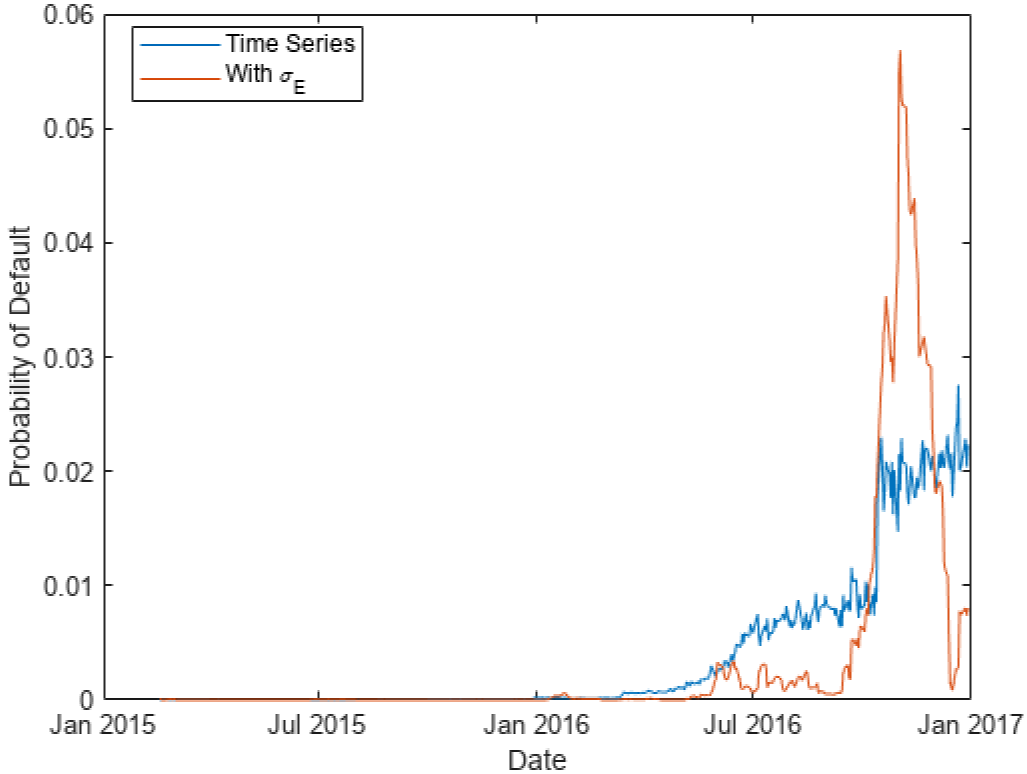
```
load MertonData.mat
Dates     = MertonDataTS.Dates;
Equity    = MertonDataTS.Equity;
Liability = MertonDataTS.Liability;
Rate      = MertonDataTS.Rate;
```

For a given data point in the returns, the corresponding equity volatility is computed from the last preceding 30 days.

```
Returns     = tick2ret(Equity);
DateReturns = Dates(2:end);
SampleSize  = length(Returns);

EstimationWindowSize = 30;
TestWindowStart      = EstimationWindowSize+1;
TestWindow           = (TestWindowStart : SampleSize)';

EquityVol = zeros(length(TestWindow),1);

for i = 1 : length(TestWindow)
    t = TestWindow(i);
    EstimationWindow = t-EstimationWindowSize:t-1;
    EquityVol(i) = sqrt(250)*std(Returns(EstimationWindow));
end
```

Compare the probabilities of default and the estimated asset and asset volatility values using the test window only.

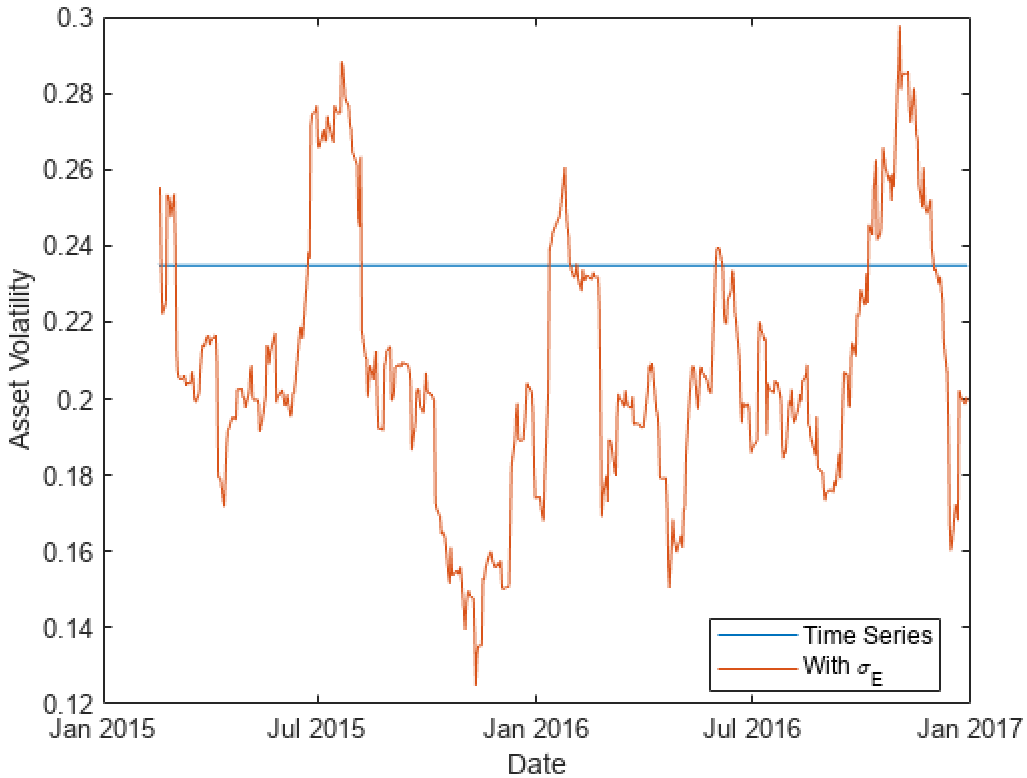
```
[PDTS,DDTS,ATS,SaTS] = mertonByTimeSeries(Equity(TestWindow),Liability(TestWindow),Rate(TestWind

[PDh,DDh,Ah,Sah] = mertonmodel(Equity(TestWindow),EquityVol,Liability(TestWindow),Rate(TestWindo

figure
plot(Dates(TestWindow),PDTS,Dates(TestWindow),PDh)
xlabel('Date')
ylabel('Probability of Default')
legend({'Time Series','With \sigma_E'},'Location','best')
```



The probabilities of default are essentially zero up to early 2016. At that point, both models start predicting positive default probabilities, but we observe some differences between the two models.

Both models calibrate asset values and asset volatilities. The asset values for both approaches match. However, the time-series method, by design, computes a single asset volatility for the entire time window, and the single-point version of the Merton model computes one volatility for each time period, as shown in the following figure.

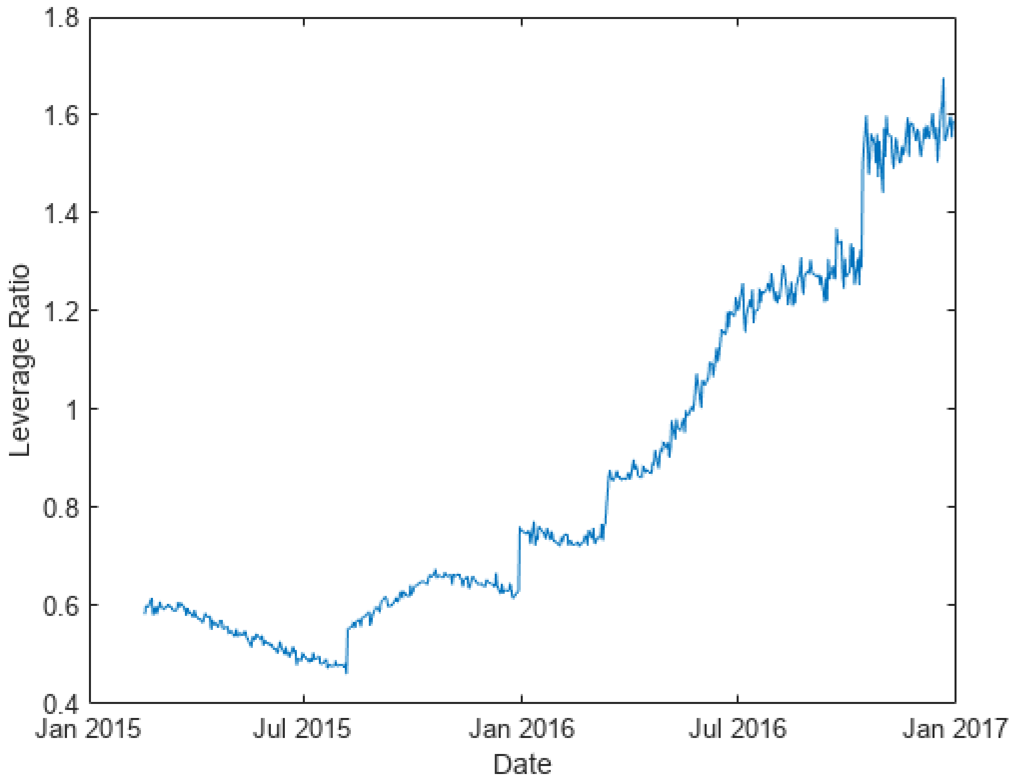
```
figure
plot(Dates(TestWindow),SaTS*ones(size(TestWindow)),Dates(TestWindow),Sah)
xlabel('Date')
ylabel('Asset Volatility')
legend({'Time Series','With \sigma_E'},'Location','best')
```



Towards the end of the time window, the single-point probability of default is above the time-series probability of default when the single-point asset volatility is also above the time-series probability (and vice versa). However, before 2016 the volatility has no effect on the default probability. This means other factors must influence the sensitivity of the default probability to the asset volatility and the overall default probability level.

The firm's *leverage ratio*, defined as the ratio of liabilities to equity, is a key factor in understanding the default probability values in this example. Earlier in the time window, the leverage ratio is low. However, in the second half of the time window, the leverage ratio grows significantly as shown in the following figure.

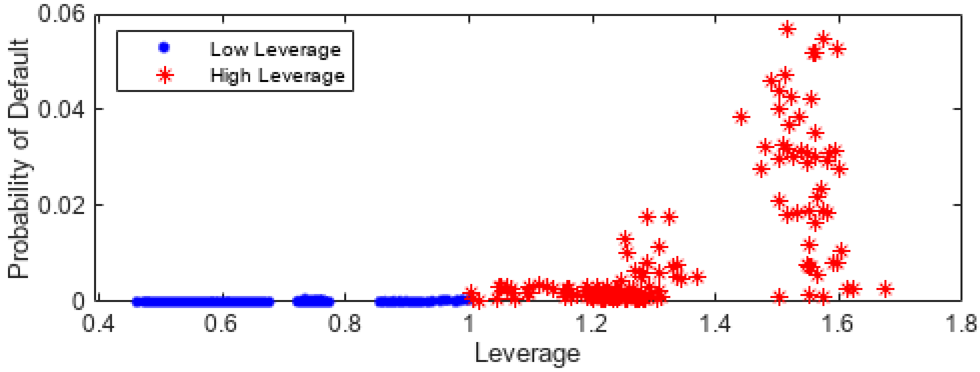
```
Leverage = Liability(TestWindow)./Equity(TestWindow);

figure
plot(Dates(TestWindow),Leverage)
xlabel('Date')
ylabel('Leverage Ratio')
```

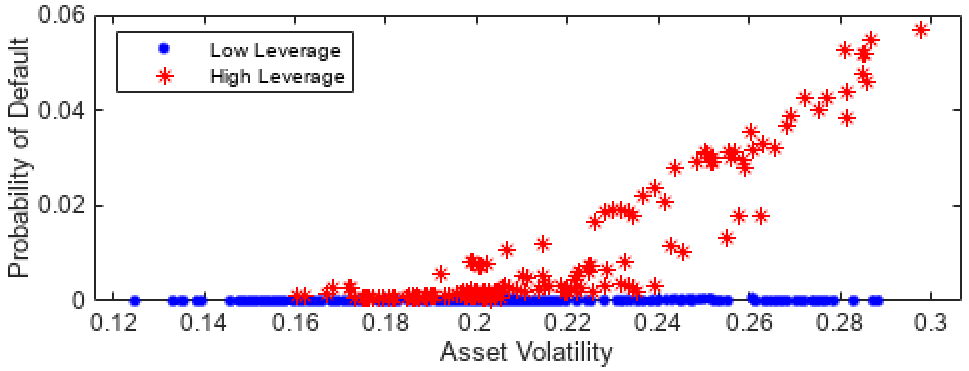


The following plot shows the default probability against the asset volatility for low and high leverage ratios. The leverage ratio is used to divide the points into two groups, depending on whether the leverage ratio is greater or smaller than a cut off value. In this example, a cut off value of `1` works well.

For low leverage, the probability of default is essentially zero, independently of the asset volatilities. For high leverage situations, such as the end of the time window, the probability of default is highly correlated with the asset volatility.

```
figure
subplot(2,1,1)
gscatter(Leverage,PDh,Leverage>1,'br','.*')
xlabel('Leverage')
ylabel('Probability of Default')
legend('Low Leverage','High Leverage','Location','northwest')
subplot(2,1,2)
gscatter(Sah,PDh,Leverage>1,'br','.*')
xlabel('Asset Volatility')
ylabel('Probability of Default')
legend('Low Leverage','High Leverage','Location','northwest')
```





See Also

`mertonmodel` | `mertonByTimeSeries`

More About

- "Default Probability by Using the Merton Model for Structural Credit Risk" on page 1-13

Calculating Regulatory Capital with the ASRF Model

This example shows how to calculate capital requirements and value-at-risk (VaR) for a credit sensitive portfolio of exposures using the asymptotic single risk factor (ASRF) model. This example also shows how to compute Basel capital requirements using an ASRF model.

The ASRF Model

The ASRF model defines capital as the credit value at risk (VaR) in excess of the expected loss (EL).

$$capital = VaR - EL$$

where the EL for a given counterparty is the exposure at default (EAD) multiplied by the probability of default (PD) and the loss given default (LGD).

$$EL = EAD * PD * LGD$$

To compute the credit VaR, the ASRF model assumes that obligor credit quality is modeled with a latent variable (A) using a one factor model where the single common factor (Z) represents systemic credit risk in the market.

$$A_i = \sqrt{\rho_i} \cdot Z + \sqrt{1 - \rho_i} \cdot \epsilon$$

Under this model, default losses for a particular scenario are calculated as:

$$L = EAD \cdot I \cdot LGD$$

where I is the default indicator, and has a value of 1 if $A_i < \Phi_A^{-1}(PD_i)$ (meaning the latent variable has fallen below the threshold for default), and a value of 0 otherwise. The expected value of the default indicator conditional on the common factor is given by:

$$E(I_i|Z) = \Phi_\epsilon(\frac{\Phi_A^{-1}(PD_i) - \sqrt{\rho_i}Z}{\sqrt{1 - \rho_i}})$$

For well diversified and perfectly granular portfolios, the expected loss conditional on a value of the common factor is:

$$L|Z = \sum_i EAD_i \cdot LGD_i \cdot \Phi_\epsilon(\frac{\Phi_A^{-1}(PD_i) - \sqrt{\rho_i}Z}{\sqrt{1 - \rho_i}})$$

You can then directly compute particular percentiles of the distribution of losses using the cumulative distribution function of the common factor. This is the credit VaR, which we compute at the α confidence level:

$$creditVaR(\alpha) = \sum_i EAD_i \cdot LGD_i \cdot \Phi_\epsilon(\frac{\Phi_A^{-1}(PD_i) - \sqrt{\rho_i}\Phi_Z^{-1}(1 - \alpha)}{\sqrt{1 - \rho_i}})$$

It follows that the capital for a given level of confidence, α, is:

$$capital(\alpha) = \sum_i EAD_i \cdot LGD_i \cdot [\Phi_\epsilon(\frac{\Phi_A^{-1}(PD_i) - \sqrt{\rho_i}\Phi_Z^{-1}(1 - \alpha)}{\sqrt{1 - \rho_i}}) - PD_i]$$

Basic ASRF

The portfolio contains 100 credit sensitive contracts and information about their exposure. This is simulated data.

```
load asrfPortfolio.mat
disp(portfolio(1:5,:))
```

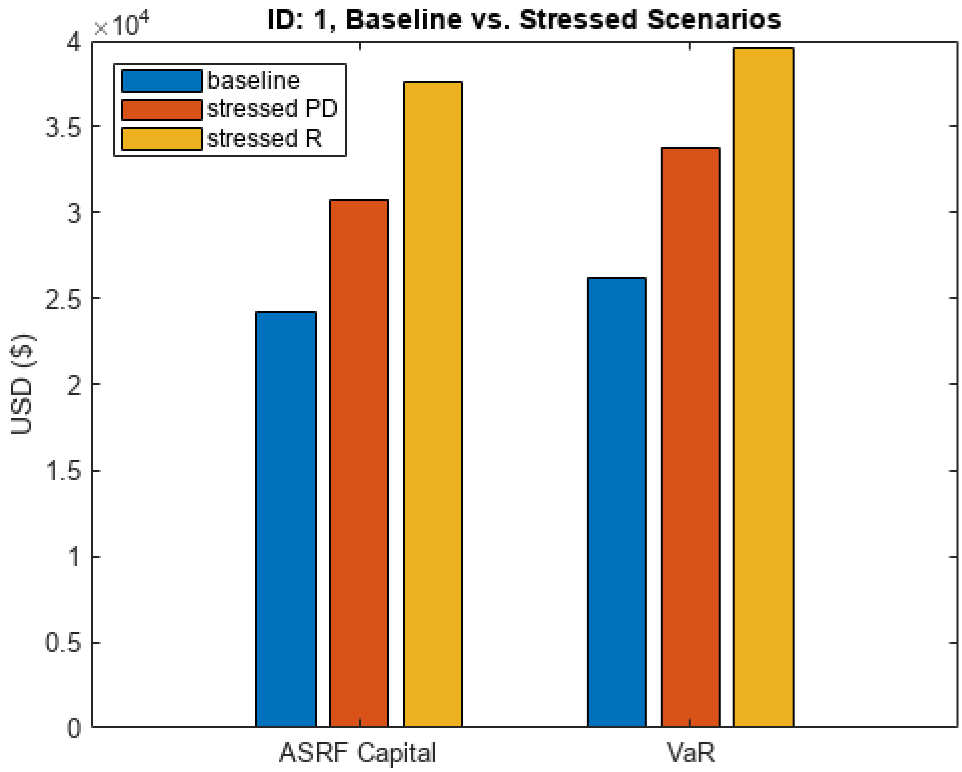
```
    ID       EAD           PD         LGD     AssetClass     Sales      Maturity  
    __    __________    _________    ____    __________    _____    ___________

    1      2.945e+05     0.013644     0.5      "Bank"        NaN     02-Jun-2023
    2     1.3349e+05    0.0017519     0.5      "Bank"        NaN     05-Jul-2021
    3     3.1723e+05      0.01694     0.4      "Bank"        NaN     07-Oct-2018
    4     2.8719e+05     0.013624    0.35      "Bank"        NaN     27-Apr-2022
    5     2.9965e+05     0.013191    0.45      "Bank"        NaN     07-Dec-2022
```

The asset correlations (ρ) in the ASRF model define the correlation between similar assets. The square root of this value, $\sqrt{\rho}$, specifies the correlation between a counterparty's latent variable (A) and the systemic credit factor (Z). Asset correlations can be calibrated by observing correlations in the market or from historical default data. Correlations can also be set using regulatory guidelines (see Basel Capital Requirements section).

Because the ASRF model is a fast, analytical formula, it is convenient to perform sensitivity analysis for a counterparty by varying the exposure parameters and observing how the capital and VaR change.

The following plot shows the sensitivity to PD and asset correlation. The LGD and EAD parameters are scaling factors in the ASRF formula so the sensitivity is straightforward.

```
% Counterparty ID
id = 1;

% Set the default asset correlation to 0.2 as the baseline.
R = 0.2;

% Compute the baseline capital and VaR.
[capital0, var0] = asrf(portfolio.PD(id),portfolio.LGD(id),R,'EAD',portfolio.EAD(id));
% Stressed PD by 50%
[capital1, var1] = asrf(portfolio.PD(id) * 1.5,portfolio.LGD(id),R,'EAD',portfolio.EAD(id));
% Stressed Correlation by 50%
[capital2, var2] = asrf(portfolio.PD(id),portfolio.LGD(id),R * 1.5,'EAD',portfolio.EAD(id));

c = categorical({'ASRF Capital','VaR'});
bar(c,[capital0 capital1 capital2; var0 var1 var2]);
legend({'baseline','stressed PD','stressed R'},'Location','northwest')
title(sprintf('ID: %d, Baseline vs. Stressed Scenarios',id));
ylabel('USD ($)');
```



Basel Capital Requirements

When computing regulatory capital, the Basel documents have additional model specifications on top of the basic ASRF model. In particular, Basel II/III defines specific formulas for computing the asset correlation for exposures in various asset classes as a function of the default probability.

To set up the vector of correlations according to the definitions established in Basel II/III:

```
R = zeros(height(portfolio),1);

% Compute the correlations for corporate, sovereign, and bank exposures.
idx = portfolio.AssetClass == "Corporate" |...
    portfolio.AssetClass == "Sovereign" |...
    portfolio.AssetClass == "Bank";

R(idx) = 0.12 * (1-exp(-50*portfolio.PD(idx))) / (1-exp(-50)) +...
    0.24 * (1 - (1-exp(-50*portfolio.PD(idx))) / (1-exp(-50)));

% Compute the correlations for small and medium entities.
idx = portfolio.AssetClass == "Small Entity" |...
    portfolio.AssetClass == "Medium Entity";

R(idx) = 0.12 * (1-exp(-50*portfolio.PD(idx))) / (1-exp(-50)) +...
    0.24 * (1 - (1-exp(-50*portfolio.PD(idx))) / (1-exp(-50))) -...
    0.04 * (1 - (portfolio.Sales(idx)/1e6 - 5) / 45);

% Compute the correlations for unregulated financial institutions.
```

```
idx = portfolio.AssetClass == "Unregulated Financial";

R(idx) = 1.25 * (0.12 * (1-exp(-50*portfolio.PD(idx))) / (1-exp(-50)) +...
    0.24 * (1 - (1-exp(-50*portfolio.PD(idx))) / (1-exp(-50))));
```

Find the basic ASRF capital using the Basel-defined asset correlations. The default value for the VaR level is 99.9%.

```
asrfCapital = asrf(portfolio.PD,portfolio.LGD,R,'EAD',portfolio.EAD);
```

Additionally, the Basel documents specify a maturity adjustment to be added to each capital calculation. Here we compute the maturity adjustment and update the capital requirements.

```
maturityYears = years(portfolio.Maturity - settle);

b = (0.11852 - 0.05478 * log(portfolio.PD)).^2;
maturityAdj = (1 + (maturityYears - 2.5) .* b)  ./ (1 - 1.5 .* b);

regulatoryCapital = asrfCapital .* maturityAdj;

fprintf('Portfolio Regulatory Capital : $%.2f\n',sum(regulatoryCapital));
```

```
Portfolio Regulatory Capital : $2371316.24
```

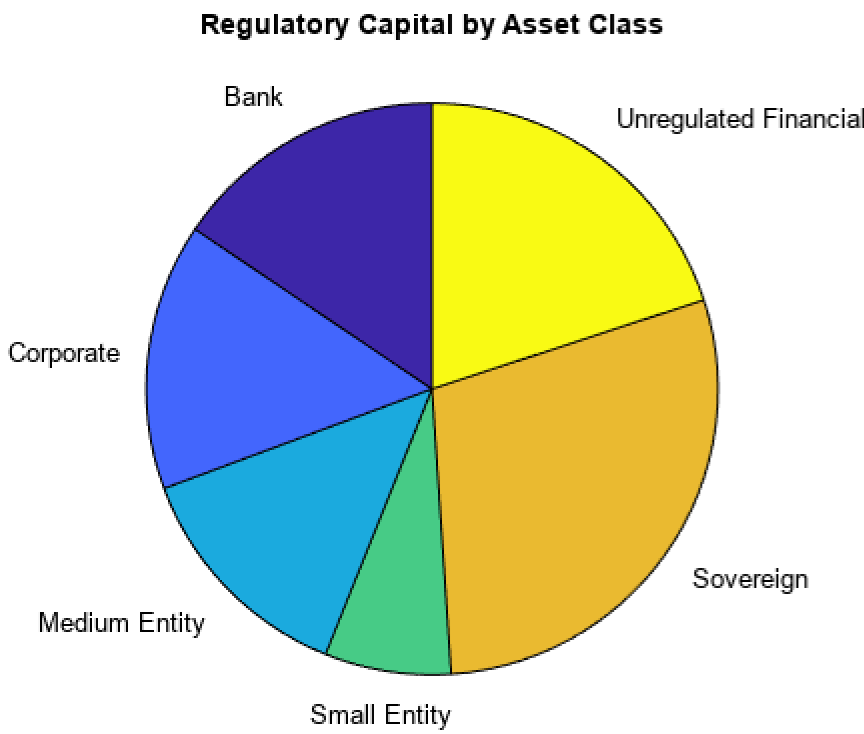
Risk weighted assets (RWA) are calculated as capital * 12.5.

```
RWA = regulatoryCapital * 12.5;

results = table(portfolio.ID,portfolio.AssetClass,RWA,regulatoryCapital,'VariableNames',...
    {'ID','AssetClass','RWA','Capital'});

% Results table
disp(results(1:5,:))
```

```
    ID    AssetClass        RWA         Capital
    __    __________    __________    _______

    1       "Bank"      4.7766e+05     38213
    2       "Bank"           79985    6398.8
    3       "Bank"      2.6313e+05     21050
    4       "Bank"      2.9449e+05     23560
    5       "Bank"      4.1544e+05     33235
```

Aggregate the regulatory capital by asset class.

```
summary = groupsummary(results,"AssetClass","sum","Capital");
pie(summary.sum_Capital,summary.AssetClass)
title('Regulatory Capital by Asset Class');
```




```
disp(summary(:,["AssetClass" "sum_Capital"]))

        AssetClass            sum_Capital
    ______________________    ___________

    "Bank"                    3.6894e+05
    "Corporate"               3.5811e+05
    "Medium Entity"           3.1466e+05
    "Small Entity"             1.693e+05
    "Sovereign"               6.8711e+05
    "Unregulated Financial"    4.732e+05
```

References

1. Basel Committe on Banking Supervision. "*International Convergence of Capital Measurement and Capital Standards*." June 2006 (https://www.bis.org/publ/bcbs128.pdf).

2. Basel Committe on Banking Supervision. "*An Explanatory Note on the Basel II IRB Risk Weight Functions*." July 2005 (https://www.bis.org/bcbs/irbriskweight.pdf).

3. Gordy, M.B. "A Risk-Factor Model Foundation for Ratings-Based Bank Capital Rules." *Journal of Financial Intermediation*. Vol. 12, pp. 199-232, 2003.

See Also
`asrf`

One-Factor Model Calibration

This example demonstrates techniques to calibrate a one-factor model for estimating portfolio credit losses using the `creditDefaultCopula` or `creditMigrationCopula` classes.

This example uses equity return data as a proxy for credit fluctuations. With equity data, sensitivity to a single factor is estimated as a correlation between a stock and an index. The data set contains daily return data for a series of equities, but the one-factor model requires calibration on a year-over-year basis. Assuming that there is no autocorrelation, then the daily cross-correlation between a stock and the market index is equal to the annual cross-correlation. For stocks exhibiting autocorrelation, this example shows how to compute implied annual correlations incorporating the effect of autocorrelation.

Fitting a One-Factor Model

Since corporate defaults are rare, it is common to use a proxy for creditworthiness when calibrating default models. The one-factor copula models the credit worthiness of a company using a latent variable, A:

$$A = wX + \sqrt{1 - w^2}\epsilon$$

where X is the systemic credit factor, w is the weight that defines the sensitivity of a company to the one factor, and ϵ is the idiosyncratic factor. w and ϵ have mean of 0 and variance of 1 and typically are assumed to be either Gaussian or else t distributions.

Compute the correlation between X and A:

$$Corr(A, X) = \frac{Cov(A, X)}{\sigma_A \sigma_X}$$

Since X and A have a variance of `1` by construction and ϵ is uncorrelated with X, then:

$$\begin{aligned} Corr(A, X) &= Cov(A, X) = Cov(wX + \sqrt{1 - w^2}\epsilon, X) \\ &= wCov(X, X) + \sqrt{1 - w^2}Cov(X, \epsilon) = w \end{aligned}$$

If you use stock returns as a proxy for A and the market index returns are a proxy for X, then the weight parameter, w, is the correlation between the stock and the index.

Prepare the Data

Use the returns of the Dow Jones Industrial Average (DJIA) as a signal for the overall credit movement of the market. The returns for the 30 component companies are used to calibrate the sensitivity of each company to the systemic credit movement. Weights for other companies in the stock market are estimated in the same way.

```
% Read one year of DJIA price data.
t = readtable('dowPortfolio.xlsx');

% The table contains dates and the prices for each company at market close
% as well as the DJIA.
disp(head(t(:,1:7)))
```

```
       Dates         DJI      AA       AIG      AXP      BA        C  
    ___________    _____    _____    _____    _____    _____    _____
```

```
    03-Jan-2006    10847    28.72    68.41    51.53    68.63    45.26
    04-Jan-2006    10880    28.89    68.51    51.03    69.34    44.42
    05-Jan-2006    10882    29.12     68.6    51.57    68.53    44.65
    06-Jan-2006    10959    29.02    68.89    51.75    67.57    44.65
    09-Jan-2006    11012    29.37    68.57    53.04    67.01    44.43
    10-Jan-2006    11012    28.44    69.18    52.88    67.33    44.57
    11-Jan-2006    11043    28.05     69.6    52.59     68.3    44.98
    12-Jan-2006    10962    27.68    69.04     52.6     67.9    45.02
```

```
% We separate the dates and the index from the table and compute daily returns using
% tick2ret.
dates = t{2:end,1};
index_adj_close = t{:,2};
stocks_adj_close = t{:,3:end};

index_returns = tick2ret(index_adj_close);
stocks_returns = tick2ret(stocks_adj_close);
```

Compute Single Factor Weights

Compute the single-factor weights from the correlation coefficients between the index returns and the stock returns for each company.

```
[C,daily_pval] = corr([index_returns stocks_returns]);
w_daily = C(2:end,1);
```

These values can be used directly when using a one-factor `creditDefaultCopula` or `creditMigrationCopula`.

Linear regression is often used in the context of factor models. For a one-factor model, a linear regression of the stock returns on the market returns is used by exploiting the fact that the correlation coefficient matches the square root of the coefficient of determination (*R*-squared) of a linear regression.

```
w_daily_regress = zeros(30,1);
for i = 1:30
    lm = fitlm(index_returns,stocks_returns(:,i));
    w_daily_regress(i) = sqrt(lm.Rsquared.Ordinary);
end

% The regressed R values are equal to the index cross correlations.
fprintf('Max Abs Diff : %e\n',max(abs(w_daily_regress(:) - w_daily(:))))
```

```
Max Abs Diff : 8.326673e-16
```

This linear regression fits a model of the form $A = \alpha + \beta X + \epsilon$, which in general does not match the one-factor model specifications. For example, A and X do not have a zero mean and a standard deviation of 1. In general, there is no relationship between the coefficient β and the standard deviation of the error term ϵ. Linear regression is used above only as a tool to get the correlation coefficient between the variables given by the square root of the *R*-squared value.

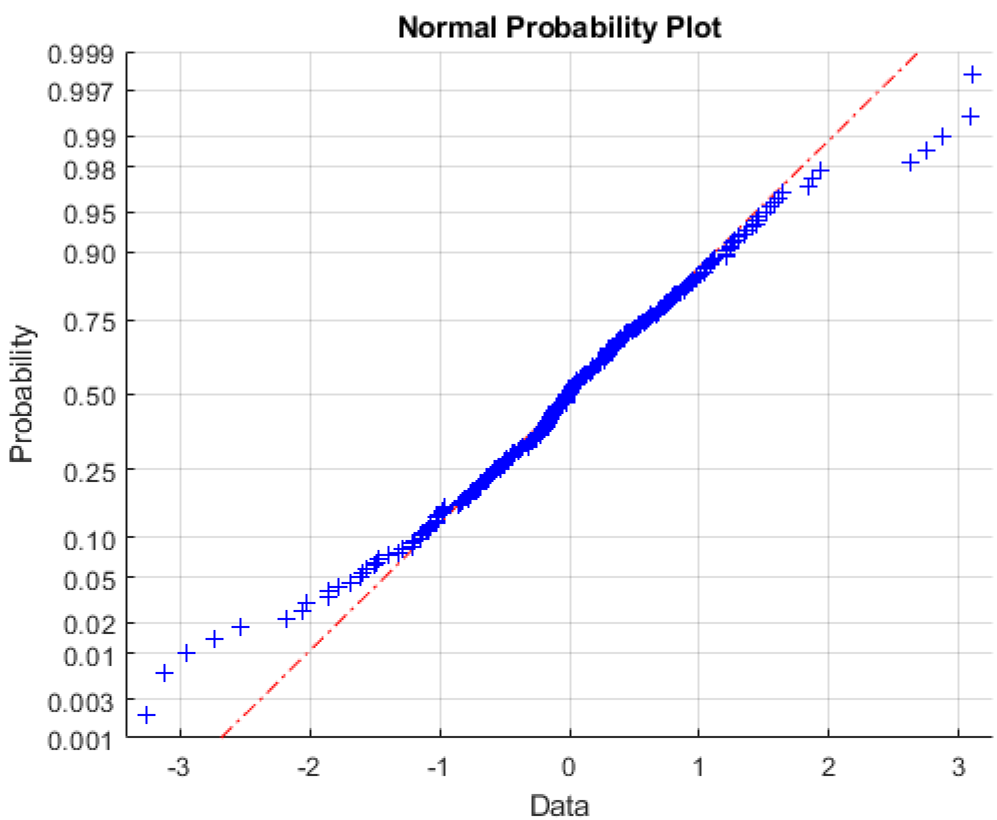
For one-factor model calibration, a useful alternative is to fit a linear regression using the standardized stock and market return data $\widetilde{A}$ and $\widetilde{X}$. "Standardize" here means to subtract the mean and divide by the standard deviation. The model is $\widetilde{A} = \widetilde{\alpha} + \widetilde{\beta}\,\widetilde{X} + \widetilde{\epsilon}$. However, because both $\widetilde{A}$ and $\widetilde{X}$ have a zero mean, the intercept $\widetilde{\alpha}$ is always zero, and because both $\widetilde{A}$ and $\widetilde{X}$ have standard deviation of 1, the standard deviation of the error term satisfies $\mathrm{std}(\tilde{\epsilon}) = \sqrt{1 - \tilde{\beta}^2}$. This exactly matches the specifications of the coefficients of a one-factor model. The one-factor parameter *w* is set to the coefficient $\tilde{\beta}$, and is the same as the value found directly through correlation earlier.

```
w_regress_std = zeros(30,1);
index_returns_std = zscore(index_returns);
stocks_returns_std = zscore(stocks_returns);
for i = 1:30
    lm = fitlm(index_returns_std,stocks_returns_std(:,i));
    w_regress_std(i) = lm.Coefficients{'x1','Estimate'};
end

% The regressed R values are equal to the index cross correlations.
fprintf('Max Abs Diff : %e\n',max(abs(w_regress_std(:) - w_daily(:))))
```

```
Max Abs Diff : 5.551115e-16
```

This approach makes it natural to explore the distributional assumptions of the variables. The `creditDefaultCopula` and `creditMigrationCopula` objects support either normal distributions, or *t* distributions for the underlying variables. For example, when using `normplot` the market returns have heavy tails, therefore a *t*-copula is more consistent with the data.

```
normplot(index_returns_std)
```



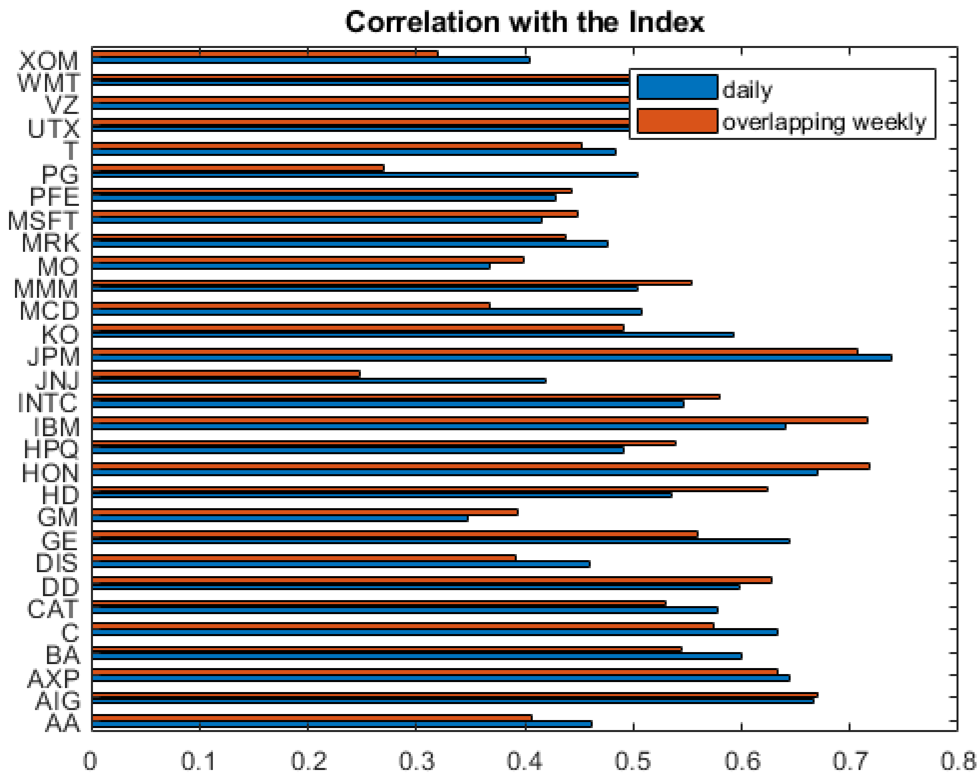
Estimating Correlations for Longer Periods

The weights are computed based on the daily correlation between the stocks and the index. However, the usual goal is to estimate potential losses from credit defaults at some time further in the future, often one year out.

To that end, it is necessary to calibrate the weights such that they correspond to the one-year correlations. It is not practical to calibrate directly against historical annual return data since any reasonable data set does not have enough data to be statistically significant due to the sparsity of the data points.

You then face the problem of computing annual return correlation from a more frequently sampled data set, for example, daily returns. One approach to solving this problem is to use an overlapping window. This way you can consider the set of all overlapping periods of a given length.

```
% As an example, consider an overlapping 1-week window.
index_overlapping_returns = index_adj_close(6:end) ./ index_adj_close(1:end-5) - 1;
stocks_overlapping_returns = stocks_adj_close(6:end,:) ./ stocks_adj_close(1:end-5,:) - 1;

C = corr([index_overlapping_returns stocks_overlapping_returns]);
w_weekly_overlapping = C(2:end,1);

% Compare the correlation with the daily correlation.
% Show the daily vs. the overlapping weekly correlations.
barh([w_daily w_weekly_overlapping])
yticks(1:30)
yticklabels(t.Properties.VariableNames(3:end))
title('Correlation with the Index');
legend('daily','overlapping weekly');
```



The maximum cross-correlation *p*-value for daily returns show a strong statistical significance.

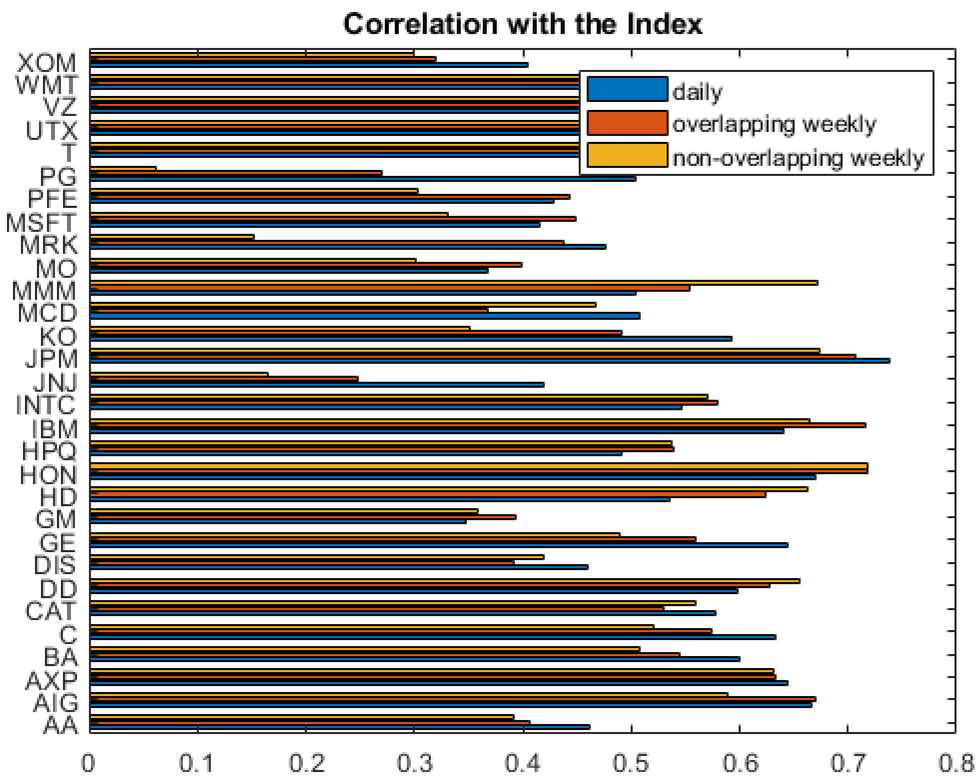
```
maxdailypvalue = max(daily_pval(2:end,1));
disp(table(maxdailypvalue,...
    'VariableNames',{'Daily'},...
    'rownames',{'Maximum p-value'}))
```

```
                       Daily
                    __________

    Maximum p-value    1.5383e-08
```

Moving to an overlapping rolling-window-style weekly correlation gives slightly different correlations. This is a convenient way to estimate longer period correlations from daily data. However, the returns of adjacent overlapping windows are correlated so the corresponding *p*-values for the overlapping weekly returns are not valid since the *p*-value calculation in the `corr` function does not account for overlapping window data sets. For example, adjacent overlapping window returns are composed of many of the same datapoints. This tradeoff is necessary since moving to nonoverlapping windows could result is an unacceptably sparse sample.

```
% Compare to non-overlapping weekly returns
fridays = weekday(dates) == 6;
index_weekly_close = index_adj_close(fridays);
stocks_weekly_close = stocks_adj_close(fridays,:);

index_weekly_returns = tick2ret(index_weekly_close);
stocks_weekly_returns = tick2ret(stocks_weekly_close);
```

```
[C,weekly_pval] = corr([index_weekly_returns stocks_weekly_returns]);
w_weekly_nonoverlapping = C(2:end,1);
maxweeklypvalue = max(weekly_pval(2:end,1));

% Compare the correlation with the daily and overlapping.
barh([w_daily w_weekly_overlapping w_weekly_nonoverlapping])
yticks(1:30)
yticklabels(t.Properties.VariableNames(3:end))
title('Correlation with the Index');
legend('daily','overlapping weekly','non-overlapping weekly');
```



The *p*-values for the nonoverlapping weekly correlations are much higher, indicating a loss of statistical significance.

```
% Compute the number of samples in each series.
numDaily = numel(index_returns);
numOverlapping = numel(index_overlapping_returns);
numWeekly = numel(index_weekly_returns);

disp(table([maxdailypvalue;numDaily],[NaN;numOverlapping],[maxweeklypvalue;numWeekly],...
    'VariableNames',{'Daily','Overlapping','Non_Overlapping'},...
    'rownames',{'Maximum p-value','Sample Size'}))
```

```
                         Daily        Overlapping     Non_Overlapping
                       __________    ____________    _______________
```

```
Maximum p-value    1.5383e-08        NaN             0.66625
Sample Size               250        246                  50
```

Extrapolating Annual Correlation

A common assumption with financial data is that asset returns are temporally uncorrelated. That is, the asset return at time T is uncorrelated to the previous return at time T-1. Under this assumption, the annual cross-correlation is exactly equal to the daily cross-correlation.

Let X_t be the daily log return of the market index on day t and A_t be the daily return of a correlated asset. Using CAPM, the relation is modeled as:

$$A_t = \alpha + \beta X_t + \epsilon_t$$

The one-factor model is a special case of this relationship.

Under the assumption that asset and index returns are each uncorrelated with their respective past, then:

y, $\forall s \neq t$:

$$cov(X_s, X_t) = 0$$

$$cov(\epsilon_s, \epsilon_t) = 0$$

$$cov(A_s, A_t) = 0$$

Let the aggregate annual (log) return for each series be

$$\bar{X} = \sum_{t=1}^{T} X_t$$

$$\bar{A} = \sum_{t=1}^{T} A_t$$

where T could be 252 depending on the underlying daily data.

Let $\sigma_X^2 = var(X_t)$ and $\sigma_A^2 = var(A_t)$ be the daily variances, which are estimated from the daily return data.

The daily covariance between X_t and A_t is:

$$cov(X_t, A_t) = cov(X_t, \alpha + \beta X_t + \epsilon_t) = \beta \sigma_X^2$$

The daily correlation between X_t and A_t is:

$$corr(X_t, A_t) = \frac{cov(X_t, A_t)}{\sqrt{\sigma_X^2 \sigma_A^2}} = \beta \frac{\sigma_X}{\sigma_A}$$

Consider the variances and covariances for the aggregate year of returns. Under the assumption of no autocorrelation:

$$var(\bar{X}) = var(\sum_{t=1}^{T} X_t) = T\sigma_X^2$$

$$var(\bar{A}) = var(\sum_{t=1}^{T} A_t) = T\sigma_A^2$$

$$cov(\bar{X}, \bar{A}) = cov[\sum_{t=1}^{T} X_t, \sum_{t=1}^{T} (\alpha + \beta X_t + \epsilon_t)] = \beta cov(\bar{X}, \bar{X}) = \beta var(\bar{X}) = \beta T\sigma_x^2$$

The annual correlation between the asset and the index is:

$$corr(\bar{X}, \bar{A}) = \frac{cov(\bar{X}, \bar{A})}{\sqrt{var(\bar{X})var(\bar{A})}} = \frac{\beta T\sigma_X^2}{\sqrt{T\sigma_X^2 T\sigma_A^2}} = \beta\frac{\sigma_X}{\sigma_A} = w$$

Under the assumption of no autocorrelation, notice that the daily cross-correlation is in fact *equal* to the annual cross-correlation. You can use this assumption directly in the one-factor model by setting the one-factor weights to the daily cross-correlation.

Handling Autocorrelation

If the assumption that assets have no autocorrelation is loosened, then the transformation from daily to annual cross-correlation between assets is not as straightforward. The $var(\bar{X})$ now has additional terms.

First consider the simplest case of computing the variance of $\bar{X}$ when T is equal to 2.

$$var(\bar{X}) = [\sigma_1 \ \ \sigma_2]\begin{bmatrix} 1 & \rho_{12} \\ \rho_{12} & 1 \end{bmatrix}\begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix} = \sigma_1^2 + \sigma_2^2 + 2\rho_{12}\sigma_1\sigma_2$$

Since $\sigma_1 = \sigma_2 = \sigma_X$, then:

$$var(\bar{X}) = \sigma_X^2(2 + 2\rho_{12})$$

Consider $T = 3$. Indicate the correlation between daily returns that are k days apart as $\rho_{\Delta k}$.

$$var(\bar{X}) = [\sigma_1 \ \ \sigma_2 \ \ \sigma_3]\begin{bmatrix} 1 & \rho_{\Delta 1} & \rho_{\Delta 2} \\ \rho_{\Delta 1} & 1 & \rho_{\Delta 1} \\ \rho_{\Delta 2} & \rho_{\Delta 1} & 1 \end{bmatrix}\begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \end{bmatrix} = \sigma_1^2 + \sigma_2^2 + \sigma_3^2 + 2\rho_{\Delta 1}\sigma_1\sigma_2 + 2\rho_{\Delta 1}\sigma_2\sigma_3 + 2\rho_{\Delta 2}\sigma_1\sigma_3 = \sigma_X^2(3$$

$$+ 4\rho_{\Delta 1} + 2\rho_{\Delta 2})$$

In the general case, for the variance of an aggregate T-day return with autocorrelation from trailing k days, there is:

$$var(\bar{X}) = 2\sigma_X^2(T/2 + (T-1)\rho_{\Delta 1}^X + (T-2)\rho_{\Delta 2}^X + \ . \ . \ . + (T-k)\rho_{\Delta k}^X)$$

This is also the same formula for the asset variance:

$$var(\bar{A}) = 2\sigma_A^2(T/2 + (T-1)\rho_{\Delta 1}^A + (T-2)\rho_{\Delta 2}^A + \ . \ . \ . \ + (T-k)\rho_{\Delta k}^A)$$

The covariance between $\bar{X}$ and $\bar{A}$ as shown earlier is equal to $\beta var(\bar{X})$.

Therefore, the cross-correlation between the index and the asset with autocorrelation from a trailing 1 through k days is:

$$corr(\bar{X}, \bar{A}) = \frac{cov(\bar{X}, \bar{A})}{\sqrt{var(\bar{X})var(\bar{A})}} = \frac{\beta var(\bar{X})}{\sqrt{var(\bar{X})var(\bar{A})}} = \beta\sqrt{\frac{var(\bar{X})}{var(\bar{A})}} = \ . \ . \ .$$

$$corr(\bar{X}, \bar{A}) = \beta\sqrt{\frac{2\sigma_X^2(T/2 + (T-1)\rho_{\Delta 1}^X + (T-2)\rho_{\Delta 2}^X + \ . \ . \ . \ + (T-k)\rho_{\Delta k}^X)}{2\sigma_A^2(T/2 + (T-1)\rho_{\Delta 1}^A + (T-2)\rho_{\Delta 2}^A + \ . \ . \ . \ + (T-k)\rho_{\Delta k}^A)}}$$

$$corr(\bar{X}, \bar{A}) = \beta\frac{\sigma_X}{\sigma_A}\sqrt{\frac{T/2 + (T-1)\rho_{\Delta 1}^X + (T-2)\rho_{\Delta 2}^X + \ . \ . \ . \ + (T-k)\rho_{\Delta k}^X}{T/2 + (T-1)\rho_{\Delta 1}^A + (T-2)\rho_{\Delta 2}^A + \ . \ . \ . \ + (T-k)\rho_{\Delta k}^A}}$$

Note that $\beta\frac{\sigma_X}{\sigma_A}$ is the weight under the assumption of no autocorrelation. The square root term provides the adjustment to account for autocorrelation in the series. The adjustment depends more on the difference between the index autocorrelation and the stock autocorrelation, rather than the magnitudes of these autocorrelations. So the annual one-factor weight adjusted for autocorrelation is:

$$w_{adjusted} = w\sqrt{\frac{T/2 + (T-1)\rho_{\Delta 1}^X + (T-2)\rho_{\Delta 2}^X + \ . \ . \ . \ + (T-k)\rho_{\Delta k}^X}{T/2 + (T-1)\rho_{\Delta 1}^A + (T-2)\rho_{\Delta 2}^A + \ . \ . \ . \ + (T-k)\rho_{\Delta k}^A}}$$

Compute Weights with Autocorrelation

Look for autocorrelation in each of the stocks with the previous day's return, and adjust the weights to incorporate the effect of a one-day autocorrelation.

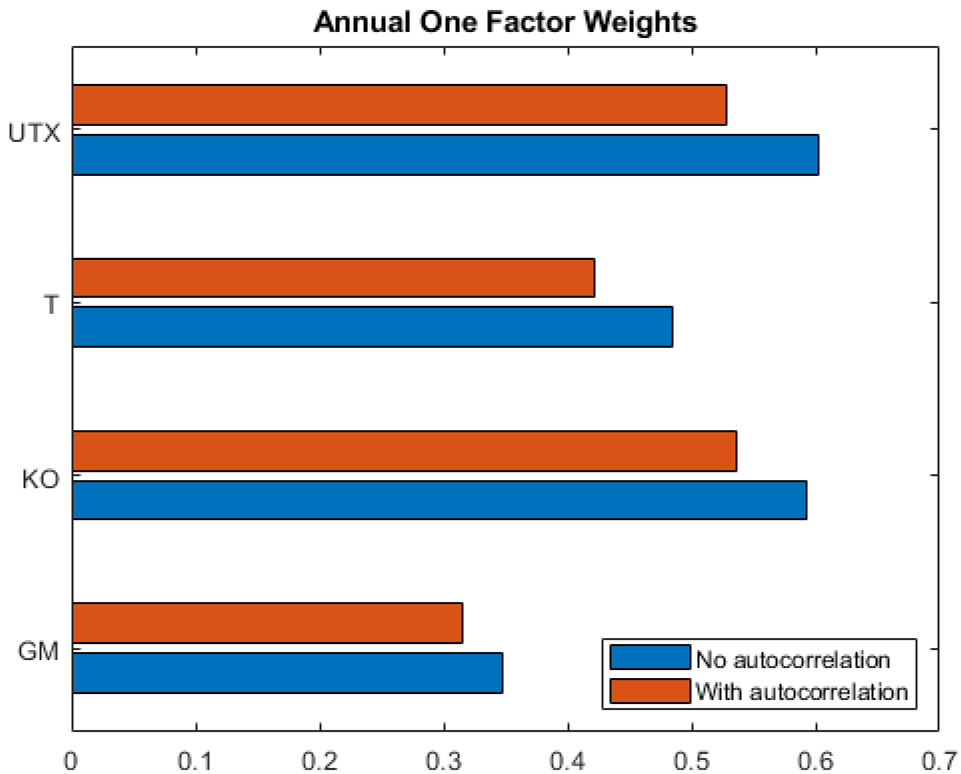
```
corr1 = zeros(30,1);
pv1 = zeros(30,1);
for stockidx = 1:30
    [corr1(stockidx),pv1(stockidx)] = corr(stocks_returns(2:end,stockidx),stocks_returns(1:end-1
end
autocorrIdx = find(pv1 < 0.05)
```

```
autocorrIdx = 4×1

    10
    18
    26
    27
```

There are four stocks with low p-values that may indicate the presence of autocorrelation. Estimate the annual cross-correlation with the index under this model, considering the one-day autocorrelation.

```
% The weights based off of yearly cross correlation are equal to the daily cross
% correlation multiplied by an additional factor.
T = 252;
```

```
w_yearly = w_daily;
[rho_index, pval_index] = corr(index_returns(1:end-1),index_returns(2:end));

% Check to see if the index has any significant autocorrelation.
fprintf('One day autocorrelation in the index p-value: %f\n',pval_index);

One day autocorrelation in the index p-value: 0.670196

if pval_index < 0.05
    % If the p-value indicates there is no significant autocorrelation in the index,
    % set its rho to 0.
    rho_index = 0;
end

w_yearly(autocorrIdx) = w_yearly(autocorrIdx) .*...
    sqrt((T/2 + (T-1) .* rho_index) ./ (T/2 + (T-1) .* corr1(autocorrIdx)));

% Compare the adjusted annual cross correlation values to the daily values.
barh([w_daily(autocorrIdx) w_yearly(autocorrIdx)])
yticks(1:4);
allNames = t.Properties.VariableNames(3:end);
yticklabels(allNames(autocorrIdx))
title('Annual One Factor Weights');
legend('No autocorrelation','With autocorrelation','location','southeast');
```



See Also

`creditDefaultCopula` | `simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios`

Related Examples

- “Credit Simulation Using Copulas” on page 4-2
- “creditDefaultCopula Simulation Workflow” on page 4-5

More About

- “Risk Modeling with Risk Management Toolbox” on page 1-3

Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models

This example shows how to work with consumer credit panel data to create through-the-cycle (TTC) and point-in-time (PIT) models and compare their respective probabilities of default (PD).

The PD of an obligor is a fundamental risk parameter in credit risk analysis. The PD of an obligor depends on customer-specific risk factors as well as macroeconomic risk factors. Because they incorporate macroeconomic conditions differently, TTC and PIT models produce different PD estimates.

A TTC credit risk measure primarily reflects the credit risk trend of a customer over the long term. Transient, short-term changes in credit risk that are likely to be reversed with the passage of time get smoothed out. The predominant features of TTC credit risk measures are their high degree of stability over the credit cycle and the smoothness of change over time.

A PIT credit risk measure utilizes all available and pertinent information as of a given date to estimate the PD of a customer over a given time horizon. The information set includes not just expectations about the credit risk trend of a customer over the long term but also geographic, macroeconomic, and macro-credit trends.

Previously, according to the Basel II rules, regulators called for the use of TTC PDs, losses given default (LGDs), and exposures at default (EADs). However, with to the new IFRS9 and proposed CECL accounting standards, regulators now require institutions to use PIT projections of PDs, LGDs, and EADs. By accounting for the current state of the credit cycle, PIT measures closely track the variations in default and loss rates over time.

Load Panel Data

The main data set in this example (`data`) contains the following variables:

- `ID` — Loan identifier.
- `ScoreGroup` — Credit score at the beginning of the loan, discretized into three groups: `High Risk`, `Medium Risk`, and `Low Risk`.
- `YOB` — Years on books.
- `Default` — Default indicator. This is the response variable.
- `Year` — Calendar year.

The data also includes a small data set (`dataMacro`) with macroeconomic data for the corresponding calendar years:

- `Year` — Calendar year.
- `GDP` — Gross domestic product growth (year over year).
- `Market` — Market return (year over year).

The variables `YOB`, `Year`, `GDP`, and `Market` are observed at the end of the corresponding calendar year. `ScoreGroup` is a discretization of the original credit score when the loan started. A value of `1` for `Default` means that the loan defaulted in the corresponding calendar year.

This example uses simulated data, but you can apply the same approach to real data sets.

Load the data and view the first 10 rows of the table. The panel data is stacked and the observations for the same ID are stored in contiguous rows, creating a tall, thin table. The panel is unbalanced because not all IDs have the same number of observations.

```
load RetailCreditPanelData.mat
disp(head(data,10));
```

```
    ID     ScoreGroup     YOB    Default    Year
    __    ___________    ___    _______    ____

    1     Low Risk        1        0       1997
    1     Low Risk        2        0       1998
    1     Low Risk        3        0       1999
    1     Low Risk        4        0       2000
    1     Low Risk        5        0       2001
    1     Low Risk        6        0       2002
    1     Low Risk        7        0       2003
    1     Low Risk        8        0       2004
    2     Medium Risk     1        0       1997
    2     Medium Risk     2        0       1998
```

```
nRows = height(data);
UniqueIDs = unique(data.ID);
nIDs = length(UniqueIDs);
fprintf('Total number of IDs: %d\n',nIDs)
```

```
Total number of IDs: 96820
```

```
fprintf('Total number of rows: %d\n',nRows)
```

```
Total number of rows: 646724
```

Default Rates by Year

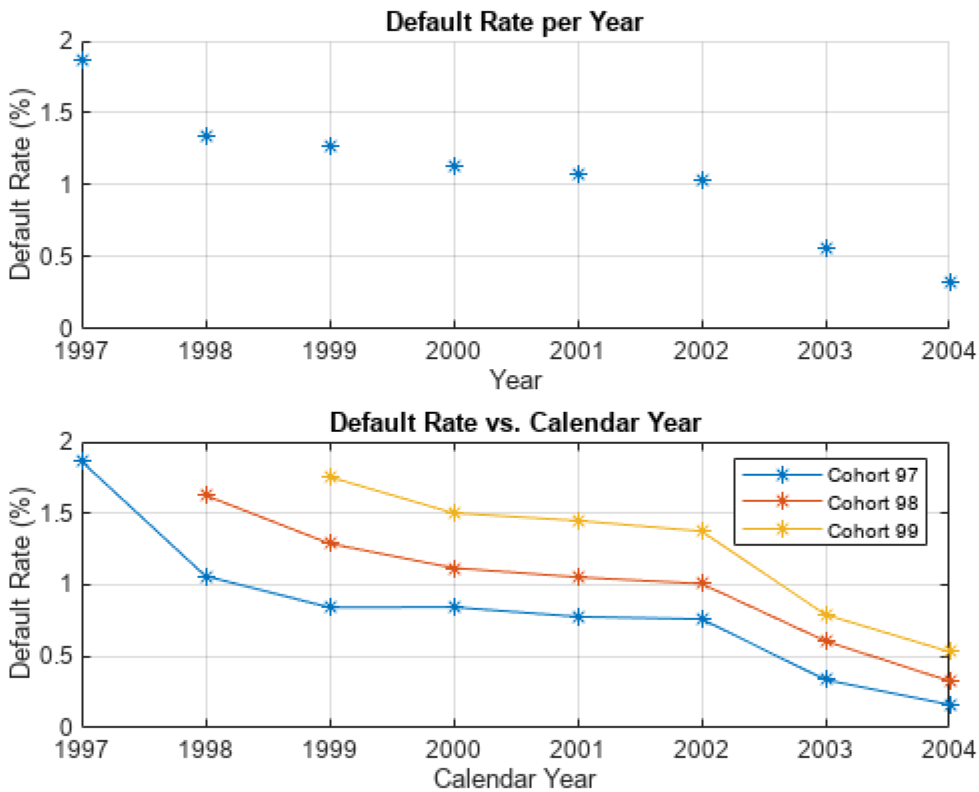
Use `Year` as a grouping variable to compute the observed default rate for each year. Use the `groupsummary` function to compute the mean of the `Default` variable, grouping by the `Year` variable. Plot the results on a scatter plot which shows that the default rate goes down as the years increase.

```
DefaultPerYear = groupsummary(data,'Year','mean','Default');
NumYears = height(DefaultPerYear);
disp(DefaultPerYear)
```

```
    Year    GroupCount    mean_Default
    ____    __________    ____________

    1997      35214         0.018629
    1998      66716         0.013355
    1999      94639         0.012733
    2000      92891         0.011379
    2001      91140         0.010742
    2002      89847         0.010295
    2003      88449        0.0056417
    2004      87828        0.0032905
```

```
subplot(2,1,1)
scatter(DefaultPerYear.Year, DefaultPerYear.mean_Default*100,'*');
grid on
xlabel('Year')
```

```
ylabel('Default Rate (%)')
title('Default Rate per Year')
% Get IDs of the 1997, 1998, and 1999 cohorts
IDs1997 = data.ID(data.YOB==1&data.Year==1997);
IDs1998 = data.ID(data.YOB==1&data.Year==1998);
IDs1999 = data.ID(data.YOB==1&data.Year==1999);
% Get default rates for each cohort separately
ObsDefRate1997 = groupsummary(data(ismember(data.ID,IDs1997),:),...
    'YOB','mean','Default');

ObsDefRate1998 = groupsummary(data(ismember(data.ID,IDs1998),:),...
    'YOB','mean','Default');

ObsDefRate1999 = groupsummary(data(ismember(data.ID,IDs1999),:),...
    'YOB','mean','Default');
% Plot against the calendar year
Year = unique(data.Year);
subplot(2,1,2)
plot(Year,ObsDefRate1997.mean_Default*100,'-*')
hold on
plot(Year(2:end),ObsDefRate1998.mean_Default*100,'-*')
plot(Year(3:end),ObsDefRate1999.mean_Default*100,'-*')
hold off
title('Default Rate vs. Calendar Year')
xlabel('Calendar Year')
ylabel('Default Rate (%)')
legend('Cohort 97','Cohort 98','Cohort 99')
grid on
```



The plot shows that the default rate decreases over time. Notice in the plot that loans starting in the years 1997, 1998, and 1999 form three cohorts. No loan in the panel data starts after 1999. This is depicted in more detail in the "Years on Books Versus Calendar Years" section of the example on “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36. The decreasing trend in this plot is explained by the fact that there are only three cohorts in the data and that the pattern for each cohort is decreasing.

TTC Model Using `ScoreGroup` and Years on Books

TTC models are largely unaffected by economic conditions. The first TTC model in this example uses only `ScoreGroup` and `YOB` as predictors of the default rate.

Generate training and testing data sets by splitting the existing data into training and testing data sets that are used for model creation and validation, respectively.

```
NumTraining = floor(0.6*nIDs);

rng('default');
TrainIDInd = randsample(nIDs,NumTraining);
TrainDataInd = ismember(data.ID,UniqueIDs(TrainIDInd));
TestDataInd = ~TrainDataInd;
```

Use the `fitLifetimePDModel` function to fit a `Logistic` model.

```
TTCModel = fitLifetimePDModel(data(TrainDataInd,:),'logistic',...
   'ModelID','TTC','IDVar','ID','AgeVar','YOB','LoanVars','ScoreGroup',...
```

```
   'ResponseVar','Default');
disp(TTCModel.Model)
```

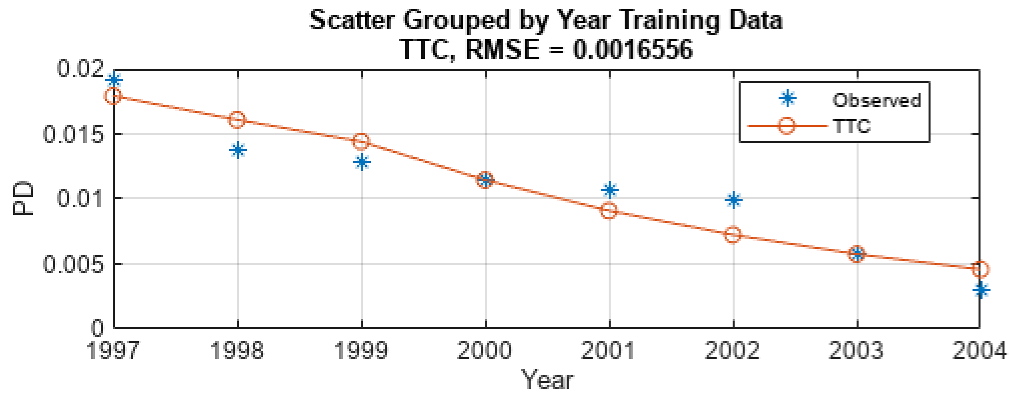
```
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB
    Distribution = Binomial

Estimated Coefficients:
                              Estimate       SE        tStat       pValue
                              ________    ________    _______    __________

    (Intercept)                -3.2453    0.033768    -96.106             0
    ScoreGroup_Medium Risk     -0.7058    0.037103    -19.023    1.1014e-80
    ScoreGroup_Low Risk        -1.2893    0.045635    -28.253    1.3076e-175
    YOB                       -0.22693    0.008437    -26.897    2.3578e-159


388018 observations, 388014 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.83e+03, p-value = 0
```

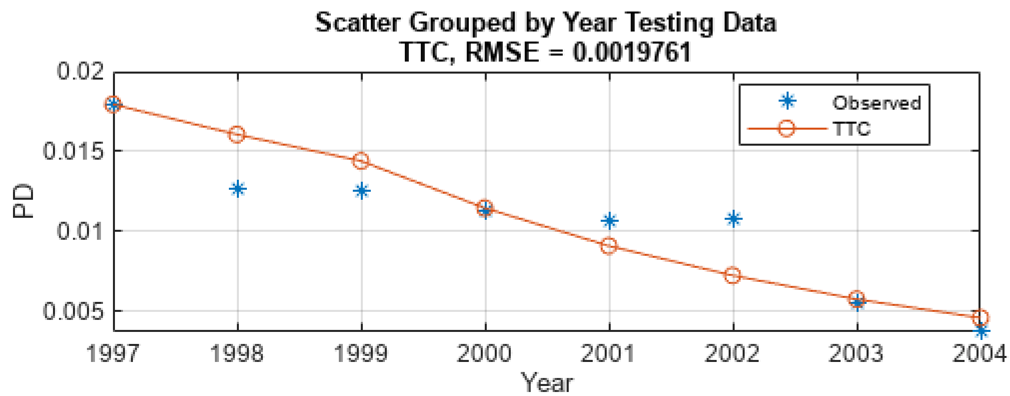
Predict the PD for the training and testing data sets using `predict`.

```
data.TTCPD = zeros(height(data),1);

% Predict the in-sample
data.TTCPD(TrainDataInd) = predict(TTCModel,data(TrainDataInd,:));
% Predict the out-of-sample
data.TTCPD(TestDataInd) = predict(TTCModel,data(TestDataInd,:));
```

Visualize the in-sample fit and out-of-sample fit using `modelCalibrationPlot`.

```
figure;
subplot(2,1,1)
modelCalibrationPlot(TTCModel,data(TrainDataInd,:),'Year','DataID',"Training Data")
subplot(2,1,2)
modelCalibrationPlot(TTCModel,data(TestDataInd,:),'Year','DataID',"Testing Data")
```





PIT Model Using ScoreGroup, Years on Books, GDP, and Market Returns

PIT models vary with the economic cycle. The PIT model in this example uses `ScoreGroup`, `YOB`, `GDP`, and `Market` as predictors of the default rate. Use the `fitLifetimePDModel` function to fit a `Logistic` model.

```
% Add the GDP and Market returns columns to the original data

data = join(data, dataMacro);
disp(head(data,10))
```

```
    ID    ScoreGroup     YOB    Default    Year      TTCPD       GDP     Market
    __    ___________    ___    _______    ____    _________    _____    ______

    1     Low Risk        1        0       1997    0.0084797     2.72      7.61
    1     Low Risk        2        0       1998    0.0067697     3.57     26.24
    1     Low Risk        3        0       1999    0.0054027     2.86      18.1
    1     Low Risk        4        0       2000    0.0043105     2.43      3.19
    1     Low Risk        5        0       2001    0.0034384     1.26    -10.51
    1     Low Risk        6        0       2002    0.0027422    -0.59    -22.95
    1     Low Risk        7        0       2003    0.0021867     0.63      2.78
    1     Low Risk        8        0       2004    0.0017435     1.85      9.48
    2     Medium Risk     1        0       1997     0.015097     2.72      7.61
    2     Medium Risk     2        0       1998     0.012069     3.57     26.24
```

```
PITModel = fitLifetimePDModel(data(TrainDataInd,:),'logistic',...
   'ModelID','PIT','IDVar','ID','AgeVar','YOB','LoanVars','ScoreGroup',...
```

```
   'MacroVars',{'GDP' 'Market'},'ResponseVar','Default');
disp(PITModel.Model)
```

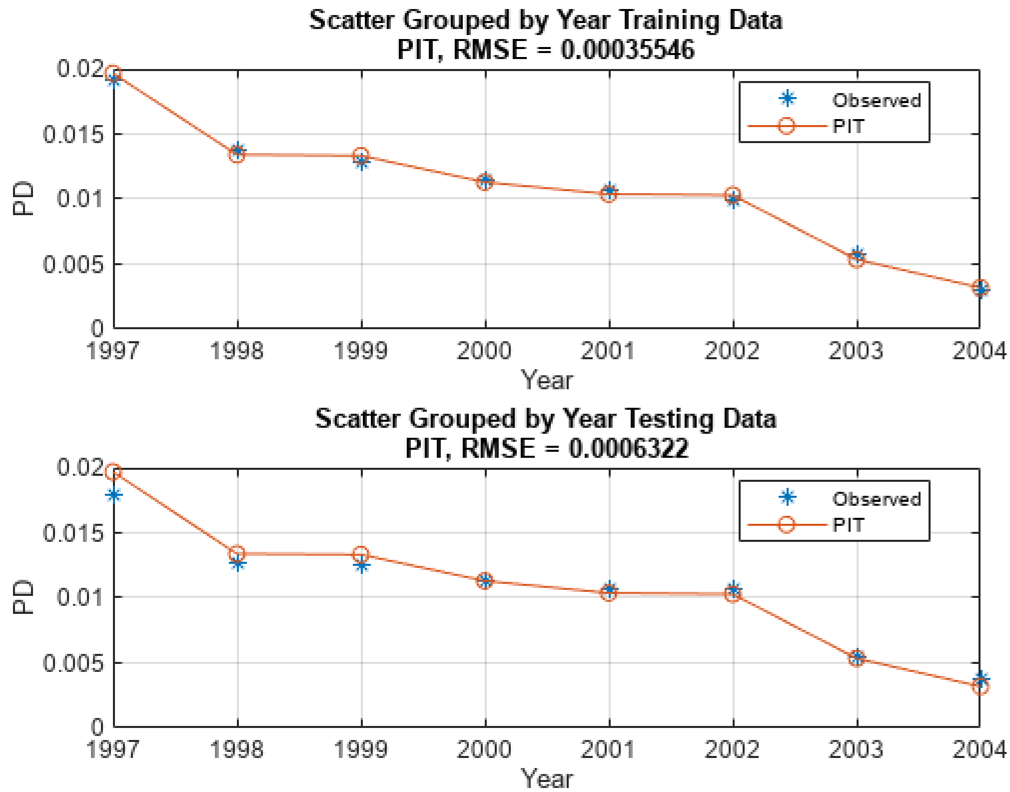
```
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue   
                              __________    _________    ________    ___________

    (Intercept)                   -2.667      0.10146     -26.287    2.6919e-152
    ScoreGroup_Medium Risk      -0.70751     0.037108     -19.066     4.8223e-81
    ScoreGroup_Low Risk          -1.2895     0.045639     -28.253    1.2892e-175
    YOB                         -0.32082     0.013636     -23.528    2.0867e-122
    GDP                         -0.12295     0.039725      -3.095      0.0019681
    Market                    -0.0071812    0.0028298     -2.5377       0.011159


388018 observations, 388012 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.97e+03, p-value = 0
```

Predict the PD for training and testing data sets using `predict`.

```
data.PITPD = zeros(height(data),1);

% Predict in-sample
data.PITPD(TrainDataInd) = predict(PITModel,data(TrainDataInd,:));
% Predict out-of-sample
data.PITPD(TestDataInd) = predict(PITModel,data(TestDataInd,:));
```

Visualize the in-sample fit and out-of-sample fit using `modelCalibrationPlot`.

```
figure;
subplot(2,1,1)
modelCalibrationPlot(PITModel,data(TrainDataInd,:),'Year','DataID',"Training Data")
subplot(2,1,2)
modelCalibrationPlot(PITModel,data(TestDataInd,:),'Year','DataID',"Testing Data")
```



In the PIT model, as expected, the predictions match the observed default rates more closely than in the TTC model. Although this example uses simulated data, qualitatively, the same type of model improvement is expected when moving from TTC to PIT models for real world data, although the overall error might be larger than in this example. The PIT model fit is typically better than the TTC model fit and the predictions typically match the observed rates.

Calculate TTC PD Using the PIT Model

Another approach for calculating TTC PDs is to use the PIT model and then replace the `GDP` and `Market` returns with the respective average values. In this approach, you use the mean values over an entire economic cycle (or an even longer period) so that only baseline economic conditions influence the model, and any variability in default rates is due to other risk factors. You can also enter forecasted baseline values for the economy that are different from the mean observed for the most recent economic cycle. For example, using the median instead of the mean reduces the error.

You can also use this approach of calculating TTC PDs by using the PIT model as a tool for scenario analysis, however; this cannot be done in the first version of the TTC model. The added advantage of this approach is that you can use a single model for both the TTC and PIT predictions. This means that you need to validate and maintain only one model.

```
% Modify the data to replace the GDP and Market returns with the corresponding average values
data.GDP(:) = median(data.GDP);
data.Market = repmat(mean(data.Market), height(data), 1);
disp(head(data,10));
```

```
    ID    ScoreGroup    YOB    Default    Year      TTCPD      GDP    Market      PITPD
    __    __________    ___    _______    ____    _________    ____    ______    _________
```

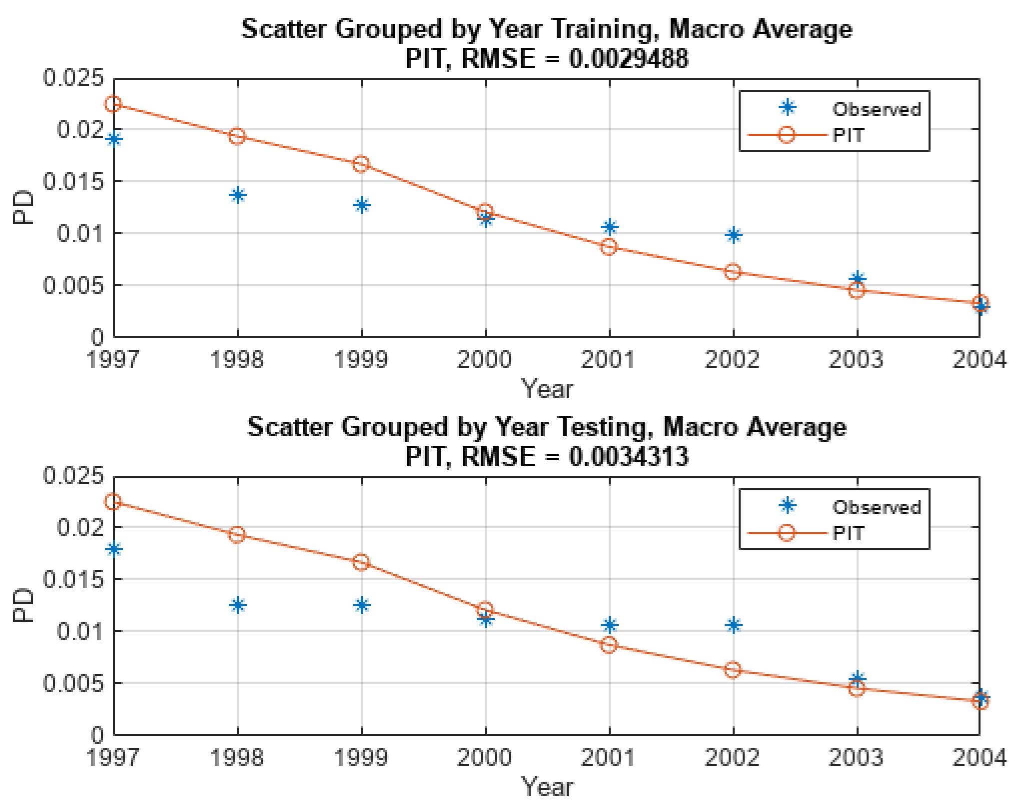
1	Low Risk	1	0	1997	0.0084797	1.85	3.2263	0.0093187
1	Low Risk	2	0	1998	0.0067697	1.85	3.2263	0.005349
1	Low Risk	3	0	1999	0.0054027	1.85	3.2263	0.0044938
1	Low Risk	4	0	2000	0.0043105	1.85	3.2263	0.0038285
1	Low Risk	5	0	2001	0.0034384	1.85	3.2263	0.0035402
1	Low Risk	6	0	2002	0.0027422	1.85	3.2263	0.0035259
1	Low Risk	7	0	2003	0.0021867	1.85	3.2263	0.0018336
1	Low Risk	8	0	2004	0.0017435	1.85	3.2263	0.0010921
2	Medium Risk	1	0	1997	0.015097	1.85	3.2263	0.016554
2	Medium Risk	2	0	1998	0.012069	1.85	3.2263	0.0095319

Predict the PD for training and testing data sets using `predict`.

```
data.TTCPD2 = zeros(height(data),1);

% Predict in-sample
data.TTCPD2(TrainDataInd) = predict(PITModel,data(TrainDataInd,:));
% Predict out-of-sample
data.TTCPD2(TestDataInd) = predict(PITModel,data(TestDataInd,:));
```

Visualize the in-sample fit and out-of-sample fit using `modelCalibrationPlot`.

```
f = figure;
subplot(2,1,1)
modelCalibrationPlot(PITModel,data(TrainDataInd,:),'Year','DataID',"Training, Macro Average")
subplot(2,1,2)
modelCalibrationPlot(PITModel,data(TestDataInd,:),'Year','DataID',"Testing, Macro Average")
```



Reset original values of the GDP and Market variables. The TTC PD values predicted using the PIT model and median or mean macro values are stored in the TTCPD2 column and that column is used to compare the predictions against other models below.

```
data.GDP = [];
data.Market = [];
data = join(data,dataMacro);
disp(head(data,10))
```

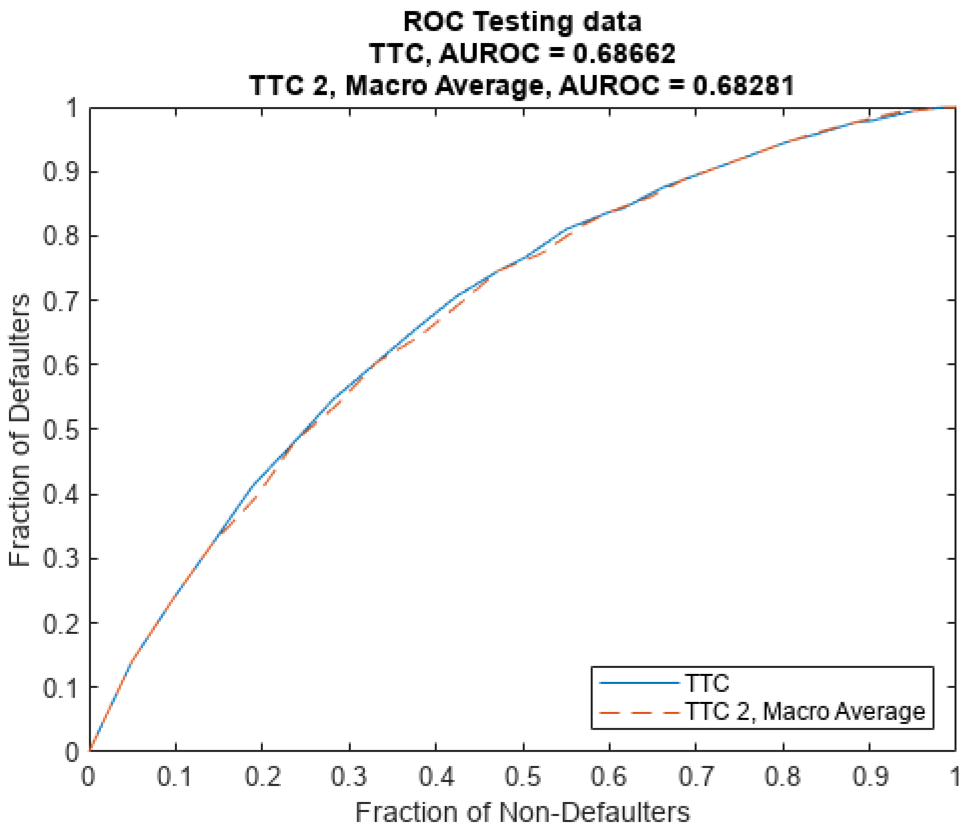
```
    ID     ScoreGroup     YOB    Default    Year      TTCPD        PITPD        TTCPD2       GDP
    __    ___________    ___    _______    ____    _________    _________    _________    _____

    1     Low Risk        1        0       1997    0.0084797    0.0093187     0.010688     2.72
    1     Low Risk        2        0       1998    0.0067697     0.005349    0.0077772     3.57
    1     Low Risk        3        0       1999    0.0054027    0.0044938    0.0056548     2.86
    1     Low Risk        4        0       2000    0.0043105    0.0038285    0.0041093     2.43
    1     Low Risk        5        0       2001    0.0034384    0.0035402    0.0029848     1.26
    1     Low Risk        6        0       2002    0.0027422    0.0035259    0.0021674    -0.59
    1     Low Risk        7        0       2003    0.0021867    0.0018336    0.0015735     0.63
    1     Low Risk        8        0       2004    0.0017435    0.0010921    0.0011422     1.85
    2     Medium Risk     1        0       1997     0.015097     0.016554     0.018966     2.72
    2     Medium Risk     2        0       1998     0.012069    0.0095319     0.013833     3.57
```

Compare the Models

First, compare the two versions of the TTC model.

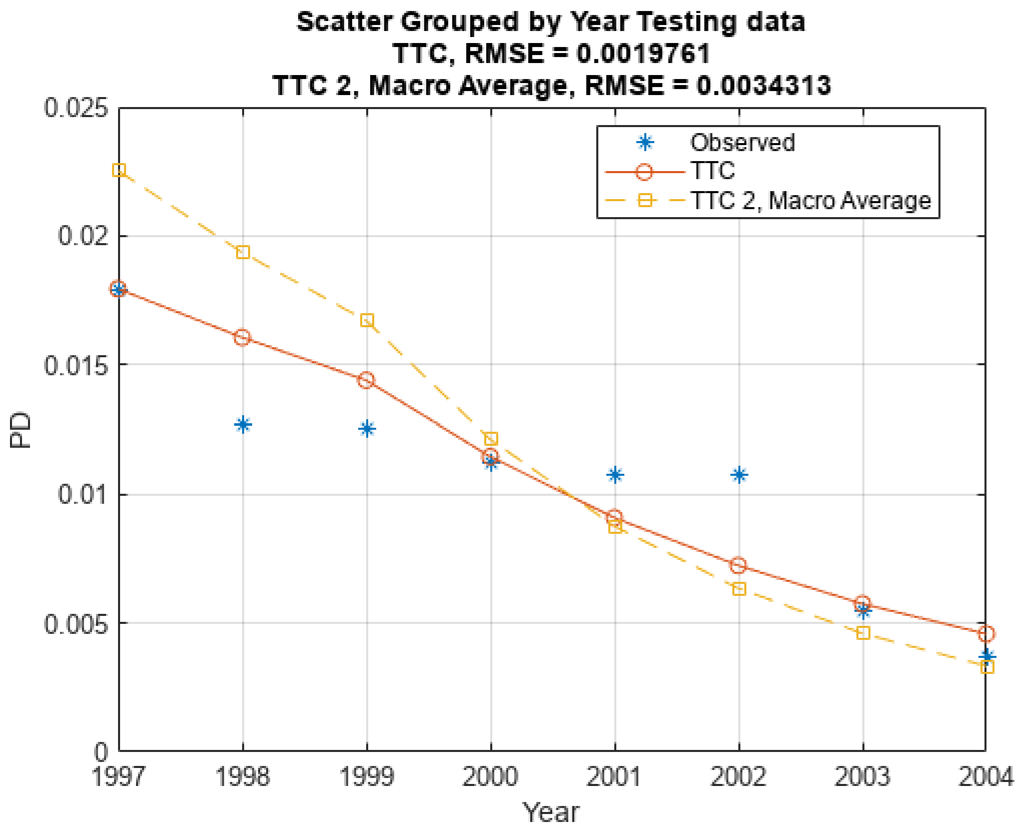
Compare the model discrimination using modelDiscriminationPlot. The two models have very similar performance ranking customers, as measured by the receiver operating characteristic (ROC) curve and the area under the ROC curve (AUROC, or simply AUC) metric.

```
figure;
modelDiscriminationPlot(TTCModel,data(TestDataInd,:),"DataID",'Testing data',"ReferencePD",data.T
```



However, the TTC model is more accurate, the predicted PD values are closer to the observed default rates. The plot generated using `modelCalibrationPlot` demonstrates that the root mean squared error (RMSE) reported in the plot confirms the TTC model is more accurate for this data set.

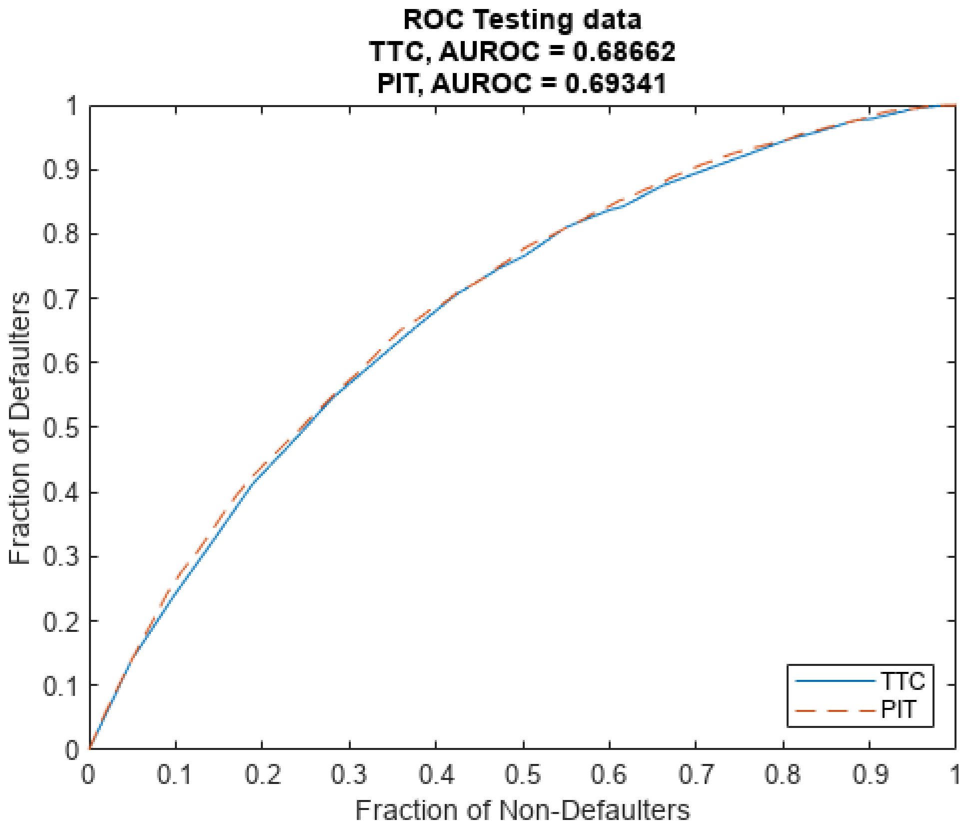
```
modelCalibrationPlot(TTCModel,data(TestDataInd,:),'Year',"DataID",'Testing data',"ReferencePD",da
```



Use `modelDiscriminationPlot` to compare the TTC model and the PIT model.

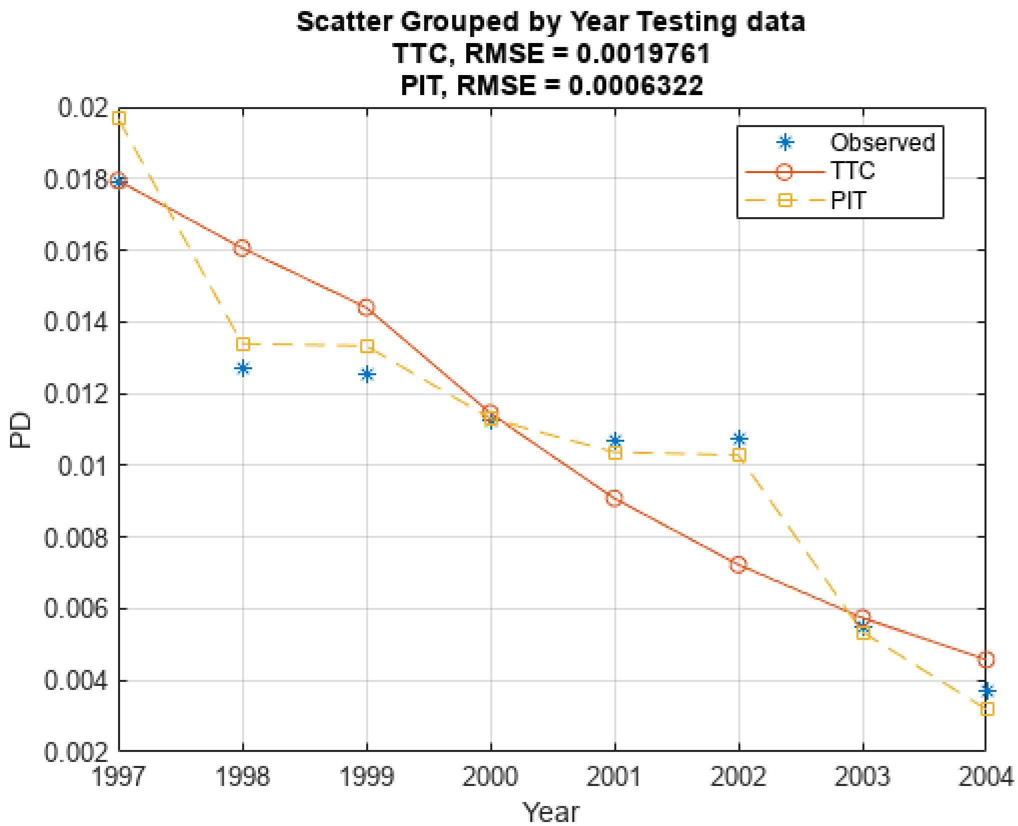
The AUROC is only slightly better for the PIT model, showing that both models are comparable regarding ranking customers by risk.

```
figure;
modelDiscriminationPlot(TTCModel,data(TestDataInd,:),"DataID",'Testing data',"ReferencePD",data.I
```



Use `modelCalibrationPlot` to visualize the model accuracy, or model calibration. The plot shows that the PIT model performs much better, with predicted PD values much closer to the observed default rates. This is expected, since the predictions are sensitive to the macro variables, whereas the TTC model only uses the initial score and the age of the model to make predictions.

```
modelCalibrationPlot(TTCModel,data(TestDataInd,:),'Year',"DataID",'Testing data',"ReferencePD",da
```



You can use `modelDiscrimination` to programmatically access the AUROC and the RMSE without creating a plot.

```
DiscMeasure = modelDiscrimination(TTCModel,data(TestDataInd,:),"DataID",'Testing data',"Reference
disp(DiscMeasure)
```

```
                              AUROC
                             _______

    TTC, Testing data        0.68662
    PIT, Testing data        0.69341
```

```
CalMeasure = modelCalibration(TTCModel,data(TestDataInd,:),'Year',"DataID",'Testing data',"Refere
disp(CalMeasure)
```

```
                                               RMSE
                                             _________

    TTC, grouped by Year, Testing data       0.0019761
    PIT, grouped by Year, Testing data       0.0006322
```

Although all models have comparable discrimination power, the accuracy of the PIT model is much better. However, TTC and PIT models are often used for different purposes, and the TTC model may be preferred if having more stable predictions over time is important.

References

1. Generalized Linear Models documentation, see “Generalized Linear Models”.
2. Baesens, B., D. Rosch, and H. Scheule. *Credit Risk Analytics.* Wiley, 2016.

Model Loss Given Default

This example shows how to fit different types of models to loss given default (LGD) data. This example demonstrates the following approaches:

- Basic nonparametric approach using mean values on page 4-94
- Simple regression model on page 4-96
- Tobit (censored) regression model on page 4-98
- Beta regression model on page 4-100
- Two-stage model on page 4-103

For all of these approaches, this example shows:

- How to fit a model using training data where the LGD is a function of other variables or predictors.
- How to predict on testing data.

The Model Comparison on page 4-106 section contains a detailed comparison that includes visualizations and several prediction error metrics for of all models in this example.

The Regression, Tobit, and Beta models are fitted using the `fitLGDModel` function from Risk Management Toolbox™. For more information, see "Overview of Loss Given Default Models" on page 1-31.

Introduction

LGD is one of the main parameters for credit risk analysis. Although there are different approaches to estimate credit loss reserves and credit capital, common methodologies require the estimation of probabilities of default (PD), loss given default (LGD), and exposure at default (EAD). The reserves and capital requirements are computed using formulas or simulations that use these parameters. For example, the loss reserves are usually estimated as the expected loss (EL), given by the following formula:

$EL = PD * LGD * EAD$.

Practitioners have decades of experience modeling and forecasting PDs. However, the modeling of LGD (and also EAD) started much later. One reason is the relative scarcity of LGD data compared to PD data. Credit default data (for example, missed payments) is easier to collect and more readily available than are the losses ultimately incurred in the event of a default. When an account is moved to the recovery stage, the information can be transferred to a different system, loans can get consolidated, the recovery process may take a long time, and multiple costs are incurred during the process, some which are hard to track in detail. However, banks have stepped up their efforts to collect data that can be used for LGD modeling, in part due to regulations that require the estimation of these risk parameters, and the modeling of LGD (and EAD) is now widespread in industry.

This example uses simulated LGD data, but the workflow has been applied to real data sets to fit LGD models, predict LGD values, and compare models. The focus of this example is not to suggest a particular approach, but to show how these different models can be fit, how the models are used to predict LGD values, and how to compare the models. This example is also a starting point for variations and extensions of these models; for example, you may want to use more advanced classification and regression tools as part of a two-stage model.

The three predictors in this example are loan specific. However, you can use the approaches described in this example with data sets that include multiple predictors and even macroeconomic variables. Also, you can use models that include macroeconomic predictors for stress testing or lifetime LGD modeling to support regulatory requirements such as CCAR, IFRS 9, and CECL. For more information, see “Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations” on page 4-195.

LGD Data Exploration

The data set in this example is simulated data that captures common features of LGD data. For example, a common feature is the distribution of LGD values, which has high frequencies at `0` (full recovery), and also many observations at `1` (no recovery at all). Another characteristic of LGD data is a significant amount of "noise" or "unexplained" data. You can visualize this "noise" in scatter plots of the response against the predictors, where the dots do not seem to follow a clear trend, and yet some underlying relationships can be detected. Also, it is common to get significant prediction errors for LGD models. Empirical studies show that LGD models have high prediction errors in general. For example, in [4 on page 4-112] the authors report R-squared values ranging from 4% to 43% for a range of models across different portfolios. In this example, all approaches get R-squared values just under 10%. Moreover, finding useful predictors in practice may require important insights into the lending environment of a specific portfolio, for example, knowledge of the legal framework and the collection process [2 on page 4-111]. The simulated data set includes only three predictors and these are variables frequently found in LGD models, namely, the loan-to-value ratio, the age of the loan, and whether the borrower lives in the property or if the borrower bought it for investment purposes.

Data preparation for LGD modeling is beyond the scope of this example. This example assumes the data has been previously prepared, since the focus of the example is on how to fit LGD models and how to use them for prediction. Data preparation for LGD modeling requires a significant amount of work in practice. Data preparation requires consolidation of account information, pulling data from multiple data sources, accounting for different costs and discount rates, and screening predictors [1 on page 4-111] [2 on page 4-111].

Load the data set from the `LGDData.mat` file. The data set is stored in the `data` table. It contains the three predictors and the LGD variable, which is the response variable.

Here is a preview of the data and the descriptions of the data set and the variables.

```
load('LGDData.mat')
disp(head(data))
```

```
      LTV         Age          Type           LGD   
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

```
disp(data.Properties.Description)
```

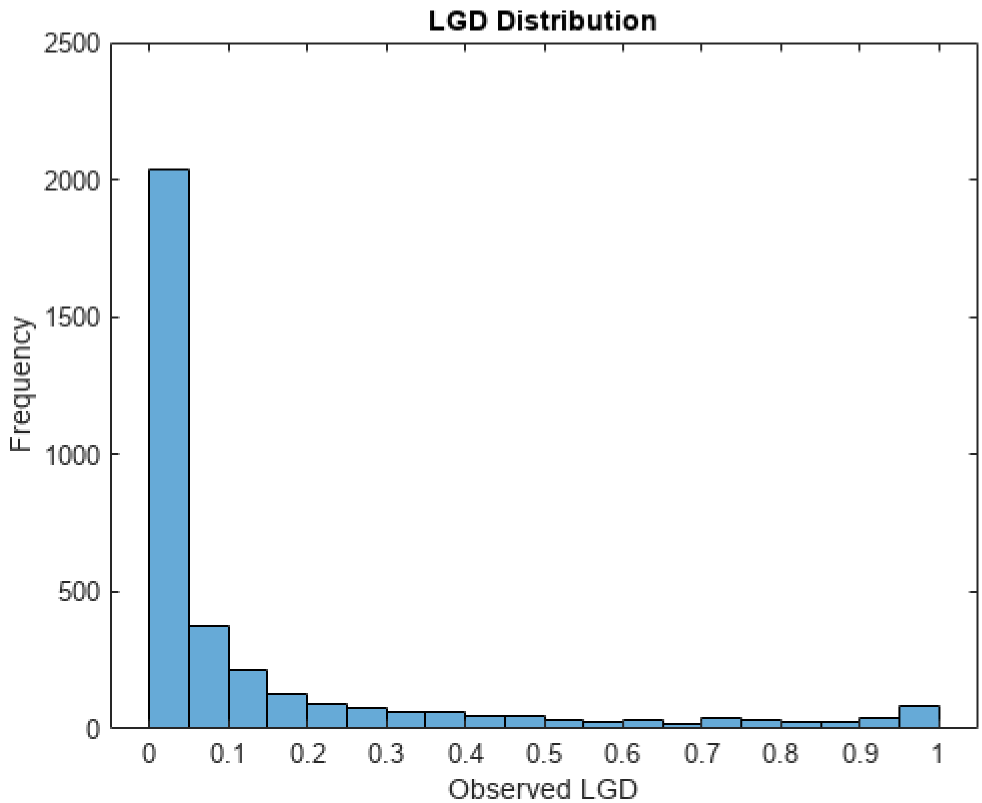
```
Loss given default (LGD) data. This is a simulated data set.
```

```
disp([data.Properties.VariableNames' data.Properties.VariableDescriptions'])
```

```
    {'LTV' }    {'Loan-to-Value (LTV) ratio at the time of default'  }
    {'Age' }    {'Age of the loan in years at the time of default'   }
    {'Type'}    {'Type of property, either residential or investment'}
    {'LGD' }    {'Loss given default'                                }
```

LGD data commonly has values of `0` (no losses, full recovery) or `1` (no recovery at all). The distribution of values in between `0` and `1` takes different shapes depending on the portfolio type and other characteristics.

```
histogram(data.LGD)
title('LGD Distribution')
ylabel('Frequency')
xlabel('Observed LGD')
```



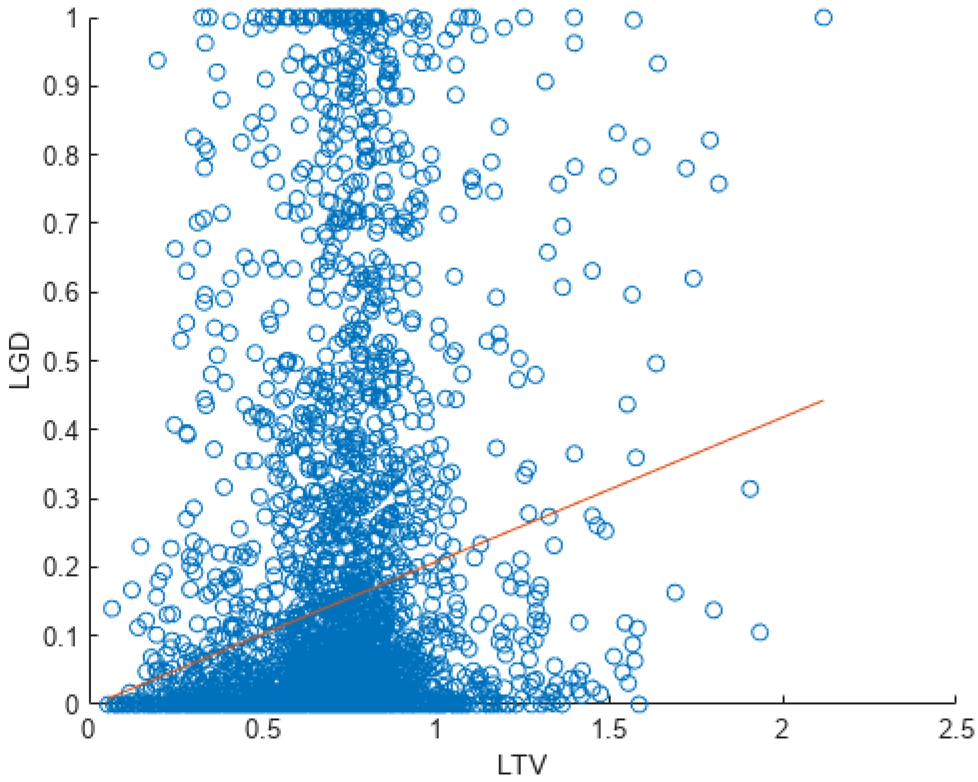
Explore the relationships between the predictors and the response. The Spearman correlation between the selected predictor and the LGD is displayed first. The Spearman correlation is one of the rank order statistics commonly used for LGD modeling [5 on page 4-112].

```
SelectedPredictor = LTV ;

fprintf('Spearman correlation between %s and LGD: %g',SelectedPredictor,corr(double(data.(Selecte
```

```
Spearman correlation between LTV and LGD: 0.271204
```

```
if isnumeric(data.(SelectedPredictor))
    scatter(data.(SelectedPredictor),data.LGD)
    X = [ones(height(data),1) data.(SelectedPredictor)];
```

```
    b = X\data.LGD;
    y = X*b;
    hold on
    plot(data.(SelectedPredictor),y)
    ylim([0 1])
    hold off
    xlabel(SelectedPredictor)
    ylabel('LGD')
end
```

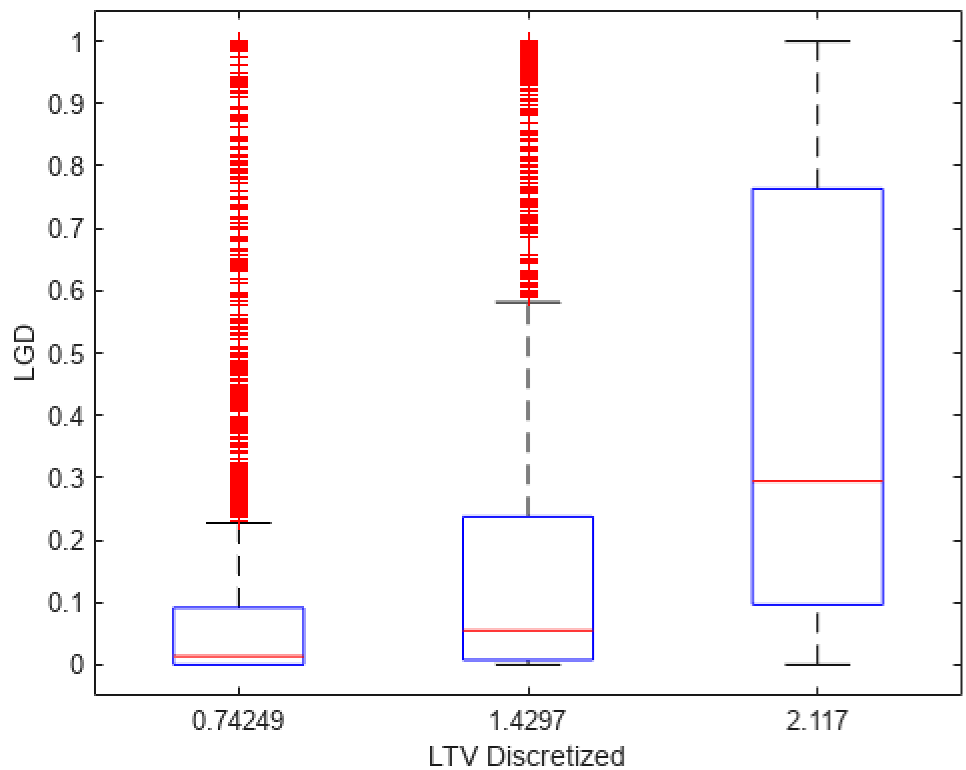


For numeric predictors, there is a scatter plot of the LGD against the selected predictor values, with a superimposed linear fit. There is a significant amount of noise in the data, with points scattered all over the plot. This is a common situation for LGD data modeling. The density of the dots is sometimes different in different areas of the plot, suggesting relationships. The slope of the linear fit and the Spearman correlation give more information about the relationship between the selected predictor and the response.

Visually assessing the density of the points in a scatter plot might not be a reliable approach to understand the distribution of the data. To better understand the distribution of LGD values for different levels of a selected predictor, create a box plot.

```
% Choose the number of discretization levels for numeric predictors
NumLevels = 3 ;
if isnumeric(data.(SelectedPredictor))
    PredictorEdges = linspace(min(data.(SelectedPredictor)),max(data.(SelectedPredictor)),NumLeve
    PredictorDiscretized = discretize(data.(SelectedPredictor),PredictorEdges,'Categorical',strin
```

```
    boxplot(data.LGD,PredictorDiscretized)
    xlabel([SelectedPredictor ' Discretized'])
    ylabel('LGD')
else
    boxplot(data.LGD,data.(SelectedPredictor))
    xlabel(SelectedPredictor)
    ylabel('LGD')
end
```



For categorical data, the box plot is straightforward since a small number of levels are already given. For numeric data, you can discretize the data first and then generate the box plot. Different box sizes and heights show that the distribution of LGD values changes for different predictor levels. A monotonic trend in the median (red horizontal line in the center of the boxes) shows a potential linear relationship between the predictor and the LGD (though possibly a mild relationship, due to the wide distributions).

Mean LGD Over Different Groups

The basic approach to predict LGD is to simply use the mean of the LGD data. Although this is a straightforward approach, easy to understand and use, the downside is that the mean is a constant value and this approach sheds no light on the sensitivity of LGD to other risk factors. In particular, the predictors in the data set are ignored.

To introduce sensitivity to predictors, the mean LGD values can be estimated over different groups or segments of the data, where the groups are defined using ranges of the predictor values. This approach is still a relatively straightforward approach, yet it can noticeably reduce the prediction error compared to a single mean LGD value for all observations.

To start, separate the data set into training and testing data. The same training and testing data sets are used for all approaches in this example.

```
NumObs = height(data);
% Reset the random stream state, for reproducibility
% Comment this line out to generate different data partitions each time the example is run
rng('default');
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

In this example, the groups are defined using the three predictors. `LTV` is discretized into low and high levels. `Age` is discretized into young and old loans. `Type` already has two levels, namely, residential and investment. The groups are all the combinations of these values (for example, low LTV, young loan, residential, and so on).

The number of levels and the specific cut off points are for illustration purposes only, you can base other discretizations on different criteria. Also, using all predictors for the discretization may not be ideal when the data set contains many predictors. In some cases, using a single predictor, or a couple of predictors, may be enough to find useful groups with distinct mean LGD values. When the data includes macro information, the grouping may include a macro variable; for example, the mean LGD value should be different over recessions vs. economic expansions.

Compute the mean LGD over the eight data groups using the training data.

```
% Discretize LTV
LTVEdges = [0 0.5 max(data.LTV)];
data.LTVDiscretized = discretize(data.LTV,LTVEdges,'Categorical',{'low','high'});
% Discretize Age
AgeEdges = [0 2 max(data.Age)];
data.AgeDiscretized = discretize(data.Age,AgeEdges,'Categorical',{'young','old'});
% Find group means on training data
gs = groupsummary(data(TrainingInd,:),{'LTVDiscretized','AgeDiscretized','Type'},'mean','LGD');
disp(gs)
```

```
    LTVDiscretized    AgeDiscretized       Type        GroupCount    mean_LGD
    ______________    ______________    ___________    __________    ________

         low              young         residential        163        0.12166
         low              young         investment          26       0.087331
         low              old           residential        175       0.021776
         low              old           investment          23        0.16379
         high             young         residential       1134        0.16489
         high             young         investment         257        0.25977
         high             old           residential        265       0.066068
         high             old           investment          50        0.11779
```

For prediction, the test data is mapped into the eight groups, and then the corresponding group mean is set as the predicted LGD value.

```
LGDGroupTest = findgroups(data(TestInd,{'LTVDiscretized' 'AgeDiscretized' 'Type'}));
LGDPredictedByGroupMeans = gs.mean_LGD(LGDGroupTest);
```

Store the observed LGD and the predicted LGD in a new table `dataLGDPredicted`. This table stores predicted LGD values for all other approaches in the example.

```
dataLGDPredicted = table;
dataLGDPredicted.Observed = data.LGD(TestInd);
```

```
dataLGDPredicted.GroupMeans = LGDPredictedByGroupMeans;
disp(head(dataLGDPredicted))
```

```
    Observed     GroupMeans
    _________    __________

    0.0064766     0.066068 
     0.007947      0.12166 
     0.063182      0.25977 
            0     0.066068 
      0.10904      0.16489 
            0      0.16489 
      0.89463      0.16489 
            0     0.021776 
```

The Model Comparison on page 4-106 section has a more detailed comparison of all models that includes visualizations and prediction error metrics.

Simple Regression Model

A natural approach is to use a regression model to explicitly model a relationship between the LGD and some predictors. LGD data, however, is bounded in the unit interval, whereas the response variable for linear regression models is, in theory, unbounded.

To apply simple linear regression approaches, the LGD data can be transformed. A common transformation is the logit function, which leads to the following regression model:

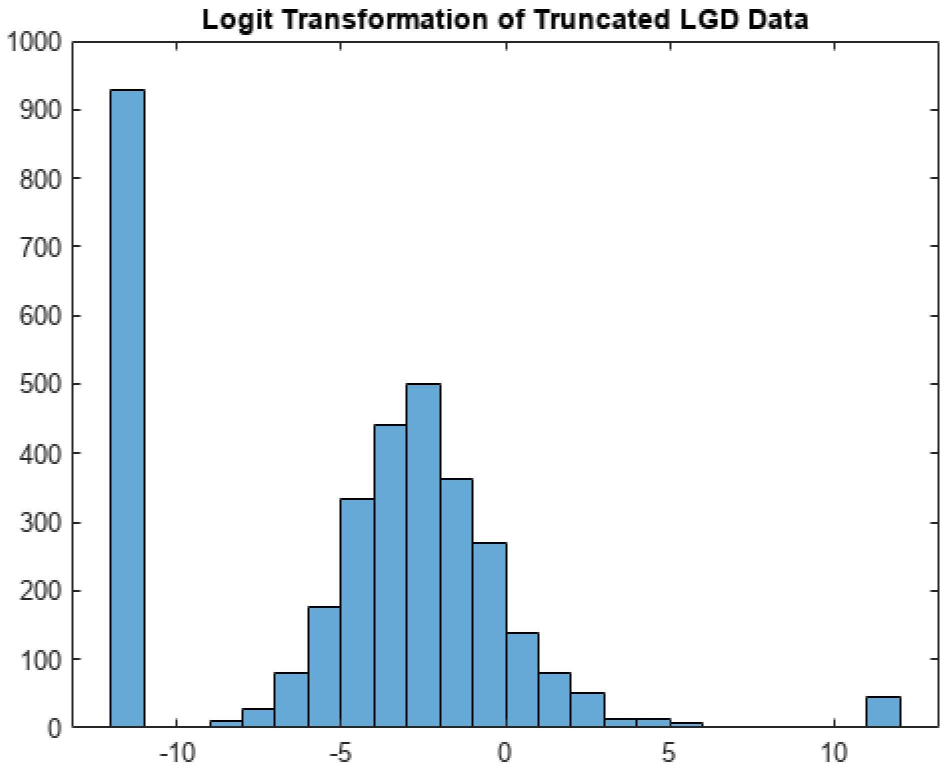
$$\log\left(\frac{\mathrm{LGD}}{1-\mathrm{LGD}}\right) = X\beta + \epsilon, \text{with } \epsilon \sim N\left(0, \sigma^2\right)$$

LGD values of `0` or `1` cause the logit function to take infinite values, so the LGD data is typically truncated before applying the transformation.

```
data.LGDTruncated = data.LGD;
data.LGDTruncated(data.LGD==0) = 0.00001;
data.LGDTruncated(data.LGD==1) = 0.99999;
data.LGDLogit = log(data.LGDTruncated./(1-data.LGDTruncated));
```

Below is the histogram of the transformed LGD data that uses the logit function. The range of values spans positive and negative values, which is consistent with the linear regression requirements. The distribution still shows significant mass probability points at the ends of the distribution.

```
histogram(data.LGDLogit)
title('Logit Transformation of Truncated LGD Data')
```



Other transformations are suggested in the literature [1 on page 4-111]. For example, instead of the logit function, the truncated LGD values can be mapped with the inverse standard normal distribution (similar to a probit model).

Fit a regression model using the `fitLGDModel` function from Risk Management Toolbox™ using the training data. By default, a logit transformation is applied to the LGD response data with a boundary tolerance of `1e-5`. For more information on the supported transformations and optional arguments, see `Regression`.

```
mdlRegression = fitLGDModel(data(TrainingInd,:),'regression','PredictorVars',{'LTV' 'Age' 'Type'
disp(mdlRegression)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Regression"
          Description: ""
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

```
disp(mdlRegression.UnderlyingModel)
```

```
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type
```

```
Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

The model coefficients match the findings in the exploratory data analysis, with a positive coefficient for `LTV`, a negative coefficient for `Age`, and a positive coefficient for investment properties in the `Type` variable.

The `Regression` LGD models support prediction and apply the inverse transformation so the predicted LGD values are in the LGD scale. For example, for the model fitted above that uses the logit transformation, the inverse logit transformation (also known as the logistic, or sigmoid function) is applied by the predict function to return an LGD predicted value.

```
dataLGDPredicted.Regression = predict(mdlRegression,data(TestInd,:));
disp(head(dataLGDPredicted))
```

```
    Observed     GroupMeans    Regression
    _________    __________    __________

    0.0064766     0.066068     0.00091169
     0.007947      0.12166      0.0036758
     0.063182      0.25977        0.18774
            0     0.066068      0.0010877
      0.10904      0.16489       0.011213
            0      0.16489       0.041992
      0.89463      0.16489       0.052947
            0     0.021776     3.7188e-06
```

The Model Comparison on page 4-106 section at the end of this example has a more detailed comparison of all models that includes visualizations and prediction error metrics. In particular, the histogram of the predicted LGD values shows that the regression model predicts many LGD values near zero, even though the high probability near zero was not explicitly modeled.

Tobit Regression Model

Tobit or censored regression is designed for models where the response is bounded. The idea is that there is an underlying (latent) linear model but that the observed response values, in this case the LGD values, are truncated. For this example, use a model censored on both sides with a left limit at `0` and a right limit at `1`, corresponding to the following model formula

$$\mathrm{LGD}_i = \min(\max(0, Y_i^*), 1)$$

with:

$$Y_i^* = X_i\beta + \epsilon_i$$
$$= \beta_0 + \beta_1 X^1{}_i + \cdots + \beta_k X^k{}_i + \epsilon_i ,$$
$$\text{with } \epsilon_i \sim N\left(0, \sigma^2\right)$$

The model parameters are all the βs and the standard deviation of the error σ.

Fit the Tobit regression model with `fitLGDModel` using the training data. By default, a model censored on both sides is fitted with limits at `0` and `1`. For more information on Tobit models, see `Tobit`.

```
mdlTobit = fitLGDModel(data(TrainingInd,:),'tobit','CensoringSide','both','PredictorVars',{'LTV'
disp(mdlTobit)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

```
disp(mdlTobit.UnderlyingModel)
```

```
Tobit regression model:
     LGD = max(0,min(Y*,1))
     Y* ~ 1 + LTV + Age + Type

Estimated coefficients:
                       Estimate        SE         tStat       pValue  
                       _________    _________    _______    __________

    (Intercept)         0.058257     0.027265     2.1367      0.032737
    LTV                  0.20126     0.031354     6.4189    1.6932e-10
    Age                -0.095407    0.0072653    -13.132             0
    Type_investment      0.10208     0.018058     5.6531    1.7915e-08
    (Sigma)              0.29288    0.0057036      51.35             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1521
Number of right-censored observations: 25
Log-likelihood: -698.383
```

Tobit models predict using the unconditional expected value of the response, given the predictor values. For more information, see “Loss Given Default Tobit Models” on page 6-694.

```
dataLGDPredicted.Tobit = predict(mdlTobit,data(TestInd,:));
disp(head(dataLGDPredicted))
```

```
    Observed     GroupMeans    Regression     Tobit  
    _________    __________    __________    ________

    0.0064766     0.066068     0.00091169    0.087889
     0.007947      0.12166      0.0036758     0.12432
```

```
    0.063182      0.25977       0.18774      0.32043
           0     0.066068     0.0010877     0.093354
     0.10904      0.16489      0.011213      0.16718
           0      0.16489      0.041992      0.22382
     0.89463      0.16489      0.052947      0.23695
           0     0.021776     3.7188e-06    0.010234
```

The Model Comparison on page 4-106 section at the end of this example has a more detailed comparison of all models that includes visualizations and prediction error with different metrics. The histogram of the predicted LGD values for the Tobit model does not have a U-shaped distribution, but it ranks well compared to other models.

Beta Regression Model

In a Beta regression model for LGD, the model does not directly predict a single LGD value, it predicts an entire distribution of LGDs (given the predictor values). From that distribution, a value must be determined to predict a single LGD value for a loan, typically the mean of that distribution.

Technically, given the predictor values $X_1, X_{2, \ldots}$ and model coefficients b and c, you can:

- Compute values for the parameters μ (mean) and ϕ (precision, sometimes called the "sample size" of the beta distribution with the following formulas:

$$\mu = \frac{1}{1 + \exp(-b_0 - b_1 X_1 - \cdots)}$$

$$\phi = \exp(c_0 + c_1 X_1 + \cdots)$$

- Compute values for α and β, the typical parameters of the beta distribution, with these formulas:

$$\alpha = \mu \phi$$

$$\beta = (1 - \mu) \phi$$

- Evaluate the density function of the corresponding beta distribution for a given level of LGD, where Γ is the gamma function; see [1 on page 4-111] for details:

$$f_{\text{beta}}(\text{LGD} \mid \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \text{LGD}^{\alpha - 1}(1 - \text{LGD})^{\beta - 1}$$

For fitting the model, once the density function is evaluated, you can update the likelihood function and find the optimal coefficients with a maximum likelihood approach.

For prediction, once the model coefficients are fit, a prediction can be made, typically using the mean of the distribution, that is, the μ parameter, as the predicted LGD value.

Fit a Beta regression model using the `fitLGDModel` function from Risk Management Toolbox™ using the training data. By default, a boundary tolerance of `1e-5` is applied for the LGD response data. For more information on Beta models, see `Beta`.

```
mdlBeta = fitLGDModel(data(TrainingInd,:),'beta','PredictorVars',{'LTV' 'Age' 'Type'},'ResponseVa
disp(mdlBeta)
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-05
              ModelID: "Beta"
```

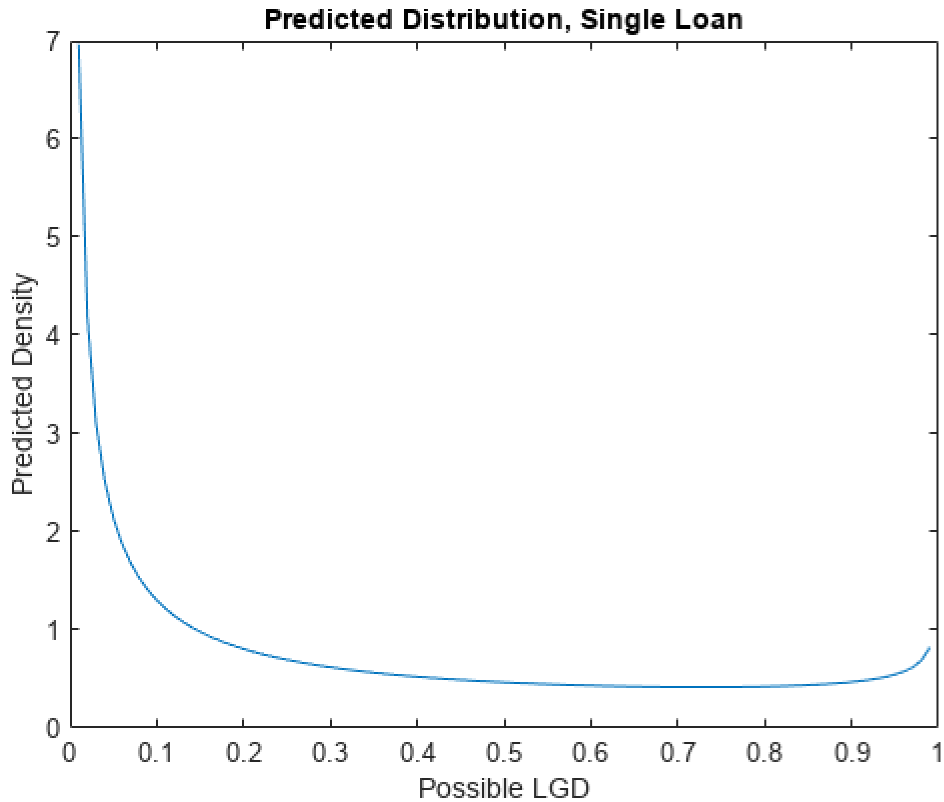
```
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

```
disp(mdlBeta.UnderlyingModel)
```

```
Beta regression model:
     logit(LGD) ~ 1_mu + LTV_mu + Age_mu + Type_mu
     log(LGD) ~ 1_phi + LTV_phi + Age_phi + Type_phi

Estimated coefficients:
                           Estimate       SE        tStat       pValue  
                           ________    ________    _______    __________

    (Intercept)_mu          -1.3772     0.13201    -10.433             0
    LTV_mu                  0.60269     0.15087     3.9947    6.7023e-05
    Age_mu                 -0.47464    0.040264    -11.788             0
    Type_investment_mu      0.45372    0.085143     5.3289     1.094e-07
    (Intercept)_phi        -0.16337     0.12591    -1.2975       0.19462
    LTV_phi                0.055892     0.14719    0.37973       0.70419
    Age_phi                 0.22887    0.040335     5.6743    1.5863e-08
    Type_investment_phi    -0.14102    0.078155    -1.8044      0.071311

Number of observations: 2093
Log-likelihood: -5291.04
```

For prediction, recall that the beta regression links the predictors to an entire beta distribution. For example, suppose that a loan has an `LTV` of `0.7` and an Age of `1.1` years, and it is an investment property. The beta regression model gives us a prediction for the α and β parameters for this loan, and the model predicts that for this loan the range of possible LGD values follows the corresponding beta distribution.

```
Estimate = mdlBeta.UnderlyingModel.Coefficients.Estimate;
NumCols = mdlBeta.UnderlyingModel.NumCoefficients/2;

XExample = [1 0.7 1.1 1];
MuExample = 1/(1+exp(-XExample*Estimate(1:NumCols)));
NuExample = exp(XExample*Estimate(NumCols+1:end));
AlphaExample = MuExample*NuExample;
BetaExample = (1-MuExample)*NuExample;

xDomain = 0.01:0.01:0.99;
pBeta = betapdf(xDomain,AlphaExample,BetaExample);
plot(xDomain,pBeta)
title('Predicted Distribution, Single Loan')
xlabel('Possible LGD')
ylabel('Predicted Density')
```



The shape of the distribution has the U-shaped pattern of the data. However, this is a predicted *distribution* of LGD values for a single loan. This distribution can be very useful for simulation purposes. However, to predict an LGD value for this loan, a method is required that goes from an entire distribution to a single value.

One way to predict would be to randomly draw a value from the previous distribution. This would tend to give predicted values towards the ends of the unit interval, and the overall shape of the distribution for a data set would match the U-shaped patter of observed LGD values. However, even if the *shape* of the distribution looked correct, a random draw from the distribution does not work well for prediction purposes. Two points with the same predictor values would have two different predicted LGDs, which is counterintuitive. Moreover, the prediction error at the observation level could be large, since many loans with small observed LGDs could get random predictions of large LGDs, and vice versa.

To reduce the prediction error at the individual level, the expected value of the beta distribution is typically used to predict. The distribution of predicted values with this approach does not have the expected U-shaped pattern because the mean value tends to be away from the boundaries of the unit interval. However, by using the mean of the beta distribution, all observations with the same predictor values get the same predicted LGDs. Moreover, the mean may not be close to values that are on the ends of the distribution, but the average error might be smaller compared to the random draws from the previous approach.

Use `predict` with Beta models to predict using the mean of the beta distribution. Remember that the expected value of the distribution is the μ parameter, so the mean value prediction is straightforward.

```
dataLGDPredicted.Beta = predict(mdlBeta,data(TestInd,:));
disp(head(dataLGDPredicted))
```

```
    Observed     GroupMeans    Regression     Tobit        Beta   
    _________    __________    __________    ________    ________

    0.0064766     0.066068     0.00091169    0.087889    0.093695
     0.007947      0.12166      0.0036758     0.12432     0.14915
     0.063182      0.25977        0.18774     0.32043     0.35263
            0     0.066068      0.0010877    0.093354    0.096434
      0.10904      0.16489       0.011213     0.16718     0.18858
            0      0.16489       0.041992     0.22382      0.2595
      0.89463      0.16489       0.052947     0.23695     0.26767
            0     0.021776     3.7188e-06    0.010234    0.021315
```

The Model Comparison on page 4-106 section at the end of this example has a more detailed comparison of all models that includes visualizations and prediction error metrics. In particular, the histogram of the predicted LGD values shows that the beta regression approach does not produce a U-shaped distribution. However, this approach does have good performance under the other metrics reported.

Two-Stage Model

Two-stage LGD models separate the case with no losses (LGD equal to `0`) from the cases with actual losses (LGD greater than `0`) and build two models. The stage 1 model is a classification model to predict whether the loan will have positive LGD. The stage 2 model a regression-type model to predict the actual LGD when the LGD is expected to be positive. The prediction is the expected value of the two combined models, which is the product of the probability of having a loss (stage 1 prediction) times the expected LGD value (stage 2 prediction).

In this example, a logistic regression model is used for the stage 1. Stage two is a regression on a logit transformation of the positive LGD data, fitted using `fitLGDModel`. More sophisticated models can be used for stage 1 and stage 2 models, see for example [4 on page 4-112] or [6 on page 4-112]. Also, another extension is to explicitly handle the LGD = `1` boundary. The stage 1 model would produce probabilities of observing an LGD of `0`, an LGD of `1`, and an LGD value strictly between `0` and `1`. The stage 2 model would predict LGD values when the LGD is expected to be strictly between `0` and `1`.

Use `fitglm` to fit the stage 1 model using the training data. The response variable is an indicator with a value of `1` if the observed LGD is positive, and `0` if the observed LGD is zero.

```
IndLGDPositive = data.LGD>0;
data.LGDPositive = IndLGDPositive;
disp(head(data))
```

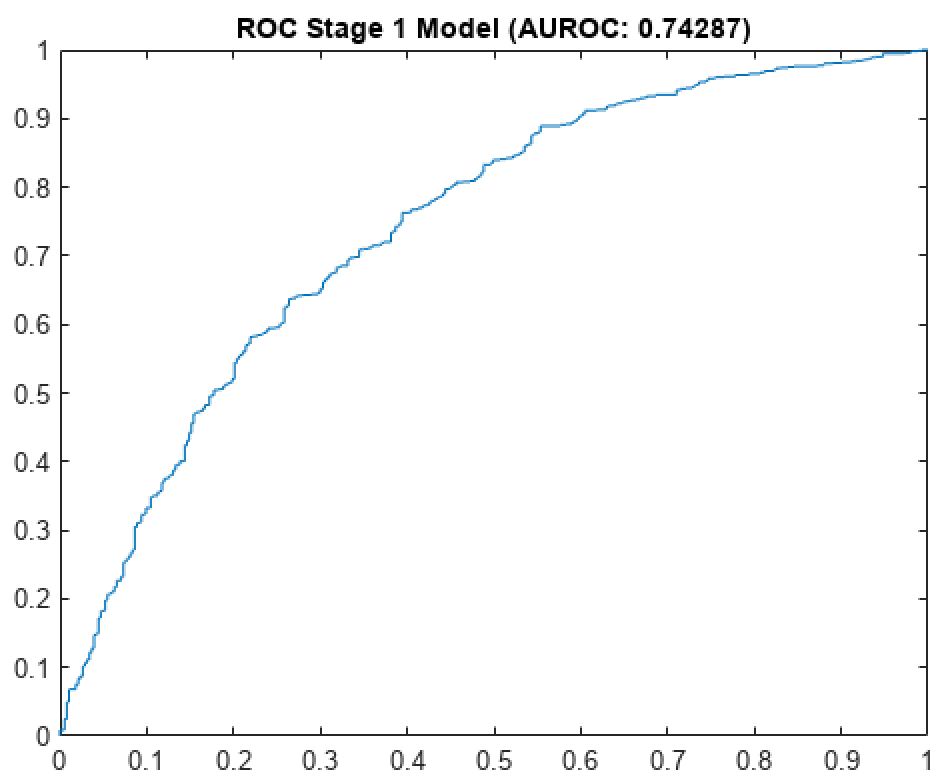
```
      LTV        Age          Type           LGD        LTVDiscretized    AgeDiscretized    LGDTrun
    _______    _______    ___________    _________    ______________    ______________    _______

    0.89101    0.39716    residential     0.032659         high              young           0.032
    0.70176     2.0939    residential      0.43564         high              old              0.43
    0.72078     2.7948    residential    0.0064766         high              old            0.0064
    0.37013      1.237    residential     0.007947         low               young           0.007
    0.36492     2.5818    residential            0         low               old                1e
      0.796     1.5957    residential      0.14572         high              young            0.14
    0.60203     1.1599    residential     0.025688         high              young           0.025
    0.92005    0.50253    investment      0.063182         high              young           0.063
```

```
mdl1 = fitglm(data(TrainingInd,:),"LGDPositive ~ 1 + LTV + Age + Type",'Distribution',"binomial");
disp(mdl1)
```

```
Generalized linear regression model:
    logit(LGDPositive) ~ 1 + LTV + Age + Type
    Distribution = Binomial

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)          1.3157     0.21193     6.2083    5.3551e-10
    LTV                  1.3159     0.25328     5.1954    2.0433e-07
    Age                -0.79597    0.053607    -14.848    7.1323e-50
    Type_investment     0.66784     0.17019     3.9241    8.7051e-05


2093 observations, 2089 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 404, p-value = 2.68e-87
```

A receiver operating characteristic (ROC) curve plot for the stage 1 model is commonly reported using test data, as well as the area under the ROC curve (AUROC or simply AUC).

```
PredictedProbLGDPositive = predict(mdl1,data(TestInd,:));
[x,y,~,AUC] = perfcurve(data.LGDPositive(TestInd),PredictedProbLGDPositive,1);
plot(x,y)
title(sprintf('ROC Stage 1 Model (AUROC: %g)',AUC))
```



Fit the stage 2 model with `fitLGDModel` using only the training data with a positive LGD. This is the same type of model used earlier in the Regression on page 4-100 section, however, this time it is fitted using only observations from the training data with positive LGDs.

```
dataLGDPositive = data(TrainingInd&IndLGDPositive,{'LTV','Age','Type','LGD'});
mdl2 = fitLGDModel(dataLGDPositive,'regression');
disp(mdl2.UnderlyingModel)
```

```
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)         -2.9083     0.27538    -10.561    3.2039e-25
    LTV                  1.3883     0.31838     4.3604     1.384e-05
    Age                -0.46116    0.081939    -5.6281    2.1608e-08
    Type_investment     0.78223     0.18096     4.3226    1.6407e-05


Number of observations: 1546, Error degrees of freedom: 1542
Root Mean Squared Error: 2.8
R-squared: 0.0521,  Adjusted R-Squared: 0.0503
F-statistic vs. constant model: 28.3, p-value = 8.73e-18
```

Perform prediction on the test data using the combined models for stage 1 and stage 2. The predicted LGD is the product of the probability of observing a positive LGD from the stage 1 model times the expected LGD value predicted by the stage 2 model. Recall that regression models fitted using `fitLGDModel` apply the inverse transformation during prediction, so the predicted value is a valid LGD value.

```
PredictedLGDPositive = predict(mdl2,data(TestInd,:));

% PredictedProbLGDPositive is computed before the ROC curve above
% Final LGD prediction is the product of stage 1 and stage 2 predictions
dataLGDPredicted.TwoStage = PredictedProbLGDPositive.*PredictedLGDPositive;

disp(head(dataLGDPredicted))
```

```
    Observed     GroupMeans     Regression      Tobit       Beta       TwoStage 
    _________    __________    __________    ________    ________    _________

    0.0064766     0.066068     0.00091169    0.087889    0.093695     0.020038
     0.007947      0.12166      0.0036758     0.12432     0.14915     0.034025
     0.063182      0.25977        0.18774     0.32043     0.35263       0.2388
            0     0.066068      0.0010877    0.093354    0.096434     0.022818
      0.10904      0.16489       0.011213     0.16718     0.18858     0.060072
            0      0.16489       0.041992     0.22382      0.2595     0.097685
      0.89463      0.16489       0.052947     0.23695     0.26767      0.11142
            0     0.021776     3.7188e-06    0.010234    0.021315    0.0003689
```

The Model Comparison on page 4-106 section at the end of this example has a more detailed comparison of all models that includes visualizations and prediction error metrics. This approach also ranks well compared to other models and the histogram of the predicted LGD values shows high frequencies near `0`.

Model Comparison

To evaluate the performance of LGD models, different metrics are commonly used. One metric is the R-squared of the linear fit regressing the observed LGD values on the predicted values. A second metric is some correlation or rank order statistic; this example uses the Spearman correlation. For prediction error, root mean squared error (RMSE) is a common metric. Also, a simple metric sometimes reported is the difference between the mean LGD value in the data and the mean LGD value of the predictions.

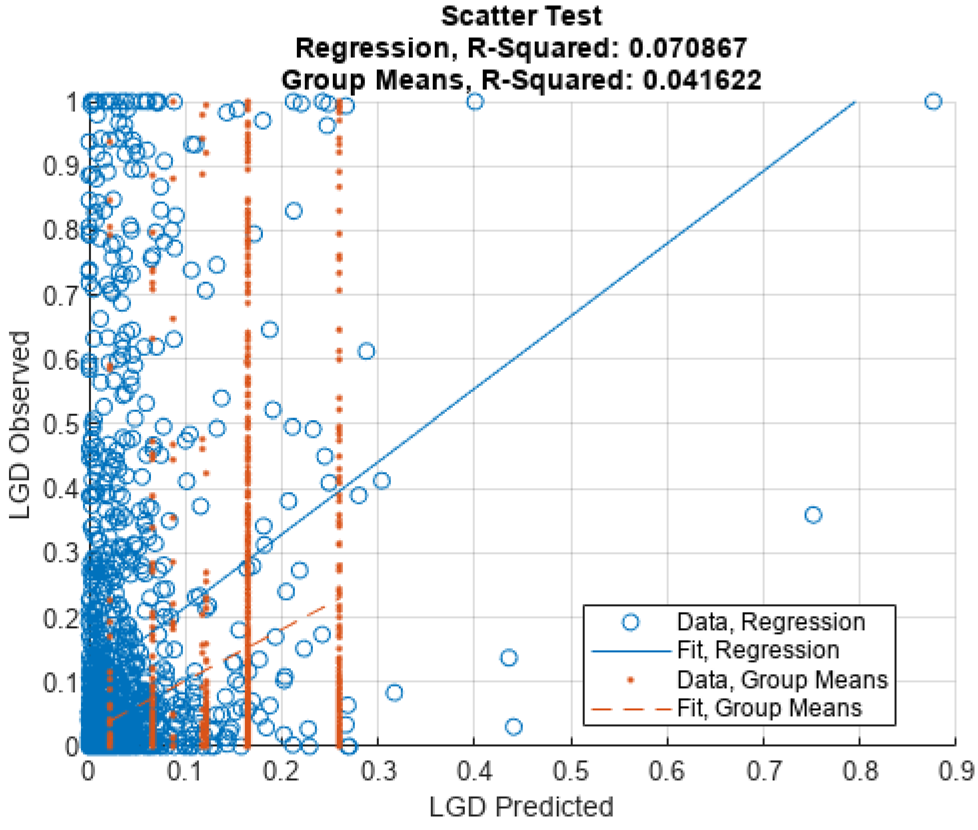
The Regression, Tobit, and Beta models directly support these metrics with the `modelCalibration` function, including comparing against a reference model. For example, here is a report of these metrics for the regression model, passing the predictions of the simple nonparametric model as reference.

```
CalMeasure = modelCalibration(mdlRegression,data(TestInd,:),'DataID','Test','ReferenceLGD',dataL
disp(CalMeasure)
```

	RSquared	RMSE	Correlation	SampleMeanError
Regression, Test	0.070867	0.25988	0.42152	0.10759
Group Means, Test	0.041622	0.2406	0.33807	-0.0078124

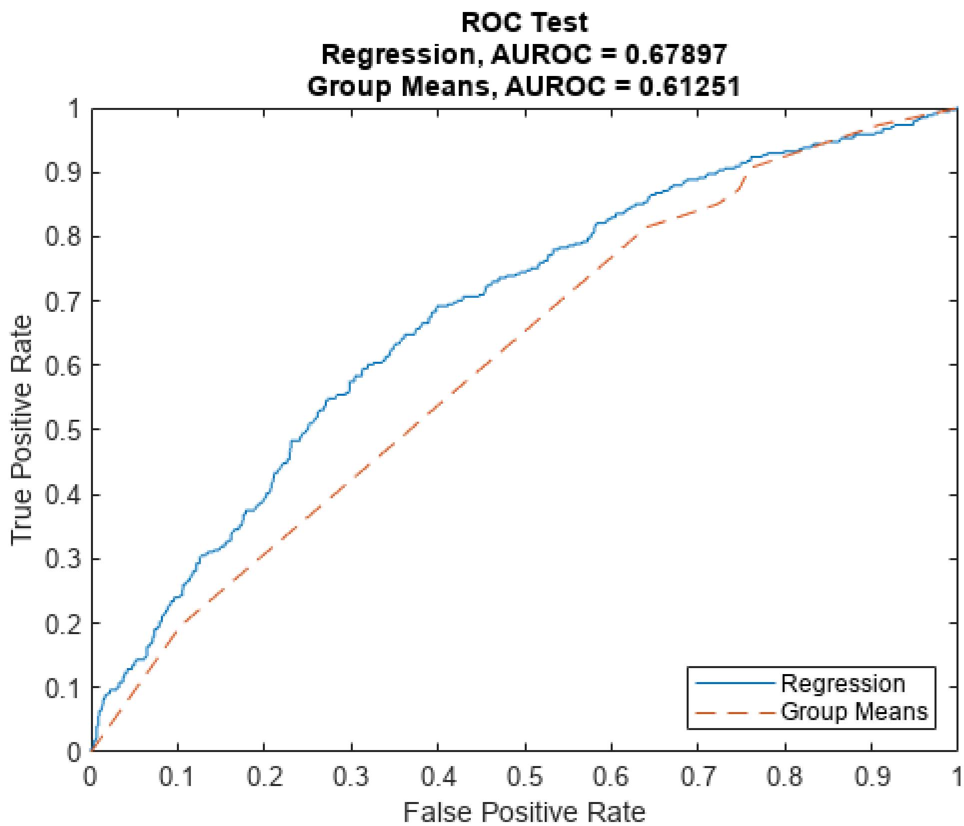
A visualization is also directly supported with `modelCalibrationPlot`.

```
modelCalibrationPlot(mdlRegression,data(TestInd,:),'DataID','Test','ReferenceLGD',dataLGDPredicte
```



In addition, `Regression`, `Tobit` and `Beta` models support model discrimination tools, with the `modelDiscrimination` and `modelDiscriminationPlot` functions. For model discrimination, the LGD data is discretized (high LGD vs. low LGD) and the ROC curve and the corresponding AUROC are computed in a standard way. For more information, see `modelDiscrimination` and `modelDiscriminationPlot`. For example, here is the ROC curve for the regression model, with the ROC curve of the nonparametric model as reference.

```
modelDiscriminationPlot(mdlRegression,data(TestInd,:),'DataID','Test','ReferenceLGD',dataLGDPred
```



The rest of this model validation section works with the predicted LGD values from all the models to compute the metrics mentioned above (R-squared, Spearman correlation, RMSE and sample mean error). It also shows a scatter plot, a histogram, and a box plot to further analyze the performance of the models.

The four metrics are reported below, sorted by decreasing R-squared values.

```
ModelNames = dataLGDPredicted.Properties.VariableNames(2:end); % Remove 'Observed'
NumModels = length(ModelNames);

SampleMeanError = zeros(NumModels,1);
RSquared = zeros(NumModels,1);
Spearman = zeros(NumModels,1);
RMSE = zeros(NumModels,1);
lmAll = struct;

meanLGDTest = mean(dataLGDPredicted.Observed);
```

```
for ii=1:NumModels

    % R-squared, and store linear model fit for visualization section
    Formula = ['Observed ~ 1 + ' ModelNames{ii}];
    lmAll.(ModelNames{ii}) = fitlm(dataLGDPredicted,Formula);
    RSquared(ii) = lmAll.(ModelNames{ii}).Rsquared.Ordinary;

    % Spearman correlation
    Spearman(ii) = corr(dataLGDPredicted.Observed,dataLGDPredicted.(ModelNames{ii}),'type','Spea

    % Root mean square error
    RMSE(ii) = sqrt(mean((dataLGDPredicted.Observed-dataLGDPredicted.(ModelNames{ii})).^2));

    % Sample mean error
    SampleMeanError(ii) = meanLGDTest-mean(dataLGDPredicted.(ModelNames{ii}));

end

PerformanceMetrics = table(RSquared,Spearman,RMSE,SampleMeanError,'RowNames',ModelNames);
PerformanceMetrics = sortrows(PerformanceMetrics,'RSquared','descend');
disp(PerformanceMetrics)
```

```
                  RSquared    Spearman     RMSE      SampleMeanError
                  ________    ________    _______    _______________

    TwoStage      0.090814    0.41987     0.24197        0.060619
    Tobit          0.08527    0.42217     0.23712       -0.034412
    Beta          0.080804    0.41557     0.24112       -0.052396
    Regression    0.070867    0.42152     0.25988         0.10759
    GroupMeans    0.041622    0.33807      0.2406      -0.0078124
```

For the particular training vs. test partition used in this example, the two-stage model has the highest R-squared, although for other partitions, Tobit has the highest R-squared value. Even though the group means approach does not have a high R-squared value, it typically has the smallest sample mean error (mean of predicted LGD values minus mean LGD in the test data). The group means are also competitive for the RMSE metric.

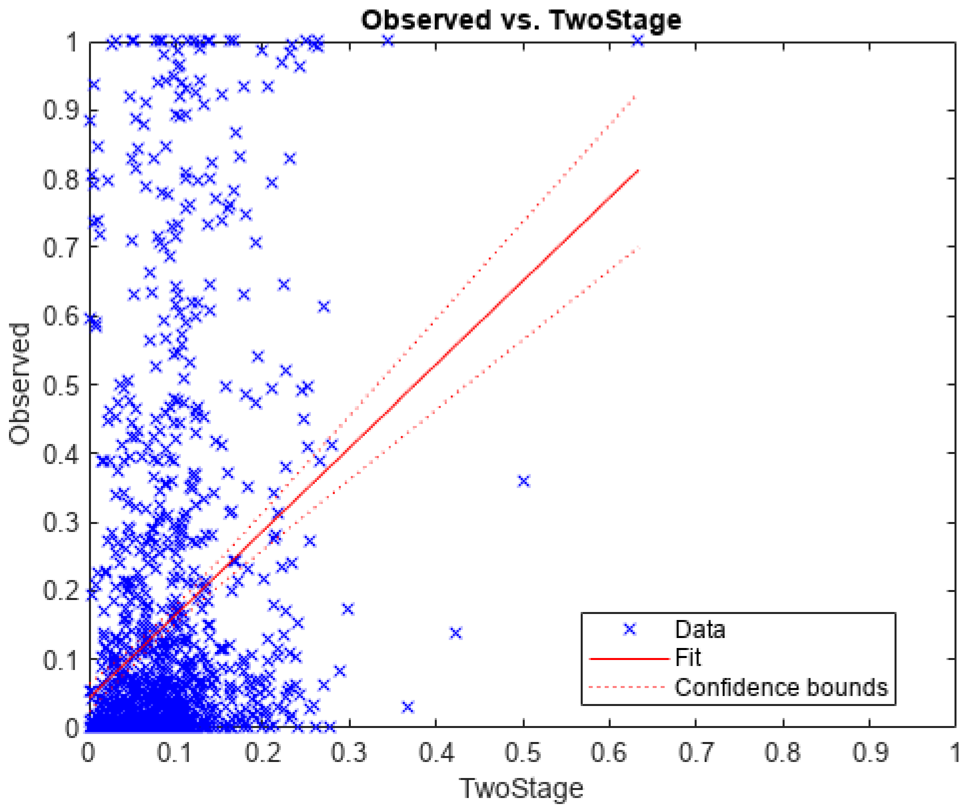
Report the model performance one approach at a time, including visualizations. Display the metrics for the selected model.

```
ModelSelected = Two Stage ;
disp(PerformanceMetrics(ModelSelected,:))
```

```
                RSquared    Spearman     RMSE      SampleMeanError
                ________    ________    _______    _______________

    TwoStage    0.090814    0.41987     0.24197       0.060619
```

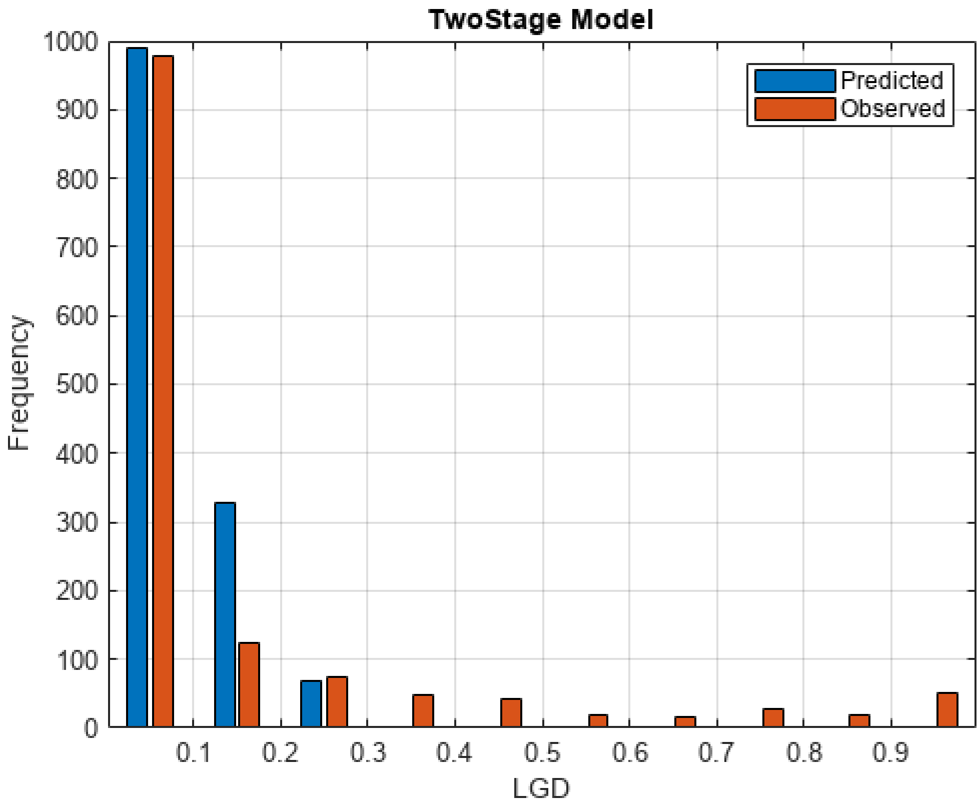
Plot the regression fit (observed LGD vs. predicted LGD), which is a common visual tool to assess the model performance. This is essentially the same visualization as the `modelCalibrationPlot` function shown above, but using the `plot` function of the fitted linear models. The R-squared reported above is the R-squared of this regression. The plot shows a significant amount of error for all models. A good predictive model would have the points located mostly along the diagonal, and not be scattered all over the unit square. However, the metrics above do show some differences in predictive performance for different models that can be important in practice.

```
plot(lmAll.(ModelSelected))
xlim([0 1])
ylim([0 1])
```



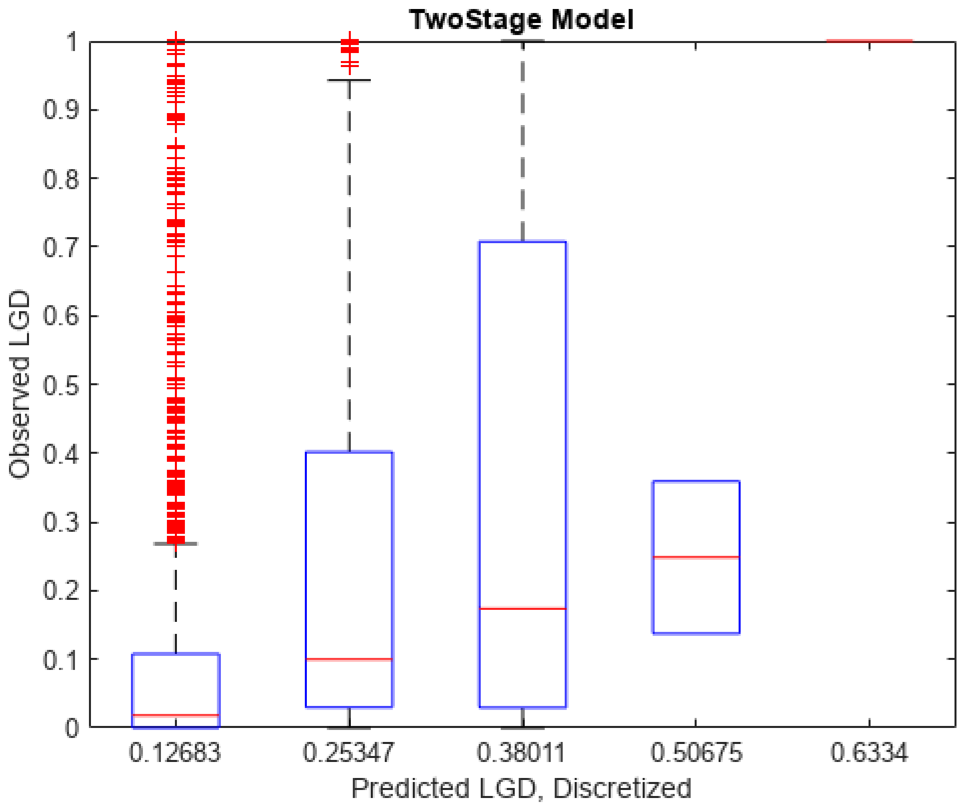
Compare the histograms of the predicted and observed LGD values. For some models, the distribution of predicted values shows high frequencies near zero, similar to the U shape of the observed LGD distribution. However, matching the shape of the distribution does not mean high accuracy at the level of individual predictions; some models show better prediction error even though their histogram does not have a U shape.

```
LGDEdges = 0:0.1:1; % Ten bins to better show the distribution shape
y1 = histcounts(dataLGDPredicted.(ModelSelected),LGDEdges);
y2 = histcounts(dataLGDPredicted.Observed,LGDEdges);
bar((LGDEdges(1:end-1)+LGDEdges(2:end))/2,[y1; y2])
title(strcat(ModelSelected,' Model'))
ylabel('Frequency')
xlabel('LGD')
legend('Predicted','Observed')
grid on
```



Show the box plot of the observed LGD values for different ranges of predicted LGD values. A coarser discretization (five bins only) smooths some noise out and better summarizes the underlying relationship. Ideally, the median (red horizontal line in the middle) should have a monotonic trend and be clearly different from one level to the next. Tall boxes also mean that there is a significant amount of error around the predicted values, which in some cases may be due to very few observations in that level. For a good predictive model, the boxes should be short and be located near the diagonal as you move from one level to the next.

```
LGDEdges = linspace(min(dataLGDPredicted.(ModelSelected)),max(dataLGDPredicted.(ModelSelected)),6
LGDDiscretized = discretize(dataLGDPredicted.(ModelSelected),LGDEdges,'Categorical',string(LGDEdg
boxplot(dataLGDPredicted.Observed,LGDDiscretized)
ylim([0 1])
title(strcat(ModelSelected,' Model'))
xlabel('Predicted LGD, Discretized')
ylabel('Observed LGD')
```



Summary

This example shows multiple approaches for LGD modeling and prediction. The Regression, Tobit, and Beta models (including the regression model of the second stage in the two-stage model) are fitted using the `fitLGDModel` function from Risk Management Toolbox.

The workflow in this example can be adapted to further analyze the models discussed here or to implement and validate other modeling approaches. This example can be extended to perform a more thorough comparison of LGD models (see for example [3 on page 4-112] and [4 on page 4-112]).

The example can also be extended to perform a cross-validation analysis to either benchmark alternative models or to fine-tune hyperparameters. For example, better cut off points for the group means could be selected using cross-validation, or alternative transformations of the LGD response values (logit, probit) could be benchmarked to select the one with the best performance. This example can also be a starting point to perform a backtesting analysis using out-of-time data; see for example [5 on page 4-112].

References

[1] Baesens, B., D. Rosch, and H. Scheule. *Credit Risk Analytics*. Wiley, 2016.

[2] Johnston Ross, E., and L. Shibut. "What Drives Loss Given Default? Evidence from Commercial Real Estate Loans at Failed Banks." *Federal Deposit Insurance Corporation, Center for Financial Research*, Working Paper 2015-03, March 2015.

[3] Li, P., X. Zhang, and X. Zhao. "Modeling Loss Given Default. "*Federal Deposit Insurance Corporation, Center for Financial Research*, Working Paper 2018-03, July 2018.

[4] Loterman, G., I. Brown, D. Martens, C. Mues, and B. Baesens. "Benchmarking Regression Algorithms for Loss Given Default Modeling." *International Journal of Forecasting.* Vol. 28, No.1, pp. 161–170, 2012.

[5] Loterman, G., M. Debruyne, K. Vanden Branden, T. Van Gestel, and C. Mues. "A Proposed Framework for Backtesting Loss Given Default Models." *Journal of Risk Model Validation.* Vol. 8, No. 1, pp. 69-90, March 2014.

[6] Tanoue, Y., and S. Yamashita. "Loss Given Default Estimation: A Two-Stage Model with Classification Tree-Based Boosting and Support Vector Logistic Regression." *Journal of Risk.* Vol. 21 No. 4, pp. 19-37, 2019.

Compare Logistic Model for Lifetime PD to Champion Model

This example shows how to compare a new `Logistic` model for lifetime PD against a "champion" model.

Load Data

Load the portfolio data, which includes loan and macro information.

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);
disp(head(data))
```

```
    ID     ScoreGroup     YOB     Default     Year      GDP      Market
    __    ___________    ___     _______     ____     _____     ______

    1       Low Risk      1          0        1997      2.72       7.61
    1       Low Risk      2          0        1998      3.57      26.24
    1       Low Risk      3          0        1999      2.86       18.1
    1       Low Risk      4          0        2000      2.43       3.19
    1       Low Risk      5          0        2001      1.26     -10.51
    1       Low Risk      6          0        2002     -0.59     -22.95
    1       Low Risk      7          0        2003      0.63       2.78
    1       Low Risk      8          0        2004      1.85       9.48
```

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Fit Logistic Model

For this example, fit a new `Logistic` model using only score group information but no age information. First, you can validate this model in a standalone fashion. For more information, see “Basic Lifetime PD Model Validation” on page 4-129.

Age information is important in this data set. The new model does not perform as well as the champion model (which includes age, score group, and macro vars).

Fit a new `Logistic` model using `fitLifetimePDModel`.

```
ModelType = "logistic";
pdModel = fitLifetimePDModel(data(TrainDataInd,:),ModelType,...
   'ModelID','LogisticNoAge',...
   'IDVar','ID',...
   'LoanVars','ScoreGroup',...
   'MacroVars',{'GDP','Market'},...
   'ResponseVar','Default');
disp(pdModel)
```

```
  Logistic with properties:

            ModelID: "LogisticNoAge"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: ""
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Compare Performance of the Logistic Model to Champion Model

To compare the new `Logistic` model to a champion model, you need access to the predictions of the champion model. The champion model might even have different predictors, so the mapping between the data being used and the exact inputs of the champion model might require an intermediate preprocessing step. This example assumes that you have a black-box tool to get the predictions from the champion model.

Compare the model performance for both models using `modelDiscrimination`.

```
DataSetChoice = Testing ;
if DataSetChoice=="Training"
    Ind = TrainDataInd;
else
    Ind = TestDataInd;
end

ChampionPD = getChampionModelPDs(data(Ind,:));

[DiscMeasure,DiscData] = modelDiscrimination(pdModel,data(Ind,:),'ShowDetails',true,'DataID',Data
   'ReferencePD',ChampionPD,'ReferenceID',"Champion");
disp(DiscMeasure)
```

```
                             AUROC      Segment      SegmentCount
                            _______    __________    ____________

    LogisticNoAge, Testing  0.66503    "all_data"     2.5863e+05
    Champion, Testing       0.70018    "all_data"     2.5863e+05
```

```
disp(head(DiscData))
```

```
        ModelID            X           Y           T
    _______________    ________    ________    ________

    "LogisticNoAge"           0           0     0.02287
    "LogisticNoAge"     0.04673    0.090978     0.02287
    "LogisticNoAge"    0.064656     0.14922    0.022711
    "LogisticNoAge"     0.10982     0.22764    0.020553
    "LogisticNoAge"     0.14421       0.311    0.018483
    "LogisticNoAge"     0.19237     0.41454     0.01722
    "LogisticNoAge"     0.23558     0.43738    0.014125
    "LogisticNoAge"     0.27979     0.52037    0.012812
```

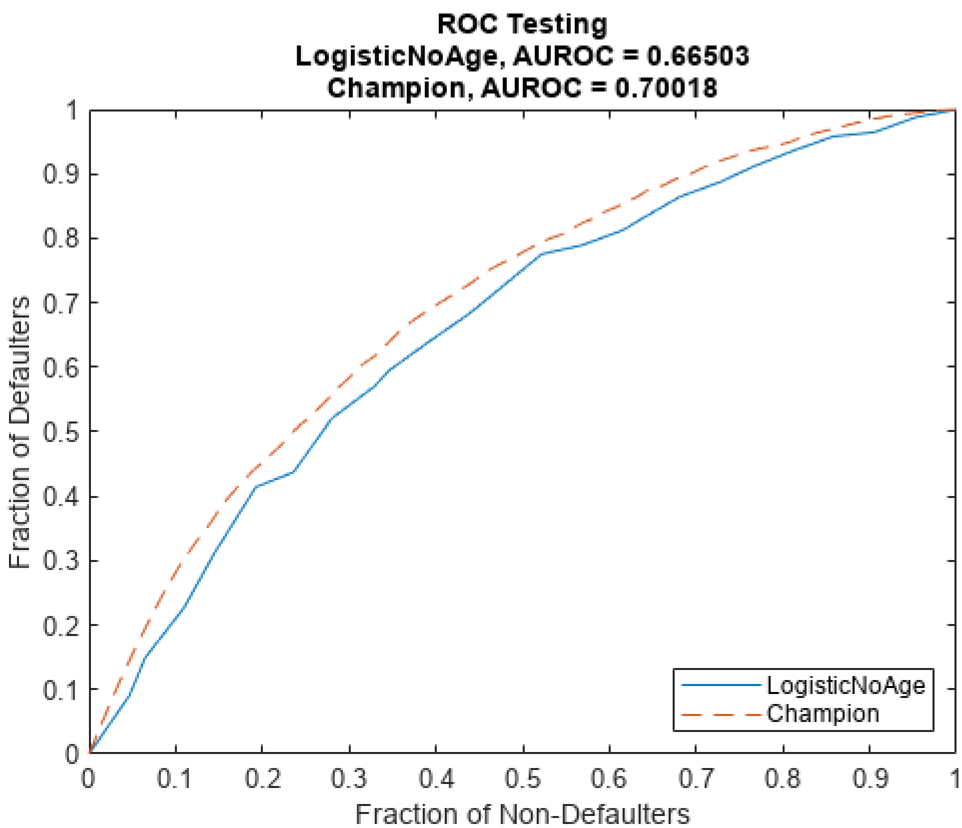
```
disp(tail(DiscData))
```

```
     ModelID         X          Y          T
    __________    _______    _______    __________
```

```
    "Champion"    0.88743    0.98021     0.0032242
    "Champion"    0.90293    0.98477     0.0025583
    "Champion"    0.91884    0.98896     0.0023801
    "Champion"    0.93303    0.99239     0.0018756
    "Champion"    0.94995    0.99391     0.0017711
    "Champion"    0.96705    0.99695     0.0016436
    "Champion"    0.98295    0.99886     0.0012847
    "Champion"          1          1    0.00086887
```

Use `modelDiscriminationPlot` to plot the ROC.

```
modelDiscriminationPlot(pdModel,data(Ind,:),'DataID',DataSetChoice,...
   'ReferencePD',ChampionPD,'ReferenceID',"Champion");
```




```
[DiscMeasure,DiscData] = modelDiscrimination(pdModel,data(Ind,:),'ShowDetails',true,'SegmentBy',
   'ReferencePD',ChampionPD,'ReferenceID',"Champion");
disp(DiscMeasure)
```

	AUROC	Segment	SegmentCount
LogisticNoAge, YOB=1, Testing	0.64879	1	38728
Champion, YOB=1, Testing	0.64972	1	38728
LogisticNoAge, YOB=2, Testing	0.65699	2	37812
Champion, YOB=2, Testing	0.66496	2	37812
LogisticNoAge, YOB=3, Testing	0.63508	3	36973
Champion, YOB=3, Testing	0.64774	3	36973

```
    LogisticNoAge, YOB=4, Testing    0.62656       4          36418
    Champion, YOB=4, Testing         0.66204       4          36418
    LogisticNoAge, YOB=5, Testing     0.6205       5          35818
    Champion, YOB=5, Testing         0.65439       5          35818
    LogisticNoAge, YOB=6, Testing    0.61739       6          35384
    Champion, YOB=6, Testing         0.63156       6          35384
    LogisticNoAge, YOB=7, Testing    0.64016       7          24730
    Champion, YOB=7, Testing         0.63117       7          24730
    LogisticNoAge, YOB=8, Testing    0.63339       8          12764
    Champion, YOB=8, Testing         0.63339       8          12764
```

```
disp(head(DiscData))
```

```
        ModelID         YOB       X          Y            T
    _______________    ___    _______    _______    _________

    "LogisticNoAge"     1           0          0     0.022711
    "LogisticNoAge"     1     0.12062    0.22401     0.022711
    "LogisticNoAge"     1     0.23459    0.41435     0.018483
    "LogisticNoAge"     1     0.33329    0.59151      0.01722
    "LogisticNoAge"     1     0.45578    0.69107      0.01151
    "LogisticNoAge"     1      0.5683    0.77452     0.009347
    "LogisticNoAge"     1     0.67031    0.84919    0.0087028
    "LogisticNoAge"     1     0.78943     0.9063    0.0064814
```

```
disp(tail(DiscData))
```

```
        ModelID         YOB       X         Y           T
    _______________    ___    _______    ______    __________

    "LogisticNoAge"     8           0         0      0.014125
    "LogisticNoAge"     8     0.31762    0.5625      0.014125
    "LogisticNoAge"     8     0.65751    0.8125     0.0071273
    "LogisticNoAge"     8           1         1     0.0040058
    "Champion"          8           0         0     0.0040291
    "Champion"          8     0.31762    0.5625     0.0040291
    "Champion"          8     0.65751    0.8125     0.0017711
    "Champion"          8           1         1    0.00086887
```

Compare Calibration Against Champion Model

Compare the calibration of the two models with `modelCalibration`.

```
GroupingVar = YOB;
[CalMeasure,CalData] = modelCalibration(pdModel,data(Ind,:),GroupingVar,'DataID',DataSetChoice,.
   'ReferencePD',ChampionPD,'ReferenceID',"Champion");
disp(CalMeasure)
```

```
                                                RMSE
                                             __________

    LogisticNoAge, grouped by YOB, Testing    0.0031021
    Champion, grouped by YOB, Testing        0.00046476
```

```
disp(head(CalData))
```

```
      ModelID       YOB       PD        GroupCount
    __________    ___    _________    __________
```

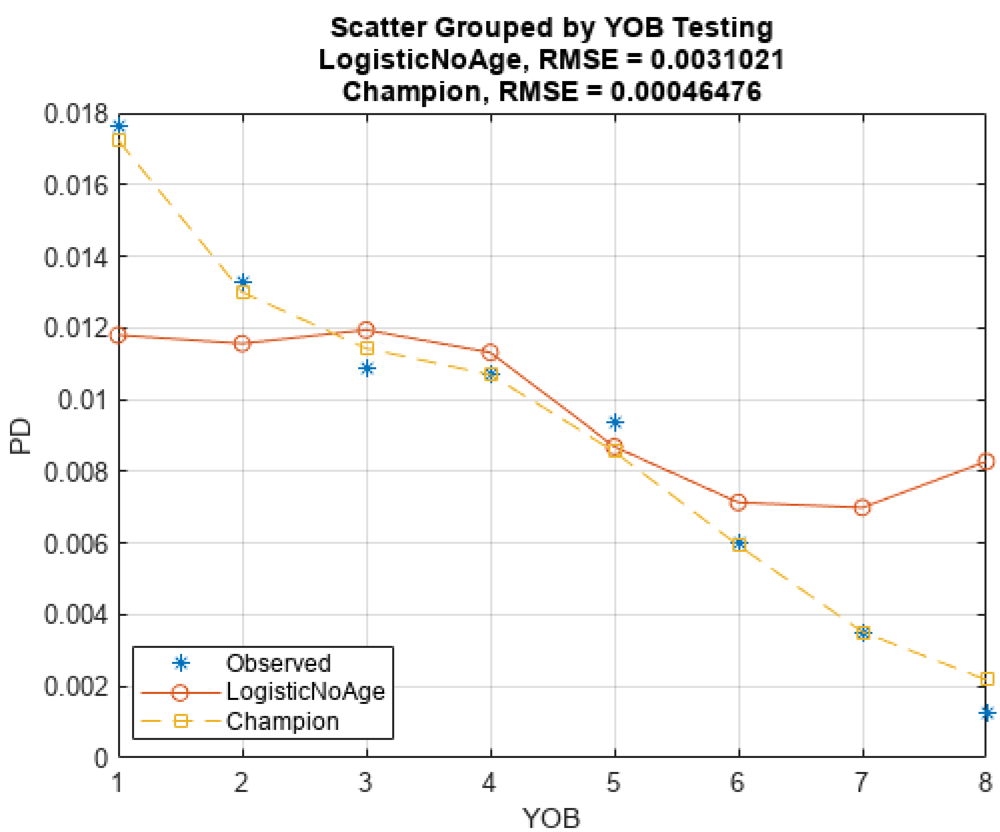
```
    "Observed"     1       0.017636      38728
    "Observed"     2       0.013303      37812
    "Observed"     3       0.010846      36973
    "Observed"     4       0.010709      36418
    "Observed"     5      0.0093528      35818
    "Observed"     6      0.0060197      35384
    "Observed"     7      0.0034776      24730
    "Observed"     8      0.0012535      12764
```

```
disp(tail(CalData))
```

```
     ModelID       YOB        PD         GroupCount
    __________    ___     _________     __________

    "Champion"     1       0.017244      38728
    "Champion"     2       0.012999      37812
    "Champion"     3       0.011428      36973
    "Champion"     4       0.010693      36418
    "Champion"     5      0.0085574      35818
    "Champion"     6       0.005937      35384
    "Champion"     7      0.0035193      24730
    "Champion"     8      0.0021802      12764
```

Use `modelCalibrationPlot` to visualize the model calibration.

```
modelCalibrationPlot(pdModel,data(Ind,:),GroupingVar,'DataID',DataSetChoice,...
   'ReferencePD',ChampionPD,'ReferenceID',"Champion");
```



```
[CalMeasure,CalData] = modelCalibration(pdModel,data(Ind,:),["YOB","ScoreGroup"],'DataID',DataSe
   'ReferencePD',ChampionPD,'ReferenceID',"Champion");
disp(CalMeasure)
```

```
                                                     RMSE
                                                  _________

    LogisticNoAge, grouped by YOB, ScoreGroup, Testing    0.0036974
    Champion, grouped by YOB, ScoreGroup, Testing         0.0010716
```

```
disp(head(CalData))
```

```
     ModelID      YOB     ScoreGroup        PD        GroupCount
    __________    ___    ___________    _________    __________

    "Observed"     1     High Risk       0.030877      13084
    "Observed"     1     Medium Risk     0.013541      12998
    "Observed"     1     Low Risk       0.0081449      12646
    "Observed"     2     High Risk       0.022838      12567
    "Observed"     2     Medium Risk     0.012376      12767
    "Observed"     2     Low Risk       0.0046482      12478
    "Observed"     3     High Risk       0.017651      12067
    "Observed"     3     Medium Risk    0.0092652      12520
```

```
unstack(CalData,'PD','ModelID')
```

```
ans=24×6 table
    YOB     ScoreGroup     GroupCount    Champion     LogisticNoAge    Observed 
    ___    ___________    __________    _________    _____________    _________

     1     High Risk        13084        0.028165       0.019641       0.030877
     1     Medium Risk      12998        0.014833      0.0099388       0.013541
     1     Low Risk         12646        0.008422      0.0055911      0.0081449
     2     High Risk        12567         0.02167       0.019337       0.022838
     2     Medium Risk      12767        0.011123      0.0098141       0.012376
     2     Low Risk         12478       0.0061856      0.0055194      0.0046482
     3     High Risk        12067        0.019285       0.020139       0.017651
     3     Medium Risk      12520       0.0098085       0.010179      0.0092652
     3     Low Risk         12386       0.0054096      0.0057356       0.005813
     4     High Risk        11798        0.018136       0.019175       0.018562
     4     Medium Risk      12325       0.0091921      0.0096563      0.0094929
     4     Low Risk         12295       0.0050562      0.0054292       0.004392
     5     High Risk        11481        0.014818       0.014806       0.016288
     5     Medium Risk      12120       0.0072853       0.007454      0.0080033
     5     Low Risk         12217       0.0039358      0.0041822      0.0041745
     6     High Risk        11250         0.01049       0.012153      0.0096889
      ⋮
```

Compare Two Models Under Development

You can also compare two new models under development.

```
pdModelTTC = fitLifetimePDModel(data(TrainDataInd,:),"probit",...
   'ModelID','ProbitTTC',...
   'AgeVar','YOB',...
   'IDVar','ID',...
   'LoanVars','ScoreGroup',...
```

```
   'ResponseVar','Default',...
   'Description',"TTC model, no macro variables, probit.");
disp(pdModelTTC)
```

```
  Probit with properties:

            ModelID: "ProbitTTC"
        Description: "TTC model, no macro variables, probit."
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ""
        ResponseVar: "Default"
```

```
pdModelTTC.UnderlyingModel
```

```
ans = 
Compact generalized linear regression model:
    probit(Default) ~ 1 + ScoreGroup + YOB
    Distribution = Binomial

Estimated Coefficients:
                              Estimate        SE         tStat       pValue   
                              _________    _________    _______    ___________

    (Intercept)                 -1.8275     0.013636    -134.02              0
    ScoreGroup_Medium Risk     -0.26441     0.014158    -18.676     7.7165e-78
    ScoreGroup_Low Risk        -0.46734     0.016327    -28.624     3.371e-180
    YOB                       -0.081761    0.0031333    -26.094    4.2244e-150


388097 observations, 388093 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.7e+03, p-value = 0
```

Compare the calibrations.

```
[CalMeasureTTC,CalDataTTC] = modelCalibration(pdModelTTC,data(Ind,:),["YOB","ScoreGroup"],'DataI
   'ReferencePD',predict(pdModel,data(Ind,:)),'ReferenceID',pdModel.ModelID);
disp(CalMeasureTTC)
```

```
                                                        RMSE   
                                                     _________

    ProbitTTC, grouped by YOB, ScoreGroup, Testing       0.0016726
    LogisticNoAge, grouped by YOB, ScoreGroup, Testing    0.0036974
```

```
unstack(CalDataTTC,'PD','ModelID')
```

ans=*24×6 table*

YOB	ScoreGroup	GroupCount	LogisticNoAge	Observed	ProbitTTC
1	High Risk	13084	0.019641	0.030877	0.028114
1	Medium Risk	12998	0.0099388	0.013541	0.014865
1	Low Risk	12646	0.0055911	0.0081449	0.0087364
2	High Risk	12567	0.019337	0.022838	0.023239
2	Medium Risk	12767	0.0098141	0.012376	0.012053

```
 2     Low Risk         12478          0.0055194       0.0046482    0.0069786
 3     High Risk        12067           0.020139        0.017651     0.019096
 3     Medium Risk      12520           0.010179       0.0092652    0.0097145
 3     Low Risk         12386          0.0057356        0.005813    0.0055406
 4     High Risk        11798           0.019175        0.018562     0.015599
 4     Medium Risk      12325          0.0096563       0.0094929    0.0077825
 4     Low Risk         12295          0.0054292        0.004392    0.0043722
 5     High Risk        11481           0.014806        0.016288     0.012666
 5     Medium Risk      12120           0.007454       0.0080033    0.0061971
 5     Low Risk         12217          0.0041822       0.0041745    0.0034292
 6     High Risk        11250           0.012153       0.0096889     0.010223
   ⋮
```

Black-Box Champion Prediction Function

```
function PD = getChampionModelPDs(data)

m = load('LifetimeChampionModel.mat');
PD = predict(m.pdModel,data);

end
```

See Also

fitLifetimePDModel | predict | predictLifetime | modelDiscrimination | modelCalibration | modelCalibrationPlot | Logistic | Probit | Cox

Related Examples

- “Basic Lifetime PD Model Validation” on page 4-129
- “Expected Credit Loss Computation” on page 4-124
- “Compare Lifetime PD Models Using Cross-Validation” on page 4-121
- “Compare Model Discrimination and Model Calibration to Validate of Probability of Default” on page 4-144
- “Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models” on page 4-75

Compare Lifetime PD Models Using Cross-Validation

This example shows how to compare three lifetime PD models using cross-validation.

Load Data

Load the portfolio data, which includes load and macro information. This is a simulated data set used for illustration purposes.

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);
disp(head(data))
```

```
    ID     ScoreGroup     YOB    Default    Year     GDP     Market
    __    ___________    ___    _______    ____    _____    ______

    1      Low Risk       1         0       1997     2.72      7.61
    1      Low Risk       2         0       1998     3.57     26.24
    1      Low Risk       3         0       1999     2.86      18.1
    1      Low Risk       4         0       2000     2.43      3.19
    1      Low Risk       5         0       2001     1.26    -10.51
    1      Low Risk       6         0       2002    -0.59    -22.95
    1      Low Risk       7         0       2003     0.63      2.78
    1      Low Risk       8         0       2004     1.85      9.48
```

Cross Validation

Because the data is panel data, there are multiple rows for each customer. You set up cross validation partitions over the customer IDs, not over the rows of the data set. In this way, a customer can be in either a training set or a test set, but the rows corresponding to the same customer are not split between training and testing.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

NumFolds = 5;
rng('default'); % for reproducibility
c = cvpartition(nIDs,'KFold',NumFolds);
```

Compare `Logistic`, `Probit`, `Cox` lifetime PD models using the same variables.

```
CVModels = ["logistic";"probit";"cox"];
NumModels = length(CVModels);

AUROC = zeros(NumFolds,NumModels);
RMSE = zeros(NumFolds,NumModels);

for ii=1:NumFolds

   fprintf('Fitting models, fold %d\n',ii);

   % Get indices for ID partition
   TrainIDInd = training(c,ii);
   TestIDInd = test(c,ii);
   % Convert to row indices
   TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
```

```
    TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));

    % For each model, fit with training data, measure with test data
    for jj=1:NumModels
       % Fit model with training data
       pdModel = fitLifetimePDModel(data(TrainDataInd,:),CVModels(jj),...
              'IDVar','ID','AgeVar','YOB','LoanVars','ScoreGroup',...
              'MacroVars',{'GDP','Market'},'ResponseVar','Default');
       % Measure discrimination on test data
       DiscMeasure = modelDiscrimination(pdModel,data(TestDataInd,:));
       AUROC(ii,jj) = DiscMeasure.AUROC;

       % Measure calibration on test data, grouping by YOB (age) and score group
       CalMeasure = modelCalibration(pdModel,data(TestDataInd,:),["YOB" "ScoreGroup"]);
       RMSE(ii,jj) = CalMeasure.RMSE;
    end
end
```

```
Fitting models, fold 1
Fitting models, fold 2
Fitting models, fold 3
Fitting models, fold 4
Fitting models, fold 5
```

Using the discrimination and accuracy measures for the different folds, you can compare the models. In this example, the metrics are displayed. You can also compare the mean AUROC or the mean RMSE by comparing the proportion of times a model is superior regarding discrimination or accuracy. The three models in this example are very comparable.

```
AUROCTable = array2table(AUROC,"RowNames",strcat("Fold ",string(1:NumFolds)),"VariableNames",strc
```

AUROCTable=*5×3 table*

	AUROC_logistic	AUROC_probit	AUROC_cox
Fold 1	0.69558	0.6957	0.69565
Fold 2	0.70265	0.70335	0.70366
Fold 3	0.69055	0.69037	0.69008
Fold 4	0.70268	0.70232	0.70296
Fold 5	0.68784	0.68781	0.68811

```
RMSETable = array2table(RMSE,"RowNames",strcat("Fold ",string(1:NumFolds)),"VariableNames",strcat
```

RMSETable=*5×3 table*

	RMSE_logistic	RMSE_probit	RMSE_cox
Fold 1	0.0019412	0.0020972	0.0020048
Fold 2	0.0011167	0.0011644	0.0011612
Fold 3	0.0011536	0.0011802	0.0012766
Fold 4	0.0010269	0.00097877	0.00099473
Fold 5	0.0015965	0.001485	0.0015829

See Also

fitLifetimePDModel | predict | predictLifetime | modelDiscrimination | modelCalibration | modelCalibrationPlot | Logistic | Probit | Cox

Related Examples

- "Basic Lifetime PD Model Validation" on page 4-129
- "Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
- "Expected Credit Loss Computation" on page 4-124
- "Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
- "Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75

Expected Credit Loss Computation

This example shows how to perform expected credit loss (ECL) computations with `portfolioECL` using simulated loan data, macro scenario data, and an existing lifetime probability of default (PD) model.

Load Data and Model

Load loan data ready for prediction, macro scenario data, and corresponding scenario probabilities.

```
load DataPredictLifetime.mat
disp(LoanData)
```

```
     ID      ScoreGroup      YOB    Year
    ____    _____________    ___    ____

    1304    "Medium Risk"     4     2020
    1304    "Medium Risk"     5     2021
    1304    "Medium Risk"     6     2022
    1304    "Medium Risk"     7     2023
    1304    "Medium Risk"     8     2024
    1304    "Medium Risk"     9     2025
    1304    "Medium Risk"    10     2026
    2067    "Low Risk"        7     2020
    2067    "Low Risk"        8     2021
    2067    "Low Risk"        9     2022
    2067    "Low Risk"       10     2023
```

```
disp(head(MultipleScenarios,10))
```

```
    ScenarioID    Year    GDP     Market
    __________    ____    ____    ______

    "Severe"      2020    -0.9     -5.5
    "Severe"      2021    -0.5     -6.5
    "Severe"      2022     0.2       -1
    "Severe"      2023     0.8      1.5
    "Severe"      2024     1.4        4
    "Severe"      2025     1.8      6.5
    "Severe"      2026     1.8      6.5
    "Severe"      2027     1.8      6.5
    "Adverse"     2020     0.1     -0.5
    "Adverse"     2021     0.2     -2.5
```

```
disp(ScenarioProbabilities)
```

```
                 Probability
                 ___________

    Severe           0.1
    Adverse          0.2
    Baseline         0.3
    Favorable        0.2
    Excellent        0.2
```

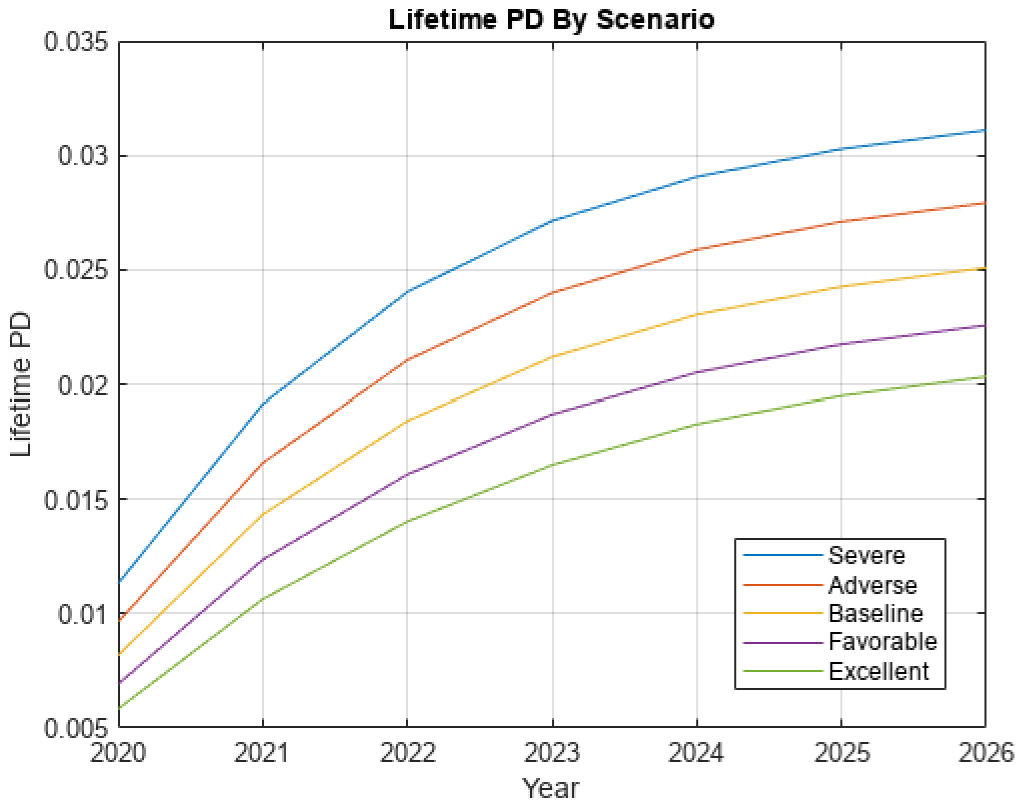
```
load LifetimeChampionModel.mat
disp(pdModel)
```

```
  Probit with properties:

            ModelID: "Champion"
        Description: "A sample model used as champion model for illustration purposes."
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Visualize Lifetime PDs

For ECL computations, only the marginal PDs are required. However, first you can visualize the lifetime PDs.

```
CompanyIDChoice = 1304 ;
CompanyID = str2double(CompanyIDChoice);
IndCompany = LoanData.ID == CompanyID;
Years = LoanData.Year(IndCompany);
NumYears = length(Years);

ScenarioID = unique(MultipleScenarios.ScenarioID,'stable');
NumScenarios = length(ScenarioID);

LifetimePD = zeros(NumYears,NumScenarios);
for ii=1:NumScenarios
   IndScenario = MultipleScenarios.ScenarioID==ScenarioID(ii);
   data = join(LoanData(IndCompany,:),MultipleScenarios(IndScenario,:));
   LifetimePD(:,ii) = predictLifetime(pdModel,data);
end

plot(Years,LifetimePD)
xticks(Years)
grid on
xlabel('Year')
ylabel('Lifetime PD')
title('Lifetime PD By Scenario')
legend(ScenarioID,'Location','best')
```



Compute ECL

The computation of ECL requires a marginal PD values, LGD values, and EAD values, effective interest rate, plus the scenarios and scenario probabilities.

Compute the lifetime ECL using the `portfolioECL` function. The inputs to this function are tables, where the first column is an ID variable that indicates which rows correspond to which loan. Because the projections cover multiple periods for each loan, and the remaining life of different loans may be different, the ID variable is an important input. For each ID, the credit projections must be provided, period-by-period, until the end of the life of each loan. Typically, the marginal PD has a multi-period and multi-scenario size. This example assumes constant LGD and EAD values. This means that the same LGD and EAD is used for all periods, and the LGD and EAD values are not sensitive to the scenarios. Hence, the marginal PD input has multiple rows and columns per ID, whereas the LGD and EAD inputs have one scalar value per ID. To offer flexibility for different input dimensions for marginal PD, LGD, and EAD inputs, these inputs are separated into three separate tables in the syntax of `portfolioECL`.

```
ScenarioID = unique(MultipleScenarios.ScenarioID,'stable');
NumScenarios = length(ScenarioID);
```

Predict marginal PD for each scenario. The `predictLifetime` function is called for the entire portfolio at once, and the marginal PDs for each scenario are stored as columns.

```
MarginalPD = zeros(height(LoanData),NumScenarios);
for ii=1:NumScenarios
   IndScenario = MultipleScenarios.ScenarioID==ScenarioID(ii);
```

```
   data = join(LoanData,MultipleScenarios(IndScenario,:));
   MarginalPD(:,ii) = predictLifetime(pdModel,data,'ProbabilityType','marginal');
end
```

Convert to the required table input format, with the ID column.

```
MarginalPDTable = array2table(MarginalPD);
MarginalPDTable.Properties.VariableNames = ScenarioID;
MarginalPDTable = addvars(MarginalPDTable,LoanData.ID,'Before',1,'NewVariableNames','ID');
disp(MarginalPDTable)
```

```
     ID       Severe        Adverse       Baseline      Favorable     Excellent 
    ____    __________    __________    __________    __________    __________

    1304      0.011316     0.0096361     0.0081783      0.006918     0.0058324
    1304     0.0078277     0.0069482     0.0061554     0.0054425     0.0048028
    1304     0.0048869     0.0044693     0.0040823     0.0037243     0.0033938
    1304     0.0031017     0.0029321     0.0027698     0.0026147     0.0024668
    1304     0.0019309     0.0018923     0.0018538     0.0018153      0.001777
    1304     0.0012157     0.0012197     0.0012233     0.0012264     0.0012293
    1304    0.00082053    0.00082322    0.00082562    0.00082775    0.00082964
    2067     0.0022199      0.001832     0.0015067      0.001235     0.0010088
    2067     0.0014464     0.0012534     0.0010841    0.00093599    0.00080662
    2067     0.0008343    0.00074897    0.00067168    0.00060175    0.00053857
    2067    0.00049107    0.00045839    0.00042769    0.00039887    0.00037183
```

The LGD and EAD table inputs are small tables with one row per ID.

```
UniqueIDs = unique(LoanData.ID,'stable');
NumIDs = length(UniqueIDs);

LGD = 0.55;
LGDTable = table(UniqueIDs, repmat(LGD,NumIDs,1),'VariableNames',{'ID','LGD'});
disp(LGDTable)
```

```
     ID     LGD 
    ____    ____

    1304    0.55
    2067    0.55
```

```
EAD = 100000;
EADTable = table(UniqueIDs, repmat(EAD,NumIDs,1),'VariableNames',{'ID','EAD'});
disp(EADTable)
```

```
     ID      EAD 
    ____    _____

    1304    1e+05
    2067    1e+05
```

For simplicity, assume the same effective interest rate for both loans.

```
EffRate = 0.045;
```

Call the `portfolioECL` function. The first output is the total ECL, or provisions, for the portfolio.

```
[totalECL, ECLByID, ECLByPeriod] = portfolioECL(MarginalPDTable, LGDTable, EADTable, 'InterestRa
    'ScenarioNames',ScenarioID, 'ScenarioProbabilities',ScenarioProbabilities.Probability, 'IDVa
```

```
fprintf('Total portfolio lifetime ECL is: %.2f\n',totalECL)
```

```
Total portfolio lifetime ECL is: 1401.00
```

The second output, `ECLByID`, shows the ECL for each ID. The third output, `ECLByPeriod`, shows the ECL for each period, and each scenario. Use the dropdown to select an ID and display the corresponding ECL information.

```
CompanyIDChoice = 1304;
CompanyID = str2double(CompanyIDChoice);

disp(ECLByID(ECLByID.ID==CompanyID,:))
```

ID	ECL
1304	1217.3

```
disp(ECLByPeriod(ECLByPeriod.ID==CompanyID,:))
```

ID	TimePeriod	Severe	Adverse	Baseline	Favorable	Excellent
1304	1	595.58	507.16	430.44	364.11	306.97
1304	2	394.24	349.95	310.02	274.11	241.9
1304	3	235.53	215.4	196.75	179.5	163.57
1304	4	143.05	135.23	127.75	120.59	113.77
1304	5	85.219	83.517	81.816	80.118	78.429
1304	6	51.346	51.514	51.665	51.798	51.917
1304	7	33.162	33.271	33.368	33.454	33.531

For more information, see the “Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations” on page 4-195 example that shows a detailed workflow for ECL calculations, including the determination of macro scenarios, the use of lifetime PD, LGD and EAD models, and a visualization of credit projections and provisions for each ID to drill down to a loan level.

See Also

`fitLifetimePDModel` | `predict` | `predictLifetime` | `modelDiscrimination` | `modelCalibration` | `modelCalibrationPlot` | `Logistic` | `Probit` | `Cox`

Related Examples

- “Basic Lifetime PD Model Validation” on page 4-129
- “Compare Lifetime PD Models Using Cross-Validation” on page 4-121
- “Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
- “Compare Model Discrimination and Model Calibration to Validate of Probability of Default” on page 4-144
- “Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models” on page 4-75

Basic Lifetime PD Model Validation

This example shows how to perform basic model validation on a lifetime probability of default (PD) model by viewing the fitted model, estimated coefficients, and *p*-values. For more information on model validation, see `modelDiscrimination` and `modelCalibration`.

Load Data

Load the portfolio data.

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Fit Model and Review Model Goodness of Fit

Create training and test datasets to perform a basic model validation.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Fit the model using `fitLifetimePDModel` for a `Logistic`, `Probit`, or `Cox` model.

```
ModelType = probit;
pdModel = fitLifetimePDModel(data(TrainDataInd,:),ModelType,...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
disp(pdModel)
```

```
  Probit with properties:

            ModelID: "Probit"
```

```
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Display the PD model and review the fit statistics, such as the *p*-values.

```
disp(pdModel.UnderlyingModel)
```

```
Compact generalized linear regression model:
    probit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue
                              __________    _________    _______    ___________

    (Intercept)                  -1.6267      0.03811    -42.685              0
    ScoreGroup_Medium Risk      -0.26542      0.01419    -18.704     4.5503e-78
    ScoreGroup_Low Risk         -0.46794     0.016364    -28.595     7.775e-180
    YOB                         -0.11421    0.0049724    -22.969    9.6208e-117
    GDP                        -0.041537     0.014807    -2.8052      0.0050291
    Market                    -0.0029609    0.0010618    -2.7885      0.0052954


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

```
pdModel.UnderlyingModel.Coefficients
```

```
ans=6×4 table
                               Estimate        SE         tStat       pValue
                              __________    _________    _______    ___________

    (Intercept)                  -1.6267      0.03811    -42.685              0
    ScoreGroup_Medium Risk      -0.26542      0.01419    -18.704     4.5503e-78
    ScoreGroup_Low Risk         -0.46794     0.016364    -28.595     7.775e-180
    YOB                         -0.11421    0.0049724    -22.969    9.6208e-117
    GDP                        -0.041537     0.014807    -2.8052      0.0050291
    Market                    -0.0029609    0.0010618    -2.7885      0.0052954
```

See Also

`fitLifetimePDModel` | `predict` | `predictLifetime` | `modelDiscrimination` | `modelCalibration` | `modelCalibrationPlot` | `Logistic` | `Probit` | `Cox`

Related Examples

- "Compare Lifetime PD Models Using Cross-Validation" on page 4-121
- "Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113

Basic Loss Given Default Model Validation

This example shows how to perform basic model validation on a loss given default (LGD) model by viewing the fitted model, estimated coefficients, and *p*-values. For more information on model validation, see `modelDiscrimination` and `modelCalibration`.

Load Data

Load the portfolio data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age          Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Fit Model and Review Model Goodness of Fit

Create training and test datasets to perform a basic model validation.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Fit the model using `fitLifetimePDModel`.

```
ModelType = regression;
lgdModel = fitLGDModel(data(TrainingInd,:),ModelType,...
    'ModelID','Example',...
     'Description','Example LGD regression model.',...
     'PredictorVars',{'LTV' 'Age' 'Type'},...
     'ResponseVar','LGD');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Example"
          Description: "Example LGD regression model."
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying statistical model. The displayed information contains the coefficient estimates, as well as their standard errors, *t*-statistics and *p*-values. The underlying statistical model also shows the number of observations and other fit metrics.

```
lgdModel.UnderlyingModel
```

```
ans = 
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

In the case of the underlying statistical model for a `Regression` model, the underlying model is returned as a compact linear model object. The compact version of the underlying `Regression` model is an instance of the `classreg.regr.CompactLinearModel` class. For more information, see `fitlm` and `CompactLinearModel`.

See Also

`fitLGDModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibartionPlot` | `Regression` | `Tobit`

Related Examples

- “Model Loss Given Default” on page 4-90
- “Compare Tobit LGD Model to Benchmark Model” on page 4-133
- “Compare Loss Given Default Models Using Cross-Validation” on page 4-140

More About

- “Overview of Loss Given Default Models” on page 1-31

Compare Tobit LGD Model to Benchmark Model

This example shows how to compare a `Tobit` model for loss given default (LGD) against a benchmark model.

Load Data

Load the LGD data.

```
load LGDData.mat
disp(head(data))
```

```
      LTV        Age         Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Split the data into training and test sets.

```
NumObs = height(data);

rng('default'); % For reproducibility
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Fit Tobit Model

Fit a `Tobit` LGD model with training data. By default, the last column of the data is used as a response variable and all other columns are used as predictor variables.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'tobit');
disp(lgdModel)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

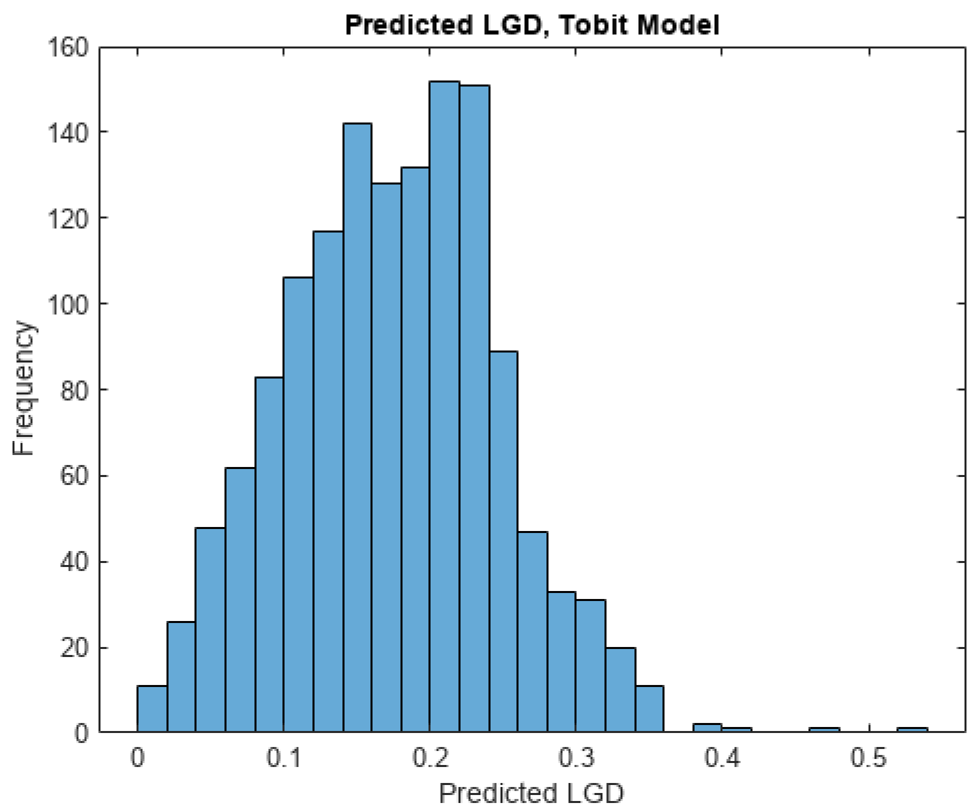
```
disp(lgdModel.UnderlyingModel)
```

```
Tobit regression model:
     LGD = max(0,min(Y*,1))
     Y* ~ 1 + LTV + Age + Type
```

```
Estimated coefficients:
                         Estimate        SE         tStat       pValue  
                         _________    _________    _______    __________

    (Intercept)           0.058257     0.027265     2.1367      0.032737
    LTV                    0.20126     0.031354     6.4189    1.6932e-10
    Age                  -0.095407    0.0072653    -13.132             0
    Type_investment        0.10208     0.018058     5.6531    1.7915e-08
    (Sigma)                0.29288    0.0057036      51.35             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1521
Number of right-censored observations: 25
Log-likelihood: -698.383
```

You can now use this model for prediction or validation. For example, use `predict` to predict LGD on test data and visualize the predictions with a histogram.

```
lgdPredTobit = predict(lgdModel,data(TestInd,:));
histogram(lgdPredTobit)
title('Predicted LGD, Tobit Model')
xlabel('Predicted LGD')
ylabel('Frequency')
```



Create Benchmark Model

In this example, the benchmark model is a lookup table model that segments the data into groups and assigns the mean LGD of the group to all group members. In practice, this common benchmarking approach is easy to understand and use.

The groups in this example are defined using the three predictors. `LTV` is discretized into low and high levels. `Age` is discretized into young and old loans. `Type` already has two levels, namely, residential and investment. The groups are all the combinations of these values (for example, low LTV, young loan, residential, and so on). The number of levels and the specific cutoff points are only for illustration purposes. The benchmark model uses the same predictors as the `Tobit` model in this example, but you can use other variables to define the groups. In fact, the benchmark model could be a black-box model as long as the predicted LGD values are available for the same customers as in this data set.

```
% Add the discretized variables as new colums in the table.
% Discretize the LTV.
LTVEdges = [0 0.5 max(data.LTV)];
data.LTVDiscretized = discretize(data.LTV,LTVEdges,'Categorical',{'low','high'});
% Discretize the Age.
AgeEdges = [0 2 max(data.Age)];
data.AgeDiscretized = discretize(data.Age,AgeEdges,'Categorical',{'young','old'});
% Type is already a categorical variable with two levels.
```

Finding the group means on the training data is effectively the fitting of the model. Note that the group counts are small for some groups. Adding many groups comes with reduced group counts for some groups and more unstable estimates.

```
% Find the group means on training data.
gs = groupsummary(data(TrainingInd,:),{'LTVDiscretized','AgeDiscretized','Type'},'mean','LGD');
disp(gs)
```

```
    LTVDiscretized    AgeDiscretized       Type        GroupCount    mean_LGD
    ______________    ______________    ___________    __________    ________

         low              young         residential       163         0.12166
         low              young         investment         26        0.087331
         low              old           residential       175        0.021776
         low              old           investment         23         0.16379
         high             young         residential      1134         0.16489
         high             young         investment        257         0.25977
         high             old           residential       265        0.066068
         high             old           investment         50         0.11779
```

To predict an LGD for a new observation, you need to find its group and then assign the group mean as the predicted LGD. Use the `findgroups` function, which takes the discretized variables as input. For a completely new data point, the `LTV` and `Age` information needs to be discretized first by using the `discretize` function before you use the `findgroups` function.

```
LGDGroup = findgroups(data(TestInd,{'LTVDiscretized' 'AgeDiscretized' 'Type'}));
lgdPredMeansTest = gs.mean_LGD(LGDGroup);
```

There are eight unique values in the predictions, as expected, one for each group.

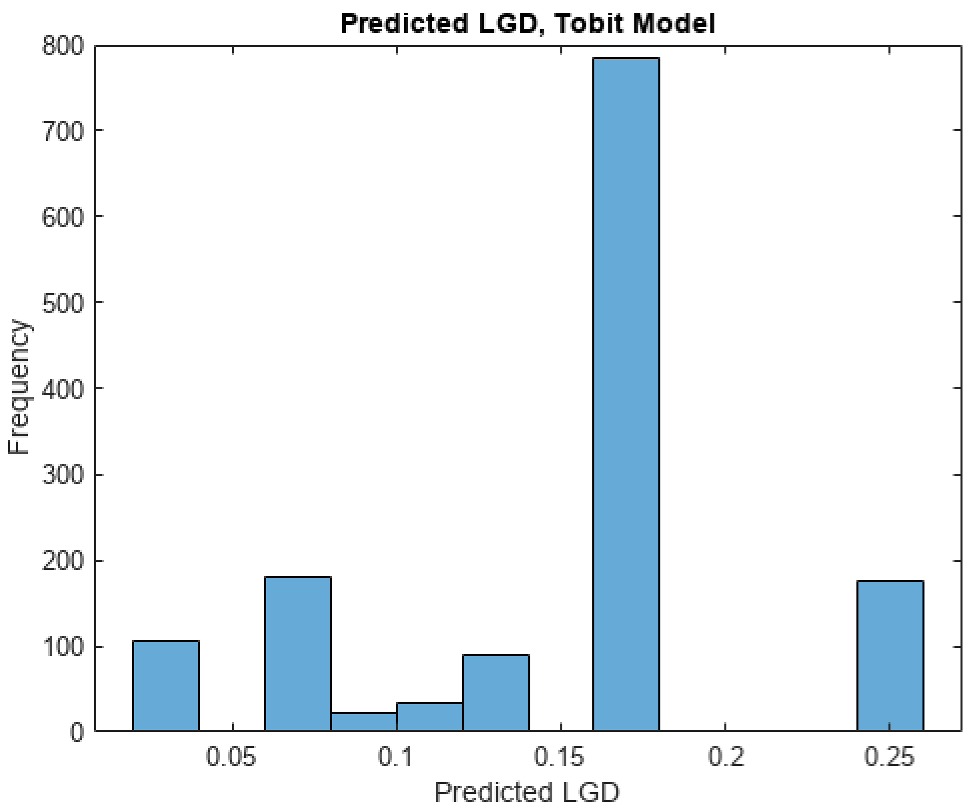
```
disp(unique(lgdPredMeansTest))
```

```
    0.0218
    0.0661
```

```
    0.0873
    0.1178
    0.1217
    0.1638
    0.1649
    0.2598
```

The histogram of the predictions also shows the discrete nature of the model.

```
histogram(lgdPredMeansTest)
title('Predicted LGD, Tobit Model')
xlabel('Predicted LGD')
ylabel('Frequency')
```



To have all the predictions available for both training and test sets to make comparisons, add a column with LGD predictions for the entire data set.

```
LGDGroup = findgroups(data(:,{'LTVDiscretized' 'AgeDiscretized' 'Type'}));
data.lgdPredMeans = gs.mean_LGD(LGDGroup);
```

Compare Performance

Compare the performance of the Tobit model and the benchmark model using the validation functions in the `Tobit` model.

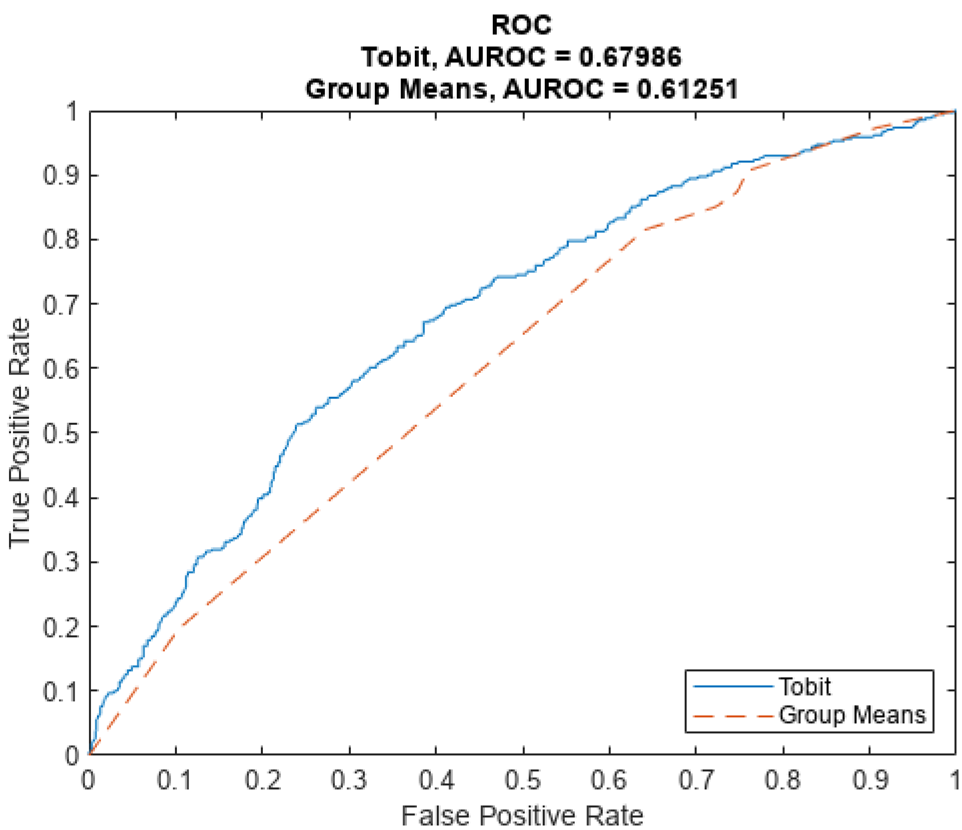
Start with the area under the receiver operating characteristic (ROC) curve, or AUROC metric, using `modelDiscrimination`.

```
DataSetChoice = Testing ;
if DataSetChoice=="Training"
    Ind = TrainingInd;
else
    Ind = TestInd;
end

DiscMeasure = modelDiscrimination(lgdModel,data(Ind,:),'ShowDetails',true,'ReferenceLGD',data.lgd
```

```
DiscMeasure=2×3 table
                    AUROC       Segment      SegmentCount
                   _______    ___________    ____________

    Tobit          0.67986    "all_data"         1394
    Group Means    0.61251    "all_data"         1394
```

Use `modelDiscriminationPlot` to visualize the ROC curve.

```
modelDiscriminationPlot(lgdModel,data(Ind,:),'ReferenceLGD',data.lgdPredMeans(Ind),'ReferenceID',
```



Use `modelCalibration` to compute the calibration metrics.

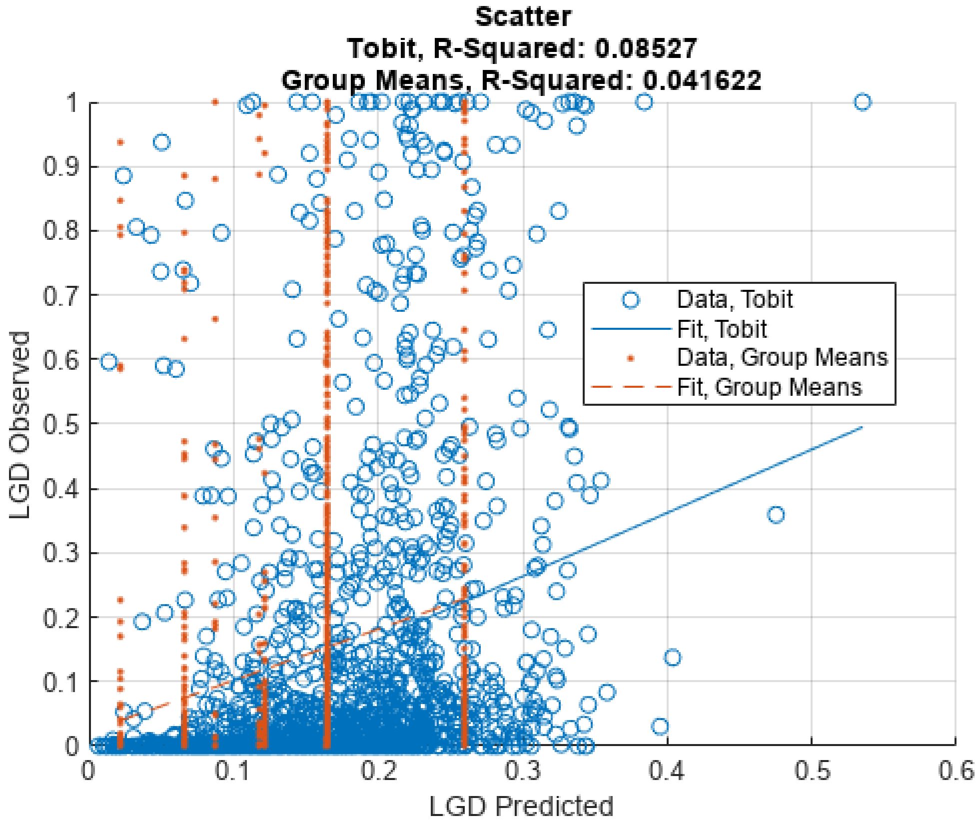
```
CalMeasure = modelCalibration(lgdModel,data(Ind,:),'ReferenceLGD',data.lgdPredMeans(Ind),'Referen
```

```
CalMeasure=2×4 table
                   RSquared      RMSE      Correlation    SampleMeanError
```

```
                 ________    _______    ___________    _______________

    Tobit        0.08527     0.23712      0.29201         -0.034412
    Group Means  0.041622     0.2406      0.20401         -0.0078124
```

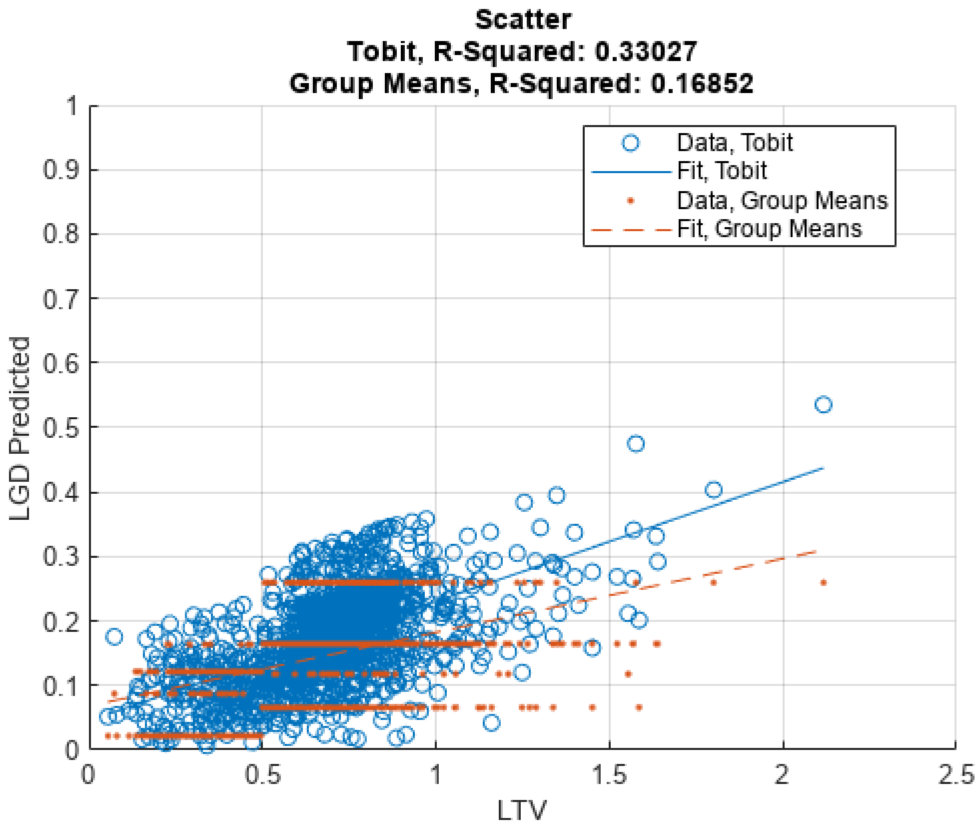
Use `modelCalibrationPlot` to visualize the scatter plot of the observed LGD values against predicted LGD values.

```
modelCalibrationPlot(lgdModel,data(Ind,:),'ReferenceLGD',data.lgdPredMeans(Ind),'ReferenceID','G
```



Then you can use `modelCalibrationPlot` to visualize the scatter plot of the predicted LGD values against the LTV values.

```
modelCalibrationPlot(lgdModel,data(Ind,:),'ReferenceLGD',data.lgdPredMeans(Ind),'ReferenceID','G
```



See Also

`fitLGDModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibartionPlot` | `Regression` | `Tobit`

Related Examples

- "Model Loss Given Default" on page 4-90
- "Basic Loss Given Default Model Validation" on page 4-131
- "Compare Loss Given Default Models Using Cross-Validation" on page 4-140

More About

- "Overview of Loss Given Default Models" on page 1-31

Compare Loss Given Default Models Using Cross-Validation

This example shows how to compare loss given default (LGD) models using cross-validation.

Load Data

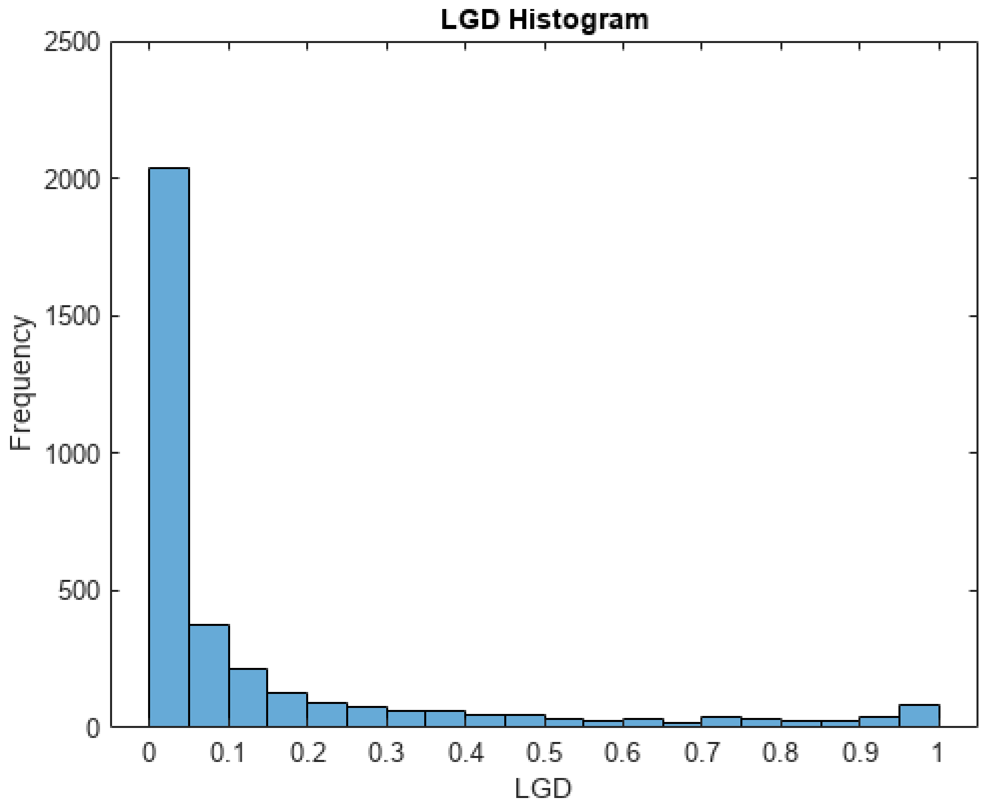
Load the LGD data. This data set is simulated for illustration purposes.

```
load LGDData.mat
disp(head(data))
```

```
      LTV        Age         Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

The histogram of LGD values for this data set shows a significant number of values at or near `0` (full recovery) and only a relatively small fraction of values at or near `1` (total loss).

```
histogram(data.LGD)
xlabel('LGD')
ylabel('Frequency')
title('LGD Histogram')
```



Cross-Validate Models

Compare three `Tobit` LGD models by varying the censoring side choice between the three supported options (`"both"`, `"left"`, and `"right"`). For more information, see the `'CensoringSide'` name-value argument for a `Tobit` object.

Use the `cvpartition` function to generate random partitions on the data for a *k*-fold cross-validation. For each partition, fit a `Tobit` model on the training data with each of the censoring side options and then obtain two validation metrics using the test data. This example uses the validation metrics for area under the receiver operating characteristic curve (AUROC) and the R-squared metric. For more information, see `modelDiscrimination` and `modelCalibration`.

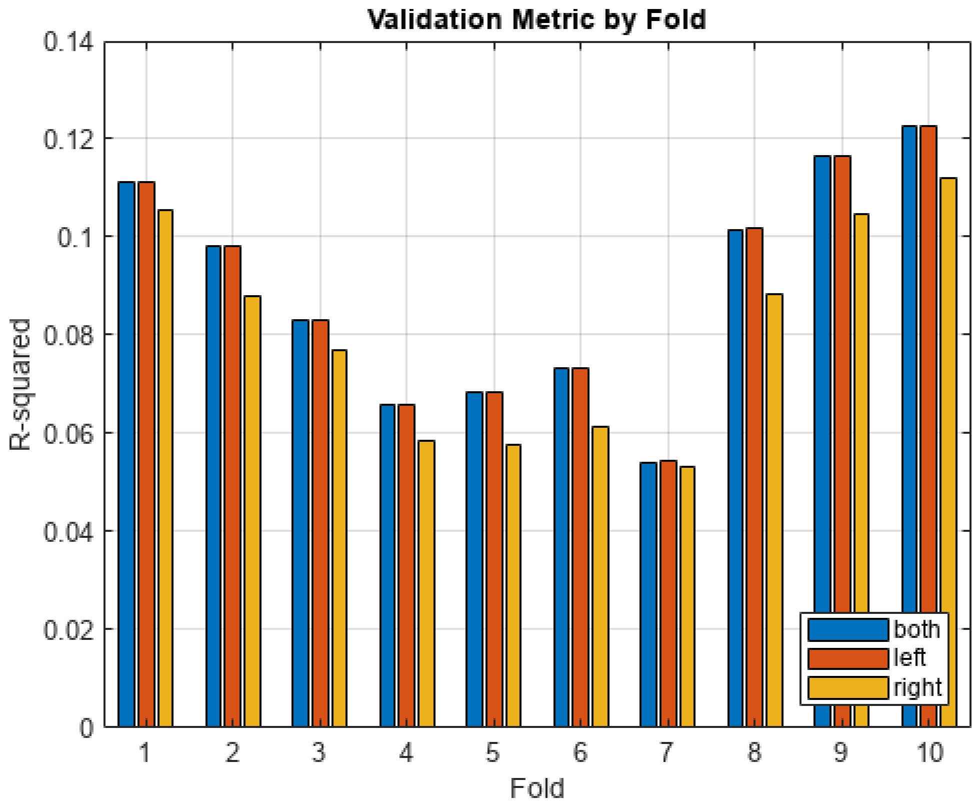
```
NumFolds = 10;
rng('default'); % For reproducibility
c = cvpartition(height(data),'KFold',NumFolds);

ModelCensoringSide = ["both" "left" "right"];

NumModels = length(ModelCensoringSide);

AUROC = zeros(NumFolds,NumModels);
RSquared = zeros(NumFolds,NumModels);

for ii=1:NumFolds

   fprintf('Fitting models, fold %d\n',ii);
```

```
    % Get the partition indices.
    TrainInd = training(c,ii);
    TestInd = test(c,ii);

    % For each model, fit with training data, measure with test data.
    for jj=1:NumModels
        % Fit the model with training data.
        lgdModel = fitLGDModel(data(TrainInd,:),'Tobit','CensoringSide',ModelCensoringSide(jj));

        % Measure the model discrimination on test data.
        DiscMeasure = modelDiscrimination(lgdModel,data(TestInd,:));
        AUROC(ii,jj) = DiscMeasure.AUROC;

        % Measure the model calibration on test data.
        CalMeasure = modelCalibration(lgdModel,data(TestInd,:));
        RSquared(ii,jj) = CalMeasure.RSquared;
    end
end
```

```
Fitting models, fold 1
Fitting models, fold 2
Fitting models, fold 3
Fitting models, fold 4
Fitting models, fold 5
Fitting models, fold 6
Fitting models, fold 7
Fitting models, fold 8
Fitting models, fold 9
Fitting models, fold 10
```

Visualize the results for a selected metric for the three models side-by-side.

```
SelectedMetric = R-squared ;
if SelectedMetric=="AUROC"
    PlotData = AUROC;
else
    PlotData = RSquared;
end

bar(1:NumFolds,PlotData)
xlabel('Fold')
ylabel(SelectedMetric)
title('Validation Metric by Fold')
legend(ModelCensoringSide,'Location','southeast')
grid on
```



The AUROC values for the three models are comparable across the folds, indicating that the three versions of the model effectively separate the low LGD and high LGD cases.

Regarding accuracy, the R-squared metric is low for the three models. However, the `"right"` censored model shows a lower R-squared metric than the other two models across the folds. The observed LGD data has many observations at or near `0` (total recovery). To improve the accuracy of the models, include an explicit limit at `0` when censoring on the `"left"` and on `"both"` sides.

See Also

`fitLGDModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibartionPlot` | `Regression` | `Tobit`

Related Examples

- “Model Loss Given Default” on page 4-90
- “Basic Loss Given Default Model Validation” on page 4-131
- “Compare Tobit LGD Model to Benchmark Model” on page 4-133

More About

- “Overview of Loss Given Default Models” on page 1-31

Compare Model Discrimination and Model Calibration to Validate of Probability of Default

This example shows some differences between discrimination and calibration metrics for the validation of probability of default (PD) models.

The lifetime PD models in Risk Management Toolbox™ (see `fitLifetimePDModel`) support the area under the receiver operating characteristic curve (AUROC) as a discrimination (rank-ordering performance) metric and the root mean squared error (RMSE) as a calibration (predictive ability) metric. The AUROC metric measures ranking, whereas the RMSE measures the precision of the predicted values. The example shows that it is possible to have:

- Same discrimination, different calibration
- Same calibration, different discrimination

Therefore, it is important to look at both discrimination and calibration as part of a model validation framework.

There are several different metrics for PD model discrimination and model calibration. For more information, see References on page 4-149. Different metrics may have different characteristics and the behavior demonstrated in this example does not necessarily generalize to other discrimination and calibration metrics. The goal of this example is to emphasize the importance of using both discrimination and calibration metrics to assess model predictions.

Load and Fit Data

Load credit data and fit a `Logistic` lifetime PD model using `fitLifetimePDModel`.

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);
pdModel = fitLifetimePDModel(data,"logistic",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
disp(pdModel)
```

```
  Logistic with properties:

            ModelID: "Logistic"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

```
disp(pdModel.UnderlyingModel)
```

```
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial
```

```
Estimated Coefficients:
                               Estimate        SE         tStat       pValue
                              __________    _________    _______    ___________

    (Intercept)                  -2.6799     0.078374    -34.193    3.0262e-256
    ScoreGroup_Medium Risk      -0.69409     0.028701    -24.184    3.2931e-129
    ScoreGroup_Low Risk          -1.2979     0.035548    -36.511    7.4134e-292
    YOB                         -0.31534     0.010529    -29.949    4.5479e-197
    GDP                           -0.128      0.03068    -4.1723     3.0157e-05
    Market                    -0.0073407    0.0021916    -3.3496     0.00080942


646724 observations, 646718 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 3.2e+03, p-value = 0
```

Same Discrimination, Different Calibration

Discrimination measures only ranking of customers, that is, whether riskier customers get assigned higher PDs than less risky customers. Therefore, if you scale the probabilities or apply another monotonic transformation that results in valid probabilities, the AUROC measure does not change.

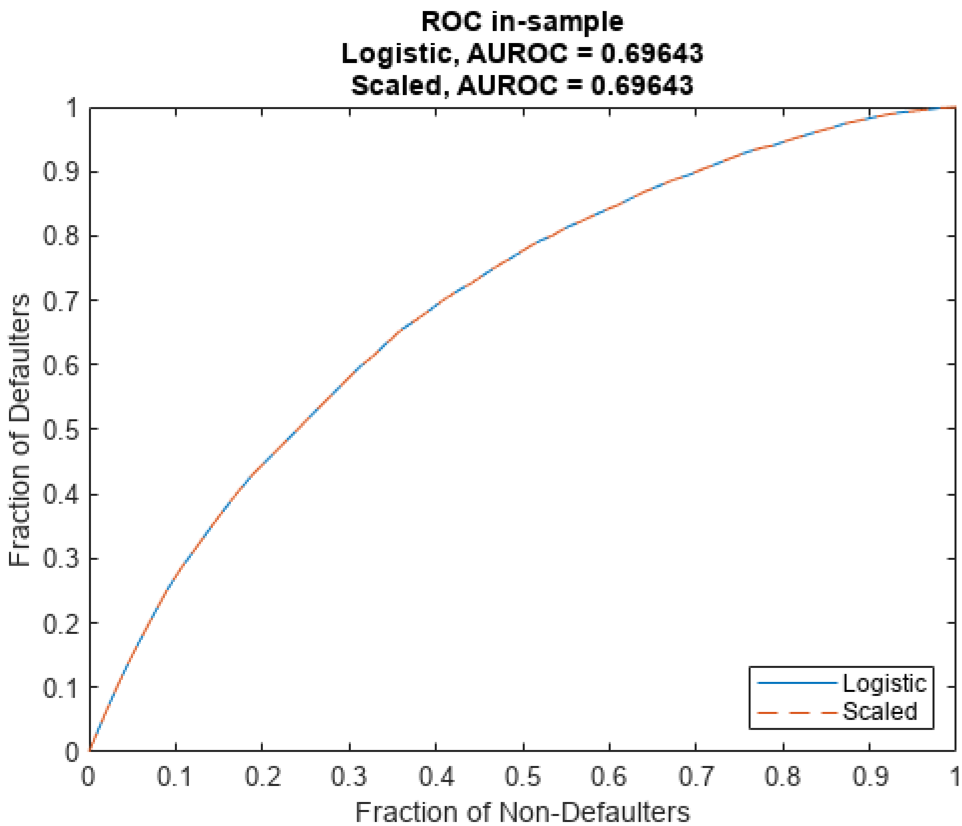
For example, multiply the predicted PDs by a factor of 2, which preserves the ranking (where the worse customers have higher PDs). To compare the results, pass the modified PDs as reference PDs.

```
PD0 = predict(pdModel,data);
PD1 = 2*PD0;

disp([PD0(1:10) PD1(1:10)])
```

```
    0.0090    0.0181
    0.0052    0.0104
    0.0044    0.0088
    0.0038    0.0076
    0.0035    0.0071
    0.0036    0.0072
    0.0019    0.0037
    0.0011    0.0022
    0.0164    0.0328
    0.0094    0.0189
```

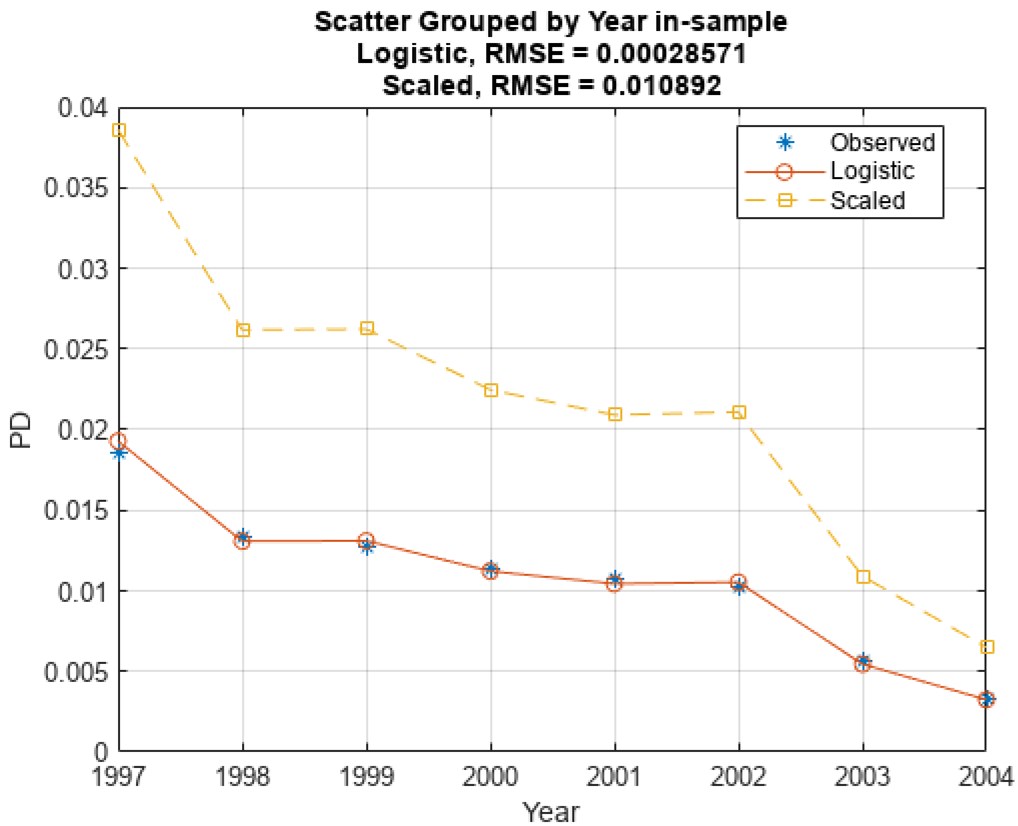
Verify that the discrimination measure is not affected using `modelDiscriminationPlot`.

```
modelDiscriminationPlot(pdModel,data,'DataID','in-sample','ReferencePD',PD1,'ReferenceID','Scale
```



Use `modelCalibrationPlot` to visualize the observed default rates compared to the predicted probabilities of default (PD). The calibration, however, is severely affected by the change. The modified PDs are far away from the observed default rates and the RMSE for the modified PDs is orders of magnitude higher than the RMSE of the original PDs.

```
modelCalibrationPlot(pdModel,data,'Year',"DataID",'in-sample','ReferencePD',PD1,"ReferenceID",'Sc
```



Same Calibration, Different Discrimination

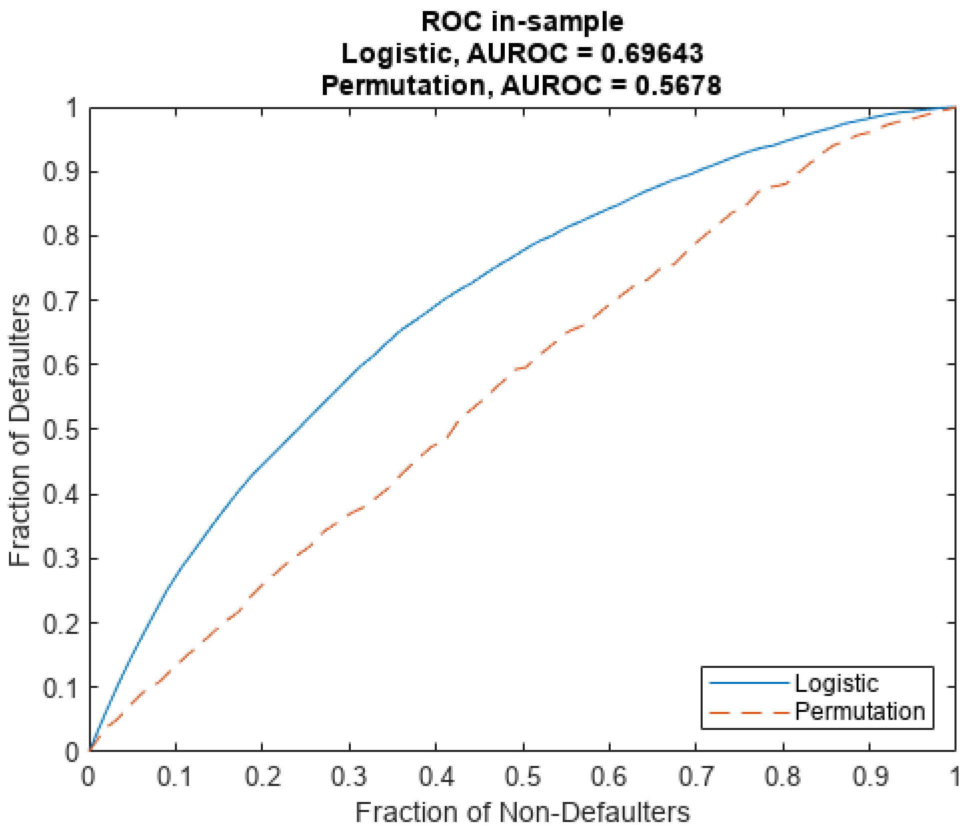
On the other hand, you can also modify the predicted PDs to keep the calibration metric unchanged and worsen the discrimination metric.

One way to do this is to permute the PDs within a group. By doing this, the ranking within each group is affected, but the average PD for the group is unchanged.

```
rng('default'); % for reproducibility
PD1 = PD0;
for Year=1997:2004
   Ind = data.Year==Year;
   PDYear = PD0(Ind);
   PD1(Ind) = PDYear(randperm(length(PDYear)));
end
```

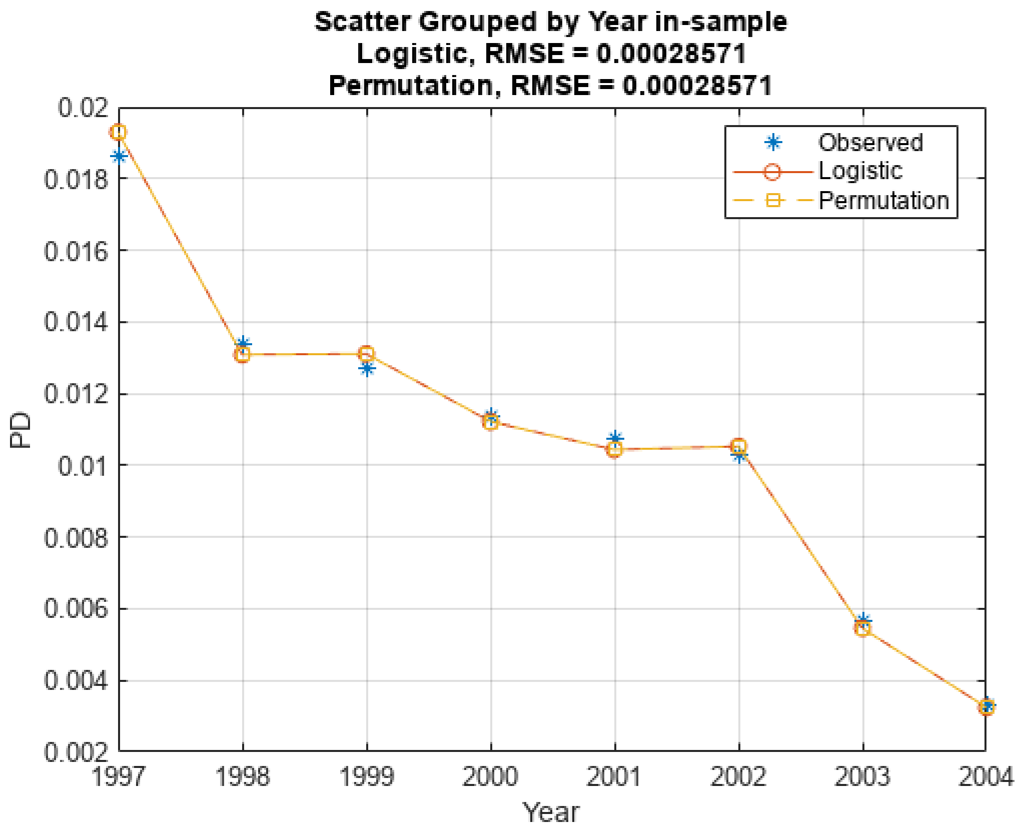
Verify that the discrimination measure is worse for the modified PDs using `modelDiscriminationPlot`.

```
modelDiscriminationPlot(pdModel,data,'DataID','in-sample','ReferencePD',PD1,'ReferenceID','Permu
```



The `modelCalibrationPlot` function measures model calibration for PDs on grouped data. As long as the average PD for the group is unchanged, the reported calibration using the same grouping variable does not change.

```
modelCalibrationPlot(pdModel,data,'Year',"DataID",'in-sample','ReferencePD',PD1,"ReferenceID",'Pe
```



This example shows that discrimination and calibration metrics do not necessarily go hand in hand. Different predictions may have similar RMSE but much different AUROC, or similar AUROC but much different RMSE. Therefore, it is important to look at both discrimination and calibration as part of a model validation framework.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Basel Committee on Banking Supervision, "Studies on the Validation of Internal Rating Systems", Working Paper No. 14, 2005.

See Also

`Probit` | `Logistic` | `Cox` | `modelCalibration` | `modelCalibrationPlot` | `modelDiscriminationPlot` | `modelDiscrimination` | `predictLifetime` | `predict` | `fitLifetimePDModel`

Related Examples

- “Basic Lifetime PD Model Validation” on page 4-129
- “Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
- “Compare Lifetime PD Models Using Cross-Validation” on page 4-121

- “Expected Credit Loss Computation” on page 4-124
- “Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models” on page 4-75

More About

- “Overview of Lifetime Probability of Default Models” on page 1-25

Compare Results for Regression and Tobit EAD Models

This example shows how to use `fitEADModel` to create a `Regression` model and a `Tobit` model for exposure at default (EAD) and then compare the results.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age      Marriage        Limit         Drawn          EAD
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a `Regression` and a `Tobit` model type.

```
ModelTypeR = Regression ;
ModelTypeT = Tobit ;
```

Select Conversion Measure

Select the conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Regression EAD Model

Use `fitEADModel` to create a `Regression` model using the `EADData`.

```
eadModelRegression = fitEADModel(EADData,ModelTypeR,'PredictorVars',{'UtilizationRate','Age','Ma
    'ConversionMeasure',ConversionMeasure,'DrawnVar','Drawn','LimitVar','Limit','ResponseVar','E
disp(eadModelRegression);
```

```
  Regression with properties:

    ConversionTransform: "logit"
      BoundaryTolerance: 1.0000e-07
```

```
             ModelID: "Regression"
         Description: ""
     UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
       PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
         ResponseVar: "EAD"
            LimitVar: "Limit"
            DrawnVar: "Drawn"
   ConversionMeasure: "lcf"
```

Display the underlying model. The underlying `Regression` model's response variable is the `logit` transformation of the EAD response data. Use the `'BoundaryTolerance'`, `'LimitVar'`, and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModelRegression.UnderlyingModel);
```

```
Compact linear regression model:
    EAD_lcf_logit ~ 1 + UtilizationRate + Age + Marriage

Estimated Coefficients:
                            Estimate        SE         tStat       pValue
                            _________    _________    _______    __________

    (Intercept)               -2.4745      0.29892    -8.2781    1.6448e-16
    UtilizationRate            6.0045      0.19901     30.172    7.703e-182
    Age                     -0.020095    0.0073019     -2.752     0.0059471
    Marriage_not married     -0.03509      0.13935    -0.2518        0.8012


Number of observations: 4378, Error degrees of freedom: 4374
Root Mean Squared Error: 4.48
R-squared: 0.173,  Adjusted R-Squared: 0.173
F-statistic vs. constant model: 305, p-value = 5.7e-180
```

Create Tobit EAD Model

Use `fitEADModel` to create a `Tobit` model using the `EADData`.

```
eadModelTobit = fitEADModel(EADData,ModelTypeT,'PredictorVars',{'UtilizationRate','Age','Marriage
    'ConversionMeasure',ConversionMeasure,'DrawnVar','Drawn','LimitVar','Limit','ResponseVar','E
disp(eadModelTobit);
```

```
  Tobit with properties:

        CensoringSide: "right"
            LeftLimit: 0.4000
           RightLimit: 0.5000
              ModelID: "Tobit"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying `Tobit` model's response variable is the `complog` transformation of the EAD response data. Use the `'LimitVar'`, `'DrawnVar'`, `'CensoringSide'`, `'RightLimit'`, `'LeftLimit'`, and `'SolverOptions'` name-value arguments to modify the transformation.

```
disp(eadModelTobit.UnderlyingModel);
```

```
Tobit regression model, right-censored:
     EAD_lcf = min(Y*,0.5)
     Y* ~ 1 + UtilizationRate + Age + Marriage

Estimated coefficients:
                             Estimate         SE         tStat       pValue
                            __________    __________    ________    ________

    (Intercept)               0.18088       0.021124     8.5628            0
    UtilizationRate           0.42381       0.013869     30.558            0
    Age                    -0.0014564     0.00052238     -2.788     0.005326
    Marriage_not married   -0.0040197      0.0096584   -0.41619      0.67729
    (Sigma)                   0.27917      0.0043245     64.555            0

Number of observations: 4378
Number of left-censored observations: 0
Number of uncensored observations: 2802
Number of right-censored observations: 1576
Log-likelihood: -1756.98
```

Predict EAD for Regression Model

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-vale argument.

```
predictedEADRegression = predict(eadModelRegression,EADData(TestInd,:),'ModelLevel','ead');
predictedConversionRegression = predict(eadModelRegression,EADData(TestInd,:),'ModelLevel','Conve
```

Predict EAD for Tobit Model

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-vale argument.

```
predictedEADTobit = predict(eadModelTobit,EADData(TestInd,:),'ModelLevel','ead');
predictedConversionTobit = predict(eadModelTobit,EADData(TestInd,:),'ModelLevel','ConversionMeasu
```

Validate EAD Regression Model

For model validation of the `Regression` model, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

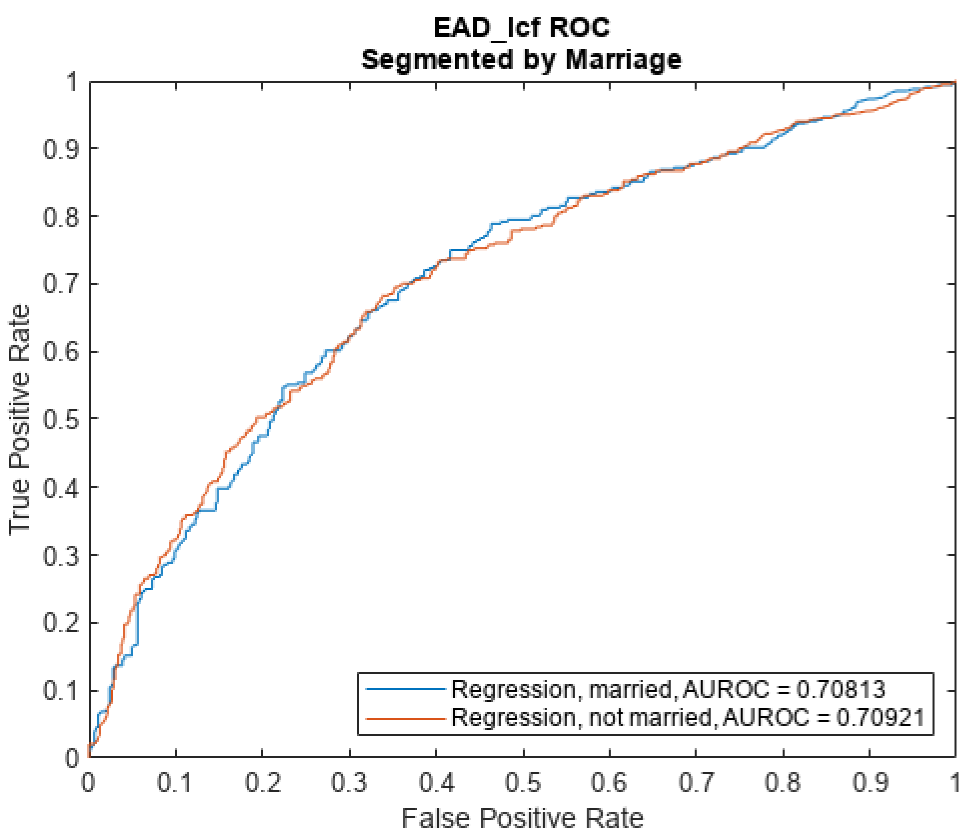
Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ConversionMeasure ;
[DiscMeasureRegression, DiscDataRegression] = modelDiscrimination(eadModelRegression,EADData(Tes
```

```
DiscMeasureRegression=1×3 table
                   AUROC      Segment       SegmentCount
                  _______    ___________    ____________

    Regression    0.70898    "all_data"         1751


DiscDataRegression=1534×3 table
        X             Y           T
```

```
    __________    _________    _______

             0            0    0.95722
             0    0.0027778    0.95722
             0    0.0041667     0.9566
             0    0.0055556    0.95639
             0    0.0083333    0.95576
    0.00096993    0.0097222    0.95555
    0.00096993     0.016667     0.9549
     0.0019399     0.016667    0.95474
     0.0019399     0.018056    0.95468
     0.0038797     0.018056    0.95403
     0.0048497     0.019444    0.95381
     0.0058196     0.019444    0.95314
     0.0067895     0.020833    0.95291
     0.0067895     0.022222    0.95233
     0.0087294     0.026389    0.95224
     0.0087294     0.031944      0.952
       :
```

```
modelDiscriminationPlot(eadModelRegression,EADData(TestInd, :),'ModelLevel',ModelLevel,'SegmentBy
```

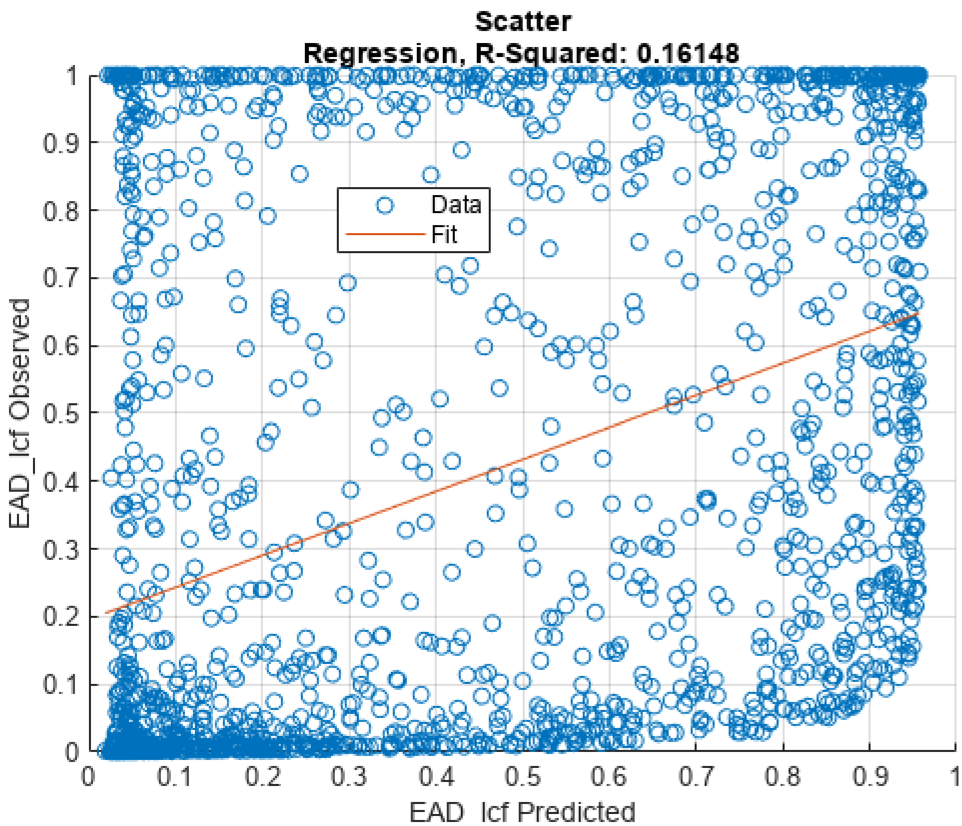


Use `modelCalibration` and then `modelCalibrationPlot` to show a scatter plot of the predictions.

```
YData = [Observed ▾];

[CalMeasureRegression,CalDataRegression] = modelCalibration(eadModelRegression,EADData(TestInd,:)
```

```
CalMeasureRegression=1×4 table
                  RSquared     RMSE      Correlation    SampleMeanError
                  ________    _______    ___________    _______________

    Regression    0.16148     0.41023      0.40184         -0.025994


CalDataRegression=1751×3 table
     Observed     Predicted_Regression    Residuals_Regression
    __________    ____________________    ____________________

       0.99919          0.17519                   0.824
     0.0020632          0.17343                -0.17137
       0.03741           0.7527                -0.71529
       0.75518          0.89867                -0.14349
    0.00076139         0.042389               -0.041628
        0.9998          0.95153                0.048274
     0.0056134           0.1338                -0.12819
      0.048451         0.043424               0.0050276
       0.01448         0.059339               -0.044858
       0.95329          0.67009                  0.2832
       0.97847            0.939                 0.03947
       0.71895          0.80122               -0.082271
       0.79096           0.3791                 0.41186
      0.042816          0.52542                 -0.4826
       0.97169           0.2119                 0.75979
       0.99182          0.62543                 0.36639
          ⋮
```

```
modelCalibrationPlot(eadModelRegression, EADData(TestInd,:), 'ModelLevel', ModelLevel, 'YData', Y
```



Validate EAD Tobit Model

For model validation of the `Tobit` model, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

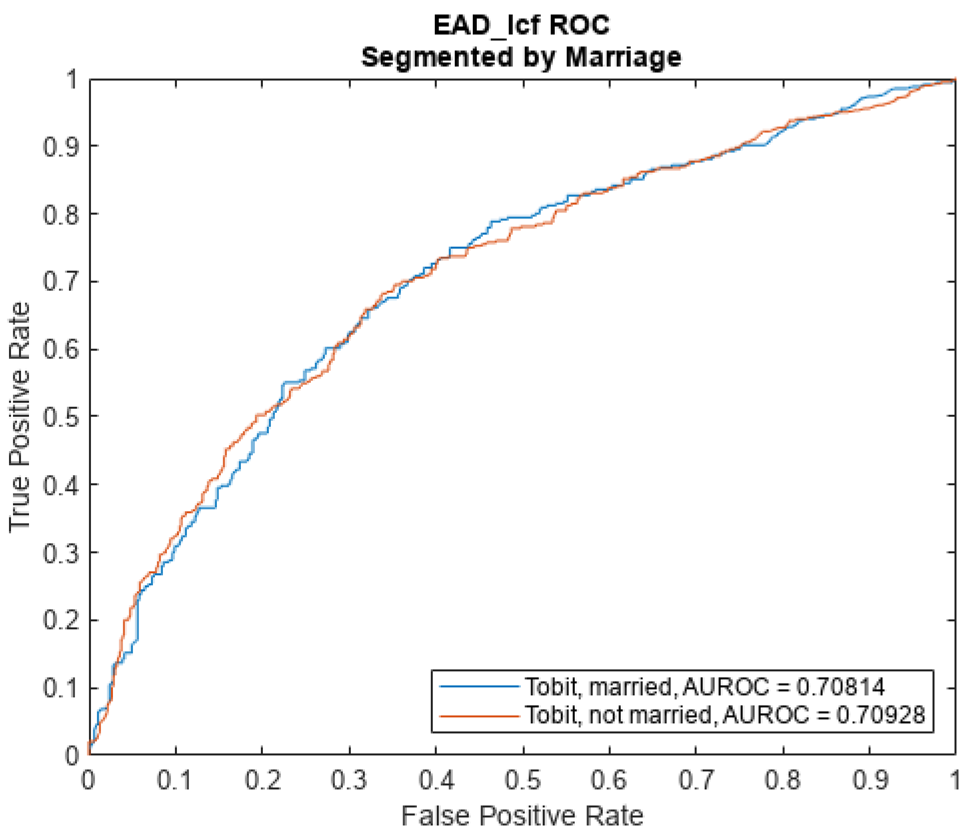
Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ConversionMeasure ;
[DiscMeasureTobit,DiscDataTobit] = modelDiscrimination(eadModelTobit,EADData(TestInd,:),'ShowDeta
```

```
DiscMeasureTobit=1×3 table
              AUROC       Segment       SegmentCount
             _______    __________    ____________

    Tobit    0.70909    "all_data"        1751


DiscDataTobit=1534×3 table
        X               Y            T
    __________    _________    _______

             0            0    0.42178
             0    0.0027778    0.42178
             0    0.0041667     0.4212
             0    0.0055556    0.42076
    0.00096993    0.0069444    0.42062
    0.00096993    0.0097222    0.42018
```

```
    0.00096993      0.011111     0.42004
    0.00096993      0.018056      0.4196
     0.0019399      0.018056      0.4195
     0.0029098      0.019444     0.41945
     0.0048497      0.019444     0.41901
     0.0058196      0.020833     0.41887
     0.0058196      0.022222     0.41854
     0.0067895      0.022222     0.41842
     0.0067895      0.023611     0.41827
     0.0067895      0.029167     0.41827
       ⋮
```

```
modelDiscriminationPlot(eadModelTobit,EADData(TestInd, :),'ModelLevel',ModelLevel,'SegmentBy','Ma
```



Use`modelCalibration` and then `modelCalibrationPlot`. to show a scatter plot of the predictions.

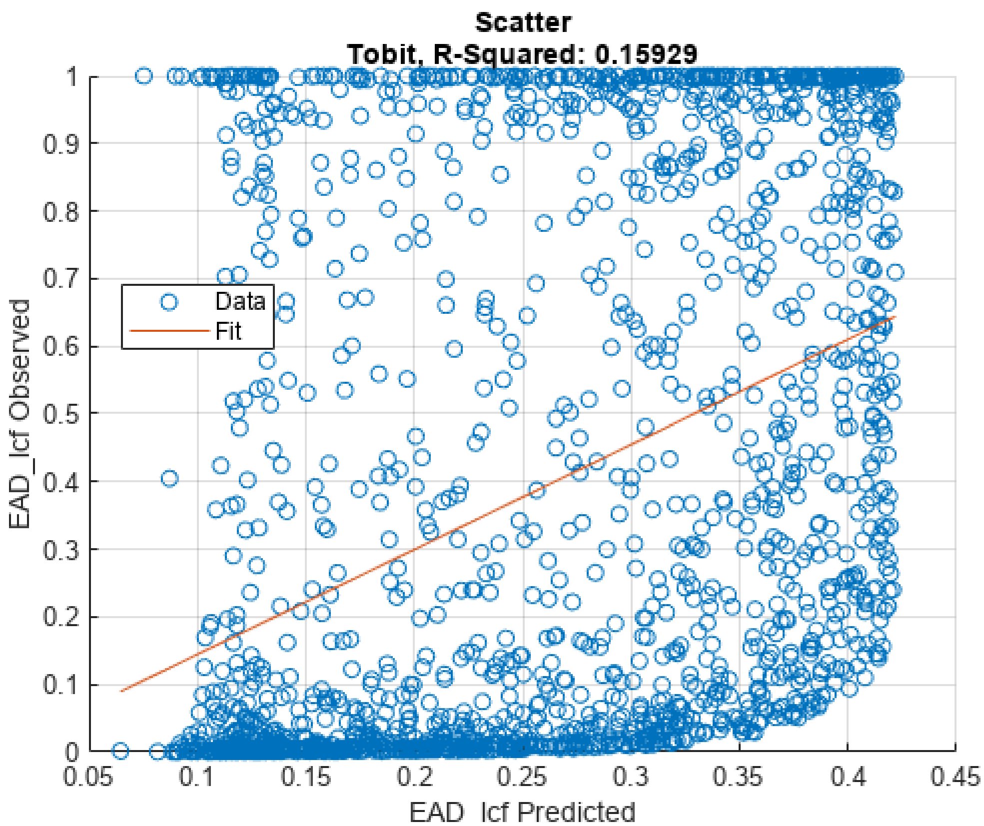
```
YData = Observed ;
```

```
[CalMeasureTobit,CalDataTobit] = modelCalibration(eadModelTobit,EADData(TestInd,:),'ModelLevel',M
```

```
CalMeasureTobit=1×4 table
                          RSquared     RMSE      Correlation    SampleMeanError
                          ________    _______    ___________    _______________
```

```
    Tobit     0.15929      0.39572      0.39911          0.13366

CalDataTobit=1751×3 table
     Observed      Predicted_Tobit    Residuals_Tobit
    __________    _______________    _______________

       0.99919        0.21657             0.78261
     0.0020632        0.21571            -0.21365
       0.03741        0.35115            -0.31374
       0.75518        0.39272             0.36245
    0.00076139        0.12184            -0.12107
        0.9998        0.41744             0.58237
     0.0056134        0.19913            -0.19351
      0.048451        0.12215           -0.073701
       0.01448        0.14323            -0.12875
       0.95329        0.33415             0.61914
       0.97847        0.41069             0.56778
       0.71895         0.3627             0.35624
       0.79096        0.27467             0.51629
      0.042816        0.30579            -0.26297
       0.97169        0.23025             0.74144
       0.99182        0.32461             0.66721
         ⋮
```

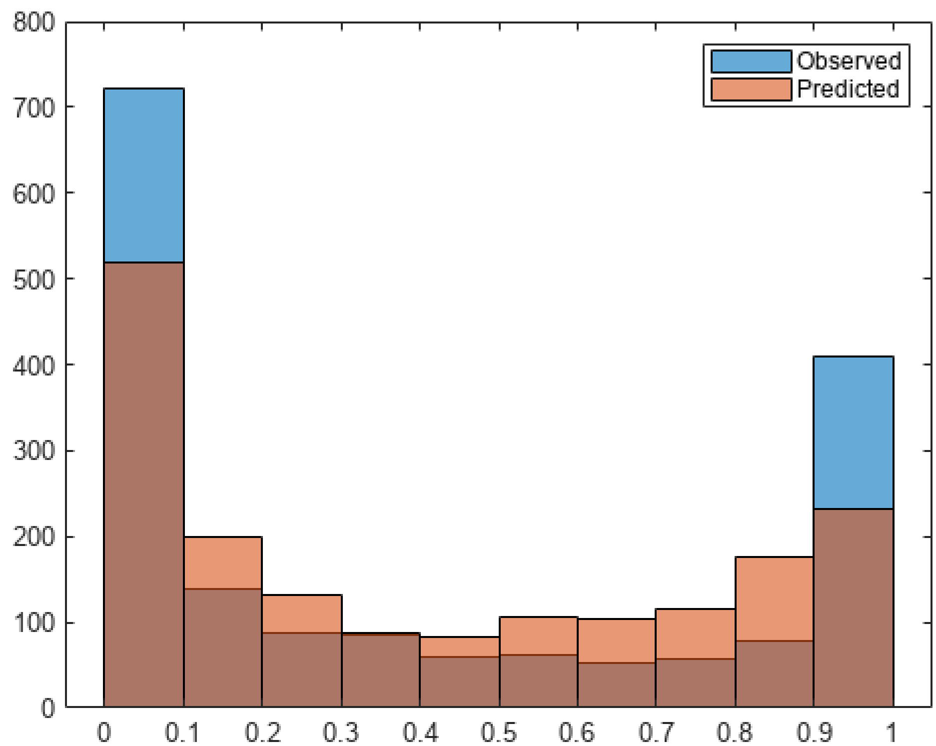
```
modelCalibrationPlot(eadModelTobit,EADData(TestInd,:),'ModelLevel',ModelLevel,'YData',YData);
```



Plot Histograms of Observed with Respect to Predicted EAD

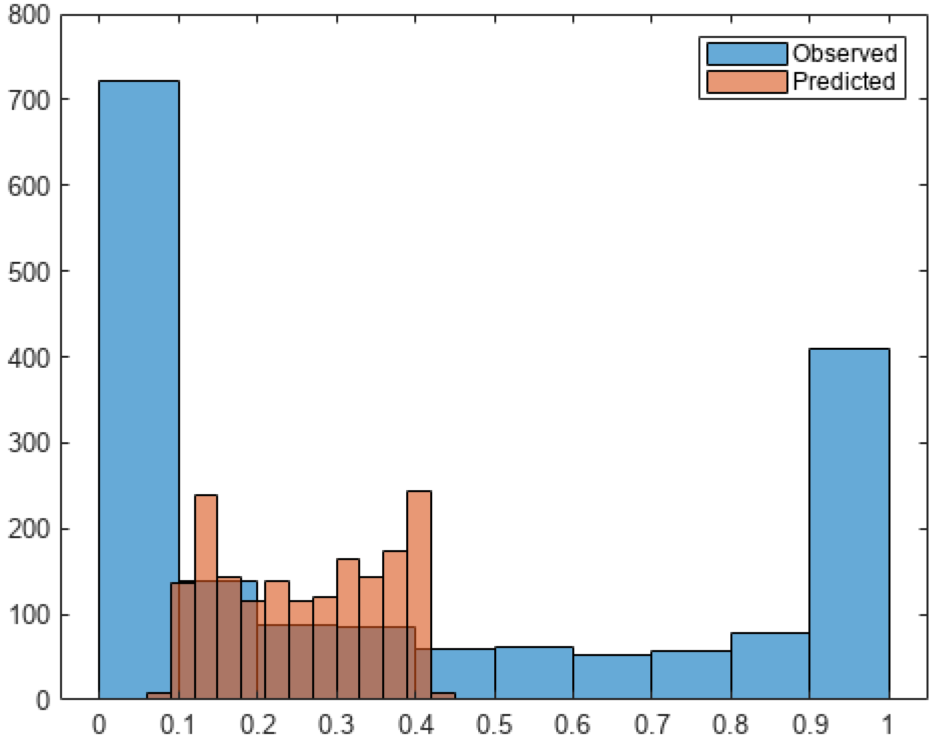
Plot a histogram of observed with respect to the predicted EAD for the `Regression` model.

```
figure;
histogram(CalDataRegression.Observed);
hold on;
histogram(CalDataRegression.(('Predicted_' + ModelTypeR)));
legend('Observed','Predicted');
```



Plot a histogram of observed with respect to the predicted EAD for the `Tobit` model.

```
figure;
histogram(CalDataTobit.Observed);
hold on;
histogram(CalDataTobit.(('Predicted_' + ModelTypeT)));
legend('Observed','Predicted');
```



For both the `Tobit` and `Regression` models, the `Age` and `UtilizationRate` predictors are statistically significant, while the `Marriage` predictor is not statistically significant. Also, the `Tobit` and `Regression` models have different R-square values.

See Also

`Regression` | `Tobit` | `fitEADModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot`

More About

- "Overview of Exposure at Default Models" on page 1-34

Mean Square Error of Prediction for Estimated Ultimate Claims

This example shows a workflow for estimating ultimate claims using a `developmentTriangle` object with simulated reported claims and then calculating the corresponding mean square error of prediction (MSEP).

Actuaries use different techniques to estimate the ultimate claims for different years. In addition to the claim values, an actuary needs to know how well the estimates predict the outcomes of random variables and the uncertainties in the estimates for the ultimate claims. To measure the quality of the estimated ultimate claims, you can calculate the MSEP.

Load Data

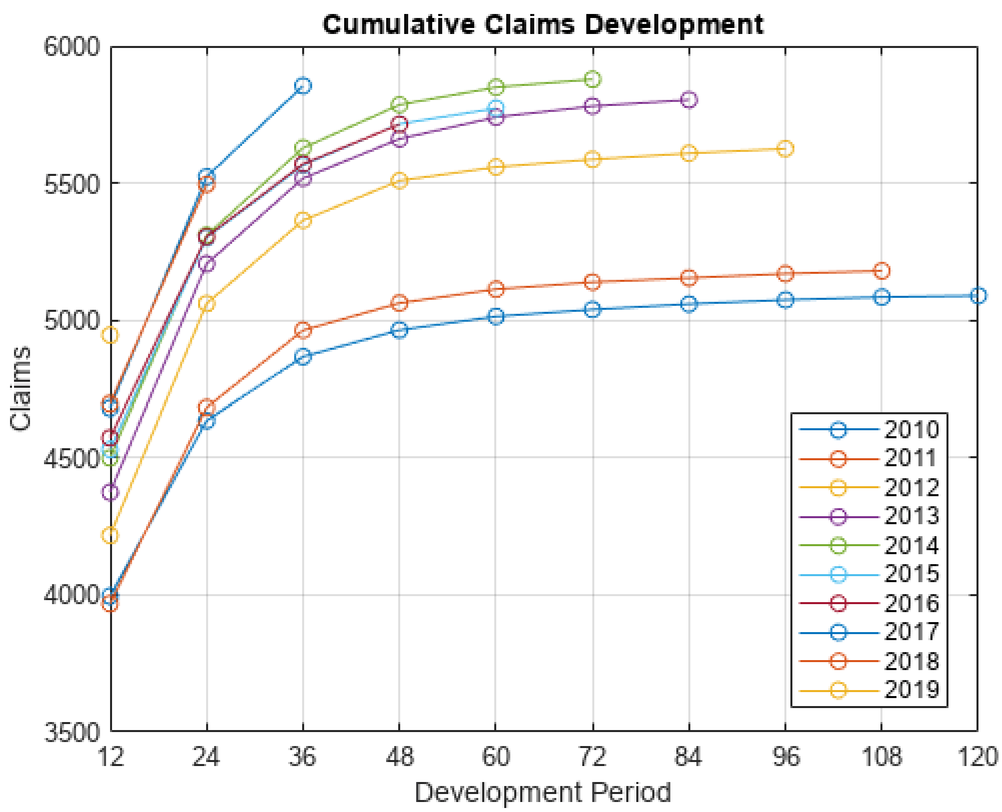
```
load('InsuranceClaimsData.mat');
disp(head(data));
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Create `developmentTriangle`

Create a `developmentTriangle` object and use `claimsPlot` to visualize the `developmentTriangle`. For more information on unpaid claims estimation, see “Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

```
dTriangle = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Clai
dTriangleTable = view(dTriangle);
% Visualize the development triangle
claimsPlot(dTriangle);
```



Analyze `developmentTriangle`

Use `linkRatios` to calculate the age-to-age factors.

```
factorsTable = linkRatios(dTriangle);
```

Use `linkRatioAverages` to calculate the averages of the age-to-age factors.

```
averageFactorsTable = linkRatioAverages(dTriangle);
dTriangle.SelectedLinkRatio = averageFactorsTable{'Volume-weighted Average',:};
dTriangle.TailFactor = 1;
selectedFactorsTable = cdfSummary(dTriangle);
```

Display the full development triangle using the `fullTriangle` function.

```
fullTriangleTable = fullTriangle(dTriangle);
disp(fullTriangleTable);
```

	12	24	36	48	60	72	84	96	108
2010	3995.7	4635	4866.8	4964.1	5013.7	5038.8	5059	5074.1	5084
2011	3968	4682.3	4963.2	5062.5	5113.1	5138.7	5154.1	5169.6	5179
2012	4217	5060.4	5364	5508.9	5558.4	5586.2	5608.6	5625.4	5636
2013	4374.2	5205.3	5517.7	5661.1	5740.4	5780.6	5803.7	5821.1	5832
2014	4499.7	5309.6	5628.2	5785.8	5849.4	5878.7	5900.8	5918.5	5930
2015	4530.2	5300.4	5565.4	5715.7	5772.8	5804.1	5825.9	5843.4	5855
2016	4572.6	5304.2	5569.5	5714.3	5775.4	5806.7	5828.6	5846.1	5857

```
    2017    4680.6    5523.1    5854.4    6000.9    6065.1      6098    6120.9    6139.3    6151
    2018    4696.7    5495.1    5804.4    5949.6    6013.3    6045.9    6068.6    6086.8      609
    2019    4945.9    5819.2    6146.7    6300.5    6367.9    6402.4    6426.5    6445.8    6458
```

Compute the total reserves using `ultimateClaims`.

```
IBNR = ultimateClaims(dTriangle) - dTriangle.LatestDiagonal;
IBNR = array2table(IBNR, 'RowNames', dTriangleTable.Properties.RowNames, 'VariableNames', {'IBNR
IBNR{'Total',1} = sum(IBNR{:,:});
disp(IBNR);
```

```
              IBNR 
             ______

    2010          0
    2011     5.1857
    2012      16.89
    2013     34.886
    2014     57.583
    2015     88.148
    2016     149.34
    2017     303.29
    2018     609.99
    2019     1519.3
    Total    2784.6
```

Calculate Estimated Standard Deviations

The `developmentTriange` link ratios are estimated using the formula:

$$\widehat{f}_j = \frac{\sum_{i=0}^{I-j-1} C_{i,j+1}}{\sum_{i=0}^{I-j-1} C_{i,j}}$$

Along, with the link ratios, the variance parameters are estimated as:

$$\widehat{\sigma_j}^2 = \frac{1}{I-j-1}\sum_{i=0}^{I-j-1} C_{i,j}\left(\frac{C_{i,j+1}}{C_{i,j}} - \widehat{f}_j\right)^2$$

Since the last variance parameter σ_{J-1}^2 cannot be estimated with the estimator $\widehat{\sigma}_{J-1}^2$, the Mack extrapolation method is used to estimate of σ_{J-1}^2:

$$\widehat{\sigma}_{J-1}^2 = \min\left\{\frac{\widehat{\sigma}_{J-2}^4}{\widehat{\sigma}_{J-3}^2}; \widehat{\sigma}_{J-3}^2; \widehat{\sigma}_{J-2}^2\right\}$$

Using this formula, you can compute the estimated conditional process standard deviations.

```
currentSelectedFactors = dTriangle.SelectedLinkRatio;
estimatedStandardDeviations = currentSelectedFactors;
for i=1:width(estimatedStandardDeviations)-1
    estimatedStandardDeviations(1,i) = sqrt(sum(((factorsTable{1:end-i,i} - currentSelectedFacto
end
estimatedStandardDeviations(1,end) = sqrt(min([estimatedStandardDeviations(1,end-1)^4 / estimate
```

Calculate Reserves and Estimated Conditional Process Standard Deviations

Using the latest `developmentTriange` diagonal information and projected ultimate claims from the `developmentTriangle` object, the `ReservesTable` is calculated.

```
h = height(dTriangleTable);
ReservesTable = array2table(NaN(h, 9));
ReservesTable.Properties.RowNames = dTriangleTable.Properties.RowNames;
ReservesTable.Properties.VariableNames = {'Latest Diagonal','Projected Ultimate Claims','Reserves
ReservesTable.("Latest Diagonal") = dTriangle.LatestDiagonal;
ReservesTable.("Projected Ultimate Claims") = ultimateClaims(dTriangle);
ReservesTable.("Reserves") = IBNR.IBNR(1:end-1,:);
```

Estimate the conditional process variance for the ultimate claim of a single accident year as:

$$\widehat{\mathrm{Var}}\left(C_{i,J}\middle|D_I\right) = \left(\widehat{C_{i,J}}^{\mathrm{CL}}\right)^2 \sum_{j=I-i}^{J-1} \frac{\widehat{\sigma_j}^2 / \widehat{f_j}^2}{\widehat{C_{i,j}}^{\mathrm{CL}}}$$

and estimate the conditional process variance for aggregated accident years as:

$$\widehat{\mathrm{Var}}\left(\sum_{i=1}^{I} C_{i,J} \mid D_I\right) = \sum_{i=1}^{I} \widehat{\mathrm{Var}}\left(C_{i,J} \mid D_I\right)$$

Calculate the estimated conditional variational coefficient for origin year *i* relative to the estimated reserves as:

$$V_{\mathrm{CO}_i} = \widehat{V_{\mathrm{CO}}}\left(C_{i,J} - C_{i,I-i} \mid D_I\right) = \frac{\widehat{\mathrm{Var}}\left(C_{i,J} \mid D_I\right)^{\frac{1}{2}}}{\widehat{C_{i,J}}^{\mathrm{CL}} - C_{i,I-i}}$$

```
summationFactors = zeros(1,h);
for i=length(summationFactors)-1:-1:1
    summationFactors(i) = (estimatedStandardDeviations(1,i)^2 / currentSelectedFactors(1,i)^2) /
end
summationFactors = fliplr(summationFactors)';
ReservesTable.("Estimated conditional process standard deviation") = sqrt(ReservesTable.("Projec
ReservesTable.("Estimated conditional variational coefficient") = ReservesTable.("Estimated condi
ReservesTable('Total',:) = array2table([NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN]);
ReservesTable{"Total","Reserves"} = sum(ReservesTable.("Reserves")(1:end-1));
ReservesTable{"Total","Estimated conditional process standard deviation"} = sqrt(sum(ReservesTab
ReservesTable{"Total","Estimated conditional variational coefficient"} = ReservesTable{"Total","E
disp(ReservesTable(:,(2:5)));
```

```
            Projected Ultimate Claims    Reserves    Estimated conditional process standard dev
            _________________________    ________    __________________________________________

    2010             5089.4                    0                         0
    2011             5185.1               5.1857                 0.0072309
    2012             5642.3                16.89                  0.011214
    2013             5838.6               34.886                  0.014452
    2014             5936.3               57.583                    2.7832
    2015               5861               88.148                    5.8489
    2016             5863.6               149.34                    11.634
    2017             6157.7               303.29                    22.586
    2018             6105.1               609.99                    36.512
    2019             6465.2               1519.3                    77.982
    Total               NaN               2784.6                     90.01
```

In addition to these claculated estimates, you can obtain the estimator for the conditional estimation error for origin year *i* as:

$$\widehat{\mathrm{Var}}\left(\widehat{C_{i,J}}^{\mathrm{CL}} \mid D_I\right) = C_{i,I-i}^2\left(\prod_{j=I-i}^{J-1}\left(\widehat{f_j}^2 + \frac{\widehat{\sigma_j}^2}{S_j^{[I-j-1]}}\right) - \prod_{j=I-i}^{J-1}\widehat{f_j}^2\right)$$

where

$$S_j^{[I-j-1]} = \sum_{i=0}^{I-j-1} C_{i,j}$$

```
factor1 = zeros(h,1);
factor2 = zeros(h,1);
factor1(2) = currentSelectedFactors(1,h-1)^2 + estimatedStandardDeviations(1,h-1)^2/sum(dTriangle
factor2(2) = currentSelectedFactors(1,h-1)^2;
for i = 3:length(factor1)
    factor1(i) = (currentSelectedFactors(1,h-i+1)^2 + estimatedStandardDeviations(1,h-i+1)^2/sum
    factor2(i) = currentSelectedFactors(1,h-i+1)^2 * factor2(i-1);
end
Var_hat = sqrt(dTriangle.LatestDiagonal.^2 .* (factor1 - factor2));

ReservesTable.("Conditional Var_hat")(1:end-1) = Var_hat;
ReservesTable.("variation for Var_hat")(1:end-1) = ReservesTable.("Conditional Var_hat")(1:end-1
```

Using the previous formulas, the estimator for the conditional MSEP of the ultimate claim for a single origin year *i* is:

$$\widehat{\mathrm{msep}}_{C_{i,J}|D_I}\left(\widehat{C_{i,J}}^{\mathrm{CL}}\right) = \left(\widehat{C_{i,J}}^{\mathrm{CL}}\right)^2 \sum_{j=I-i}^{J-1}\frac{\widehat{\sigma_j}^2}{\widehat{f_j}^2}\left(\frac{1}{\widehat{C_{i,j}}^{\mathrm{CL}}} + \frac{1}{S_j^{[I-j-1]}}\right)$$

And the estimator for the conditional MSEP of the ultimate claim for aggregated origin years is:

$$\widehat{\mathrm{msep}}_{\sum_i C_{i,J}|D_I}\left(\sum_{i=1}^{I}\widehat{C_{i,J}}^{\mathrm{CL}}\right) = \sum_{i=1}^{I}\widehat{\mathrm{msep}}_{C_{i,J}|D_I}\left(\widehat{C_{i,J}}^{\mathrm{CL}}\right)$$

$$+ 2\sum_{1\le i<k\le I}\widehat{C_{i,J}}^{\mathrm{CL}}\widehat{C_{k,J}}^{\mathrm{CL}}\sum_{j=I-i}^{J-1}\frac{\widehat{\sigma_j}^2/\widehat{f_j}^2}{S_j^{[I-j-1]}}$$

```
summationFactorsMSEP = zeros(h,1);
for i=2:length(summationFactorsMSEP)
    summationFactorsMSEP(i) = (((estimatedStandardDeviations(1,h-i+1)^2 / currentSelectedFactors
end
msep = sqrt(ReservesTable.("Projected Ultimate Claims")(1:end-1).^2 .* summationFactorsMSEP);
ReservesTable.MSEP(1:end-1) = msep;
ReservesTable.("MSEP Uncertainty")(1:end-1) = ReservesTable.MSEP(1:end-1) ./ ReservesTable.("Rese

ReservesTable{'Total','Conditional Var_hat'} = sqrt(sum(ReservesTable.("Conditional Var_hat")(1:e
ReservesTable{'Total','variation for Var_hat'} = ReservesTable{'Total','Conditional Var_hat'} / R

disp(ReservesTable(:,[2,3,6,7]));
```

```
          Projected Ultimate Claims     Reserves     Conditional Var_hat     variation for Var_ha
          _________________________     ________     ___________________     ____________________
```

2010	5089.4	0	0	NaN
2011	5185.1	5.1857	0.0072985	0.14074
2012	5642.3	16.89	0.0099066	0.058655
2013	5838.6	34.886	0.011503	0.032972
2014	5936.3	57.583	1.4539	2.5248
2015	5861	88.148	2.7754	3.1486
2016	5863.6	149.34	5.0379	3.3735
2017	6157.7	303.29	9.1852	3.0285
2018	6105.1	609.99	13.941	2.2854
2019	6465.2	1519.3	28.137	1.852
Total	NaN	2784.6	33.25	1.1941

Calculate MSEP

Measure the quality of the estimated ultimate claims by calculating the `MSEP` and `MSEP Uncertainty`.

```
summationFactorsCovarianceTerm = zeros(h,1);
for i=2:length(summationFactorsCovarianceTerm)
    summationFactorsCovarianceTerm(i) = ((estimatedStandardDeviations(1,h-i+1)^2 / currentSelecte
end

totalSum = 0;
for i = 2:h
totalSum = totalSum + sum(dTriangle.LatestDiagonal(i,1) * fullTriangleTable{i+1:end, h-i+1} * sum
end

covarianceTerm = 2 * totalSum;
totalMSEP = sqrt(sum(ReservesTable.MSEP(1:end-1) .^ 2) + covarianceTerm);

ReservesTable{'Total','MSEP'} = totalMSEP;
ReservesTable{'Total','MSEP Uncertainty'} = ReservesTable{'Total','MSEP'} / ReservesTable{'Total
disp(ReservesTable(:,[1,2,3,8,9]));
```

	Latest Diagonal	Projected Ultimate Claims	Reserves	MSEP	MSEP Uncerta
2010	5089.4	5089.4	0	0	NaN
2011	5179.9	5185.1	5.1857	0.010274	0.19812
2012	5625.4	5642.3	16.89	0.014963	0.088593
2013	5803.7	5838.6	34.886	0.018471	0.052945
2014	5878.7	5936.3	57.583	3.14	5.453
2015	5772.8	5861	88.148	6.474	7.3445
2016	5714.3	5863.6	149.34	12.678	8.4897
2017	5854.4	6157.7	303.29	24.383	8.0394
2018	5495.1	6105.1	609.99	39.083	6.4071
2019	4945.9	6465.2	1519.3	82.903	5.4568
Total	NaN	NaN	2784.6	100.45	3.6074

References

1. Wüthrich, Mario, and Michael Merz. *Stochastic Claims Reserving Methods in Insurance.* Hoboken, NJ: Wiley, 2008.

2 Friedland, Jacqueline. "Estimating Unpaid Claims Using Basic Techniques." Arlington, VA: Casualty Actuarial Society, 2010.

See Also

`developmentTriangle` | `view` | `linkRatios` | `linkRatiosPlot` | `linkRatioAverages` | `cdfSummary` | `ultimateClaims` | `claimsPlot` | `fullTriangle` | `chainLadder` | `expectedClaims` | `bornhuetterFerguson` | `capeCod`

More About

- "Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

Bootstrap Using Chain Ladder Method

This example shows how to apply a chain ladder bootstrap method to generate several `developmentTriangle` objects to estimate the ultimate claims.

Deterministic claim estimation methods produce point estimates of reserve values with no information about the uncertainty of these estimates. The goal of a stochastic claim estimation method is to assess the variability of estimated reserve values. The chain ladder bootstrapping approach is a simulation-based method to randomly modify the `developmentTriangle` data and produce a distribution of estimated reserves that represents the variability of the estimated reserve values. This example is based on the work of Wüthrich and Merz [1 on page 4-177].

Load Data

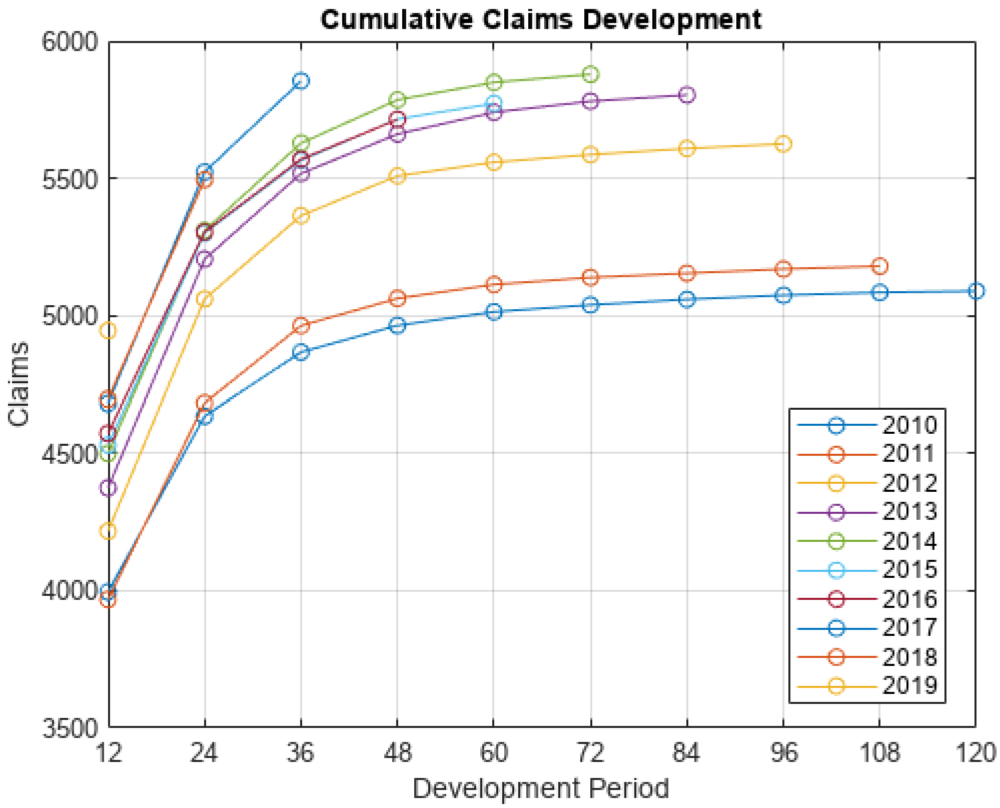
```
load('InsuranceClaimsData.mat');
disp(head(data));
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Create developmentTriangle

Create a `developmentTriangle` object and use `claimsPlot` to visualize the `developmentTriangle`. For more information on unpaid claims estimation, see “Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16.

```
dTriangle = developmentTriangle(data);
dTriangleTable = view(dTriangle);
% visualize the development triangle
claimsPlot(dTriangle)
```



Analyze the `developmentTriangle`

The `developmentTriangle` link ratios are estimated using the formula:

$$\widehat{f}_j = \frac{\sum_{i=0}^{I-j-1} C_{i,j+1}}{\sum_{i=0}^{I-j-1} C_{i,j}}$$

Use `linkRatios` to calculate the age-to-age factors.

```
factorsTable = linkRatios(dTriangle);
```

Use `linkRatioAverages` to calculate the averages of the age-to-age factors.

```
averageFactorsTable = linkRatioAverages(dTriangle);
disp(averageFactorsTable);
```

	12-24	24-36	36-48	48-60	60-72	72-84
Simple Average	1.1767	1.0563	1.0249	1.0107	1.0054	1.003
Simple Average - Latest 5	1.172	1.056	1.0268	1.0108	1.0054	1.003
Simple Average - Latest 3	1.17	1.0533	1.027	1.0117	1.0057	1.003
Medial Average - Latest 5x1	1.1733	1.0567	1.0267	1.0103	1.005	1.00
Volume-weighted Average	1.1766	1.0563	1.025	1.0107	1.0054	1.003
Volume-weighted Average - Latest 5	1.172	1.056	1.0268	1.0108	1.0054	1.003
Volume-weighted Average - Latest 3	1.1701	1.0534	1.027	1.0117	1.0057	1.003
Geometric Average - Latest 4	1.17	1.055	1.0267	1.011	1.0055	1.003

Display the selected age-to-age factors table and calculate the cumulative development factor (CDF) using `cdfSummary`.

```
dTriangle.SelectedLinkRatio = averageFactorsTable{'Volume-weighted Average',:};
currentSelectedFactors = dTriangle.SelectedLinkRatio;
dTriangle.TailFactor = 1;
selectedFactorsTable = cdfSummary(dTriangle);
disp(selectedFactorsTable);
```

```
                              12-24      24-36      36-48      48-60      60-72      72-84
                             _______    _______    _______    _______    _______    ______    _

    Selected                  1.1766     1.0563      1.025     1.0107     1.0054    1.0038
    CDF to Ultimate           1.3072      1.111     1.0518     1.0261     1.0153    1.0098
    Percent of Total Claims  0.76501    0.90008    0.95075    0.97453    0.98496    0.9903    (
```

DIsplay the latest diagonal.

```
latestDiagonal = dTriangle.LatestDiagonal;
```

Compute the projected ultimate claims using `ultimateClaims`.

```
projectedUltimateClaims = ultimateClaims(dTriangle);
```

Display the full development triangle using `fullTriangle`.

```
fullTriangleTable = fullTriangle(dTriangle);
disp(fullTriangleTable);
```

```
              12        24        36        48        60        72        84        96       108
            ______    ______    ______    ______    ______    ______    ______    ______    _____

    2010    3995.7      4635    4866.8    4964.1    5013.7    5038.8      5059    5074.1    5084
    2011      3968    4682.3    4963.2    5062.5    5113.1    5138.7    5154.1    5169.6    5179
    2012      4217    5060.4      5364    5508.9    5558.4    5586.2    5608.6    5625.4    5636
    2013    4374.2    5205.3    5517.7    5661.1    5740.4    5780.6    5803.7    5821.1    5832
    2014    4499.7    5309.6    5628.2    5785.8    5849.4    5878.7    5900.8    5918.5    5930
    2015    4530.2    5300.4    5565.4    5715.7    5772.8    5804.1    5825.9    5843.4    5855
    2016    4572.6    5304.2    5569.5    5714.3    5775.4    5806.7    5828.6    5846.1    5857
    2017    4680.6    5523.1    5854.4    6000.9    6065.1      6098    6120.9    6139.3    6151
    2018    4696.7    5495.1    5804.4    5949.6    6013.3    6045.9    6068.6    6086.8      609
    2019    4945.9    5819.2    6146.7    6300.5    6367.9    6402.4    6426.5    6445.8    6458
```

Compute the total reserves using `ultimateClaims`.

```
IBNR = ultimateClaims(dTriangle) - dTriangle.LatestDiagonal;
IBNR = array2table(IBNR, 'RowNames', dTriangleTable.Properties.RowNames, 'VariableNames', {'IBNR
IBNR{'Total',1} = sum(IBNR{:,:});
disp(IBNR);
```

```
              IBNR
             ______

    2010          0
    2011     5.1857
    2012      16.89
    2013     34.886
    2014     57.583
    2015     88.148
```

```
    2016      149.34
    2017      303.29
    2018      609.99
    2019      1519.3
    Total     2784.6
```

Bootstrap Chain Ladder

To derive the resampling approaches, the Time Series Model of the distribution-free chain ladder (CL) model is defined as:

$$C_{i,j+1} = f_j C_{i,j} + \sigma_j \sqrt{C_{i,j}} \epsilon_{i,j+1}$$

For the link ratio selected above, Wüthrich [1 on page 4-177] and Mack [2 on page 4-177] show that the standard deviation is estimated as:

$$\widehat{\sigma_j}^2 = \frac{1}{I-j-1} \sum_{i=0}^{I-j-1} C_{i,j} \left(\frac{C_{i,j+1}}{C_{i,j}} - \widehat{f_j} \right)^2$$

$$\widehat{\sigma_{J-1}}^2 = \min\left\{ \frac{\widehat{\sigma_{J-2}}^4}{\widehat{\sigma_{J-3}}^3} ; \widehat{\sigma_{J-3}}^2 ; \widehat{\sigma_{J-2}}^2 \right\}$$

```
estimatedStandardDeviations = currentSelectedFactors;
for i=1:width(estimatedStandardDeviations)-1
    estimatedStandardDeviations(1,i) = sqrt(sum(((factorsTable{1:end-i,i} - currentSelectedFactor
end
estimatedStandardDeviations(1,end) = sqrt(min([estimatedStandardDeviations(1,end-1)^4 / estimated

disp(estimatedStandardDeviations);
```

```
    0.8667    0.3699    0.2420    0.1310    0.0673    0.0361    0.0001    0.0001    0.0001
```

To apply the bootstrap method, you need to find the appropriate residuals that allow for the construction of the empirical distribution $\widehat{F_n}$ to construct the bootstrap observations.

Consider the following residuals for $i + j \leq I, j \geq 1$.

$$\widetilde{\epsilon_{i,j}} = \frac{F_{i,j} - \widehat{f_{j-1}}}{\sigma_{j-1} C_{i,j-1}^{-1/2}} \text{ where } F_{i,j} = \frac{C_{i,j}}{C_{i,j-1}}$$

Following Wüthrich [1 on page 4-177], you can scale the residuals to adjust their variance upwards. Unscaled residuals tend to result in lighter tails in the simulated distribution.

Adjust the residuals such that the bootstrap distribution has an adjusted variance function.

$$Z_{i,j} = \left(1 - \frac{C_{i,j-1}}{\sum_{i=0}^{I-j} C_{i,j-1}} \right)^{-\frac{1}{2}} \frac{F_{i,j} - \widehat{f_{j-1}}}{\widehat{\sigma_{j-1}} C_{i,j-1}^{-\frac{1}{2}}}$$

You can apply the bootstrap algorithm using three different versions:

- Efron's nonparametric bootstrap for residuals $\widetilde{\epsilon_{i,j}}$

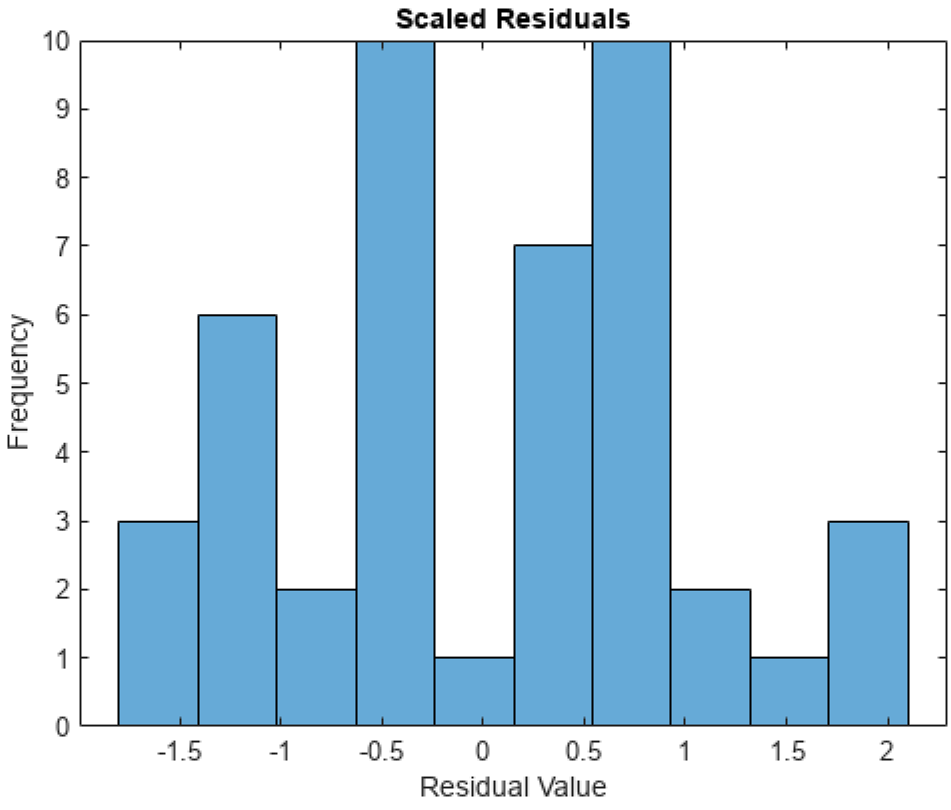
- Efron's nonparametric bootstrap for scaled residuals $Z_{i,j}$
- Parametric bootstrap under the assumption that the residuals have a standard Gaussian distribution, that is $Z^*_{i,j}$ is resampled from $N(0,1)$

This example uses the second version (Efron's nonparametric bootstrap for scaled residuals) to calculate $Z_{i,j}$.

```
% Create a copy of the factors table and modify it to create the
% residuals table
residuals = factorsTable.Variables;

colSums = sum(dTriangle.Claims,'omitnan');
for i=1:height(residuals)
    for j=1:width(residuals)
        residuals(i,j) = (1 - (dTriangleTable{i,j}/colSums(j)))^-0.5 * (factorsTable{i,j} - curre
    end
end
```

The residuals $\{Z_{i,j}, i + j \le I\}$ define a bootstrap distribution.

```
residualsVector = residuals(:);
residualsVector(isnan(residualsVector)) = [];
histogram(residualsVector,10)
title('Scaled Residuals')
xlabel('Residual Value')
ylabel('Frequency')
```



To simulate a new reserves scenario with the bootstrap method, follow these steps.

Step 1: Resample a triangle of residuals from the bootstrap distribution.

Resample the independent and identically distributed (i.i.d.) residuals $\left\{Z*_{i,\,j}, i + j \le I\right\}$ from the bootstrap distribution.

```
resampledResiduals = residuals;

rng('default');
rng(1);

for i = 1:height(residuals)-1
    for j = 1:width(residuals)-i+1
        resampledResiduals(i,j) = datasample(residuals(~isnan(residuals)), 1);
    end
end

disp(resampledResiduals);
```

```
   -1.5522   -0.5120   -1.2668    0.7776   -1.3649    0.2799   -0.5495   -1.3146   -1.5364
   -0.4041   -1.5522   -0.4784   -1.2189   -0.7591    0.2610   -0.4784   -1.5522       NaN
   -0.4091   -1.3649   -0.5495   -1.6767   -0.8571   -1.3143   -0.4879       NaN       NaN
   -0.7591    1.3226    1.0791    0.2610    0.2861   -0.7591       NaN       NaN       NaN
    0.2799   -1.5522   -0.8571    0.3243   -0.4879       NaN       NaN       NaN       NaN
   -1.3143   -0.4784    0.5556   -1.2668       NaN       NaN       NaN       NaN       NaN
    1.9550         0    1.9550       NaN       NaN       NaN       NaN       NaN       NaN
    0.7693    0.5169       NaN       NaN       NaN       NaN       NaN       NaN       NaN
    0.2799       NaN       NaN       NaN       NaN       NaN       NaN       NaN       NaN
       NaN       NaN       NaN       NaN       NaN       NaN       NaN       NaN       NaN
```

Step 2: Compute bootstrapped claims.

Define $C^*_{i,\,0} = C_{i,\,0}$ and, for $j \ge 1$, assume that:

$$C^*_{i,\,j} = \widehat{f_{j-1}} C^*_{i,\,j-1} + \widehat{\sigma_{j-1}} \sqrt{C^*_{i,\,j-1}} Z^*_{i,\,j}$$

This expression represents the new simulated claim values. Using the simulated claim values, you can create a new `developmentTriangle` to estimate new reserve values.

```
bootstrappedClaims = dTriangleTable.Variables;

for j = 2:width(bootstrappedClaims)
    bootstrappedClaims(:,j) = currentSelectedFactors(1,j-1).*bootstrappedClaims(:,j-1) + estimate
end

stackedClaims = reshape(bootstrappedClaims',100,1);
stackedClaims = stackedClaims(~isnan(stackedClaims));
newData = data;
newData.values = stackedClaims;
bootstrappedDevelopmentTriangle = developmentTriangle(newData,'Claims','values');
```

Step 3: Select a link ratio consistent with the model.

The volume-weighted average is the link ratio that is consistent with the model used in this bootstrap approach.

```
bootstrappedAverageFactorsTable = linkRatioAverages(bootstrappedDevelopmentTriangle);
bootstrappedDevelopmentTriangle.SelectedLinkRatio = bootstrappedAverageFactorsTable{'Volume-weigh
bootstrappedDevelopmentTriangle.TailFactor = 1;
bootstrappedSelectedFactorsTable = cdfSummary(bootstrappedDevelopmentTriangle);
disp(bootstrappedSelectedFactorsTable);
```

```
                              12-24      24-36     36-48      48-60      60-72     72-84      8
                             _______    _______   ______    _______    _______   ______     _

    Selected                  1.1751      1.054    1.0253     1.0099     1.0048    1.0036
    CDF to Ultimate            1.301     1.1072    1.0504     1.0245     1.0145    1.0096
    Percent of Total Claims  0.76861    0.90321     0.952    0.97609    0.98572    0.9905    0
```

Use `fullTriangle` to display the full development triangle corresponding to the selected link ratio.

```
bootstrappedFullTriangle = fullTriangle(bootstrappedDevelopmentTriangle);
disp(bootstrappedFullTriangle);
```

```
              12        24        36        48        60        72        84        96        108
            ______    ______    ______    ______    ______    ______    ______    ______    _____

    2010    3995.7    4616.2    4863.2    4963.4    5023.7    5044.5    5064.1    5079.3    5089
    2011      3968    4646.6      4869    4982.8    5024.8    5048.4    5068.1    5083.3    5093
    2012      4217    4938.6    5181.1    5301.1    5341.9    5366.6    5383.3    5399.5    5410
    2013    4374.2    5103.1    5425.3    5580.2    5642.5    5674.5    5693.8    5710.9    5722
    2014    4499.7    5310.5    5567.5    5691.3    5755.4    5784.2    5804.8    5822.2    5833
    2015    4530.2    5253.5    5536.3    5684.8    5733.2      5761    5781.5    5798.8    5810
    2016    4572.6    5494.6    5803.9    5985.1    6044.2    6073.5    6095.1    6113.4    6125
    2017    4680.6    5552.6    5879.4    6028.2    6087.7    6117.2      6139    6157.4    6169
    2018    4696.7    5542.6      5842    5989.8    6048.9    6078.2    6099.9    6118.2    6130
    2019    4945.9      5812      6126      6281      6343    6373.7    6396.4    6415.6    6428
```

Step 4: Compute the total reserves.

Compute the total reserves from the simulated `developmentTriangle`.

```
bootstrappedDevelopmentTriangleTable = view(bootstrappedDevelopmentTriangle);
bootstrappedIBNR = ultimateClaims(bootstrappedDevelopmentTriangle) - bootstrappedDevelopmentTrian
bootstrappedIBNR = array2table(bootstrappedIBNR, 'RowNames', bootstrappedDevelopmentTriangleTable
bootstrappedIBNR{'Total',1} = sum(bootstrappedIBNR{:,:});
disp(bootstrappedIBNR);
```

```
              IBNR 
             ______

    2010          0
    2011     5.0881
    2012     16.188
    2013     34.197
    2014     55.485
    2015     83.048
    2016     146.61
    2017     296.45
    2018     593.94
    2019       1489
    Total      2720
```

You can repeat the previous steps many times to genreate a full, simulated, distribution of reserves. The simulation produces reserves for each year and for the total reserves.

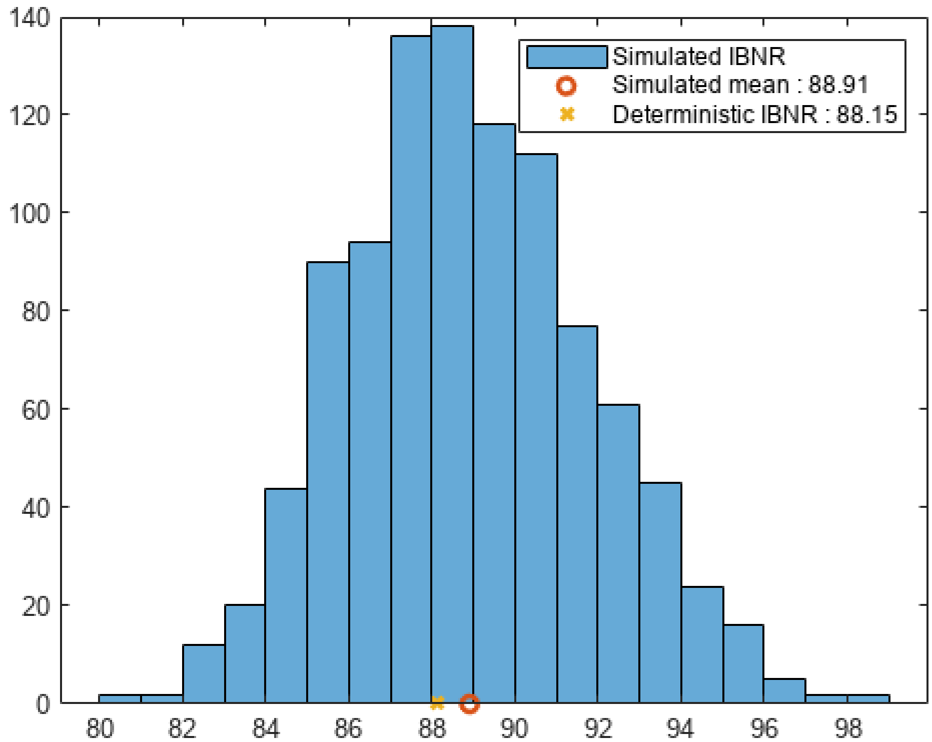
Simulate Multiple Bootstrapped Scenarios

Create `1000` bootstrapped development triangles and calculate the incurred-but-not-reported (IBNR) for each `developmentTriangle`.

```
n = 1000;

simulatedIBNR = zeros(10,n);
for i = 1:n
    simulatedResiduals = residuals;

    for j = 1:height(residuals)-1
        for k = 1:width(residuals)-j+1
            simulatedResiduals(j,k) = datasample(residuals(~isnan(residuals)),1);
        end
    end

    simulatedClaims = dTriangleTable.Variables;

    for j = 2:width(simulatedClaims)
        simulatedClaims(:,j) = currentSelectedFactors(1,j-1).*simulatedClaims(:,j-1) + estimatedS
    end

    simulatedClaims = reshape(simulatedClaims',100,1);
    simulatedClaims = simulatedClaims(~isnan(simulatedClaims));
    simulatedData = data;
    simulatedData.ReportedClaims = simulatedClaims;
    simulatedDevelopmentTriangle = developmentTriangle(simulatedData);

    simulatedAverageFactorsTable = linkRatioAverages(simulatedDevelopmentTriangle);
    simulatedDevelopmentTriangle.SelectedLinkRatio = simulatedAverageFactorsTable{'Volume-weighte
    simulatedDevelopmentTriangle.TailFactor = 1;
    simulatedLatestDiagonal = simulatedDevelopmentTriangle.LatestDiagonal;
    simulatedProjectedUltimateClaims = ultimateClaims(simulatedDevelopmentTriangle);

    simulatedIBNR(:,i) = simulatedProjectedUltimateClaims - simulatedLatestDiagonal;

end

simulatedIBNR(end+1,:) = sum(simulatedIBNR);
```

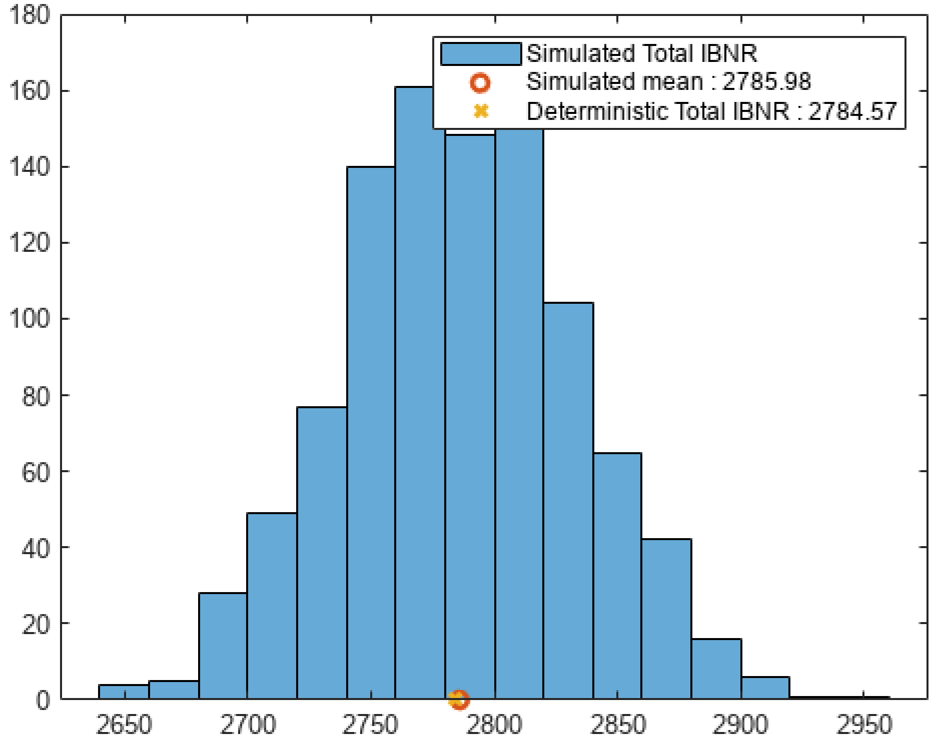
Select a year to plot the distribution of the IBNR, calculate the mean, and compare that mean to a calculated deterministic value.

```
originYear = 5 ;
histogram(simulatedIBNR(originYear+1,:));
hold on;
plot(mean(simulatedIBNR(originYear+1,:)),0,'O','LineWidth',2)
plot(IBNR{originYear+1,1},0,'X','LineWidth',2);
legend('Simulated IBNR',['Simulated mean : ' num2str(round(mean(simulatedIBNR(originYear+1,:)),2
hold off;
```



Plot a histogram of the totals for IBNRs, simulated means, and deterministic values.

```
histogram(simulatedIBNR(11,:));
hold on;
plot(mean(simulatedIBNR(11,:)),0,'O','LineWidth',2)
plot(IBNR{11,1},0,'X','LineWidth',2);
legend('Simulated Total IBNR',['Simulated mean : ' num2str(round(mean(simulatedIBNR(11,:)),2))],
hold off;
```



References

1. Wüthrich, Mario, and Michael Merz. *Stochastic Claims Reserving Methods in Insurance.* Hoboken, NJ: Wiley, 2008
2. Mack, Thomas. "Distribution-Free Calculation of the Standard Error of Chain Ladder Reserve Estimates." *Astin Bulletin*. Vol. 23, No. 2, 1993.

See Also

`developmentTriangle` | `view` | `linkRatios` | `linkRatiosPlot` | `linkRatioAverages` | `cdfSummary` | `ultimateClaims` | `claimsPlot` | `fullTriangle` | `chainLadder` | `expectedClaims` | `bornhuetterFerguson` | `capeCod`

More About

- “Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

Interpret and Stress-Test Deep Learning Networks for Probability of Default

Train a credit risk for probability of default (PD) prediction using a deep neural network. The example also shows how to use the locally interpretable model-agnostic explanations (LIME) and Shapley values interpretability techniques to understand the predictions of the model. In addition, the example analyzes model predictions for out-of-sample values and performs a stress-testing analysis.

The “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36 example presents a similar workflow but uses a logistic model. The “Modeling Probabilities of Default with Cox Proportional Hazards” on page 4-28 example uses a Cox regression, or Cox proportional hazards model. However, interpretability techniques are not discussed in either of these examples because the models are simpler and interpretable. The “Compare Deep Learning Networks for Credit Default Prediction” (Deep Learning Toolbox) example focuses on alternative network designs and fits simpler models without the macroeconomic variables.

While you can use these alternative, simpler models successfully to model credit risk, this example introduces explainability tools for exploring complex-modeling techniques in credit applications. To visualize and interpret the model predictions, you use Deep Learning Toolbox™ and the `lime` and `shapley` functions. To run this example, you:

1. Load and prepare credit data, reformat predictors, and split the data into training, validation, and testing sets.
2. Define a network architecture, select training options, and train the network. (A saved version of the trained network `residualTrainedNetworkMacro` is available for convenience.)
3. Apply the LIME and Shapley interpretability techniques on observations of interest (or "query points") to determine if the importance of predictors in the model is as expected.
4. Explore extreme predictor out-of-sample values to investigate the behavior of the model for new, extreme data.
5. Use the model to perform a stress-testing analysis of the predicted PD values.

Load Credit Default Data

Load the retail credit panel data set including its macroeconomic variables. The main data set (`data`) contains the following variables:

- `ID`: Loan identifier
- `ScoreGroup`: Credit score at the beginning of the loan, discretized into three groups, `High Risk`, `Medium Risk`, and `Low Risk`
- `YOB`: Years on books
- `Default`: Default indicator; the response variable
- `Year`: Calendar year

The small data set (`dataMacro`) contains macroeconomic data for the corresponding calendar years:

- `Year`: Calendar year
- `GDP`: Gross domestic product growth (year over year)
- `Market`: Market return (year over year)

The variables YOB, Year, GDP, and Market are observed at the end of the corresponding calendar year. The score group is a discretization of the original credit score when the loan started. A value of 1 for Default means that the loan defaulted in the corresponding calendar year.

The third data set (dataMacroStress) contains baseline, adverse, and severely adverse scenarios for the macroeconomic variables. This table is for the stress-testing analysis.

This example uses simulated data, but the same approach has been successfully applied to real data sets.

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);
head(data)
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Encode Categorical Variables

To train a deep learning network, you must first encode the categorical ScoreGroup variable to one-hot encoded vectors.

View the order of the ScoreGroup categories.

```
categories(data.ScoreGroup)'
```

```
ans = 1×3 cell
    {'High Risk'}    {'Medium Risk'}    {'Low Risk'}
```

```
ans = 1×3 cell

{'High Risk'} {'Medium Risk'} {'Low Risk'}
```

One-hot encode the ScoreGroup variable.

```
riskGroup = onehotencode(data.ScoreGroup,2);
```

Add the one-hot vectors to the table.

```
data.HighRisk = riskGroup(:,1);
data.MediumRisk = riskGroup(:,2);
data.LowRisk = riskGroup(:,3);
```

Remove the original ScoreGroup variable from the table using removevars.

```
data = removevars(data,{'ScoreGroup'});
```

Move the Default variable to the end of the table, as this variable is the response you want to predict.

```
data = movevars(data,'Default','After','LowRisk');
```

View the first few rows of the table. The `ScoreGroup` variable is split into multiple columns with the categorical values as the variable names.

```
head(data)
```

ID	YOB	Year	GDP	Market	HighRisk	MediumRisk	LowRisk	Default
1	1	1997	2.72	7.61	0	0	1	0
1	2	1998	3.57	26.24	0	0	1	0
1	3	1999	2.86	18.1	0	0	1	0
1	4	2000	2.43	3.19	0	0	1	0
1	5	2001	1.26	-10.51	0	0	1	0
1	6	2002	-0.59	-22.95	0	0	1	0
1	7	2003	0.63	2.78	0	0	1	0
1	8	2004	1.85	9.48	0	0	1	0

Split Data

Partition the data set into training, validation, and test partitions using the unique loan ID numbers. Set aside 60% of the data for training, 20% for validation, and 20% for testing.

Find the unique loan IDs.

```
idx = unique(data.ID);
numObservations = length(idx);
```

Determine the number of observations for each partition.

```
numObservationsTrain = floor(0.6*numObservations);
numObservationsValidation = floor(0.2*numObservations);
numObservationsTest = numObservations - numObservationsTrain - numObservationsValidation;
```

Create an array of random indices corresponding to the observations and partition it using the partition sizes.

```
rng('default'); % for reproducibility
idxShuffle = idx(randperm(numObservations));

idxTrain = idxShuffle(1:numObservationsTrain);
idxValidation = idxShuffle(numObservationsTrain+1:numObservationsTrain+numObservationsValidation)
idxTest = idxShuffle(numObservationsTrain+numObservationsValidation+1:end);
```

Find the table entries corresponding to the data set partitions.

```
idxTrainTbl = ismember(data.ID,idxTrain);
idxValidationTbl = ismember(data.ID,idxValidation);
idxTestTbl = ismember(data.ID,idxTest);
```

Keep the variables of interest for the task (`YOB`, `Default`, and `ScoreGroup`) and remove all other variables from the table.

```
data = removevars(data,{'ID','Year'});
head(data)
```

YOB	GDP	Market	HighRisk	MediumRisk	LowRisk	Default

```
    1      2.72      7.61       0            0            1          0
    2      3.57     26.24       0            0            1          0
    3      2.86      18.1       0            0            1          0
    4      2.43      3.19       0            0            1          0
    5      1.26    -10.51       0            0            1          0
    6     -0.59    -22.95       0            0            1          0
    7      0.63      2.78       0            0            1          0
    8      1.85      9.48       0            0            1          0
```

Partition the table of data into training, validation, and testing partitions using the indices.

```
tblTrain = data(idxTrainTbl,:);
tblValidation = data(idxValidationTbl,:);
tblTest = data(idxTestTbl,:);
```

Define Network Architecture

You can use different deep learning architectures for the task of predicting credit default probabilities. Smaller networks are quick to train, but deeper networks can learn more abstract features. Choosing a neural network architecture requires balancing computation time against accuracy. This example uses a residual architecture. For an example of other networks, see the "Compare Deep Learning Networks for Credit Default Prediction" (Deep Learning Toolbox) example.

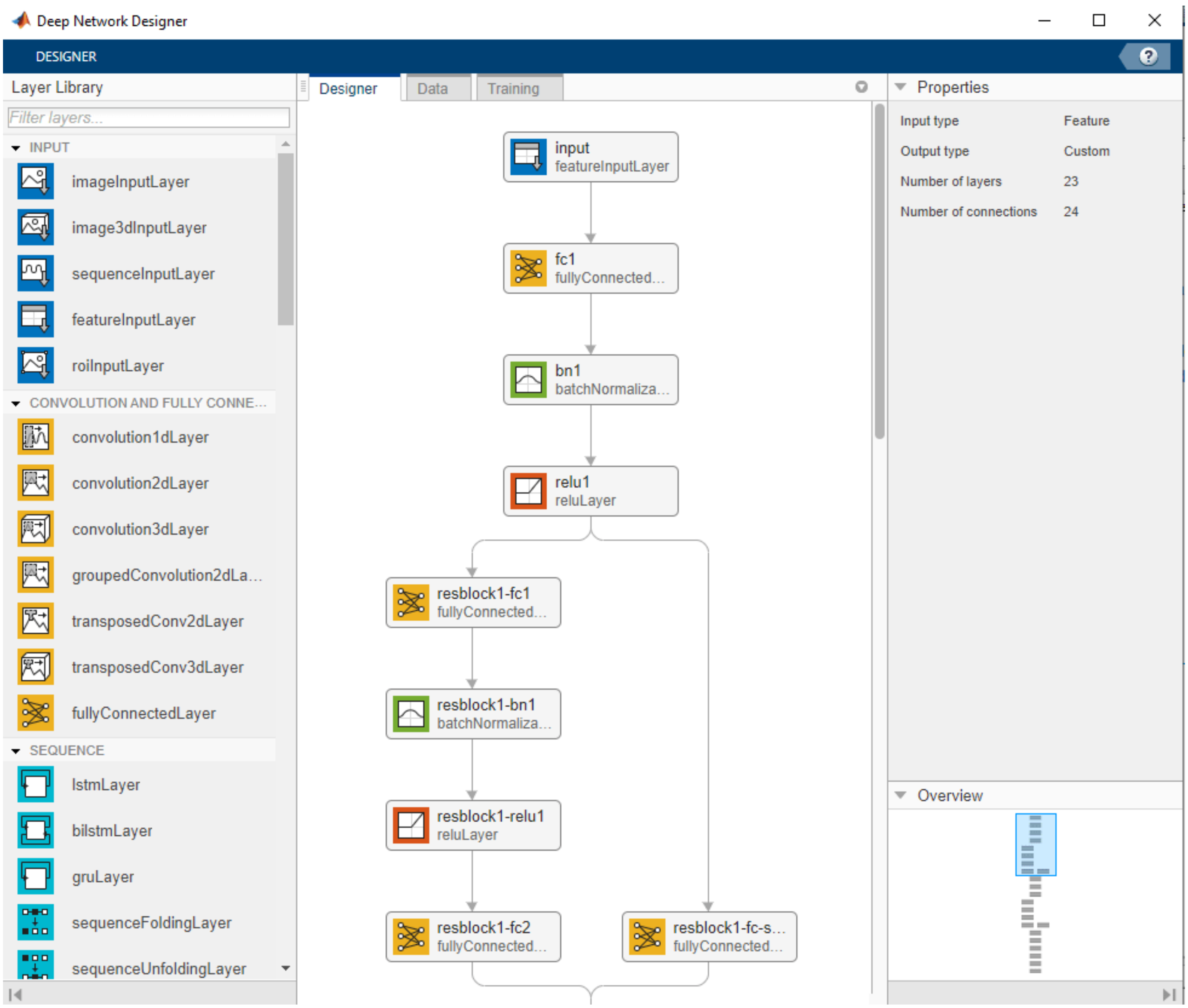
Create a residual architecture (ResNet) from multiple stacks of fully connected layers and ReLU activations. ResNet architectures are state of the art in deep learning applications and popular in deep learning literature. Originally developed for image classification, ResNets have proven successful across many domains [1 on page 4-193].

```
residualLayers = [
    featureInputLayer(6, 'Normalization', 'zscore', 'Name', 'input')
    fullyConnectedLayer(16, 'Name', 'fc1','WeightsInitializer','he')
    batchNormalizationLayer('Name', 'bn1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(32, 'Name', 'resblock1-fc1','WeightsInitializer','he')
    batchNormalizationLayer('Name', 'resblock1-bn1')
    reluLayer('Name', 'resblock1-relu1')
    fullyConnectedLayer(32, 'Name', 'resblock1-fc2','WeightsInitializer','he')
    additionLayer(2, 'Name', 'resblock1-add')
    batchNormalizationLayer('Name', 'resblock1-bn2')
    reluLayer('Name', 'resblock1-relu2')
    fullyConnectedLayer(64, 'Name', 'resblock2-fc1','WeightsInitializer','he')
    batchNormalizationLayer('Name', 'resblock2-bn1')
    reluLayer('Name', 'resblock2-relu1')
    fullyConnectedLayer(64, 'Name', 'resblock2-fc2','WeightsInitializer','he')
    additionLayer(2, 'Name', 'resblock2-add')
    batchNormalizationLayer('Name', 'resblock2-bn2')
    reluLayer('Name', 'resblock2-relu2')
    fullyConnectedLayer(1, 'Name', 'fc2','WeightsInitializer','he')
    sigmoidLayer('Name', 'sigmoid')
    BinaryCrossEntropyLossLayer('output')];

residualLayers = layerGraph(residualLayers);
residualLayers = addLayers(residualLayers,fullyConnectedLayer(32, 'Name', 'resblock1-fc-shortcut
residualLayers = addLayers(residualLayers,fullyConnectedLayer(64, 'Name', 'resblock2-fc-shortcut

residualLayers = connectLayers(residualLayers, 'relu1', 'resblock1-fc-shortcut');
residualLayers = connectLayers(residualLayers, 'resblock1-fc-shortcut', 'resblock1-add/in2');
```

```
residualLayers = connectLayers(residualLayers, 'resblock1-relu2', 'resblock2-fc-shortcut');
residualLayers = connectLayers(residualLayers, 'resblock2-fc-shortcut', 'resblock2-add/in2');
```

You can visualize the network using Deep Network Designer (Deep Learning Toolbox) or the `analyzeNetwork` (Deep Learning Toolbox) function.

```
deepNetworkDesigner(residualLayers)
```



Specify Training Options

In this example, train each network with these training options:

- Train using the Adam optimizer.
- Set the initial learning rate to `0.001`.
- Set the mini-batch size to `512`.

- Train for 75 epochs.
- Turn on the training progress plot and turn off the command window output.
- Shuffle the data at the beginning of each epoch.
- Monitor the network accuracy during training by specifying validation data and using it to validate the network every 1000 iterations.

```
options = trainingOptions('adam', ...
   'InitialLearnRate',0.001, ...
   'MiniBatchSize',512, ...
   'MaxEpochs',75, ...
   'Plots','training-progress', ...
   'Verbose',false, ...
   'Shuffle','every-epoch', ...
   'ValidationData',tblValidation, ...
   'ValidationFrequency',1000);
```

The “Compare Deep Learning Networks for Credit Default Prediction” (Deep Learning Toolbox) example fits the same type of network, but it excludes the macroeconomic predictors. In that example, if you increase the number of epochs from 50 to 75, you can improve accuracy without overfitting concerns.

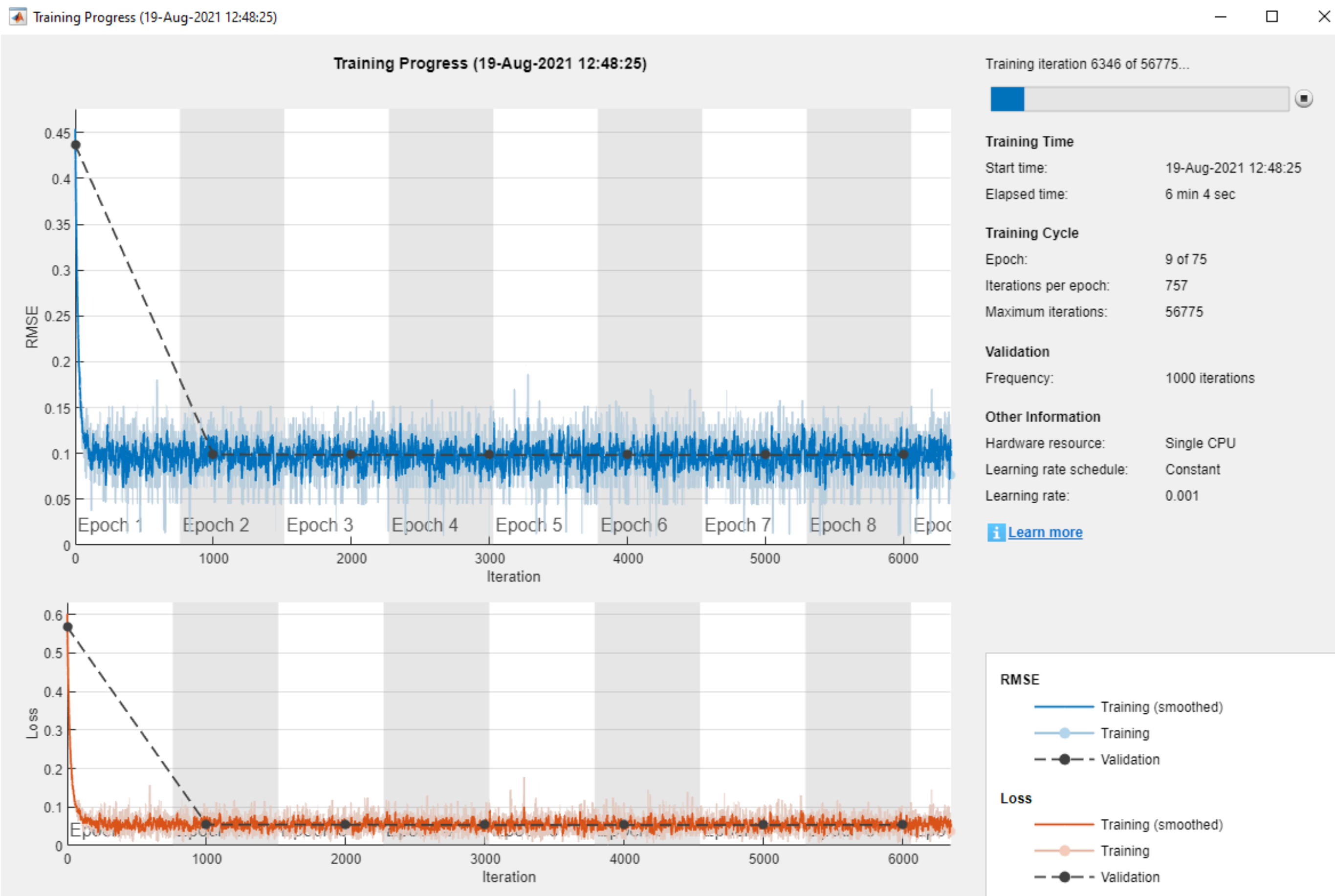
You can perform optimization programmatically or interactively using Experiment Manager (Deep Learning Toolbox). For an example showing how to perform a hyperparameter sweep of the training options, see “Create a Deep Learning Experiment for Classification” (Deep Learning Toolbox).

Train Network

Train the network using the architecture that you defined, the training data, and the training options. By default, `trainNetwork` (Deep Learning Toolbox) uses a GPU if one is available; otherwise, it uses a CPU. Training on a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information, see “Deep Learning with MATLAB on Multiple GPUs” (Deep Learning Toolbox). You can also specify the execution environment by using the `'ExecutionEnvironment'` name-value argument of `trainingOptions` (Deep Learning Toolbox).

To avoid waiting for the training, load pretrained networks by setting the `doTrain` flag to `false`. To train the networks using `analyzeNetwork` (Deep Learning Toolbox), set the `doTrain` flag to `true`. The Training Progress window displays progress. The training time using an NVIDIA® GeForce® RTX 2080 is about 35 minutes for 75 epochs.

```
doTrain = false;

if doTrain
   residualNetMacro = trainNetwork(tblTrain,'Default',residualLayers,options);
else
   load residualTrainedNetworkMacro.mat
end
```



Test Network

Use the `predict` (Deep Learning Toolbox) function to predict the default probability of the test data using the trained networks.

```
tblTest.residualPred = predict(residualNetMacro,tblTest(:,1:end-1));
```

Plot Default Rates by Year on Books

To assess the performance of the network, use the `groupsummary` function to group the true default rates and corresponding predictions by years on the books (represented by the `YOB` variable) and calculate the mean value.

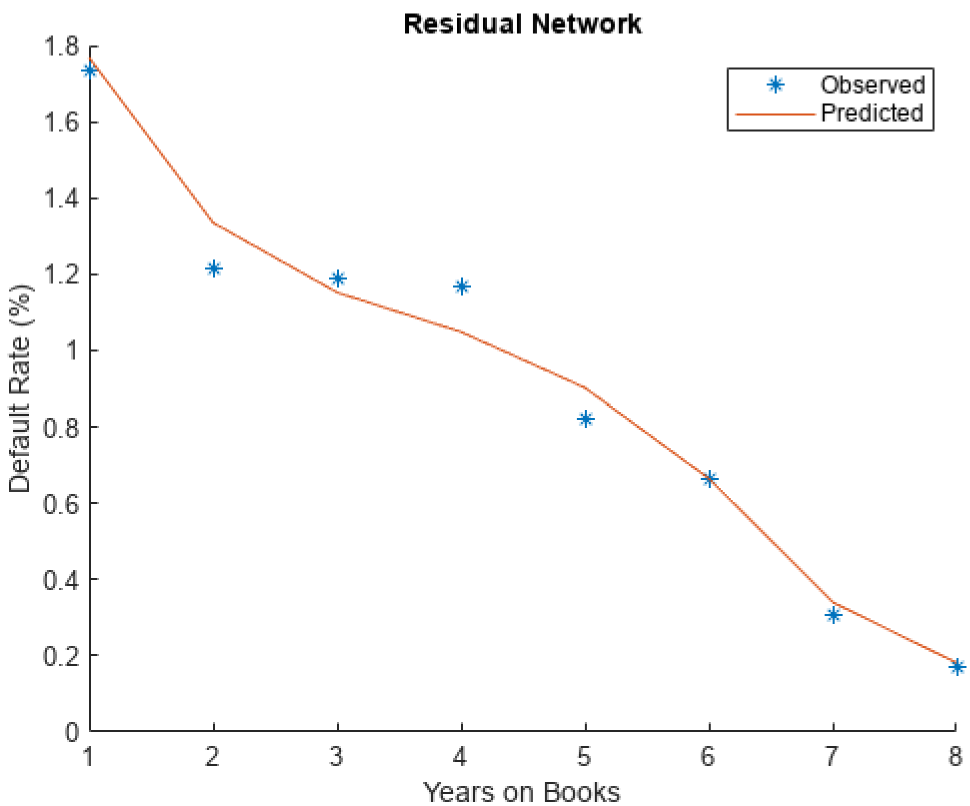
```
summaryYOB = groupsummary(tblTest,'YOB','mean',{'Default','residualPred'});
head(summaryYOB)
```

YOB	GroupCount	mean_Default	mean_residualPred
1	19364	0.017352	0.017688
2	18917	0.012158	0.013354
3	18526	0.011875	0.011522
4	18232	0.011683	0.010485
5	17925	0.0082008	0.0090247
6	17727	0.0066565	0.0066525

```
     7        12294        0.0030909            0.0034051
     8         6361        0.0017293            0.0018151
```

Plot the true average default rate against the average predictions by `YOB`.

```
figure
scatter(summaryYOB.YOB,summaryYOB.mean_Default*100,'*');
hold on
plot(summaryYOB.YOB,summaryYOB.mean_residualPred*100);
hold off
title('Residual Network')
xlabel('Years on Books')
ylabel('Default Rate (%)')
legend('Observed','Predicted')
```



The plot shows a good fit on the test data. The model seems to capture the overall trend as the age of the loan (`YOB` value) increases, as well as changes in the steepness of the trend.

The rest of this example shows some ways to better understand the model. First, it reviews standard explainability techniques that you can apply to this model, specifically, the `lime` and `shapley` functions. Then, it explores the behavior of the model in new (out-of-sample) data values. Finally, the example uses the model to predict PD values under stressed macroeconomic conditions, also known as stress testing.

Explain Model with LIME and Shapley

The local interpretable model-agnostic explanations (LIME) method and the Shapley method both aim to explain the behavior of the model at a particular observation of interest or "query point." More specifically, these techniques help you to understand the importance of each variable in the prediction made for a particular observation. For more information, see `lime` and `shapley`.

For illustration purposes, choose two observations from the data to better interpret the model predictions. The response values (last column) are not needed.

The first observation is a seasoned, low-risk loan. In other words, it has an initial score of `LowRisk` and eight years on the books.

```
obs1 = data(8,1:end-1);
disp(obs1)
```

```
    YOB    GDP     Market    HighRisk    MediumRisk    LowRisk
    ___    ____    ______    ________    __________    _______

     8     1.85     9.48        0            0            1
```

The second observation is a new, high-risk loan. That is, the score is `HighRisk` and it is in its first year on the books.

```
obs2 = data(88,1:end-1);
disp(obs2)
```

```
    YOB    GDP     Market    HighRisk    MediumRisk    LowRisk
    ___    ____    ______    ________    __________    _______

     1     2.72     7.61        1            0            0
```

Both `lime` and `shapley` require a reference data set with predictor values. This reference data can be the training data itself, or any other reference data where the model can be evaluated to explore the behavior of the model. More data points allow the explainability methods to understand the behavior of the model in more regions. However, a large data set can also slow down the computations, especially for `shapley`. For illustration purposes, use the first 1000 rows from the training data set. The response values (last column) are not needed.

```
predictorData = data(1:1000,1:end-1);
```

`lime` and `shapley` also require a function handle to the `predict` (Deep Learning Toolbox) function. Treat `predict` (Deep Learning Toolbox) like a black-box model and call it multiple times to make predictions on data and gather information on the behavior of the model.

```
blackboxFcn = @(x)predict(residualNetMacro,x);
```

Create `lime` Object

Create a `lime` object by passing the black-box function handle and the selected predictor data.

Randomly generated synthetic data underlying `lime` can affect the importance. The report may change depending on the synthetic data generated. It can also change due to optional arguments, such as the `'KernelWidth'` parameter that controls the area around the observation of interest ("query point") while you fit the local model.

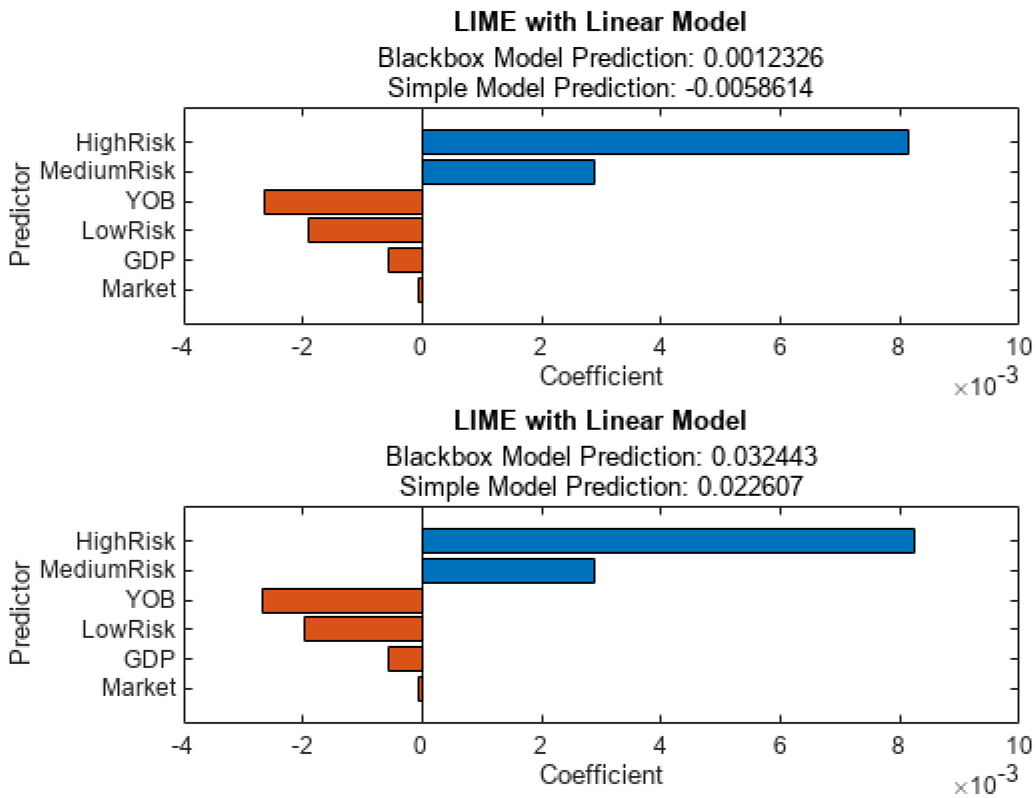
```
explainerLIME = lime(blackboxFcn,predictorData,'Type','regression');
```

Choose a number of important predictors of interest and fit a local model around the selected observations. For illustration purposes, the model contains all of the predictors.

```
numImportantPredictors = 6;
explainerObs1 = fit(explainerLIME,obs1,numImportantPredictors);
explainerObs2 = fit(explainerLIME,obs2,numImportantPredictors);
```

Plot the importance for each predictor.

```
figure
subplot(2,1,1)
plot(explainerObs1);
subplot(2,1,2)
plot(explainerObs2);
```



The `lime` results are quite similar for both observations. The information in the plots show that the most important variables are the `High Risk` and `Medium Risk` variables. `High Risk` and `Medium Risk` contribute positively to higher probabilities of default. On the other hand, `YOB`, `LowRisk`, `GDP`, and `Market` have a negative contribution to the default probability. The `Market` variable does not seem to contribute as much as the other variables. The values in the plots are coefficients of a simple model fitted around the point of interest, so the values can be interpreted as sensitivities of the PD to the different predictors, and these results seem to align with expectations. For example, PD predictions decrease as the `YOB` value (age of the loan) increases, consistent with the downward trend observed in the model fit plot in the Test Network on page 4-184 section.

Create shapley Object

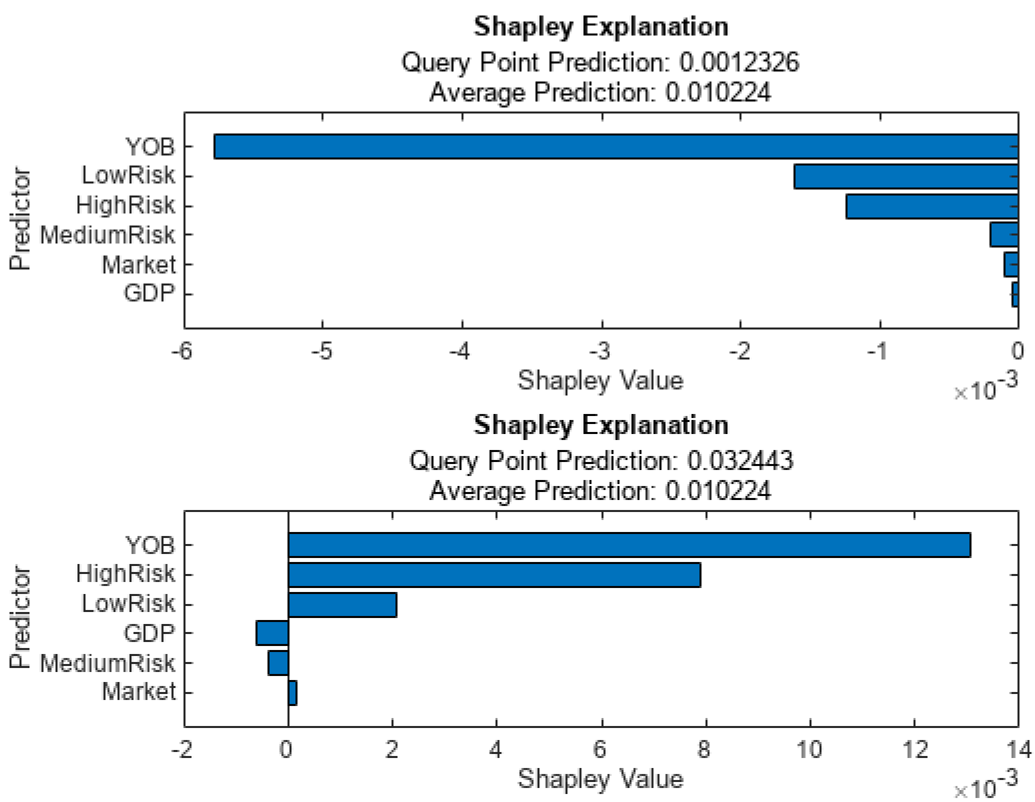
The steps for creating a `shapley` object are the same as for `lime`. Create a `shapley` object by passing the black-box function handle and the predictor data selected previously.

The `shapley` analysis can also be affected by randomly generated data, and it requires different methods to control the simulations required for the analysis. For illustration purposes, create the `shapley` object with default settings.

```
explainerShapley = shapley(blackboxFcn,predictorData);
```

Find and plot the importance of predictors for each query point. `shapley` is more computationally intensive than `lime`. As the number of rows in the predictor data increases, the computational time for the `shapley` results increases. For large data sets, using parallel computing is recommended (see the `'UseParallel'` option in `shapley`).

```
explainerShapleyObs1 = fit(explainerShapley, obs1);
explainerShapleyObs2 = fit(explainerShapley, obs2);
figure;
subplot(2,1,1)
plot(explainerShapleyObs1)
subplot(2,1,2)
plot(explainerShapleyObs2)
```



In this case, the results look different for the two observations. The `shapley` results explain the deviations from the average PD prediction. For the first observation, which is a very low risk observation, the predicted value is well below the average PD Therefore, all `shapley` values are negative, with `YOB` being the most important variable in this case, followed by `LowRisk`. For the second observation, which is a very high risk observation, most `shapley` values are positive, with `YOB` and `HighRisk` as the main contributors to a predicted PD well above average.

Explore Out-of-Sample Model Predictions

Splitting the original data set into training, validation, and testing helps prevent overfitting. However, the validation and test data sets share similar characteristics with the training data, for example, the range of values for YOB, or the observed values for the macroeconomic variables.

```
rangeYOB = [min(data.YOB) max(data.YOB)]
```

```
rangeYOB = 1×2

     1     8
```

```
rangeGDP = [min(data.GDP) max(data.GDP)]
```

```
rangeGDP = 1×2

   -0.5900    3.5700
```

```
rangeMarket = [min(data.Market) max(data.Market)]
```

```
rangeMarket = 1×2

  -22.9500   26.2400
```

You can explore the behavior of the out-of-sample (OOS) model in two different ways. First, you can predict for age values (YOB variable) larger than the maximum age value observed in the data. You can predict YOB values up to 15. Second, you can predict for economic conditions not observed in the data either. This example uses two extremely severe macroeconomic situations, where both the GDP and Market values are very negative and outside the range of values in the data.

Start by setting up a baseline scenario where the last macroeconomic data in the sample is used as reference. The YOB values go out of sample for all scenarios.

```
dataBaseline = table;
dataBaseline.YOB = repmat((1:15)',3,1);
dataBaseline.GDP = zeros(size(dataBaseline.YOB));
dataBaseline.Market = zeros(size(dataBaseline.YOB));
dataBaseline.HighRisk = zeros(size(dataBaseline.YOB));
dataBaseline.MediumRisk = zeros(size(dataBaseline.YOB));
dataBaseline.LowRisk = zeros(size(dataBaseline.YOB));

dataBaseline.GDP(:) = data.GDP(8);
dataBaseline.Market(:) = data.Market(8);
dataBaseline.HighRisk(1:15) = 1;
dataBaseline.MediumRisk(16:30) = 1;
dataBaseline.LowRisk(31:45) = 1;

disp(head(dataBaseline))
```

YOB	GDP	Market	HighRisk	MediumRisk	LowRisk
1	1.85	9.48	1	0	0
2	1.85	9.48	1	0	0
3	1.85	9.48	1	0	0

```
    4      1.85      9.48       1            0            0
    5      1.85      9.48       1            0            0
    6      1.85      9.48       1            0            0
    7      1.85      9.48       1            0            0
    8      1.85      9.48       1            0            0
```

Create two new extreme scenarios that include out-of-sample values not only for `YOB`, but also for the macroeconomic variables. This example uses pessimistic scenarios, but you could repeat the analysis for optimistic situations to explore the behavior of the model in either kind of extreme situation.

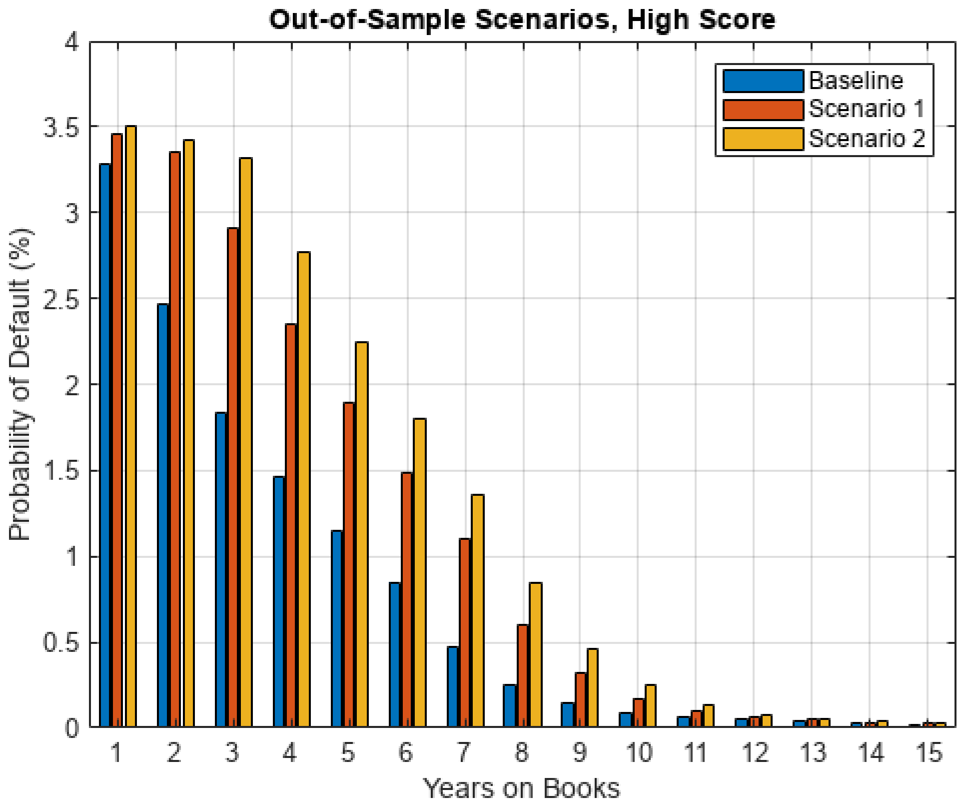
```
dataExtremeS1 = dataBaseline;
dataExtremeS1.GDP(:) = -1;
dataExtremeS1.Market(:) = -25;
dataExtremeS2 = dataBaseline;
dataExtremeS2.GDP(:) = -2;
dataExtremeS2.Market(:) = -40;
```

Predict PD values for all scenarios using `predict` (Deep Learning Toolbox).

```
dataBaseline.PD = predict(residualNetMacro,dataBaseline);
dataExtremeS1.PD = predict(residualNetMacro,dataExtremeS1);
dataExtremeS2.PD = predict(residualNetMacro,dataExtremeS2);
```

Visualize the results for a selected score. For convenience, the average of the PD values over the three scores is visualized as a summary.

```
ScoreSelected = High ;
switch ScoreSelected
   case 'High'
      ScoreInd = dataBaseline.HighRisk==1;
      PredPDYOB = [dataBaseline.PD(ScoreInd) dataExtremeS1.PD(ScoreInd) dataExtremeS2.PD(ScoreInd
   case 'Medium'
      ScoreInd = dataBaseline.MediumRisk==1;
      PredPDYOB = [dataBaseline.PD(ScoreInd) dataExtremeS1.PD(ScoreInd) dataExtremeS2.PD(ScoreInd
   case 'Low'
      ScoreInd = dataBaseline.LowRisk==1;
      PredPDYOB = [dataBaseline.PD(ScoreInd) dataExtremeS1.PD(ScoreInd) dataExtremeS2.PD(ScoreInd
   case 'Average'
      PredPDYOBBase = groupsummary(dataBaseline,'YOB','mean','PD');
      PredPDYOBS1 = groupsummary(dataExtremeS1,'YOB','mean','PD');
      PredPDYOBS2 = groupsummary(dataExtremeS2,'YOB','mean','PD');
      PredPDYOB = [PredPDYOBBase.mean_PD PredPDYOBS1.mean_PD PredPDYOBS2.mean_PD];
end

figure;
bar(PredPDYOB*100);
xlabel('Years on Books')
ylabel('Probability of Default (%)')
legend('Baseline','Scenario 1','Scenario 2')
title(strcat("Out-of-Sample Scenarios, ",ScoreSelected," Score"))
grid on
```



The overall results are in line with expectations, since the PD values decrease as the `YOB` value increases, and worse economic conditions result in higher PD values. However, the relative increase of the predicted PD values shows an interesting result. For `Low` and `Medium` scores, there is a significant increase for the first year on books (`YOB = 1`). In contrast, for `High` scores, the relative increase from baseline, to the first extreme scenario, then to the second extreme case, is small. This result suggests an implicit upper limit in the predicted values in the structure of the model. The extreme scenarios in this exercise seem unlikely to occur, however, for extreme but plausible scenarios, this behavior would require investigation with stress testing.

Stress-Test Predicted Probabilities of Default (PD)

Because the model includes macroeconomic variables, it can be used to perform a stress-testing analysis (see for example [2 on page 4-193], [3 on page 4-193] on page 4-193, [4 on page 4-193]). The steps are similar to the previous section except that the scenarios are plausible scenarios set periodically at an institution level, or set by regulators to be used by all institutions.

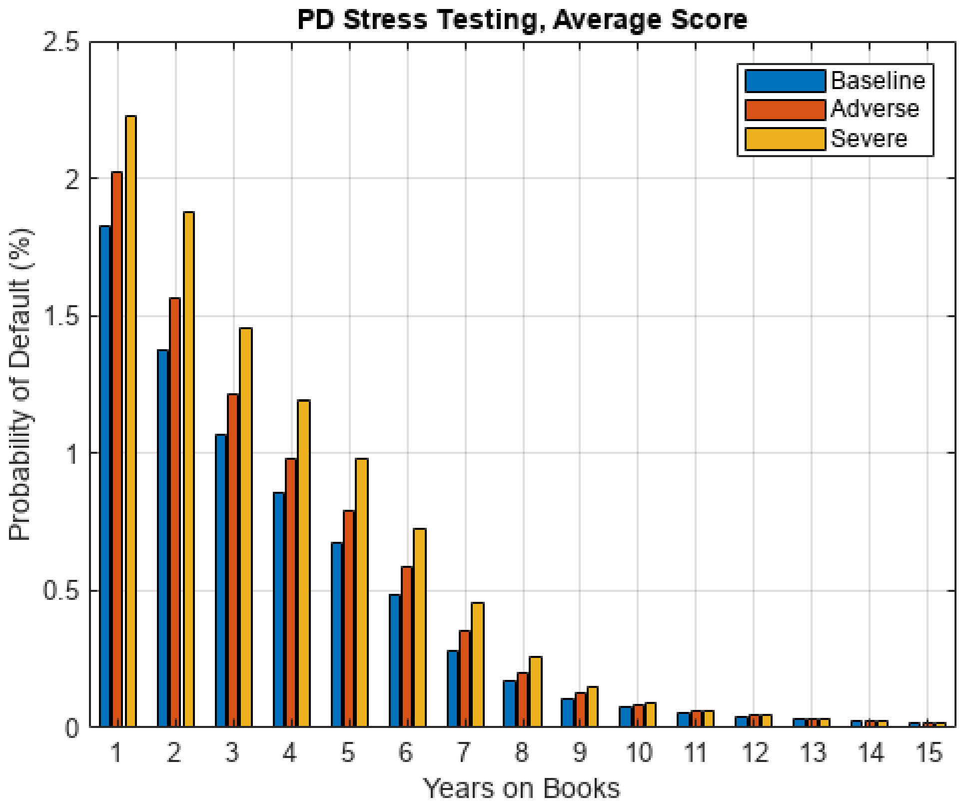
The `dataMacroStress` data set contains three scenarios for the stress testing of the model, namely, baseline, adverse, and severely adverse scenarios. The adverse and severe scenarios are relative to the baseline scenario, and the macroeconomic conditions are plausible given the baseline. These scenarios fall within the range of values observed in the data used for training and validation. The stress testing of the PD values for given macroeconomic scenarios is conceptually different from the exercise in the previous section, where the focus is on exploring the behavior of the model on out-of-sample data, regardless of how plausible those extreme scenarios are from an economic point of view.

Following the prior steps, you generate PD predictions for each score level and each scenario.

```
dataBaselineStress = dataBaseline(:,1:end-1);
dataAdverse = dataBaselineStress;
dataSevere = dataBaselineStress;

dataBaselineStress.GDP(:) = dataMacroStress{'Baseline','GDP'};
dataBaselineStress.Market(:) = dataMacroStress{'Baseline','Market'};

dataAdverse.GDP(:) = dataMacroStress{'Adverse','GDP'};
dataAdverse.Market(:) = dataMacroStress{'Adverse','Market'};

dataSevere.GDP(:) = dataMacroStress{'Severe','GDP'};
dataSevere.Market(:) = dataMacroStress{'Severe','Market'};
```

Use the `predict` (Deep Learning Toolbox) function to predict PD values for all scenarios. Visualize the results for a selected score.

```
dataBaselineStress.PD = predict(residualNetMacro,dataBaselineStress);
dataAdverse.PD = predict(residualNetMacro,dataAdverse);
dataSevere.PD = predict(residualNetMacro,dataSevere);

ScoreSelected = Average ;
switch ScoreSelected
   case 'High'
      ScoreInd = dataBaselineStress.HighRisk==1;
      PredPDYOBStress = [dataBaselineStress.PD(ScoreInd) dataAdverse.PD(ScoreInd) dataSevere.PD(
   case 'Medium'
      ScoreInd = dataBaselineStress.MediumRisk==1;
      PredPDYOBStress = [dataBaselineStress.PD(ScoreInd) dataAdverse.PD(ScoreInd) dataSevere.PD(
   case 'Low'
      ScoreInd = dataBaselineStress.LowRisk==1;
      PredPDYOBStress = [dataBaselineStress.PD(ScoreInd) dataAdverse.PD(ScoreInd) dataSevere.PD(
   case 'Average'
      PredPDYOBBaseStress = groupsummary(dataBaselineStress,'YOB','mean','PD');
      PredPDYOBAdverse = groupsummary(dataAdverse,'YOB','mean','PD');
      PredPDYOBSevere = groupsummary(dataSevere,'YOB','mean','PD');
      PredPDYOBStress = [PredPDYOBBaseStress.mean_PD PredPDYOBAdverse.mean_PD PredPDYOBSevere.mea
end

figure;
bar(PredPDYOBStress*100);
xlabel('Years on Books')
ylabel('Probability of Default (%)')
legend('Baseline','Adverse','Severe')
title(strcat("PD Stress Testing, ",ScoreSelected," Score"))
grid on
```



The overall results are in line with expectations. As in the Explore Out-of-Sample Model Predictions on page 4-189section, the predictions for the `High` score in the first year on books (`YOB = 1`) needs to be reviewed, since the relative increase in the predicted PD from one scenario to the next seems smaller than for other scores and loan ages. All other predictions show a reasonable pattern that are consistent with expectations.

References

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. pp. 770–778, 2016.

[2] Federal Reserve, Comprehensive Capital Analysis and Review (CCAR): https://www.federalreserve.gov/bankinforeg/ccar.htm

[3] Bank of England, Stress Testing: https://www.bankofengland.co.uk/financial-stability

[4] European Banking Authority, EU-Wide Stress Testing: https://www.eba.europa.eu/risk-analysis-and-data/eu-wide-stress-testing

See Also

More About

- "Get Started with Deep Network Designer" (Deep Learning Toolbox)

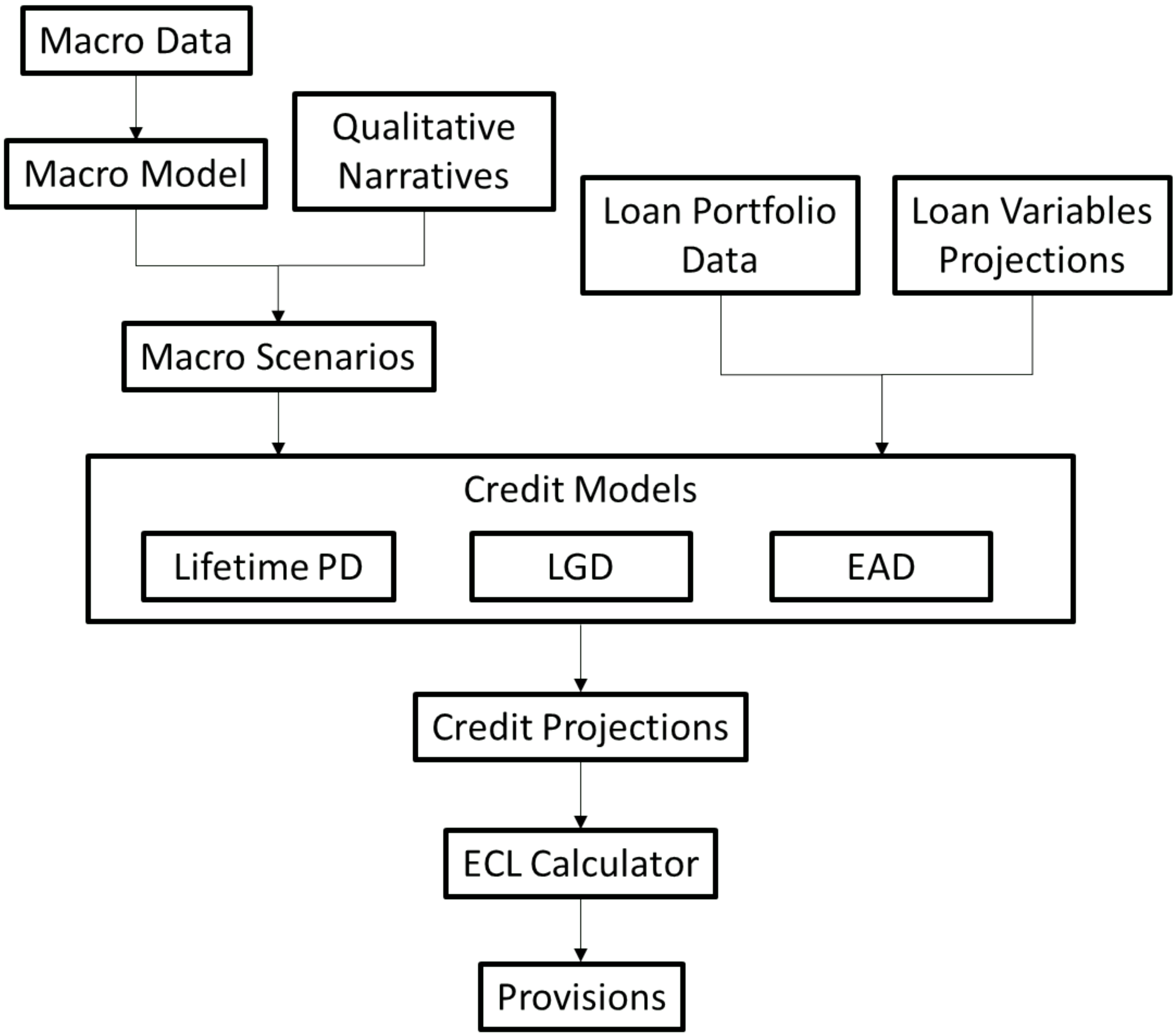
Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations

This example shows how to generate macroeconomic scenarios and perform expected credit loss (ECL) calculations for a portfolio of loans. The workflow in this example shows important computational steps required to estimate provisions for credit losses following regulations such as IFRS 9 on page 4-222 or CECL on page 4-222. In practice, the determination of loan provisions is a much more involved operational program requiring the collaboration of multiple departments in an institution. The goal of this example is to show useful computational tools that can support this process.

This example includes two parts:

- Part 1 on page 4-196 produces macroeconomic scenarios for the credit analysis and fits a vector autoregression (VAR) model to macroeconomic data. The workflow describes three qualitative scenario narratives with selected macro projections covering a few quarters ahead. It uses the conditional forecasting capabilities of the fitted VAR model to generate longer term projections for the macroeconomic variables.
- Part 2 on page 4-208 focuses on the credit analysis of an existing portfolio of loans. It uses three existing credit models are used, namely, a lifetime probability of default (PD) model, a loss given default (LGD) model, and an exposure at default (EAD) model. These models predict lifetime PD, LGD, and EAD values several periods ahead, given projections on loan-specific predictor variables and the macroeconomic scenarios from Part 1. The workflow estimates the loan loss provisions using the `portfolioECL` function, given the marginal PD, LGD, and EAD projections for different scenarios.

The following diagram summarizes the workflow in this example.



Part 1: Macroeconomic Scenarios

This part of the example shows how to fit an econometric model and how to use the fitted model with hypothesized conditions to produce macroeconomic scenarios.

In practice, the determination and approval of macroeconomic scenarios is an involved process. Institutions source macroeconomic scenarios from external vendors or they maintain internal macroeconomic models and produce scenarios. An institution can have existing models to generate scenarios, which are reviewed and updated infrequently. Some macroeconomic model developers use econometric models, while others prefer to explore alternative statistical models, such as machine learning or deep learning models. For more details, see “Sequence and Numeric Feature Data Workflows” (Deep Learning Toolbox).

For concreteness, the example fits a vector autoregressive (VAR) econometric model. For alternative econometric models, such as "Vector Error-Correction Models" (Econometrics Toolbox), see the example, "Model the United States Economy" (Econometrics Toolbox). Also, the Econometric Modeler (Econometrics Toolbox) enables you to interactively fit and analyze econometric models, including VAR and VEC models. For alternative time series modeling techniques, see Statistical and Machine Learning Toolbox™ and Deep Learning Toolbox™.

Part 1 describes the following workflow:

1. Visualize and transform data on page 4-197.
2. Conduct stationarity tests and select model lags on page 4-199.
3. Fit a VAR model on page 4-201.
4. Define macro scenarios on page 4-203.

This example uses the `Data_USEconModel.mat` data set included with Econometrics Toolbox. The data set includes several macroeconomic variables measured quarterly from March, 1947, through March, 2009 (near the start of the global financial crisis). For more details on the series, inspect the `Description` variable in the data file.

The measurements help emphasize the sensitivity of the credit projections to stressed macroeconomic conditions. You can apply the steps in this example to fit macroeconomic models using recent data.

Visualize and Transform Data

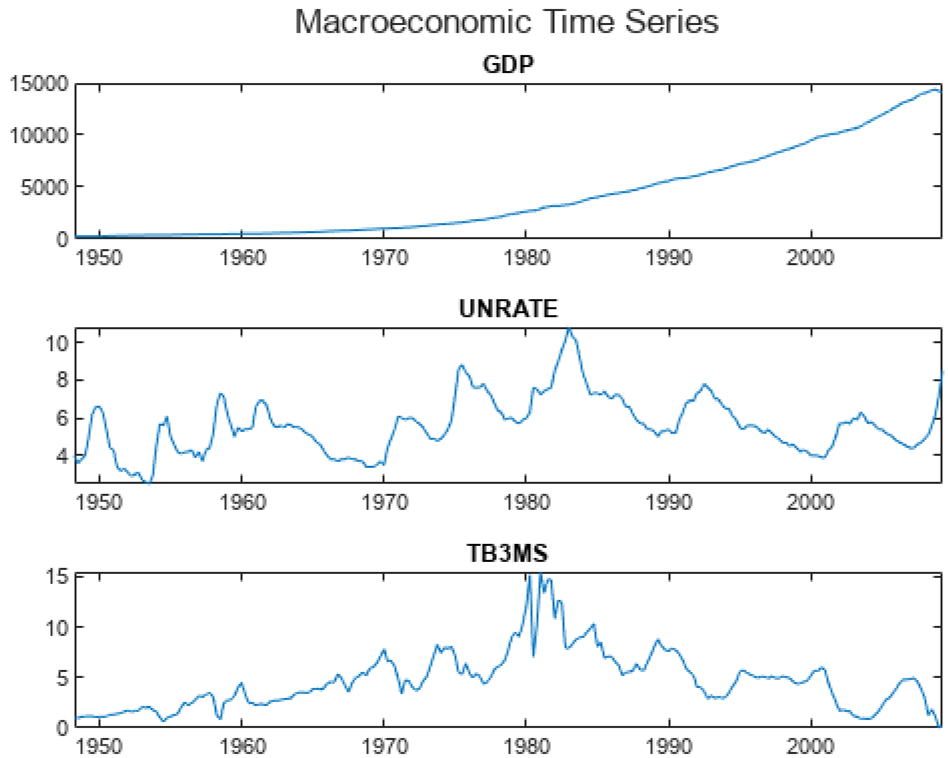
This example includes the following variables in the model:

- `GDP`: Gross domestic product
- `UNRATE`: Unemployment rate
- `TB3MS`: Three-month treasury bill yield

The variables sufficiently determine scenarios for the credit models in Part 2 of this example. However, the model can include additional variables, for example, variables that can explain the dynamics of the economy enough to improve model predictions.

Load the data. Visualize the raw macroeconomic time series.

```
load Data_USEconModel
varnames = ["GDP" "UNRATE" "TB3MS"];
DataMacro = rmmissing(DataTimeTable(:,varnames));  % Remove initial rows with UNRATE = NaN
DataMacro.Time.Format = "default";

figure
t = tiledlayout(3,1);
nexttile
plot(DataMacro.Time,DataMacro.GDP)
title("GDP")
nexttile
plot(DataMacro.Time,DataMacro.UNRATE)
title("UNRATE")
nexttile
plot(DataMacro.Time,DataMacro.TB3MS)
title("TB3MS")
title(t,"Macroeconomic Time Series")
```



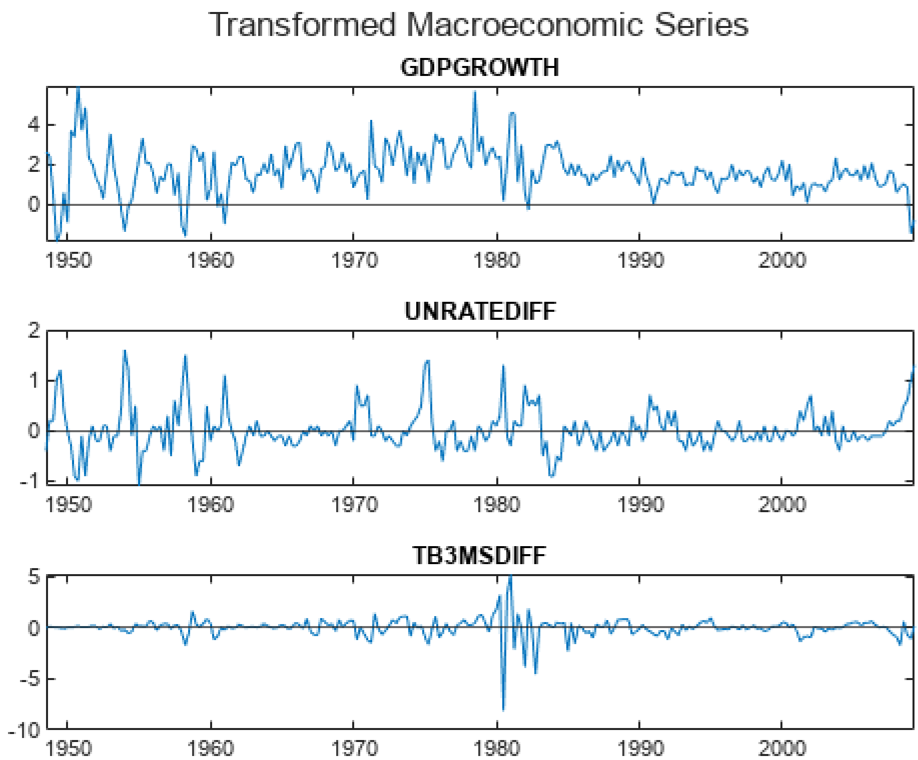
All series appear nonstationary and have different scales. Because model estimation with nonstationary variables is problematic (see "Time Series Regression IV: Spurious Regression" (Econometrics Toolbox)), transform each series appropriately.

- The GDP series shows exponential growth and has a larger scale than the other variables. Because GDP percent growth from quarter to quarter is common to report, is close in scale to the other variables, and is required by the credit models in Part 2, try to stabilize the series by computing its percent returns.
- The unemployment and interest rate series do not show exponential growth, but they appear to have a stochastic trend. You can stabilize the series by applying the first difference to each. The credit models in Part 2 use the original units of these series; you can back-transform the series to their original units for the credit analysis.

Transform each series, and plot the results. Include the transformed variables in the timetable, prepad each transformation with NaN to synchronize the series (the first difference operation of a series results in one less observation).

```
DataMacro.GDPGROWTH = 100*[NaN; price2ret(DataMacro.GDP)];  % Growth rate in percent
DataMacro.UNRATEDIFF = [NaN; diff(DataMacro.UNRATE)];
DataMacro.TB3MSDIFF = [NaN; diff(DataMacro.TB3MS)];
DataMacro = rmmissing(DataMacro);                           % Remove initial missing values
numObs = height(DataMacro);

figure
t = tiledlayout(3,1);
nexttile
```

```
plot(DataMacro.Time,DataMacro.GDPGROWTH)
yline(0)
title("GDPGROWTH")
nexttile
plot(DataMacro.Time,DataMacro.UNRATEDIFF)
yline(0)
title("UNRATEDIFF")
nexttile
plot(DataMacro.Time,DataMacro.TB3MSDIFF)
yline(0)
title("TB3MSDIFF")
title(t,"Transformed Macroeconomic Series")
```



The series appear stable. To confirm the visual conclusions, conduct stationarity tests on each series.

Conduct Stationarity Tests and Select Model Lags

To fit a VAR model, all variables must be stationarity. Econometrics Toolbox includes several tests for stationarity; see “Unit Root Tests” (Econometrics Toolbox).

This example uses the augmented Dickey-Fuller (ADF) test for unit root nonstationarity (see `adftest` (Econometrics Toolbox)). The null hypothesis of the test is that the input series is unit root nonstationary. Specified options determine the alternative hypothesis. The `Model` option of the ADF test specifies the dynamic structure of the alternative model for the test variable, specifically, autoregressive `AR`, autoregressive with drift `ARD`, or trend-stationary `TS`. The `Lags` option specifies the number of AR lags in the alternative model, specifically, the number p where y_{t-p} (the value of the series p periods back) influences the evolution of the series to the current step y_t. The ADF test results give information on whether the series is stationarity and on the structure of the autoregressive model that is appropriate for the series.

Use the ADF test to assess whether each series is stationary and to determine the appropriate dynamic model with number of lags for each series. The drop-down controls enable you to set model choices and variables for the ADF test. The `adftest` call runs separate tests from lags 0 through 4. The current settings in the code assesses whether `TB3MSDIFF` is stationary, assuming an `ARD` model.

```
ADFModelChoice = ARD ;
DataVariableChoice = TB3MSDIFF ;
ADFTbl = adftest(DataMacro,'model',ADFModelChoice,'Lags',0:4,'DataVariable',DataVariableChoice)
```

```
ADFTbl=5×8 table
               h      pValue      stat      cValue     Lags    Alpha     Model       Test
             _____    ______    _______    _______    ____    _____    _______    ______

    Test 1    true    0.001     -17.668    -2.8738     0      0.05     {'ARD'}    {'T1'}
    Test 2    true    0.001       -14.8    -2.8738     1      0.05     {'ARD'}    {'T1'}
    Test 3    true    0.001     -8.3548    -2.8739     2      0.05     {'ARD'}    {'T1'}
    Test 4    true    0.001     -7.6124    -2.8739     3      0.05     {'ARD'}    {'T1'}
    Test 5    true    0.001     -6.0646     -2.874     4      0.05     {'ARD'}    {'T1'}
```

`ADFTbl` is a table containing the results of all conducted tests for assessing whether `TB3MSDIFF` is stationary. Each row is a separate test for each lag in the `Lags` variable. Values of h indicate whether to reject the null hypothesis at `Alpha` level of significance. Small enough p-values in the `pValue` variable suggest whether to reject the null hypothesis. `h = 1` suggests to reject the null hypothesis in favor of the alternative, and `h = 0` suggests failure to reject the null hypothesis. In this case, `h = 1` for all lags, so there is evidence that the `TB3MSDIFF` series is a stationary, autoregressive process with drift among all lags.

For each untransformed and transformed series in this example, use the drop-down controls to assess whether the series is unit root nonstationary against each alternative model including the AR lags 1 through 4.

The results are as follows:

- `GDP`: The ADF test fails to reject the null hypothesis for all settings. Exclude this series from the VAR model.
- `GDPGROWTH`: The ADF test rejects the null hypothesis for all settings. Include this series in the VAR model.
- `UNRATE`: The tests return mixed results. The tests that use an `ARD` alternative model with lags 1 through 3 reject the null hypothesis. Therefore, the unemployment rate does not require transforming because it has a deterministic trend. Include the raw series in the VAR model with a drift term. Also, use of the raw series is convenient because the series does not need to be back-transformed; it simplifies the scenario determination.
- `TB3MS`: The ADF test fails to reject the null hypothesis for all settings. Exclude this series from the VAR model.
- `TB3MSDIFF`: The ADF test rejects the null hypothesis for all settings. Include this series in the VAR model.

In addition to using ADF test results, you can determine the number of lags by using Akaike information criterion (AIC), Bayesian information criterion (BIC), and likelihood ratio tests. For an example, see “Select Appropriate Lag Order” (Econometrics Toolbox). These approaches are useful, but they have caveats (for example, they are sensitive to the scaling of the series).

This example proceeds using two lags, which is reasonable from an application standpoint (it specifies to include information up to two quarters back). It leads to a less complex model, which is easier to analyze.

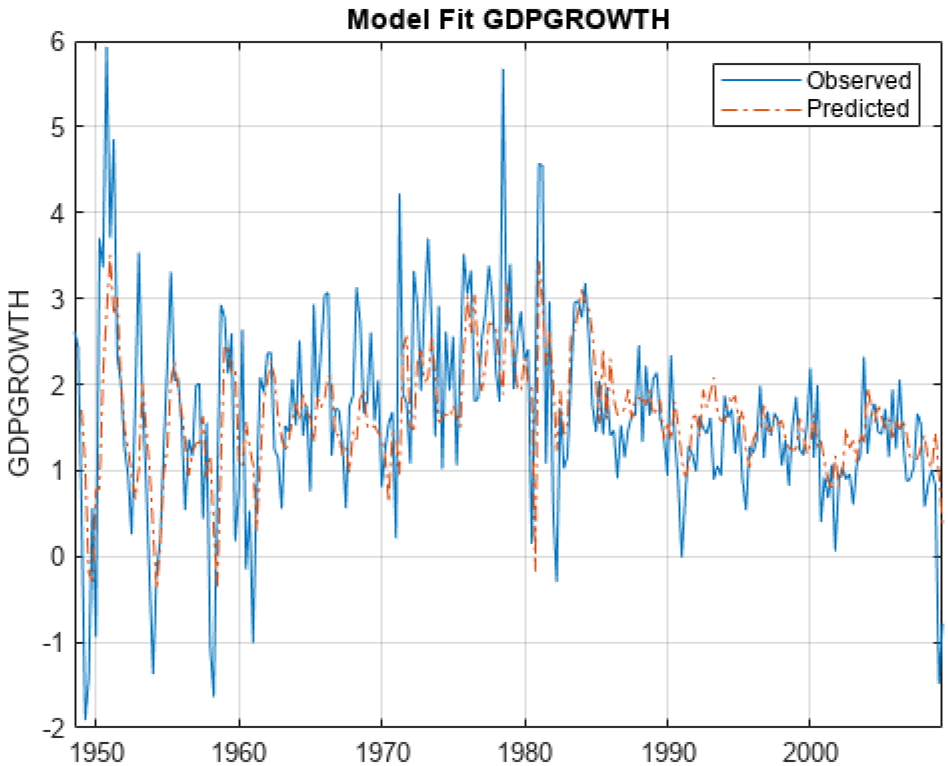
Fit VAR Model

Fit a VAR model that includes the GDP growth, unemployment rate, and the first difference of the interest rate time series as endogenous variables. Include two AR lags in the model. For details, see `varm` (Econometrics Toolbox) and `estimate` (Econometrics Toolbox).

```
varnames = ["GDPGROWTH" "UNRATE" "TB3MSDIFF"];
numVars = length(varnames);
numLags = 2;

% 3-D VAR(2) model template
Mdl = varm(numVars,numLags);
Mdl.SeriesNames = varnames;

% Fit the model
[MacroVARModel,~,~,Residuals] = estimate(Mdl,DataMacro{:,varnames});
```

Visualize the fitted values for each series by using the drop-down control.

```
numRes = size(Residuals,1);
DataMacro.GDPGROWTH_Predicted = NaN(numObs,1);
DataMacro.GDPGROWTH_Predicted(end-numRes+1:end) = DataMacro.GDPGROWTH(end-numRes+1:end) - Residua
DataMacro.UNRATE_Predicted = NaN(numObs,1);
DataMacro.UNRATE_Predicted(end-numRes+1:end) = DataMacro.UNRATE(end-numRes+1:end) - Residuals(:,2
DataMacro.TB3MSDIFF_Predicted = NaN(numObs,1);
DataMacro.TB3MSDIFF_Predicted(end-numRes+1:end) = DataMacro.TB3MSDIFF(end-numRes+1:end) - Residua

PlotVariableChoice = GDPGROWTH;

figure
plot(DataMacro.Time,DataMacro.(PlotVariableChoice))
hold on
plot(DataMacro.Time,DataMacro.(strcat(PlotVariableChoice,"_Predicted")),"-.")
hold off
title(strcat("Model Fit ",PlotVariableChoice))
ylabel(PlotVariableChoice)
legend("Observed","Predicted")
grid on
```



The model seems to have less variability than the actual data; the fitted values do not seem to go as high or as low as the most extreme values in the data. This result suggests the model error is smaller for small to medium GDP changes, but the errors are larger for more extreme GDP changes. Therefore, you can expect the model residuals to show heavy tails. A thorough analysis of residuals and autocorrelation is out of the scope of this example. For more details, see “Time Series Regression VI: Residual Diagnostics” (Econometrics Toolbox). This observation means the model seems more reliable for forecasts, which work with expected values (central tendencies around current levels), and less reliable for analyses of tail behavior (very extreme shocks away from current levels).

A review of model coefficients is an important model validation step. The way the series and their lagged information influence the latest values of the series should have a reasonable explanation.

Display the estimated coefficients and inferences.

```
summarize(MacroVARModel)
```

```
   AR-Stationary 3-Dimensional VAR(2) Model

    Effective Sample Size: 242
    Number of Estimated Parameters: 21
    LogLikelihood: -681.275
    AIC: 1404.55
    BIC: 1477.82

                     Value      StandardError    TStatistic      PValue
                  _________    _____________    __________    __________
```

```
    Constant(1)    -0.010079       0.26158        -0.03853        0.96926
    Constant(2)       0.3433      0.099669          3.4444     0.00057241
    Constant(3)      0.11773       0.25581         0.46024        0.64535
    AR{1}(1,1)       0.32725      0.073009          4.4823     7.3852e-06
    AR{1}(2,1)     -0.071852      0.027819         -2.5829      0.0097982
    AR{1}(3,1)      0.098221        0.0714          1.3756        0.16893
    AR{1}(1,2)      -0.22429       0.19147         -1.1714        0.24145
    AR{1}(2,2)        1.4205      0.072958           19.47     1.9537e-84
    AR{1}(3,2)      -0.50519       0.18725         -2.6979      0.0069782
    AR{1}(1,3)      0.092278      0.067309           1.371        0.17039
    AR{1}(2,3)      0.040064      0.025647          1.5621        0.11825
    AR{1}(3,3)      -0.30309      0.065826         -4.6045     4.1352e-06
    AR{2}(1,1)       0.15851      0.063174           2.509       0.012107
    AR{2}(2,1)      0.016333      0.024071         0.67854        0.49743
    AR{2}(3,1)       0.10361      0.061782          1.6771       0.093525
    AR{2}(1,2)       0.37458       0.19298           1.941       0.052255
    AR{2}(2,2)       -0.4632      0.073531         -6.2994     2.9882e-10
    AR{2}(3,2)       0.42524       0.18873          2.2532       0.024246
    AR{2}(1,3)      -0.15819        0.0655         -2.4152       0.015728
    AR{2}(2,3)       0.03071      0.024957          1.2305        0.21851
    AR{2}(3,3)       -0.3519      0.064056         -5.4936      3.939e-08

   Innovations Covariance Matrix:
    0.8575   -0.1668    0.2065
   -0.1668    0.1245   -0.1156
    0.2065   -0.1156    0.8201

   Innovations Correlation Matrix:
    1.0000   -0.5103    0.2463
   -0.5103    1.0000   -0.3616
    0.2463   -0.3616    1.0000
```

For example, for GDP growth, the diagonal coefficients are positive, with the first lag coefficient about twice as large as the second lag coefficient. This result means there is some inertia in the GDP series, some positive memory, and the more recent quarter has more influence than the growth two periods ago. For unemployment, the first lag diagonal coefficient is positive (and relatively large), but the second lag coefficient is negative. This result suggests that there is strong inertia for unemployment with respect to the previous quarter, but that there is a rebound effect with respect to two quarters before. Expert judgment is required to analyze model coefficients and raise flags in unexpected situations (a thorough analysis of the model coefficients and their especially statistical significance are beyond the scope of this example).

Define Macroeconomic Scenarios

Econometric model forecasts use the model parameters and recent observations of the time series to make predictions. The model forecast is a natural candidate for a macroeconomic scenario. However, the data and the model fully determine the forecasts, without qualitative views.

Forecast the estimated VAR model into a 30-period horizon by using `forecast` (Econometrics Toolbox). Initialize the forecasts by specifying the estimation data as a presample.

```
NumPeriods = 30;
ModelForecast = forecast(MacroVARModel,NumPeriods,DataMacro{:,varnames});
```

For later use by models and visualizations, append the latest observed quarter.

```
ModelForecast = [DataMacro{end,varnames}; ModelForecast];
fhTime = dateshift(DataMacro.Time(end),"end","quarter",0:NumPeriods)
```

```
fhTime = 1x31 datetime
   31-Mar-2009   30-Jun-2009   30-Sep-2009   31-Dec-2009   31-Mar-2010   30-Jun-2010   30-Sep-201
```

```
TTForecast = array2timetable(ModelForecast,RowTimes=fhTime,VariableNames=varnames);
```

The model forecasts interest rate changes to `TB3MSDIFF`. Postprocess the forecast to recover the predicted values for the interest rate in `TB3MS`.

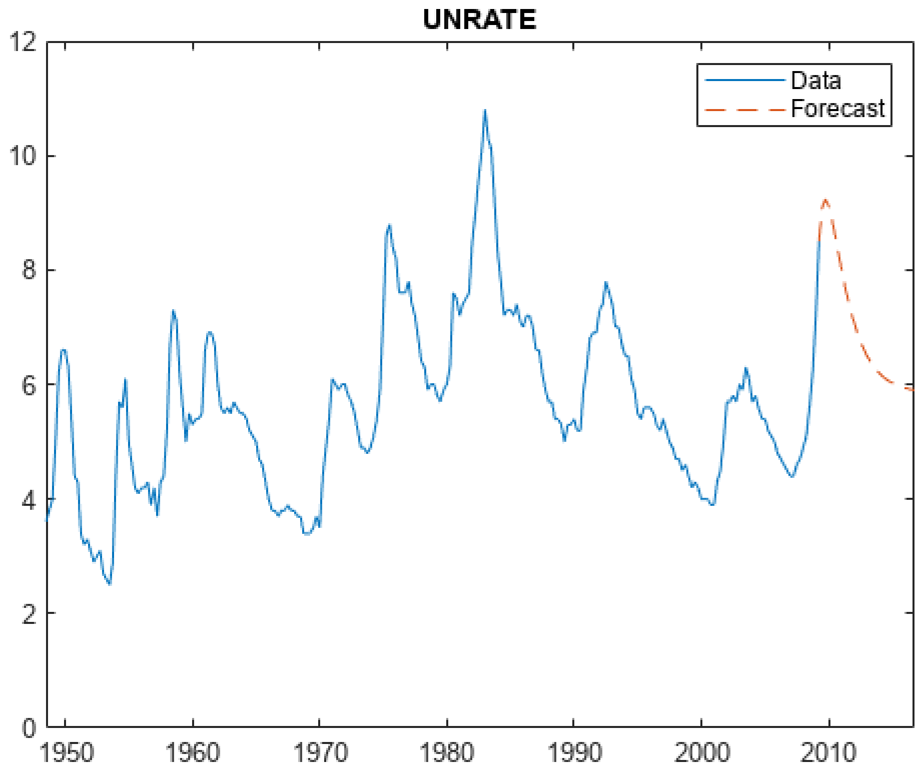
```
TTForecast.TB3MS = TTForecast.TB3MSDIFF;
TTForecast.TB3MS(1) = DataMacro.TB3MS(end);
TTForecast.TB3MS = cumsum(TTForecast.TB3MS)
```

TTForecast=*31×4 timetable*

Time	GDPGROWTH	UNRATE	TB3MSDIFF	TB3MS
31-Mar-2009	-0.78191	8.5	0.18	0.21
30-Jun-2009	0.47974	9.088	-1.0128	-0.80284
30-Sep-2009	1.0466	9.2336	-0.64918	-1.452
31-Dec-2009	1.842	9.1257	0.023293	-1.4287
31-Mar-2010	2.2753	8.8953	-0.055275	-1.484
30-Jun-2010	2.4409	8.6173	-0.072599	-1.5566
30-Sep-2010	2.5506	8.3213	0.063912	-1.4927
31-Dec-2010	2.5904	8.0292	0.087927	-1.4048
31-Mar-2011	2.556	7.7556	0.069531	-1.3352
30-Jun-2011	2.4975	7.5053	0.081449	-1.2538
30-Sep-2011	2.4306	7.28	0.085102	-1.1687
31-Dec-2011	2.3547	7.0803	0.074533	-1.0942
31-Mar-2012	2.2781	6.9051	0.067146	-1.027
30-Jun-2012	2.2065	6.7522	0.061337	-0.96567
30-Sep-2012	2.1401	6.6197	0.053405	-0.91227
31-Dec-2012	2.0797	6.5054	0.045859	-0.86641
⋮				

Visualize each model forecast by using the drop-down control.

```
VarToPlot = UNRATE ;
figure
plot(DataMacro.Time,DataMacro{:,VarToPlot})
hold on
plot(TTForecast.Time,TTForecast{:,VarToPlot},"--")
yline(0)
hold off
title(VarToPlot)
legend("Data","Forecast")
```



- The GDP growth reverts quickly to the average levels, with one more quarter of economic contraction, and growth rates over 1% by the third quarter.
- The unemployment rate peaks at just over 9% and then goes back to average levels, at a slower pace than the GDP.
- The model forecast predicts significantly negative interest rates going forward for a long period of time.

To define the scenarios for the credit analysis in Part 2, this example uses scenario narratives with qualitative views up to a few quarters ahead. This example combines the scenario narratives with conditional forecasting of the model to extend the predictions to several years ahead, where the predicted values revert to long-term levels. This approach can determine values for variables not described in the narrative, but are included in the macroeconomic model. For example, if only the GDP growth and unemployment rate are in the narrative, the corresponding interest rate scenario can be determined with conditional forecasting.

This example explores three scenarios: slow recovery, baseline, and fast recovery.

Slow Recovery Scenario

In this scenario, the GDP contracts for four more quarters, with a maximum contraction of 1.5% one quarter ahead. The unemployment rate peaks at 11% three quarters ahead. The interest rate remains negative for four quarters.

Forecast the model under this scenario. Preprocess the raw interest rate series of the estimated model, then postprocess their forecasts to the original units.

```
TTSlow = TTForecast; % Initialize forecast variables

TTSlow.GDPGROWTH(2:7) = [-1.5; -1.2; -0.8; -0.3; 0.1; 0.5];
TTSlow.GDPGROWTH(8:end) = NaN;

TTSlow.UNRATE(2:7) = [9.3; 10.1; 11.0; 10.5; 9.9; 9.1];
TTSlow.UNRATE(8:end) = NaN;

TTSlow.TB3MS(2:7) = [-0.5; -0.25; -0.15; -0.05; 0.0; 0.05];
TTSlow.TB3MS(8:end) = NaN;

TTSlow.TB3MSDIFF(2:end) = diff(TTSlow.TB3MS);
TTSlow{2:end,varnames} = forecast(MacroVARModel,NumPeriods,DataMacro{:,varnames},YF=TTSlow{2:end
TTSlow.TB3MS(8:end) = TTSlow.TB3MS(7) + cumsum(TTSlow.TB3MSDIFF(8:end));
```

Baseline Scenario

In this scenario, the GDP contracts for three more quarters, with a maximum contraction of 1% one quarter ahead. The unemployment rate peaks at 10% three quarters ahead. The interest rate remains negative for 2 quarters.

Forecast the model under this scenario. Preprocess the raw interest rate series using the estimated model, then postprocess their forecasts to the original units.

```
TTBaseline = TTForecast;

TTBaseline.GDPGROWTH(2:6) = [-1.0; -0.5; -0.25; 0.2; 0.6];
TTBaseline.GDPGROWTH(7:end) = NaN;

TTBaseline.UNRATE(2:6) = [9.0; 9.5; 10.0; 9.5; 9.0];
TTBaseline.UNRATE(7:end) = NaN;

TTBaseline.TB3MS(2:7) = [-0.25; -0.05; 0.01; 0.1; 0.15; 0.20];
TTBaseline.TB3MS(8:end) = NaN;

TTBaseline.TB3MSDIFF(2:end) = diff(TTBaseline.TB3MS);
TTBaseline{2:end,varnames} = forecast(MacroVARModel,NumPeriods,DataMacro{:,varnames},YF=TTBaseli
TTBaseline.TB3MS(8:end) = TTBaseline.TB3MS(7) + cumsum(TTBaseline.TB3MSDIFF(8:end));
```

Fast Recovery Scenario

In this scenario, the GDP contracts for one more quarter only, with a contraction of 0.5%. The unemployment rate peaks at 9% three quarters ahead. The interest rate is zero in the next quarter and remains positive after that. The fast recovery scenario is similar to the model forecast, but in the forecast the GDP grows faster and the interest rate is negative for a long period of time.

Forecast the model under this scenario. Preprocess the raw interest rate series using the estimated model, then postprocess their forecasts to the original units.

```
TTFast = TTForecast;

TTFast.GDPGROWTH(2:5) = [-0.5; 0.05; 0.5; 0.9];
TTFast.GDPGROWTH(6:end) = NaN;

TTFast.UNRATE(2:5) = [8.8; 8.9; 9.0; 8.7];
TTFast.UNRATE(6:end) = NaN;

TTFast.TB3MS(2:7) = [0; 0.1; 0.15; 0.25; 0.25; 0.30];
```

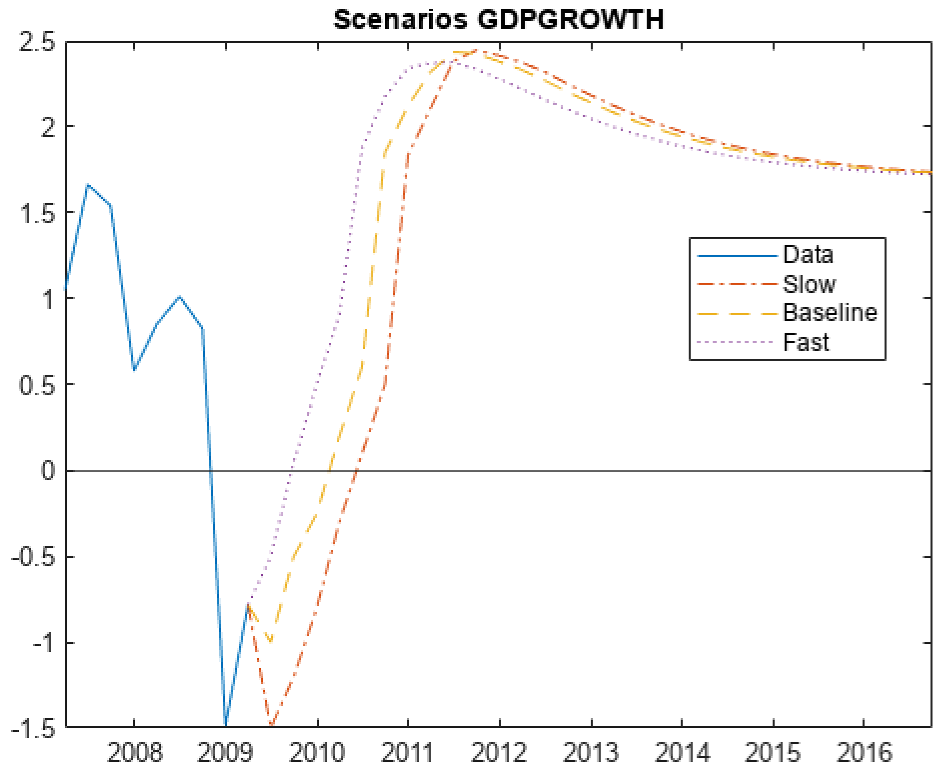
```
TTFast.TB3MS(8:end) = NaN;

TTFast.TB3MSDIFF(2:end) = diff(TTFast.TB3MS);
TTFast{2:end,varnames} = forecast(MacroVARModel,NumPeriods,DataMacro{:,varnames},YF=TTFast{2:end
TTFast.TB3MS(8:end) = TTFast.TB3MS(7) + cumsum(TTFast.TB3MSDIFF(8:end));
```

Assign a probability of 0.30 to the slow recovery scenario, 0.60 to the baseline scenario, and 0.10 to the fast recovery.

```
scenarioProb = [0.30; 0.60; 0.10];
```

Visualize the scenarios. Use the drop-down and slide controls to fine-tune the selected values that support the narratives. Visualize the long-term levels for different scenarios.

```
VarToPlot = "GDPGROWTH";
StartYearPlot = 2007;
EndYearPlot = 2016;

figure
DataPlotInd = (year(DataMacro.Time) >= StartYearPlot) & (year(DataMacro.Time) <= EndYearPlot);
ScenarioPlotInd = year(TTSlow.Time) <= EndYearPlot;
plot(DataMacro.Time(DataPlotInd),DataMacro{DataPlotInd,VarToPlot})
hold on
plot(TTSlow.Time(ScenarioPlotInd),TTSlow{ScenarioPlotInd,VarToPlot},"-.")
plot(TTBaseline.Time(ScenarioPlotInd),TTBaseline{ScenarioPlotInd,VarToPlot},"--")
plot(TTFast.Time(ScenarioPlotInd),TTFast{ScenarioPlotInd,VarToPlot},":")
yline(0)
hold off
title(strcat("Scenarios ",VarToPlot))
legend("Data","Slow","Baseline","Fast",Location="best")
```



The extended forecasts cross paths in some cases. For example, for GDP, the speed-up of the economy after the crisis occurs sooner in the fast recovery scenario, and then it starts settling down towards a long-term mean level while the economy is just speeding up for other scenarios.

The macroeconomic scenarios are defined. This example proceeds to the credit analysis, specifically, the computation of expected credit losses (ECL).

Part 2: ECL Calculations

The second part of this example focuses on the workflow for computing the lifetime expected credit loss (ECL). ECL is the amount of provisions required for loan losses.

Part 2 works with an existing portfolio of loans for which there are three existing credit models: a lifetime probability of default (PD) model, a loss given default (LGD) model, and an exposure at default (EAD) model. These models predict lifetime PD, LGD, and EAD values several periods ahead, given projections on loan-specific predictor variables and the macroeconomic scenarios produced in Part 1 on page 4-196.

Given the marginal PD, LGD, and EAD projections for different scenarios, you can use the `portfolioECL` function to estimate the lifetime loan loss provisions. This example presents a comparison of 1-year ECL vs. lifetime ECL and this corresponds to stage 1 vs. stage 2 of the IFRS 9 regulation. In this example, the estimated provisions for the lifetime ECL and 1-year ECL are very high because the three macroeconomic scenarios correspond to challenging economic conditions. For reference, this example also estimates 1-year provisions using an average macroeconomic scenario. This approach is similar to a through-the-cycle (TTC) method that shows in normal times the same models produce significantly lower provisions. You can use a visualization of credit projections and provisions for each ID to drill down to a loan level.

Part 2 describes the following workflow:

1. Prepare loan data for macroeconomic scenarios on page 4-209.
2. Predict lifetime PD, LGD. and EAD on page 4-211.
3. Compute lifetime ECL on page 4-213.
4. Compute 1-year ECL on page 4-215.
5. Compute 1-year ECL with average macroeconomic levels on page 4-217.
6. Visualize loan-level results on page 4-219.

Both the portfolio data and the models are simulated to capture characteristics that may be found in practice and are intended only for illustration purposes.

Prepare for Loan Data for Macroeconomic Scenarios

The `ECLPortfolioMacroExample.mat` file contains loan portfolio data. This data is a simulated existing portfolio of loans. The `ECLPortfolioSnapshot` table indicates which loans are in the portfolio.

```
load ECLPortfolioMacroExample.mat
```

```
head(ECLPortfolioSnapshot,5)
```

```
    ID       Time        Age    ScoreGroupOrig    Balance    Limit    EffIntRate
    __    ___________    ___    ______________    _______    _____    __________

    1     31-Mar-2009    12       LowRisk            3474     9000       5.04   
    2     31-Mar-2009     5       LowRisk          6232.7     9100       1.51   
    3     31-Mar-2009     3       HighRisk         2420.6     6000       3.13   
    4     31-Mar-2009     2       LowRisk          7274.2    10100       0.28   
    5     31-Mar-2009     1       MediumRisk       3611.8     5800       0.96   
```

The `ECLPortfolioProjections` table contains portfolio projections. These values are projected for the remaining life of each loan and for each of the loan variables. For example, the age (measured in quarters) is required for the lifetime PD and LGD models. Because the first loan is 12 quarters old, the projected age for the remaining periods starts at 13 and goes up by one all the way to the end of the loan, at 20 quarters (5 years). All models use the score group at origination and that is a constant value. The loan-to-value (LTV) ratio included in this simulated data set and models is the LTV at origination, which is constant. In practice, if the most recent LTV were included, it would have to be projected all the way through the end of the life of each loan.

```
head(ECLPortfolioProjections,10)
```

```
    ID    Age    ScoreGroupOrig    LTVOrig    Limit       Time    
    __    ___    ______________    _______    _____    ___________

    1     13        LowRisk        0.81866    9000     30-Jun-2009
    1     14        LowRisk        0.81866    9000     30-Sep-2009
    1     15        LowRisk        0.81866    9000     31-Dec-2009
    1     16        LowRisk        0.81866    9000     31-Mar-2010
    1     17        LowRisk        0.81866    9000     30-Jun-2010
    1     18        LowRisk        0.81866    9000     30-Sep-2010
    1     19        LowRisk        0.81866    9000     31-Dec-2010
```

```
    1     20        LowRisk        0.81866    9000     31-Mar-2011
    2      6        LowRisk        0.79881    9100     30-Jun-2009
    2      7        LowRisk        0.79881    9100     30-Sep-2009
```

The credit models need projected values of GDP growth, unemployment rate, and interest rate, all lagged by one period. First, store the macroeconomic information in the `MacroScenarios` table, where the macroeconomic scenarios are stacked.

```
ScenarioIDs = ["SlowRecovery";"Baseline";"FastRecovery"];
NumScenarios = length(ScenarioIDs);

MacroScenarios = table;
NumForecastPeriods = height(TTForecast)-1; % Remove initial period, keep future periods only
MacroScenarios.ScenarioID = repelem(ScenarioIDs,NumForecastPeriods);
MacroScenarios.Time = repmat(TTForecast.Time(2:end),NumScenarios,1);

MacroScenarios.GDPGROWTHLAG = NaN(height(MacroScenarios),1);
MacroScenarios.GDPGROWTHLAG(MacroScenarios.ScenarioID=="SlowRecovery") = TTSlow.GDPGROWTH(1:end-
MacroScenarios.GDPGROWTHLAG(MacroScenarios.ScenarioID=="Baseline") = TTBaseline.GDPGROWTH(1:end-
MacroScenarios.GDPGROWTHLAG(MacroScenarios.ScenarioID=="FastRecovery") = TTFast.GDPGROWTH(1:end-

MacroScenarios.UNRATELAG = NaN(height(MacroScenarios),1);
MacroScenarios.UNRATELAG(MacroScenarios.ScenarioID=="SlowRecovery") = TTSlow.UNRATE(1:end-1);
MacroScenarios.UNRATELAG(MacroScenarios.ScenarioID=="Baseline") = TTBaseline.UNRATE(1:end-1);
MacroScenarios.UNRATELAG(MacroScenarios.ScenarioID=="FastRecovery") = TTFast.UNRATE(1:end-1);

MacroScenarios.TB3MSLAG = NaN(height(MacroScenarios),1);
MacroScenarios.TB3MSLAG(MacroScenarios.ScenarioID=="SlowRecovery") = TTSlow.TB3MS(1:end-1);
MacroScenarios.TB3MSLAG(MacroScenarios.ScenarioID=="Baseline") = TTBaseline.TB3MS(1:end-1);
MacroScenarios.TB3MSLAG(MacroScenarios.ScenarioID=="FastRecovery") = TTFast.TB3MS(1:end-1);

head(MacroScenarios)
```

```
      ScenarioID          Time         GDPGROWTHLAG    UNRATELAG    TB3MSLAG
    ______________    ___________    ____________    _________    ________

    "SlowRecovery"    30-Jun-2009      -0.78191          8.5          0.21
    "SlowRecovery"    30-Sep-2009          -1.5          9.3          -0.5
    "SlowRecovery"    31-Dec-2009          -1.2         10.1         -0.25
    "SlowRecovery"    31-Mar-2010          -0.8           11         -0.15
    "SlowRecovery"    30-Jun-2010          -0.3         10.5         -0.05
    "SlowRecovery"    30-Sep-2010           0.1          9.9             0
    "SlowRecovery"    31-Dec-2010           0.5          9.1          0.05
    "SlowRecovery"    31-Mar-2011        1.8334       8.6536      -0.19292
```

To make predictions with the credit models, use the macroeconomic scenarios together with the loan data containing all the loan projection periods correctly aligned in time with the macroeconomic projected values. There are different ways to implement this. In this example, you can stack the portfolio projections three times, one for each scenario, and then join the stacked projections with the macro scenarios. Although this stacking method uses more memory, you can easily drill down to the loan level for different scenarios (see Loan-Level Results on page 4-219).

Stack portfolio projections by scenario in the `ECLProjectionsByScenario` table, and apply a join operation with the `MacroScenarios` table. The result is to add the macro variables to the larger table, using the `Time` and `ScenarioID` variables as keys for the join.

```
ECLProjectionsByScenario = repmat(ECLPortfolioProjections,NumScenarios,1);
ECLProjectionsByScenario = addvars(ECLProjectionsByScenario,repelem(ScenarioIDs,height(ECLPortfo
```

```
ECLProjectionsByScenario = join(ECLProjectionsByScenario,MacroScenarios);
head(ECLProjectionsByScenario,10)
```

```
      ScenarioID      ID    Age    ScoreGroupOrig    LTVOrig    Limit       Time        GDPGROWTH
    ______________    __    ___    ______________    _______    _____    ___________    _________

    "SlowRecovery"    1     13        LowRisk        0.81866    9000     30-Jun-2009      -0.7819
    "SlowRecovery"    1     14        LowRisk        0.81866    9000     30-Sep-2009           -1
    "SlowRecovery"    1     15        LowRisk        0.81866    9000     31-Dec-2009           -1
    "SlowRecovery"    1     16        LowRisk        0.81866    9000     31-Mar-2010           -0
    "SlowRecovery"    1     17        LowRisk        0.81866    9000     30-Jun-2010           -0
    "SlowRecovery"    1     18        LowRisk        0.81866    9000     30-Sep-2010            0
    "SlowRecovery"    1     19        LowRisk        0.81866    9000     31-Dec-2010            0
    "SlowRecovery"    1     20        LowRisk        0.81866    9000     31-Mar-2011         1.83
    "SlowRecovery"    2      6        LowRisk        0.79881    9100     30-Jun-2009      -0.7819
    "SlowRecovery"    2      7        LowRisk        0.79881    9100     30-Sep-2009           -1
```

Predict Lifetime PD, LGD, and EAD

Load the existing credit models and use them to make predictions using the prepared data. In this workflow, new credit models for PD, LGD, and EAD are not fit; the assumption is that these credit models were previously fit and reviewed and are ready to use for ECL calculations. The credit models are all instances of models created with Risk Management Toolbox™ using `fitLifetimePDModel`, `fitLGDModel`, and `fitEADModel`.

```
load ECLCreditModelsMacroExample.mat
```

The lifetime PD model is a `Probit` model with the predictors: age of the loan, score group at origination, LTV ratio at origination, and the lagged GDP growth and unemployment rate values.

```
disp(pdECLModel)
```

```
  Probit with properties:

            ModelID: "PD-ECL-Probit"
        Description: "Lifetime PD model example fitted to simulated data."
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "Age"
           LoanVars: ["ScoreGroupOrig"    "LTVOrig"]
          MacroVars: ["GDPGROWTHLAG"    "UNRATELAG"]
        ResponseVar: "Default"
```

The LGD model is a `Tobit` model with the predictors: age of the loan, score group at origination, LTV ratio at origination, and the lagged GDP growth.

```
disp(lgdECLModel)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "LGD-ECL-Tobit"
        Description: "LGD model example fitted to simulated data."
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["Age"    "ScoreGroupOrig"    "LTVOrig"    "GDPGROWTHLAG"]
        ResponseVar: "LGD"
```

The EAD model is a `Regression` model, based on the limit conversion factor (LCF) conversion measure, with the predictors: score group at origination, LTV ratio at origination, and the lagged unemployment rate and interest rate values. Because the underlying model predicts LCF, the EAD model requires the credit limit variable to make EAD predictions, even though the credit limit is not a predictor of the underlying LCF model.

```
disp(eadECLModel)
```

```
  Regression with properties:

    ConversionTransform: "logit"
      BoundaryTolerance: 1.0000e-07
                ModelID: "EAD-ECL-Regression"
            Description: "EAD model example fitted to simulated data."
        UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
          PredictorVars: ["ScoreGroupOrig"    "LTVOrig"    "UNRATELAG"    "TB3MSLAG"]
            ResponseVar: "EAD"
               LimitVar: "Limit"
               DrawnVar: ""
      ConversionMeasure: "lcf"
```

Store the predictions in the `ECLProjectionsByScenario` table. The predictions can also be stored in separate tables and used directly as inputs for the `portfolioECL` function. For an example of using separate tables, see “Calculate ECL Based on Marginal PD, LGD, and EAD Predictions” on page 6-497. Storing predictions in the `ECLProjectionsByScenario` table with scenarios stacked allows you to review prediction details at a very granular level because the predictions are stored side by side with projected predictor values in each row.

Store the lifetime (cumulative) PD and the marginal PD. Typically, the cumulative PD is used for reporting purposes and the marginal PD is used as an input for ECL computations with the `portfolioECL` function.

```
ECLProjectionsByScenario.PDLifetime = zeros(height(ECLProjectionsByScenario),1);
ECLProjectionsByScenario.PDMarginal = zeros(height(ECLProjectionsByScenario),1);
ECLProjectionsByScenario.LGD = zeros(height(ECLProjectionsByScenario),1);
ECLProjectionsByScenario.EAD = zeros(height(ECLProjectionsByScenario),1);

for ii=1:NumScenarios

   ScenIndECLData = ECLProjectionsByScenario.ScenarioID==ScenarioIDs(ii);

   ECLProjectionsByScenario.PDLifetime(ScenIndECLData) = predictLifetime(pdECLModel,ECLProjectio
   ECLProjectionsByScenario.PDMarginal(ScenIndECLData) = predictLifetime(pdECLModel,ECLProjectio
   ECLProjectionsByScenario.LGD(ScenIndECLData) = predict(lgdECLModel,ECLProjectionsByScenario(S
   ECLProjectionsByScenario.EAD(ScenIndECLData) = predict(eadECLModel,ECLProjectionsByScenario(S

end
head(ECLProjectionsByScenario,10)
```

```
      ScenarioID      ID    Age    ScoreGroupOrig    LTVOrig    Limit       Time        GDPGROWT
    ______________    __    ___    ______________    _______    _____    ___________    ________

    "SlowRecovery"    1     13        LowRisk        0.81866    9000     30-Jun-2009     -0.781
    "SlowRecovery"    1     14        LowRisk        0.81866    9000     30-Sep-2009         -1
    "SlowRecovery"    1     15        LowRisk        0.81866    9000     31-Dec-2009         -1
    "SlowRecovery"    1     16        LowRisk        0.81866    9000     31-Mar-2010         -0
    "SlowRecovery"    1     17        LowRisk        0.81866    9000     30-Jun-2010         -0
```

```
    "SlowRecovery"    1      18         LowRisk        0.81866    9000     30-Sep-2010         0
    "SlowRecovery"    1      19         LowRisk        0.81866    9000     31-Dec-2010         0
    "SlowRecovery"    1      20         LowRisk        0.81866    9000     31-Mar-2011       1.833
    "SlowRecovery"    2       6         LowRisk        0.79881    9100     30-Jun-2009     -0.7819
    "SlowRecovery"    2       7         LowRisk        0.79881    9100     30-Sep-2009         -1
```

Compute Lifetime ECL

Compute the lifetime ECL using the `portfolioECL` function. The inputs to this function are tables, where the first column is an ID variable that indicates which rows correspond to which loan. Because the projections cover multiple periods for each loan, and the remaining life of different loans may be different, the ID variable is an important input. Then, for each ID, the credit projections must be provided, period by period, until the end of the life of each loan. Typically, the marginal PD has a multi-period and multi-scenario size. However, in some situations, more commonly for LGD or EAD inputs, the credit projections may not have values for each period, or may not be sensitive to the scenarios. A typical case for this situation is a loan that repays the principal at maturity, where the exposure is considered constant for each period and independent of the macroeconomic scenarios. In this case, the EAD table input has one scalar value per ID that applies for all periods and all scenarios. To offer flexibility for different input dimensions for marginal PD, LGD, and EAD inputs, these inputs are separated into three separate tables in the syntax of `portfolioECL`.

Reformat the credit projections stored in the `ECLProjectionsByScenario` table into an intermediate table that includes the `Time` variable. You can use these intermediate tables to look at the detailed predictions period-by-period with different scenarios side-by-side. Also, these tables with a `Time` variable are useful for logical indexing to extract the predictions for only the first year.

```
PDMarginalUnstacked = ECLProjectionsByScenario(:,["ScenarioID" "ID" "Time" "PDMarginal"]);
PDMarginalUnstacked = unstack(PDMarginalUnstacked,"PDMarginal","ScenarioID");
PDMarginalUnstacked = movevars(PDMarginalUnstacked,"SlowRecovery","Before","Baseline");
disp(head(PDMarginalUnstacked))
```

```
    ID       Time        SlowRecovery    Baseline     FastRecovery
    __    ___________    ____________    _________    ____________

    1     30-Jun-2009     0.0092969      0.0092969      0.0092969
    1     30-Sep-2009      0.010737      0.0090867      0.0078466
    1     31-Dec-2009       0.01015      0.0076948       0.005958
    1     31-Mar-2010     0.0096304       0.006847      0.0045918
    1     30-Jun-2010     0.0064312      0.0045587      0.0032064
    1     30-Sep-2010     0.0042121      0.0030213      0.0019542
    1     31-Dec-2010     0.0025718      0.0016771      0.0013505
    1     31-Mar-2011     0.0013569      0.0011649     0.00096951
```

```
LGDUnstacked = ECLProjectionsByScenario(:,["ScenarioID" "ID" "Time" "LGD"]);
LGDUnstacked = unstack(LGDUnstacked,"LGD","ScenarioID");
LGDUnstacked = movevars(LGDUnstacked,"SlowRecovery","Before","Baseline");

EADUnstacked = ECLProjectionsByScenario(:,["ScenarioID" "ID" "Time" "EAD"]);
EADUnstacked = unstack(EADUnstacked,"EAD","ScenarioID");
EADUnstacked = movevars(EADUnstacked,"SlowRecovery","Before","Baseline");
```

The final step to preparing the inputs for `portfolioECL` is to remove the `Time` column. There are other ways to prepare these inputs, for example, you can store the columns side-by-side directly during prediction. In this example, the extra steps for preparation are illustrated so that you can use the `ECLProjectionsByScenario` table for detailed analysis at a loan level (see Loan-Level Results on page 4-219).

```
PDMarginalLifetimeInput = PDMarginalUnstacked(:,[1 3:end]);
disp(head(PDMarginalLifetimeInput))
```

```
    ID    SlowRecovery    Baseline     FastRecovery
    __    ____________    _________    ____________

    1      0.0092969      0.0092969      0.0092969
    1       0.010737      0.0090867      0.0078466
    1        0.01015      0.0076948       0.005958
    1      0.0096304       0.006847      0.0045918
    1      0.0064312      0.0045587      0.0032064
    1      0.0042121      0.0030213      0.0019542
    1      0.0025718      0.0016771      0.0013505
    1      0.0013569      0.0011649     0.00096951
```

```
LGDLifetimeInput = LGDUnstacked(:,[1 3:end]);
EADLifetimeInput = EADUnstacked(:,[1 3:end]);
```

Each loan has an effective interest rate for discounting that is determined at loan recognition.

```
EIRInput = ECLPortfolioSnapshot(:,["ID" "EffIntRate"]);
EIRInput.EffIntRate = EIRInput.EffIntRate/100; % Convert to decimal
```

Use the `portfolioECL` function.

```
[totalECL,idECL,periodECL] = portfolioECL(PDMarginalLifetimeInput,LGDLifetimeInput,EADLifetimeInp
   ScenarioNames=ScenarioIDs,ScenarioProbabilities=scenarioProb,InterestRate=EIRInput);
```

The `totalECL` output is the lifetime ECL of the portfolio which is the total provisions amount required by the portfolio.

```
fprintf('Total Portfolio Lifetime ECL: %s',cur2str(totalECL))
```

```
Total Portfolio Lifetime ECL: $67701.21
```

The `idECL` output is the lifetime ECL at a loan level. Join the `idECL` output with the score group and the balance information from the ECL portfolio snapshot table. With this data, you can compute the lifetime ECL as a percent of the loan balance. You can also aggregate by score group. The `TotalsByScore` table shows that the lifetime ECL is significantly higher for lower quality score groups.

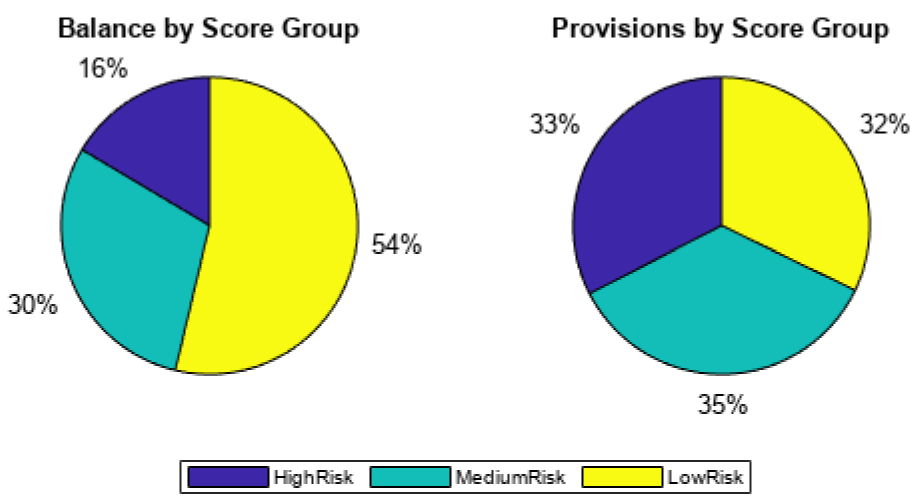
```
idECL = join(idECL,ECLPortfolioSnapshot(:,["ID" "ScoreGroupOrig" "Balance"]));
idECL.ECLPercent = 100*idECL.ECL./idECL.Balance;
TotalsByScore = groupsummary(idECL,"ScoreGroupOrig","sum",["ECL" "Balance"]);
TotalsByScore.ECLPercent = 100*TotalsByScore.sum_ECL./TotalsByScore.sum_Balance
```

```
TotalsByScore=3×5 table
    ScoreGroupOrig    GroupCount    sum_ECL    sum_Balance    ECLPercent
    ______________    __________    _______    ___________    __________

      HighRisk            76         22027      1.828e+05        12.05
      MediumRisk         121         23921      3.312e+05       7.2225
      LowRisk            130         21753     5.9629e+05       3.6481
```

The provisions in `TotalsByScore` table are very high. These results are explored in the Compute 1-Year ECL on page 4-215 and Compute 1-Year ECL with Average Macroeconomic Levels on page 4-217 sections.

Visualize the distribution of the portfolio balance and the distribution of provisions by score group. Only about 1/6 of the assets are allocated to high-risk loans, yet the provisions for high risk make up about 1/3 of the total provisions. On the other end, more than half of the assets are allocated to low-risk loans, yet the provisions for low risk are less than 1/3 of the total provisions.

```
figure
tiledlayout(1,2)
nexttile
pie(TotalsByScore.sum_Balance)
title('Balance by Score Group')
nexttile
pie(TotalsByScore.sum_ECL)
title('Provisions by Score Group')
leg = legend(TotalsByScore.ScoreGroupOrig,'Location','south','Orientation','horizontal');
leg.Layout.Tile = 'south';
```



Compute 1-Year ECL

As noted in Compute Lifetime ECL on page 4-213, the lifetime provisions in `TotalsByScore` table are very high. To better understand why, you can compute 1-year provisions to understand the impact of the lifetime part of the ECL beyond the first year.

The difference between lifetime ECL and 1-year ECL is important for the IFRS 9 regulation where stage 1 loans (performing loans) use 1-year ECL for provisioning, whereas stage 2 loans (increased credit risk) and stage 3 loans (credit impaired) use lifetime ECL. [1 on page 4-222]

To obtain a 1-year ECL, prepare the marginal PD, LGD, and EAD input tables so that they cover one year ahead. You can leverage the unstacked tables created in Compute Lifetime ECL on page 4-213. These unstacked tables contain a `Time` variable to do logical indexing and obtain the inputs for a 1-year ECL. In the new inputs, each ID has only four periods and any loan with a remaining life of less than four quarters has fewer periods.

```
FourthQuarterAhead = ECLPortfolioProjections.Time(4);
PDMarginal1YearInput = PDMarginalUnstacked(PDMarginalUnstacked.Time<=FourthQuarterAhead,[1 3:end]
disp(head(PDMarginal1YearInput))
```

```
    ID    SlowRecovery     Baseline     FastRecovery
    __    ____________    _________    ____________

    1      0.0092969      0.0092969     0.0092969
    1       0.010737      0.0090867     0.0078466
    1        0.01015      0.0076948      0.005958
    1      0.0096304       0.006847     0.0045918
    2        0.02793        0.02793       0.02793
    2       0.030942       0.026838      0.023675
    2       0.028822       0.022818      0.018364
    2       0.026954       0.020285      0.014466
```

```
LGD1YearInput = LGDUnstacked(LGDUnstacked.Time<=FourthQuarterAhead,[1 3:end]);
EAD1YearInput = EADUnstacked(EADUnstacked.Time<=FourthQuarterAhead,[1 3:end]);
```

Use `portfolioECL` with the new inputs.

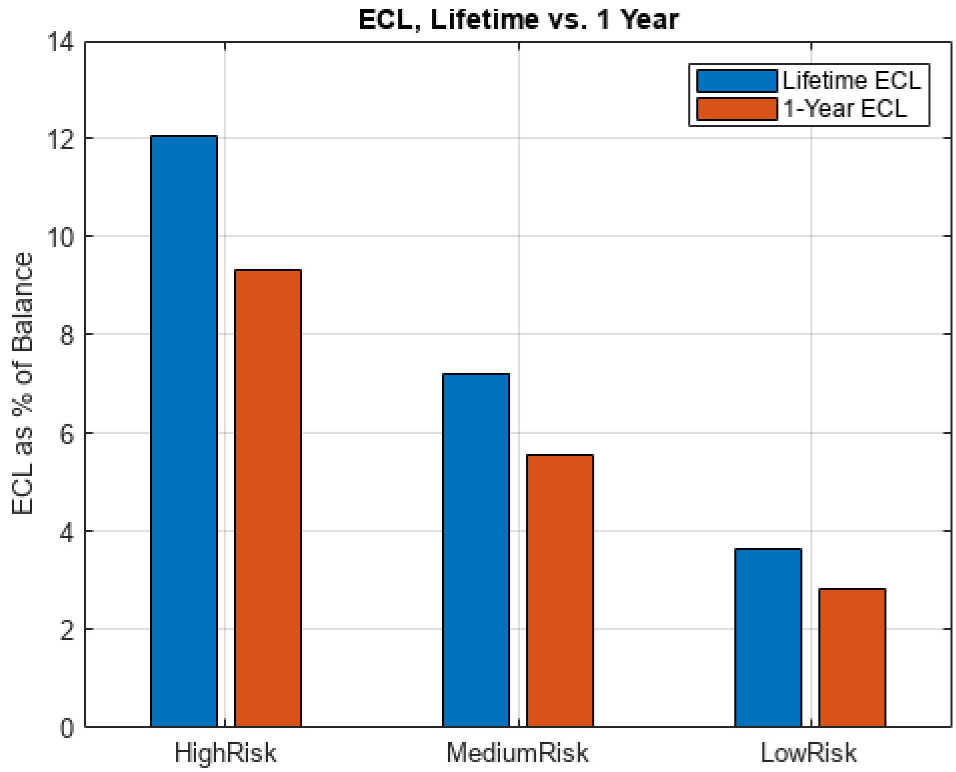
```
[totalECL1Year,idECL1Year,periodECL1Year] = portfolioECL(PDMarginal1YearInput,LGD1YearInput,EAD1Y
   ScenarioNames=ScenarioIDs,ScenarioProbabilities=scenarioProb,InterestRate=EIRInput);
```

Expand the ID level ECL table with the score group and balance information.

```
idECL1Year.ScoreGroupOrig = idECL.ScoreGroupOrig;
idECL1Year.Balance = idECL.Balance;
idECL1Year.ECLPercent = 100*idECL1Year.ECL./idECL1Year.Balance;

TotalsByScore1Year = groupsummary(idECL1Year,"ScoreGroupOrig","sum",["ECL" "Balance"]);
TotalsByScore1Year.ECLPercent = 100*TotalsByScore1Year.sum_ECL./TotalsByScore1Year.sum_Balance;
```

Compare the 1-year ECL with the lifetime ECL by plotting the lifetime and 1-year ECL values as a proportion of balance for each score group.

```
figure;
bar([TotalsByScore.ECLPercent TotalsByScore1Year.ECLPercent])
xticklabels(TotalsByScore.ScoreGroupOrig)
ylabel('ECL as % of Balance')
legend('Lifetime ECL','1-Year ECL')
title('ECL, Lifetime vs. 1 Year')
grid on
```



The lifetime ECL shows an important increase with respect to the 1-year ECL. However, the 1-year ECL values are high. In general, the increment from 1-year ECL to lifetime ECL is not expected to be large because marginal PD values tend to decrease as loans age, and in some cases, the risk implied by LGD and EAD projections can also decrease with time. An example of this behavior is an amortizing loan.

The macroeconomic scenarios in this example are extremely adverse and the provisions are conditional on the macroeconomic scenarios. Compute 1-Year ECL with Average Macroeconomic Levels on page 4-217 explores the impact of these adverse scenarios on the ECL estimates.

Compute 1-Year ECL with Average Macroeconomic Levels

To compare the 1-year ECL with more normal macroeconomic conditions, the macroeconomic scenarios from Define Macroeconomic Scenarios on page 4-203 are replaced with a long-term average of the macroeconomic variables. To simplify, use the long-term average of the macroeconomic variables to make predictions only four quarters ahead. This approach is similar to a through-the-cycle (TTC) reserving approach because the credit projections reflect average macroeconomic conditions.

Use the mean macroeconomic levels over the entire sample.

```
MeanGDPGROWTH = mean(DataMacro.GDPGROWTH);
MeanUNRATE = mean(DataMacro.UNRATE);
MeanTB3MS = mean(DataMacro.TB3MS);
```

To make predictions, you need the projected predictors values and data only one year ahead.

```
ECLPortfolio1YearMacroAverage = ECLPortfolioProjections(ECLPortfolioProjections.Time<=FourthQuart
ECLPortfolio1YearMacroAverage.GDPGROWTHLAG = zeros(height(ECLPortfolio1YearMacroAverage),1);
ECLPortfolio1YearMacroAverage.UNRATELAG = zeros(height(ECLPortfolio1YearMacroAverage),1);
ECLPortfolio1YearMacroAverage.TB3MSLAG = zeros(height(ECLPortfolio1YearMacroAverage),1);
```

Append the macroeconomic average values and use the same average value for all periods going forward.

```
ECLPortfolio1YearMacroAverage.GDPGROWTHLAG(:) = MeanGDPGROWTH;
ECLPortfolio1YearMacroAverage.UNRATELAG(:) = MeanUNRATE;
ECLPortfolio1YearMacroAverage.TB3MSLAG(:) = MeanTB3MS;
```

Predict credit values. You need only the marginal PD values for ECL calculations, but you can also store the lifetime PD values for the analysis in the Loan-Level Results on page 4-219 section.

```
ECLPortfolio1YearMacroAverage.PDLifetime = predictLifetime(pdECLModel,ECLPortfolio1YearMacroAvera
ECLPortfolio1YearMacroAverage.PDMarginal = predictLifetime(pdECLModel,ECLPortfolio1YearMacroAvera
ECLPortfolio1YearMacroAverage.LGD = predict(lgdECLModel,ECLPortfolio1YearMacroAverage);
ECLPortfolio1YearMacroAverage.EAD = predict(eadECLModel,ECLPortfolio1YearMacroAverage);
```

In this case, the inputs for the `portfolioECL` function have only one scenario.

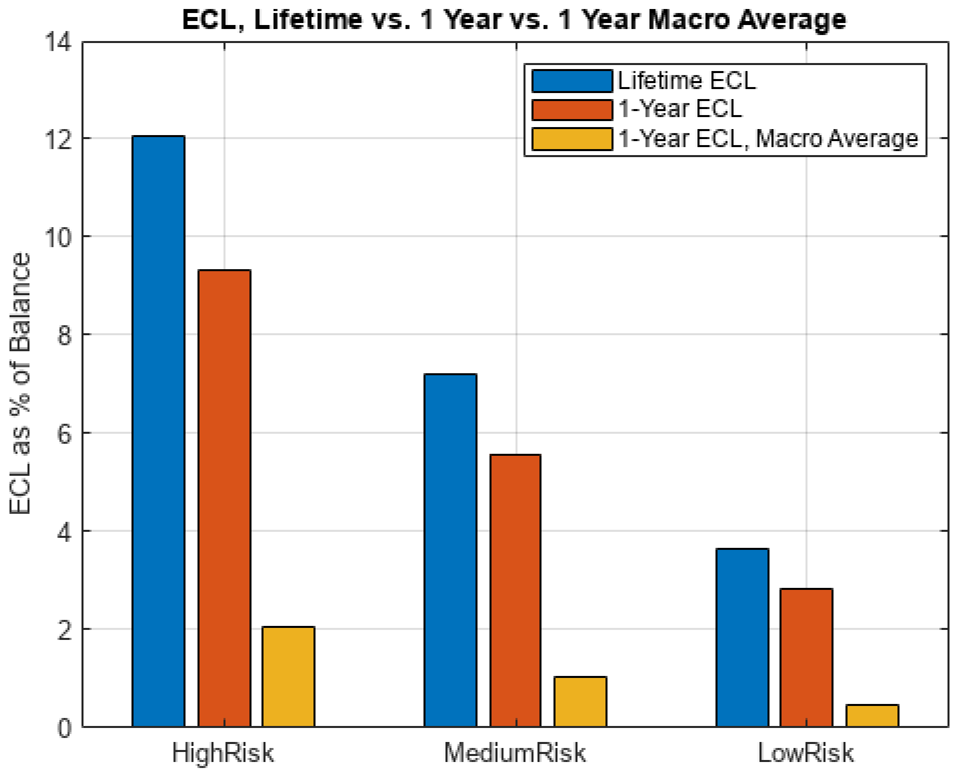
```
PDMarginal1YearMacroAverageInput = ECLPortfolio1YearMacroAverage(:,["ID" "PDMarginal"]);
LGD1YearMacroAverageInput = ECLPortfolio1YearMacroAverage(:,["ID" "LGD"]);
EAD1YearMacroAverageInput = ECLPortfolio1YearMacroAverage(:,["ID" "EAD"]);
```

Obtain the 1-year ECL and expand the loan-level results table.

```
[totalECL1YearMacroAverage,idECL1YearMacroAverage,periodECL1YearMacroAverage] = portfolioECL(PDMa

idECL1YearMacroAverage.ScoreGroupOrig = idECL.ScoreGroupOrig;
idECL1YearMacroAverage.Balance = idECL.Balance;
idECL1YearMacroAverage.ECLPercent = 100*idECL1YearMacroAverage.ECL./idECL1YearMacroAverage.Balanc

TotalsByScore1YearMacroAverage = groupsummary(idECL1YearMacroAverage,"ScoreGroupOrig","sum",["ECL
TotalsByScore1YearMacroAverage.ECLPercent = 100*TotalsByScore1YearMacroAverage.sum_ECL./TotalsByS
```

Compare the 1-year ECL with average macroeconomic values to the 1-year ECL and lifetime ECL approach using the initial macroeconomic scenarios.

```
figure;
bar([TotalsByScore.ECLPercent TotalsByScore1Year.ECLPercent TotalsByScore1YearMacroAverage.ECLPer
xticklabels(TotalsByScore.ScoreGroupOrig)
ylabel('ECL as % of Balance')
legend('Lifetime ECL','1-Year ECL','1-Year ECL, Macro Average')
title('ECL, Lifetime vs. 1 Year vs. 1 Year Macro Average')
grid on
```



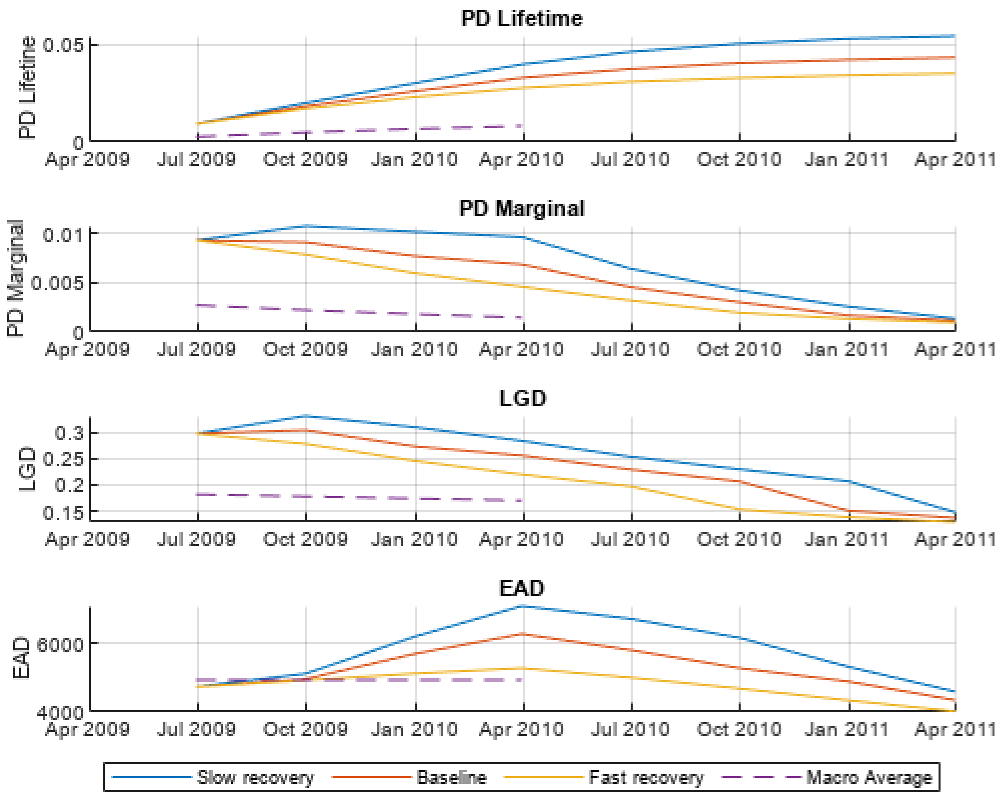
These results show that the severely adverse macroeconomic scenarios defined in Define Macroeconomic Scenarios on page 4-203 drive the high provisions. The 1-year ECL values with average macroeconomic levels are much lower with the low-risk 1-year ECL at 0.5% of the current provisions balance. The medium-risk 1-year ECL is at 1% of the current provisions balance and the high-risk 1-year ECL is 2% of the current provisions balance.

Visualize Loan-Level Results

You can explore the ECL predictions and the results at a loan level. Use `SelectedID` to enter any loan ID in the portfolio. The resulting visualizations show the predicted lifetime PD, marginal PD, LGD, and EAD over the remaining life of the loan. The plot shows the predictions for each macroeconomic scenario defined in Define Macroeconomic Scenarios on page 4-203 as well as the macroeconomic average scenario (1-year predictions only).

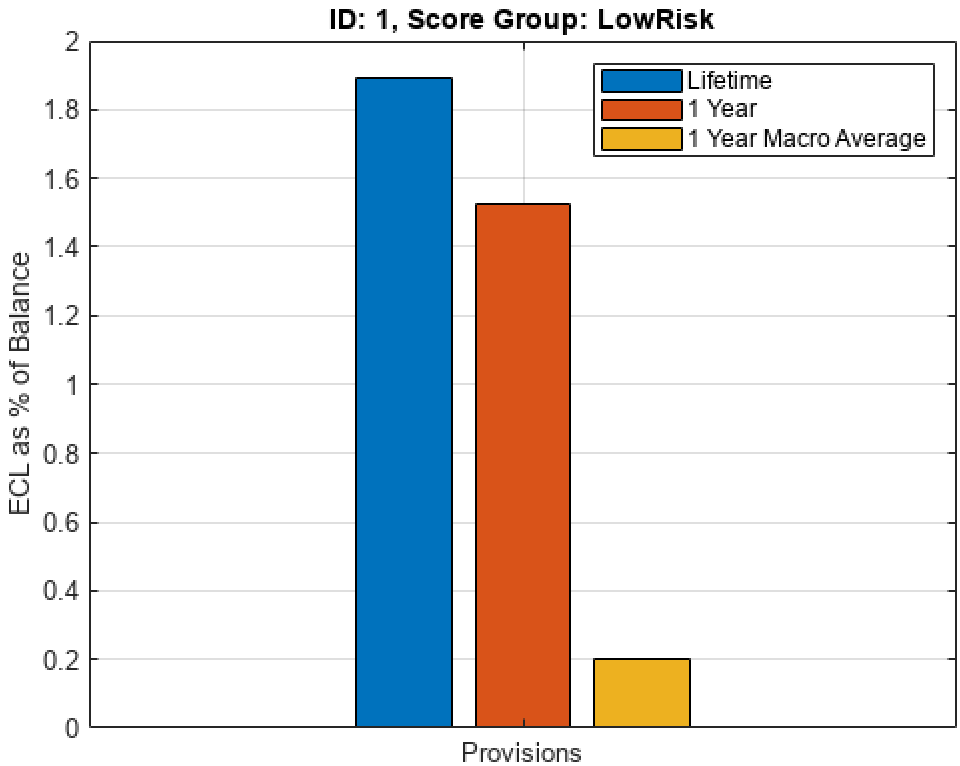
```
SelectedID = 1 ;
IDDataLifetime = ECLProjectionsByScenario(ECLProjectionsByScenario.ID==SelectedID,:);
IDData1YearMacroAverage = ECLPortfolio1YearMacroAverage(ECLPortfolio1YearMacroAverage.ID==Selecte

figure;
t = tiledlayout(4,1);
nexttile
hold on
for ii=1:NumScenarios
   ScenPlotInd = IDDataLifetime.ScenarioID==ScenarioIDs(ii);
   plot(IDDataLifetime.Time(ScenPlotInd),IDDataLifetime.PDLifetime(ScenPlotInd))
end
```

```
plot(IDData1YearMacroAverage.Time,IDData1YearMacroAverage.PDLifetime,'--')
hold off
ylabel('PD Lifetime')
title('PD Lifetime')
grid on

nexttile
hold on
for ii=1:NumScenarios
   ScenPlotInd = IDDataLifetime.ScenarioID==ScenarioIDs(ii);
   plot(IDDataLifetime.Time(ScenPlotInd),IDDataLifetime.PDMarginal(ScenPlotInd))
end
plot(IDData1YearMacroAverage.Time,IDData1YearMacroAverage.PDMarginal,'--')
hold off
ylabel('PD Marginal')
title('PD Marginal')
grid on

nexttile
hold on
for ii=1:NumScenarios
   ScenPlotInd = IDDataLifetime.ScenarioID==ScenarioIDs(ii);
   plot(IDDataLifetime.Time(ScenPlotInd),IDDataLifetime.LGD(ScenPlotInd))
end
plot(IDData1YearMacroAverage.Time,IDData1YearMacroAverage.LGD,'--')
hold off
ylabel('LGD')
title('LGD')
grid on

nexttile
hold on
for ii=1:NumScenarios
   ScenPlotInd = IDDataLifetime.ScenarioID==ScenarioIDs(ii);
   plot(IDDataLifetime.Time(ScenPlotInd),IDDataLifetime.EAD(ScenPlotInd))
end
plot(IDData1YearMacroAverage.Time,IDData1YearMacroAverage.EAD,'--')
hold off
ylabel('EAD')
title('EAD')
grid on

leg = legend("Slow recovery","Baseline","Fast recovery","Macro Average","Orientation","horizonta
leg.Layout.Tile = 'south';
```



The following plot shows the ECL for the loan, the ECL for the lifetime case, the 1-year ECL case, and the 1-year ECL using average macroeconomic values.

```
figure;
IDECLInd = idECL.ID == SelectedID;
bar(categorical("Provisions"),[idECL.ECLPercent(IDECLInd) idECL1Year.ECLPercent(IDECLInd) idECL1Y
legend('Lifetime','1 Year','1 Year Macro Average')
ylabel('ECL as % of Balance')
TitleStr = sprintf('ID: %g, Score Group: %s',SelectedID,idECL.ScoreGroupOrig(IDECLInd));
title(TitleStr)
grid on
```



Conclusion

This example covers an entire workflow, including the determination of macroeconomic scenarios and the estimation of provisions using lifetime ECL computations. The example also shows some tools and visualizations to analyze the results at a portfolio level, score-group level, and loan level.

The example demonstrates tools from the Econometrics Toolbox™ and Risk Management Toolbox™ that support this workflow, including macroeconomic modeling tools such as vector autoregressive (VAR) models, and credit risk tools such as the lifetime probability of default (PD) models (`fitLifetimePDModel`), loss given default (LGD) models (`fitLGDModel`), exposure at default (EAD) models (`fitEADModel`), and the `portfolioECL` function.

References

[1] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* Elsevier, 2019.

[2] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Prescient Models LLC, 2018.

[3] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`fitLifetimePDModel` | `fitEADModel` | `fitLGDModel` | `portfolioECL`

Related Examples

- "Expected Credit Loss Computation" on page 4-124
- "Modeling Probabilities of Default with Cox Proportional Hazards" on page 4-28

Create Custom Lifetime PD Model for Decision Tree Model with Function Handle

This example shows how to fit a decision tree model for credit scoring and then use the `customLifetimePDModel` object to create a lifetime model for probability of default.

Fit a Decision Tree Model for Credit Scoring

Load the credit scorecard data using a data set from Refaat [1 on page 4-230]. The data set in this example contains one row per loan.

```
load CreditCardData.mat
disp(head(data))
```

```
    CustID    CustAge    TmAtAddress    ResStatus     EmpStatus    CustIncome    TmWBank    Other
    ______    _______    ___________    __________    _________    __________    _______    _____

      1         53           62         Tenant        Unknown        50000         55         Yes
      2         61           22         Home Owner    Employed       52000         25         Yes
      3         47           30         Tenant        Employed       37000         61         No 
      4         50           75         Home Owner    Employed       53000         20         Yes
      5         68           56         Home Owner    Employed       53000         14         Yes
      6         65           13         Home Owner    Employed       48000         59         Yes
      7         34           32         Home Owner    Unknown        32000         26         Yes
      8         50           57         Other         Employed       51000         33         No 
```

Fit a decision tree model using `fitctree` from Statistics and Machine Learning Toolbox™. The data set in this example contains 1200 observations, which is not a large number. This example uses the data to train the model, but you can split larger data sets into training and testing sets.

```
CategoricalPreds = {'ResStatus','EmpStatus','OtherCC'};
dt = fitctree(data,'status~CustAge+TmAtAddress+ResStatus+EmpStatus+CustIncome+TmWBank+OtherCC+Uti
    'MaxNumSplits',30,'CategoricalPredictors',CategoricalPreds);
disp(dt)
```

```
  ClassificationTree
           PredictorNames: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tr
             ResponseName: 'status'
    CategoricalPredictors: [3 4 7]
               ClassNames: [0 1]
           ScoreTransform: 'none'
          NumObservations: 1200
```

```
view(dt)
```

```
Decision tree for classification
 1  if CustIncome<30500 then node 2 elseif CustIncome>=30500 then node 3 else 0
 2  if TmWBank<60 then node 4 elseif TmWBank>=60 then node 5 else 1
 3  if TmWBank<32.5 then node 6 elseif TmWBank>=32.5 then node 7 else 0
 4  if TmAtAddress<13.5 then node 8 elseif TmAtAddress>=13.5 then node 9 else 1
 5  if UtilRate<0.255 then node 10 elseif UtilRate>=0.255 then node 11 else 0
 6  if CustAge<60.5 then node 12 elseif CustAge>=60.5 then node 13 else 0
 7  if CustAge<46.5 then node 14 elseif CustAge>=46.5 then node 15 else 0
 8  if CustIncome<24500 then node 16 elseif CustIncome>=24500 then node 17 else 1
 9  if TmWBank<56.5 then node 18 elseif TmWBank>=56.5 then node 19 else 1
10  if CustAge<21.5 then node 20 elseif CustAge>=21.5 then node 21 else 0
```

```
11  class = 1
12  if EmpStatus=Employed then node 22 elseif EmpStatus=Unknown then node 23 else 0
13  if TmAtAddress<131 then node 24 elseif TmAtAddress>=131 then node 25 else 0
14  if TmAtAddress<97.5 then node 26 elseif TmAtAddress>=97.5 then node 27 else 0
15  class = 0
16  class = 0
17  if ResStatus in {Home Owner Tenant} then node 28 elseif ResStatus=Other then node 29 else 1
18  if TmWBank<52.5 then node 30 elseif TmWBank>=52.5 then node 31 else 0
19  class = 1
20  class = 1
21  class = 0
22  if UtilRate<0.375 then node 32 elseif UtilRate>=0.375 then node 33 else 0
23  if UtilRate<0.005 then node 34 elseif UtilRate>=0.005 then node 35 else 0
24  if CustIncome<39500 then node 36 elseif CustIncome>=39500 then node 37 else 0
25  class = 1
26  if UtilRate<0.595 then node 38 elseif UtilRate>=0.595 then node 39 else 0
27  class = 1
28  class = 1
29  class = 0
30  class = 1
31  class = 0
32  class = 0
33  if UtilRate<0.635 then node 40 elseif UtilRate>=0.635 then node 41 else 0
34  if CustAge<49 then node 42 elseif CustAge>=49 then node 43 else 1
35  if CustIncome<57000 then node 44 elseif CustIncome>=57000 then node 45 else 0
36  class = 1
37  class = 0
38  class = 0
39  if CustIncome<34500 then node 46 elseif CustIncome>=34500 then node 47 else 1
40  class = 1
41  class = 0
42  class = 1
43  class = 0
44  class = 0
45  class = 1
46  class = 0
47  class = 1
```

The decision tree `predict` function returns a predicted `class` in the first output, where `class = 0` means no-default, and `class = 1` means default (same as the response data that you used to train the model). The `predict` function also returns the corresponding prediction `scores` or class probabilities as the second output.

In this example, you are interested in the probability of default, which is the class probability for `class = 1` (the second column of the class probability output).

```
[~,ObservationClassProb] = predict(dt,data);
pdDT = ObservationClassProb(:,2);
```

Wrap Decision Tree Model as Lifetime PD Model

To wrap the decision tree model as a lifetime PD model, a function handle to a PD prediction function is required. For the decision tree in this example, the `predict` method of the decision tree does not return the PD values directly. Therefore, first create a helper function that takes the decision tree model and the data as inputs and returns the PD predictions. This helper function is implemented as `myDTPredictFcn` in Local Functions on page 4-230. Then define a function handle to this function, `predictFcnHandle`, that takes data as input and returns the PD.

```
predictFcnHandle = @(data)myDTPredictFcn(dt,data);
```

Create an instance of a custom lifetime PD model by passing the function handle to `customLifetimePDModel`. You need to specify variable names using name-value arguments because these variable names are used by the base class `LifetimePDModel`.

```
pdModel = customLifetimePDModel(predictFcnHandle,'ModelID','MyDTModel','IDVar','CustID','LoanVars
```

```
pdModel = 
  CustomLifetimePD with properties:

            ModelID: "MyDTModel"
        Description: ""
    UnderlyingModel: @(data)myDTPredictFcn(dt,data)
              IDVar: "CustID"
             AgeVar: ""
           LoanVars: ["CustAge"    "TmAtAddress"    "ResStatus"    "EmpStatus"    "CustIncome"
          MacroVars: ""
        ResponseVar: "status"
```

Predict and Validate Scores Using the Custom Lifetime PD Model

Use the `predict` function of the lifetime PD model to make PD predictions.

```
CondPD = predict(pdModel,data);
```

The predictions are the same as predicting directly with the original decision tree model.

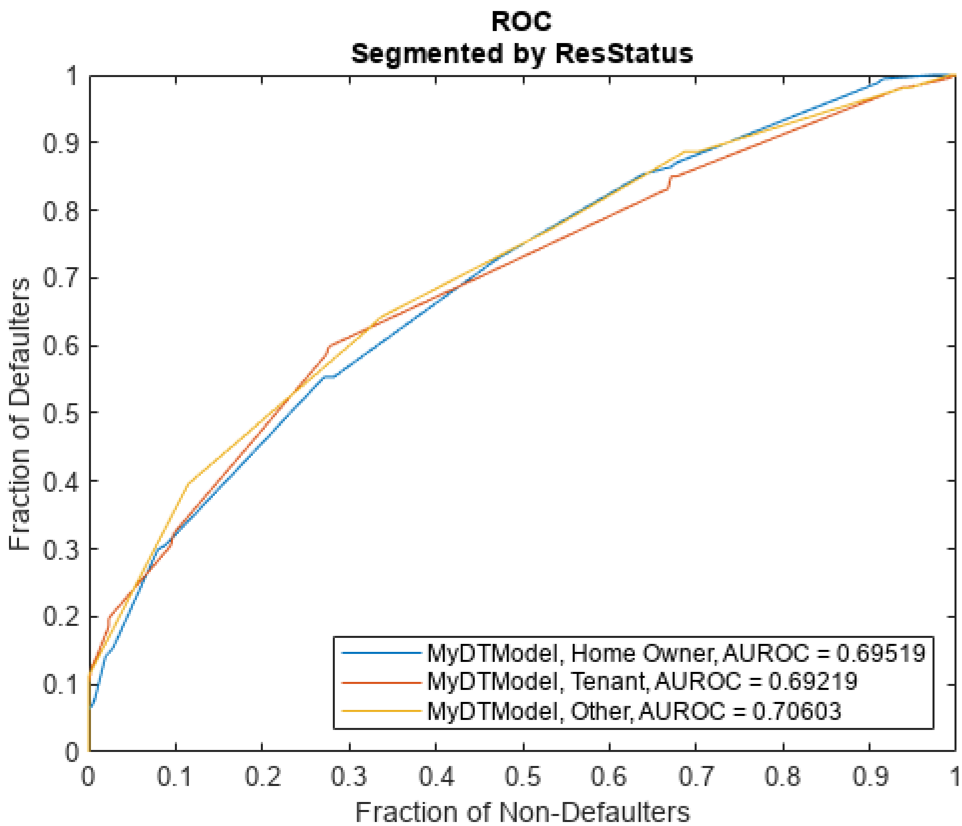
```
CondPDOriginal = myDTPredictFcn(dt,data);
isequal(CondPD,CondPDOriginal)
```

```
ans = logical
   1
```

By wrapping the decision tree as a lifetime PD model, all the validation capabilities of lifetime PD models are available.

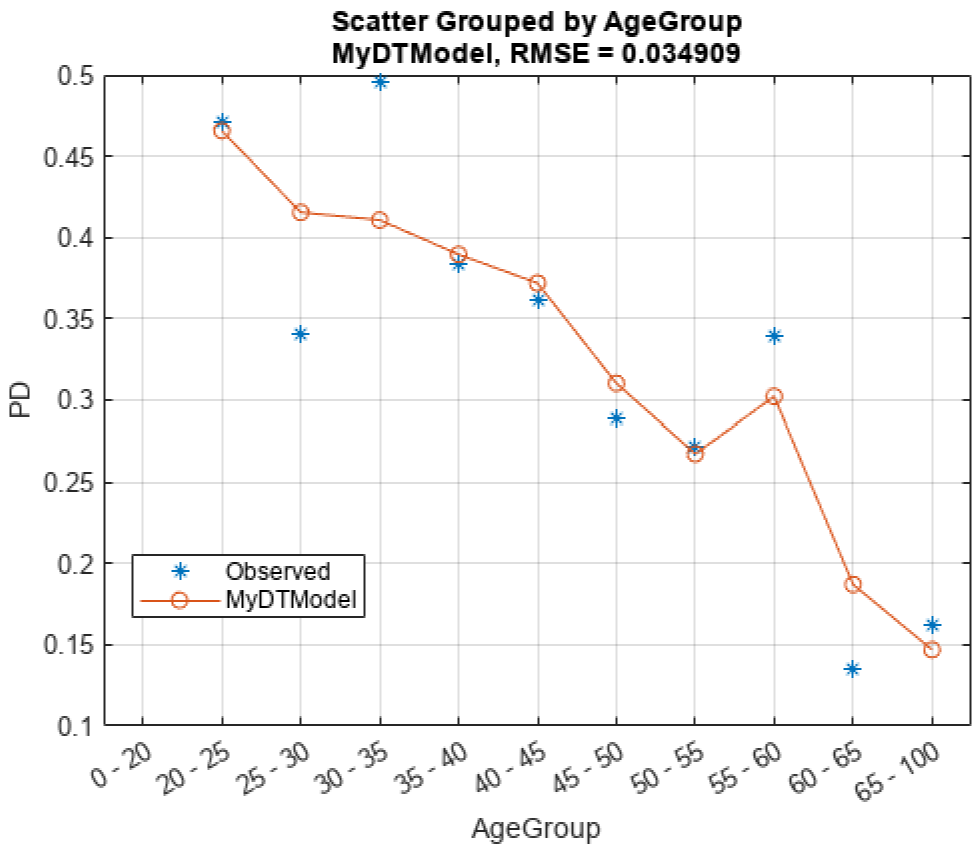
For example, use `modelDiscriminationPlot` to plot the ROC curve. The next plot shows the ROC curve for each of the residential status levels. All of the residential status levels show good discrimination in the training data.

```
modelDiscriminationPlot(pdModel,data,'SegmentBy','ResStatus')
```



Also, you can use `modelCalibrationPlot` to visualize the calibration of the model. A grouping variable is required to compare the average PD for the group against the default rate in the group. For illustration purposes, define an `AgeGroup` variable. You can also use other variables, including model predictors, as grouping variables.

```
AgeGroupEdges = [0,20:5:65,100];
AgeGroupLabels = strcat(string(AgeGroupEdges(1:end-1))," - ",string(AgeGroupEdges(2:end)));
data.AgeGroup = discretize(data.CustAge,AgeGroupEdges,'categorical',AgeGroupLabels);
modelCalibrationPlot(pdModel,data,'AgeGroup')
```



The default rates are high in this data set. The safest (60—65 and 65—100) age groups default at a rate higher than 10% and the 0—20 age group has a default rate of almost 50%. However, the predicted PDs are close to the observed default rates for most age groups. In other words, the model has good calibration in the training data.

Predict Lifetime PD

Lifetime PD is the cumulative probability of default over multiple periods. Therefore, the input for the `predictLifetime` function should contain multiple rows per ID. In this example, the data you use for training and validation contains only one observation per ID. If you pass it to `predictLifetime`, the output would be the same as the output of the `predict` function. For more information, see "Lifetime PD" on page 6-347 and "Data Input for Lifetime Prediction" on page 6-348.

The `predictLifetime` function is typically used for predictions on outstanding loans, where the predictor variable values must be projected, period-by-period, for several periods into the future. To project predictor values to prepare data for lifetime prediction, suppose you have an existing customer with ID 1234, 35 years old, with 36 months in the current address, owns her house, is employed with an income of $75,000, a bank customer for 50 months, average monthly balance in the account is $895 dollars with a utilization rate of 27%, and does not have another credit card with the bank.

```
% Use first row as a template,
% removing response and age group.
dataLifetime = data(1,1:end-2);
dataLifetime.CustID = 1234;
dataLifetime.CustAge = 35;
```

```
dataLifetime.TmAtAddress = 36;
dataLifetime.ResStatus = 'Home Owner';
dataLifetime.EmpStatus = 'Employed';
dataLifetime.CustIncome = 75000;
dataLifetime.TmWBank = 50;
dataLifetime.OtherCC = 'No';
dataLifetime.AMBalance = 895;
dataLifetime.UtilRate = 0.27;
disp(dataLifetime)
```

```
    CustID    CustAge    TmAtAddress    ResStatus     EmpStatus    CustIncome    TmWBank    Other
    ______    _______    ___________    __________    _________    __________    _______    _____

     1234       35           36         Home Owner    Employed       75000         50          No
```

To make projections for three periods ahead, you need projections for each variable. It is important to know what time interval is implicit in the underlying model because each PD model has a time interval. For more information, see “Time Interval for Logistic Models” on page 6-623 and “Time Interval and Data Input for Lifetime Prediction” on page 6-348.

In this example, use the time interval of 1 year, in other words, assume that the decision tree model predicts 1-year PDs. In this case, age is easy to project because the customer is one year older on each subsequent time period, that is, on each subsequent row in the data. Other time variables, such as time at address and time with bank, are easily projected if you assume there will be no change in address and that the customer will continue with the bank. You can project other variables as well with assumptions for each of them. For example, for `CustIncome`, you can keep it constant, as in this example, where your assumption might be that the customer does not update their income information. Or, you could assume some income growth instead. In this example, for simplicity, all variables other than time variables, are kept constant.

```
dataLifetime = repmat(dataLifetime,3,1);
% No changes to the ID value, same customer
dataLifetime.CustAge(2:3) = dataLifetime.CustAge(1)+[1;2]; % one year older each year
dataLifetime.TmAtAddress(2:3) = dataLifetime.TmAtAddress(1)+[12; 24]; % 12 extra months each year
% No changes to ResStatus, EmpStatus or CustIncome
dataLifetime.TmWBank(2:3) = dataLifetime.TmWBank(1)+[12; 24]; % 12 extra months each year
% No changes to OtherCC, AMBalance or UtilRate
disp(dataLifetime)
```

```
    CustID    CustAge    TmAtAddress    ResStatus     EmpStatus    CustIncome    TmWBank    Other
    ______    _______    ___________    __________    _________    __________    _______    _____

     1234       35           36         Home Owner    Employed       75000         50          No
     1234       36           48         Home Owner    Employed       75000         62          No
     1234       37           60         Home Owner    Employed       75000         74          No
```

Use `predictLifetime` to make a lifetime prediction.

```
pdLifetime = predictLifetime(pdModel,dataLifetime)
```

```
pdLifetime = 3×1

    0.2527
    0.4415
    0.5826
```

References

[1] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

Local Functions

```
function CondPD = myDTPredictFcn(DTmodel,data)
%myDTPredictFcn Predict conditional PD with Decision Tree model.

[~,ObservationClassProb] = predict(DTmodel,data);
CondPD = ObservationClassProb(:,2);

end
```

See Also

`customLifetimePDModel` | `fitLifetimePDModel` | `fitEADModel` | `fitLGDModel` | `portfolioECL`

Related Examples

- “Create Custom Lifetime PD Model for Credit Scorecard Model with Function Handle” on page 3-131
- “Expected Credit Loss Computation” on page 4-124
- “Modeling Probabilities of Default with Cox Proportional Hazards” on page 4-28

Measure Transition Risk for Loan Portfolios with Respect to Climate Scenarios

This example shows the effect of transition risk on portfolios of loans from two banks given three different climate scenarios. Potentially, climate change is a large structural change affecting the economy and the financial system. Clear physical risks are associated with climate change, including increases in the global average temperature and an increased frequency and severity of extreme weather events. These events could result in significant macroeconomic and financial system impacts. In addition to physical risk, another type of risk called *transition risk* arises from changes in policy and new technologies, such as the growth of renewable energy.

Several countries around the globe are working on projects to understand and model different climate policies. For example, these climate initiatives are examples of projects investigating economic risk:

- LIMITS (**L**ow climate **IM**pact scenarios and the **I**mplications of required **T**ight emission control **S**trategies) was a research effort that included twelve partners from Europe, China, India, Japan, and the United States. The main objectives of the project were: (a) to provide an assessment of the emissions reductions strategies at a world level for the major global economies, and (b) to disseminate scientific knowledge for climate and energy policy. [4 on page 4-246]
- The Bank of Canada launched a pilot project with the Office of the Superintendent of Financial Institutions.The goal of the pilot project was to understand risks to the economy and the financial system related to climate change. This work included developing a set of climate transition scenarios relevant to Canada that explore pathways consistent with achieving certain climate targets. [1 on page 4-246]
- The Massachusetts Institute of Technology (MIT) developed the Economic Projection and Policy Analysis (EPPA) model that is part of the MIT Integrated Global Systems Model (IGSM) that represents the human systems. EPPA is a recursive-dynamic multi-regional general equilibrium model of the world economy, which is built on the Global Trade Analysis Project (GTAP) data set and additional data for the greenhouse gas and urban gas emissions. IGSM is designed to develop projections of economic growth and anthropogenic emissions of greenhouse related gases and aerosols. [3 on page 4-246]
- The Climate Integrated Assessment Models Explorer repository contains a set of tools to explore different data sets hosted by the IIASA Energy program (ENE). This repositiory contains an example that computes changes to multiple bank loan portfolios as a result of climate shocks. [2 on page 4-246]

This example follows an approach of Monsaterolo [5 on page 4-247] to develop a novel climate stress-test methodology for portfolios of loans to energy infrastructure projects and follows the workflow:

1. Obtain the climate scenario data on page 4-231.
2. Compute the market share shocks on page 4-232.
3. Obtain the loan portfolio data on page 4-235.
4. Create a valuation framework for loan contracts subject to climate policy shocks on page 4-237.
5. Compute the distribution of changes in the loan portfolio values on page 4-240.

Obtain Climate Scenario Data

This example uses climate scenarios developed by MIT in collaboration with the Bank of Canada. The scenarios are described as:

- *Baseline* (2019 policies) — baseline scenario consistent with climate policies in place at the end of 2019
- *Below 2°C Immediate* — immediate policy action scenario to limit average global warming to below 2°C
- *Below 2°C Delayed* — delayed policy action scenario to limit average global warming to below 2°C
- *Net-Zero 2050* (1.5°C) — more ambitious immediate policy action scenario to limit average global warming to 1.5°C that includes current net-zero commitments by some countries

The Climate Transition Scenario data is provided by the Bank of Canada and is available free of charge at www.bankofcanada.ca. [1 on page 4-246]

The data is converted to a MAT file and is loaded from `BankOfCanadaClimateScenarioData.mat`.

Compute Market Share Shocks

Load the data file for the Climate Transition Scenario provided by the Bank of Canada. This data set contains energy information of different sectors for various geographies around the world. The `CL_VARIABLE` column in the data set contains both input information used to model the different climate scenarios, as well as, output information of energy usage. This example focuses on the different forms of primary energy, that is: bioenergy, coal, gas, hydro, nuclear, oil, and renewables (wind and solar). The example computes the market shares for each of the different energy forms and uses this information to calculate climate shocks. The different geographies in the data set are Africa, Canada, China, Europe, India, Japan, the United States, and the rest of the world.

```
load BankOfCanadaClimateScenarioData.mat
head(ClimateTransitionScenarioData)
```

```
    k    CL_GEOGRAPHY    CL_SECTOR              CL_VARIABLE                       CL_UNIT
    _    ____________    _________    _______________________________    ___________________    _

    1       Canada       National     Carbon price                       US$2014/tCO2e          
    2       Canada       National     Carbon price                       US$2014/tCO2e          
    3       Canada       National     Emissions | total GHG (scope 1)    Million tonnes CO2e    
    4       Canada       National     Emissions | total GHG (scope 1)    Million tonnes CO2e    
    5       Canada       National     Input price | Coal                 Index (2014 = 1)       
    6       Canada       National     Input price | Coal                 Index (2014 = 1)       
    7       Canada       National     Input price | Crops                Index (2014 = 1)       
    8       Canada       National     Input price | Crops                Index (2014 = 1)       
```

Trim and preprocess the original data set for what is required for this example.

```
% This example uses only the Primary Energy variables to compute market
% shares for different geographies.
VariableSubset = {'Primary Energy | Bioenergy', 'Primary Energy | Coal', 'Primary Energy | Gas', 
'Primary Energy | Hydro', 'Primary Energy | Nuclear', 'Primary Energy | Oil', ...
'Primary Energy | Renewables (wind&solar)', 'Primary Energy | Total'};
ClimateTransitionScenarioData = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioD

% Remove columns 'k','CL_SECTOR' and 'CL_UNIT' and then sort the rows.
ClimateTransitionScenarioData = removevars(ClimateTransitionScenarioData,{'k','CL_SECTOR','CL_UNI
ClimateTransitionScenarioData = sortrows(ClimateTransitionScenarioData);

% Pull market share data out according to climate scenario.
baseline = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO, 'Bas
b2delayed = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO, 'Be
b2immediate = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO,
```

```
netzero2050 = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO,

% Compile all the scenarios into one data set.
MarketShareData = baseline;
MarketShareData = removevars(MarketShareData, "CL_SCENARIO");
MarketShareData.Properties.VariableNames(4) = "BASELINE";
MarketShareData.BELOW_2C_IMMEDIATE = b2immediate.CL_VALUE;
MarketShareData.BELOW_2C_DELAYED = b2delayed.CL_VALUE;
MarketShareData.NETZERO_2050 = netzero2050.CL_VALUE;
head(MarketShareData)
```

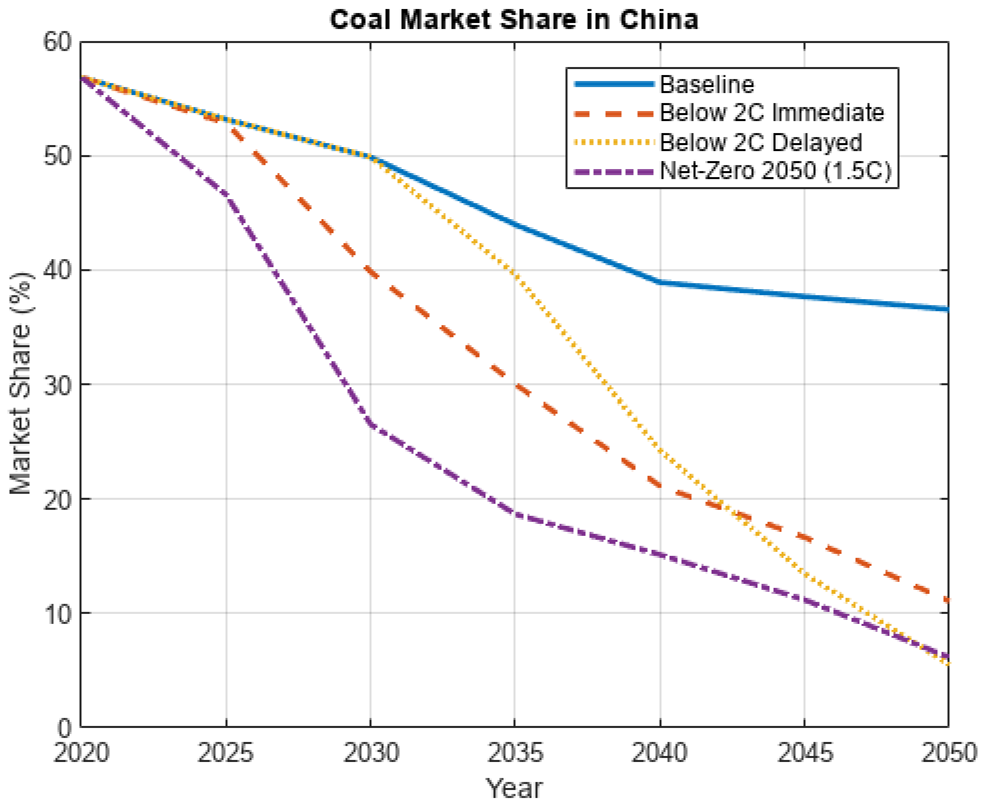
CL_GEOGRAPHY	CL_VARIABLE	CL_YEAR	BASELINE	BELOW_2C_IMMEDIATE	BI
Africa	Primary Energy | Bioenergy	2020	15.502	15.502	
Africa	Primary Energy | Bioenergy	2025	15.302	15.302	
Africa	Primary Energy | Bioenergy	2030	15.221	15.203	
Africa	Primary Energy | Bioenergy	2035	15.072	15.042	
Africa	Primary Energy | Bioenergy	2040	15.016	15.055	
Africa	Primary Energy | Bioenergy	2045	14.249	14.273	
Africa	Primary Energy | Bioenergy	2050	13.591	14	
Africa	Primary Energy | Coal	2020	4.5909	4.5909	

Select the geography and subsector, and then compute the market share. By default, in this example, a coal project in China is selected.

```
Geography = China ;
Sector = Coal ;
SectorFullName = "Primary Energy | " + Sector;
BaseSector = "Primary Energy | Total";

Years = (2020:5:2050)';
GeographyData = MarketShareData(MarketShareData.CL_GEOGRAPHY == Geography, :);
MarketShare = GeographyData{GeographyData.CL_VARIABLE == SectorFullName, 4:7} ./ GeographyData{Ge
```

Plot the market shares for different climate scenarios. For the coal sector in China, the baseline scenario shows a drop in market share. However, this drop is accelerates in each of the three climate scenarios. The *Below 2C Delayed* scenario coincides with the baseline up until 2030, after which there is a sudden fall in market share, whereas the other two scenarios (*Below 2C Immediate* and *Net-Zero 2050*) lose market share more gradually until 2050.

```
figure;
msPlot = plot(Years, MarketShare, 'LineWidth',2);
grid on
set(msPlot, {'LineStyle'}, {'-';'--';':';'-.'})
legend("Baseline", "Below 2C Immediate", "Below 2C Delayed", "Net-Zero 2050 (1.5C)", 'Location',
xlim([2020 2050])
ylabel('Market Share (%)')
xlabel('Year')
title(Sector + " Market Share in " + Geography)
```

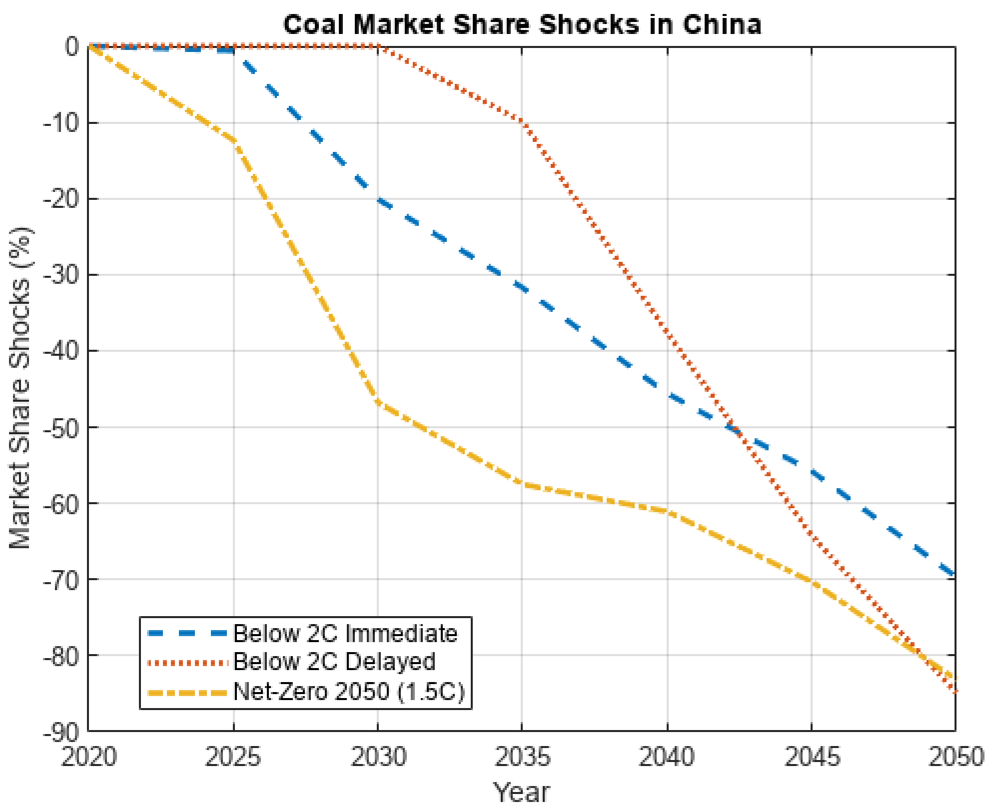


Compute shocks for all climate scenarios.

```
Shocks = (MarketShare(:,2:4) - MarketShare(:,1))./MarketShare(:,1)*100;
```

Plot the shocks for different scenarios. For the default coal sector in China, the *Below 2C Immediate* and the *Net-Zero 2050* scenarios have a gradual negative market share shock over time. However, the *Below 2C Delayed* scenario shows no market share shock until 2030, and then there is a sudden drop until 2050.

```
figure;
shp = plot(Years, Shocks, 'LineWidth',2);
grid on
set(shp, {'LineStyle'}, {'--';':';'-.'})
xlabel('Year')
ylabel('Market Share Shocks (%)')
xlim([2020 2050])
legend("Below 2C Immediate", "Below 2C Delayed", "Net-Zero 2050 (1.5C)", 'Location','best');
title(Sector + " Market Share Shocks in " + Geography);
```



Obtain Loan Portfolio Data

This example uses a second data set (`ClimateLoanPortfolioData.mat`) for a loan portfolio. This data set contains simulated loan data of two fictitious banks: Bank 1 and Bank 2. The portfolio of loans are from different geographies and sectors. The example uses the face value of the loans to compute the change in reserves that a bank has to allocate when a climate shock occurs.

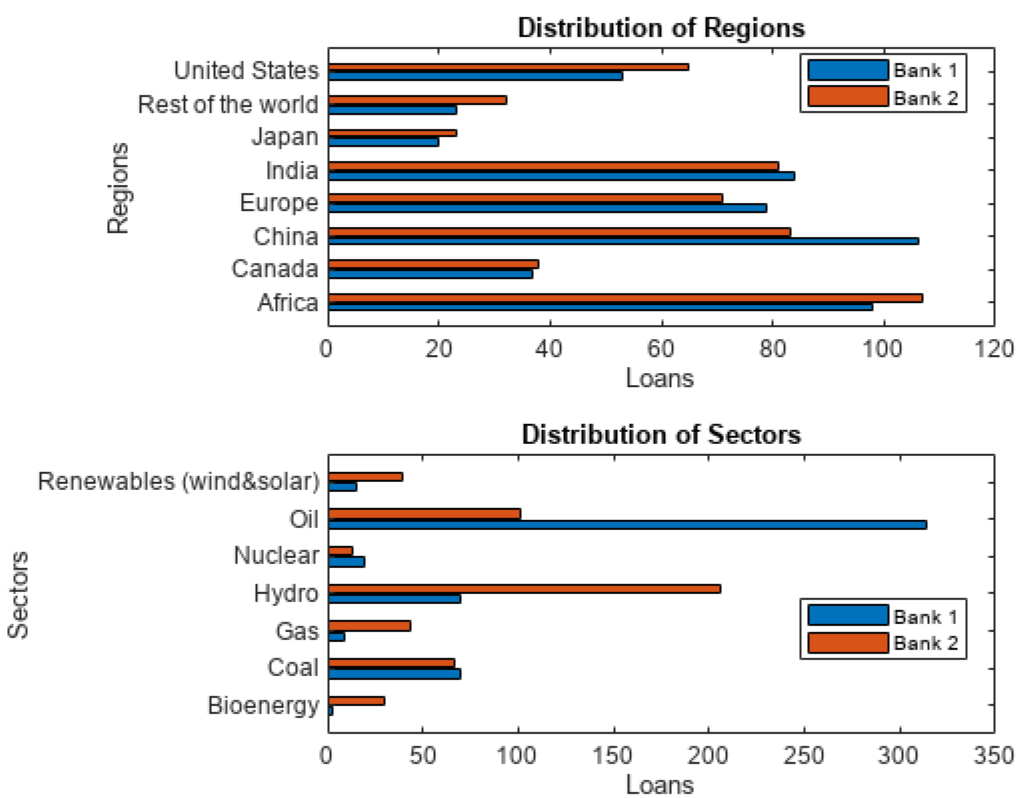
The distribution of loans for the two banks, by energy sector, is:

Sector	Bank 1	Bank 2
Bioenergy	2%	5%
Coal	15%	15%
Gas	2%	10%
Hydro	15%	40%
Nuclear	2%	3%
Oil	60%	20%
Renewables (wind&solar)	4%	7%
Total	100%	100%

Generate histograms to show the distribution of loans, by region and sector, for the two simulated banks.

```
load ClimateLoanPortfolioData
LoanPortfolioDataBank1 = ClimateLoanPortfolioData(ClimateLoanPortfolioData.Bank=='Bank1',:);
```

```
LoanPortfolioDataBank2 = ClimateLoanPortfolioData(ClimateLoanPortfolioData.Bank=='Bank2',:);

[CountsRegion1, BinsRegion1] = histcounts(LoanPortfolioDataBank1.BorrowerRegion);
[CountsSector1, BinsSector1] = histcounts(LoanPortfolioDataBank1.BorrowerSector);
[CountsRegion2, BinsRegion2] = histcounts(LoanPortfolioDataBank2.BorrowerRegion);
[CountsSector2, BinsSector2] = histcounts(LoanPortfolioDataBank2.BorrowerSector);

figure
tiledlayout(2,1)
nexttile
barh(categorical(BinsRegion1), [CountsRegion1; CountsRegion2]);
xlabel('Loans')
ylabel('Regions')
title('Distribution of Regions')
legend({'Bank 1', 'Bank 2'},'Location','best')
nexttile
barh(categorical(BinsSector1), [CountsSector1; CountsSector2]);
xlabel('Loans')
ylabel('Sectors')
title('Distribution of Sectors')
legend({'Bank 1', 'Bank 2'},'Location','best')
```



Bank 1 is heavily invested in fossil fuel projects (primarily oil) and Bank 2 is invested in green energy projects (primarily hydro). Regarding the distribution of projects across geographies, both banks are similar, with a larger number of projects in developing regions like China, India, and Africa.

Create Valuation Framework for Loan Contracts Subject to Climate Policy Shocks

Using the same notation as in Monsaterolo [5 on page 4-247], consider a bank i endowed with a portfolio of investments in a set of projects through loan contracts. Each loan is represented by a distinct value j. The goal is to carry out a valuation of this loan portfolio that accounts for climate policy shocks. The methodology assumes an underlying structural model that is similar to the Merton model (see `mertonmodel`). The valuation model includes three time steps: t_0, t^*, and T_j, with $t_0 < t^* < T_j$. Time step t_0 denotes the time at which the valuation is carried out, t^* denotes the time at which the climate policy shock potentially occurs, and T_j denotes the maturity of the loan j.

The valuation of bank i's loan portfolio is written as

$$\mathrm{A}_i(t_0) = \sum_j A_{i,j}\left(t_0, T_j\right).$$

Consider an approach based on the expected value of the loan

$$A_{i,j}\left(t_0, T_j\right) = p_j\left(t_0, T_j\right) r_j F_{i,j} + \left(1 - p_j\left(t_0, T_j\right)\right) F_{i,j} = F_{i,j} - \left(F_{i,j}\left(1 - r_j\right) p_j\left(t_0, T_j\right)\right),$$

where $F_{i,j}$ is the face value of the loan, r_j is the recovery rate on the loan contract, and $p_j\left(t_0, T_j\right)$ is the probability, based on the information available at time t_0 that the borrower, j, defaults on the loan at maturity T_j.

Therefore, the expected value of the loan is the face value of the loan $F_{i,j}$ minus the reserves or provisions that need to be set aside by the bank for that loan $F_{i,j}\left(1 - r_j\right) p_j\left(t_0, T_j\right)$.

At time t^* the occurrence of a climate policy shock implies that the economy switches from a business-as-usual scenario characterized by no climate policy (B) to scenario P, where the market shares of some economic sectors are affected. This change in default probability implies a proportional change in the expected value of the loan

$$\Delta A_{i,j}\left(t_0, T_j, P\right) = -F_{i,j}\left(1 - r_j\right) \Delta p_j(P),$$

where $\Delta p_j(P)$ denotes the difference of the default probability going from scenario B to P.

This change is the negative of the change in provisions for the loan. That is, if the value goes down, it is due to an increase in provisions driven by the change in probability.

Assume that the policy shock impacts the borrower's balance sheet, and thus the expected value of the loan. We define a market share shock, $u_{S,R}(P, M, t^*)$, as

$$u_{S,R}(P, M, t^*) = \frac{\left(m_{S,R}(P, M, t^*) - m_{S,R}(B, M, t^*)\right)}{m_{S,R}(B, M, t^*)}.$$

Assume that a relative change in the market share of borrower j's sector S within the geographic region R, denoted by $u_{S,R}(P, M, t^*)$, implies a proportional relative change in j's profitability. Also, because the net worth is the integral of profits over one period of time, the relative change in net worth and profit coincide. Therefore, it is equivalent to assume that a relative change in net worth is proportional to the relative shock in market share

$$\frac{\Delta E_j}{E_j} = \chi u_{S,R}(P, M, t^*),$$

where χ denotes the elasticity of profitability with respect to market share. Monsaterolo [5 on page 4-247] assumes a value of χconstant and equal to 1 (typical empirical values range from 0.2 and 0.6).

Another assumption by Monsaterolo [5 on page 4-247] is that the probability distribution $p(\eta_j)$ of the shocks on the borrower's asset side follows a uniform distribution with support $\delta$and mean μ, for a given model M, region, and sector. Therefore, the change in default probability is expressed as

$$\Delta P = \frac{\theta_j(P) - \theta_j(B)}{\delta} = -\frac{E_j}{\delta}\chi u_{S,R}(P, M, t^*).$$

The change in expected value of the loan, conditional to a change from scenario B to scenario P becomes:

$$\Delta A_{i,j} = F_{i,j}(1 - r_j)\frac{E_j}{\delta}\chi u_{S,R}(P, M, t^*).$$

Summing the projects j in the portfolio, you obtain the total change in loan value:

$$\sum_j \Delta A_{i,j}(t_0, T_j, P) = \sum_j F_{i,j}(1 - r_j)\frac{E_j}{\delta}\chi u_{S,R}(P, M, t^*).$$

This example computes the change in value for one loan. By default, the 12th row of the data set is selected, which is an oil project in the United States for Bank 1. you can choose another loan by using the `Loan` slider. You can also select a different `ClimateScenario` from the three available scenarios. By default, the *Below* 2°*C Delayed* scenario is selected.

```
Loan = 12;
ClimateScenario = Below 2°C delayed;
Geography = string(ClimateLoanPortfolioData{Loan,'BorrowerRegion'});
Sector = string(ClimateLoanPortfolioData{Loan,'BorrowerSector'});
LoanID = ClimateLoanPortfolioData{Loan,'LoanID'};
FaceValueOfLoan = ClimateLoanPortfolioData{Loan,"FaceValue"};
SectorFullName = "Primary Energy | " + Sector;
```

Compute the value of the selected `Loan`.

```
disp(ClimateLoanPortfolioData(Loan,:));
```

LoanID	BorrowerCreditRating	LoanType	Bank	InterestRate	InterestType
"786801JSP"	A2	Term	Bank1	0.028	Fixed

You can modify the recovery rate r_j, as well as, χ. By default, $r_j = 0.4$ and $\chi = 0.3$. A normal range of χ is from 0.2 to 0.6.

Following the Monsaterolo [5 on page 4-247] discussion in appendix I, set $\frac{E_j}{\delta} = 1$ to correspond to the assumption that the magnitude of the initial net worth and width of the distribution of the idiosyncratic shocks are comparable. You can adjust the `EjDeltaRatio` value using the slider.

```
RecoveryRate = 0.4;
Chi = 0.3;
```

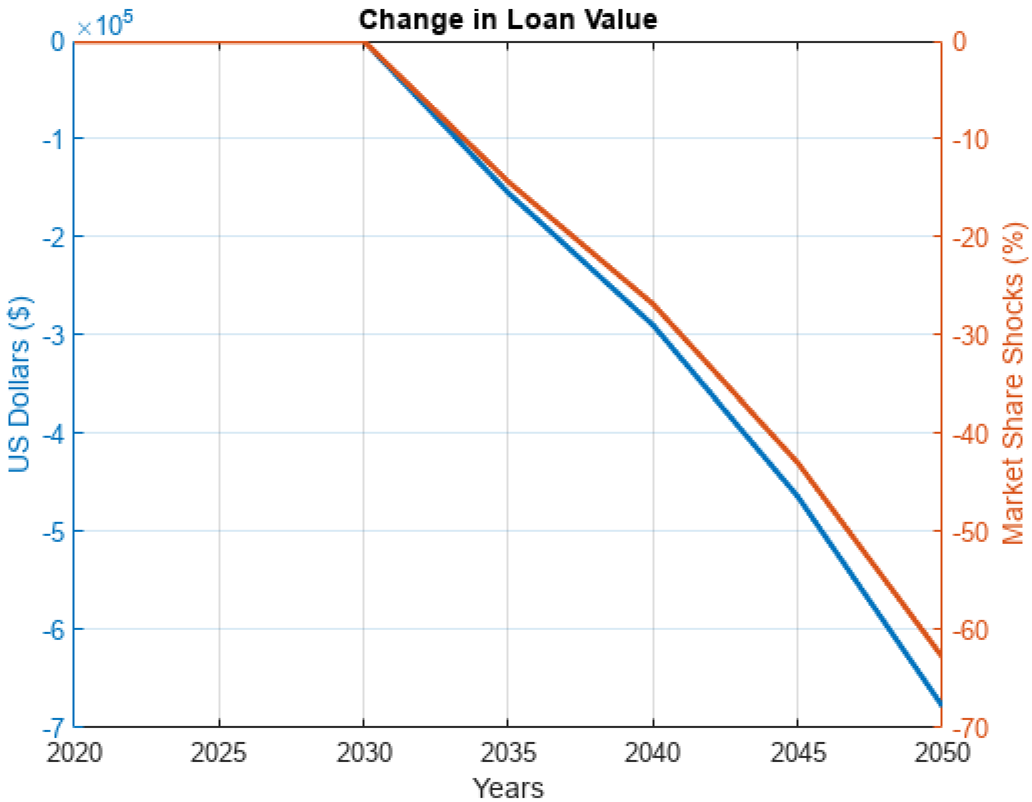
```
EjDeltaRatio =  1  ;
TargetYear = 2035 ;

GeographyData = MarketShareData(MarketShareData.CL_GEOGRAPHY == Geography, :);
MarketShare = GeographyData{GeographyData.CL_VARIABLE == SectorFullName, 4:7} ./ GeographyData{Ge
Shocks = (MarketShare(:,2:4) - MarketShare(:,1))./MarketShare(:,1)*100;

ChangeInDefaultProbability = -EjDeltaRatio.*Chi.*(Shocks(:,ClimateScenario)/100);
ChangeInValue = -FaceValueOfLoan.*(1-RecoveryRate).*ChangeInDefaultProbability;
```

The change in value of a loan is directly translated into a change in the reserves that need to be allocated by the bank for that particular loan.

```
disp("The change in value of the loan in the selected climate scenario and target year = $" + nur
```

```
The change in value of the loan in the selected climate scenario and target year = $-155161.6488
```

```
% Plot change in value of loan over time superimposed over corresponding
% shock.
f = figure;
ax = axes(f);
yyaxis(ax, 'left')
plot(Years, ChangeInValue, 'LineWidth', 2)
xlabel('Years');
ylabel('US Dollars ($)')
yyaxis(ax, 'right')
plot(Years, Shocks(:, ClimateScenario), 'LineWidth', 2);
ylabel('Market Share Shocks (%)');
title('Change in Loan Value');
grid on
```



Compute Change in Value of Entire Portfolio of Loans

For each of the two banks, Bank 1 and Bank 2, compute the total change in loan value of the entire portfolio of loans for each climate scenario. You can do this by summing up the changes in values of each individual loan.

```
% For each of the region and sector pairs, compute the market shocks for
% all scenarios and store these values.
MarketShocks = struct();

Bank1Combos = unique(table(LoanPortfolioDataBank1.BorrowerRegion, LoanPortfolioDataBank1.Borrower
Bank2Combos = unique(table(LoanPortfolioDataBank2.BorrowerRegion, LoanPortfolioDataBank2.Borrower
TotalCombos = union(Bank1Combos, Bank2Combos);

for i = 1:height(TotalCombos)
    Region = string(TotalCombos.Region(i));
    Sector = string(TotalCombos.Sector(i));
    SectorFullName = "Primary Energy | " + Sector;
    GeographyData = MarketShareData(MarketShareData.CL_GEOGRAPHY == Region, :);
    MarketShare = GeographyData{GeographyData.CL_VARIABLE == SectorFullName, 4:7} ./ GeographyDat
    Shocks = (MarketShare(:,2:4) - MarketShare(:,1))./MarketShare(:,1)*100;
    if Sector == "Renewables (wind&solar)"
        SectorSplit = strsplit(Sector);
        Sector = SectorSplit(1);
    end
    MarketShocks.(strrep(Region,' ','')).(strrep(Sector,' ','')) = Shocks;
end
```

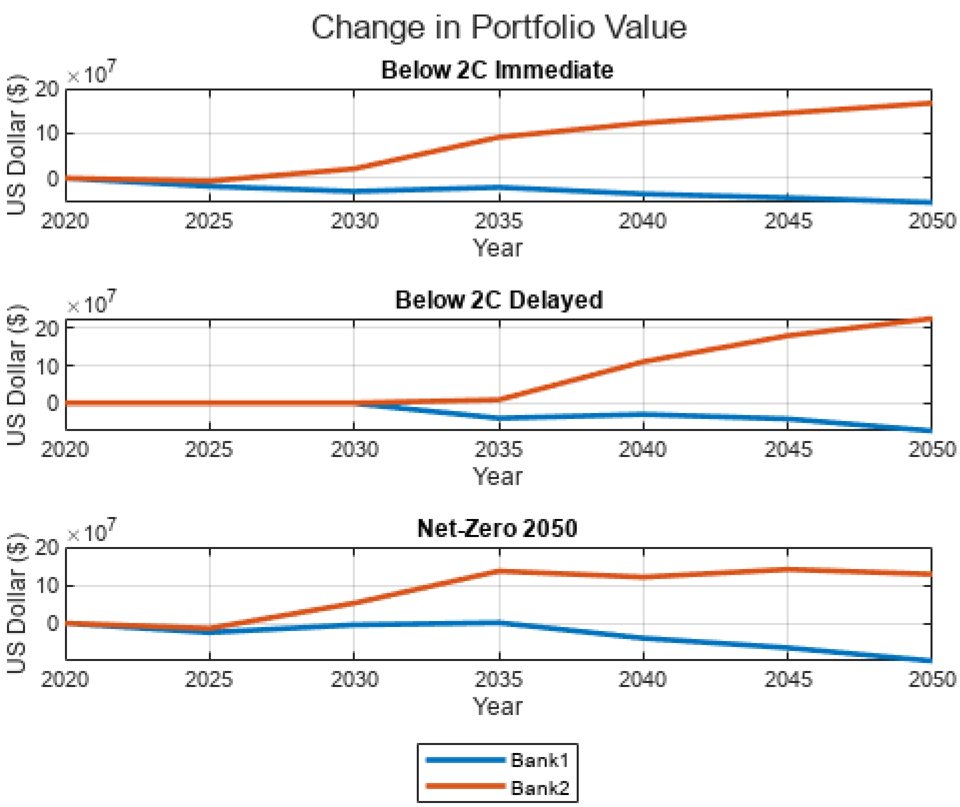
Create a table containing the values of each loan, for each bank, for each climate scenario, and for each target year. Use the same model parameter values for simplicity. However, you can change these parameters for different issuers.

```
RecoveryRate = 0.46 ;
Chi = 0.3 ;
EjDeltaRatio = 1 ;

LoanValues = ClimateLoanPortfolioData;
LoanValues = removevars(LoanValues,{'LoanType','BorrowerCreditRating','InterestRate','InterestTy
LoanValues = repelem(LoanValues, 7, 1);
LoanValues.Year = repmat([2020; 2025; 2030; 2035; 2040; 2045; 2050], 1000, 1);
LoanValues.Below2CImmediate = zeros(7000, 1);
LoanValues.Below2CDelayed = zeros(7000, 1);
LoanValues.NetZero2050 = zeros(7000, 1);

for i = 1:7:height(LoanValues)
    Region = string(LoanValues.BorrowerRegion(i));
    Sector = string(LoanValues.BorrowerSector(i));
    if Sector == "Renewables (wind&solar)"
        SectorSplit = strsplit(Sector);
        Sector = SectorSplit(1);
    end
    Value = LoanValues.FaceValue(i);
    ChangeInValue = Value.*(1-RecoveryRate).*EjDeltaRatio.*Chi.*(MarketShocks.(strrep(Region,' '
    LoanValues{i:i+6, {'Below2CImmediate','Below2CDelayed','NetZero2050'}} = ChangeInValue;
end
```

Compare the change in portfolio values of the two banks, for each climate scenario, and for all target years.

```
TPVBank1 = zeros(length(Years),3);
TPVBank2 = zeros(length(Years),3);
for i = 1:length(Years)
    TBank1 = LoanValues((LoanValues.Bank == "Bank1") & (LoanValues.Year == Years(i)), :);
    TBank2 = LoanValues((LoanValues.Bank == "Bank2") & (LoanValues.Year == Years(i)), :);
    TPVBank1(i,:) = sum(TBank1{:,7:9});
    TPVBank2(i,:) = sum(TBank2{:,7:9});
end

figure;
t = tiledlayout(3,1);
nexttile
plot(Years, [TPVBank1(:,1),TPVBank2(:,1)], 'LineWidth', 2)
xlabel('Year');
ylabel('US Dollar ($)')
title('Below 2C Immediate')
grid on
nexttile
plot(Years, [TPVBank1(:,2),TPVBank2(:,2)], 'LineWidth', 2)
xlabel('Year');
ylabel('US Dollar ($)')
title('Below 2C Delayed')
grid on
nexttile
```

```
plot(Years, [TPVBank1(:,3),TPVBank2(:,3)], 'LineWidth', 2)
xlabel('Year');
ylabel('US Dollar ($)')
title('Net-Zero 2050')
leg = legend({'Bank1','Bank2'});
leg.Layout.Tile = 'south';
grid on
title(t,'Change in Portfolio Value');
```



Considering that Bank 1 is weighted towards fossil fuels and Bank 2 is weighted toward green energy, the portfolio value of Bank 1 decreases over time, while the portfolio value of Bank 2 increases over time for each climate scenario.

From a reserves standpoint, the provisions of Bank 2 steadily increase over time and those of Bank 1 decrease over time.

Compute Distribution of Changes in Loan Portfolio Values

The Compute Change in Value of Entire Portfolio of Loans on page 4-240 section illustrated the aggregate changes to the portfolio value. This section focuses on the entire distribution of value changes for particular climate scenarios.

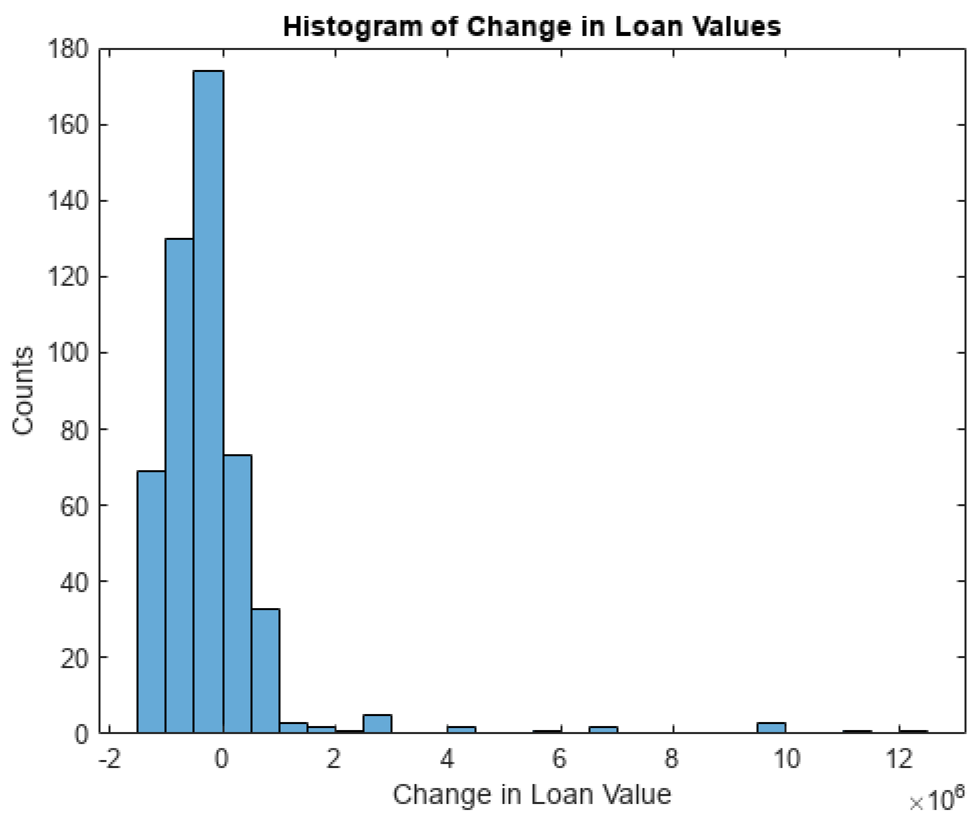
Select the bank, climate scenario, and target year to compute the quartiles of the change in portfolio values.

```
Bank = Bank1;
ClimateScenario = Below 2°C delayed;
TargetYear = 2050;
```

```
NewTable = LoanValues((LoanValues.Bank == Bank) & (LoanValues.Year == str2double(TargetYear)), :
```

Plot the histogram of the change in loan values for the selected bank and climate scenario. Based on the default selection of Bank 1 under the *Below 2C Delayed* climate scenario for the target year 2050, you see that the distribution of the change in loan values has a long right tail and most of the frequency is below zero. This change occurs because Bank 1 has more projects focusing on fossil fuels, which lose market share, and thus value over time under the *Below 2C Delayed* climate scenario.

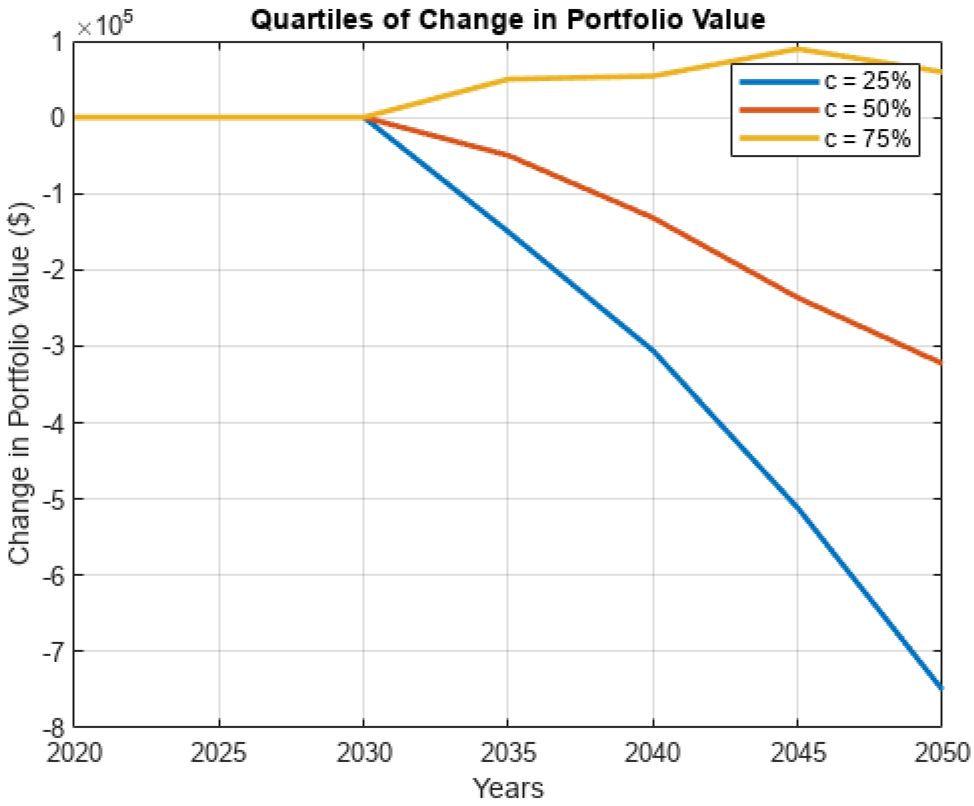
```
f = figure;
h = histogram(NewTable{:,6+ClimateScenario});
h.Parent.XLabel.String = 'Change in Loan Value';
h.Parent.YLabel.String = 'Counts';
h.Parent.Title.String = 'Histogram of Change in Loan Values';
```



Compute and plot the quartiles of the changes in portfolio values over time for the selected bank.

```
PLV = zeros(length(Years),3);
for i = 1:length(Years)
    t = LoanValues((LoanValues.Bank == Bank) & (LoanValues.Year == Years(i)), :);
    PLV(i,1) = prctile(t{:,6+ClimateScenario}, 25);
```

```
    PLV(i,2) = prctile(t{:,6+ClimateScenario}, 50);
    PLV(i,3) = prctile(t{:,6+ClimateScenario}, 75);
end

figure;
plot(Years, PLV, 'LineWidth', 2)
xlabel('Years');
ylabel('Change in Portfolio Value ($)')
title('Quartiles of Change in Portfolio Value')
legend({'c = 25%','c = 50%','c = 75%'})
grid on
```



To compute some standard metrics of risk such as the Value-at-Risk (VaR) of the portfolio, you need to know the joint probability distribution of the idiosyncratic shocks and the probability of occurrence of climate policy shocks. In the absence of these estimations, Monsaterolo [5 on page 4-247] defines a project-level climate VaR as the value such that, conditional to the same climate policy shock for all *n loans*, the fraction of loans leading to losses higher than the VaR equals the confidence level *c*

$$|\{j|\Delta A_{i,j}(t_0, T_j, P, B) \geq \mathrm{VaR}\}| / n = c.$$

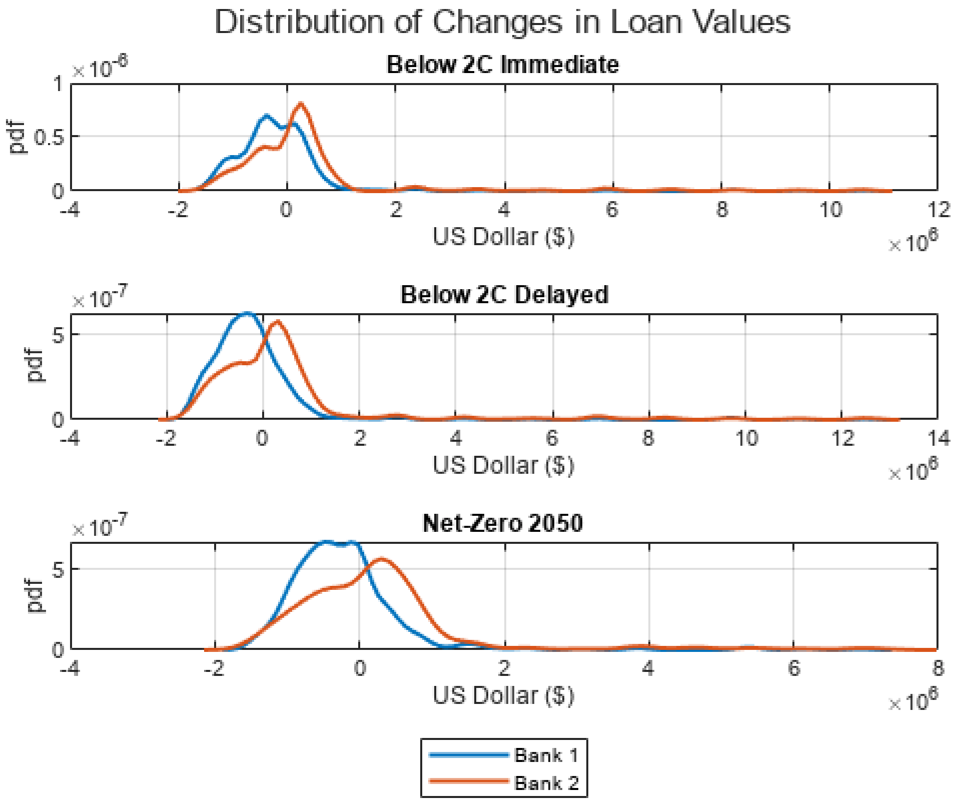
This project-level climate VaR metric is a percentile of the distribution of value changes for the portfolio.

```
ConfidenceLevel = [1%  ▾];
ProjVaR = -prctile(NewTable{:,6+ClimateScenario}, ConfidenceLevel);
disp("The project-level climate VaR at the " + ConfidenceLevel + "% confidence level = $" + num2s
```

```
The project-level climate VaR at the 1% confidence level = $1453621.915
```

Plot a graph of distributions of changes in loan values for each target year and for a given scenario and bank. Use the kernel smoothing function estimate for univariate data. The estimate is based on a normal kernel function and is evaluated at equally spaced points that cover the range of the data.

```
[F1Bank1, Xi1Bank1] = ksdensity(TBank1.Below2CImmediate);
[F1Bank2, Xi1Bank2] = ksdensity(TBank2.Below2CImmediate);

[F2Bank1, Xi2Bank1] = ksdensity(TBank1.Below2CDelayed);
[F2Bank2, Xi2Bank2] = ksdensity(TBank2.Below2CDelayed);

[F3Bank1, Xi3Bank1] = ksdensity(TBank1.NetZero2050);
[F3Bank2, Xi3Bank2] = ksdensity(TBank2.NetZero2050);

figure;
t = tiledlayout(3,1);
ax1 = nexttile;
plot(Xi1Bank1, F1Bank1, 'LineWidth', 1.5)
hold on
plot(Xi1Bank2, F1Bank2, 'LineWidth', 1.5)
ax1.Title.String = "Below 2C Immediate";
xlabel('US Dollar ($)')
ylabel('pdf')
grid on
ax2 = nexttile;
plot(Xi2Bank1, F2Bank1, 'LineWidth', 1.5)
hold on
plot(Xi2Bank2, F2Bank2, 'LineWidth', 1.5)
ax2.Title.String = "Below 2C Delayed";
xlabel('US Dollar ($)')
ylabel('pdf')
grid on
ax3 = nexttile;
plot(Xi3Bank1, F3Bank1, 'LineWidth', 1.5)
hold on
plot(Xi3Bank2, F3Bank2, 'LineWidth', 1.5)
leg = legend("Bank 1","Bank 2");
leg.Layout.Tile = 'south';
ax3.Title.String = "Net-Zero 2050";
xlabel('US Dollar ($)')
ylabel('pdf')
grid on
title(t,'Distribution of Changes in Loan Values');
```



Conclusion

Following the work Monsaterolo [5 on page 4-247], this example demonstrates how the market shares of different energy sectors in different geographies change under specific climate scenarios. The market share changes are converted into market share shocks and you can use these shocks to compute the change in value of a portfolio of loans. Using this approach, you can model additional climate scenarios and then apply the Monsaterolo [5 on page 4-247] methodology. In addition, you can use this approach to value a portfolio of other assets such as bonds.

References

[1] Bank of Canada Climate Transition Scenario Data and pilot project available at https://www.bankofcanada.ca/2022/01/climate-transition-scenario-data/ and https://www.bankofcanada.ca/wp-content/uploads/2021/11/BoC-OSFI-Using-Scenario-Analysis-to-Assess-Climate-Transition-Risk.pdf.

[2] Climate IAM Explorer available at https://github.com/mathworks/Climate-IAM-Explorer.

[3] EPPA Model Structure available at https://globalchange.mit.edu/research/research-tools/eppa and https://globalchange.mit.edu/research/research-tools/human-system-model.

[4] LIMITS information available at https://tntcat.iiasa.ac.at/LIMITSDB/dsd?Action=htmlpage&page=about.

[5] Monasterolo, I., Zheng, Jiani I., and S. Battiston. "Climate Transition Risk and Development Finance: An Assessment of China's Overseas Energy Investments Portfolio." *China and the World Economy*. *26*, *6*(116–142), 2018. Available at https://doi.org/10.1111/cwe.12264.

See Also

Related Examples

- “Assess Physical and Transition Risk for Mortgages” on page 4-248

Assess Physical and Transition Risk for Mortgages

This example shows an approach to assess physical and transition risks for mortgages. Physical and transition risks are the two main categories of climate change risks. Physical risks for mortgages relate to natural events such as flooding and wildfires. Transition risks for mortgages derive from policy changes related to the transition away from fossil fuels. For example, a mortgage transition risk is a change in energy-efficiency standards for buildings.

Multiple institutions, including central banks, offer climate scenarios that include projections on emissions and economic variables. The economic variables include unemployment rate and gross domestic product (GDP) [1 on page 4-264], [2 on page 4-265], [4 on page 4-265]. Projections of physical variables are also available from meteorological agencies about changes in precipitation or sea-level rise [7 on page 4-265], [8 on page 4-265]. In some countries, central bank estimates project energy-efficiency upgrade costs [9 on page 4-265]. This example brings together economic variables, physical variables, and transition costs to assess the impact of physical and transition risk on mortgage loan provisions and capital requirements.

The workflow in this example is:

1. “”Define climate scenarios on page 4-249: In this example, the data is simulated, but captures trends similar to those in real climate scenario data sets. The economic variables in this example are residential and commercial real estate price indices. The only physical variable is precipitation change. There are three climate scenarios: "Early Action," "Delayed Action," and "No Action."
2. “”Compute baseline loan-to-value (LTV) ratio projections on page 4-252: The loan balance projections are straightforward for mortgages. To project the property value, this example uses the real estate price indices. This example uses the resulting loan-to-value (LTV) ratio as a reference to compare against the LTV ratio adjusted for physical and transition risk.
3. “”Specfiy physical risk on page 4-253: This example proposes a simple adjustment to the baseline house price index (HPI) scenario using physical variables and precipitation changes to adust the property value for flood risk.
4. “”Specify transition risk on page 4-255: This example makes adjustments to the property value based on its current energy efficiency rating and the estimated energy-efficiency transition costs necessary to reach a higher rating.
5. Adjust LTV ratio projections on page 4-257: With the property value adjusted for physical and transition risk, an adjusted LTV ratio is projected.
6. Estimate provisions and capital on page 4-258: This example assumes a probability of default (PD) model is available that includes LTV as a predictor. This model is the basis for the provisions and capital requirements computations. Use the PD model to compute the lifetime PD and the lifetime ECL, or provisions. Also use the PD model for capital requirements calculations in the form of risk-weighted assets (RWA).

Computational approaches to measure mortgage physical and transition risks require a combination of data from multiple sources, qualitative assessments, and expert judgement. The time horizon for this type of analysis is decades. There is no data to assess sensitivities to risk drivers because the climate scenarios have not yet been realized and not all the feedback loops and impacts are understood. Multiple models and assumptions must be brought together with qualitative adjustments and simplifications, and interdisciplinary collaboration is important. This example points out the places where different data sources, models, and qualitative adjustments intersect.

This example is extensible. You can apply additional economic variables to the analysis, such as unemployment rate, as long as the PD model also includes these variables. You can also add an LGD model to the analysis, as long as it is sensitive to the variables used in this analysis (for example, age of the loan, LTV ratio, and the economic variables). Multiple risks are explicitly excluded from this example, for example, exposure to wild fires or coastal flooding due to sea-level rise. You can use a similar approach to incorporate these physical risks into an analysis. Also, this example analyzes only an existing mortgage, and the effects of climate impact towards the end of the loan are not large because the credit risk is much smaller in the late years of the loan. You can complement this loan analysis with hypothetical new loans starting in the future. The loan analysis of this example focuses only on property value, yet additional risk considerations can include insurance considerations. For example, insurance considerations might be the rising costs of insurance due to more severe climate events or the fact that a property may become uninsurable. Therefore, the results of this example are limited and do not show a comprehensive assessment of the impact of climate change on mortgage loans.

Define Climate Scenarios

This example uses simulated data with the following three different climate scenarios:

- "**Early Action**" — The world takes immediate action and climate policies are put into effect. There may be an initial impact on the economy, but this scenario leads to the best conditions in the long run.
- "**Delayed Action**" — No significant policy changes take place until 2030. The climate policies at that point are more aggressive, hence initially there is a more important impact on the economy, but the economic conditions improve later on.
- "**No Action**" — The assumption is that no climate policies are in effect. Initially, there are less economic disruptions because there are no policy changes, but the effects of climate change have an impact on the economy in the long run. Also, the overall state of the economy is worse in this scenario than the other two scenarios by the end of the simulated time horizon. Arguably, the worst conditions in the "No Action" scenario would take place after year 2050 when the simulated scenarios end. These trends are similar to those in real climate scenario data sets.

The economic variables in this example are residential and commercial real estate price indices. These variables are available in some real climate scenario data sets. An alternative for the economic variables is a general price index, which is commonly available in climate scenario data.

Load the simulated data and generate plots for the residential and commercial price indices. These projected indices already incorporate climate impact in the aggregate level, showing overall trends in the average market value of the properties. Because the mortgage physical and transition risks are incorporated in the Compute LTV Projections Adjusted for Physical and Transition Risk on page 4-257 section, there are property-specific adjustmets to these property value projections.

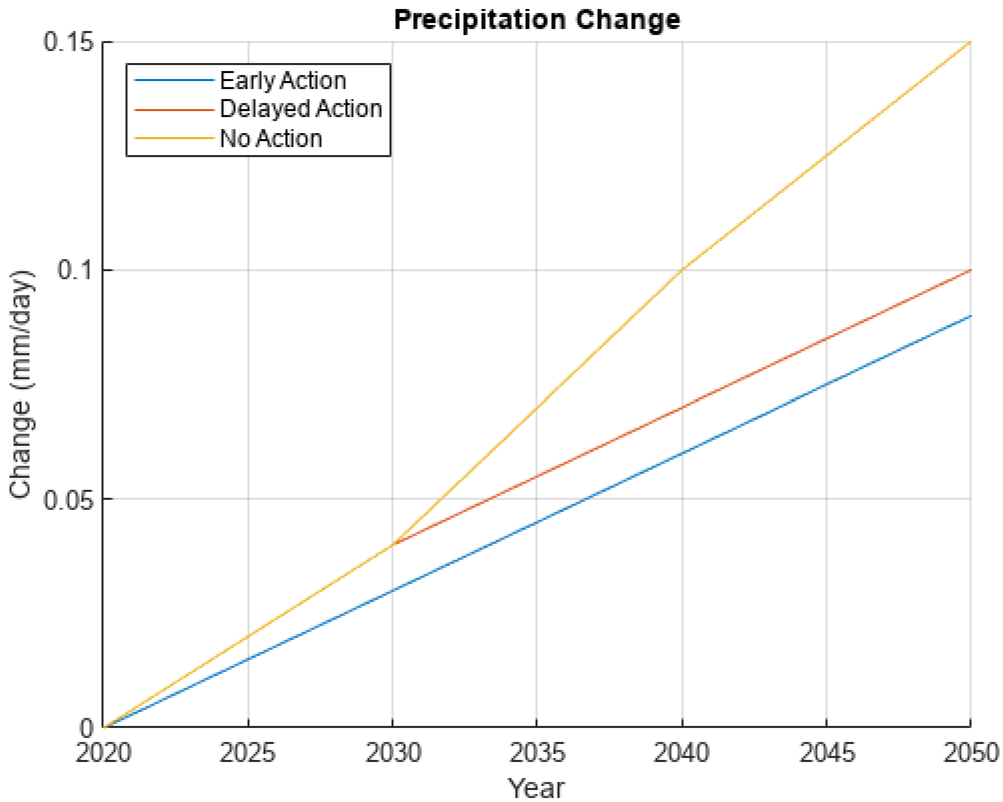
```
load SimulatedClimateScenarioData.mat
ScenarioLabels = ["Early Action" "Delayed Action" "No Action"];

figure
t = tiledlayout(2,1);

nexttile
PropertyType = "Residential";
VarName = strcat("RealEstate",PropertyType);
hold on
for s = ScenarioLabels
   Ind = EconomicVariables.Scenario == s;
```

```
    plot(EconomicVariables.Year(Ind),EconomicVariables.(VarName)(Ind))
end
hold off
title(PropertyType)
xlabel('Year')
ylabel('Index')
legend(ScenarioLabels,'Location','northwest')
grid on

nexttile
PropertyType = "Commercial";
VarName = strcat("RealEstate",PropertyType);
hold on
for s = ScenarioLabels
    Ind = EconomicVariables.Scenario == s;
    plot(EconomicVariables.Year(Ind),EconomicVariables.(VarName)(Ind))
end
hold off
title(PropertyType)
xlabel('Year')
ylabel('Index')
legend(ScenarioLabels,'Location','northwest')
grid on

title(t,"Real Estate Price Indices")
```

The simulated data also includes physical variables. In this example, there is information on precipitation change. The periodicity of these projections is not the same as the economic variables, so you must interpolate the data.

The increase in precipitation is worse for the "No Action" scenario. Properties with higher flood risk are affected by the precipitation changes more than properties with low flood risk. You can use this data for the physical risk adjustments to the property value projections.

```
PhysicalVariablesYearly = EconomicVariables(:,["Year" "Scenario"]);
PhysicalVariablesYearly.PrecipitationChange = zeros(height(PhysicalVariablesYearly),1);
for s = ScenarioLabels
   IndYearly = PhysicalVariablesYearly.Scenario == s;
   IndOrig = PhysicalVariables.Scenario == s;
   PhysicalVariablesYearly.PrecipitationChange(IndYearly) = interp1(PhysicalVariables.Year(IndOri
end

figure;
hold on
for s = ScenarioLabels
   Ind = PhysicalVariablesYearly.Scenario == s;
   plot(PhysicalVariablesYearly.Year(Ind),PhysicalVariablesYearly.PrecipitationChange(Ind))
end
hold off
title('Precipitation Change')
xlabel('Year')
ylabel('Change (mm/day)')
legend(ScenarioLabels,'Location','northwest')
grid on
```



Compute Baseline Loan-to-Value Projections

The loan-to-value (LTV) ratio is the main link between the scenario variables and the credit analysis. You can compare a baseline projection against projections that include the effects of the physical and transition risk.

Start by choosing a climate scenario.

```
ScenarioChoice = Early Action ;
ScenarioInd = EconomicVariables.Scenario == ScenarioChoice;
```

For the LTV projections, project the mortgage balance and the property value separately, and then work with yearly projections using 2020 as the current year.

To project the mortgage balance, start from the current loan balance and use the age, term, and loan's interest rate to project the balance forward, using standard annuity techniques. Define the exposure at the end of the year as the scheduled payment amount plus the remaining balance after the payment.

```
CurrentYear = 2020; % Assume end of year
Age = 12; % Assume loan is already this age, next end of year will be age + 1
Term = 30; % Original mortgage term
Rate = 0.0575; % Assume fixed rate
CurrentBalance = 90000; % Current mortgage balance at end of current year

[PrincipalPayment,InterestPayment,RemainingBalance] = amortize(Rate,(Term-Age),CurrentBalance);
```

```
Projections = table;
Projections.Year = (CurrentYear+1:CurrentYear+(Term-Age))';
Projections.Age = (Age+1:Term)';
% Exposure at the end of next year is the expected payment amount plus remaining
% balance after the payment
Projections.LoanBalance = PrincipalPayment'+InterestPayment'+RemainingBalance';
```

For the property value, start out with the current property value and use the corresponding real estate price index (either residential or commercial) to project the value into the future. Different climate scenarios lead to different projected values.

You can compute the LTV ratio projections from the loan balance and the property values in each time period.

```
CurrentValue = 150000;
PropertyType = Commercial;

PropertyTypeVarName = strcat("RealEstate",PropertyType);
Projections = join(Projections,EconomicVariables(ScenarioInd,["Year" PropertyTypeVarName]));
Projections.Properties.VariableNames{end} = 'PriceIndex';
Projections.ValueReference = CurrentValue*Projections.PriceIndex/100;
Projections.LTVReference = Projections.LoanBalance./Projections.ValueReference;
disp(Projections)
```

Year	Age	LoanBalance	PriceIndex	ValueReference	LTVReference
2021	13	95175	101.2	1.518e+05	0.62698
2022	14	92022	102.4	1.536e+05	0.5991
2023	15	88687	103.6	1.554e+05	0.5707
2024	16	85161	104.8	1.572e+05	0.54174
2025	17	81432	106	1.59e+05	0.51215
2026	18	77489	107.8	1.617e+05	0.47921
2027	19	73319	109.6	1.644e+05	0.44598
2028	20	68909	111.4	1.671e+05	0.41238
2029	21	64245	113.2	1.698e+05	0.37836
2030	22	59313	115	1.725e+05	0.34385
2031	23	54098	116.8	1.752e+05	0.30878
2032	24	48583	118.6	1.779e+05	0.27309
2033	25	42751	120.4	1.806e+05	0.23672
2034	26	36583	122.2	1.833e+05	0.19958
2035	27	30061	124	1.86e+05	0.16162
2036	28	23164	126.2	1.893e+05	0.12237
2037	29	15870	128.4	1.926e+05	0.082399
2038	30	8156.7	130.6	1.959e+05	0.041637

This result is a reference LTV ratio for the loan, for the selected climate scenario. The Physical Risk for Floods on page 4-253 and Transition Risk for Energy Efficiency Upgrades on page 4-255 sections describe the additional impact on the property value, specific to each mortgage.

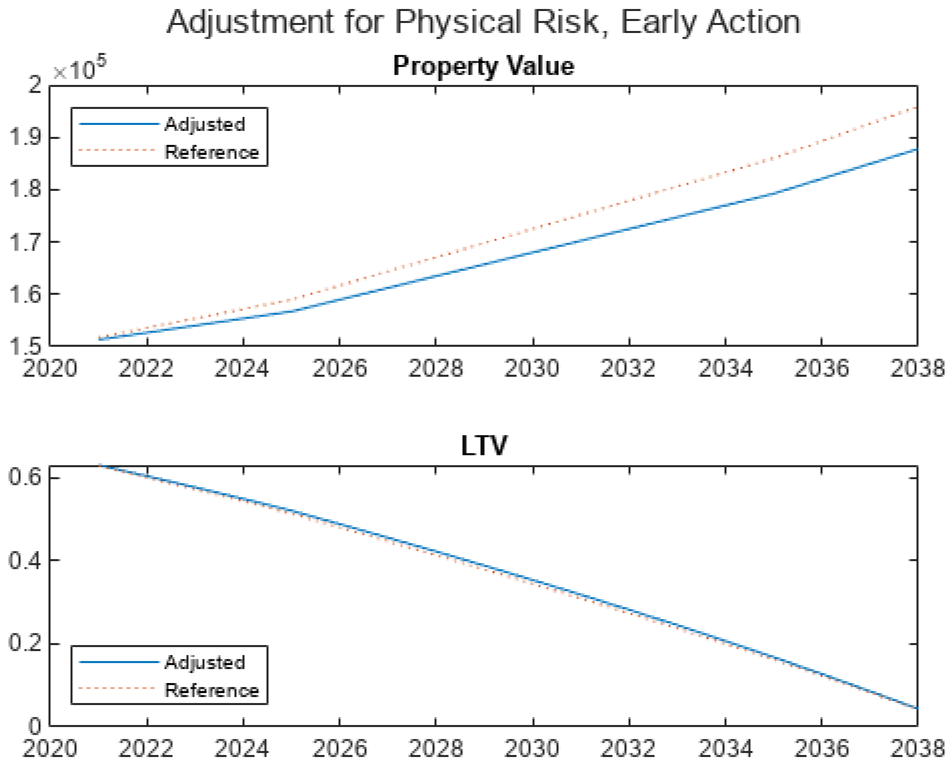
Specify Physical Risk for Floods

To incorporate risk of flooding, adjust the price index using property-specific flood risk information from the property with a sensitivity parameter for precipitation change. Assume there are known flood risk ratings for the properties. For properties in a flood area, reduce the baseline property value projections as the projected precipitation increases.

This simple approach that requires qualitative views to determine the sensitivity parameters. You could incorpoate a more sophisticated model where the projected drop in property value includes additional information from the property. You might distinguish additional types of flood risk. For example, precipitation changes are significant for flash-flooding risk. However, the value of coastal properties is sensitive to sea-level rise, which is a physical variable often projected with climate scenario data. Interdisciplinary collaboration can greatly enhance the granularity and quality of these adjustments.

Calibrate the sensitivity parameters. Then, display the impact of the flood risk adjustment on the property value at the end of the mortgage term.

```
Projections = join(Projections,PhysicalVariablesYearly(ScenarioInd,["Year" "PrecipitationChange"

FloodRisk = High ;
switch FloodRisk
   case "High"
      PrecipitationSens = -100;
   case "Medium"
      PrecipitationSens = -50;
   case "Low"
      PrecipitationSens = 25;
end

Projections.PriceIndexPhysical = Projections.PriceIndex + PrecipitationSens*Projections.Precipita
Projections.ValuePhysical = CurrentValue*Projections.PriceIndexPhysical/100;
Projections.LTVPhysical = Projections.LoanBalance./Projections.ValuePhysical;

figure
t = tiledlayout(2,1);
nexttile
plot(Projections.Year,Projections.ValuePhysical,'-',Projections.Year,Projections.ValueReference,
title('Property Value')
legend('Adjusted','Reference','Location','northwest')
nexttile
plot(Projections.Year,Projections.LTVPhysical,'-',Projections.Year,Projections.LTVReference,':')
title('LTV')
legend('Adjusted','Reference','Location','southwest')
title(t,strcat("Adjustment for Physical Risk, ",ScenarioChoice))
```




```
fprintf('Value adjustment due to physical risk in the last year of the mortgage (%d): %4.2f%%',Pr
```

```
Value adjustment due to physical risk in the last year of the mortgage (2038): -4.13%
```

Even though the property value drop is noticeable at the end of the mortgage term, the effect on the LTV ratio is small. This effect is because mortgages are amortizing loans and the balance and the corresponding LTV ratio are small toward the maturity of the loan. However, if a future loan is assessed, the impact on LTV ratio would be more significant near the origination of the loan, resulting in a different impact on the credit analysis.

Specify Transition Risk for Energy Efficiency Upgrades

Changing regulations for energy efficiency of properties represent a transition risk. For example, new regulations may require a minimum energy efficiency rating to rent a property. An energy efficiency rating itself may be required to sell a property in order to inform prospective buyers and lenders about potential maintenance and upgrade costs. New energy efficiency regulations affect the market value of a property.

This example assumes there are known energy efficiency ratings for the properties. Examples of these efficiency ratings are Energy Perfomance Certificates [9 on page 4-265] or Energy Star [10 on page 4-265]. This example uses a simulated scale with five levels: "Low", "Medium Low", "Medium", "Medium High", and "High":

```
EnergyRatingScale = string(EnergyEfficiencyUpgradeCost.Properties.VariableNames);
disp(EnergyRatingScale)
```

```
    "High"    "Medium High"    "Medium"    "Medium Low"    "Low"
```

Assume that there are estimates of the cost to upgrade to a higher rating. The estimates in this example are simulated, but an example of these kinds of estimates can be found in the CBES Guidance document [3 on page 4-265].

```
disp(EnergyEfficiencyUpgradeCost)
```

	High	Medium High	Medium	Medium Low	Low
High	0	NaN	NaN	NaN	NaN
Medium High	30000	0	NaN	NaN	NaN
Medium	40000	20000	0	NaN	NaN
Medium Low	50000	30000	12000	0	NaN
Low	70000	50000	35000	25000	0

To enhance this example, you could replace these estimates with a more granular model, where different characteristics of the property can help predict the upgrade costs.

Assume each property has a current rating and that there is a maximum attainable rating.

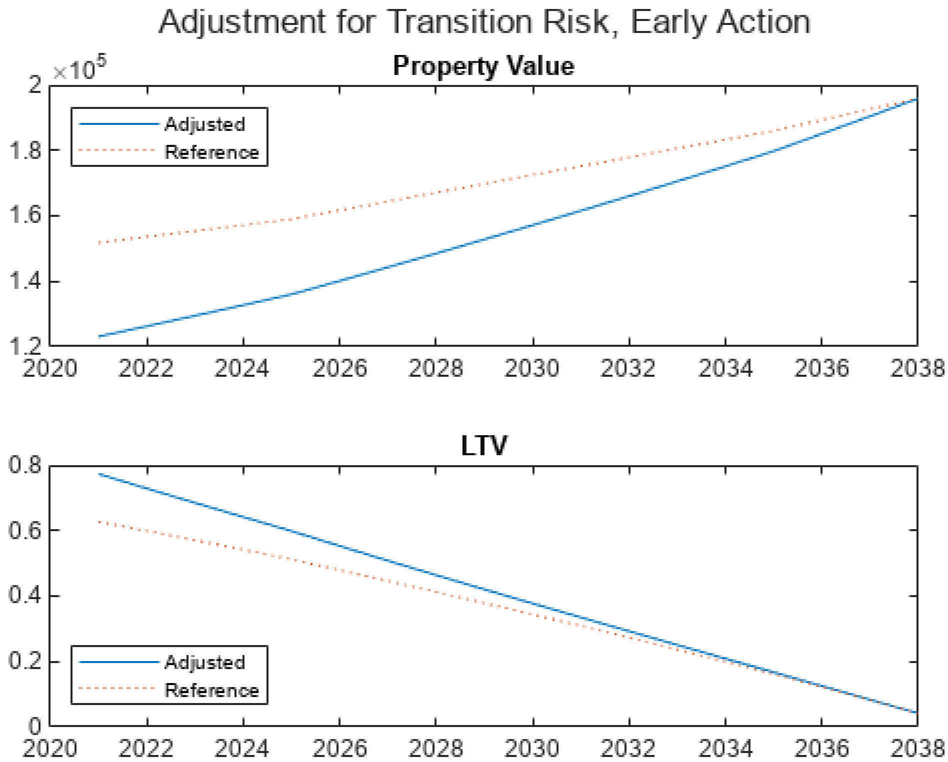
```
CurrentEER = Medium Low ;
MaxEER = Medium High ; % Cannot be worse than current rating
```

```
TotalEnergyUpgradeCost = EnergyEfficiencyUpgradeCost{CurrentEER,MaxEER}
```

```
TotalEnergyUpgradeCost = 30000
```

The estimated cost is an estimated amount as of the current year. In the "Early Action" climate scenario, deduct this cost from the property value immediately, where the rationale is that this cost would be deducted from the property price when the property is sold. Then assume improvements are made in the following years until the property reaches its maximum energy efficiency rating. The initial drop in value gradually disappears and the adjusted projected property value matches the baseline value at the end of the loan. For the "Delayed Action" climate scenario, the same approach applies except the drop in value occurs in 2030 and the subsequent investments occur in a shorter time span. The yearly costs are adjusted by the price index of the corresponding scenario. In the "No Action" climate scenario, the upgrades never take place and, therefore, there is no transition risk impact.

```
Projections.ValueTransition = adjustValueProjections(Projections.PriceIndex,CurrentValue,TotalEne
Projections.LTVTransition = Projections.LoanBalance./Projections.ValueTransition;

figure
t = tiledlayout(2,1);
nexttile
plot(Projections.Year,Projections.ValueTransition,'-',Projections.Year,Projections.ValueReference
title('Property Value')
legend('Adjusted','Reference','Location','northwest')
nexttile
plot(Projections.Year,Projections.LTVTransition,'-',Projections.Year,Projections.LTVReference,':
title('LTV')
legend('Adjusted','Reference','Location','southwest')
title(t,strcat("Adjustment for Transition Risk, ",ScenarioChoice))
```



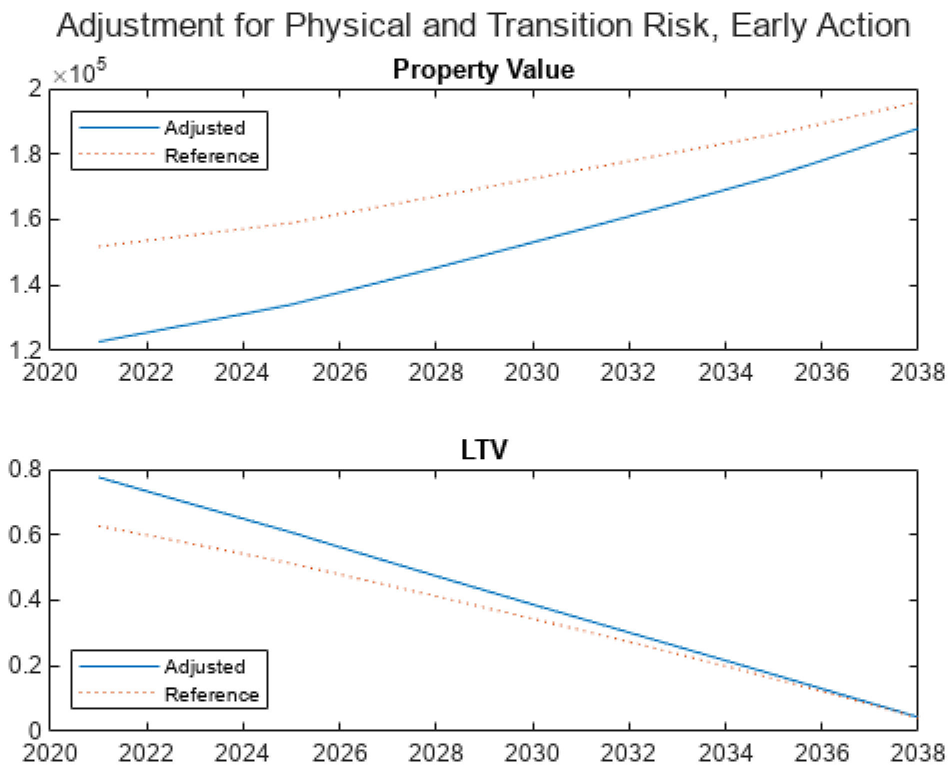
Different implementations are possible to adjust the property value with the transition costs. It is also possible to include disposable income in the analysis to adjust the disposable income by the investments required each year to enhance the efficiency of the property. In that case, the credit models in this analysis would need to include disposable income as a predictor to capture the effect of the projections for provisions or capital. You can add other types of transition risk to this approach as long as the impact of these risks can be captured in the projected values of the variables included in the credit models.

Compute LTV Projections Adjusted for Physical and Transition Risk

To adjust LTV projections for both physical and transition risks, combine physical and transition risk adjustments from the Physical Risk for Floods on page 4-253 and Transition Risk for Energy Efficiency Upgrades on page 4-255 sections.

```
Projections.ValuePhysicalTransition = adjustValueProjections(Projections.PriceIndexPhysical,Curre
Projections.LTVPhysicalTransition = Projections.LoanBalance./Projections.ValuePhysicalTransition

figure
t = tiledlayout(2,1);
nexttile
plot(Projections.Year,Projections.ValuePhysicalTransition,'-',Projections.Year,Projections.ValueR
title('Property Value')
legend('Adjusted','Reference','Location','northwest')
nexttile
plot(Projections.Year,Projections.LTVPhysicalTransition,'-',Projections.Year,Projections.LTVRefer
title('LTV')
```

```
legend('Adjusted','Reference','Location','southwest')
title(t,strcat("Adjustment for Physical and Transition Risk, ",ScenarioChoice))
```



Estimate Provisions and Capital

To assess the impact of the physical and transition risk adjustments, use credit models that include the LTV ratio as a predictor and then estimate the provisions and capital with and without the adjustments.

This example uses a lifetime probability of default (PD) model (see `fitLifetimePDModel`) that includes the LTV ratio and the age of the mortgage as predictors.

```
load SimulatedClimatePDModel.mat
disp(ClimatePDModel)
```

```
  Probit with properties:

            ModelID: "Probit"
        Description: "Simulated ad-hoc lifetime PD model for climate risk analysis of mortgages."
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "Age"
           LoanVars: "LTV"
          MacroVars: ""
        ResponseVar: "Default"
```

An ad-hoc model is fitted for this analysis, including the projected variables of interest, in this case the LTV ratio. The training data is a historical data set that is representative of the current mortgage loan portfolio. Alternatively, you could use an existing PD model for a mortgage loan portfolio if it includes the projected variables as predictors. You can create this type of PD model using `customLifetimePDModel`. In this example, age is included because it is a common predictor variable for credit models, especially for lifetime PD models in the context of lifetime expected credit losses. You can compute lifetime expected credit loses (ECL) using `portfolioECL`. For more information, see “Overview of Lifetime Probability of Default Models” on page 1-25.

You can also include other economic variables in this analysis. Climate scenarios typically include other economic variables commonly included in risk models, such as the unemployment rate. You can add these additional variables as long as the credit model uses these as predictors.

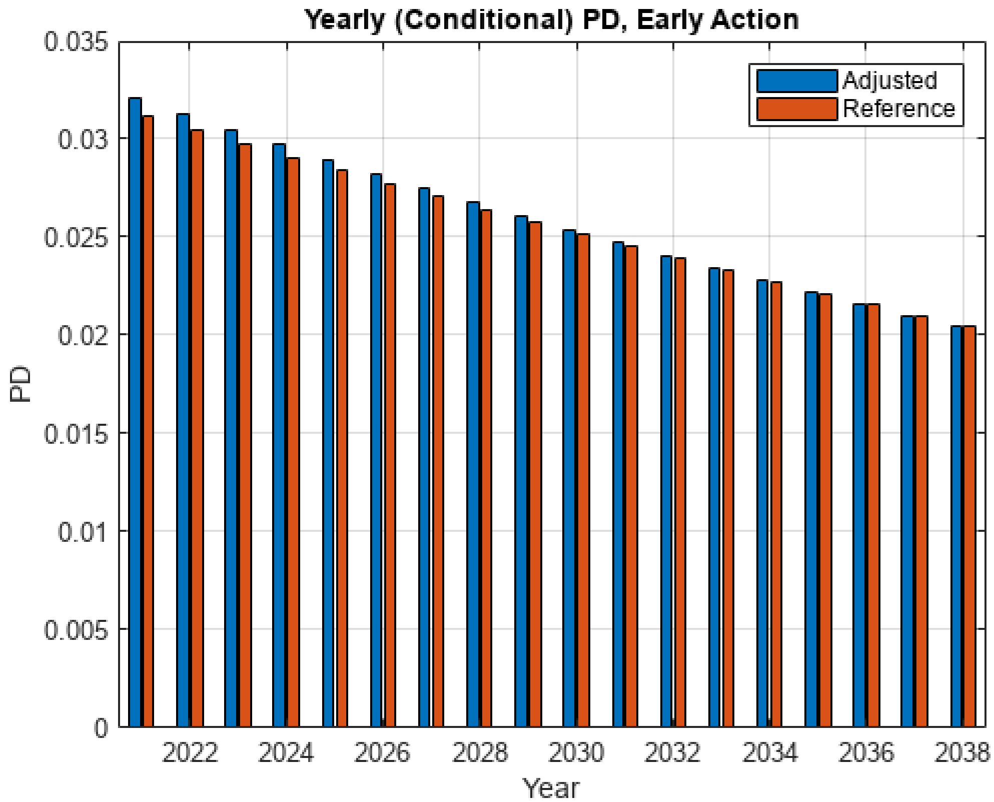
You can also include a loss given default (LGD) model (see `fitLGDModel`) in this analysis, provided it includes some or all of the projected loan variables and economic variables as predictors. An exposure-at-default (EAD) model (see `fitEADModel`) is not as relevant for mortgages because they use a standard payment schedule.

Reformat the adjusted projections to use these projections with the `predict` function for the credit model.

```
dataPredictAdjusted = Projections(:,{'Year' 'Age' 'LTVPhysicalTransition'});
dataPredictAdjusted.Properties.VariableNames{3} = 'LTV';
dataPredictAdjusted = addvars(dataPredictAdjusted,ones(height(dataPredictAdjusted),1),'NewVariab

dataPredictReference = dataPredictAdjusted;
dataPredictReference.LTV = Projections.LTVReference;
```

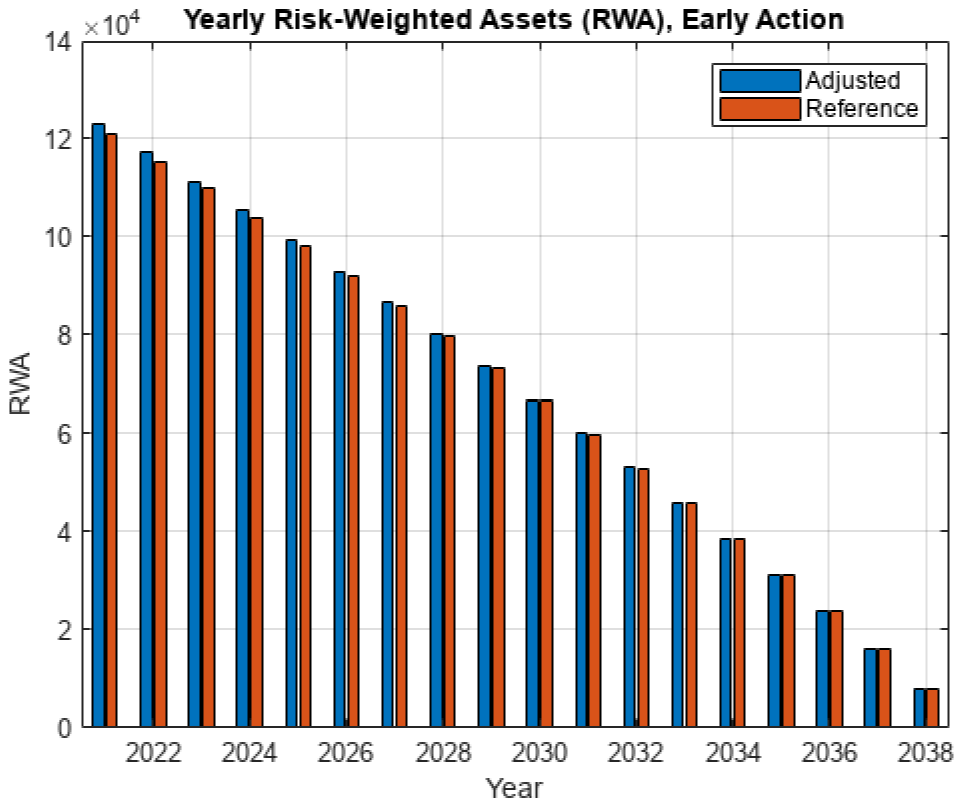
Calculate the yearly, conditional PD using `predict`.

```
PDAdjusted = predict(ClimatePDModel,dataPredictAdjusted);
PDReference = predict(ClimatePDModel,dataPredictReference);

figure;
bar(dataPredictAdjusted.Year,[PDAdjusted,PDReference])
title(strcat("Yearly (Conditional) PD, ",ScenarioChoice))
xlabel('Year')
ylabel('PD')
legend('Adjusted','Reference')
grid on
```



The PD values decrease with age. In the early years (2022 to 2024), the difference between the reference and adjusted projections are noticeable because they have an impact on the early PD values. For climate risk, more adjustments to projections happen many years into the future, where the PD values are smaller and have a decreasing the impact on downstream credit computations. One such downstream computation is the yearly risk-weighted assets (RWA) value, which is equivalent to the capital requirements. The computation of RWA follows Basel II and uses the asymptotic single risk factor (ASRF) model (see `asrf`). In this example, the `asrf` function uses the conditional yearly PD, LGD, and EAD values. If an LGD model is available, you can use its predictions here. For EAD, use the projected loan balances.

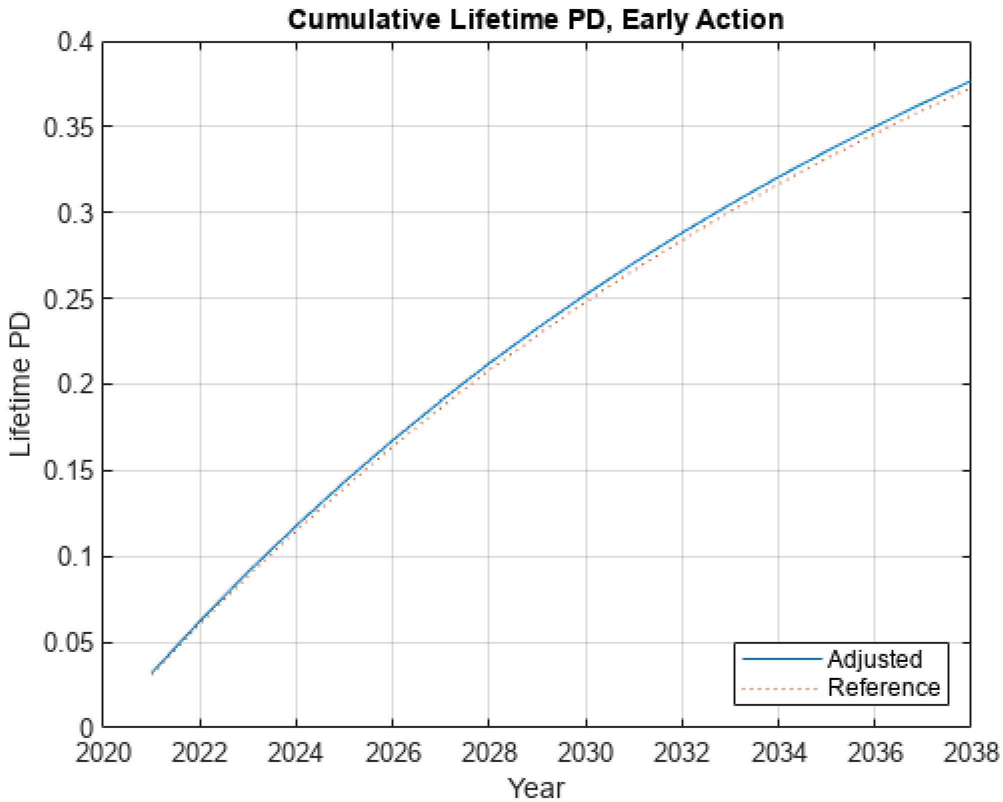
```
LGD = 0.50;
EAD = Projections.LoanBalance;
MortgageAssetCorrelation = 0.15;

CapAdjusted = asrf(PDAdjusted,LGD,MortgageAssetCorrelation,'EAD',EAD);
RWAAdjusted = 12.5*CapAdjusted;

CapReference = asrf(PDReference,LGD,MortgageAssetCorrelation,'EAD',EAD);
RWAReference = 12.5 * CapReference;

figure;
bar(dataPredictAdjusted.Year,[RWAAdjusted,RWAReference])
title(strcat("Yearly Risk-Weighted Assets (RWA), ",ScenarioChoice))
xlabel('Year')
ylabel('RWA')
legend('Adjusted','Reference')
grid on
```



The RWA value decreases with time because mortgages are amortizing loans and the decreasing PD values also contribute to this pattern.

To compute provisions, use a lifetime credit analysis. Start by comparing the cumulative lifetime PD values using `predictLifetime`.

```
LPDAdjusted = predictLifetime(ClimatePDModel,dataPredictAdjusted); % With adjusted LTV
LPDReference = predictLifetime(ClimatePDModel,dataPredictReference); % With reference LTV

figure;
plot(dataPredictAdjusted.Year,LPDAdjusted,'-',dataPredictReference.Year,LPDReference,':')
title(strcat("Cumulative Lifetime PD, ",ScenarioChoice))
xlabel('Year')
ylabel('Lifetime PD')
legend('Adjusted','Reference','Location','southeast')
grid on
```

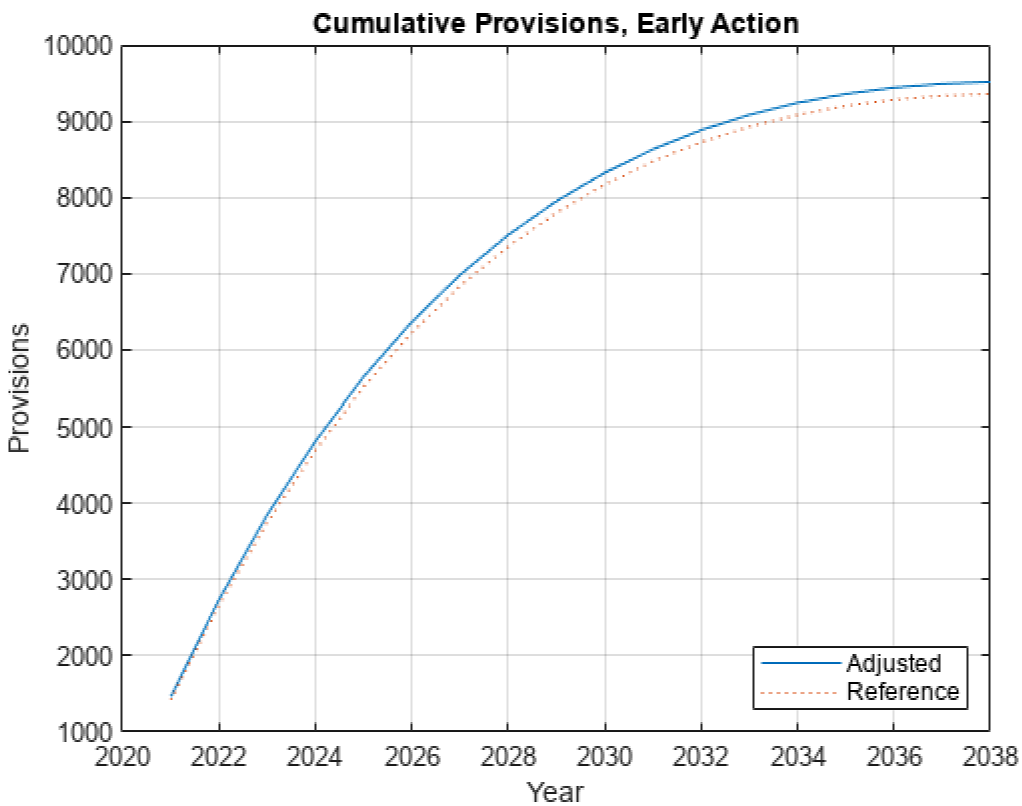


Use the `portfolioECL` function to get the lifetime ECL, or lifetime provisions. To get this projection, you need the marginal version of the lifetime PD values, the LGD and EAD values, and an effective interest rate for the loan.

```
NumRemainingYears = height(Projections);
MPDTable = table;
MPDTable.ID = ones(NumRemainingYears,1);
% Marginal PDs for adjusted
MPDTable.MPD = predictLifetime(ClimatePDModel,dataPredictAdjusted,'ProbabilityType','marginal');

LGDTable = table;
LGDTable.ID = 1;
LGDTable.LGD = 0.50;

EADTable = table;
EADTable.ID = ones(NumRemainingYears,1);
EADTable.EAD = Projections.LoanBalance;

EIR = 0.045; % Effective interest rate

[LifetimeECLAdjusted,~,ECLAdjustedPerPeriod] = portfolioECL(MPDTable,LGDTable,EADTable,Periodici

% Marginal PDs for reference
MPDTable.MPD = predictLifetime(ClimatePDModel,dataPredictReference,'ProbabilityType','marginal')
[LifetimeECLReference,~,ECLReferencePerPeriod] = portfolioECL(MPDTable,LGDTable,EADTable,Periodi
```

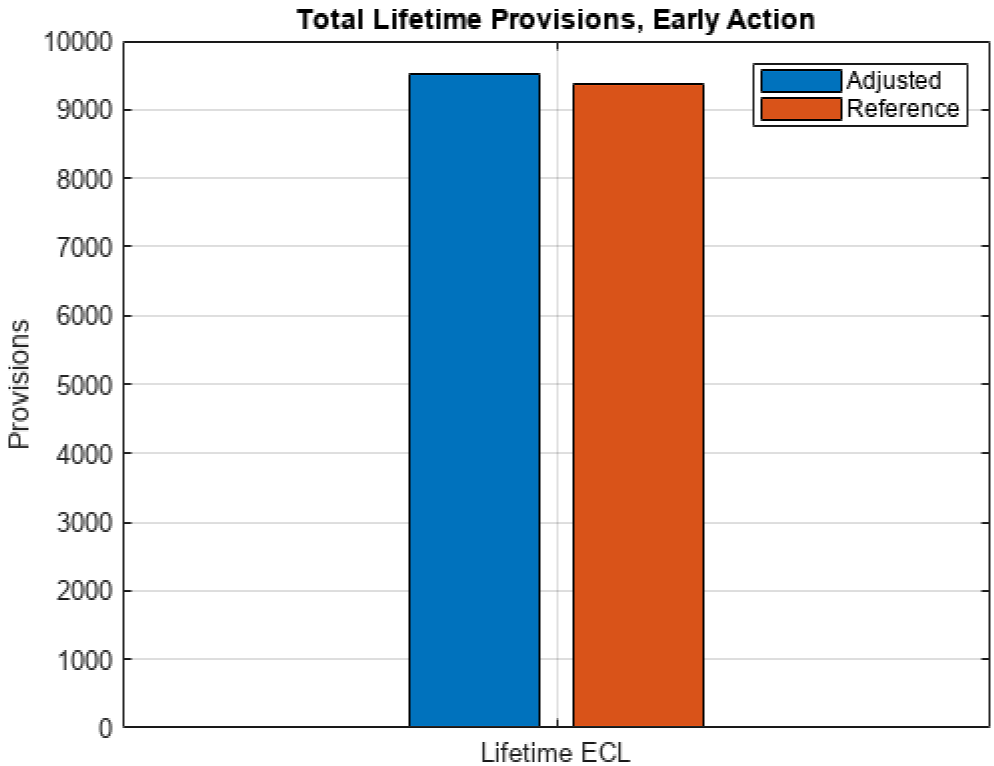
The following plot shows how the discounted yearly provisions accumulate during the remaining life of the loan.

```
CumulECLAdjusted = cumsum(ECLAdjustedPerPeriod.Scenario1);
CumulECLReference = cumsum(ECLReferencePerPeriod.Scenario1);
figure;
plot(dataPredictAdjusted.Year,CumulECLAdjusted,'-',dataPredictReference.Year,CumulECLReference,'
xlabel('Year')
ylabel('Provisions')
title(strcat("Cumulative Provisions, ",ScenarioChoice))
grid on
legend('Adjusted','Reference','Location','southeast')
```



The following plot shows the total impact that the physical and transition risk adjustments have on lifetime provisions.

```
figure;
bar(categorical({'Lifetime ECL'}),[LifetimeECLAdjusted LifetimeECLReference])
title(strcat("Total Lifetime Provisions, ",ScenarioChoice))
ylabel('Provisions')
legend('Adjusted','Reference')
grid on
```




```
fprintf('Lifetime provisions, percent increase / decrease relative to reference: %4.2f%%',100*(Li
```

```
Lifetime provisions, percent increase / decrease relative to reference: 1.67%
```

The "Early Action" climate scenario shows a larger increase with respect to the baseline. However, this increase may be because the shocks occur early, where the exposure and PD values are larger. A hypothetical loan starting in the future may lead to different impacts. A more comprehensive analysis may better capture the impact of future shocks.

Conclusion

This example shows a workflow to incorporate some physical and transition risks into a climate-related credit analysis for mortgages. The methodology in this example is a simple approach to bring together existing information and models to assess the impact of climate change on provisions.

Climate risk is a complex developing area and this example is a starting point that you can extend in different directions. You might add more variables (such as rate of unemployment or sea-level rise) and other models (such as LGD, flood, or cost models) to the analysis. Bringing an entire portfolio of mortgages could also shed light on portfolio composition or potentially a dynamic balance sheet analysis. You might also explore debt serviceability to analyze the disposable income projections in the analysis. Another area of investigation could be additional insurance considerations, such as projections on the cost of insurance or whether a property is no longer insurable.

References

[1] Bank of Canada, *Climate Transition Scenario Data*, https://www.bankofcanada.ca/2022/01/climate-transition-scenario-data/.

[2] Bank of England, *Key elements of the 2021 Biennial Exploratory Scenario: Financial risks from climate change*, June 2021, https://www.bankofengland.co.uk/stress-testing/2021/key-elements-2021-biennial-exploratory-scenario-financial-risks-climate-change.

[3] Bank of England, *Guidance for participants of the 2021 Biennial Exploratory Scenario: Financial risks from climate change*, June 2021, https://www.bankofengland.co.uk/-/media/boe/files/stress-testing/2021/the-2021-biennial-exploratory-scenario-on-the-financial-risks-from-climate-change.pdf.

[4] European Central Bank, Banking Supervision, *Climate Risk Stress Test*, January 2022, https://www.bankingsupervision.europa.eu/ecb/pub/pdf/ssm.macrofinancialscenariosclimateriskstresstest2022~bcac934986.en.pdf.

[5] U.S. Federal Government, 2022: U.S. Climate Resilience Toolkit. [Online] http://toolkit.climate.gov. Climate Explorer: https://toolkit.climate.gov/tool/climate-explorer-0. Accessed June, 2022.

[6] Intergovernmental Panel on Climate Change (IPCC), https://www.ipcc.ch/.

[7] National Oceanic and Atmospheric Administration (NOAA), https://www.noaa.gov/.

[8] Met Office, https://www.metoffice.gov.uk/.

[9] Energy Performance Certificate Wiki: https://en.wikipedia.org/wiki/Energy_performance_certificate.

[10] Energy Star Wiki: https://en.wikipedia.org/wiki/Energy_Star.

Local Functions

```
function ValueAdjusted = adjustValueProjections(PriceIndex,CurrentValue,TotalCost,ScenarioChoice)

NumRemainingYears = length(PriceIndex);
ValueAdjusted = zeros(NumRemainingYears,1);
PriceIndexPrevious = [100;PriceIndex(1:end-1)];
PriceIndexRate = (PriceIndex-PriceIndexPrevious)./PriceIndexPrevious;

switch ScenarioChoice
   case "Early Action"
      CostPerYear = TotalCost/NumRemainingYears;
      TransitionCost = CostPerYear*PriceIndex/100;
      ValueAdjusted(1) = (CurrentValue-TotalCost)*(1+PriceIndexRate(1)) + TransitionCost(1);
      for ii=2:NumRemainingYears
         ValueAdjusted(ii) = ValueAdjusted(ii-1)*(1+PriceIndexRate(ii)) + TransitionCost(ii);
      end
   case "Delayed Action"
      TransitionCost = zeros(NumRemainingYears,1);
      ValueAdjusted = CurrentValue*PriceIndex/100; % Initialize
```

```
        if NumRemainingYears>=10
           % TotalCost = TotalCost*PriceIndex(9)/100; % Adjust by price index
           CostPerYear = TotalCost/(NumRemainingYears-9); % Make upgrades in remaining years
           TransitionCost(10:end) = CostPerYear*PriceIndex(10:end)/100;
           ValueAdjusted(10) = (ValueAdjusted(9)-TotalCost*PriceIndex(9)/100)*(1+PriceIndexRate(10
           for ii=11:NumRemainingYears
              ValueAdjusted(ii) = ValueAdjusted(ii-1)*(1+PriceIndexRate(ii)) + TransitionCost(ii);
           end

        end
     case "No Action"
        ValueAdjusted = CurrentValue*PriceIndex/100; % No upgrades
  end

  end
```

See Also

Related Examples

- “Measure Transition Risk for Loan Portfolios with Respect to Climate Scenarios” on page 4-231

More About

- “Overview of Lifetime Probability of Default Models” on page 1-25

External Websites

- Modeling the Impact of Transition and Physical Climate Risks on a Portfolio of Mortgages (13 min 52 sec)

Analyze Transition Scenarios for Climate-Related Financial Risks

This example shows how to visualize transition scenarios to understand climate-related risks to the economy and financial systems.

Background

In late 2020, the Bank of Canada initiated a project to understand the climate impact on financial systems of Canada and the United States. The resulting data set [1] on page 4-278 is the basis of the Bank of Canada report "Transition Scenarios for Analyzing Climate-Related Financial Risk" [2] on page 4-278. From this project, data for three climate scenarios captures the evolution of the global economy. These three climate scenarios are in the `BankOfCanadaClimateScenarioData.mat` file. The Bank of Canada report summarizes the global economy by ten emission-intensive sectors across eight global regions from 2020 to 2050. A fourth scenario is the benchmark scenario and reflects the climate policies of 2019., which mitigates effects due to the COVID-19 pandemic that started in 2020.

This example consists of two parts that relate to section 2 and section 4 of the Bank of Canada report.

Climate Impact of Green House Gas Emissions on page 4-267, uses MATLAB® code to re-create the graphs for each scenario that show the progress in the mitigation of greenhouse gas emissions to 2050 and the impact of natural-based solutions such us forests.

Impact of Climate Policies on page 4-271, uses MATLAB® code to re-create the graphs that demonstrate how the scenario policies affect economies at a global, regional, and sectoral level with the focus on Canada and the United States. Graphs in this example provide the following information:

- Impacts in greenhouse gas emissions due to the increase in shadow carbon price
- Financial impacts in terms of net income
- Macroeconomic impacts in terms of gross domestic product (GDP)

For information on the climate impact of different transition scenarios on loan default probabilities, see “Measure Transition Risk for Loan Portfolios with Respect to Climate Scenarios” on page 4-231.

Impact of Green House Gas Emissions

Climate Scenario Data

The data for this example includes these climate scenarios developed by MIT using the Economic Projection and Policy Analysis (EPPA) Mode in collaboration with the Bank of Canada [3 on page 4-278].

- *Baseline (2019 Policies)* — Baseline scenario consistent with climate policies in place at the end of 2019
- *Below 2°C Immediate* — Immediate policy action scenario to limit average global warming to below 2°C by 2100
- *Below 2°C Delayed* — Delayed policy action scenario to limit average global warming to below 2°C by 2100
- *Net-Zero 2050 (1.5°C)* — More ambitious immediate policy action scenario to limit average global warming to 1.5°C by 2050 that includes current net-zero commitments by some countries

The climate transition scenario data is provided by the Bank of Canada and is available free of charge at www.bankofcanada.ca. [1 on page 4-278] on page 4-278

Load this data, converted to a MAT-file from `BankOfCanadaClimateScenarioData.mat`.

```
load BankOfCanadaClimateScenarioData.mat;
head(ClimateTransitionScenarioData);
```

```
    k    CL_GEOGRAPHY    CL_SECTOR            CL_VARIABLE                     CL_UNIT
    _    ____________    _________    _______________________________    ___________________    _

    1       Canada       National     Carbon price                       US$2014/tCO2e          E
    2       Canada       National     Carbon price                       US$2014/tCO2e          E
    3       Canada       National     Emissions | total GHG (scope 1)    Million tonnes CO2e    E
    4       Canada       National     Emissions | total GHG (scope 1)    Million tonnes CO2e    E
    5       Canada       National     Input price | Coal                 Index (2014 = 1)       E
    6       Canada       National     Input price | Coal                 Index (2014 = 1)       E
    7       Canada       National     Input price | Crops                Index (2014 = 1)       E
    8       Canada       National     Input price | Crops                Index (2014 = 1)       E
```

Use the preprocessBankOfCanadaData on page 4-285 helper function to keep only the variables that this example uses for analysis.

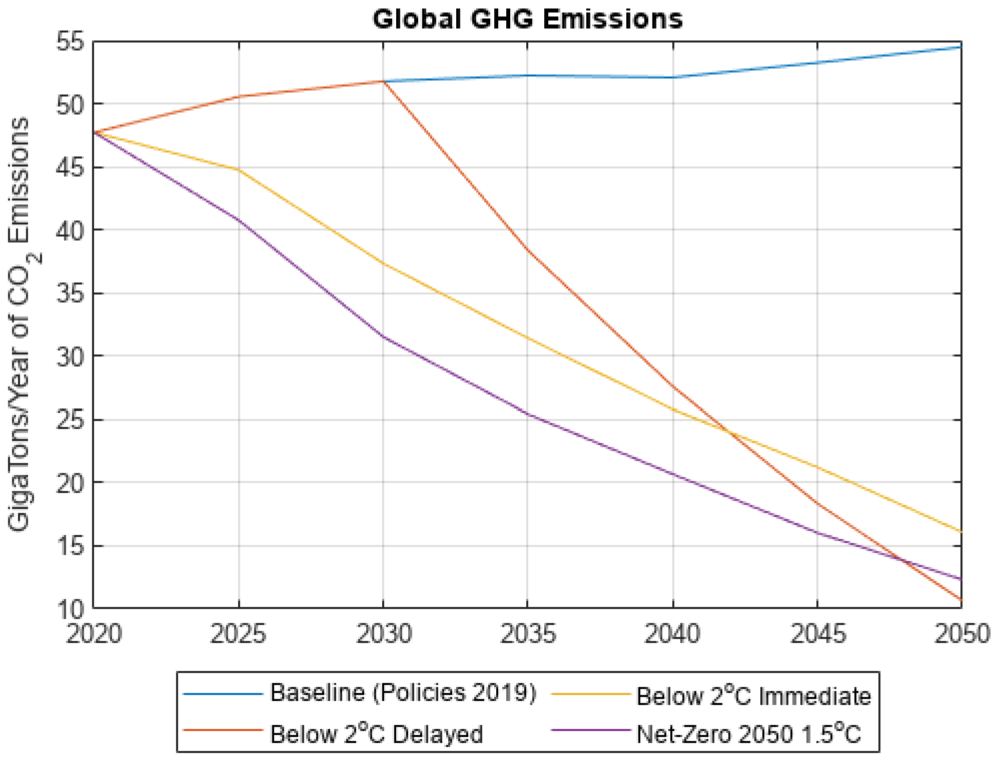
```
 [ClimateTransitionScenarioData, options] = preprocessBankOfCanadaData(ClimateTransitionScenarioD
 regions = options.regions;
```

Global Carbon Dioxide Emissions

The following plot shows the impact of each climate scenario in terms of carbon dioxide (CO_2) emissions. To understand the various scenarios and their effect on the economy, first you need to explore how each scenario affects the total emissions of carbon dioxide globaly and for each geographic region. The contribution of the forests is important in this analysis because forests can act as both carbon dioxide emmiters and carbon dioxide sinks.

Use the plotVariableByCountry on page 4-279 function to plot the global greenhouse gas (GHG) emissions.

```
options = updateOptions(options,1e-3,"Global GHG Emissions","GigaTons/Year of CO_2 Emissions");
plotVariableByCountry('Global',{'Emissions | total GHG (scope 1)'},'Global', options);
```



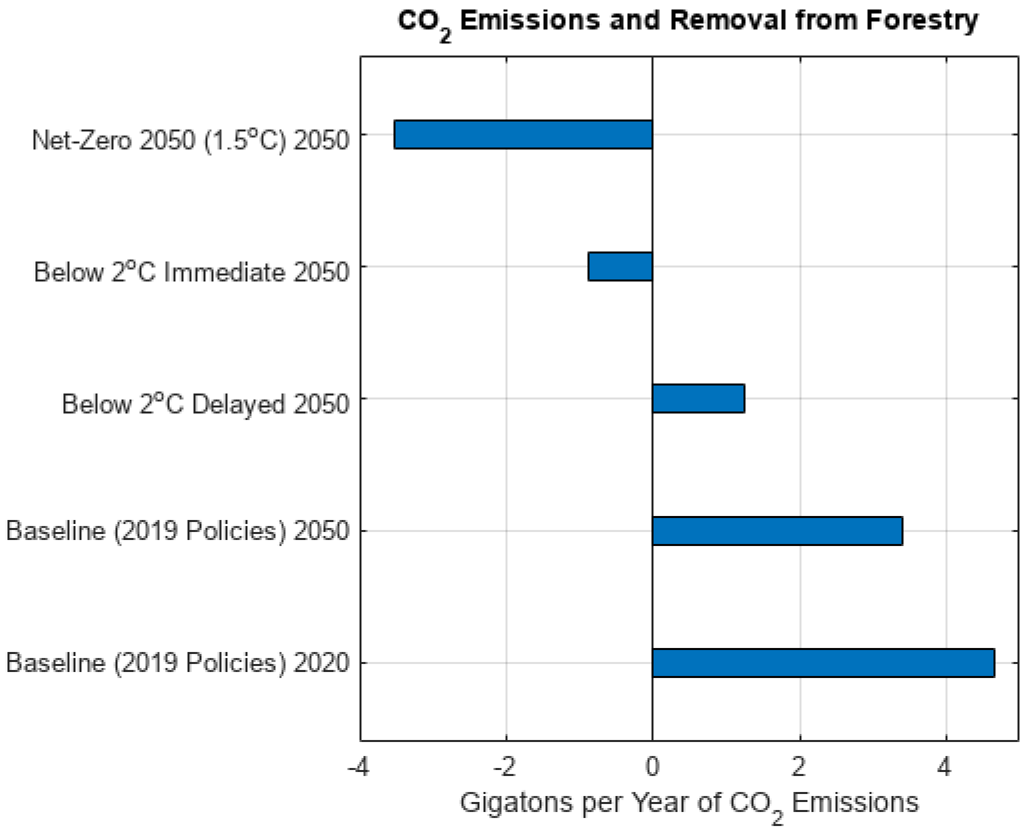
As expected, the global GHG emissions for the *Baseline (2019 Policies)* are increasing. The GHG emissions for the *Below 2°C Delayed* scenario are increasing in the same manner until 2030, when a reduction in carbon dioxde emmissions begins, and then shows a sharp reduction in emmisions until 2050. The two other scenarios, *Below 2°C Immediate* and *Net-Zero 2050 (1.5°C)* show sharper decreases as the actions for emission mitigation start on 2020. The *Net-Zero 2050 (1.5°C)* scenario shows a faster decrease in carbon dioxied emissions because the target is an average decrease in global temperature of abour 1.5 degrees by 2050. This target is a more aggressive assumption than the *Below 2°C Delayed* scenario, which aims to acheive an average decrease in global temperature of about 2 degrees by 2100.

Forestry Carbon Dioxide Emissions and Removal

The global forests, a nature-based solution to cardon dioxide emissions, have a modest but not negligible contribution to the climate scenarios. Observe the impact of removing carbon dioxide emissions from forestry.

```
% C02 emmissions and removal from forestry
baseline_C02FOR = options.baseline(ismember(options.baseline.CL_VARIABLE,'Emissions/removals fro
b2delayed_C02FOR = options.b2delayed(ismember(options.b2delayed.CL_VARIABLE,'Emissions/removals 
b2immediate_C02FOR = options.b2immediate(ismember(options.b2immediate.CL_VARIABLE,'Emissions/rem
netzero2050_C02FOR = options.netzero2050(ismember(options.netzero2050.CL_VARIABLE,'Emissions/rem

% Global C02 emmissions and removal from forestry
baseline_C02FOR_GLOBAL = sortrows(baseline_C02FOR(ismember(baseline_C02FOR.CL_GEOGRAPHY,'Global'
b2delayed_C02FOR_GLOBAL = sortrows(b2delayed_C02FOR(ismember(b2delayed_C02FOR.CL_GEOGRAPHY,'Glob
b2immediate_C02FOR_GLOBAL = sortrows(b2immediate_C02FOR(ismember(b2immediate_C02FOR.CL_GEOGRAPHY
```

```
netzero2050_C02FOR_GLOBAL = sortrows(netzero2050_C02FOR(ismember(netzero2050_C02FOR.CL_GEOGRAPHY

figure
x_value = [baseline_C02FOR_GLOBAL.CL_VALUE(1);baseline_C02FOR_GLOBAL.CL_VALUE(end);b2delayed_C02
y_value = categorical(["Baseline (2019 Policies) 2020";strcat(["Baseline (2019 Policies)"; "Belo
b = barh(y_value, x_value*1e-3,0.2,'BaseValue',0);
xlabel ("Gigatons per Year of CO_2 Emissions")
title("CO_2 Emissions and Removal from Forestry")
grid on
```

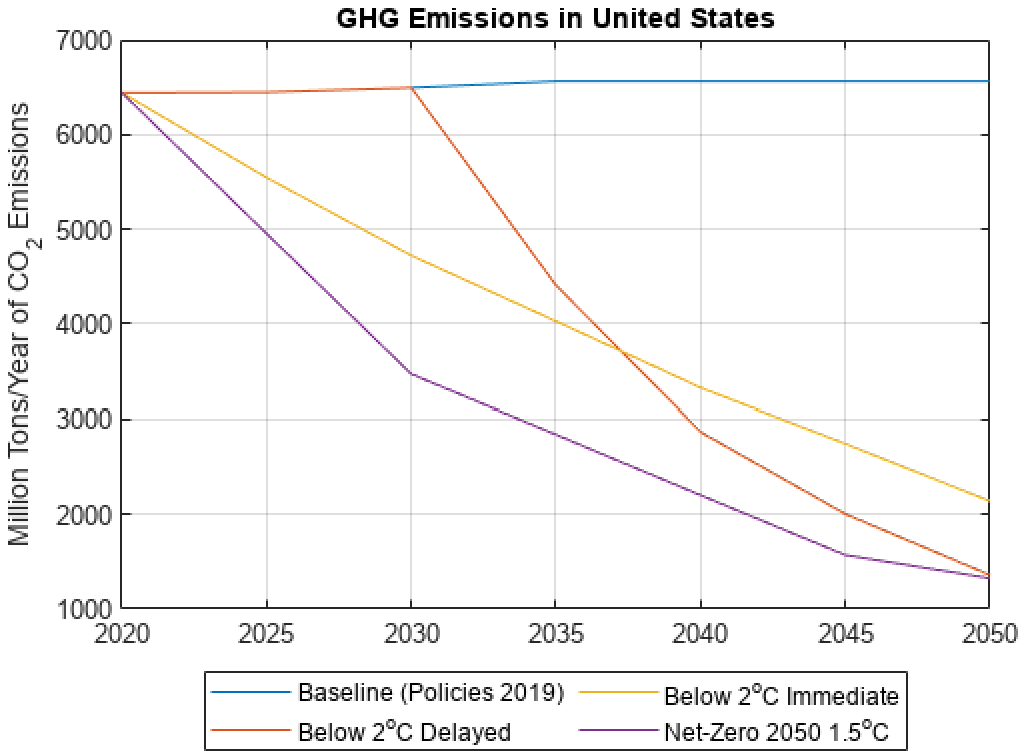


In 2020, global forests were responsible for 4 gigatons of carbon dioxed emissions. By 2050, the carbon dioxide emissions from global forests are projected to decrease for all scenarios. In the projection for the *Below 2°C Immediate* and the *Net-Zero 2050 (1.5°C)* scenarios, the global forests become a natural carbon sink.

Regional Carbon Dioxide Emissions

The following plots present the projected GHG emissions for various regions that you can select using the the dropdown control. Use the plotVariableByCountry on page 4-279 function to plot the GHG emissions for the selected `region`.

```
region = United States ;
options = updateOptions(options,1,strcat("GHG Emissions in ", region),"Million Tons/Year of CO_2
plotVariableByCountry(region,'Emissions | total GHG (scope 1)',{'National','Global'},options);
```



The patterns for Canada and the United States are similar to the patterns for the Global GHG Emissions on page 4-268 plot. As described in [2 on page 4-278], these countries have different climate policies. The climate policies for the United States include:

- Renewable shares in electricity generation
- Corporate Average Fuel Economy standards for both passenger and commercial vehicles

The climate policies for Canada [4 on page 4-279] include:

- Phaseout of traditional coal-fired generation of electricity
- Renewable shares in electricity generation
- Corporate Average Fuel Economy standards for both passenger and commercial vehicles
- Regulations on methane emissions

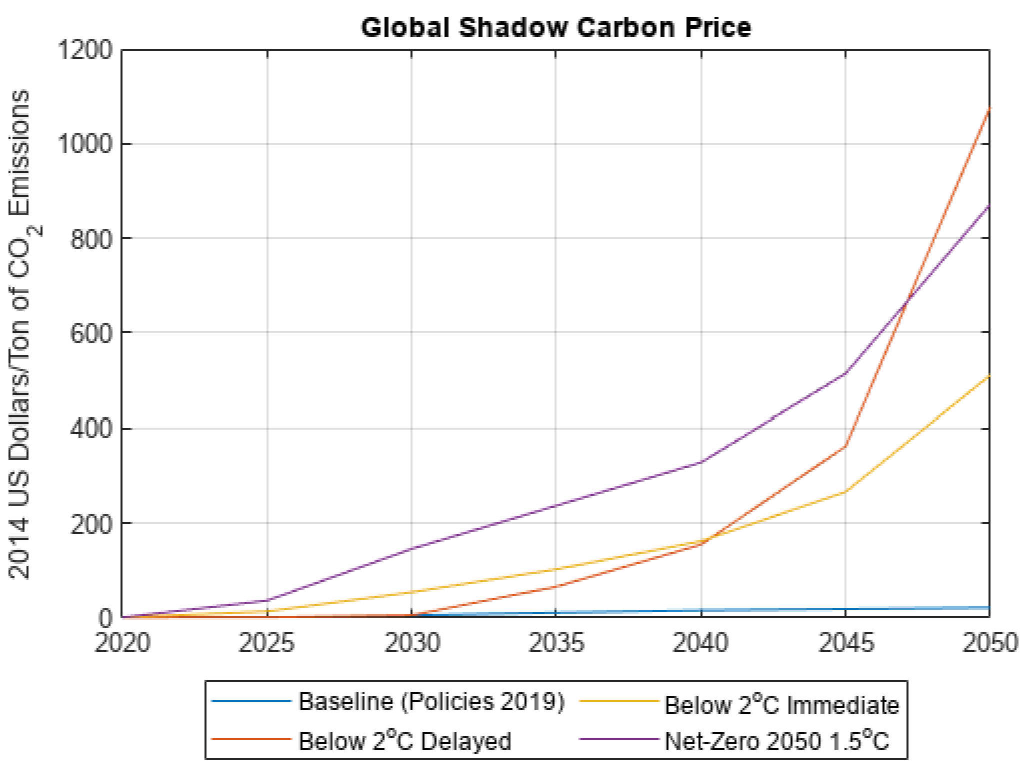
Impact of Climate Policies

Transitioning to low-carbon economies brings significant changes across industries. Industries must move away from fossil-fuel sources to meet their energy demands and adapt noncarbon emitting energy sources like electricity. The plots in this section show the carbon price, GHG emissions, and energy production for the four climate scenarios.

Shadow Carbon Price

To achieve the emission mitigation targets, the model must increase the shadow price of carbon. The *shadow carbon price* applies a theoretical surcharge per ton of carbon emissions. The more aggressive the scenario, the greater the increase of the carbon price. Use the plotVariableByCountry on page 4-279 function to plot the shadow carbon price for the selected region. When `region` is `Global`, the "global" price is the GDP-weighted average across geographies. As described in [2 on page 4-278], the carbon price is an output in this model. The model aims to reduce emissions by a predetermined amount, incorporating noncarbon tax policies first. Then, the model calculates a shadow carbon price to capture the remaining intensity required in government climate policy to meet the emissions targets. The carbon price is modeled as a tax, where the tax revenue is returned to households as lump-sum transfers in the same period.

```
region = Global ;
options = updateOptions(options,1,strcat(region, " Shadow Carbon Price"),"2014 US Dollars/Ton of
plotVariableByCountry(region,'Carbon price',{'Global','National'},options)
```

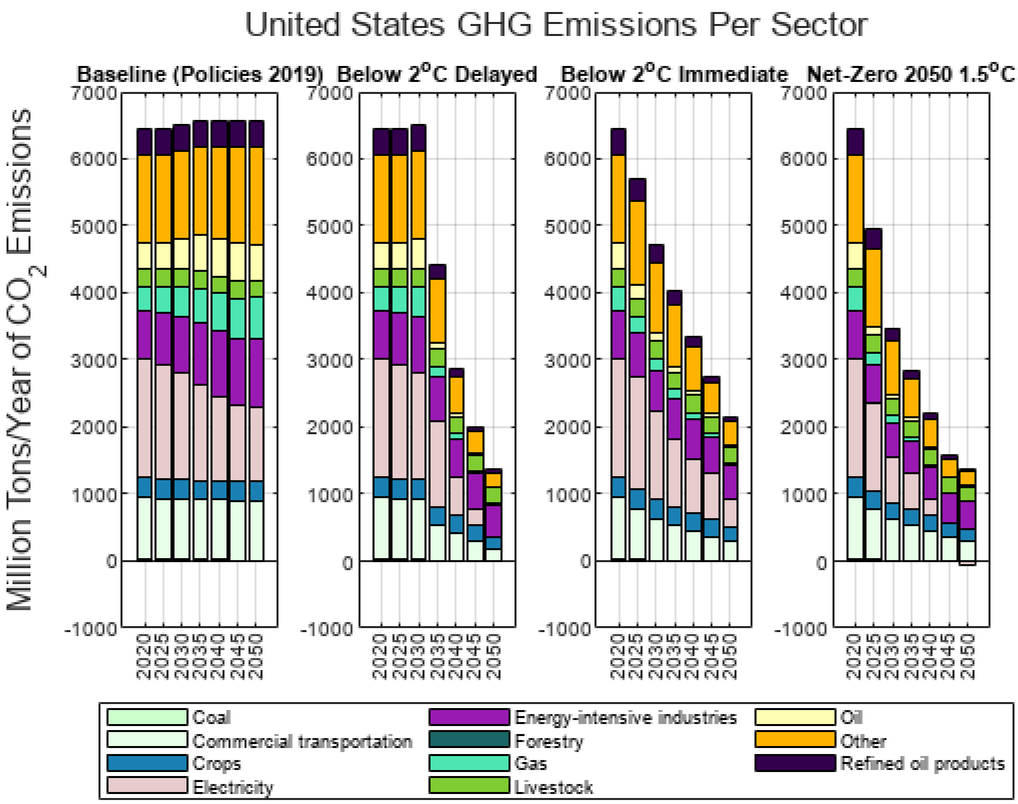


The *Baseline (2019 Policies)* scenario does not reflect any significant change in the carbon shadow price. On the other hand, the shadow carbon price for the *Below 2°C Delayed* scenario shows sharp increases because after 2030 countries have to make up for the lost time in mitigating the gas emissions. The *Below 2°C Immediate* and the *Net-Zero 2050 (1.5°C)* scenarios have a smoother increase of the shadow carbon price because there is time to achieve the goals by 2100 and 2050, respectively. However, the *Net-Zero 2050 (1.5°C)* scenario exhibits faster increases of the shadow carbon price as the target of 2050 is more aggressive.

Regional Sectorial Carbon Dioxide Emissions

Use the plotVariableBySector on page 4-280 function to show bar graphs for the predicted GHG emissions, by sector, for the four climate scenarios. Low-carbon policies lead to changes in all economies and not only the fossil-fuel economies like coal, oil, natural gas, and refined oil.

```
region = United States ;
if (strcmp(region, "Global"))
    options = updateOptions(options,1e-3,strcat(region, " GHG Emissions Per Sector"),"Gigatons/ye
else
    options = updateOptions(options,1,strcat(region, " GHG Emissions Per Sector"),"Million Tons/Y
end
plotVariableBySector(region,{'Emissions | total GHG (scope 1)','Emissions and removals from fores
```



You can see that the contribution of the fossil-fuel sectors to gas emissions significantly lowers by midcentury as the demand for these products goes down.

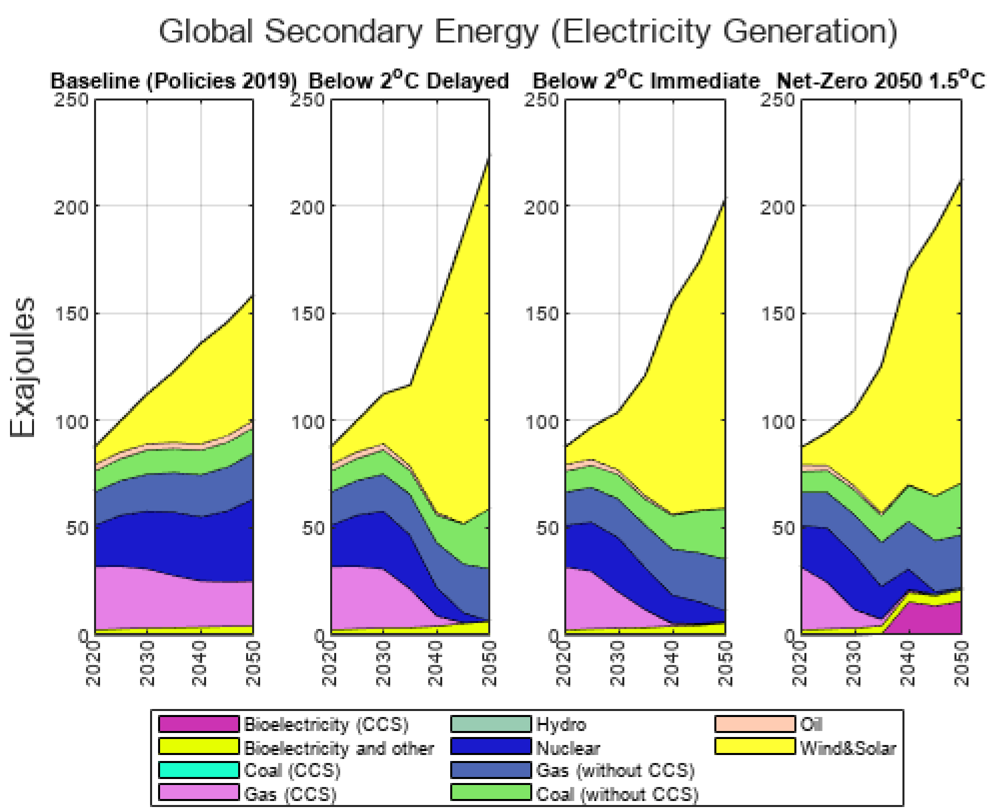
The largest impact in the reduction of gas emissions is achieved by the electricity sector. The electricity sector uses low to zero emission technologies like wind and solar. Also many sectors leverage electricity as a substitute to fossil-fuel products. For example, in the commercial transportation sector, you can see the transition to electric vehicles.

For Canada and the United States, the patterns are similar to the Global GHG Emissions on page 4-268 plot, but there are some key differences. In Canada, the electricity sector completely substitutes the fossil-fuel products, achieving negative gas emissions by midcentury, while leveraging sophisticated technologies. In the United States, the forest carbon sequestration plays a major role in the carbon budget.

Sectorial Energy Production

Use the plotEnergyBySector on page 4-281 function to visualize the use of primary and secondary energy, globally, by sector. Primary energy refers to the amount of energy that a sector delivers and secondary energy refers to the amount of electrical energy each sector generates.

```
options.energytype= secondary|electricity ;
if strcmp(options.energytype,"primary")
    options = updateOptions(options,1e-3,"Global Primary Energy","Exajoules");
else
    %1TWh = 0.0036 EJ
    options = updateOptions(options,36*1e-4,"Global Secondary Energy (Electricity Generation)","
end
plotEnergyBySector('Global',{'Global','Electricity'},options)
```

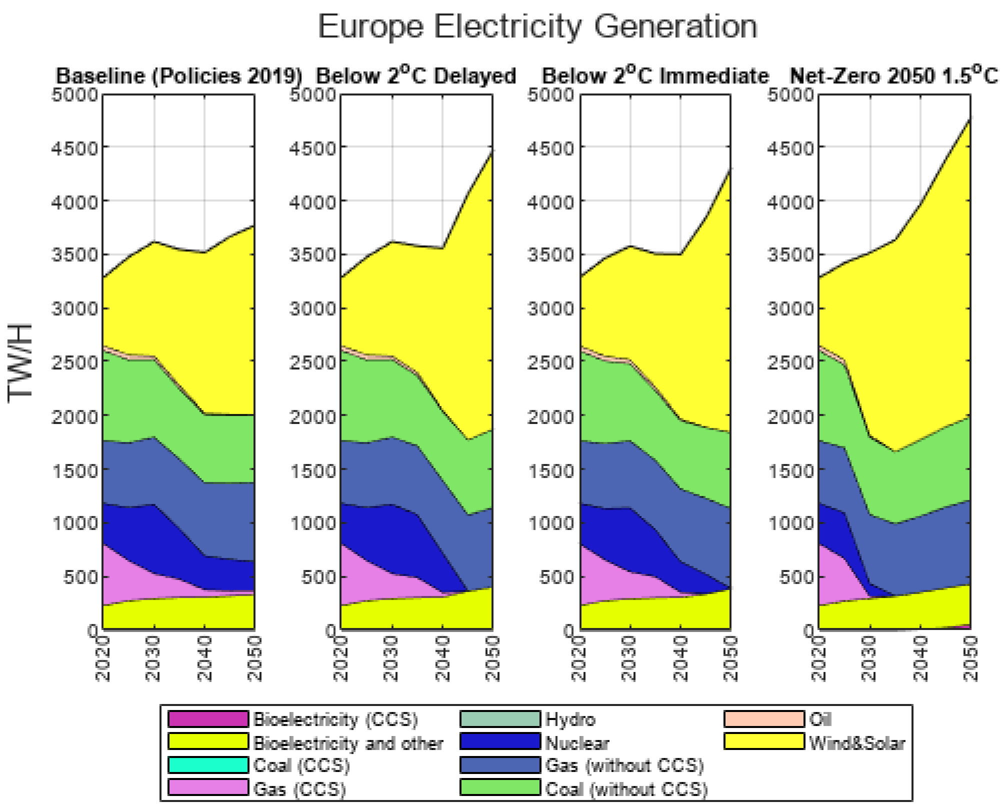


Globally, the dominant energy type is fossil fuels. However, by midcentury renewable energies become the dominant energy types. Electrification supports decarbonization in many sectors. The production sectors are moving away from fossil-fuel products and adapting electrification, contributing to the increased generation of electricity.

Regional Electricity Generation

Use the plotEnergyBySector on page 4-281 function to show bar graphs for electricity generation by region.

```
region = Europe ;
options = updateOptions(options, 1,strcat(region," Electricity Generation"), "TW/H",'secondary')
plotEnergyBySector(region,'Electricity',options)
```



Regional Net Component Changes for Electricity Sector

When a sector is not efficient, the direct emission costs increase because the sector produces more emissions, which also increases indirect costs. Similarly, the revenue decreases because the sector is not very efficient and the capital expenditures are projected to be low.

Therefore, the net income of a company is computed as:

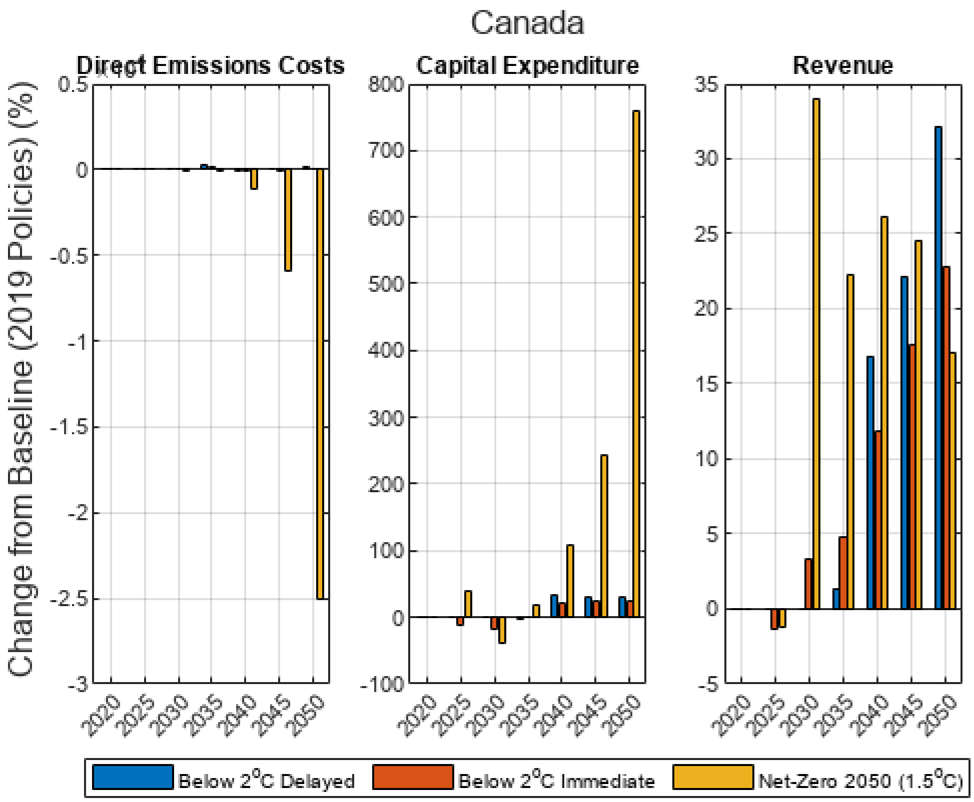
Net Income = Revenues - Direct Emission Costs - Indirect Costs - Capital Expenditures [2 on page 4-278]

- *Revenues = (output price * production)*
- *Direct Emissions costs = (carbon price * scope1 emissions),* where direct emissions cost refers to the increase in a sector's cost associated with the release of greenhouse gases from burning fossil fuels
- *Indirect Costs = (input price * inputs in production),* where indirect cost refers to the direct emission cost of upstream sectors that is passed to the sector.

- *Capital Expenditures = (capital price * new capital added)*, where capital expenditures refers to the cost of investing in new technologies so the sector can become more efficient

Use the plotNetComponents on page 4-283 function to show the bar graphs for the evolution of the net components for the electricity sector in a selected region.

```
region = Canada ;
options = updateOptions(options, 1,region, " Change from Baseline (2019 Policies) (%)");
plotNetComponents(region,'Electricity',options)
```



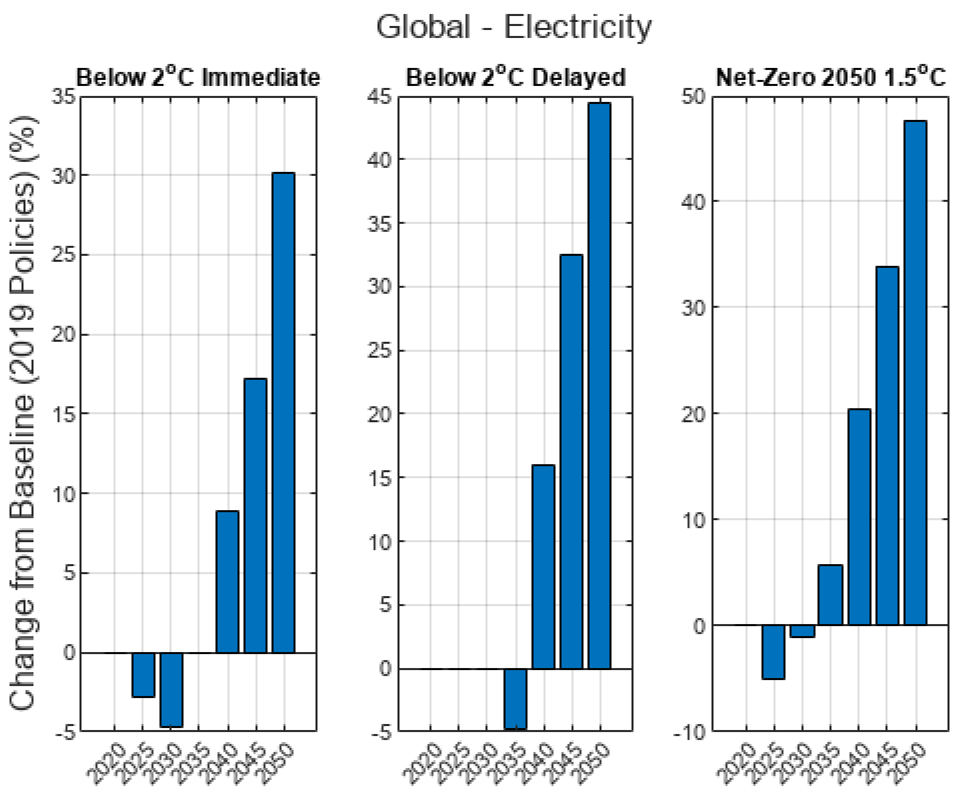
For Canada and the United States the direct emission costs are dropping as the policies become stricter and the carbon price increases. To maintain efficiency, the *Capital Expenditures* also rise. You can see a large impact in both components for the *Net-Zero 2050 (1.5°C)* scenario where energy technology shows a rapid evolution. However, this change results in lower revenue than the *Below 2°C Immediate* and *Below 2°C Delayed* scenarios where the technology evolution remains steady.

Regional and Sectorial Net Component Changes

Use the plotNetIncomeBySector on page 4-283 function to generate the bar graphs for net component changes by sector and region.

```
sector = Electricity ;
region = Global ;
```

```
options = updateOptions(options, 1,strcat(region, " - " ,sector), "Change from Baseline (2019 Po
plotNetIncomeBySector(region,sector,options)
```

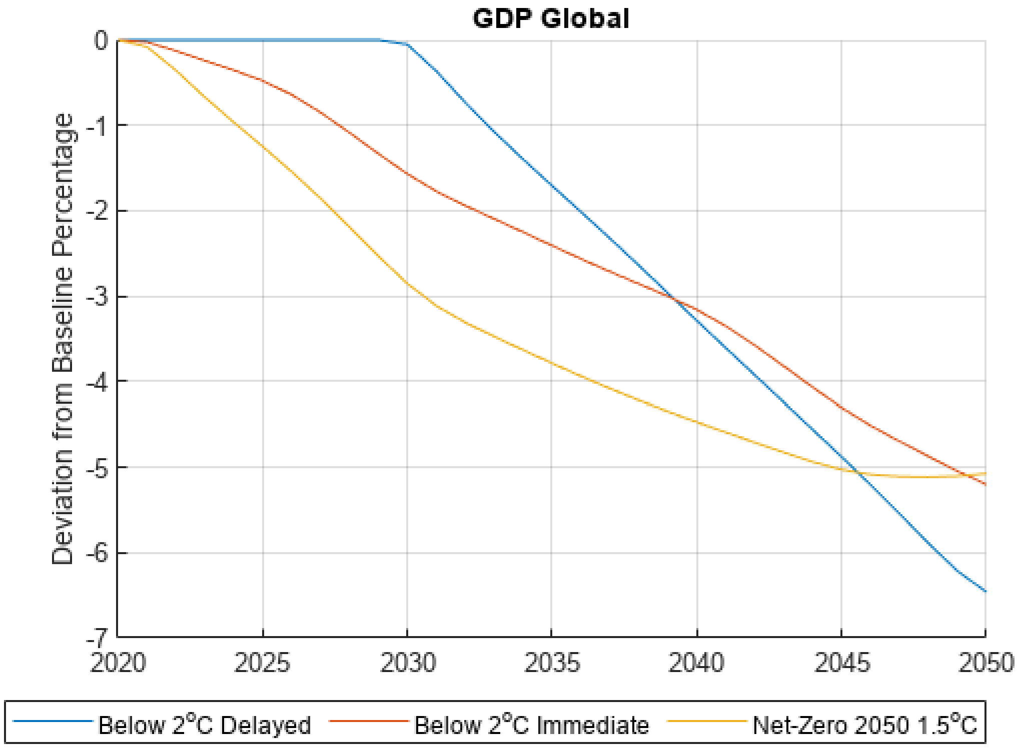


The fossil-fuel sectors present negative net income change as the demand for their products become less. Electricity, on the other hand, presents a postive change in net income as this becomes the dominant energy source in the future.

GDP Impact

Use the plotVariableByCountry on page 4-279 function to generate a plot that shows the GDP change across all three climate scenarios compared with the *Baseline (2019 Policies)* scenario.

```
region = Global ;
options = updateOptions(options, 1,strcat("GDP ", region), "Deviation from Baseline Percentage")
plotVariableByCountry(region,'GDP', [], options);
```



The regional and global GDP impacts show similar patterns. Delayed policy action results in sharper impacts to the GDP by 2050 (see *Below 2°C Delayed*), while immediate policy action results in more gradual decrease of the GDP (see *Below 2°C Immediate* and *Net-Zero 2050 (1.5°C)*).

Summary

The climate scenarios illustrate the important sectoral restructuring that the global economies need to undertake to meet climate targets. The climate scenarios show that every sector contributes to the transition and that the financial impacts vary across sectors. These impacts depend on how emissions and capital expenditures affect the sectors and on how the decarbonization of economies affects product demand. The four climate scenarios also highlight the risks of significant macroeconomic impacts, in particular for commodity-exporting countries like Canada. The economic impacts for Canada are driven mostly by declines in global prices of commodities rather than by domestic policy decisions. Finally, the climate scenarios show that delaying climate policy action increases the overall economic impacts and risks to financial stability.

References

[1] https://www.bankofcanada.ca/2022/01/climate-transition-scenario-data

[2] "Transition Scenarios for Analyzing Climate-Related Financial Risks" available at: https://www.bankofcanada.ca/wp-content/uploads/2021/11/sdp2022-1.pdf.

[3] EPPA Model Structure available at https://globalchange.mit.edu/research/research-tools/eppa and https://globalchange.mit.edu/research/research-tools/human-system-model.

[4] "Government of Canada's Pan-Canadian Framework on Clean Growth and Climate Change" available at https://www.canada.ca/en/services/environment/weather/climatechange/pan-canadian-framework.html.

Local Functions

plotVariableByCountry

```
function plotVariableByCountry(country,variable,sector,options)

GDPflag = false; % Flag that the function is going to be used for GDP plots
if strcmp(variable,'GDP')
    GDPflag = true;
    if strcmp(country,'Global')
        variable = "Global GDP";
        sector = "Global";
    elseif strcmp(country,'Canada')
        variable = "Real GDP";
        sector = "National";
    else
        variable = "US GDP";
        sector = "National";
    end
end


% C02 emmissions removal from forestry
baseline_sc = options.baseline(ismember(options.baseline.CL_VARIABLE,variable),:);
b2delayed_sc = options.b2delayed(ismember(options.b2delayed.CL_VARIABLE,variable),:);
b2immediate_sc = options.b2immediate(ismember(options.b2immediate.CL_VARIABLE,variable),:);
netzero2050_sc = options.netzero2050(ismember(options.netzero2050.CL_VARIABLE,variable),:);

% Global C02 emmissions removal from forestry
baseline_sc_co = sortrows(baseline_sc((ismember(baseline_sc.CL_GEOGRAPHY,country) & ismember(base
b2delayed_sc_co = sortrows(b2delayed_sc((ismember(b2delayed_sc.CL_GEOGRAPHY,country) & ismember(b
b2immediate_sc_co = sortrows(b2immediate_sc((ismember(b2immediate_sc.CL_GEOGRAPHY,country) &ismem
netzero2050_sc_co = sortrows(netzero2050_sc((ismember(netzero2050_sc.CL_GEOGRAPHY,country) & isme

figure
if ~GDPflag, plot(baseline_sc_co.CL_YEAR,baseline_sc_co.CL_VALUE.*options.factor), end
hold on
plot(b2delayed_sc_co.CL_YEAR,b2delayed_sc_co.CL_VALUE.*options.factor)
plot(b2immediate_sc_co.CL_YEAR, b2immediate_sc_co.CL_VALUE.*options.factor)
plot(netzero2050_sc_co.CL_YEAR,netzero2050_sc_co.CL_VALUE.*options.factor)
hold off
if ~GDPflag
    legend('Baseline (Policies 2019)','Below 2^oC Delayed','Below 2^oC Immediate','Net-Zero 2050
else
    legend('Below 2^oC Delayed','Below 2^oC Immediate','Net-Zero 2050 1.5^oC','Location','southou
end
ylabel(options.yLabel)
title(options.title)
grid on
```

```
end
```

plotVariableBySector

```
function plotVariableBySector(country,variable,options)

baseline_sc = options.baseline(ismember(options.baseline.CL_VARIABLE,variable),:);
b2delayed_sc = options.b2delayed(ismember(options.b2delayed.CL_VARIABLE,variable),:);
b2immediate_sc = options.b2immediate(ismember(options.b2immediate.CL_VARIABLE,variable),:);
netzero2050_sc = options.netzero2050(ismember(options.netzero2050.CL_VARIABLE,variable),:);

baseline_sc_co = sortrows(baseline_sc(ismember(baseline_sc.CL_GEOGRAPHY,country),:),"CL_YEAR","as
b2delayed_sc_co = sortrows(b2delayed_sc(ismember(b2delayed_sc.CL_GEOGRAPHY,country),:),"CL_YEAR",
b2immediate_sc_co = sortrows(b2immediate_sc(ismember(b2immediate_sc.CL_GEOGRAPHY,country),:),"CL_
netzero2050_sc_co = sortrows(netzero2050_sc(ismember(netzero2050_sc.CL_GEOGRAPHY,country),:),"CL_

YRS = unique(baseline_sc_co.CL_YEAR);
SEC = unique(baseline_sc_co.CL_SECTOR); SEC(SEC=="Oil & Gas") = []; SEC(SEC=="Global") = []; SEC(

scenariosTitles = {'Baseline (Policies 2019)','Below 2^oC Delayed','Below 2^oC Immediate','Net-Ze

% Perform initiatilization
for ii = 1:length(YRS)
    for jj = 1:length(SEC)
        idx = jj+((ii-1)*length(SEC));
        fun = @max;
        max_baseline_sc_co(idx,:) = varfun(fun,baseline_sc_co((ismember(baseline_sc_co.CL_YEAR,YR
        max_b2delayed_sc_co(idx,:) = varfun(fun,b2delayed_sc_co((ismember(b2delayed_sc_co.CL_YEAR
        max_b2immediate_sc_co(idx,:) = varfun(fun,b2immediate_sc_co((ismember(b2immediate_sc_co.C
        max_netzero2050_sc_co(idx,:) = varfun(fun,netzero2050_sc_co((ismember(netzero2050_sc_co.C
    end
end

T1 = [];
T2 = [];
T3 = [];
T4 = [];

% Prepare data to stack
for ii = 1:length(YRS)
    T1 = [T1; table2array(max_baseline_sc_co(ismember(max_baseline_sc_co.max_CL_YEAR,YRS(ii)),"ma
    T2 = [T2; table2array(max_b2delayed_sc_co(ismember(max_b2delayed_sc_co.max_CL_YEAR,YRS(ii)),"
    T3 = [T3; table2array(max_b2immediate_sc_co(ismember(max_b2immediate_sc_co.max_CL_YEAR,YRS(ii
    T4 = [T4; table2array(max_netzero2050_sc_co(ismember(max_netzero2050_sc_co.max_CL_YEAR,YRS(ii

end

% Create colorSet
rng(0)
colorSet = round(rand(length(SEC),3),1);

figure
t = tiledlayout(1,4);
t.Title.String = options.title;
t.YLabel.String = options.yLabel;
colororder(colorSet)
```

```
ax1 = nexttile;
bar(ax1,YRS,T1,'stacked');
xtickangle(90)
grid on
title(scenariosTitles(1))


ax2 = nexttile;
bar(ax2,YRS,T2,'stacked');
xtickangle(90)
grid on
title(scenariosTitles(2))

ax3 = nexttile;
bar(ax3,YRS,T3,'stacked');
xtickangle(90)
grid on
title(scenariosTitles(3))

ax4 = nexttile;
bar(ax4,YRS,T4,'stacked')
xtickangle(90)
grid on
title(scenariosTitles(4))

linkaxes([ax1 ax2 ax3 ax4],'xy')

lg = legend(SEC,'Location','southoutside','NumColumns',3);
lg.Layout.Tile = 'South'; % Legend placement with tiled layout

end
```

plotEnergyBySector

```
function plotEnergyBySector(country,sector,options)

Prim =  {'Primary Energy | Bioenergy','Primary Energy | Coal','Primary Energy | Gas', 'Primary En
    'Primary Energy | Nuclear','Primary Energy | Oil','Primary Energy | Renewables (wind&solar)'}
Sec =  {'Secondary Energy | Electricity| Bioelectricity (CCS)', ...
    'Secondary Energy | Electricity| Bioelectricity and other', ...
    'Secondary Energy | Electricity| Coal (CCS)', ...
    'Secondary Energy | Electricity| Gas (CCS)', ...
    'Secondary Energy | Electricity| Hydro', ...
    'Secondary Energy | Electricity| Nuclear', ...
    'Secondary Energy | Electricity| Gas (without CCS)', ...
    'Secondary Energy | Electricity| Coal (without CCS)', ...
    'Secondary Energy | Electricity| Oil', ...
    'Secondary Energy | Electricity| Wind&Solar'};
if strcmp(options.energytype,'primary')
    EnergyVars = Prim;
else
    EnergyVars = Sec;
end


baseline_co = sortrows(options.baseline((ismember(options.baseline.CL_GEOGRAPHY,country) & ismemb
b2delayed_co = sortrows(options.b2delayed((ismember(options.b2delayed.CL_GEOGRAPHY,country) & ism
b2immediate_co = sortrows(options.b2immediate((ismember(options.b2immediate.CL_GEOGRAPHY,country)
```

```
netzero2050_co = sortrows(options.netzero2050((ismember(options.netzero2050.CL_GEOGRAPHY,country)

scenariosTitles = {'Baseline (Policies 2019)','Below 2^oC Delayed','Below 2^oC Immediate','Net-Ze


YRS = unique(baseline_co.CL_YEAR);
VARS = unique(baseline_co.CL_VARIABLE);

T1 = [];
T2 = [];
T3 = [];
T4 = [];


% Prepare data to stack
for ii = 1:length(YRS)
    T1 = [T1; table2array(baseline_co(ismember(baseline_co.CL_YEAR,YRS(ii)),"CL_VALUE"))'.*optior
    T2 = [T2; table2array(b2delayed_co(ismember(b2delayed_co.CL_YEAR,YRS(ii)),"CL_VALUE"))'.*opt:
    T3 = [T3; table2array(b2immediate_co(ismember(b2immediate_co.CL_YEAR,YRS(ii)),"CL_VALUE"))'.*
    T4 = [T4; table2array(netzero2050_co(ismember(netzero2050_co.CL_YEAR,YRS(ii)),"CL_VALUE"))'.*

end

rng(0)
colorSet = round(rand(length(VARS),3),1);

figure
t = tiledlayout(1,4);
t.Title.String = options.title;
t.YLabel.String = options.yLabel;
colororder(colorSet)

ax1 = nexttile;
area(YRS,T1);
grid on
xtickangle(90)
title(scenariosTitles(1))


ax2 = nexttile;
area(YRS,T2);
grid on
xtickangle(90)
title(scenariosTitles(2))

ax3 = nexttile;
area(YRS,T3);
xtickangle(90)
grid on
title(scenariosTitles(3))


ax4 = nexttile;
area(YRS,T4)
xtickangle(90)
grid on
title(scenariosTitles(4))
```

```
if strcmp(options.energytype,'primary')
    newVars = strrep(EnergyVars,'Primary Energy | ','');
else
    newVars = strrep(EnergyVars,'Secondary Energy | Electricity| ','');
end
lg  = legend(newVars,'Location','southoutside','NumColumns',3) ;
lg.Layout.Tile = 'South'; % Legend placement with tiled layout

linkaxes([ax1 ax2 ax3 ax4],'xy')
xtickangle(90)

end
```

plotNetComponents

```
function plotNetComponents(country,sector,options)

NetVars =  {'Direct emissions costs', ...
    'Capital expenditure', ...
    'Revenue'};
NetVarsTitles = {'Direct Emissions Costs', 'Capital Expenditure', 'Revenue'};

figure
t = tiledlayout(1,length(NetVars));
t.Title.String = options.title;
t.YLabel.String = options.yLabel;

for ii = 1:length(NetVars)
    baseline_co = sortrows(options.baseline((ismember(options.baseline.CL_GEOGRAPHY,country) & is
    b2delayed_co = sortrows(options.b2delayed((ismember(options.b2delayed.CL_GEOGRAPHY,country) &
    b2immediate_co = sortrows(options.b2immediate((ismember(options.b2immediate.CL_GEOGRAPHY,coun
    netzero2050_co = sortrows(options.netzero2050((ismember(options.netzero2050.CL_GEOGRAPHY,coun

    nexttile(ii)
    bar(unique(baseline_co.CL_YEAR),[(b2delayed_co.CL_VALUE-baseline_co.CL_VALUE)./baseline_co.CL
        (b2immediate_co.CL_VALUE-baseline_co.CL_VALUE)./baseline_co.CL_VALUE,...
        (netzero2050_co.CL_VALUE-baseline_co.CL_VALUE)./baseline_co.CL_VALUE].*100);
    grid on


    title(NetVarsTitles(ii))

end
lg  =  legend('Below 2^oC Delayed','Below 2^oC Immediate','Net-Zero 2050 (1.5^oC)','Location','so
lg.Layout.Tile = 'South'; % Legend placement with tiled layout
end
```

plotNetIncomeBySector

```
function plotNetIncomeBySector(country,sector, options)

NetVars =  {'Capital expenditure', ...
    'Direct emissions costs', ...
    'Revenue', ...
    'Indirect costs'};

figure
t = tiledlayout(1,3);
```

```
t.Title.String = options.title;
t.YLabel.String = options.yLabel;

% Capital expenditure
baseline_CE = sortrows(options.baseline((ismember(options.baseline.CL_GEOGRAPHY,country) & ismemb
b2delayed_CE = sortrows(options.b2delayed((ismember(options.b2delayed.CL_GEOGRAPHY,country) & ism
b2immediate_CE = sortrows(options.b2immediate((ismember(options.b2immediate.CL_GEOGRAPHY,country)
netzero2050_CE = sortrows(options.netzero2050((ismember(options.netzero2050.CL_GEOGRAPHY,country)


% Direct emissions costs
baseline_DEC = sortrows(options.baseline((ismember(options.baseline.CL_GEOGRAPHY,country) & ismem
b2delayed_DEC  = sortrows(options.b2delayed((ismember(options.b2delayed.CL_GEOGRAPHY,country) & i
b2immediate_DEC  = sortrows(options.b2immediate((ismember(options.b2immediate.CL_GEOGRAPHY,countr
netzero2050_DEC  = sortrows(options.netzero2050((ismember(options.netzero2050.CL_GEOGRAPHY,countr


% Revenue
baseline_RE = sortrows(options.baseline((ismember(options.baseline.CL_GEOGRAPHY,country) & ismemb
b2delayed_RE = sortrows(options.b2delayed((ismember(options.b2delayed.CL_GEOGRAPHY,country) & ism
b2immediate_RE = sortrows(options.b2immediate((ismember(options.b2immediate.CL_GEOGRAPHY,country)
netzero2050_RE = sortrows(options.netzero2050((ismember(options.netzero2050.CL_GEOGRAPHY,country)


% Indirect costs
baseline_IC = sortrows(options.baseline((ismember(options.baseline.CL_GEOGRAPHY,country) & ismemb
b2delayed_IC = sortrows(options.b2delayed((ismember(options.b2delayed.CL_GEOGRAPHY,country) & ism
b2immediate_IC = sortrows(options.b2immediate((ismember(options.b2immediate.CL_GEOGRAPHY,country)
netzero2050_IC = sortrows(options.netzero2050((ismember(options.netzero2050.CL_GEOGRAPHY,country)

if strcmp(sector,"Electricity")
    baseline_NI = baseline_RE.CL_VALUE -baseline_DEC.CL_VALUE  - baseline_CE.CL_VALUE;
    b2delayed_NI = b2delayed_RE.CL_VALUE -b2delayed_DEC.CL_VALUE  - b2delayed_CE.CL_VALUE;
    b2immediate_NI = b2immediate_RE.CL_VALUE -b2immediate_DEC.CL_VALUE  - b2immediate_CE.CL_VALUE
    netzero2050_NI = netzero2050_RE.CL_VALUE -netzero2050_DEC.CL_VALUE  - netzero2050_CE.CL_VALUE
else
    % Net incomes
    baseline_NI = baseline_RE.CL_VALUE -baseline_DEC.CL_VALUE - baseline_IC.CL_VALUE - baseline_C
    b2delayed_NI = b2delayed_RE.CL_VALUE -b2delayed_DEC.CL_VALUE - b2delayed_IC.CL_VALUE - b2dela
    b2immediate_NI = b2immediate_RE.CL_VALUE -b2immediate_DEC.CL_VALUE - b2immediate_IC.CL_VALUE
    netzero2050_NI = netzero2050_RE.CL_VALUE -netzero2050_DEC.CL_VALUE - netzero2050_IC.CL_VALUE
end


nexttile;
bar(unique(baseline_RE.CL_YEAR),((b2immediate_NI-baseline_NI)./baseline_NI) .*100);
title('Below 2^oC Immediate')
grid on

nexttile;
bar(unique(baseline_RE.CL_YEAR),((b2delayed_NI-baseline_NI)./baseline_NI) .*100);
title('Below 2^oC Delayed')
grid on

nexttile;
bar(unique(baseline_RE.CL_YEAR),((netzero2050_NI-baseline_NI)./baseline_NI) .*100);
title('Net-Zero 2050 1.5^oC')
grid on
```

```
end
```

preprocessBankOfCanadaData

```
function [ClimateTransitionScenarioData,options] = preprocessBankOfCanadaData(ClimateTransitionSc

VariableSubset = {'Direct emissions costs', ... % Emissions
    'Emission intensity','Emissions (scope 1)| CH4', ...
    'Emissions (scope 1)| CO2', ...
    'Emissions (scope 1)| HFC', ...
    'Emissions (scope 1)| N2O', ...
    'Emissions (scope 1)| PFC', ...
    'Emissions (scope 1)| SF6', ...
    'Emissions (scope 2)| total GHG', ...
    'Emissions | total GHG (scope 1)', ...
    'Emissions/removals from forestry', ...
    'Carbon price' ...  % Shadow carbon price
    'Primarsy Energy | Bioenergy', ...  % Primary energy
    'Primary Energy | Coal', ...
    'Primary Energy | Gas', ...
    'Primary Energy | Hydro', ...
    'Primary Energy | Nuclear', ...
    'Primary Energy | Oil', ...
    'Primary Energy | Renewables (wind&solar)', ...
    'Primary Energy | Total', ...
    'Secondary Energy | Electricity| Bioelectricity (CCS)', ... % Secondary energy for electrici
    'Secondary Energy | Electricity| Bioelectricity and other', ...
    'Secondary Energy | Electricity| Coal (CCS)', ...
    'Secondary Energy | Electricity| Coal (without CCS)', ...
    'Secondary Energy | Electricity| Gas (CCS)', ...
    'Secondary Energy | Electricity| Gas (without CCS)', ...
    'Secondary Energy | Electricity| Hydro', ...
    'Secondary Energy | Electricity| Nuclear', ...
    'Secondary Energy | Electricity| Oil', ...
    'Secondary Energy | Electricity| Wind&Solar', ...
    'Capital expenditure', ...     % Components of net income
    'Direct emissions costs', ...
    'Indirect costs', ...
    'Revenue', ...
    'US GDP', ... % GDPs
    'Global GDP', ...
    'Real GDP'};

% Keep only the specific VARIABLES
ClimateTransitionScenarioData = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioD

% Find unique values of the categories. These are useful for the controls
% that appear later.
regions = string(unique(ClimateTransitionScenarioData.CL_GEOGRAPHY));
sectors = string(unique(ClimateTransitionScenarioData.CL_SECTOR));
vars = string(unique(ClimateTransitionScenarioData.CL_VARIABLE));
SCE = unique(ClimateTransitionScenarioData.CL_SCENARIO);

% Remove table variables from data
ClimateTransitionScenarioData = removevars(ClimateTransitionScenarioData,{'k'});
ClimateTransitionScenarioData = sortrows(ClimateTransitionScenarioData);
```

```
% Pull data by scenario
baseline = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO, 'Bas
b2delayed = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO, 'Be
b2immediate = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO,
netzero2050 = ClimateTransitionScenarioData(ismember(ClimateTransitionScenarioData.CL_SCENARIO,

% Options for local functions
options = struct;
options.regions = regions;
options.sectors = sectors;
options.var = vars;
options.scenarios = SCE;
options.baseline = baseline;
options.b2delayed = b2delayed;
options.b2immediate = b2immediate;
options.netzero2050 = netzero2050;
end
```

addData

```
function options = updateOptions(options,varargin)
% varargin
% 1st: factor
% 2nd: title
% 3rd: ylabel
% 4th: energy type
if nargin<3
    options.factor = varargin{1};
elseif nargin<4
    options.factor = varargin{1};
    options.title = varargin{2};
elseif nargin<5
    options.factor = varargin{1};
    options.title = varargin{2};
    options.yLabel = varargin{3};
else
    options.factor = varargin{1};
    options.title = varargin{2};
    options.yLabel = varargin{3};
    options.energytype  = varargin{4};
end

end
```

See Also

Related Examples

- “Measure Transition Risk for Loan Portfolios with Respect to Climate Scenarios” on page 4-231
- “Assess Physical and Transition Risk for Mortgages” on page 4-248

External Websites

- Modeling the Impact of Transition and Physical Climate Risks on a Portfolio of Mortgages (13 min 52 sec)

Interpretability and Explainability for Credit Scoring

This example shows different techniques for interpreting and explaining the logic behind credit scoring predictions.

While credit scorecard models, in general, are straightforward to interpret, this example uses a black-box model, without revealing the logic, to show the workflow for explaining predictions. In this example, you work with the `creditscorecard` object from Financial Toolbox™ and pass the scoring function to interpretability tools in Statistics and Machine Learning Toolbox™. These tools include:

- Partial dependence plots (PDP) on page 4-290
- Iindividual conditional expectation plots (ICE) on page 4-292
- Local interpretable model-agnostic explanations (LIME) on page 4-294
- Shapley values on page 4-296

These tools support regression and classification modeling, which make interpretation more efficient. For more information on these techniques, see “Interpret Machine Learning Models”. In this example, the score model of the `creditscorecard` object is used as the black-box model. For an example of this workflow, see “Interpret and Stress-Test Deep Learning Networks for Probability of Default” on page 4-178.

Background

Credit scoring is the process by which lenders assign scores to borrowers and use those scores to decide whether or not to accept a loan application. Lenders use credit scoring models to come up with these scores. Traditionally, simple, interpretable models such as credit scorecards and logistic regression have been widely used in this area. Over time, Machine Learning (ML) and Artificial Intelligence (AI) techniques were introduced to implement credit scoring models. Such techniques, while improving predictive power, also are more black-box and there is little or no explanation behind the decisions. Consequently, the credit scoring predictions of ML and AI techniques are difficult for humans to interpret. As a result, lenders are implementing different interpretability and explainability methods to get a better understanding of the logic behind the credit scoring predictions. In addition, regulators are also requiring that practitioners use more interpretability and also fairness methods to ensure that no equal opportunity laws are broken while making credit decisions. For more information on using fairness metrics, see “Explore Fairness Metrics for Credit Scoring Model” on page 3-98.

Create Credit Scorecard Model

Load credit card data and create a credit scorecard model using the `creditscorecard` object.

```
load CreditCardData
sc = creditscorecard(data,IDVar="CustID");
```

Apply automatic binning. This example uses a split algorithm, with a maximum of 5 bins per predictor and with the constraint that each bin has at least 50 observations. For more information, see `autobinning` and “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23.

```
sc = autobinning(sc,Algorithm="Split",AlgorithmOptions={"MaxNumBins",5,"MinCount",50});
```

Verify that the binning for the numeric variables has five bins or levels. For example, here is the bin information for the customer age predictor.

```
bi = bininfo(sc,"CustAge");
disp(bi)
```

```
         Bin           Good    Bad     Odds        WOE       InfoValue 
    ______________     ____    ___    ______    _________    __________

    {'[-Inf,35)'}       93     76     1.2237     -0.50255      0.038003
    {'[35,47)'  }      321    184     1.7446     -0.14791     0.0094258
    {'[47,53)'  }      194     64     3.0312      0.40456       0.03252
    {'[53,61)'  }      128     64          2    -0.011271    2.0365e-05
    {'[61,Inf]' }       67      9     7.4444        1.303      0.079183
    {'Totals'   }      803    397     2.0227          NaN       0.15915
```

For categorical variables, there may be fewer than five bins in total, since the split algorithm in `autobinning` can merge categories into a single group. For example, the residential status predictor initially has three levels: Tenant, Home Owner, and Other. The split algorithm returns only two groups.

```
[bi,cg] = bininfo(sc,"ResStatus");
disp(bi)
```

```
       Bin         Good    Bad     Odds        WOE       InfoValue
    ___________    ____    ___    ______    _________    _________

    {'Group1'}     672     344    1.9535    -0.034802    0.0010314
    {'Group2'}     131      53    2.4717      0.20049    0.0059418
    {'Totals'}     803     397    2.0227          NaN    0.0069732
```

The category grouping information shows that Tentant and Home Owner are merged into `Group1`. This grouping means Tenant and Home Owner will get the same number of points in the final scorecard.

```
disp(cg)
```

```
       Category        BinNumber
    ______________     _________

    {'Tenant'    }         1    
    {'Home Owner'}         1    
    {'Other'     }         2    
```

Fit the model coefficients using `fitmodel`. For illustration purposes, keep only five model predictors, including some categorical ones.

```
PredictorsInModel = ["CustAge" "CustIncome" "EmpStatus" "ResStatus" "UtilRate"];
sc = fitmodel(sc,PredictorVars=PredictorsInModel,VariableSelection="fullmodel",Display="off");
```

Scale the points so that `500` points correspond to odds of `2`, and the odds double every `50` points.

```
sc = formatpoints(sc,PointsOddsAndPDO=[500 2 50]);
```

Display the scorecard using `displaypoints`. The credit scorecard model is a lookup table. For example, for customer age there are five bins or levels with different points for each level. A visualization of the score, as a function of age, has a piecewise constant pattern with five levels, as shown in Partial Dependence Plot on page 4-290. For residential status, Tenant and Home Owner are in `Group1`, and they get the same number of points.

```
[ScorecardPointsTable,MinPts,MaxPts] = displaypoints(sc);
disp(ScorecardPointsTable)
```

```
     Predictors              Bin             Points
    ______________    _________________    ______

    {'CustAge'   }    {'[-Inf,35)'    }     71.84
    {'CustAge'   }    {'[35,47)'      }    91.814
    {'CustAge'   }    {'[47,53)'      }    122.93
    {'CustAge'   }    {'[53,61)'      }    99.511
    {'CustAge'   }    {'[61,Inf]'     }    173.54
    {'CustAge'   }    {'<missing>'    }       NaN
    {'ResStatus' }    {'Group1'       }    97.318
    {'ResStatus' }    {'Group2'       }    116.43
    {'ResStatus' }    {'<missing>'    }       NaN
    {'EmpStatus' }    {'Unknown'      }    85.326
    {'EmpStatus' }    {'Employed'     }    118.11
    {'EmpStatus' }    {'<missing>'    }       NaN
    {'CustIncome'}    {'[-Inf,31000)' }    68.158
    {'CustIncome'}    {'[31000,38000)'}    102.11
    {'CustIncome'}    {'[38000,42000)'}    93.302
    {'CustIncome'}    {'[42000,47000)'}    109.18
    {'CustIncome'}    {'[47000,Inf]'  }    121.21
    {'CustIncome'}    {'<missing>'    }       NaN
    {'UtilRate'  }    {'[-Inf,0.12)'  }    106.84
    {'UtilRate'  }    {'[0.12,0.3)'   }    94.647
    {'UtilRate'  }    {'[0.3,0.39)'   }    140.95
    {'UtilRate'  }    {'[0.39,0.68)'  }    69.635
    {'UtilRate'  }    {'[0.68,Inf]'   }    94.634
    {'UtilRate'  }    {'<missing>'    }       NaN
```

One "traditional" approach to measure the importance of each predictor in the credit scorecard model is to compute the percent of the total score range that comes from each predictor.

```
PtsRange = MaxPts - MinPts;
NumPred = length(PredictorsInModel);
PercentWeight  = zeros(NumPred,1);

for ii = 1 : NumPred
   Ind = strcmpi(PredictorsInModel{ii},ScorecardPointsTable.Predictors);
   MaxPtsPred = max(ScorecardPointsTable.Points(Ind));
   MinPtsPred = min(ScorecardPointsTable.Points(Ind));
   PercentWeight(ii) = 100*(MaxPtsPred-MinPtsPred)/PtsRange;
end

PredictorWeights = table(PredictorsInModel',PercentWeight,VariableNames=["Predictor" "Weight"]);
disp(PredictorWeights)
```

```
     Predictor      Weight
    ____________    ______

    "CustAge"       36.587
    "CustIncome"    19.085
    "EmpStatus"     11.795
    "ResStatus"     6.8768
    "UtilRate"      25.656
```

Customer age is the main variable in the model, since it corresponds to 36% of the total score range. A customer can get anywhere from `71.8` to `173.5` points, based on their age. This range has a difference of over 100 points between the minimum and maximum values. On the other end, residential status plays a minor role in the score, with points ranging from `97.3` to `116.4` only, a difference of less than 20 points.

An alternative to this "traditional" approach is to use the following explainability techniques from Statistics and Machine Learning Toolbox: Partial Dependence Plot on page 4-290, Individual Conditional Expectation Plot on page 4-292, Local Interpretable Model-Agnostic Explanation Plot on page 4-294, and Shapley Values on page 4-296.

Partial Dependence Plot

The partial dependence plot (PDP) shows the effect of one or two variables on the predicted score.

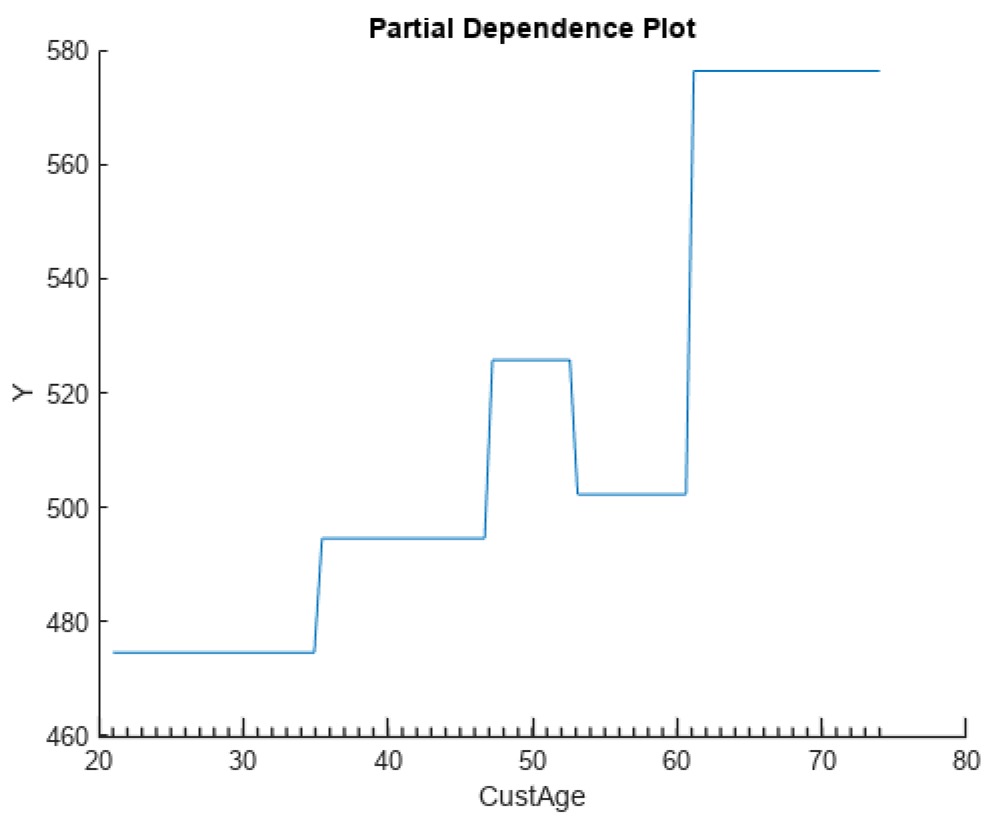
Use the `plotPartialDependence` function to pass the `score` method of the `creditscorecard` object as a black-box model.

One Predictor

Select a predictor using the dropdown option.

As an example, if customer age is selected, note the piecewise constant shape of the plot, with jumps occuring at the bin edges, and with five levels in total. This is consistent with the five bins for customer age in the credit scorecard model.

```
predictor = CustAge;

plotPartialDependence(@(tbl)score(sc,tbl),predictor,data)
```



Two Predictors

Generating a partial dependence plot with two predictors can take significantly longer than the one-predictor case. Typically, the more unique values a predictor has in the data set, the longer it takes to plot the partial dependence. Here's a report of the number of unique values in the data.

```
NumUniqueValuesTable = varfun(@(x)length(unique(x)),data(:,PredictorsInModel));
NumUniqueValuesTable.Properties.VariableNames = erase(NumUniqueValuesTable.Properties.VariableNa
disp(NumUniqueValuesTable)
```

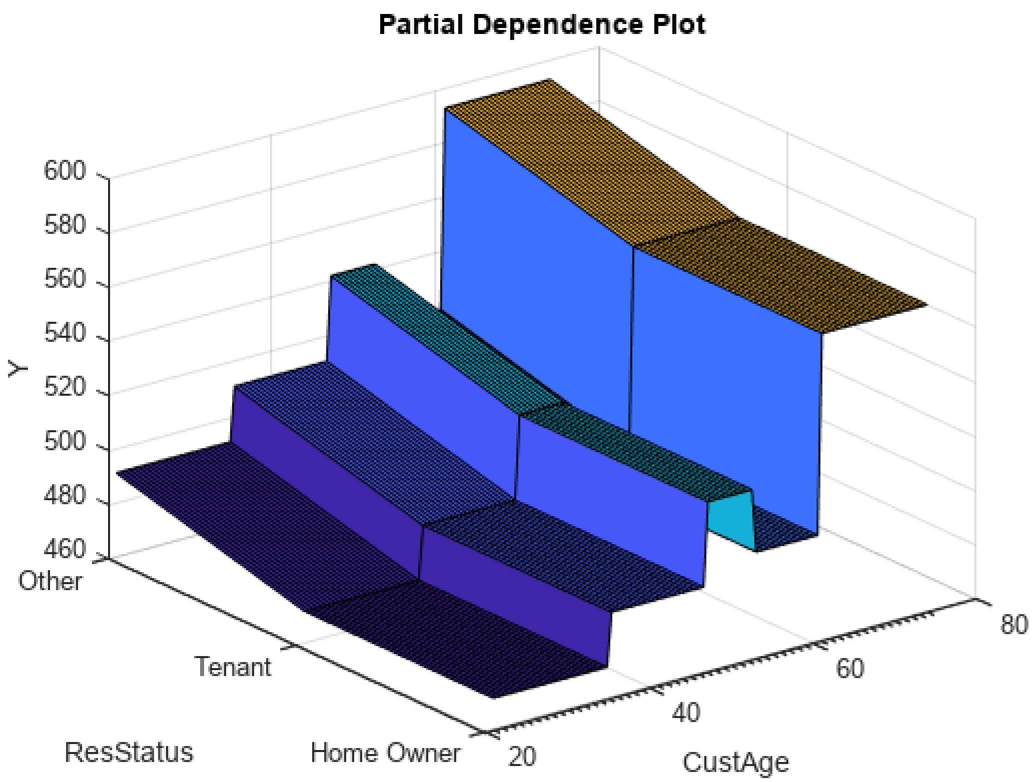
```
    CustAge    CustIncome    EmpStatus    ResStatus    UtilRate
    _______    __________    _________    _________    ________

      54           45            2            3          110
```

The categorical predictors have fewer unique levels, so these plots for categorical predictors run faster. Numeric variables like customer age are relatively discrete and so is utilization rate because this rate's values are rounded to two decimals. However, a continuous predictor (for example, the average monthly balance (`AMBalance`) in the `data` table) can have many unique values.

Select a predictor and additional predictor to then use `plotPartialDependence` to generate the PDP plot.

```
predictor = CustAge;
additionalPredictor = ResStatus;
```

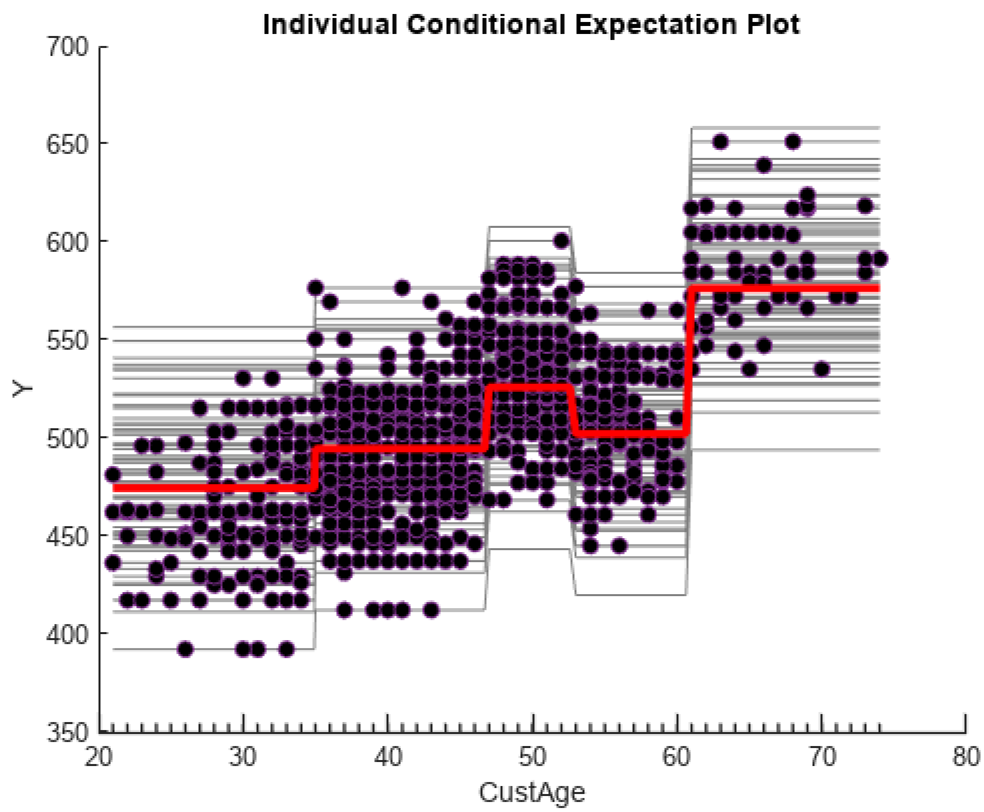
```
plotPartialDependence(@(tbl)score(sc,tbl),[predictor,additionalPredictor],data)
```



Individual Conditional Expectation Plot

Similar to the partial dependence plot, the individual conditional expectation plot (ICE) shows the effect of one of the variables on the predicted score. The red line in the ICE plot matches the partial dependence plot. While the partial dependence plot shows the average score as a function of the selected predictor, the ICE plot disaggregates and shows the score for each observation (each gray line) as a function of the selected predictor. For more information, see the **More About** section on the `plotPartialDependence` reference page.

```
predictor = CustAge;

plotPartialDependence(@(tbl)score(sc,tbl),predictor, ...
    data,Conditional="absolute")
```



Select a Query Point

The PDP and ICE plots provide a global view of the credit scorecard scores, where the score is visualized for all values of the selected predictor. In contrast, LIME and Shapley are local explainability techniques that explain the behavior of the model in a neighborhood of a *query point* of choice. For more information, see "Interpret Machine Learning Models".

To see how a query point helps to explain credit scores , use index 92 in the training data as your query point. You can select other query points by typing an index value into the text box.

```
QueryPointIndex = 92; % ID number of the observation to explain
```

Use `score` to display the query point score and the points, by predictor, for this query point.

```
[ScoresTraining,PointsTraining] = score(sc,data);
fprintf("Selected index %d, with score %g\n",QueryPointIndex,ScoresTraining(QueryPointIndex))
```

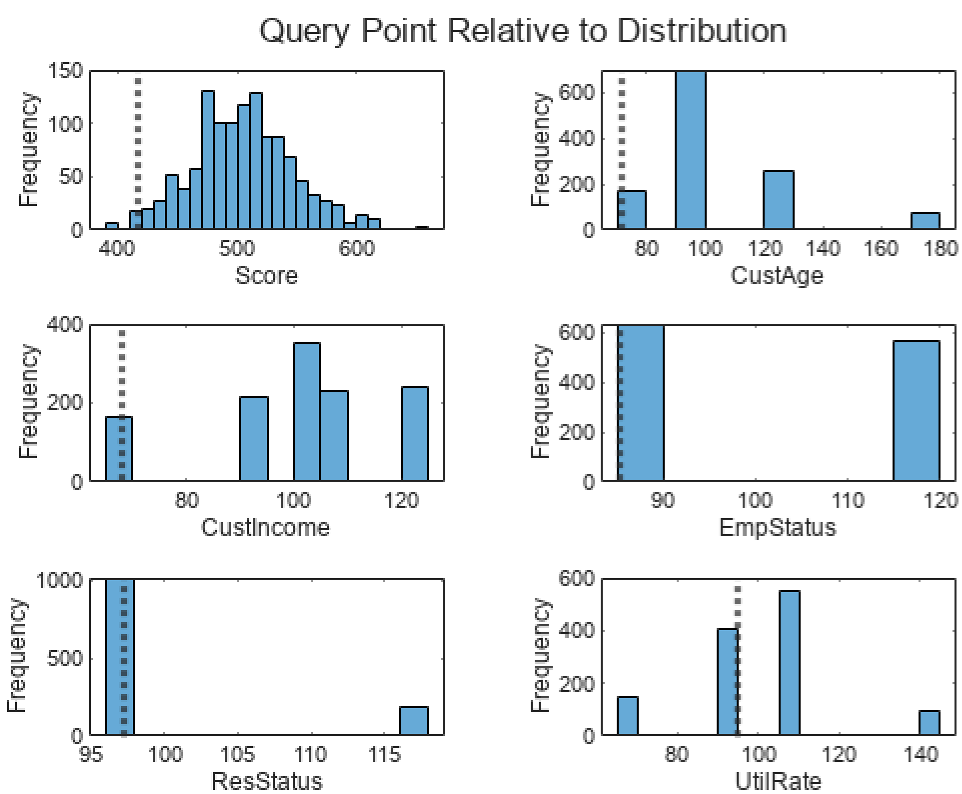
```
Selected index 92, with score 417.289
```

```
disp(PointsTraining(QueryPointIndex,:))
```

```
    CustAge    ResStatus    EmpStatus    CustIncome    UtilRate
    _______    _________    _________    __________    ________

     71.84      97.318       85.326        68.158       94.647
```

The plots that follow show the location of the query point (dotted vertical line) relative to the distribution of values for the scores and for each predictor. For example, for index `92`, the score is low relative to the distribution. For the customer age predictor, the query point is on the bottom group. This result is similar for the customer income, employment status, and residential status predictors. The points for the utilization rate predictor are closer to the middle of the distribution, but still below average.

```
figure
t = tiledlayout(3,2);
nexttile
plotQueryInHistogram("Score",QueryPointIndex,ScoresTraining,PointsTraining)
nexttile
plotQueryInHistogram("CustAge",QueryPointIndex,ScoresTraining,PointsTraining)
nexttile
plotQueryInHistogram("CustIncome",QueryPointIndex,ScoresTraining,PointsTraining)
nexttile
plotQueryInHistogram("EmpStatus",QueryPointIndex,ScoresTraining,PointsTraining)
nexttile
plotQueryInHistogram("ResStatus",QueryPointIndex,ScoresTraining,PointsTraining)
nexttile
plotQueryInHistogram("UtilRate",QueryPointIndex,ScoresTraining,PointsTraining)
title(t,"Query Point Relative to Distribution")
```



Local Interpretable Model-Agnostic Explanation Plot

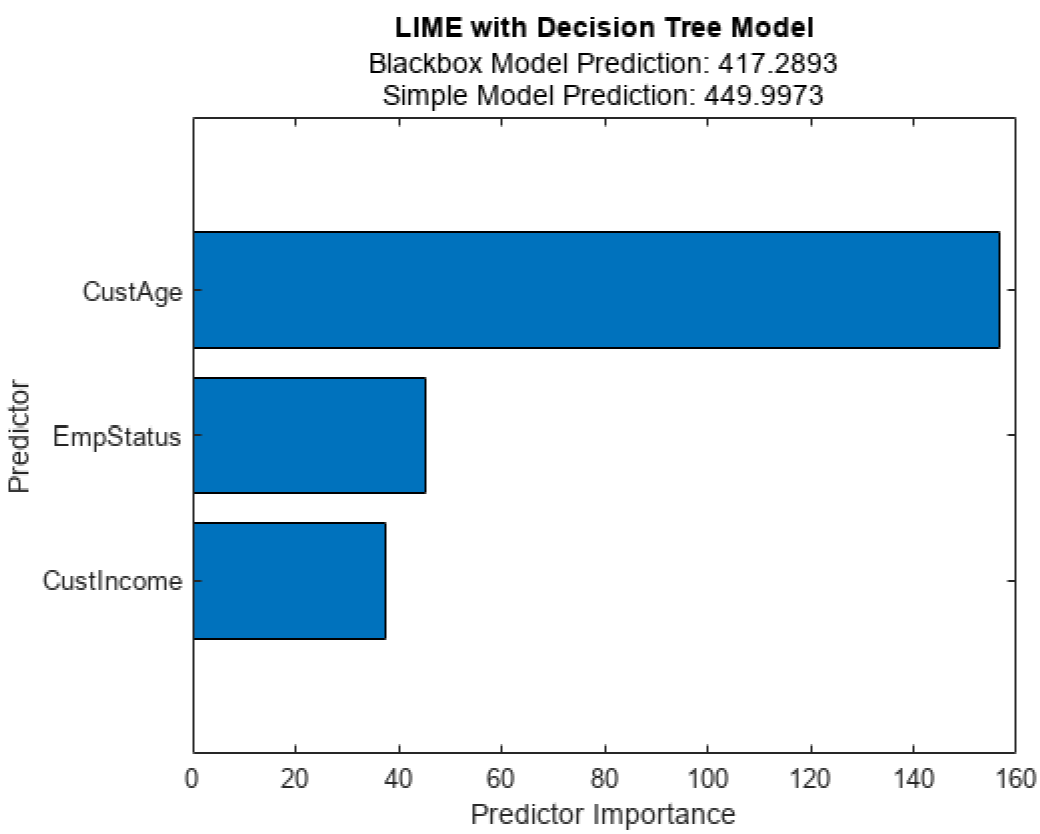
The local interpretable model-agnostic explanation (LIME) plot shows the coefficients of a local linear model near the instance of a score that you want to explain. LIME explains the scores around a particular observation, or query point, with a simple local model, such as a linear regression model or a decision tree.

Use `lime` to create a `lime` object specifying the data set of interest (the training data set), the model "type" (use "regression" to indicate a numeric prediction), and which variables are categorical. When you create a `lime` object, the toolbox generates a random synthetic data set. Use the synthetic data to fit simple local models to explain the local behavior.

```
rng('default'); % for reproducibility
limeExplainer = lime(@(tbl)score(sc,tbl),data(:,PredictorsInModel),Type="regression", ...
    CategoricalPredictors=["ResStatus" "EmpStatus"]);
```

Select a maximum number of predictors (`NumPredToExplain`) to explain and use a `SimpleModelType` of a `"tree"` to explain the local behavior of the score. The results are sensitive to the kernel width parameter (`KernelWidthChoice`) that controls how much neighbor points are weighted while fitting the linear simple model.

```
NumPredToExplain = 5 ; % number of variables/predictors to explain
KernelWidthChoice = 0.5 ;
limeExplainer = fit(limeExplainer,data(QueryPointIndex,PredictorsInModel), ...
   NumPredToExplain,SimpleModelType="tree",KernelWidth=KernelWidthChoice);

figure
f = plot(limeExplainer);
```



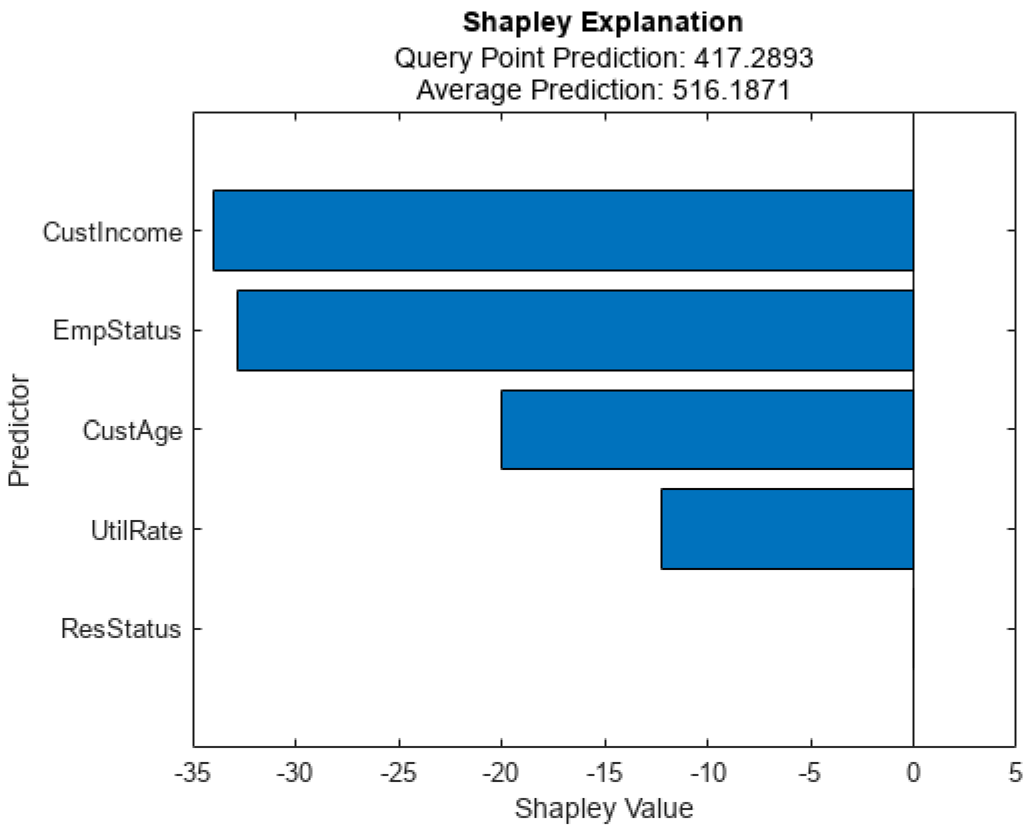
When the simple model is a tree, based on the reported predictor importance, customer age is the main predictor, followed by employment status and customer income.

Shapley Values

Shapley values explain the deviation of the predicted score from the average predicted score. The sum of the Shapley values for all predictors corresponds to the total deviation of the score for the query point from the average score.

The Shapley values are estimated based on a simulation. For larger data sets, this simulation is time consuming. For illustration purposes, this example uses only 500 rows of the training data with the `shapley` constructor.

```
rng('default'); % for reproducibility

shapleyExplainer = shapley(@(tbl)score(sc,tbl),data(500,PredictorsInModel), ...
    QueryPoint=data(QueryPointIndex,PredictorsInModel),CategoricalPredictors=["EmpStatus" "ResSta

figure
plot(shapleyExplainer)
```



For the query point with index `92`, the predicted score is `417`, whereas the average score for the training data set passed to `shapley` function is `516`. You expect the Shapley values to be negative, or at least have important negative components that explain why the predicted score is below average. In contrast, for scores above average, the Shapley values add up to a positive amount. In this example, the estimated Shapley values show that the main deviation from the average is explained by the customer income and employment status predictors, followed by the customer age and utilization rate predictors. The residential status predictor is not important. This result might be a combination of the simulation itself with the fact that residential status has a smaller impact on scores for this model.

Final Remarks

Explainability techniques are widely used to understand the behavior of predictive models. In this example, a `creditscorecard` model shows how explainability techniques, such as PDP, ICE, LIME, and Shapley are applied to explain a black-box model. Although credit scorecard models are simple and interpretable, you can apply the explainability tools in this example to other scoring models that are treated as black-box models or to supported models in Statistics and Machine Learning Toolbox. Alternatively, instead of explaining the scores, you can pass the `probdefault` function as the black-box model to explain the probability of default predictions.

Local Functions

```
function plotQueryInHistogram(VariableChoice,QueryPointIndex,Scores,PointsTable)

if VariableChoice=="Score"
   HistData = Scores;
else
   HistData = PointsTable.(VariableChoice);
end

histogram(HistData)
hold on
xline(HistData(QueryPointIndex),':','LineWidth',2.5)
hold off
xlabel(VariableChoice)
ylabel('Frequency')

end
```

See Also

Related Examples

- "Measure Transition Risk for Loan Portfolios with Respect to Climate Scenarios" on page 4-231
- "Assess Physical and Transition Risk for Mortgages" on page 4-248

External Websites

- Modeling the Impact of Transition and Physical Climate Risks on a Portfolio of Mortgages (13 min 52 sec)

Model Risk Management with Modelscape

Get Started with Modelscape

Modelscape™ provides workflow tools for model lifecycle support, including governance, automation, documentation, and operation in a unified, customizable system. Use Modelscape if you are a professional with a role related to model development and deployment in financial services. Modelscape supports models you build with MATLAB, Python®, R, SAS®, and other programming languages.

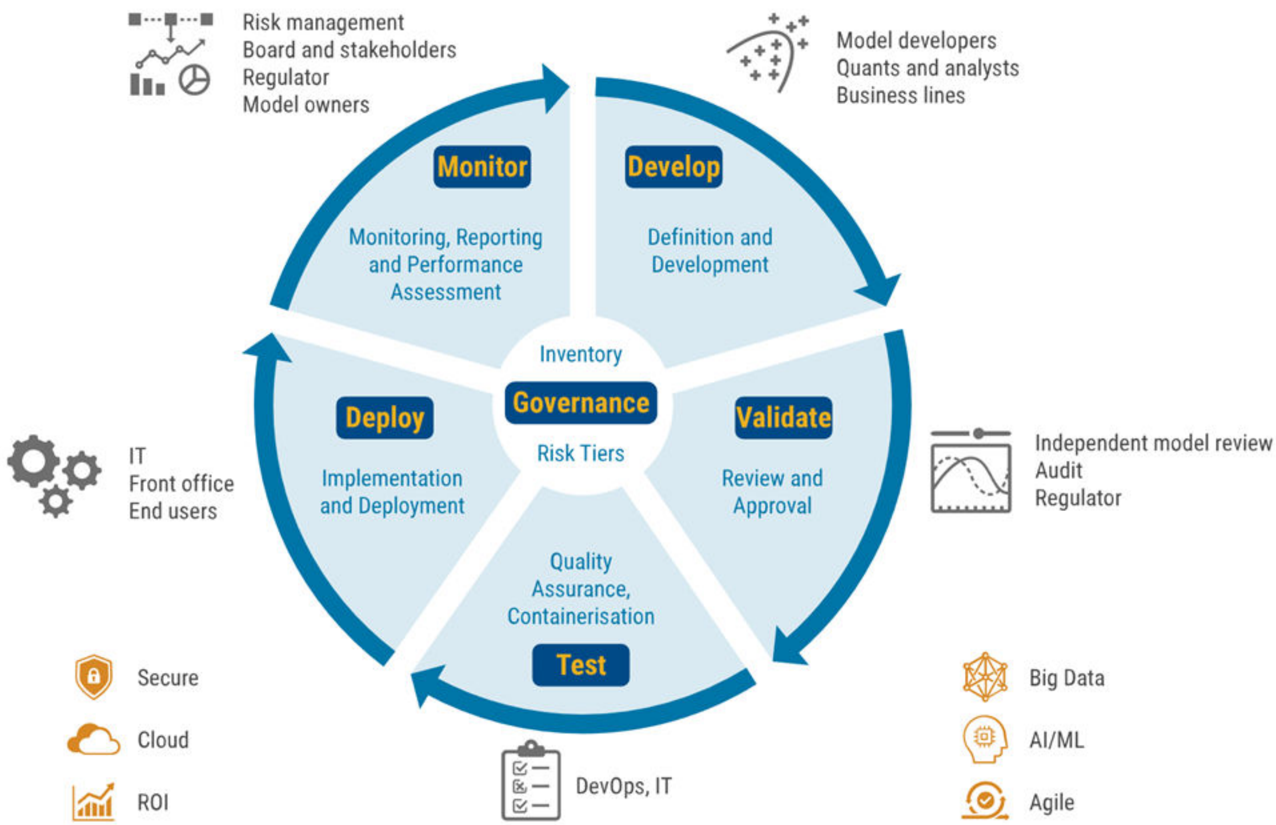
Installation

To host Modelscape for your organization, contact MathWorks Consulting Services. After you have set up Modelscape, to use it in MATLAB R2023a, download the Support Software Downloader for your operating system, run it, and install Modelscape for MATLAB.

Modelscape Workflow

Use Modelscape to implement workflows for creating, documenting, validating, deploying, and managing financially regulated models and associated data.

This figure shows how these components work together and the corresponding audience for each of the workflow steps.



Modelscape includes these workflow tools:

- “Modelscape Governance” on page 5-4 — Bring together the management of the whole lifecycle across the model inventory, workflow, risk tiering, and management reporting.
- “Modelscape Develop” on page 5-6 — Work with reusable components and automatic model documentation.
- “Modelscape Validate” on page 5-9 — Bring languages, tools, datasets, and application programming interfaces (APIs) together in one place.
- “Modelscape Test” on page 5-12 — Collate model artifacts for quality assurance, packaging, and deployment.
- “Modelscape Deploy” on page 5-13 — Securely execute, scale, and audit cross-language model that you store onsite or in cloud.

See Also

More About

- Model Risk Management Lifecycle
- MathWorks Modelscape Model Risk Management (1 min 38 sec)
- Improving Model Governance with MATLAB

Modelscape Governance

Modelscape Governance™ is an interactive environment that comprises a regulatory-compliant governance solution for all models in a business area. Use Modelscape Governance if you develop, validate, review, test, deploy, monitor, and use models. Modelscape Governance brings together Modelscape models, model versions, and lifecycles. To customize the environment further for your needs, see “Extensibility” on page 5-15.

Use Modelscape Governance to perform these tasks:

- Model metadata management for any programming language or spreadsheet
- Metadata tracking for data you use in the model development process
- Dependency analysis and tracking for models and data
- Inspection, viewing, and analysis of model and data relationships
- Viewing and reporting on the model state within the model lifecycle
- Management of the model hierarchy and dependencies

You can also extend and customize Modelscape Governance to perform these tasks:

- Score model risk and automate reporting.
- Customize workflow and approval procedures.
- Review, comment on, and approve models for state changes across the model lifecycle.
- Generate reports and customized dashboards for model governance.
- Integrate models with internal or vendor applications.

For more information about extending and customizing Modelscape Governance, see “Extensibility” on page 5-15.

Modelscape Governance Workflow

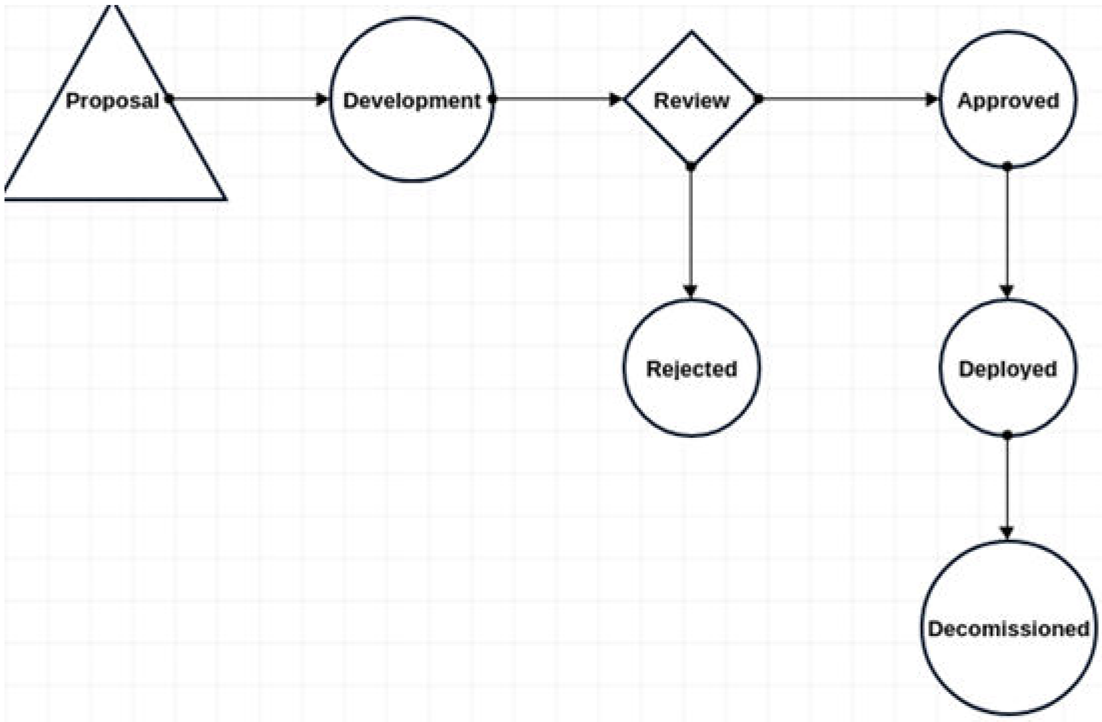
Working with Modelscape Models and Model Versions

Modelscape models are quantitative solutions that apply statistical, economic, or other techniques to given inputs to produce an output. You can use these outputs to guide pricing or other business decisions. Modelscape models must be backed by a Git™ repository and associated with a lifecycle.

Each Modelscape model has multiple model versions. Model versions are committed updates of a Modelscape model. For example, Probability of Default for Retail Credit in Europe is a Modelscape model, and its 2015 and 2020 versions are the model versions. While a Modelscape model corresponds to a Git repository, a model version refers to a Git commit.

Create Lifecycles Using Lifecycle Designer

Each Modelscape model must be associated with a lifecycle. A lifecycle represents the steps of a model version from drafting and proposal to retiring and decommissioning. At any time, each model version is in one specific state of the model lifecycle. This figure shows a simplified example lifecycle:



To create and edit lifecycles, use the Lifecycle Designer app.

Create Modelscape Models and Add Dependencies Between Modelscape Models

You can use the Inventory Browser app to create new models.

When you have more than one model, you can add dependencies between the models using the Dependency Editor in the Inventory Browser app.

See Also

Apps
Inventory Browser | Lifecycle Designer

More About

- MathWorks Modelscape Governance (1 min 41 sec)
- Facilitating Model Governance with the MathWorks Model Inventory

Modelscape Develop

Modelscape Develop™ is a set of model development and documentation tools. These tools are intended for risk managers, analysts, and quants who develop, test, and document models for risk assessment and decision support.

Modelscape Develop comprises these tools:

- Interactive apps for data preparation, model construction, model testing, and validation
- A comprehensive library of validation statistics, machine and deep learning, financial, risk, and economic algorithms
- Customizable and reusable model development templates
- Automated model documentation generation

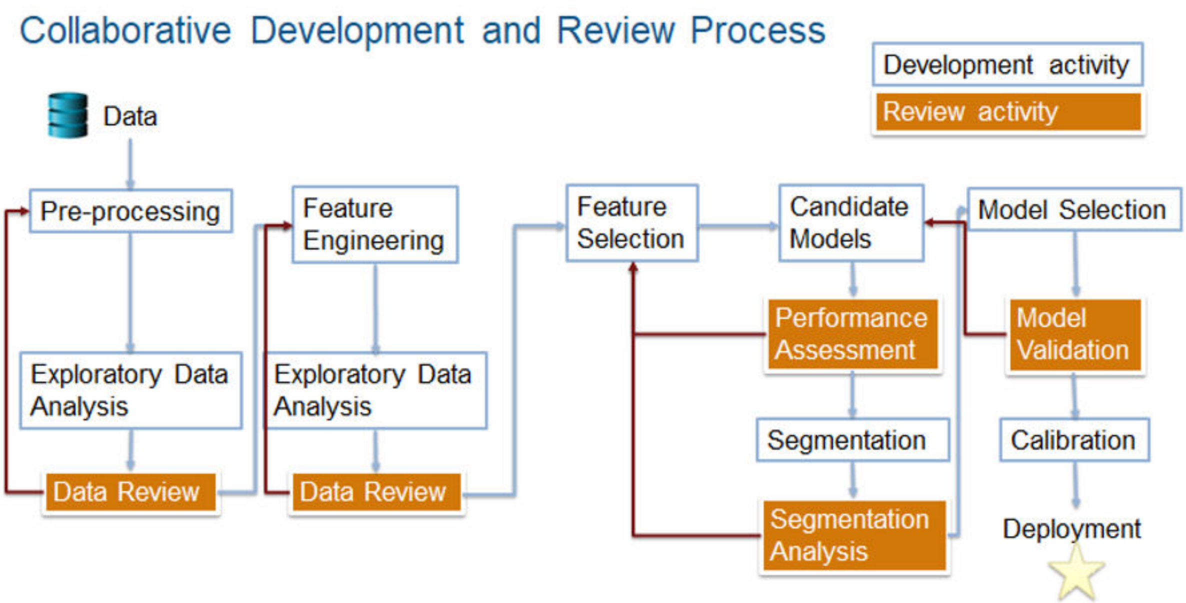
You can use Modelscape Develop to perform these tasks:

- Build, test, experiment with, and validate multiple models in parallel.
- Create models from validated pre-built functions instead of writing code or building your own libraries.
- Automate iterative workflows through code generation and reuse.
- Integrate algorithms and internal IP developed in any language or application.
- Generate live, auditable, and traceable documentation for model validation and governance.
- Preserve model development history for auditability, transparency, and knowledge transfer.

Modelscape Develop Workflow

Use Modelscape Develop to develop statistical and machine learning models in MATLAB.

This figure shows how to use the Modelscape Develop workflow in parallel with the Modelscape Validate™ workflow. The Develop workflow comprises the white boxes and the Validate workflow comprises the orange boxes. You can also perform the validation workflow independently after the development workflow.



Preprocess Data Using Live Tasks

Load data for your models in MATLAB. You can preprocess the data using the Remove Risk Factors and Screen Risk Factors live tasks.

Use Remove Risk Factors to interactively inspect variables from a data table and filter them out. You can also add the reasons for including or excluding variables and use the live task to document your analysis. For more information on how to do this, see “Remove Risk Factors” on page 5-22.

Use the Screen Risk Factors live task to interactively use predefined, customizable screening criteria to filter out input variables based on their predictive power. You can also add the reasons for including or excluding variables and use the live task to document your analysis. For more information on how to do this, see “Screen Risk Factors by Custom Criteria” on page 5-30.

You can also use the suite of metrics in Modelscape to analyze the bias in your data set. For more details, see “Fairness Metrics in Modelscape” on page 5-26.

Train Models Using MATLAB

After preprocessing your input features, you can use MATLAB to train your machine learning models. For example, you can use the **Classification Learner** and **Regression Learner** apps to train your models.

You can also use the suite of metrics in Modelscape to analyze the bias in your models. For more details, see “Fairness Metrics in Modelscape” on page 5-26.

Check Performance of Models

You can check the performance of your model using validation metrics available in Modelscape. For more details, see “Modelscape Validate” on page 5-9. After you compare models, you can select a model that suits your needs.

Perform Model Comparisons Using Experiment Manager

You can compare the performance of your models against each other or against existing models using the app. For more information, see “Model Development and Experiment Manager” on page 5-16.

Document Model Development Process

Document the results and analyses of the model in a Microsoft® Word document from MATLAB. Many workflows in financial institutions involve writing and submitting reports to internal control functions or regulatory bodies. These documents often conform to a given house style and are typically Microsoft Word documents. Using Modelscape, you can add text, visualization, and tabular content to a Microsoft Word document from MATLAB. For more information, see “Model Documentation in Modelscape” on page 5-35.

After you develop a model, you can pass it to one of these stages:

- Validation stage — For more information, see “Modelscape Validate” on page 5-9.
- Test stage — For more information, see “Modelscape Test” on page 5-12.
- Deployment stage — For more information, see “Modelscape Deploy” on page 5-13.

See Also

Apps
Remove Risk Factors | Screen Risk Factors

Related Examples

- “Remove Risk Factors” on page 5-22
- “Screen Risk Factors by Custom Criteria” on page 5-30
- “Model Development and Experiment Manager” on page 5-16
- “Fairness Metrics in Modelscape” on page 5-26
- “Model Documentation in Modelscape” on page 5-35

More About

- MathWorks Modelscape Develop (1 min 29 sec)

Modelscape Validate

Modelscape Validate is a model validation and documentation workflow. Use this workflow if you review business line data, models, assumptions, or if you build challenger models. Use Modelscape Validate to validate models written in any programming language.

You can record anomalies or observations related to models and the appropriate closure measures or actions and export the findings to a PDF file using the **Review Editor** app. The app also logs and stores all activity with the model, which ensures traceability and reproducibility.

Modelscape Validate comprises these features:

- Interactive apps for data preparation, model construction, model testing, and validation
- A comprehensive library of validation statistics, machine and deep learning, financial, risk, and economic algorithms
- Customizable and reusable model development and documentation templates
- Automated model documentation generation

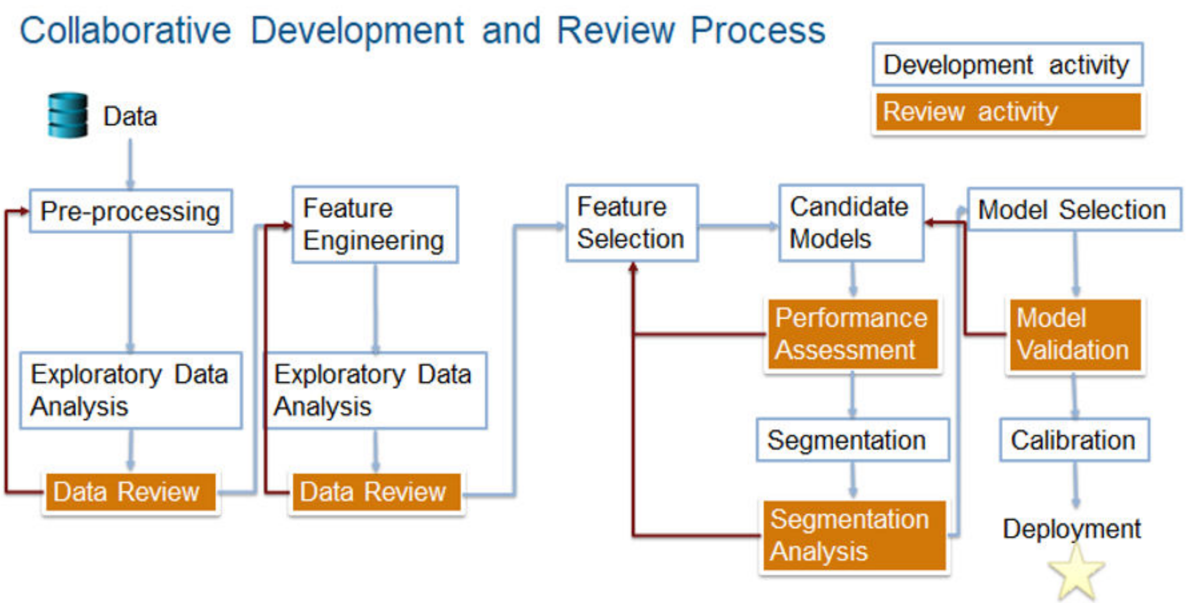
You can use Modelscape Validate to perform these tasks:

- Build, test, experiment with, and validate multiple models in parallel.
- Create models from validated pre-built functions instead of writing code or building your own libraries.
- Automate iterative workflows through code generation and reuse.
- Integrate algorithms and internal IP.
- Generate live, auditable, and traceable documentation for model validation and governance.
- Preserve model development history for auditability, transparency, and knowledge transfer.

Modelscape Validate Workflow

This figure shows how to use the Modelscape Develop workflow in parallel with the Modelscape Validate workflow. The Develop workflow comprises the white boxes and the Validate workflow comprises the orange boxes. You can also perform the validation workflow independently after the development workflow.

After you develop a model version, you can propose to deploy it. To deploy the model version, you must first validate it.



Raise Review Request

To begin validation for a model version, lock the commit version of the Modelscape model in the GitHub® repository and raise a review request for that model version. This action sends a review request to the Validator team.

Open Review Editor App

Open the Modelscape Home page, look for the model version, and open a review. Opening a review opens the Review Editor app. Use the **Review Editor** app to perform model validation on that model version.

Analyze Model Version

Create live scripts using the Review Editor app and use them to explore and analyze data. You can run the models and perform what-if analyses, which include adjusting the parameters of models, using different methods for fitting models, and observing any consequent changes in the model performance.

You can use the metrics in Modelscape to analyze the bias in your models. For more details, see "Fairness Metrics in Modelscape" on page 5-26.

You can write scripts to implement model validation suites. For an example script that shows how to validate a probability of default model, see "Credit Scorecard Validation Metrics" on page 5-48. This example shows how to use the techniques in the BCBS Working Paper 14.

If your model version uses a programming language other than MATLAB, you can still use the validation tools from Modelscape. For more information, see "Validation of External Models" on page 5-60.

For an example that shows how to validate credit models using the European Central Bank template, see "Validation of Credit Models in ECB Templates" on page 5-57.

Attach Documents to Review

You can attach documents to the Review Editor app. These documents include recommendations for improvement or evidence of model performance reports outside the **Review Editor** app. Use the app to attach files such as detailed model validation documents and scripts supporting such documents. By default, the **Review Editor** attaches the files to the project or model repository.

You can customize the location to which you save documents. See "Extensibility" on page 5-15 for more details.

Make Review Decision

Finish your review by using the submit button and sign-off your form in the app. You can customize the **Submit Review** dropdown and sign-off forms based on the needs of your organization. For more details, see "Extensibility" on page 5-15.

By default, the Review Editor app provides a full review form and a reduced review form. Use the reduced review form to approve the model if the changes are trivial. If the changes are not trivial and have a significant material impact, for example on the value of trade, use the full review form.

You can choose to approve or reject the model version. The next stages depend on the model lifecycle. If you do not approve the model version, it could be sent back to the developer. If you approve the model version, it could be sent to production and deployment.

See Also

Apps
Review Editor

Related Examples

- "Fairness Metrics in Modelscape" on page 5-26
- "Credit Scorecard Validation Metrics" on page 5-48
- "Metrics Handlers" on page 5-45
- "Validation of External Models" on page 5-60
- "Validation of Credit Models in ECB Templates" on page 5-57

More About

- MathWorks Modelscape Validate (1 min 32 sec)
- Model Validation is Everyone's Business
- BCBS Working Paper 14

Modelscape Test

Modelscape Test™ is a set of model implementation and testing workflow tools. Use these tools if you build, test, and deploy quantitative models to business systems and end users.

Modelscape Test supports these workflows:

- Test authoring frameworks for unit and performance testing
- Integration with continuous integration (CI) and continuous delivery (CD) tools
- Automated testing and reporting
- Pre- and post-production model verification reporting

You can use Modelscape Test to perform these tasks:

- Eliminate rework through minimizing or eliminating the need to recode quant models.
- Automate model testing, acceptance, and deployment to production.
- Generate live, auditable, and traceable documentation of model verification pre- and post-production deployment.
- Preserve model testing and deployment history for auditability, transparency, and knowledge transfer.

Modelscape Test Workflow

After you develop a model version, follow these steps to test your code.

Write Tests

Use MATLAB to write tests for your code. For more information, see “Write Script-Based Unit Tests”, “Write Function-Based Unit Tests”, and “Author Class-Based Unit Tests in MATLAB”.

Run in CI and Get Test Results

Set up a MATLAB project for continuous integration in a CI platform. Use this platform to create a continuous integration workflow to perform automated testing and obtain test results. For more details, see “Continuous Integration Using MATLAB Projects and Jenkins”.

Modelscape Deploy

Modelscape Deploy™ is a model execution workflow. Use this workflow if you deploy quantitative models to business systems and end users and manage their operation.

Modelscape Deploy comprises these features:

- Automated model deployment and prediction interface registration
- Model auditing of data inputs, outputs, and model use
- Single model deployment that can be called by multiple languages, applications, and web services
- Encrypted model packaging

Use Modelscape Deploy to perform these tasks:

- Maintain a single model used by multiple business applications and programming languages.
- Operationalize models that you build in MATLAB, Python, R, SAS and other languages.
- Generate model and data use reports.
- Monitor and manage server performance programmatically or through dashboards.

Modelscape Deploy Workflow

Follow these steps to build a Docker® image from your Modelscape model, create a build for the model, and the deploy the build.

Create Modelscape Model

Start with your model, such as a credit default model. Your model can be written in MATLAB or another programming language. If your model is written in MATLAB, follow all of these steps to create a Docker image. Otherwise, skip to the “Create Docker Image from Model” on page 5-13 section.

Begin by implementing a subclass that inherits from `mrm.execution.Model` class. You can use this class to make a Modelscape model. For more details, see the Work with Modelscape Deploy section in the “Model Implementation for Modelscape Deploy” on page 5-74.

Test Modelscape Model

Call the `checkModel` function with the original model, Modelscape model, and the model parameters as inputs. Performing this recommended optional step ensures that the Modelscape model has the correct set of inputs, parameters, and outputs. For more details, see `checkModel`.

Create Docker Image from Model

Create an image for deployment using the `packageModel` function. For more details about this function, see `packageModel` and “Model Implementation for Modelscape Deploy” on page 5-74. Running this function on your model creates a Docker image in the local Docker registry.

If your model is not written in MATLAB, you need to create a Docker image of the model. You also need to create a Docker image of a web server that listens to port `8080` and responds to the `signature/` and `evaluation/` endpoints with defined payloads.

Push Docker Image to Registry

Push the image to a registry that is visible to Modelscape using `docker push imageName`, where `imageName` is the name of your Docker image. For more information about the registry, talk to your system administrator or contact Consulting services at MathWorks®.

Create Build

Create a Build of your Docker Image using the `createBuild` function. For more details, see this example "Model Implementation for Modelscape Deploy" on page 5-74

Create and Execute Model Deployment

Create or use an existing deployment environment. You can use an isolated environment in your organization with specific permissions and security provisions. Ask your system administrator for more information.

Create a deployment, which is an instance of your build in the deployment environment. Creating a deployment is equivalent to creating a container from a Docker Image.

Execute the deployment from your deployment environment.

See Also

`checkModel` | `packageModel`

Related Examples

- "Model Implementation for Modelscape Deploy" on page 5-74

More About

- MathWorks Modelscape Deploy (1 min 29 sec)
- Model Execution Environment

Extensibility

Extend and customize your Modelscape environment.

Customize Inventory Browser

Customize the Inventory Browser app to make it specific to your organization. You can customize the model data entry and the model summary table. You can also add new filters to the Inventory Browser and find models with a particular attribute easily. For an example that shows how to customize the Inventory Browser app, see "Customizing Model Inventory: Risk Tiering" on page 5-78.

Implement Test Metrics

You can implement different test metrics in MATLAB using Modelscape tools. For an example that shows how to implement test metrics, see "Test Metrics in Modelscape" on page 5-85.

Customize Review Editor

You can attach review files to model reviews using the Review Editor app. These files include model validation documents and scripts supporting such documents. By default, you attach the files to the project (or model) repository. You can also store these attachments in a network folder. For an example that shows how to customize and extend your Review Editor app, see "File Attachments in Modelscape Review Editor" on page 5-68.

You can also customize the signoff forms in the Review Editor app. For an example that shows how to customize signoff forms, see "Customization of Signoff Forms in Review Editor" on page 5-70.

See Also

Related Examples

- "Customizing Model Inventory: Risk Tiering" on page 5-78
- "Test Metrics in Modelscape" on page 5-85
- "File Attachments in Modelscape Review Editor" on page 5-68
- "Customization of Signoff Forms in Review Editor" on page 5-70

Model Development and Experiment Manager

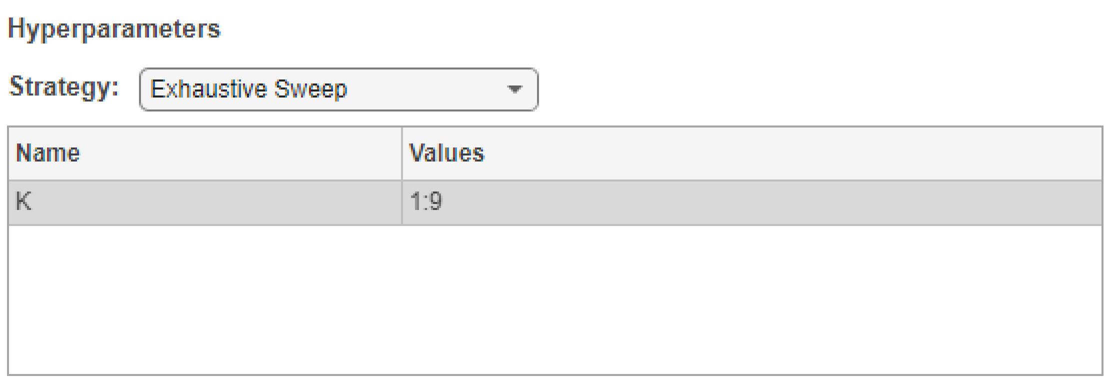
This example shows how to use MATLAB® Experiment Manager with Modelscape™ at various stages of model development.

This example sets up experiments, uses Modelscape validation metrics in the process, and bridges the gap between Experiment Manager and model documentation. This example uses a feature selection process that works through all the subsets of the predictors to find the best subset using the performance metric of the area under the receiver operating characteristic (AUROC). Such exhaustive feature selection, though computationally intense, allows you to compare against more effective methods in the process of model validation.

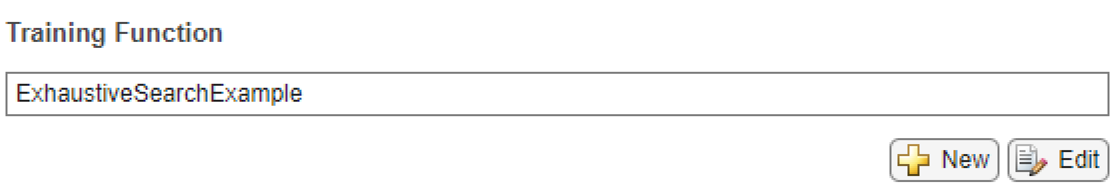
This example uses the `CreditCardData.mat` data set, which contains three tables of customer information such as age, income, and employment status. After excluding the response variable (`status`) and the customer id, this data set has nine possible predictors. This example creates an experiment with nine trials such that the Kth trial runs through all the K-element subsets of the maximal, nine-element predictor set. This example shows how to set up hyperparameters, write and run the experiment, and document the experiment results.

Write Experiment in Experiment Manager

Load the Experiment Manager App from the app gallery in MATLAB. Create a Blank Project and select ‘Custom Training’ under ‘Blank Experiments’. Set a single hyperparameter.



To write the experiment function, add a function name to the ‘Training Function’ box.



Clicking Edit opens a Live Script. Fill in the function written below. The function has two inputs. `params` is a struct whose fields correspond to the given hyperparameters (in this case `K`), and `monitor` is an `experiments.Monitor` object.

```
function output = ExhaustiveSearchExample(params,monitor)
    allData = load('CreditCardData.mat');
```

```
    data = allData.data;

    monitor.Metrics = "AUROC";
    monitor.Info = ["MaxAUROC", "MeanAUROC", "StdDevAUROC"];
    allVars = data.Properties.VariableNames;
    predictorFlags = ~ismember(allVars, {'status', 'CustID'});
    predictorVars = allVars(predictorFlags);

    N = numel(predictorVars);
    K = params.K;
    numRuns = nchoosek(N, K);
    masks = mrm.data.filter.allMasks(N,K);
    bestAUROC = 0;
    allAurocs = zeros(numRuns, 1);

    for i = 1:numRuns
        % choose a set of predictors
        thesePredictors = predictorVars(masks{i});

        % fit the model for these predictors
        sc = creditscorecard(data, 'IDVar', 'CustID', 'ResponseVar', 'status', ...
            'GoodLabel', 0, 'BinMissingData', true, ...
            'PredictorVars', thesePredictors);
        sc = autobinning(sc);
        sc = fitmodel(sc, 'VariableSelection','fullmodel');
        monitor.Progress = i/numRuns*100;

        % record performance metrics
        aurocMetric = mrm.data.validation.pd.AUROC(data.status, score(sc));
        recordMetrics(monitor, i, "AUROC", aurocMetric.Metric);
        allAurocs(i) = aurocMetric.Metric;
        if aurocMetric.Metric > bestAUROC
            bestAUROC = aurocMetric.Metric;
            updateInfo(monitor, "MaxAUROC", aurocMetric.Metric);
            output.model = sc;
            output.predictors = thesePredictors;
        end
    end
    updateInfo(monitor, "StdDevAUROC", std(allAurocs), "MeanAUROC", mean(allAurocs));
end
```

The `Monitor` object can record two types of data: 'Metrics' and 'Info'.

- Metrics are parametrized data (in this example by the index of the predictor subset). Save metric levels by calling `recordMetrics`.
- Information fields carry just a single datum per trial. Save information fields by calling `updateInfo`.

You can write several metrics and information fields. This example reports the maximum, the mean, and standard deviation of all the recorded AUROC scores. This allows you to compare the distribution of the achieved AUROC values with the mean value.

The output of the experiment consists of the optimal set of predictors along with a model fitted using that subset.

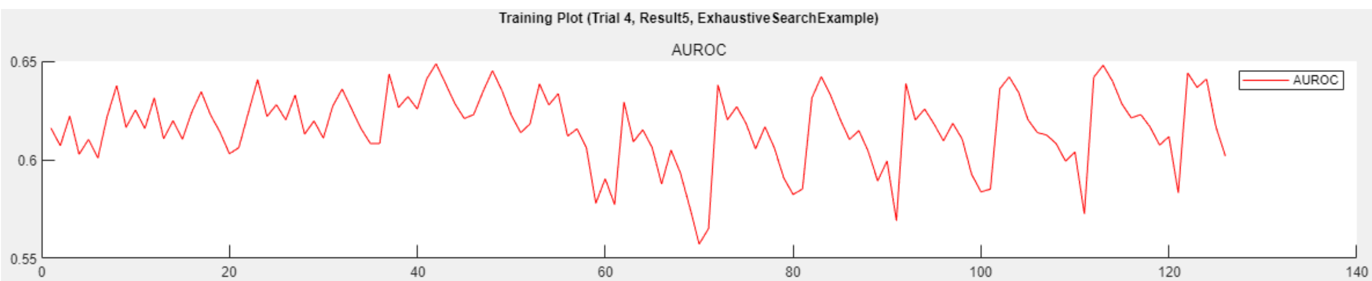
Analyze Experiment Results

Run the experiment to produce the following table.

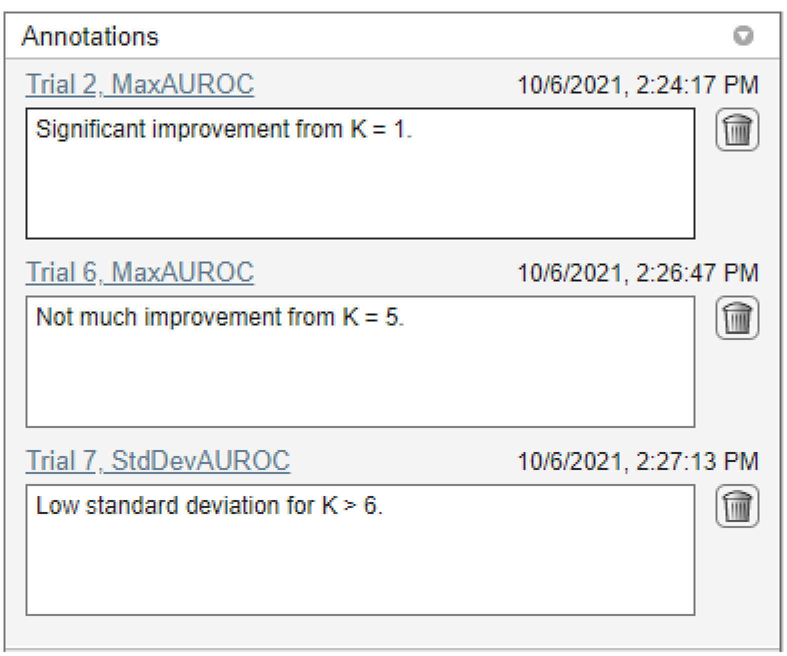
Trial	Status	Progress	Elapsed Time	K	MaxAUROC	MeanAUROC	StdDevAUROC	AUROC
1	Complete	100.0%	0 hr 0 min 3 sec	1.0000	0.5922	0.5493	0.0283	0.5263
2	Complete	100.0%	0 hr 0 min 5 sec	2.0000	0.6254	0.5793	0.0258	0.5495
3	Complete	100.0%	0 hr 0 min 10 sec	3.0000	0.6412	0.6001	0.0224	0.5624
4	Complete	100.0%	0 hr 0 min 16 sec	4.0000	0.6487	0.6160	0.0188	0.6020
5	Complete	100.0%	0 hr 0 min 19 sec	5.0000	0.6542	0.6289	0.0154	0.6442
6	Complete	100.0%	0 hr 0 min 15 sec	6.0000	0.6579	0.6394	0.0122	0.6505
7	Complete	100.0%	0 hr 0 min 8 sec	7.0000	0.6613	0.6483	0.0093	0.6545
8	Complete	100.0%	0 hr 0 min 4 sec	8.0000	0.6626	0.6559	0.0063	0.6557
9	Complete	100.0%	0 hr 0 min 2 sec	9.0000	0.6626	0.6626	0.0000	0.6626

The table shows you the maximum achieved AUROC along with the mean and standard deviation for each trial. The final column in the table shows the value of the AUROC metric for the last K-element predictor set.

Clicking 'Training Plot' in the Experiment Manager shows how the metric varies over the K-element subsets.



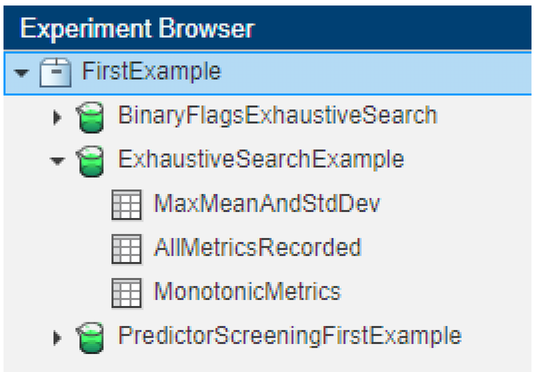
You can add 'annotations' to the summary table by right-clicking on any cell.



Document with Modelscape Reporting

To record your findings in model documentation, use Modelscape Reporting. Use function `fillReportFromWorkspace` to include development artifacts such as tables in Microsoft Word documents. For more information, see “Model Documentation in Modelscape” on page 5-35.

You can extract the summary table and the annotations from the Experiment Manager outputs and insert them in documents using `fillReportFromWorkspace`. To do this, you need the name of the experiment and a set of results. These can be found in the Experiment Browser part of Experiment Manager.



Here 'FirstExample' is the name of the *project*, 'ExhaustiveSearchExample' is the name of the *experiment*, and 'MaxMeanAndStdDev' is name of the set of *results*. You can rename the experiment and the results by right-clicking on these names.

Use function `extractExperimentResults` in either Live Script or Command Window to extract the summary table and the annotations. This call should take place in the root folder of the project - otherwise use the optional `ProjectFolder` argument to point to the correct location.

```
[results, annotations] = extractExperimentResults('ExhaustiveSearchExample', 'MaxMeanAndStdDev')
```

```
results =

  9×7 table

    Trial      Status      K     AUROC     MaxAUROC    MeanAUROC    StdDevAUROC
    _____    __________    _    _______    ________    _________    ___________

      1      "Complete"    1    0.52634    0.59218      0.54926        0.028252
      2      "Complete"    2    0.54954    0.62545      0.57925        0.025846
      3      "Complete"    3    0.56243     0.6412      0.60007        0.022431
      4      "Complete"    4    0.60196    0.64872      0.61601        0.018835
      5      "Complete"    5    0.64415    0.65425      0.62885        0.015395
      6      "Complete"    6    0.65047    0.65791      0.63941        0.012217
      7      "Complete"    7    0.65446    0.66129      0.64826        0.009329
      8      "Complete"    8    0.65573    0.66257      0.65589       0.0063327
      9      "Complete"    9    0.66258    0.66258      0.66258               0


annotations =

  3×4 table

    Trial    K       Header                        Comment
    _____    _    _____________    ____________________________________

      2      2    "MaxAUROC"       "Significant improvement from K = 1."
      6      6    "MaxAUROC"       "Not much improvement from K = 5."
      7      7    "StdDevAUROC"    "Low standard deviation for K > 6."
```

You can then insert columns from these variables into placeholders titled `FSSummary` and `FSDetails` in the Word document.

```
FSSummary = results(:,{'K','MaxAUROC','MeanAUROC','StdDevAUROC'});
FSDetails = annotations(:,{'K','Header','Comment'});
```

Push these tables to the model document.

```
previewDocument = fillReportFromWorkspace('ExhaustiveDocExample.docx');
winopen(previewDocument)
```

Your tables then appear in the Word document.

Statistical Model Document v. 1.0

Background

Feature Selection

The results of an exhaustive feature selection process are summarised in the following table.

K	MaxAUROC	MeanAUROC	StdDevAUROC
1	0.5922	0.5493	0.0283
2	0.6254	0.5793	0.0258
3	0.6412	0.6001	0.0224
4	0.6487	0.6160	0.0188
5	0.6542	0.6289	0.0154
6	0.6579	0.6394	0.0122
7	0.6613	0.6483	0.0093
8	0.6626	0.6559	0.0063
9	0.6626	0.6626	0.0000

In our analysis of the results, we made the following observations:

K	Header	Comment
2	MaxAUROC	Significant improvement from K = 1.
6	MaxAUROC	Not much improvement from K = 5.
7	StdDevAUROC	Low standard deviation for K > 6.

Model Fitting

Remove Risk Factors

This example shows how to remove or include variables from a table and record the corresponding reasons using the Modelscape™ Remove Risk Factors task.

The example also shows how to include the results of this analysis in model documents using the Modelscape reporting feature.

All columns in a table of input data may not be relevant while developing a statistical model. Not all the data in the table is necessarily usable for a statistical model. For example, randomized user identifiers (IDs) are often irrelevant, legally sensitive data such as ethnic origin or religious beliefs cannot be used, and some data can be of poor quality. This example shows you how to select relevant variables in such a table and record your reasons.

This example uses the Credit Scorecard data set, which contains three tables of customer information such as age, income, and employment status. One such table, `dataMissing`, deliberately has a few blank entries in the data set. The data could be used for developing a statistical model such as a MATLAB® credit scorecard model. The example loads the data set in the Remove Risk Factors task, marks some variables for exclusion, and documents the results using Modelscape reporting.

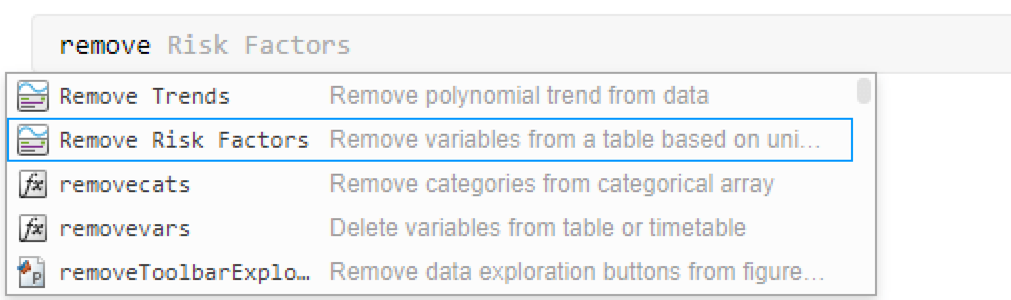
Load Data and Launch the Tool

Load the input data from CreditCardData.mat.

```
load CreditCardData
```

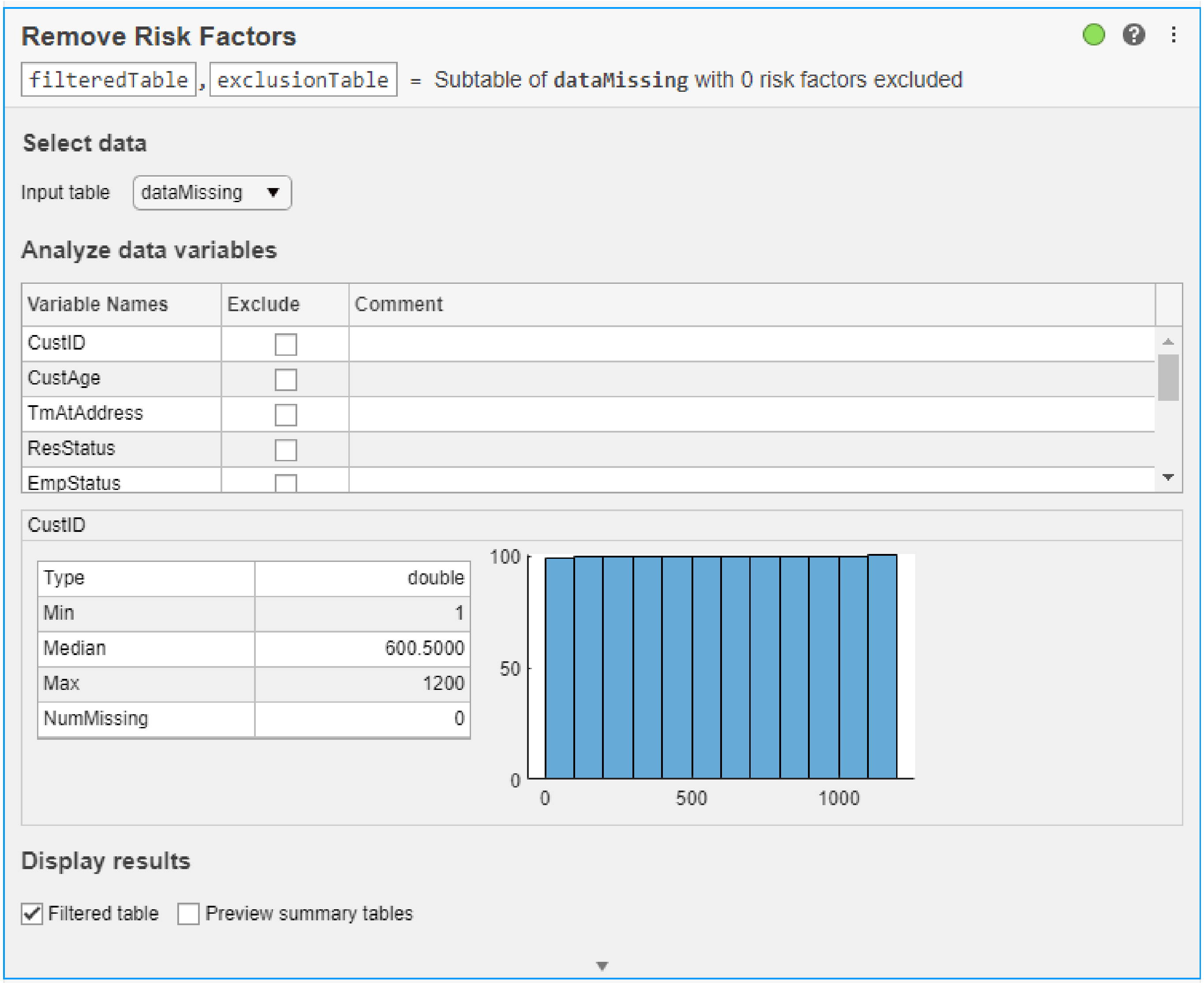
Open a new live script. There are two ways to open the Remove Risk Factors task:

1 Type `remove` and select `Remove Risk Factors` in the drop-down selection.



2. Search for the tool under Task in the Live Editor gallery.

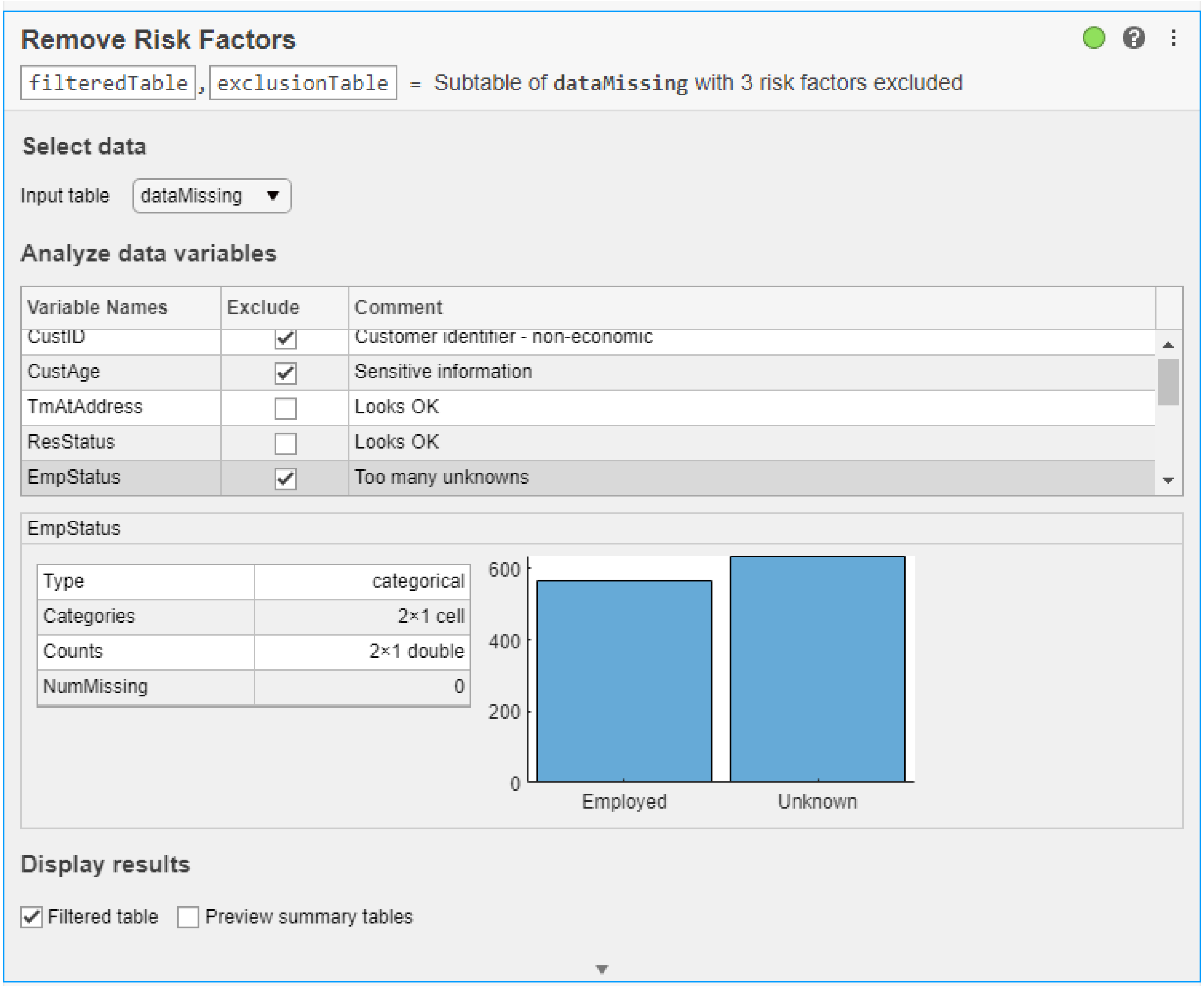
In the task, select your input data, for example `dataMissing` variable.



Inspect and Filter Variables

The task shows the summary statistics and the histogram for the first variable in the table (in this case `CustID`).

To inspect other variables, click the corresponding variable name in the `Analyze data variables` section. This section contains three columns that you can sort. The `Variable Names` column is read-only. The `Exclude` column allows you to exclude variables from the table. To do this, check the `Exclude` button to mark the corresponding variable for removal. The `Comment` column lets you add reasons for the exclusion (or inclusion) by double-clicking the box.



When you exclude variables and add comments, the task dynamically produces two outputs:

- `filteredTable`: This is a subtable of the input table without the excluded risk factors. Use this subtable in the next step of the model development process - for example feature selection.
- `exclusionTable`: This table includes all the data of the input table together with the exclusion flags and comments in the task. To view this information, tick the 'Preview summary tables' box in 'Display results' section. This information is stored in `exclusionTable.Properties.CustomProperties` meta data.

exclusionSummaryPreview = 11×3 table

	Excluded	Variable Names	Comm...
1	1	"CustID"	"Customer id...
2	1	"CustAge"	"Sensitive inf...
3	0	"TmAtAddress"	"Looks OK"
4	0	"ResStatus"	"Looks OK"
5	1	"EmpStatus"	"Too many u...
6	0	"CustIncome"	""
7	0	"TmWBank"	""
8	0	"OtherCC"	""
9	0	"AMBalance"	""

progressSummaryPreview = 4×2 table

		Risk factor count	% of Total
1	All variables	11	100
2	Excluded	3	27
3	Included	8	73
4	Commented	5	45

`progressSummaryPreview` lists the total number of variables, the excluded variables, the included variables, and the number of variables with comments. You can use this last datum to indicate whether the removal process is complete - in the end, every variable must have a reason for either exclusion or some indication that the variable has been inspected.

Document with Modelscape Reporting

Use Modelscape Reporting to document the findings of the analysis described above. Use the meta data stored in `exclusionTable` for this purpose. To include the tables shown above as `exclusionSummaryPreview` and `progressSummaryPreview` in a Word document, create document holes with titles `ExclusionSummary` and `ProgressSummary` in the Word document.

```
import mrm.data.filter.*
[ExclusionSummary, ProgressSummary] = summarizeExclusionTable(exclusionTable)
```

To create document holes in a Word document, view the Developer tab, and click the 'Rich Text Content Control' symbol **Aa** in the Controls area. Then click 'Properties' and fill in the `Title` fields.

Running `fillReportFromWorkspace` will then pick up these new variables from the MATLAB workspace and insert them into the model document.

For more information on `fillReportFromWorkspace`, see “Model Documentation in Modelscape” on page 5-35.

Fairness Metrics in Modelscape

This example shows how to detect bias in data and statistical models using a special suite of metrics in Modelscape™.

The metrics are built on the `fairnessMetrics` class from Statistics and Machine Learning Toolbox™ (SMLT).

Modelscape tools let you set thresholds for these metrics and produce reports that appear consistent with other Modelscape validation reports.

This example uses a pre-constructed SMLT fairnessMetrics object to assess bias in a credit card data set. This example assesses this data based on fifteen metrics using a RiskFairnessMetrics handler object and documents the results in a Word document.

Load Data and Create RiskFairnessMetrics Handler Object

Load a pre-prepared Modelscape bias detection object. These objects are constructed from the SMLT `fairnessMetrics` objects.

```
load FairnessEvaluator.mat
disp(fairnessData)
```

```
  fairnessMetrics with properties:

    SensitiveAttributeNames: {'AgeGroup'  'ResStatus'  'OtherCC'}
             ReferenceGroup: {'45 < Age <= 60'  'Home Owner'  'Yes'}
               ResponseName: 'Y'
              PositiveClass: 1
                BiasMetrics: [9x7 table]
               GroupMetrics: [9x20 table]
                 ModelNames: 'Model1'
```

This object carries the metrics for certain model predictions. It contains four bias metrics ("Disparate Impact", "Statistical Parity Difference", "Average Absolute Odds Difference" and "Equal Opportunity Difference"), and 11-group metrics, including "False Negative Rate" and "Rate of Positive Predictions". These metrics are based on data with three attributes ("AgeGroup", "ResStatus" and "OtherCC").

For evaluating data without model predictions, only the metrics "Disparate Impact" and "Statistical Parity Difference" would be present.

Construct a Modelscape fairness metric handler from a `fairnessMetrics` object to compute a metric for every bias and group metric.

```
riskFairnessMetrics(fairnessData)
```

```
ans =
  RiskFairnessMetricsHandler with properties:

        NegativePredictiveValue: [1x1 mrm.data.validation.fairness.NegativePredictiveValue]
    AverageAbsoluteOddsDifference: [1x1 mrm.data.validation.fairness.AverageAbsoluteOddsDifferen
               TrueNegativeRate: [1x1 mrm.data.validation.fairness.TrueNegativeRate]
              FalseNegativeRate: [1x1 mrm.data.validation.fairness.FalseNegativeRate]
      EqualOpportunityDifference: [1x1 mrm.data.validation.fairness.EqualOpportunityDifference]
```

```
      StatisticalParityDifference: [1x1 mrm.data.validation.fairness.StatisticalParityDifference]
                FalseOmissionRate: [1x1 mrm.data.validation.fairness.FalseOmissionRate]
                         Accuracy: [1x1 mrm.data.validation.fairness.Accuracy]
                  DisparateImpact: [1x1 mrm.data.validation.fairness.DisparateImpact]
        RateOfNegativePredictions: [1x1 mrm.data.validation.fairness.RateOfNegativePredictions]
        RateOfPositivePredictions: [1x1 mrm.data.validation.fairness.RateOfPositivePredictions]
                FalsePositiveRate: [1x1 mrm.data.validation.fairness.FalsePositiveRate]
               FalseDiscoveryRate: [1x1 mrm.data.validation.fairness.FalseDiscoveryRate]
          PositivePredictiveValue: [1x1 mrm.data.validation.fairness.PositivePredictiveValue]
                 TruePositiveRate: [1x1 mrm.data.validation.fairness.TruePositiveRate]
```

`RiskFairnessMetricsHandler` is one example of an Modelscape `MetricsHandler`. For more examples and further information on the properties of these objects, see “Metrics Handlers” on page 5-45.

The rest of this example describes the extra features available in `RiskFairnessMetricsHandler` not shared by the other metrics handlers.

Specify Thresholds for Fairness Metrics

You can specify thresholds for the fairness metrics using the `riskFairnessThresholds` function. Depending on the metric, `riskFairnessThresholds` expects either a single threshold level or two threshold levels:

- `DisparateImpact, StatisticalParityDifference, EqualOpportunityDifference`, and the metrics for rates of positive and negative predictive value expect two inputs. A metric level between the two inputs is considered a "Pass", and values outside this range is considered a "Fail". For the rates of positive and negative predictive value, the threshold is not compared against the rate itself, but against the deviation of the rate from the true positive or negative rate.
- Other metrics require a single threshold value. A metric value is assigned a "Pass" or a "Fail" status depending on which side of this threshold the value is. `riskFairnessThresholds` automatically works out which way these statuses go for each metric.

To set the thresholds, specify name-value pairs using the metric names in the `riskFairnessThresholds` function. For example, set the thresholds for `StatisticalParityDifference` and `FalseNegativeRate`.

```
fairnessThresholds = riskFairnessThresholds("StatisticalParityDifference", [-0.15, 0.2], ...
    "FalseNegativeRate", 0.6)
```

```
fairnessThresholds = 
  RiskFairnessThresholds with properties:

              FalseNegativeRate: 0.6000
    StatisticalParityDifference: [-0.1500 0.2000]
```

For `StatisticalParityDifference`, a value within the range -0.15 to 0.2 is assigned a "Pass", and values outside of this range are considered as "Fail". For False Negative Rate, values below 0.6 return a "Pass"; otherwise, a "Fail" is returned.

Construct a fairness metric handler with these thresholds.

```
fairnessMetricHandler = riskFairnessMetrics(fairnessData, fairnessThresholds);
```

Interrogate Fairness Metrics

Use the `report` method to interrogate fairness metrics. This method summarizes across all the metrics, sensitive attributes, and attribute groups.

```
overallSummary = report(fairnessMetricHandler);
disp(overallSummary)
```

```
            Summary Metric                Value        Status                            Diagnos
    ________________________________    ________    ___________    ______________________________

    Statistical Parity Difference        0.54197    Fail           (0.2, Inf)
    Disparate Impact                    0.050237    <undefined>    <undefined>
    Equal Opportunity Difference         0.39151    <undefined>    <undefined>
    Average Absolute Odds Difference     0.49949    <undefined>    <undefined>
    False Positive Rate                    0.775    <undefined>    <undefined>
    False Negative Rate                        1    Fail           (0.6, Inf)
    True Positive Rate                         0    <undefined>    <undefined>
    True Negative Rate                     0.225    <undefined>    <undefined>
    False Discovery Rate                       1    <undefined>    <undefined>
    False Omission Rate                      0.4    <undefined>    <undefined>
    Positive Predictive Value                  0    <undefined>    <undefined>
    Negative Predictive Value                0.6    <undefined>    <undefined>
    Rate of Negative Predictions         0.23438    <undefined>    <undefined>
    Rate of Positive Predictions         0.76562    <undefined>    <undefined>
    Accuracy                             0.42188    <undefined>    <undefined>
    Overall                                  NaN    Fail           Fails at: Statistical Parity D
```

In this example, the model outputs fail both the tests. This failure is because the values shown in this table are the 'worst' levels seen across all attributes and groups.

For more details on these failures, pass extra arguments to `report`. For example, display detailed data about Statistical Parity Difference.

```
spdSummary = report(fairnessMetricHandler, "Metrics", "StatisticalParityDifference");
disp(spdSummary)
```

```
    SensitiveAttribute | Group    StatisticalParityDifference    Status                     Diag
    __________________________    ___________________________    ______    ___________________

    AgeGroup | Age <= 30                    0.54197               Fail     (0.2, Inf)
    AgeGroup | 30 < Age <= 45               0.42456               Fail     (0.2, Inf)
    AgeGroup | 45 < Age <= 60                     0               Pass     (-0.15, 0.2]
    AgeGroup | Age > 60                    -0.21242               Fail     (-Inf, -0.15]
    ResStatus | Home Owner                        0               Pass     (-0.15, 0.2]
    ResStatus | Tenant                     0.080908               Pass     (-0.15, 0.2]
    ResStatus | Other                      -0.11961               Pass     (-0.15, 0.2]
    OtherCC | No                            0.19661               Pass     (-0.15, 0.2]
    OtherCC | Yes                                 0               Pass     (-0.15, 0.2]
    Overall                                     NaN               Fail     Fails at: AgeGroup | A
```

In the data, the worst statistical parity difference is in the under-30s age group.

For data sets with many predictors or groups, you can focus on a single attribute or attribute group.

```
spdAgeGroupReport = report(fairnessMetricHandler, ...
    "Metrics", "StatisticalParityDifference", ...
    "SensitiveAttribute","AgeGroup");
disp(spdAgeGroupReport)
```

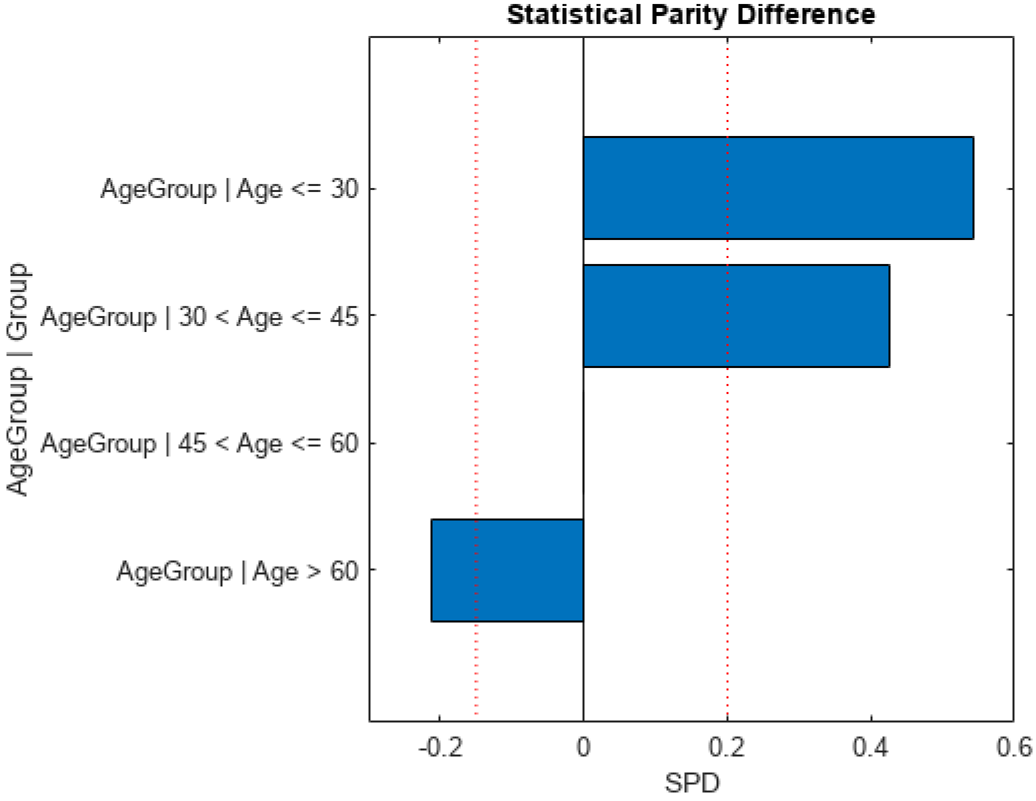
```
    AgeGroup | Group         StatisticalParityDifference    Status                  Diagr
_________________________    ___________________________    ______    ______________________

AgeGroup | Age <= 30                   0.54197               Fail     (0.2, Inf)
AgeGroup | 30 < Age <= 45              0.42456               Fail     (0.2, Inf)
AgeGroup | 45 < Age <= 60                    0               Pass     (-0.15, 0.2]
AgeGroup | Age > 60                   -0.21242               Fail     (-Inf, -0.15]
Overall                                    NaN               Fail     Fails at: AgeGroup | Ag
```

On the contrary, by omitting the "`Metrics`" argument, you can view a single attribute or attribute-group and see how this attribute or group measures by all the metrics.

Visualize Fairness Metrics

The fairness metrics handler supports different visualizations that are specific to bias detection and are not inherited from the generic `MetricsHandler` functionality. For individual metrics, `visualize` returns a bar chart across all sensitive attributes and groups, with vertical dotted lines indicating the thresholds. You can also restrict the view to a specific attribute.

```
visualize(fairnessMetricHandler, "Metric", "StatisticalParityDifference", "SensitiveAttribute", '
```



Screen Risk Factors by Custom Criteria

This example shows how to use the Screen Risk Factors task to automatically exclude risk factors from a table based on their predictive power.

This example also shows how to set up the screening criteria.

Feature selection is an important step in the development of a statistical model. Input data can have hundreds or thousands of variables, and discarding some variables often improves model interpretability, training times, and other important attributes.

This example loads the Screen Risk Factors data set, which contains a table of customer information such as age, income, and employment status. This example uses pre-defined metrics to assess risk factors individually and analyzes the predictive power of each variable relative to a given (binary) response variable. This example then shows you how to select variables automatically or semi-automatically using the Screen Risk Factors task. This example also shows you how to customize the screening criteria used to assess the risk factors.

Load Data and Pre-defined Screening Criteria

Load the example data from ScreenRiskFactorsData.mat.

```
load ScreenRiskFactorsData.mat
```

Construct pre-defined screening criteria in your workspace. Use `ExampleScreeningCriteria to generate myCriteria` object. This function returns a `ScreeningCriteria` object defined by an `mrm.data.selection.TestSuiteFactory.`

```
import mrm.data.selection.*
myCriteria = ExampleScreeningCriteria();
```

These criteria have been set up as follows:

1. For each variable, the Information Value and Chi-squared p-value are calculated.
2. These values are compared against certain thresholds that assign the metric a Pass, Fail or Undecided classification. In this case, the thresholds for the metrics are hard-coded but you can obtain the thresholds from the appropriate data in the development environment.
3. The overall classification works on the 'worst-of' basis. If the status for either Information Value or the Chi-squared p-value is a Fail, the overall status will be Fail, and so on.

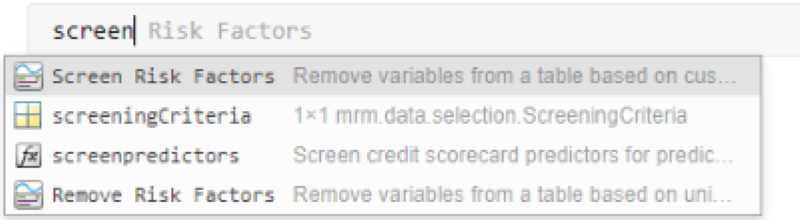
The `TestSuiteFactory` sets the `StatusInterpreter` of the metrics handler to `overallScreeningStatus`. This is where the auto-generated exclusions and comments are set. For the exclusions, the function must assign to each `MetricsHandler` state an `mrm.data.selection.ScreeningStatus` object (or an `ErrorTestStatus` or `NullStatus`) to ensures that the Screen Risk Factors task automatically marks the variable for exclusion.

In addition, the percentage of missing entries is displayed. This value does not affect the overall rating.

Launch Screen Risk Factors

Open a new live script and launch the Screen Risk Factors task. This can be done in two ways:

1) Start typing 'Screen' and select the task from the drop-down menu



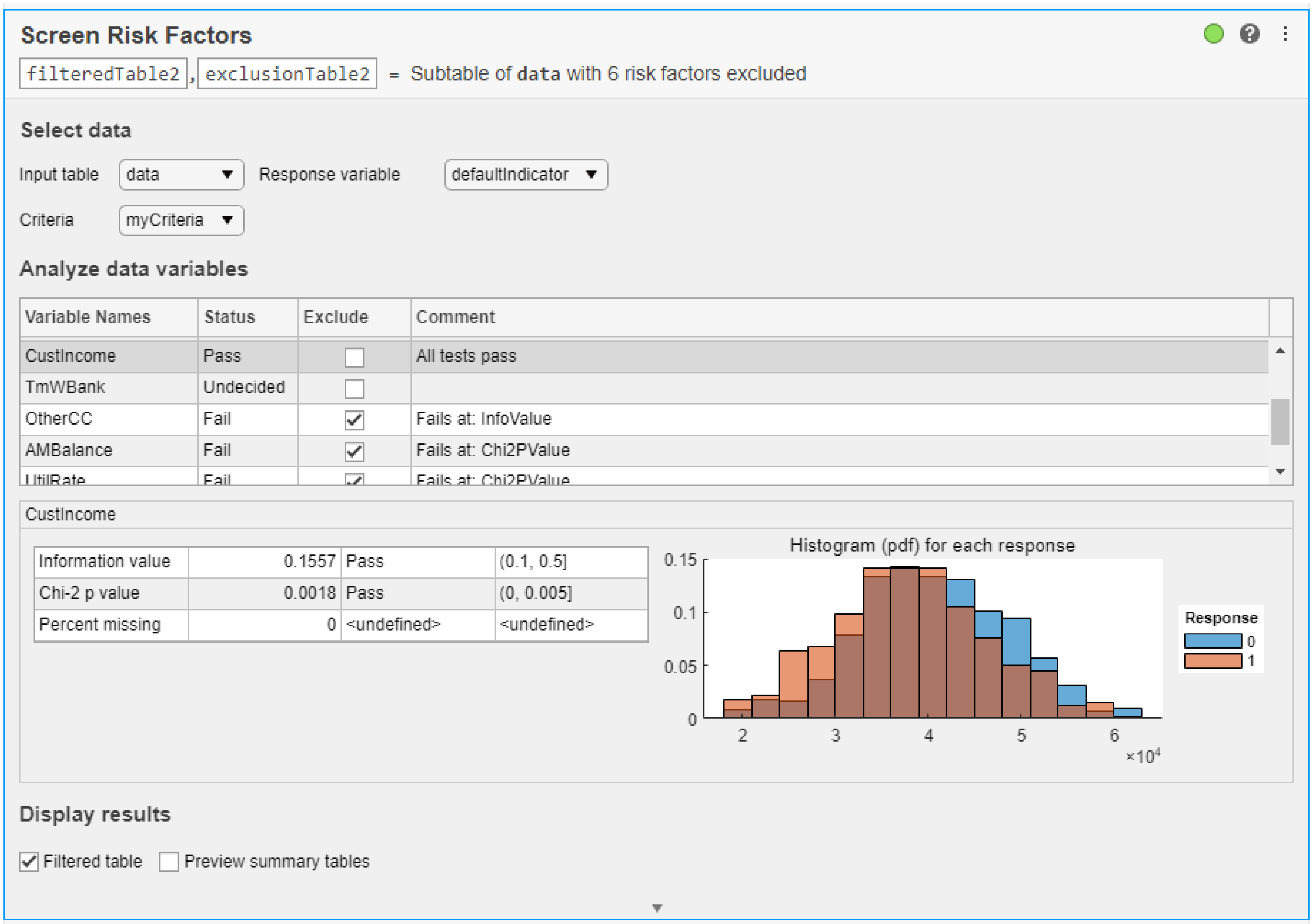
2) Search for Screen Risk Factors in the Live Task gallery

The task opens in a reduced view until the required inputs are selected:

- **Input table** must be a table or a timetable; the drop-down shows all such objects in the workspace. For this example, select `data`.
- **Response variable** drop-down shows all the binary variables in the input table. For this example, select `defaultIndicator`.
- **Criteria** should be the `ScreeningCriteria` object you wish to apply - in this case `myCriteria`.

Analyze and Remove Risk Factors

The task now expands.



The task calculates the screening metrics for each risk factor in the input table. The summary of the results is shown in the 'Analyze data variables' section. The table contains one row for each variable in the input table.

- 'Status' shows the overall classification of the variable based on the screening metrics.
- 'Exclude' shows whether the variable is to be removed from the data set.
- 'Comment' contains the reasons for excluding the variable, or for leaving the variable included.

The live task auto populates the 'Exclude' and 'Comment' columns based on the criteria. In this example, the 'Fails' are automatically excluded and 'Passes' are automatically included with automatically generated comments. The 'Undecided' risk factors are left blank for the user to analyze. You can overwrite these auto-completed values and sort the table according to any of these columns.

The area underneath the table is specific to the risk-factor variable and displays the screening metrics, as well as a double histogram that demonstrates how well (or not) the variable discriminates between the two possible responses. To switch the view to another variable, click the variable name in the table.

Document with Modelscape Reporting

The live task dynamically produces two outputs:

- `filteredTable`: This is a subtable of the input table without the excluded risk factors. Use this subtable in the next step of the model development process.
- `exclusionTable`: This table includes all the data of the input table together with the exclusion flags and comments in the Live Task. To view this information, tick the 'Preview summary tables' box in the 'Display results' section. This information is stored in `exclusionTable.Properties.CustomProperties` meta data.

```
exclusionSummaryPreview = 10×3 table
```

	Excluded	Variable Names	Comm...
1	0	"CustAge"	"All tests pass"
2	1	"TmAtAddress"	"Fails at: Chi...
3	1	"ResStatus"	"Fails at: Info...
4	1	"EmpStatus"	"Fails at: Info...
5	0	"CustIncome"	"All tests pass"
6	0	"TmWBank"	""
7	1	"OtherCC"	"Fails at: Info...
8	1	"AMBalance"	"Fails at: Chi...
9	1	"UtilRate"	"Fails at: Chi...

```
progressSummaryPreview = 4×2 table
```

		Risk factor count	% of Total
1	All variables	10	100
2	Excluded	6	60
3	Included	4	40
4	Commented	9	90

You can insert the above tables into model documentation using the Modelscape Reporting feature. To achieve this, create document holes with titles, say `ExclusionSummary` and `ProgressSummary`, in the Word document.

To create document holes in a Word document, view the Developer tab, and click the 'Rich Text Content Control' symbol **Aa** in the Controls area. Then click 'Properties', and fill in the Title fields.

```
import mrm.data.filter.*
[ExclusionSummary, ProgressSummary] = summarizeExclusionTable(exclusionTable)
```

After you have created holes, pick up the new variables from the MATLAB workspace and insert them into the model document using `fillReportFromWorkspace`.

For examples of creating document holes and for more details on the use of `fillReportFromWorkspace`, see “Model Documentation in Modelscape” on page 5-35.

Set Up Custom Criteria

To learn about test metrics, thresholds, and handlers used by screening criteria object, refer to the “Test Metrics in Modelscape” on page 5-85 and “Metrics Handlers” on page 5-45 examples.

You can customize the criteria used to screen variables in the Screen Risk Factors Live Task. The criteria must be in an `mrm.data.selection.ScreeningCriteria` object. For the class definition, run:

```
edit mrm.data.selection.ScreeningCriteria
```

This class is a holder for a handle to a function f.

```
f(inputData, 'PredictorVar', varName, 'ResponseVar', respVar)
```

The function `f` call must be well-defined and produce an `mrm.data.validation.MetricsHandler` object for any table or timetable `inputData`, any predictor variable `varName`, and for a given binary response variable `respVar`. `TestSuiteFactory` has this signature for the function call.

To see examples of these functions in the Modelscape package, run

```
edit mrm.data.selection.ExampleScreeningCriteria;
edit mrm.data.selection.TestSuiteFactory;
edit mrm.data.selection.overallScreeningStatus;
```

Model Documentation in Modelscape

This example shows how to add content to a Microsoft® Word document from MATLAB®.

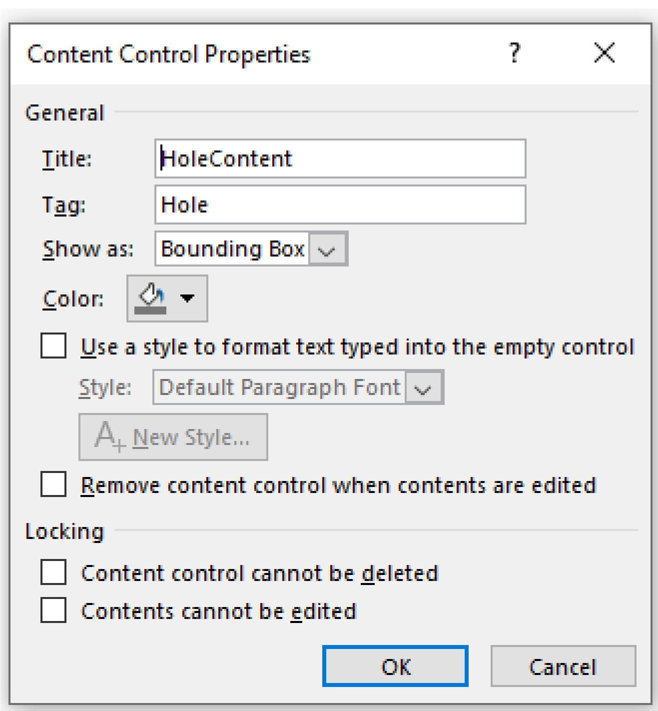
Many workflows in financial institutions involve writing and submitting reports to internal control functions or regulatory bodies. These documents often conform to a given house style and are typically Microsoft Word documents.

This example shows how to create a link from the MATLAB model development environment to the Word document. This enables the authors to push text, visualizations, and tables from MATLAB to the Microsoft Word document. This removes the need for error-prone processes involving screenshots and copy-pasting and ensures that the Word document contents are always in a consistent state. Authors can go back and forth between Microsoft Word and MATLAB, adding new text at the Microsoft Word side and refreshing the MATLAB contents as needed.

This example begins with a simple case of inserting a single MATLAB variable from the workspace to a Microsoft Word document. This example then describes the workflow you can use to insert more content from MATLAB to a Microsoft Word document.

Insert a single MATLAB Variable from Workspace to Word Document

Open a blank Microsoft Word document and add some text such as a title to it. Create a placeholder for your MATLAB content to the document - in MATLAB Report Generator terminology these are called *holes*. To do this, on the Developer tab, click the 'Rich Text Content Control' symbol **Aa** in the Controls area. Then click 'Properties' and fill in the Title and Tag fields as follows:



The 'Title' you have entered here will be the identifier of the hole - different holes will in general have different identifiers. By contrast, the tag tells MATLAB this is a placeholder it should fill in. The tag should always be 'Hole' as shown above. Your document should now look something like this:

Save your document to, say, `myTestDocument.docx`. Open MATLAB and navigate to the folder with the document.

Create the content you want to fill in, for example:

```
HoleContent = datetime("now");
```

The variable name must match the Title you chose earlier.

Preview how the content will appear in the document by calling `fillReportFromWorkspace.`

```
myDoc = "myTestDocument.docx";
previewDoc = fillReportFromWorkspace(myDoc);
winopen(previewDoc);
```

Calling `fillReportFromWorkspace` pushes the datetime content to a temporary document whose name is stored in `previewDoc`. Calling `winopen` opens this document.

If you want the datetime to be formatted in a different way, you can recreate the hole content.

```
HoleContent = datetime("now", "Format", "dd-MMM-uuuu");
```

Rerun the `fillReportFromWorkspace` and `winopen` calls. You should then see only the date in the Word document, not the hours, minutes, and seconds.

`previewDoc` is intended as a preview which you can discard later. To modify your original file `myTestDocument.docx`, close the main `myTestDocument` file and run:

```
fillReportFromWorkspace(myDoc, "OutputMode","Publish");
winopen(myDoc);
```

The date now appears in the main document. The `HoleContent` placeholder in your document remains refillable even after this operation. Running `fillReportFromWorkspace` in the `Publish` mode refreshes every time you fill the document. You will however need to close the document in Word every time you do this.

If `fillReportFromWorkspace` cannot find the required MATLAB content for a hole, it will insert the text "*Place-holder for hole id 'HoleContent'*" in red boldface font to highlight this to the user. Similarly, other formatting errors show different messages depending on the error.

Use the Model Documentation Workflow

Use the following workflow to document in Modelscape™ Reporting.

1. Create Word document.
2. Edit Word document and add holes for the contents to be filled in from MATLAB.
3. Create MATLAB contents.
4. Fill them into a preview document using `fillReportFromWorkspace` (in the `Preview` mode).
5. Repeat steps 2-4 as required.
6. Fill the MATLAB contents to the main document using `fillReportFromWorkspace` in the `Publish` mode.

Use Optional fillReportFromWorkspace Arguments

You can use the following optional arguments with `fillReportFromWorkspace`.

- `OutputMode`: '`Preview`' or '`Publish`'. '`Preview`' fills the document into a copy of the input document, whereas '`Publish`' overwrites the input document. The default option is '`Preview`'.
- `PresavedContentsFiles`: an array of strings giving the names of .mat files. `fillReportFromWorkspace` searches for the MATLAB content to insert first from these files, then from the workspace. This argument is intended to be used primarily in multi-author projects - see below.
- `NewContentsFile`: a string specifying the name of a .mat file. Inserted MATLAB contents will be saved to this file. If `PresavedContentsFiles` are provided as well, only the contents found in the workspace and not in the presaved mat files will be saved. This argument is primarily for multi-author projects.
- `MappingRules`: `FillReportMappings` object specifying the formatter mapping rules - see below for details.
- `Options`: `FillReportOptions` object specifying any overrides to formatter defaults - see below for details.

Use Supported MATLAB Content Types

This section shows you what MATLAB contents are supported and how to format them when filling the holes in a Word document.

There are two types of holes: *inline* and *block* holes. Inline holes insert contents within paragraphs (text, scalar numbers, ...) whereas block holes are for contents that require their own paragraph (such as figures and tables). You can insert some contents only into block holes

Preview Content

Use function `previewContent` to check how a MATLAB variable appears in a document.

```
previewDoc = previewContent(datetime("now"));
```

This function creates and opens a document to which the input content has been added. The resulting document is saved in a file with name `Preview-xyz.docx`, where `xyz` is the type of input content. You may delete this preview document after viewing it.

Insert Basic MATLAB Content Types

The appearance of MATLAB contents in a Word document is governed by helper classes called *formatters*. The table below lists these formatters and describes how they work for the supported core MATLAB content types.

Type	Inline supported	Formatter	Description
string char array	Yes	mrm.reporting.format.Text	Display text.
numeric (scalar)	Yes	mrm.reporting.format.Scalar	Display as text, integers with no decimal places, otherwise by default to 4 decimal places.
logical	Yes	mrm.reporting.format.Logical	Display logical true as text true, logical false as text false.
categorical	Yes	mrm.reporting.format.Categorical	Display the string corresponding to the categorical.
datetime	Yes	mrm.reporting.format.Datetime	Display as text, respecting the datetime formatting.
duration	Yes	mrm.reporting.format.Duration	Display as text, respecting the duration formatting.
table	No	mrm.reporting.format.FormalTable	Customized mlreportgen.dom.FormalTable object; see help mrm.reporting.format.FormalTable for details.
timetable	No	mrm.reporting.format.FormalTable	Use timetable2table and format as table
figure	No	mrm.reporting.format.Figure	See below for the workflow to capture figures correctly. Uses by default svg snapshot format.

To insert a plot or other kind of a figure created in a Live Script, use the following MATLAB commands.

```
FigureContent = figure(); % "open" figure
plot(rand(10)); % do the work to plot the figure
figure(); % mark the work on FigureContent as finished
```

Insert MATLAB Report Generator Types

If you are familiar with MATLAB Report Generator, you can bypass the Modelscape formatters by wrapping MATLAB content into `mlreportgen.dom` or `mlreportgen.report` objects. The supported types are listed below. Note however that these types are not guaranteed to support the `PresavedContentFiles` and `NewContentsFile` arguments used in multi-author projects, as the Report Generator objects do not, in general, allow saving and loading into MAT files.

Report Generator DOM Types

The following types are supported. The type names are shortened - for class definitions, see `mlreportgen.dom.Text`, `mlreportgen.dom.Paragraph` and so on.

Type	Inline supported
Text	Yes
Paragraph	No
Preformatted	No
Image	No
MATLABTable	No
OrderedList	No
UnorderedList	No
ExternalLink	Yes
HTML	No
HTMLFile	No

Report Generator Reporter Types

The following types are supported. Again, the type names are shortened - see `mlreportgen.report.Equation` and so on for the class definitions.

Type	Inline supported
Equation	No
Figure	No
FormalImage	No
MATLABVariable	No

Insert Composite Types

Modelscape Reporting automatically handles certain "composite" types such as arrays of numerical data (or strings, or logicals, or categoricals,...) and cell arrays of mixed-type data. The logic is the same:

1. Loop through the composite structure and format each element (double, string, logical, etc.) as explained above.

2 Display this table of formatted cells.

To see an example, run the following command (and delete the resulting Preview-cell.docx file afterwards).

```
previewDoc = previewContent({pi, true, 3; "abc", datetime("now"), hours(1)});
```

Insert File Contents

You can also insert the contents of certain types of files directly from files to the target document without loading them first into the MATLAB workspace using Modelscape Reporting. In this case the MATLAB workspace must contain a file content object for each inserted file, and the name of this object must match the title of the hole to be filled.

Construct the file content objects using a `fileContent` helper function. For example, insert the contents of `myTable.csv` as a table to a hole titled `TableFromFile` and then use `fillReportFromWorkspace`. .

```
TableFromFile = fileContent('myTable.csv');
```

`fileContent` helper will try to infer the type of the file from its extension. The following file types and recognized extensions are supported.

- `image`: all the extensions are recognized by `mlreportgen.dom.Image` including png, svg and jpg.
- `table`: csv, xml, xls* and the common spreadsheet variants.
- `text`: txt and log.

You can explicitly specifiy the file type for unrecognized extensions.

```
TableOfErrors = fileContent('errorLogs.err', 'table');
```

You can also pass arguments to `fileContent` that will be understood by the returned file content class. Currently this is supported for table contents, for which `fileContent` accepts the parameters used by `readtable.`

```
TableFromFile = fileContent('myTable.csv', 'table', 'ReadRowNames', false);
```

Supply the file content type when you use these extra arguments.

Using Custom File Contents

You can create your own file content classes. To learn more, contact MathWorks Consulting Services.

Format Controls - Individual Contents

Usually, the document hole title names match the name of the MATLAB variables to be filled in. You can control how this content appears in the Word document.

1 You can overwrite the formatter that is used for the given content.
2 You can pass extra control parameters to the formatter (be it the default formatter or a custom one).

To use these controls, use a hole title should be of the form `ClassName(ContentName, Label1, Value1, Label2, Value2,...)`, where

- `ClassName` is the name of the formatter class or function that you use (or the word `Default` in which case no override is used).
- `ContentName` is the name of the MATLAB workspace variable defining the content.
- `Label1, Value1, ...` are name-value pairs that you pass to the formatter.

The rules for looking up the overriding class are as follows:

- In decreasing order of priority, either the `ClassName` such as `mrm.reporting.format.Scalar,` or the `mrm.reporting.customize.ClassName,` or `mrm.reporting.format.ClassName,` is used.
- Otherwise, a placeholder with a warning that no class definition was found is inserted.

Example: Using `mrm.reporting.customize.Exp` for Exponential Notation

Modelscape Reporting has an example custom class `mrm.reporting.customize.Exp` that you can use for displaying numerical data in the exponential notation. This class supports an argument called `ExpPrecision` that controls how many digits are shown after the decimal point (the default being 3).

Suppose you want to display a MATLAB variable `MyNumber` that carries the number 123456.54321. Using a Word document hole title `MyNumber` for this placeholder, this content appears in a Word document as 123456.5432. To override this, rename the document hole.

- `Exp(MyNumber)` - the number appears as 1.235e+05.
- `Exp(MyNumber,"ExpPrecision",1)` - the number appears as 1.2e+05.

The name-value pairs can also be properties of the underlying Report Generator class. For example, `Exp(MyNumber, "Color", "blue")` shows the text 1.2345e+05 in blue font in the Word document.

Format Controls at Document Level

Formatting contents is not practical if you want to change the number of decimal places for *every* numeric variable inserted into a document. To apply a change to every hole in the given document, use a `FillReportOptions` object for these options, and pass the object to `fillReportFromWorkspace` using the 'Options' label. However, note that hole-level arguments take precedence over the top-level options.

The following control options are available.

- `ImageSnapshotFormat`: option used by Modelscape Figure formatter to control the format of the image taken of plots and other figures - defaults to svg.
- `MaxNumericPrecision`: number of decimal places shown in (non-integer) numeric variables - defaults to 4.
- `PlaceHolderColor`: colour of the text placed into document holes to alert user of missing MATLAB contents etc - defaults to crimson.
- `TableDisplayUnits`: option used by MRM FormalTable formatter to decide whether units are displayed in the formatted table - defaults to false.

You can add any name-value pairs to the options object to pass to `fillReportFromWorkspace`, to be picked up by the appropriate classes. For example, change the `ExpPrecision` parameter in all numerical contents formatted in the exponential notation.

```
myOptions = FillReportOptions("ExpPrecision", 1);
```

```
previewDoc = fillReportFromWorkspace(myDocument, "Options", myOptions);
```

In the setup of the example in the previous section, the content 123456.54321 placed into a hole titled as `Exp(MyNumber)` will then appear as 1.2e+05 and not as 1.235e+05.

Map Overrides

You can use properties of Report Generator DOM objects at the document level. These will be picked up by the respective formatters. For example, adding the argument `Color` with value `blue` to the options makes the text in all the contents to appear in blue - this includes doubles, logicals, categoricals, strings and so on. However, this must be used with caution. For example, `StyleName` makes sense for at least table and text formatters, but the value `Grid Table 2` will only be understood by tables. For these kinds of changes, it is better to use mapping overrides and custom classes.

To map overrides, use the `FillReportMappings` objects.

```
FillReportMappings
```

```
ans =
    "categorical"         "mrm.reporting.format.Categorical"
    "char"                "mrm.reporting.format.Text"
    "datetime"            "mrm.reporting.format.Datetime"
    "duration"            "mrm.reporting.format.Duration"
    "logical"             "mrm.reporting.format.Logical"
    "numeric"             "mrm.reporting.format.Scalar"
    "string"              "mrm.reporting.format.Text"
    "table"               "mrm.reporting.format.FormalTable"
    "timetable"           "mrm.reporting.format.FormalTable"
    "matlab.ui.Figure"    "mrm.reporting.format.Figure"
```

`FillReportMappings` objects contain two columns that should match the table shown above in the section "Basic MATLAB Content Types". You can override these mappings, add new mappings, and pass the customized mapping object to `fillReportFromWorkspace` using the `MappingRules` argument.

For example, Modelscape Reporting has a `CheckBox` formatter to display logical variables as ticked or unticked checkboxes (and not as strings 'true'/'false'). To use this formatter for all logicals in a document, you can write the following:

```
myMappings = FillReportMappings("logical", "mrm.reporting.customize.CheckBox");
previewDoc = fillReportFromWorkspace(myDocument, "MappingRules", myMappings);
```

Similarly, you can add mapping data for in-house data types that are not directly supported by Modelscape Reporting. For example, if your model development process involves a custom class `EnrichedTable` for which you have written a custom formatter `mrm.reporting.customize.MyTableFormatter` (see next section), then you can add this to the mapping rules

```
myMappings = FillReportMappings("EnrichedTable", "mrm.reporting.customize.MyTableFormatter");
```

and call `fillReportFromWorkspace` with this as the `MappingRules` argument, as above.

To find the applicable mapping, MRM Reporting runs the test `isa(content, t)` for all the types `t` that show up in the left-hand column of the `FillReportMappings` display. The applicable formatter is the one corresponding to the last type for which this returns true; if no mapping is found, a placeholder with that message is inserted into the document.

Customize Formatters

Writing new formatters may be necessary for at least two reasons:

- Changing the style of an existing formatter in a way that is not controllable through `FillReportOptions`.
- Creating a formatter for a new content type.

Place all custom formatters in a `mrm.reporting.customize` package - that is, in a `+mrm\+reporting\+customize\` folder on the MATLAB path. The interface requirements for a formatter class are as follows:

- The formatter should be a subclass either `mlreportgen.dom.Element` or `mlreportgen.report.Reporter` - in practice this will mean one of the classes listed in the section "MATLAB Report Generator Types" above.
- The formatter constructor should take as inputs `content` (the MATLAB variable to be formatted), `rpt` (an `mlreportgen.report.Report` object) and `options` (a `FillReportOptions` object) - in this order.

Alternatively, the formatter can be a function that returns either an `mlreportgen.dom.Element` or an `mlreportgen.report.Reporter` object and has the same signature as the class constructor described above.

For example, to apply Word table style `Grid Table 2` to all tables in a safe way, create the following custom formatter.

```
classdef MyTableFormatter < mrm.reporting.format.FormalTable

    methods
        function this = MyTableFormatter(content, report, options)
            arguments
                content table
                report(1,1) mlreportgen.report.Report = mlreportgen.report.Report()
                options(1,1) FillReportOptions = FillReportOptions()
            end
            this@mrm.reporting.format.FormalTable(content, report, options);
            this.StyleName = "Grid Table 2";
            mrm.reporting.internal.setDOMOptions(this, options);
        end
    end
end
```

Use this for all tables.

```
myMappings = FillReportMappings('table','mrm.reporting.customize.MyTableFormatter');
previewDoc = fillReportFromWorkspace('TestDoc.docx', 'MappingRules', myMappings);
winopen(previewDoc);
```

Note that if you want to run this example, you must ensure that the Grid Table 2 style is present in your test document. To do this, create and click an empty table, hit Ctrl+Alt+S and select Grid Table 2 from the dropdown menu to apply the style. Then save the document. You can now delete your empty table. The style definition will remain in the docx file.

Work on Multi-Author Projects

Several authors can co-operate on a documentation project simultaneously by following a slight modification of the reporting workflow explained above. Most of the work (steps 2-6) can be done by the authors in parallel; only the document creation and building the final version (steps 1, 7 and 8) should be coordinated and carried out by a lead author or similar. To avoid clashes between the hole names created by different authors, it may be advantageous to use a naming convention such as prefixing the hole names by the author's initials.

1. Create a Word document in a shared location that allows simultaneous editing, for example SharePoint.
2. Edit the Word document as before, adding holes for the contents to be filled in from MATLAB.
3. Create MATLAB contents.
4. Fill them into the Word document using `fillReportFromWorkspace` in the Preview mode and check that contents are correctly displayed. The authors may also use the `PresavedContentsFiles` argument to include any MATLAB contents the other users may have already made available.
5. Repeat steps 2-4 as required.
6. Fill in the MATLAB contents using `fillReportFromWorkspace` still in Preview mode but using the `NewContentsFile` argument to save the MATLAB contents you are contributing to the document. Store the resulting mat file next to the Word document in the shared location.
7. When all authors have contributed their document contents into different mat files, run `fillReportFromWorkspace`, passing in all the mat files through the `PresavedContentsFiles` argument. Check that all the contents are correctly displayed.
8. Run `fillReportFromWorkspace`, this time in `Publish` mode, again with all the mat files listed as the `PresavedContentsFiles` argument.

Note that authors are free to reference contents provided in each other's mat files.

Metrics Handlers

This example shows how to manage Modelscape™ test metrics and their associated threshold objects using `MetricsHandler` objects.

`MetricsHandler` produces reports that summarize the metrics and the status of the metrics in the container relative to their thresholds.

For more information about test metrics and thresholds, see “Credit Scorecard Validation Metrics” on page 5-48 and “Fairness Metrics in Modelscape” on page 5-26. To learn how to write your own metrics, see “Test Metrics in Modelscape” on page 5-85.

This example shows you how to set up some metrics and thresholds for mock data of a credit scoring model. This example creates a metrics handler object to visualize the metrics and summarize the results. It shows you how to set an overall status to the handler based on various metrics.

Set Up Test Metrics and Thresholds

Use the following random data as an example of training response data (`defaultIndicators`) and model predictions (`scores`).

```
rng('default');
scores = rand(1000,1);
defaultIndicators = double(scores + rand(1000,1) < 1);
```

Create three metrics:

1) Area under the receiver operating characteristic curve (AUROC),

2) Cumulative accuracy profile (CAP Accuracy) ratio, and

3) Kolmogorov-Smirnov statistic.

For AUROC and CAP Accuracy ratio, set values greater than 0.8 as Pass, values less than 0.7 as Fail, and values between these as Undecided, requiring further inspection. Set no thresholds for the Kolmogorov-Smirnov statistic.

```
import mrm.data.validation.TestThresholds
import mrm.data.validation.pd.*

auroc = AUROC(defaultIndicators, scores);
aurocThresholds = TestThresholds([0.7, 0.8], ["Fail", "Undecided", "Pass"]);

cap = CAPAccuracyRatio(defaultIndicators, scores);
capThresholds = TestThresholds([0.6, 0.7], ["Fail", "Undecided", "Pass"]);

ks = KSStatistic(defaultIndicators, scores);
```

Add metrics to a Metrics Handler Object

Add the metrics created in the previous section to a `MetricsHandler` object.

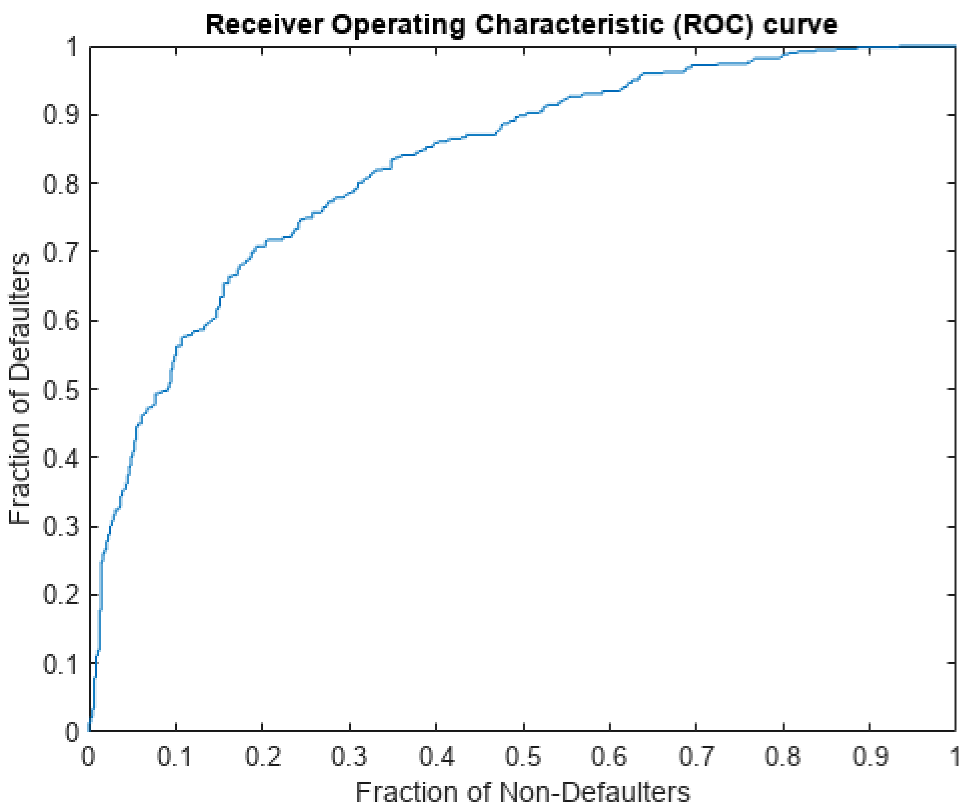
```
import mrm.data.validation.MetricsHandler
mh = MetricsHandler();
append(mh, auroc, aurocThresholds);
```

```
append(mh, cap, capThresholds);
append(mh, ks);
disp(mh)
```

```
  MetricsHandler with properties:

      CAP: [1x1 mrm.data.validation.pd.CAPAccuracyRatio]
       KS: [1x1 mrm.data.validation.pd.KSStatistic]
    AUROC: [1x1 mrm.data.validation.pd.AUROC]
```

The handler contains these three metrics that can be accessed as properties of this handler object. This allows you to access constituent metrics' diagnostics and visualizations.

```
visualize(mh.AUROC);
```



Interrogate Metrics Handlers

Use the `report` method to view the performance of the model relative to the given metrics.

```
summaryTable = report(mh);
disp(summaryTable)
```

Metric	Value	Status	Diagnostic
Area under ROC curve	0.82905	Pass	(0.8, Inf)
Accuracy ratio	0.65809	Undecided	(0.6, 0.7]
Kolmogorov-Smirnov statistic	0.51462	<undefined>	<undefined>

The model performs well on AUROC, whereas the "Undecided" status on the Accuracy Ratio suggests the model requires a closer look.

When the handler carries complex non-scalar metrics, use arguments `Keys` and `Metrics` arguments with `report`. For more information, see “Fairness Metrics in Modelscape” on page 5-26.

Set Overall Status to the Handler

For a handler with many metrics, set an overall status to the handler by associating a 'status interpreter' to the handler. This section shows how to use an interpreter supplied with Modelscape that is compatible with your threshold objects.

```
mh.StatusInterpreter = @mrm.data.validation.overallStatus;
summaryTable = report(mh);
disp(summaryTable)
```

```
                Metric                   Value        Status        Diagnostic
    ____________________________      _______    ___________    ___________

    Area under ROC curve              0.82905    Pass           (0.8, Inf)
    Accuracy ratio                    0.65809    Undecided      (0.6, 0.7]
    Kolmogorov-Smirnov statistic      0.51462    <undefined>    <undefined>
    Overall                               NaN    Undecided      <undefined>
```

The overall status is in general decided based on the status descriptions of the individual metrics. In the above case, the overall status is the 'worst' of the individual statuses - 'Undecided'.

Thresholding systems with other descriptive strings - for example "Red", "Amber", "Green" require a custom status interpreter to be implemented. To do this, see the instructions before the StatusInterpreter declaration in the MetricsHandler implementation.

```
edit mrm.data.validation.MetricsHandler
```

Alternatively, modify the interpreter as required.

```
edit mrm.data.validation.overallStatus
```

You can also set the StatusInterpreter for the handler immediately at construction:

```
mh2 = MetricsHandler('StatusInterpreter', @mrm.data.validation.overallStatus)
```

Credit Scorecard Validation Metrics

This example shows how to implement a Probability of Default (PD) model validation suite covering the techniques laid out in the BCBS Working Paper 14.

The techniques apply to any PD model. This example loads the test data, sets and computes some key quantities, and calculates various model validation metrics. The results are stored in a collection object.

```
MH = mrm.data.validation.MetricsHandler;
```

This example uses `PDModelValidation` data set which consists of a pre-fitted MATLAB® credit scorecard object. This example shows you how to compute various metrics and summarize them.

Prepare Data

Load the scorecard to validate.

```
modelDataFile = "PDModelValidation.mat";
inputData = load(modelDataFile);
```

Set the response variable and default indicator.

```
responseVar = 'status';
outcomeIndicator = 1;
```

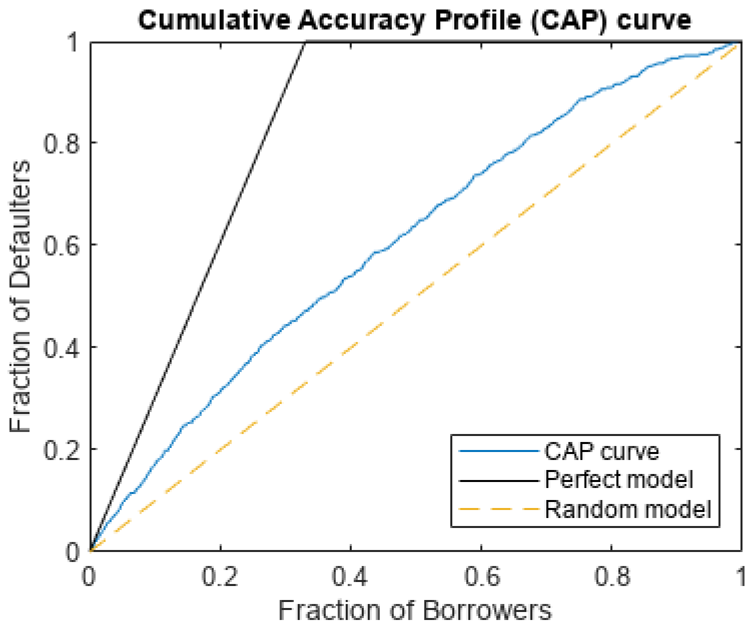
Extract key data and precompute certain key quantities

```
sc = inputData.sc;
testData = sc.Data;
scores = sc.score;
defaultProbs = probdefault(sc);
defaultIndicators = testData.(responseVar) == outcomeIndicator;
```

Visualize Cumulative Accuracy Profile and Compute Accuracy Ratio

The Cumulative Accuracy Profile (CAP) curve plots the proportion of defaulting names against the proportion of all names in the test set as one goes through the scores from the lowest to the highest. The Accuracy Ratio (AR) measures how the model relates to hypothetical perfect and random models. AR of 1 indicates the model is perfect, whereas AR of 0 indicates the scoring model is no better than a random model.

```
CAPMetric = mrm.data.validation.pd.CAPAccuracyRatio(defaultIndicators, scores);
visualize(CAPMetric);
```




```
displayResult(CAPMetric);
```

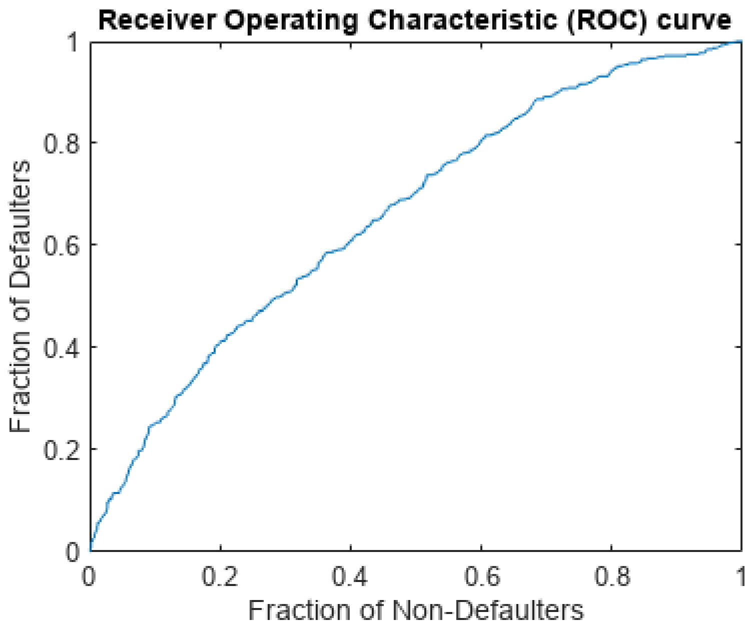
```
Accuracy ratio is 0.3223
```

```
append(MH, CAPMetric);
```

Compute and Visualize Receiver Operating Characteristic

The Receiver Operating Characteristic (ROC) curve plots the proportion of defaulting names against the proportion of non-defaulting names in the test set as the scores move from the lowest to the highest. The associated test metric is the area under the ROC curve (AUROC). AUROC of 1 indicates a perfect model.

```
ROCMetric = mrm.data.validation.pd.AUROC(defaultIndicators, scores);
visualize(ROCMetric);
```




```
displayResult(ROCMetric);
```

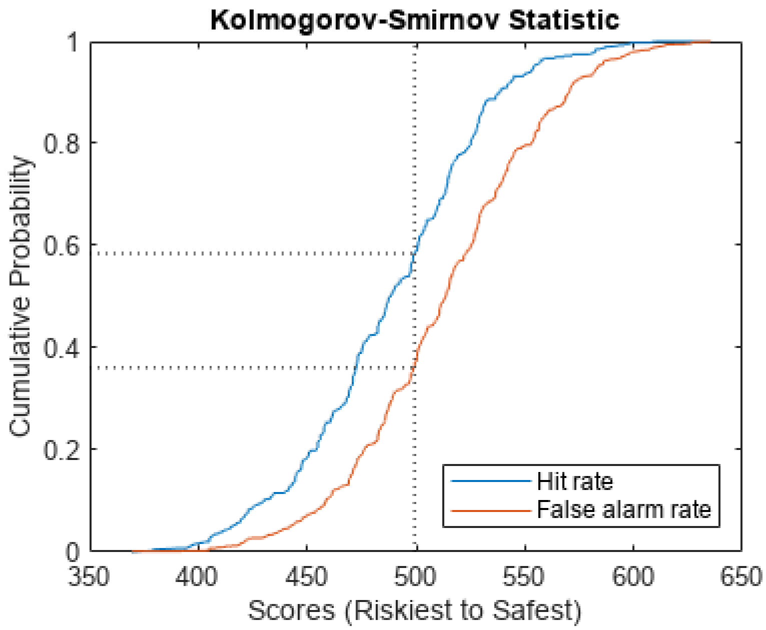
```
Area under ROC curve is 0.6611
```

```
append(MH, ROCMetric);
```

Compute and Visualize Kolmogorov-Smirnov Statistic

The Kolmogorov-Smirnov statistic indicates the maximal separation of defaulters from non-defaulters achieved by the model. The two plots in the visualization are the fractions of defaulters and non-defaulters in the test set; these plots are referred to as the Hit Rate and the False Alarm Rate in Working Paper 14.

```
KSMetric = mrm.data.validation.pd.KSStatistic(defaultIndicators, scores);
visualize(KSMetric);
```




```
displayResult(KSMetric);
```

```
Kolmogorov-Smirnov statistic is 0.2232, at score 499.2
```

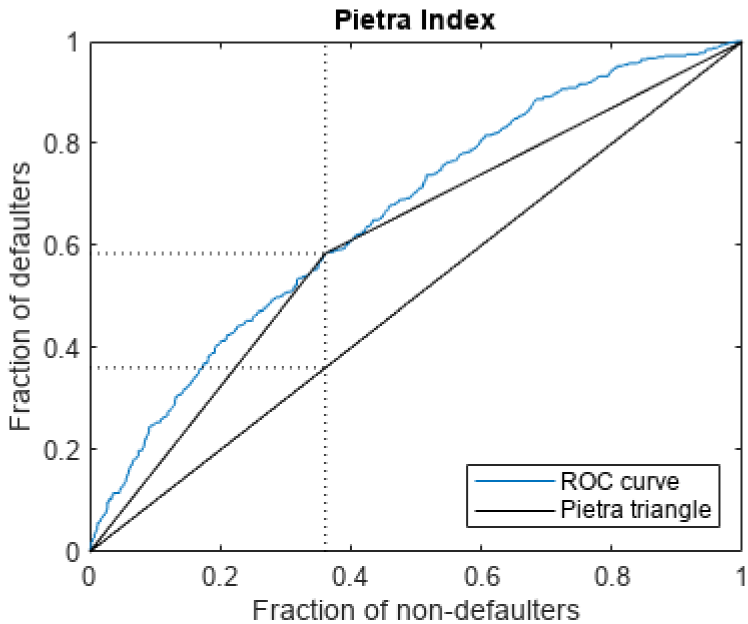
```
append(MH, KSMetric);
```

Compute and Visualize Pietra Index

Pietra Index is twice the maximal area of a triangle that can be fitted between the ROC curve of the model and the diagonal. Multiplication by 2 ensures that the range of possible values for the index is [0,1].

Calculating the Pietra Index assumes that the area under the ROC is convex. In this case the Pietra Index matches the Kolmogorov-Smirnov statistic calculated previously (indicated by the vertical dotted line in the plot).

```
PietraMetric = mrm.data.validation.pd.PietraIndex(defaultIndicators, scores);
visualize(PietraMetric);
```




```
displayResult(PietraMetric);
```

```
Pietra index is 0.2232
```

```
append(MH, PietraMetric);
```

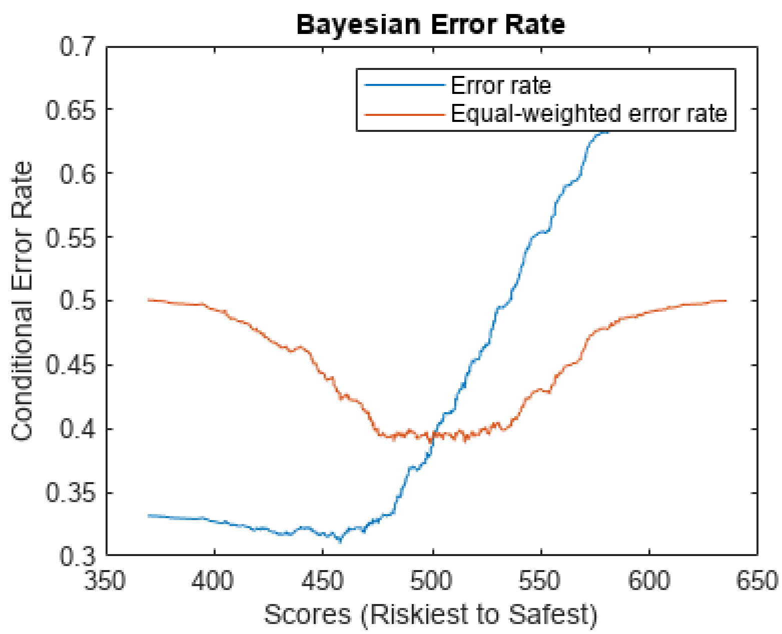
Compute and Visualize Bayesian Error Rate

Define the error rate at a given score as the weighted sum

$$ER(s) = p_D(1 - HR(s)) + (1 - p_D) * FAR(s)$$

where *pD* is the total proportion of defaulters in the test set, and *HR* and *FAR* are the Hit Rate and the False Alarm Rate functions, respectively. The Bayesian Error Rate of the model is the minimum error rate achieved in the test set. The error rate is sometimes calcualted with equal weights of 0.5 instead of p_D and $1 - p_D$; in this case the Bayesian error rate is seen to be equal to $\frac{1}{2}(1 - \mathrm{KS})$, where KS is the Kolmogorov-Smirnov statistic.

```
BERMetric = mrm.data.validation.pd.BayesianErrorRate(defaultIndicators, scores);
visualize(BERMetric);
```




```
displayResult(BERMetric);
```

	Value	Score
Bayesian error rate	0.31	457.9
Equal-weighted Bayesian error rate	0.38838	499.18

```
append(MH, BERMetric);
```

Calculate Mann-Whitney Statistic

The Mann-Whitney statistic is a rank sum metric that you calculate for the defaulter and non-defaulter populations of the test set. It recovers the area under the ROC curve calculated previously, and you can use the variance of the Mann-Whitney statistic to calculate confidence intervals for the AUROC statistic.

```
mwConfidenceLevel = 95% ;
MWMetric = mrm.data.validation.pd.MannWhitneyMetric(defaultIndicators, scores,...
    mwConfidenceLevel);
displayResult(MWMetric);
```

```
Mann-Whitney statistic 0.66113, with confidence interval (0.6606,0.66165) at level 0.95
```

```
append(MH, MWMetric);
```

Calculate Somers' D

To test the hypothesis that low scores correspond to high probability of default, calculate the Somers' D between the sorted scores and the default indicators (flipped, to preserve the ordering, so that survival is denoted by 1 and default by 0).

```
SDMetric = mrm.data.validation.pd.SomersDMetric(defaultIndicators, scores);
displayResult(SDMetric);
```

```
Somers' D is 0.3223
```

```
append(MH, SDMetric)
```

Display Brier Score

The Brier score is the squared mean error of the probabilities of default predicted by the model. Calculate the Brier score also for the "trivial model" which assigns the same probability to each name (equal to the fraction of defaulting names in the dataset).

```
BrierMetric = mrm.data.validation.pd.BrierTestMetric(defaultIndicators, defaultProbs);
displayResult(BrierMetric);
```

	Value
Brier score	0.20541
Brier score for trivial model	0.22138

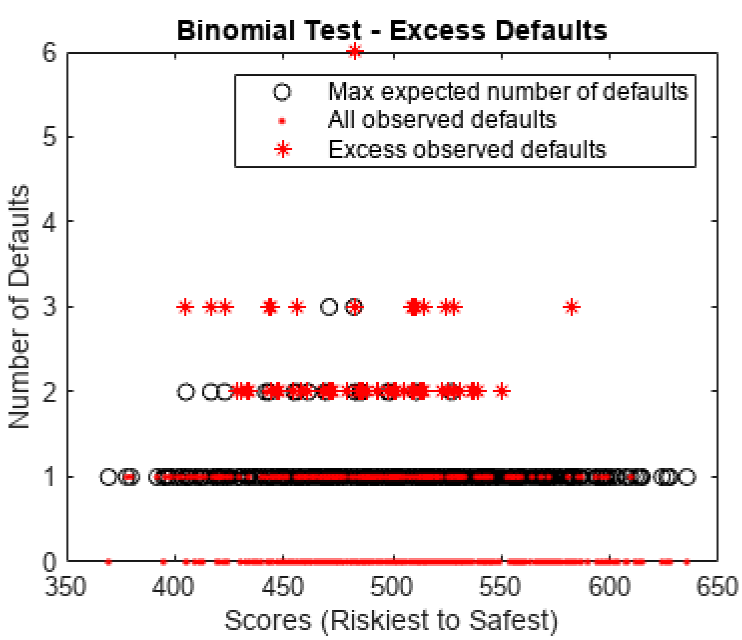
```
append(MH, BrierMetric);
```

Perform Binomial Test

The binomial test inspects each ratings category separately and tests whether the number of realized defaults is plausible given the predicted probability of default.

The confidence level is set using the quantity `q` below. For example, setting `q = 0.95` finds the scores where the number of observed defaults exceeds the number of defaults that would be expected with 95% confidence. These scores are highlighted in the visualization by an asterisk.

```
binConfidenceLevel = 95% ;
BinMetric = mrm.data.validation.pd.BinomialTest(defaultIndicators, defaultProbs, scores,...
    binConfidenceLevel);
visualize(BinMetric);
```



```
head(formatResult(BinMetric));
```

```
    Rating Category    Observed Defaults    Max Expected Number of Defaults
    _______________    _________________    _______________________________

         369.4                 0                           1               
        377.86                 1                           1               
        379.78                 1                           1               
        391.81                 1                           1               
        394.77                 0                           1               
        395.78                 1                           1               
        396.95                 1                           1               
        398.37                 1                           1               
```

```
append(MH, BinMetric);
```

Compute Hosmer-Lemeshow Metric

This test, also known as the chi-squared test, compares number of observed defaults for each ratings category with the number that would be expected from the probability of default predicted by the model. Unlike the results of the binomial test, these measures are combined into a single quantity, which for large datasets should converge to a χ_k^2-distribution, where k is the number of ratings categories in the model. The test statistic is the p-value of this χ_k^2-distribution.

```
HLMetric = mrm.data.validation.pd.HosmerLemeshowMetric(defaultIndicators, defaultProbs, scores);
displayResult(HLMetric);
```

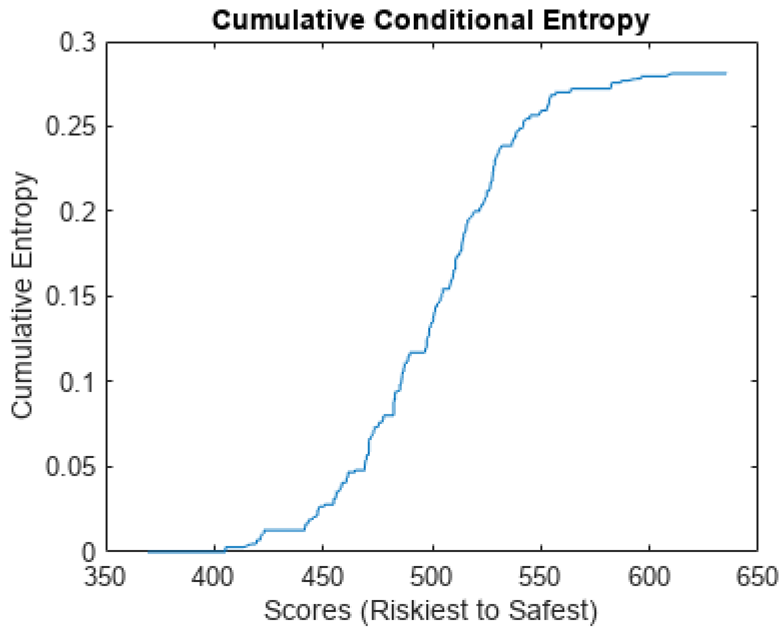
```
Hosmer-Lemeshow p-value is 0.9177
```

```
append(MH, HLMetric);
```

Compute and Visualize Entropy Measures

Conditional Information Entropy Ratio measures the capability of the model to separate the defaulters from non-defaulters. Unlike the previous metrics, it does not consider the ordering of the scores. CIER value close to zero indicates good separation of defaulting names from surviving names.

```
CIERMetric = mrm.data.validation.pd.EntropyMetric(defaultIndicators, scores);
visualize(CIERMetric);
```




```
displayResult(CIERMetric);
```

```
Conditional Information Entropy Ratio is 0.557
```

```
append(MH, CIERMetric);
```

The plot illustrates the cumulative sum of the entropies at each score weighted by the proportion of names with a given score. jumps indicate points where the model struggles to differentiate between defaulters and non-defaulters.

Display Summary

```
disp(report(MH))
```

```
                   Metric                       Value
    _______________________________________    ________

    "Accuracy ratio"                            0.32225
    "Area under ROC curve"                      0.66113
    "Kolmogorov-Smirnov statistic"              0.22324
    "Pietra index"                              0.22324
    "Bayesian error rate"                          0.31
    "Mann-Whitney u-test"                       0.66113
    "Somers' D"                                 0.32225
    "Brier score"                               0.20541
    "Binomial test failure rate"               0.087667
    "Hosmer-Lemeshow p-value"                   0.91772
    "Conditional Information Entropy Ratio"     0.55697
```

Validation of Credit Models in ECB Templates

This example shows how to fill in the European Central Bank (ECB) model validation templates using Modelscape™ in MATLAB®.

ECB has published a suite of model validation templates covering a wide array of credit models, including Probability of Default (PD) models. Find the ECB templates at this address under 'Related Documents'. Find the instructions for filling in the templates in the document Instructions for reporting the validation results of internal models - IRB Pillar I models for credit risk.

Fill in ECB Template Input Data Using Modelscape

The ECB validation templates require three kinds of inputs:

1. Model and organization metadata - for example, model and institution names, and the start and end dates of the review period
2. Directly observable quantitative data - for example, the number of customers who defaulted during the observation period
3. Model performance metrics - for example, metrics that measure the discriminatory power of the model, or credit rating migrations

Use Modelscape function `fillValidationTemplate` to fill in the last two types of inputs. You can then fill in the model names, identifiers, and other variables in Excel.

```
fillValidationTemplate(fileName, ...
      "InitialValidationData", initialValidationData, ...
      "InitialPortfolioData", reviewStartPortfolioData, ...
      "TerminalPortfolioData", reviewEndPortfolioData);
```

- `fileName` is the name of the spreadsheet you fill in. Use the standard format required by ECB: "[LEICode]_[ModelType]_[ModelID]_[ReferenceDate]_[VersionNumber].xlsx”. For more details, see the final paragraph of Section 2.2 of the ECB instructions.
- `reviewStartPortfolioData` and `reviewEndPortfolioData` are `mrm.data.validation.reporting.pd.OperationalData` objects with information about the portfolio at the beginning and the end of the *current* review period. Examples include the model PDs, exposures, and ratings for the customer populations, as well as flags to indicate credit transferals and overrides within the populations.
- `initialValidationData` is an `mrm.data.validation.reporting.pd.ValidationData` object with the information used in the *initial* validation of the model, potentially many years before the current review period. Examples of such information include the original model PDs and default indicators for some validation data set.

Set Operational Data

The templates require operational data as a table containing various data with one item per customer or entity, together with the observation date and flags such as whether the model allows technical defaults. In this section, the term *name* indicates a single obligor such as a customer.

Supply the following data as arrays of equal lengths:

1. `IDs`: a string identifier for each name in the data set

2 `PDs`: the probabilities of default predicted by the model
3 `Ratings`: the ratings assigned to the customers (as *ordered categorical* variables - see below for more details)
4 `Exposures`: the current exposure for each name

- Store **Ratings** as categorical variables with a fixed *ascending* order. Otherwise, certain data quality checks built into the ECB templates will fail. Define the categorical variables in MATLAB using the following syntax:

```
ratingLabels = categorical(["CCC", "B", "BB", "BBB", "A", "AA", "AAA"], ...
    ["CCC", "B", "BB", "BBB", "A", "AA", "AAA"], 'Ordinal', true);
```

- While the ECB instructions do not specify a choice of currency, express the **exposures** in a single reference currency. This is because various validation measures require calculating the total exposures within some classes of names.

In addition to the four vectors, the operational data must contain the following arrays of logicals of the same size as the identifier arrays. These indicate anomalous states of the names and are optional. The `OperationalData` class will default these values to false if you do not specify them.

The flags determine the correct in-scope sample, so it is important that you set them correctly.

1 `InDefault`: set to True if the name currently in default (bullet point 2.3 (c) of the ECB instructions).
2 `TechnicalDefaultIndicators`: set to True if the name is in *technical* default (2.3 (d)).
3 `OtherModel`: set to True if a different model is used for the name for rating or regulatory capital purposes (2.3 (e)).
4 `NonDefaultablePositive`: set to True if the name is non-defaultable even if it has a positive exposure (p. 10, footnote 18).
5 `Deficient`: set to True if the name should be excluded from the sample due to process deficiency (2.5.1 (c)).
6 `Outdated`: Does the name have an outdated rating or financial statements (2.5.1 (d)).
7 `Overridden`: set to True if the rating has been overridden (2.5.1 (e)).
8 `Transferred`: set to True if the name had been transferred to it the rating of some third party (2.5.1 (f)).

Finally, the `OperationalData` class should contain the following:

- `OutdatedAllowed`, `TransferralAllowed`, `OverridesAllowed`, `TechnicalDefaultsAllowed`: Single logical variables to indicate whether the model and assignment process allow for outdated accounts, rating transferals, overrides, or technical defaults (PD validation template, Sheet 2.0).
- `Date`: The 'as-of' date for the above data.

In practice, it may be convenient to save the one-datum-per-name data as a table and supply the final four logicals and the as-of date separately.

Set Validation Data

Operational data objects contain snapshots of the model predictions and other relevant data at given time points. Validation data objects, on the other hand, carry information about how the status of the names changes over time. Validation metrics assess how these changes relate to the PDs predicted by the model.

Each `ValidationData` object carries the following data.

- `ObsStartDate` and `ObsEndDate`: initial and the terminal dates of the observation period.
- `IDs`: string identifier for each name in the data set.
- `PDs`: model PDs as predicted at `ObsStartDate`.
- `Exposures`: exposures of the names in some appropriate reference currency, as of `ObsStartDate.`
- `DefaultIndicators`: logical flags indicating the names that have defaulted *during* the observation period.
- `InitialRatings` and `TerminalRatings`: rating grades predicted by the model at the beginning and the end of the observation period (as *ordered categorical* variables). The terminal ratings should have an ascending order again.

In addition, `ValidationData` uses an augmented set of possible ratings, where the following three extra 'ratings' are allowed (see point 2.5.1 (h) of the ECB instructions):

1. `Default` - for names that defaulted during the observation period
2. `Migrated` - for non-defaulted names that have moved to a different model during the observation period
3. `Gone` - for non-defaulted names that have terminated their business relationship with the credit institution during the observation period

Note that the validation data containers should have data only for the names that have been pre-screened to be in scope for validation. Given operational data containers `operationalData1` and `operationalData2` for some reference dates, use an MRM helper function to construct the associated validation data:

```
validationData = mrm.data.validation.reporting.pd.validationDataFromOperationalData(operationalDa
```

This not only performs the appropriate screening but also takes care of the augmentation of the terminal rating categories as explained above.

Validation of External Models

This example shows how to use MATLAB® model validation tools on models existing outside of MATLAB using Modelscape™.

Such a model could be implemented in Python®, R, SAS®, or it could be a MATLAB model deployed on a web service.

This example covers the use of external models from the point of view of a Model Validator or other model end user. Although the Validator does not need to know this, this example assumes that a model has been deployed to a microservice such as Flask (Python) or Plumber (R). The example also explains how such microservices and alternative interfaces should be implemented.

This example calls an external model from MATLAB to evaluate it with different inputs. The example then shows you how to implement the API for any external model so that it can be called from MATLAB.

Call an External Model from MATLAB

This section shows you how to set up an interface and call an externally deployed model. This example uses a Python toy model although it could be implemented in any other programming language.

Setup the External Model

Use the Python code in the Appendix to set up mock data of a credit scoring model. This model adds noise to the input data, scales it, and returns the value as the output credit score of the applicant.

Run the script to make this "model" available in a test development server. The URL should be clearly visible in the output when the script is run. Copy it here if different from what is shown here:

```
ModelURL = "http://172.26.249.170:5000/";
```

In an actual model validation exercise, this information could be provided by the Model Developer as part of the validation request.

To set up a connection to this model, run the following:

```
extModel = externalModelClient("RootURL", ModelURL)
```

```
extModel = 
  ExternalModel with properties:

        InputNames: "income"
    ParameterNames: "weight"
       OutputNames: "score"
```

The model expects a single input called 'income' and a single parameter called 'weight', and returns a single output called 'score'.

The types and sizes of these inputs should be explained in model documentation, but you can also find this information in the `InputDefinition`, `ParameterDefinition` and `OutputDefinition` properties of `extModel.`

```
extModel.InputDefinition
```

```
ans = struct with fields:
       sizes: []
    dataType: [1×1 struct]
```

```
extModel.InputDefinition.dataType
```

```
ans = struct with fields:
    name: "double"
```

Empty `sizes` property indicates that a scalar is expected.

Evaluate the Model

Use the `evaluate` method of `ExternalModel` on your model. This method expects two inputs:

- The first input must be a table. Each row of the table must consist of the data for a single customer, or a single 'run'. The table is then a 'batch of runs'. The variable names of the table must match the `InputNames` shown by `ExternalModel`.
- The second input is a struct whose fields match the `ParameterNames` shown by `ExternalModel`. The values carried by this struct apply to all the runs in the batch. If the model has no parameters, omit this input.

The primary output is a table whose variable names match the `OutputNames` shown by `ExternalModel.` The rows correspond to the runs in the input batch. There may also be run-specific diagnostics consisting of one struct per run and a single batch diagnostic struct.

For your toy model, use random numbers in the range from 0 to 100,000 as customer incomes for the input data. For parameters, use a weight of 1.1.

```
N = 1000;
income = 1e5*rand(N,1);
inputData = table(income, 'VariableNames',"income");
parameters = struct('weight', 1.1);
```

Call your model.

```
[modelScores, diagnostics, batchDiagnostics] = evaluate(extModel, inputData, parameters);
head(modelScores)
```

```
             score 
             ______

    Row_1    110.66
    Row_2    137.69
    Row_3    52.155
    Row_4    87.029
    Row_5    73.207
    Row_6    26.722
    Row_7    36.671
    Row_8    24.878
```

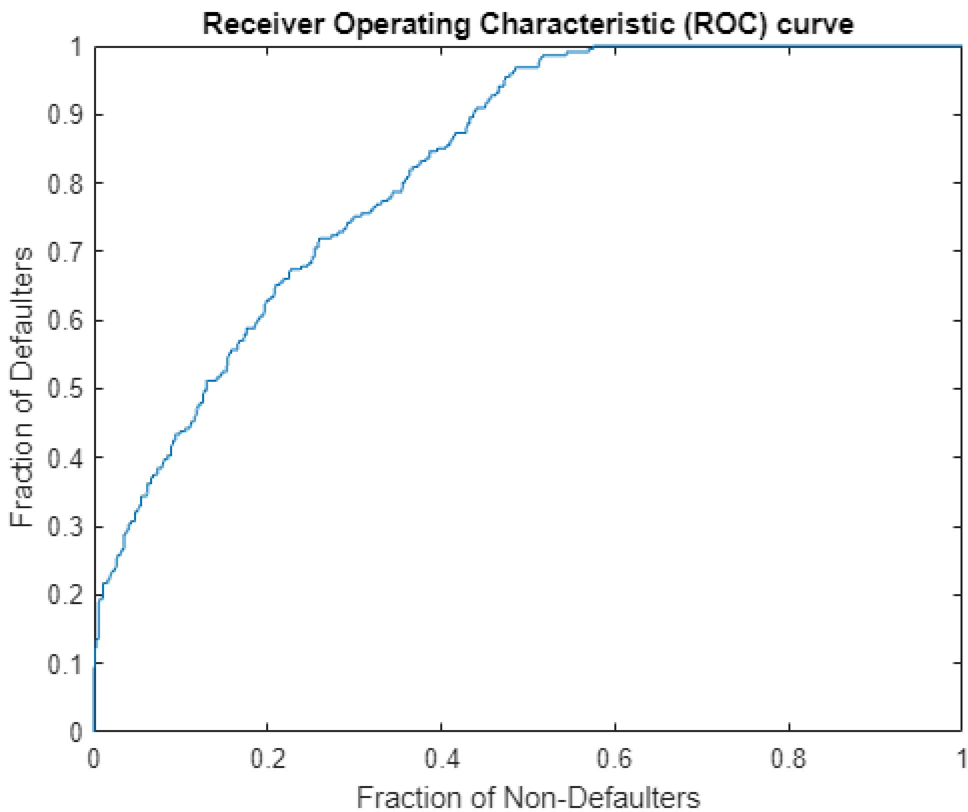
The output is a table of the same size as the inputs. Not specifying the row names in input data defaults them to Row_1, Row2, and so on.

For this example, create a mock response variable by thresholding the income. Validate the scores of the above model against this response variable.

```
defaultIndicators = income < 20000; % mocked-up response data
aurocMetric = mrm.data.validation.pd.AUROC(defaultIndicators, modelScores.score);
formatResult(aurocMetric)
```

```
ans = 
"Area under ROC curve is 0.8236"
```

```
visualize(aurocMetric);
```



The toy model returns some diagnostics to illustrate their size and shape. The run-specific `diagnostics` are a single struct with a field for every run.

```
diagnostics.Row_125
```

```
ans = struct with fields:
    noise: 1.7133e+04
```

In this case, each struct records some 'noise' term that was used in the calculation of the model prediction. Batch diagnostics consist of a single struct carrying information shared across all runs, in this case, the elapsed valuation time at the server side.

```
batchDiagnostics
```

```
batchDiagnostics = struct with fields:
    valuationTime: 0.0080
```

Extra Arguments

Under the hood, `ExternalModel` talks to the model through a REST API. If necessary, the headers and HTTP options used for the corresponding message exchanges can be modified by passing extra `Headers` and `Options` arguments to `externalModelClient. Headers` should be of type `matlab.net.http.HeaderField` objects and `Options` should be of type `matlab.net.http.HTTPOptions`.

For example, extend the connection timeout to 20 seconds.

```
options = matlab.net.http.HTTPOptions('ConnectTimeout',20);
extModel = externalModelClient("RootURL", ModelURL, "Options", options)
```

Implement an ExternalModel Interface

This part of the example explains how to implement an API for an external model to call it from MATLAB.

The `externalModelClient` function creates an object of type `mrm.validation.external.ExternalModel`. This object talks to the external model through a REST API, and it works with any model that implements the API below.

Endpoints

The API must implement two endpoints:

- `/signature` must accept a GET request and return a JSON string carrying the information about inputs, parameters and outputs.
- `/evaluate` must accept a POST request with inputs and parameters in a JSON format and must return a payload containing outputs, diagnostics, and batch diagnostics as a JSON string.

The status code for a successful response should be `200 OK`; note that this is the default in Flask, for example.

Evaluation Inputs

The `/evaluate` endpoint should accept a payload of the following format.

```
{
  "inputs": {
    "columns": [
      "customer_annual_income",
      "time_to_retirement"
    ],
    "index": [
      "run_1",
      "run_2",
      "run_3"
    ],
    "data": [
      [*, *],
      [*, *],
      [*, *]
    ]
  },
  "parameters": {
    "unemployment_rate": *,
    "model_calibration_factor": *
  }
}
```

The `columns` in `inputs` should list the input names, `index` should specify the row names, `data` should contain the actual input data one row at the time, and `parameters` should just record the parameters with their values. The asterisks indicate the values - for example doubles or strings.

Note that the `inputs` datum is compatible with the construction of Pandas DataFrames with `split` orientation; see the example implementation in the Appendix.

Response Formats

The `/signature` endpoint should return a payload of the following format:

```
{
  "inputs": [{
    "name": "customer_annual_income",
    "dataType": {
      "name": "double",
    },
    "sizes": []
  },
  {
    "name": "time_to_retirement",
    "dataType": {
      "name": "double",
    },
    "sizes": []
  }],
  "batchInputs": [{
    "name": "unemployment_rate",
    "dataType": {
      "name": "double",
    },
    "sizes": []
  },
  {
    "name": "model_calibration_factor",
    "dataType": {
      "name": "double",
    },
    "sizes": []
  }],
  "outputs": [{
    "name": "probability_of_default",
    "dataType": {
      "name": "double",
    },
    "sizes": []
  }]
}
```


The /`evaluate` endpoint should return a payload in the following format:

```
{
  "outputs": {
    "columns": [
      "probability_of_default",
      "confidence"
    ],
    "index": [
      "run_1",
      "run_2",
      "run_3"
    ],
    "data": [
      [*, *],
      [*, *],
      [*, *]
    ]
  },
  "diagnostics": {
    "run_1": {"field": "value"},
    "run_2": {"field": "value"},
    "run_3": {"field": "value"}
  },
  "batchDiagnostics": {"field": "value"}
}
```

Note again that the `outputs` data is compatible with the JSON output of Pandas dataframes with `split` orientation.

See the sample code in the Appendix for an interface implementation used in the first part of this example.

Work with Alternative APIs

This section explains how to make external models available to a Model Validator in MATLAB when the default API is either impossible or inconvenient to implement - for example when your organization already has a preferred REST API for evaluating models. For this, implement an API class that inherits from `mrm.validation.external.ExternalModelBase`. Package this implementation in a `+mrm/+validation/+external/` folder on the MATLAB path.

This custom API must populate the `InputNames`, `ParameterNames` and `OutputNames` properties shown to the Validator after an `externalModelClient` call. It must also implement the `evaluate` method which should take as inputs a table and a struct as in the default API `ExternalModel`. It is then the responsibility of the custom API to serialise the inputs, manage the REST API calls, and deserialise the outputs into tables and structs as shown above.

When a custom API has been implemented as, say, `mrm.validation.external.CustomAPI`, the Validator can initialize a connection to the model through this client by adding an `APIType` argument to the `externalModelClient` call.

```
extModelNew = externalModelClient("APIType", "CustomAPI", "RootURL", ModelURL)
```

Any further arguments will also be passed through to `CustomAPI`.

Appendix: Flask Interface

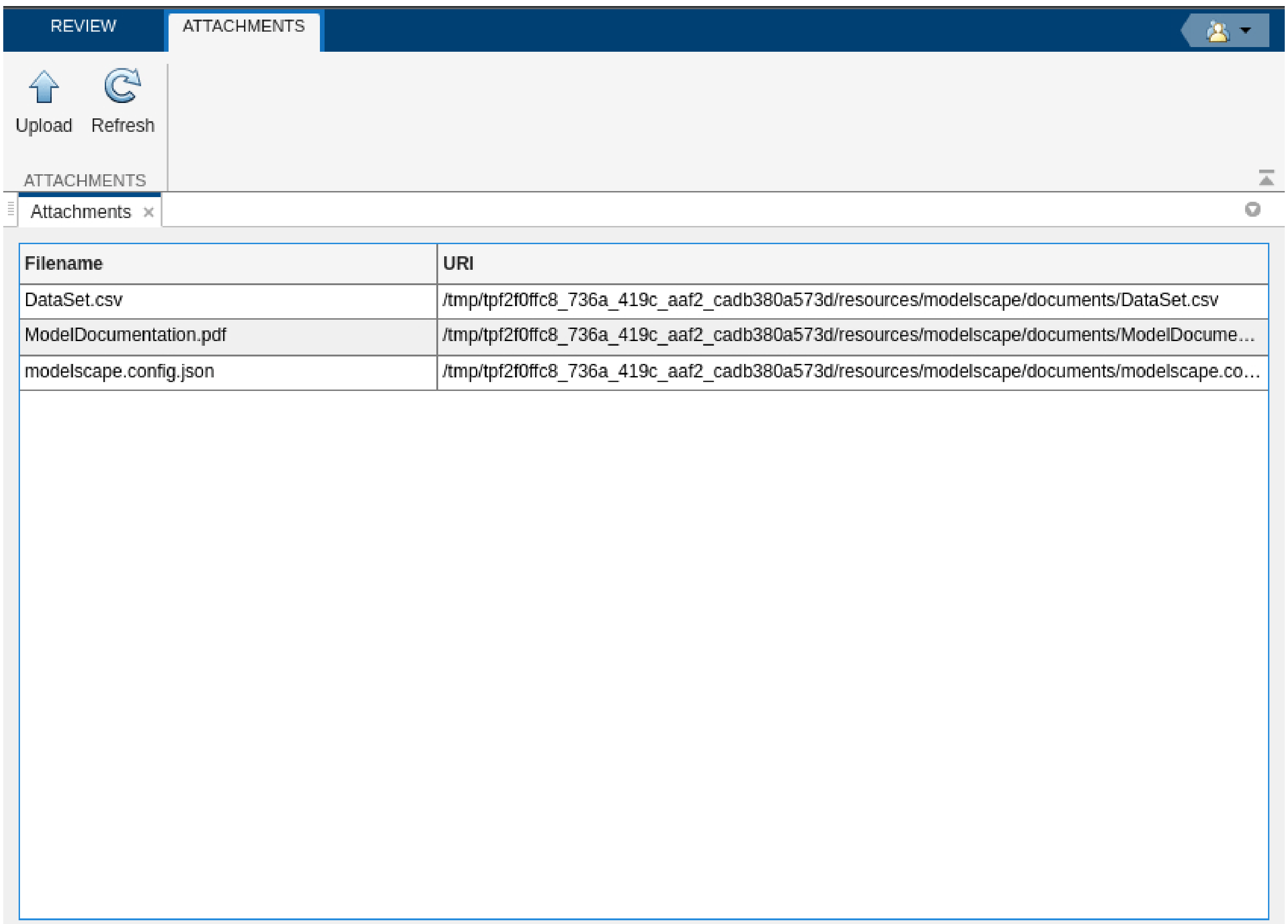
The following Python code was used for setting up the external model used in the first part of this example.

```
from flask import Flask, request, jsonify
import pandas as pd
import numpy as np
import time

toyModel = Flask(__name__)

@toyModel.route('/evaluate', methods=['POST'])
def calc():
    start = time.time()
    data = request.get_json()
    inputData = data['inputs']
    inputDF = pd.DataFrame(inputData['data'], columns=inputData['columns'], index=inputData['inde
    parameters = data['parameters']

    noise = np.random.uniform(low = -50000, high=50000, size=inputDF.shape)

    outDF = inputDF.rename(columns={'income':'score'})
    outDF = outDF.add(noise)
    outDF = outDF.mul(parameters['weight']/1000)

    diagnostics = pd.DataFrame(noise, columns=["noise"], index=inputDF.index)
    end = time.time()
    batchDiagnostics = {'valuationTime' : end - start}
    output = {'outputs': outDF.to_json(orient='split'),
              'diagnostics' : diagnostics.to_dict(orient='index'),
              'batchDiagnostics' : batchDiagnostics}
    return output

@toyModel.route('/signature', methods=['GET'])
def getInputs():
    outData = {
        'inputs': [{"name": "income", "dataType": {"name": "double"},"sizes": []}],
        'parameters': [{"name": "weight", "dataType": {"name": "double"}, "sizes": []}],
        'outputs': [{"name": "score", "dataType": {"name": "double"}, "sizes": []}]
        }
    return(jsonify(outData))


if __name__ == '__main__':
    toyModel.run(debug=True, host='0.0.0.0')
```

File Attachments in Modelscape Review Editor

This example shows how to attach files to reviews using Review Editor app in the Modescape™ Review Environment (MRE).

You can attach documents to the Review Editor app. These documents include recommendations for improvement or evidence of model performance reports outside the Review Editor app. Use the app to attach files such as detailed model validation documents and scripts supporting such documents. By default, the Review Editor attaches the files to the project or model repository You can also use the tools in the package to store the attachments in a network folder. This example explains how to use these tools, extend, and customize them.



Use Custom Repositories

Attachments repositories are subclasses of an (abstract) `modelscape.review.app.filerepository.FileRepository` base class. Construct such a subclass using a Review object, which defines the following functions:

- `upload`: how and where a given file should be stored.

- `list`: a list of the currently attached files.
- `URI`: an identifier for a given file.
- `openRepositoryLocation`: open the entire attachments repository for inspection.

The implementation must be in the MATLAB path.

For example, see the network folder repository:

```
edit modelscape.review.app.filerepository.NetworkFolderRepository;
```

Configure Attachment Repositories

Use MATLAB settings to point the MRE to the correct file repository class. To see the current active value of this setting, run:

```
s = settings;
s.mrmreview.FileRepository.Type.ActiveValue
```

```
ans =
"modelscape.review.app.filerepository.ProjectRepository"
```

The default value is `modelscape.review.app.filerepository.ProjectRepository`. If your custom repository file, `myRepository.m`, is in directory `/+mre/+custom/` on the MATLAB path, then point the MRE to it by running:

```
s.mrmreview.FileRepository.Type.TemporaryValue = "mre.custom.myRepository";
```

Querying the active value of the setting as above should now return `mre.custom.myRepository`. To reset the value of this setting, run

```
s.mrmreview.FileRepository.Type.TemporaryValue = s.mrmreview.FileRepository.Type.FactoryValue;
```

Use Network Folder Repository

You can also define other MATLAB settings and use them for configuring project repositories. Review Editor contains a repository definition for storing attachments to a network folder. To use this repository, run:

```
s = settings;
s.mrmreview.FileRepository.Type.TemporaryValue = "modelscape.review.app.filerepository.NetworkFo
```

Set the network folder to which Review Editor must copy the files to.

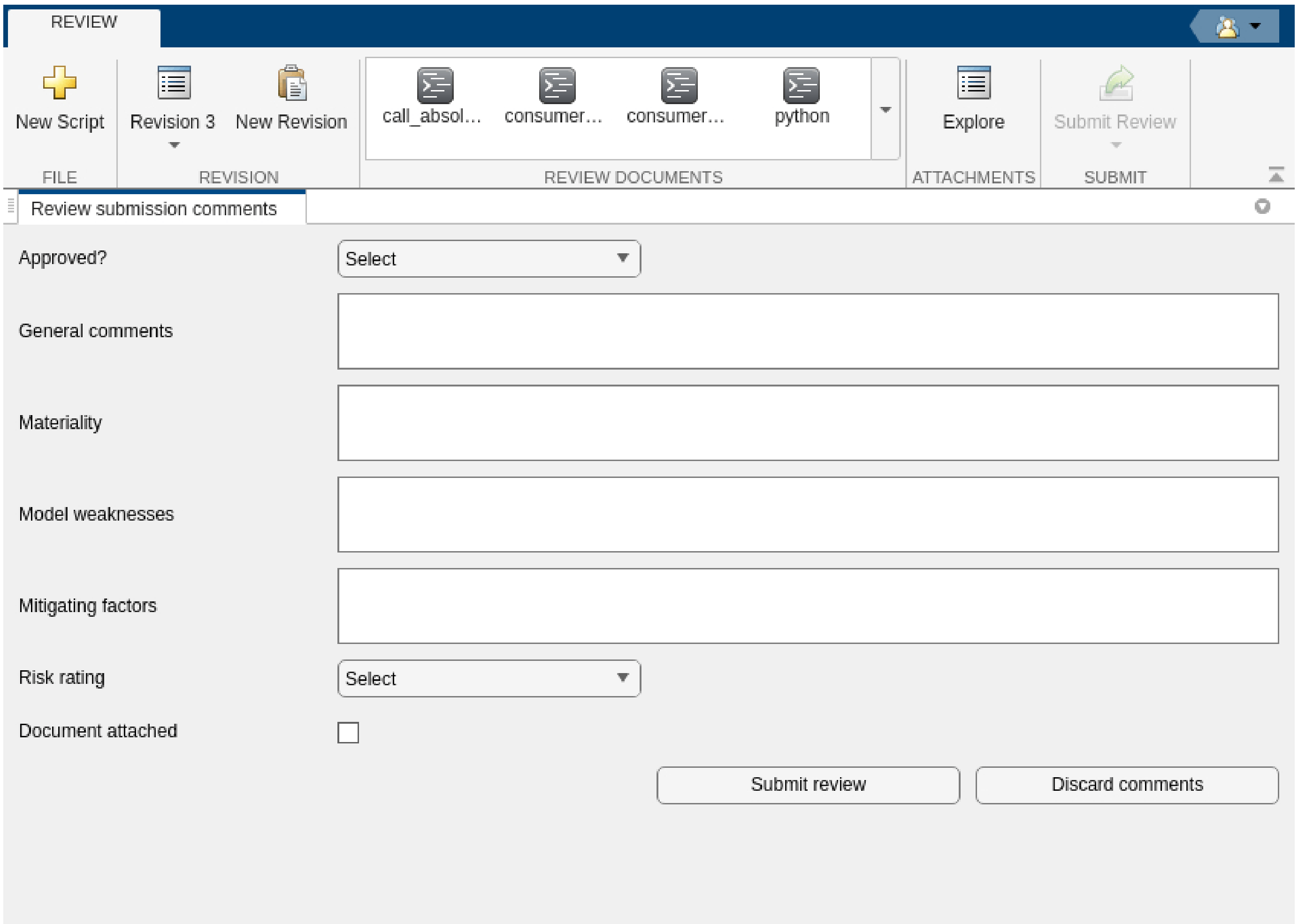
```
s.mrmreview.FileRepository.Root.TemporaryValue = "//some/network/folder/name";
```

To ensure each model gets a unique directory, this repository saves the files to a directory of the form `//some/network/folder/name`, where `name` is the string returned by the `ModelName` method of the Modelscape `Review` class.

Customization of Signoff Forms in Review Editor

This example shows how to customize review sign off forms using Review Editor app in the Modelscape™ Review Environment (MRE).

MRE comes equipped with a selection of forms that can be customized to sign off on a given review to match your validation processes. You can also create your own signoff forms using MRE.



The customization consists of two parts: formatting of the signoff forms and configuring the list of forms available to the validators.

Format Signoff Forms

Implement customized signoff forms as subclasses of an abstract interface class `modelscape.review.app.signoff.SignoffForm`. Begin by opening an example form in the Model Review Environment.

```
edit modelscape.review.app.signoff.forms.FullReview
```

In most cases, you can simply copy and rename this class, and edit it as required. The class definition must be on your MATLAB path.

You must give the validator a list of *labels* corresponding to various steps of the validation work, and a list of *controls* that allow users to input their comments. Labels are strings such as "Approved?", "Materiality" or "Known weaknesses", whereas controls include drop-down menus, checkboxes, and containers for free-form text; there should be one control for each label. The form also requires *submit* and *cancel* buttons. The layout of these labels, values, submit and cancel buttons is part of the configuration. To select the form from the submit review toolstrip dropdown, you also need a method called `fullName()`.

Configure Components of the Form

This section shows you how to configure the components of the form.

Layout

Control the layout of labels, controls and buttons by a `matlab.ui.container.GridLayout` object that uses the `Parent` object of the form as its parent container.

```
this.UIGrid = uigridlayout(this.Parent, [8 4],...
    'ColumnWidth', {'1x','1x','1x', '1x'}, ...
    'RowHeight', {25, 50, 50, 50, 50, 25, 25, 25});
```

In this case, the grid has one row per label-control pair and one row for the submit and cancel buttons. Row heights vary depending on the type of control to be used.

Labels and Controls

Insert the labels and controls into the layout grid using utility functions in the MRE package.

```
import modelscape.review.app.signoff.helpers.formatLabel;
import modelscape.review.app.signoff.helpers.formatDropDown;
import modelscape.review.app.signoff.helpers.formatTextArea;
import mmodelscape.review.app.signoff.helpers.formatCheckBox;
```

These helper functions create the labels and controls, and place them in the layout grid. Placed them in arrays called `this.Labels` and `this.Controls`. For example, place a label "Risk rating" into the first column of the 6th row and a drop-down menu with selection "High", "Medium", "Low" (and default value "Select") into the second column of that row.

```
this.Labels{6} = formatLabel(this.UIGrid, 'Risk rating', 6, 1);
this.Controls{6}= formatDropDown(this.UIGrid, 6, 2, ...
                {'Select', 'High', 'Medium', 'Low'}, 'Select');
```

Use the other two helper functions in a similar way.

Submit and Cancel Buttons

Define the remaining two components using the following helper functions of the `SignoffForm` base class.

```
createSubmitReviewButton(this, 8, 3);
createDiscardReviewButton(this, 8, 4);
```

These will place the required buttons in the 3rd and 4th column of the 8th row of the layout grid. The helper functions also configure the buttons to trigger the appropriate events in the Review app.

fullName()

For each signoff form class, define a static method `fullName(),` which returns the name, or a short description, of the implemented class. Use a concise string to identify this customized form in the signoff dropdown list of the MRE main toolstrip.

An example of this method definition is as follows:

```
methods (Static)
    function fn = fullName()
        fn = "Full review signoff";
    end
end
```

Configure the Signoff Form Selection Menu

The signoff forms for review submission are in the dropdown list under the Submit Review icon in the Review Editor app. By default, the list contains the examples of the MRE package. This section explains how to configure the list.

Configuration File

Implement each signoff form as a class definition. Encode the list of forms shown to the user into an XML file that lists these classes in the following syntax.

1. The XML root node is called `<formSelection>`.
2. Define each form in the list by a <formDefinition> node. These will contain a <className> node that sets the full name of the form class to be used

For example, the following XML defines the forms shipped with the MRE package:

```
<formSelection>
    <formDefinition>
        <className>modelscape.review.app.signoff.forms.FullReview</className>
    </formDefinition>
    <formDefinition>
        <className>modelscape.review.app.signoff.forms.ReducedReview</className>
    </formDefinition>
</formSelection>
```

This file need not be stored in the MATLAB path.

MRE SignoffFormsSelection Setting

To point the MRE to the correct XML configuration file, use MATLAB settings. To see the current value of this setting, run:

```
s = settings;
s.mrmreview.Signoff.ConfigFile.ActiveValue
```

The default value of this setting is the empty string.

If the configuration file is saved as `c:\MRE\resources\formSelection.xml`, set MRE to point to this file.

```
s.mrmreview.Signoff.ConfigFile.TemporaryValue = "c:\Modelscape\resources\formSelection.xml";
```

Querying the active value as shown above should then return the location of the custom XML.

To reset the default setting, run:

```
s.mrmreview.Signoff.ConfigFile.TemporaryValue = s.mrmreview.Signoff.ConfigFile.FactoryValue;
```

Model Implementation for Modelscape Deploy

This example shows you how to use and implement the Modelscape™ Deploy™.

Modelscape Deploy supports a generic interface for specifying model inputs, model outputs, and a single method to execute models. This example explains the interface and shows you how a model developer must implement it.

This example uses a toy model that takes inputs x and y and calculates the weighted sum z = A*x + B*y, where A and B are scalar weights. If x and y are vectors, the evaluation of the model is a **batch** evaluation for each pair of elements in x and y. The scalars A and B are configurable, but constant across a batch of evaluations – and are therefore regarded as **parameters**.

Realistic examples of inputs varying within a batch are the contract details for a book of derivative transactions, and the data corresponding to a group of credit card applicants. Examples of parameters corresponding to these inputs are the number of Monte Carlo paths used for pricing the derivatives, and certain macro-economic data used for credit-scoring the loan applicants. Note that the batch parameters are fixed for all inputs within a batch.

Work with Modelscape Deploy

Models to be executed in Modelscape Deploy must be implemented as subclasses of `mrm.execution.Model`.

```
classdef WeightedSum < mrm.execution.Model
```

Both the inputs and the outputs of the model execution must be tables. Each model class defines how to interroage the input table and populate the output table.

Methods for Inputs and Outputs

Each model must implement three methods for specifying the inputs and the outputs of the model.

- `getInputs`: returns the definition of the variables required for each evaluation in a batch - for example, the names and types of x and y above.
- `getParameters`: returns the definition of the variables that are fixed within a batch of evaluations - that is the names and types of A and B above.
- `getOutputs`: returns the definition of the output variables - variable z above.

The output, in each case, should be a struct with the following fields:

- `name`: a cell array of strings containing the names of the input/output variables.
- type: a cell array of structs defining the type of each input/output variable - each struct should have a field called `name` with any of the values listed in `mwtype`.
- `sizes`: a cell array of two-element arrays `[a b]` defining the size of each input/output variable - use NaN to indicate unrestricted size, e.g. `[1 NaN]` for a single column of an arbitrary height.

Define the toy model as follows.

```
function parameters = getInputs(~)
    doubleDatatype = struct( ...
        "name", "double");
    parameters = struct( ...
```

```
        "name", {"X", "Y"}, ...
        "dataType", {doubleDatatype, doubleDatatype});
end

function parameters = getParameters(~)
    doubleDatatype = struct( ...
        "name", "double");
    parameters = struct( ...
        "name", {"A", "B"}, ...
        "dataType", {doubleDatatype, doubleDatatype});
end

function parameters = getOutputs(~)
    doubleDatatype = struct( ...
        "name", "double");
    parameters = struct( ...
        "name", {"Z"}, ...
        "dataType", {doubleDatatype});
end
```

Evaluation Method

Define the evaluate method as follows.

```
[outputs, diagnostics, batchDiagnostics] = evaluate(this, inputs, parameters)
```

Here `inputs` should be a table, with a row for each evaluation within the batch, and `parameters` should be struct, that contains the variables that apply to all evaluations within the batch.

The `outputs` variable must be a table and contain a row for each row of the `inputs` table. The `diagnostics` output must be an array of structs, one for each row of the input table. The `batchDiagnostic` output is a single diagnostic for the whole batch and must be a scalar struct.

The toy model also has the following definition.

```
function [outputs, diagnostics, batchDiagnostics] = evaluate(~, inputs, parameters)
    outputs = table( ...
        parameters.A * inputs.X + parameters.B * inputs.Y, ...
        'VariableNames', {'Z'}, ...
        'RowNames', inputs.Properties.RowNames);
    rawDiagnostics = [inputs.Properties.RowNames, repmat({struct()}, numel(inputs.Properties.RowN
    diagnostics = struct(rawDiagnostics{:});
    batchDiagnostics = struct();
end
```

In this case the `diagnostics` structs are empty. In the more complicated examples listed above, they could carry, for instance, information about the Monte Carlo noise present in the valuation.

Create an image for Deployment

To deploy a model that implements the `mrm.execution.Model` interface, firstly, package the executable model code into a Docker® image suitable for deployment using MATLAB® Compiler SDK. Use the helper function `mrm.execution.packageModel` to do this.

```
modelInstance = WeightedSum();
outputFolder = tempname();
imageName = mrm.execution.compiler.packageModel(modelInstance, ...
    OutputFolder=outputFolder, ...
```

```
    Name="weighted-sum", ...
    Tag="v1")
```

```
Runtime Image Already Exists
Sending build context to Docker daemon    278kB

Step 1/6 : FROM matlabruntime/r2023a/prerelease/update0/308000000000000000
 ---> 9578d4e15248
Step 2/6 : COPY ./applicationFilesForMATLABCompiler /usr/bin/mlrtapp
 ---> ff0709fcec70
Step 3/6 : RUN chmod -R a+rX /usr/bin/mlrtapp/*
 ---> Running in c83375557f83
Removing intermediate container c83375557f83
 ---> da1d6f8978b3
Step 4/6 : RUN useradd -ms /bin/bash appuser
 ---> Running in aab934c26070
Removing intermediate container aab934c26070
 ---> 5e5490e6e58e
Step 5/6 : USER appuser
 ---> Running in fd6d76d6d108
Removing intermediate container fd6d76d6d108
 ---> 54efdb411a1b
Step 6/6 : ENTRYPOINT ["/opt/matlabruntime/R2023a/bin/glnxa64/muserve", "-a", "/usr/bin/mlrtapp/v
 ---> Running in 4c6771bc9751
Removing intermediate container 4c6771bc9751
 ---> 0da443f8e974
Successfully built 0da443f8e974
Successfully tagged weighted-sum:v1

DOCKER CONTEXT LOCATION:

/tmp/tp984f0556_95f6_46b1_a428_473bcc9e54dc/docker


FOR HELP GETTING STARTED WITH MICROSERVICE IMAGES, PLEASE READ:

/tmp/tp984f0556_95f6_46b1_a428_473bcc9e54dc/docker/GettingStarted.txt

Sending build context to Docker daemon  4.608kB

Step 1/7 : FROM weighted-sum:v1
 ---> 0da443f8e974
Step 2/7 : COPY ./routes.json /usr/bin/mlrtapp/routes.json
 ---> e97e7b05ec7e
Step 3/7 : USER root
 ---> Running in 01c65af653f1
Removing intermediate container 01c65af653f1
 ---> 276549f8c441
Step 4/7 : RUN useradd -u 2000 -ms /bin/bash modeluser
 ---> Running in e5d62f47d4fa
Removing intermediate container e5d62f47d4fa
 ---> 206c7ad88b83
Step 5/7 : USER 2000
 ---> Running in 98307802e3cb
Removing intermediate container 98307802e3cb
 ---> 0ac44031eb4a
Step 6/7 : EXPOSE 8080
 ---> Running in 75b018a3bd8d
```

```
Removing intermediate container 75b018a3bd8d
 ---> 314e6255d91e
Step 7/7 : CMD ["--http", "8080","--routes-file", "/usr/bin/mlrtapp/routes.json"]
 ---> Running in 1549e1f467cc
Removing intermediate container 1549e1f467cc
 ---> f703f5e9bd1b
Successfully built f703f5e9bd1b
Successfully tagged weighted-sum:v1

imageName = 
"weighted-sum:v1"
```

Deploy to Modelscape Deploy

The image must be pushed to a Docker registry visible to the Modelscape API. For the next steps, choose the model version for which you want to create a model version build. Create the build, a deployment environment, and deploy your build.

Customizing Model Inventory: Risk Tiering

This example shows how to customize the Model Inventory to hold information specific to your organization.

You can customize the model data entry and the model summary table. You can also add new filters to the Inventory Browser app to make it easy to find models with a particular value of a custom attribute.

This example uses a simple tree-based approach to Risk Tiering as an example. The model is from a paper by Mankatonia and Joshi (*Measuring model risk: a practitioner's approach,* RMA Journal, 2013). This example along with other examples are also discussed in a paper by Kiritz, Ravitz and Levonian (*Model risk tiering: an exploration of industry practices and principles*, Journal of Risk Model Validation, 2019).

Add Custom Data to Inventory Browser

The inventory data is most likely stored outside of MATLAB®, for example, in a database. Various features of the Inventory Browser such as the model entry form do not access this external resource directly - rather, they interact with it through a **client**. Modelscape™ supports both database-backed and (for test use) in-memory clients.

Add new model-specific data to the Inventory Browser as **references**. To do this:

- If necessary, create a new **type** for the reference.
- Create the reference itself.
- Associate the reference to a given model.

Create a new reference type called `RiskTieringData` with the following attributes:

- `RiskPriceValueUse`: a string Unset, True or False denoting whether the model is used to measure risk, price, or value.
- `CriticalUse`: a string Unset, True or False denoting whether the model is used for critical business decisions, regulatory reporting, or similar.
- `Exposure`: a string Unset, High, Medium, or Low denoting the exposure level of the model.
- `Override`: a string Unset, High, Medium, or Low denoting a risk tier level override.
- `RiskTier`: a string Unset, High, Medium, or Low denoting the final risk tier which is worked out from the information above.

Construct a client with a model and this reference type, and attach a reference to this model. To learn more about how to do this, contact MathWorks Consulting Services.

Open an Inventory session with this client.

```
app = mrm.inventory.InventoryApp(client);
app.open
```

Customize Model Data Entry

Customize the Inventory Browser model entry by adding new tabs next to the 'Details' included in the default view, and by changing the layout and contents of the 'Details' tab. Implement these customizations as subclasses of `mrm.inventory.model.FormCustomization`. Include the subclasses in +mrm/+inventory/+custom/+model/ folder on the MATLAB path. Implement the following methods for each subclass.

- The constructor must take two inputs: the parent mrm.inventory.model.Form object and the client carried by the Form. If you want to assign these to the Form and Client properties, leave this to the base class constructor and not implement this at all.
- Use populateCustomContents(this) to set up the additional tab and any controls such as drop-downs.
- onModelSet(this) must obtain the required references through the client and set these values to the controls. Note that the identifier of the model being displayed on the forms can be obtained from the GUIDEdit property of the parent form.
- onSubmit(this) must read the values carried by the dropdowns and other controls and use the client to update the relevant references associated to the model.

Note that you can customize the 'Details' tab by modifying the layout grid, stored as the DetailsLayout property, of the parent mrm.inventory.model.Form object. Here are some examples of possible customizations.

1. Replace controls with new custom controls by hiding the existing control. To do this, use the property Visible and create a new control in the same location in the grid.
2. Add new controls to the form by resizing the DetailsLayout grid.
3. Reorganize controls by using Layout.Row properties.

Finally, you can overwrite the labels of the 'Details' tab controls in a customization class. See mrm.inventory.model.Form for the names of properties defining the labels.

The following code snippet illustrates how you can apply these customizations to implement a risk tiering form. For the form layout, populateCustomContents creates dropdowns for RiskPriceValueUse, CriticalUse, Exposure and Override, and a non-editable label to display the resulting RiskTier. There is a button to recalculate the risk tier, as you want this to happen only once all the required inputs have been considered and set. Finally, the example shows a mechanism for displaying whether the risk tier stored in the inventory is in sync with the chosen inputs - if not, the comment '(Stale)' is added to the risk tier. The new tiering data is stored in the Inventory only when the 'Update' button is pressed.

```
classdef RiskTieringCustomTab < mrm.inventory.model.FormCustomization
%Implements a custom tab for calculating the risk tier

%   Copyright 2021-2023 The MathWorks, Inc.


    properties (Access = private)
        % Base grid
        Grid(1,1) matlab.ui.container.GridLayout

        % Tree model controls
        UsageDD(1,1) matlab.ui.control.DropDown
        CriticalUseDD(1,1) matlab.ui.control.DropDown
        ExposureLevelDD(1,1) matlab.ui.control.DropDown
        OverrideDD(1,1) matlab.ui.control.DropDown

        CalculateButton(1,1) matlab.ui.control.Button
        RiskTierLabel(1,1) matlab.ui.control.Label
```

```
        % Convenience variable for setting the stale status of the tiering
        SavedTieringData struct

        % Cache the reference type for risk tiering data
        RiskTieringReferenceType
    end

    methods
        function this = RiskTieringCustomTab(form, client)
            this@mrm.inventory.model.FormCustomization(form, client);
            this.RiskTieringReferenceType = this.Client.getReferenceTypeByName("RiskTieringData"
        end

        function populateCustomContent(this)
            parentTab = uitab(this.Form.TabGroup, ...
                              'Title', 'Risk tiering', ...
                              'Tag', 'risktiering_tab');

            % Set up grid
            this.Grid = uigridlayout(parentTab, [6 2]);
            this.Grid.RowHeight = repmat(30, 1, 6);
            this.Grid.ColumnWidth = {'1x', '1x'};

            % Row 1
            uilabel(this.Grid, 'Text', 'Does the model measure risk, price or value?');
            this.UsageDD = uidropdown(this.Grid, ...
                                      'Items', ["Unset"; "True"; "False"], ...
                                      "ItemsData", ["Unset"; "True"; "False"], ...
                                      "ValueChangedFcn", @(~,~)this.setStaleStatus);

            % Row 2
            uilabel(this.Grid, 'Text', 'Is the model usage critical?', ...
                    'Tooltip', 'Includes use for critical business decisions, regulatory purpose
            this.CriticalUseDD = uidropdown(this.Grid, ...
                                            'Items', ["Unset"; "True"; "False"], ...
                                            "ItemsData", ["Unset"; "True"; "False"], ...
                                            "ValueChangedFcn", @(~,~)this.setStaleStatus);

            % Row 3
            uilabel(this.Grid, 'Text', 'Exposure');
            this.ExposureLevelDD = uidropdown(this.Grid, ...
                                              'Items', ["Unset"; "High"; "Medium"; "Low"], ...
                                              'ItemsData', ["Unset"; "High"; "Medium"; "Low"], .
                                              'ValueChangedFcn', @(~,~)this.setStaleStatus);

            % Row 4
            % Tier names 5-7 are 'Low (Stale)' etc, so don't include them in the drop-down.
            uilabel(this.Grid, 'Text', 'Override');
            this.OverrideDD = uidropdown(this.Grid, ...
                                         'Items', ["Unset"; "Low"; "Medium"; "High"], ...
                                         "ItemsData", ["Unset"; "Low"; "Medium"; "High"], ...
                                         "ValueChangedFcn", @(~,~)this.setStaleStatus);

            % Row 5
            this.CalculateButton = uibutton(this.Grid, 'Text', 'Calculate');
            this.CalculateButton.ButtonPushedFcn = @this.onCalculateRiskTier;
            this.CalculateButton.Layout.Row = 5;
            this.CalculateButton.Layout.Column = 2;
```

```
            % Row 6
            uilabel(this.Grid, 'Text', 'Risk tier');
            this.RiskTierLabel = uilabel(this.Grid, 'Text', '');
        end

        function onModelSet(this)
            guid = this.Form.GUIDEdit.Value;
            tieringDataForThisModel = this.Client.getReferenceByModelAndType( ...
                guid, this.RiskTieringReferenceType.GUID);
            this.SavedTieringData = tieringDataForThisModel.Attributes;

            this.UsageDD.Value = this.SavedTieringData.RiskPriceValueUse;
            this.CriticalUseDD.Value = this.SavedTieringData.CriticalUse;
            this.ExposureLevelDD.Value = this.SavedTieringData.Exposure;
            this.OverrideDD.Value = this.SavedTieringData.Override;
            this.RiskTierLabel.Text = this.SavedTieringData.RiskTier;
        end

        function onSubmit(this)
            guid = this.Form.GUIDEdit.Value;
            data = containers.Map;
            data("RiskPriceValueUse") = this.UsageDD.Value;
            data("CriticalUse") = this.CriticalUseDD.Value;
            data("Exposure") = this.ExposureLevelDD.Value;
            data("Override") = this.OverrideDD.Value;
            data("RiskTier") = this.RiskTierLabel.Text;

            tieringReference = this.Client.getReferenceByModelAndType( ...
                guid, this.RiskTieringReferenceType.GUID);
            this.Client.updateReference(tieringReference.GUID, "Attributes", ...
                data);
        end
    end

    methods (Access = protected)

        function setStaleStatus(this)
            isStale = this.SavedTieringData.RiskPriceValueUse ~= this.UsageDD.Value || ...
                      this.SavedTieringData.CriticalUse ~= this.CriticalUseDD.Value || ...
                      this.SavedTieringData.Exposure ~= this.ExposureLevelDD.Value || ...
                      this.SavedTieringData.Override ~= this.OverrideDD.Value;

            if isStale && ~contains(this.RiskTierLabel.Text, "Stale") && ...
                    this.RiskTierLabel ~= "Unset"
                this.RiskTierLabel.Text = string(this.RiskTierLabel.Text) + " (Stale)";
            elseif ~isStale && contains(this.RiskTierLabel.Text, "Stale")
                this.RiskTierLabel.Text = extractBefore(this.RiskTierLabel.Text, ...
                                                        " (Stale)");
            end
        end

        function onCalculateRiskTier(this, ~, ~)
            tieringInputs.riskpricevalueflag = this.UsageDD.Value;
            tieringInputs.criticalflag = this.CriticalUseDD.Value;
            tieringInputs.exposurelevel = this.ExposureLevelDD.Value;
            tieringInputs.override =  this.OverrideDD.Value;
```

```
            riskTier = mrm.inventory.custom.riskTierTreeSimple(tieringInputs);
            this.RiskTierLabel.Text = riskTier;
        end
    end
end
```

Customize Model Summary Table

Customize the summary table of model data shown in the Inventory Browser by omitting columns, including columns corresponding to the custom data set up in the previous two sections, and reordering any of the columns shown. Implement these customizations as a single subclass of `mrm.inventory.model.TableCustomization` that must be in a `+mrm/+inventory/+custom/+model` folder on the MATLAB path. The class must implement the following methods:

- The constructor must accept a single input consisting of the user-visible headers for the default view of the model table. It must also set properties `ColumnVisible`, `ColumnOrdering` and `AllHeaders`.
- `process(this, uit, modelIds, client)` takes as its inputs the `uitable` being customized and the ids of the model to be displayed, and performs the required customizations. Client is also supplied for looking up custom data.

The following example code illustrates how this can be done. The resulting table shows only the name and the id of each model from the base product model data, and the exposure level, risk tier and any possible override from the tiering data itself. These columns are also reordered to demonstrate this capability.

```
classdef TableCustomizationExample < mrm.inventory.model.TableCustomization
%Example to illustrate addition, removal and reordering of summary table
%column.

%   Copyright 2021-2023 The MathWorks, Inc.


    methods
        function this = TableCustomizationExample(parentHeaders)
            this.ExtraHeaders = ["Risk Tier", "RiskPrice", "Exposure", "Critical", "Tier overrid
            this.AllHeaders = [parentHeaders, this.ExtraHeaders];

            baseVisible = [true, true, false]; % 1-2 of visible columns
            riskTierVisible = [true, false, true, false, true]; % 3-5 of visible columns

            this.ColumnVisible = [baseVisible, riskTierVisible];
            this.ColumnOrdering = [2 1 3 5 4]; % for visible columns only
        end

        function uit = process(this, uit, modelIds, client)
            arguments
                this
                uit matlab.ui.control.Table
                modelIds(1,:) string
                client
            end


            % Step 1: Read the risk tiering data for all the modelIds from
            % the client.
            tieringDataType = client.getReferenceTypeByName("RiskTieringData");
```

```
            tieringData = arrayfun(@(id)client.getReferenceByModelAndType(id, ...
                tieringDataType.GUID), modelIds);

            % Step 2: Arrange this to extraModelTable table with columns
            % corresponding to this.ExtraHeaders
            [tiers, materialUseFlags, exposures, criticalUseFlags, overrides] =  ...
                arrayfun(@(ref)readRiskTierData(ref), tieringData);
            extraModelData = table(tiers', materialUseFlags', exposures', ...
                criticalUseFlags', overrides', 'VariableNames', ...
                ["riskTier", "riskPriceValue", "exposure", "critical", "override"]);

            % Step 3: Concatenate this with uit.Data:
            uit.Data = [uit.Data, extraModelData];
            uit.ColumnName = this.AllHeaders;

            % Step 4: Use the helper methods from TableCustomization base
            % to reset the visibility and the ordering of the columns.
            this.setVisibility(uit);
            this.setOrdering(uit);
        end
    end
end

function [tier, materialuseflag, exposure, criticaluseflag, override] = readRiskTierData(ref)
    tier = string(ref.Attributes.RiskTier);
    materialuseflag = string(ref.Attributes.RiskPriceValueUse);
    exposure = string(ref.Attributes.Exposure);
    criticaluseflag = string(ref.Attributes.CriticalUse);
    override = string(ref.Attributes.Override);
end
```

Create Custom Filters

Inventory Browser is equipped with an interactive UI for creating filters to limit the list of models shown in the Models table. These filters make no distinction between data included in the default view and columns added as part of customization process. You can, for example, construct a filter to show only models with a ‘High’ risk tier.

Inventory Browser has two filters in the “Saved Filters” list of the filter editor: “Search by Name”, which allows you to filter by the model name, and “Create Custom Filter”, which shows a more complex filter, intended as a template for creating more complicated queries.

This section shows you how to create new filters and how to add them to the “Saved Filters” list.

To customize your own filters, implement new filters as subclasses of `mrm.inventory.model.filter.FilterDefinition` with the following properties.

- `Name()` must carry a string that is to be displayed in the Filter dropdown - for example "Filter by Risk Tier"
- `Serialization` must carry the default initialization of the filters as a JSON string

To understand the format of the serialization JSON, see `mrm.inventory.model.filter.FilterByName` for simple (“Primitive”) filters that reference just a single column and see the default serialization in mrm.inventory.model.filter.CustomFilterTemplate for more complex (“Composite”) filters.

To make the filters visible, implement a function called `modelFilters` in a +mrm/+inventory/+custom/+model/ folder on the MATLAB path. This function must take no inputs, and it must return a row array of all the filters you want to include in the Saved Filters list.

The code below illustrates this. Note that the `modelFilters` output can include a mixture of custom filters and filters shipped with Modelscape itself.

```
function filters = modelFilters()
%Example filter selection customization

%   Copyright 2022-2023 The MathWorks, Inc.

    filters = [ mrm.inventory.model.filter.FilterByName, ...
        mrm.inventory.custom.model.filter.FilterByRiskTier, ...
        mrm.inventory.model.filter.CustomFilterTemplate];
end
```

In the filter implementation, the variable name in `uit.Data` may be different from the user-visible header shows in the Inventory - here `riskTier` vs user-visible "Risk Tier". The implementation does not need to reside in any package folder, but it makes sense to have all customization code in a single location, so use +mrm/+inventory/+custom/+model/+filter here.

```
classdef FilterByRiskTier < mrm.inventory.model.filterDefinition
% Example defintion for filtering by risk tier in Modelscape Inventory

%   Copyright 2022-2023 The MathWorks, Inc.

    methods
        function this = FilterByRiskTier()
            this.name = "Filter by Risk Tier";
            this.Serialization = ['{"type":"Primitive","header":"riskTier",', ...
            '"operation":"CONTAINS","value":"","parent":[],"id":"1"}'];
        end
    end
end
```

Test Metrics in Modelscape

This example shows how to implement various test metrics in MATLAB® using Modelscape™.

For information about test metrics from the model developer's or validator's point of view, see "Credit Scorecard Validation Metrics" on page 5-48 or "Fairness Metrics in Modelscape" on page 5-26.

Write Test Metrics

The basic building block of Modelscape metrics framework is the `mrm.data.validation.TestMetric` class. This class defines the following properties:

- `Name`: a human-readable name for the test metric.
- `ShortName`: a concise name for accessing metrics in `MetricsHandler` objects. This name must be a valid MATLAB property name.
- `Value`: the value(s) carried by the metric. The values can be a scalar or a row vector of doubles.
- `Keys`: an n-by-m array of strings that parametrize the values of the metric. m is the length of `Value`. The keys default to an empty string.
- `KeyNames`: a vector of strings of size of the height of `Keys`. It defaults to "Key".
- `Diagnostics`: a free form struct carrying any diagnostics related to the calculation of the metric.

Any subclass of `TestMetric` must implement a constructor and a `compute` method to fill in these values.

For example, the Modelscape statistical parity difference (SPD) metric for bias detection has `Name` "Statistical Parity Difference" and `ShortName` "StatisticalParityDifference". The following table shows how the `Keys` and `KeyNames` are arranged.

SensitiveAttribute	Group	StatisticalParityDifference
"AgeGroup"	"45 < Age <= 60"	0
"AgeGroup"	"Age > 60"	-0.21242
"AgeGroup"	"30 < Age <= 45"	0.42456
"AgeGroup"	"Age <= 30"	0.54197
"ResStatus"	"Tenant"	0.080908
"ResStatus"	"Home Owner"	0
"ResStatus"	"Other"	-0.11961
"OtherCC"	"Yes"	0
"OtherCC"	"No"	0.19661

Here "SensitiveAttribute" and "Group" are the `KeyNames`, and the two columns with certain attribute-group combinations are the `Keys.` The `ShortName` appears as the third header, and the third column of the table carries the `Value` of the metric.

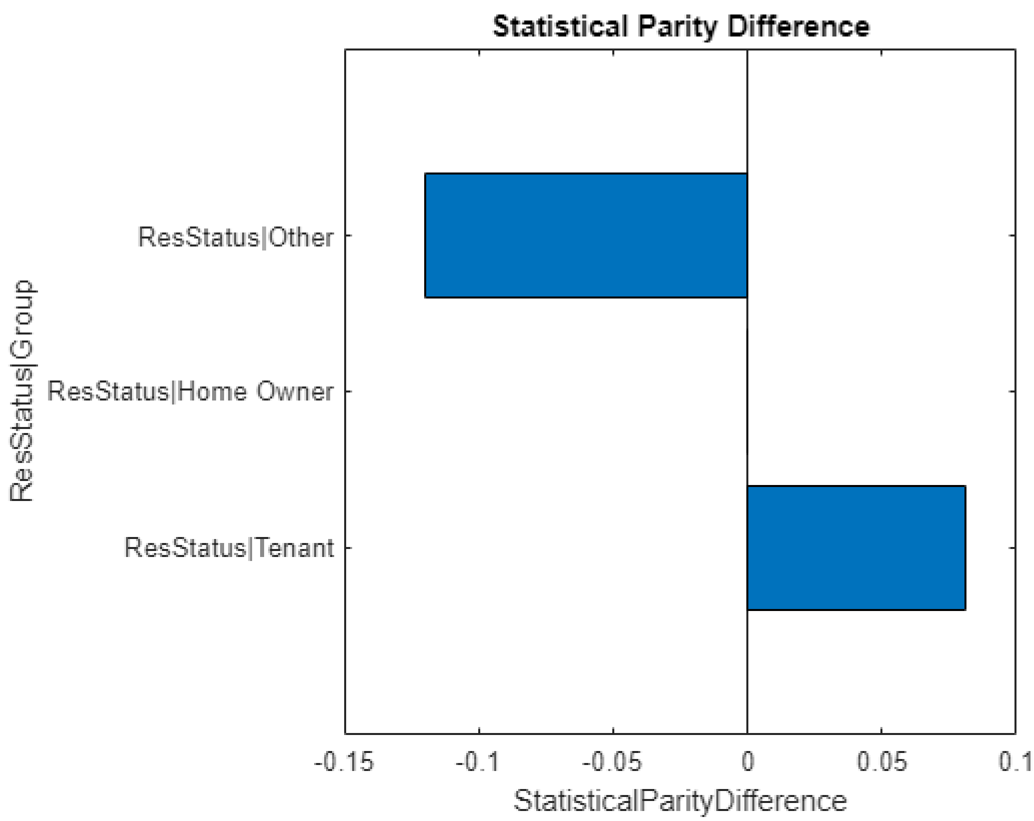
The base class has the following overridable methods:

- `ComparisonValue(this)`: use this method to change the value against which thresholds are compared - for example, in statistical hypothesis testing, this should return the p-value associated to the computed statistic.
- `formatResult(this):` returns by default a table as shown above for the SPD metric.
- `project(this):` returns a restriction of a (non-scalar) metric to a subset of keys. Extend the default implementation in a subclass to cover any diagnostic or auxiliary data carried by the subclass objects.

Write Metrics With Visualizations

To write test metrics equipped with visualizations, the metrics should inherit from `mrm.data.validation.TestMetricWithVisualization`. This class adds an additional requirement to the `TestMetric` base class to implement a visualization method with the signature `fig = visualize(this, options).` `options` allows for any name value arguments that may be useful for the given metric. For example, use a particular sensitive attribute with the `StatisticalParityDifference` metric for visualization.

```
spdFig = visualize(spdMetric, "SensitiveAttribute","ResStatus");
```



Write Metrics Projecting onto Selected Keys

The visualization above shows the SPD metrics for the `ResStatus` attribute only. This plot uses the `project` method of the `TestMetric` class that uses selected keys of a metric. For a metric with N key names, `project` accepts an array of up to N strings as the `Keys` argument. The output restricts the metric to those keys where the first key matches the first element of the array, the second key matches the second element of the array, and so on.

```
spdResStatus = project(spdMetric, "Keys", "ResStatus")
```

returns:

```
StatisticalParityDifference with properties:

     ShortName: "StatisticalParityDifference"
          Name: "Statistical Parity Difference"
         Value: [0.0809 0 -0.1196]
   Diagnostics: [1×1 struct]
          Keys: ["ResStatus|Tenant"    "ResStatus|Home Owner"    "ResStatus|Other"]
      KeyNames: "ResStatus|Group"
```

On specifying both keys, the results is a scalar metric:

```
spdTenant = project(spdMetric, "Keys", ["ResStatus", "Tenant"])
```

```
StatisticalParityDifference with properties:

     ShortName: "StatisticalParityDifference"
          Name: "Statistical Parity Difference"
         Value: 0.0809
   Diagnostics: [1×1 struct]
          Keys: "ResStatus|Tenant"
      KeyNames: "ResStatus|Tenant"
```

The base class implementation of `project` does not handle diagnostics or other auxiliary data carried by the subclass. If necessary, implement this in the subclass using the secondary `keySelection` output in `project`.

Write Summarizable Metrics

Summary metrics reveal a different aspect of non-scalar metrics. In the case of the SPD metric, across all the attribute-group pairs, the "summary" SPD value is the value with the largest deviation from the completely non-biased value of zero.

```
spdSummary = summary(spdMetric)
```

returns:

```
  StatisticalParityDifference with properties:

      ShortName: "StatisticalParityDifference"
           Name: "Statistical Parity Difference"
          Value: 0.5420
    Diagnostics: [1×1 struct]
           Keys: "Summary"
       KeyNames: ""
```

Summarize a given TestMetric class by inheriting from `mrm.data.validation.TestMetricWithSummaryValue` class and implementing the abstract `summary` method. This returns a metric of the same type with a singleton `Value`. The meaning of the summary value - if it exists- depends on the metric, so there is no default implementation for this method. However, the protected `summaryCore` method in `TestMetricWithSummaryValue` may be helpful.

Write Test Thresholds

Test metrics are often compared against thresholds to qualitatively assess of the inputs. For example, a model validator might require that the area under ROC curve should be at least 0.8 for the model to be deemed acceptable, values under 0.7 are red flags, and values between 0.7 and 0.8 require a closer look.

Use Modelscape class `mrm.data.validation.TestThresholds` to implement these thresholds. Encode the thresholds and classifications into a `TestThresholds` object.

```
aurocThresholds = mrm.data.validation.TestThresholds([0.7, 0.8], ["Fail", "Undecided", "Pass"]);
```

```
  TestThresholds with properties:

    Thresholds: [0.7000 0.8000]
        Labels: ["Fail"    "Undecided"    "Pass"]
```

These thresholds and labels govern the output of the `status` method of `TestThresholds.` For example, `status(aurocThresholds, 0.72)` returns the following.

```
  TestStatus with properties:

         Status: "Undecided"
        Comment: "(0.7, 0.8]"
    Diagnostics: [1×1 struct]
```

`Comment` indicates the interval to which the given input belongs.

Customize Thresholds

Implement thresholding regimes, with different narrative strings as Comments, or different diagnostics, as subclasses of `mrm.data.validation.TestThresholdsBase`. Implement the `status` method of the class to populate the `Comment` and `Diagnostics` properties as required.

Write Statistical Hypothesis Tests

In some cases, notably in statistical hypothesis testing, the relevant quantity to compare against test thresholds is the associated p-value (under some relevant null hypothesis). In these cases, use the test *metric* class to override the `ComparisonValue` method and return the p-value instead of the `Value` of the metric. For an example, see the Modelscape implementation of the Augmented Dickey-Fuller test.

```
edit mrm.data.validation.timeseries.ADFMetric
```

Set the thresholds against which to compare the p-values.

```
adfThreshold = mrm.data.validation.PValueThreshold(0.05)
```

```
  PValueThreshold with properties:

    Threshold: 0.0500
       Labels: ["Reject"    "Accept"]
```

This `TestThresholds` object returns `status` as "Reject" for p-values less than 0.05 and "Accept" otherwise.

Inventory Browser

Create Modelscape models and add dependencies

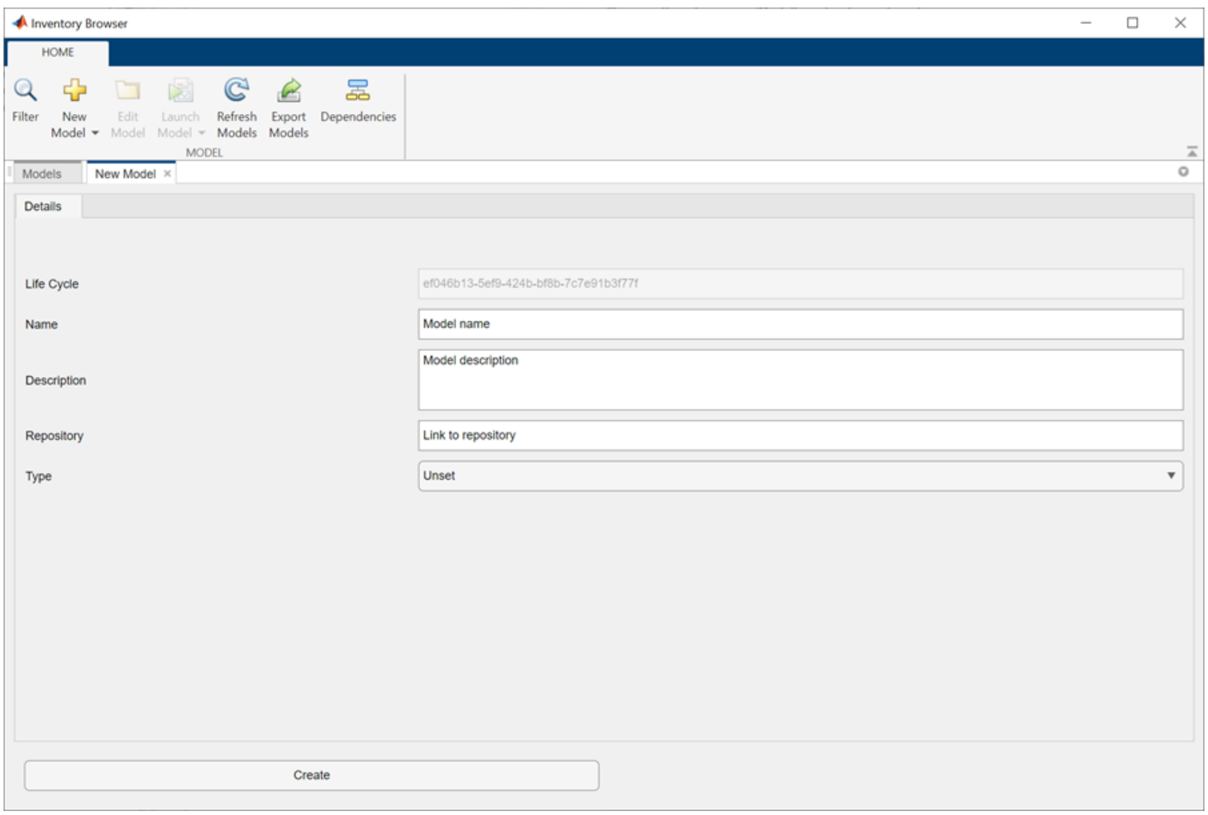
Description

Use the **Inventory Browser** app to create Modelscape models and add dependencies between models.

Modelscape models are quantitative solutions that apply statistical, economic, or other techniques to given inputs to produce an output. You can use these outputs to guide pricing or other business decisions. Modelscape models must be backed by a Git repository and associated with a lifecycle.

Each Modelscape model has multiple model versions. Model versions are committed updates of a Modelscape model. For example, Probability of Default for Retail Credit in Europe is a Modelscape model, and its 2015 and 2020 versions are the model versions. While a Modelscape model corresponds to a Git repository, a model version refers to a Git commit.

Use the model inventory browser app to create new Modelscape models and add dependencies between models.



Open the Inventory Browser App

- Launch the **Inventory Browser** app from the Modelscape Home page. To host Modelscape for your organization, contact MathWorks Consulting Services.

Examples

Create New Model

Create a new model by clicking **New Model** on the menu. Associate the model with a lifecycle. To create a lifecycle for your model, see the Lifecycle Designer app.

Add the model name and other relevant fields.

Click **Create** to create the model. The table of models on the left shows the new model.

Launch Model in Browser

Select a row in the **Models** table and click `Browser` under the **Launch Model** dropdown.

If the model has a valid repository URL, the app launches a browser that displays the repository.

Update Model Details

On the left of the app, in the table of models, double-click the row containing the model you want to update. The app opens the details windows associated with the model.

Update the fields you want to modify.

Click **Update** to save your changes.

Add Dependencies Between Models

Add dependencies between two or more Modelscape models in Inventory Browser.

Dependencies show the links between Modelscape models. When two Modelscape models are linked together by a dependency arrow, the model near the upstream link (at the head of the arrow) takes inputs from the outputs of the model near the downstream link (at the tail of the arrow).

You can also create dependencies between Modelscape models and data, or between models and Modelscape references. Modelscape references are the files or information that you can associate with a Modelscape model, model version, or Review document.

Click the **Model Dependencies** tab to show the existing dependencies between models.

Select upstream and downstream dependencies for the model and click **Add**. Alternatively, click the upstream model and hold **Ctrl** while you click the downstream model.

Click **Save** to save the dependency diagram.

Version History

Introduced in R2023a

See Also

Functions

Topics

"Modelscape Governance" on page 5-4

Lifecycle Designer

Create and edit lifecycles in Modelscape

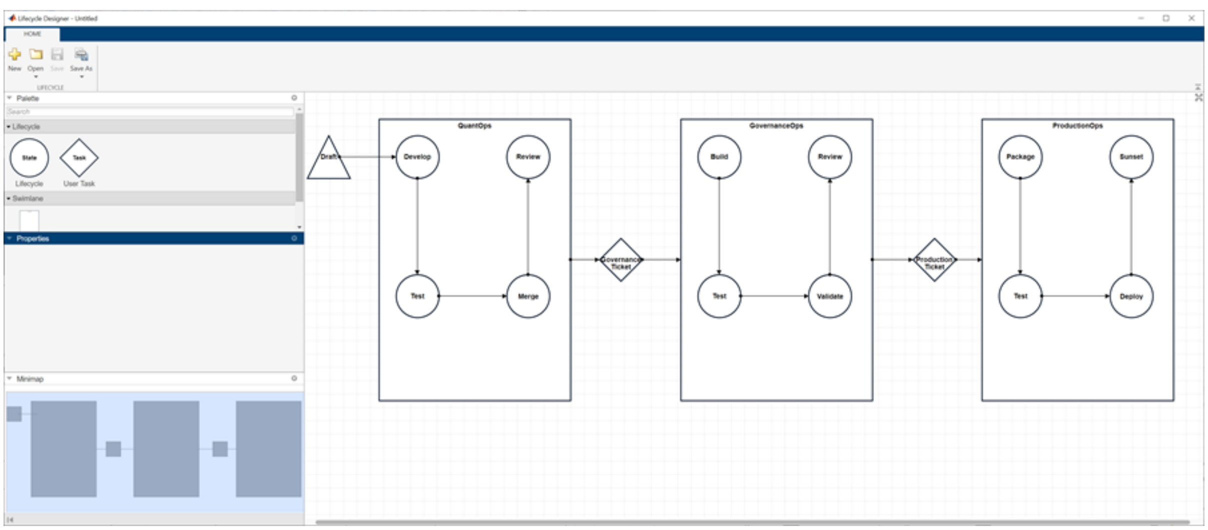
Description

Use the **Lifecycle Designer** app to create new lifecycles and edit existing lifecycles in Modelscape.

Each Modelscape model must be associated with a lifecycle and have multiple model versions. Modelscape models are quantitative methods that apply statistical, economic, or other techniques to given inputs to produce an output. A Modelscape model corresponds to a Git repository. A model version refers to a Git commit.

A lifecycle represents the steps of a model version from drafting and proposal to retiring and decommissioning. At any time, each model version is in one specific state of the model lifecycle. Using this app, you can:

- Create a new model lifecycle
- Open, view, and edit existing lifecycles



Open the Lifecycle Designer App

- Launch the **Lifecycle Designer** app from the Modelscape Home page. To host Modelscape for your organization, contact MathWorks Consulting Services.

Examples

Create New Lifecycle

Create a new lifecycle in Lifecycle Designer by selecting the **New** button on the menu.

To add a new state to the lifecycle, drag the **State** button to the corresponding location in the app. You can rename the state by double-clicking its text box.

To add a new decision to the lifecycle, drag the **Task** button to the corresponding location in the app. You can rename the task by double-clicking its text box.

Add connections between the different stages of the lifecycle. To add a connection from stage 1 to stage 2, click stage 1 and then hold **Ctrl** while you click stage 2.

To group different stages, drag a swimlane to cover the stages you want to group. You can also click and drag your mouse to select stages and click **Group** on the menu.

To ungroup stages in a swimlane, select the swimlane and click **Ungroup** on the menu.

To save your lifecycle for the first time, click `Save As...` under the **Save** dropdown menu. If you continue editing the lifecycle and want to save it under the same name, click `Save` under the **Save** dropdown menu.

Open and Edit Existing Lifecycle

Open, view, and edit a preexisting lifecycle in Lifecycle Designer.

Open an existing lifecycle by clicking the **Open** dropdown button on the menu and clicking the corresponding lifecycle.

Edit the lifecycle by adding or deleting new stages, grouping or ungrouping elements in swimlanes, and adding or removing connections between stages. When you are finished, save the lifecycle by clicking `Save` under the **Save** dropdown menu.

Version History

Introduced in R2023a

See Also

Functions

Topics

"Modelscape Governance" on page 5-4

Review Editor

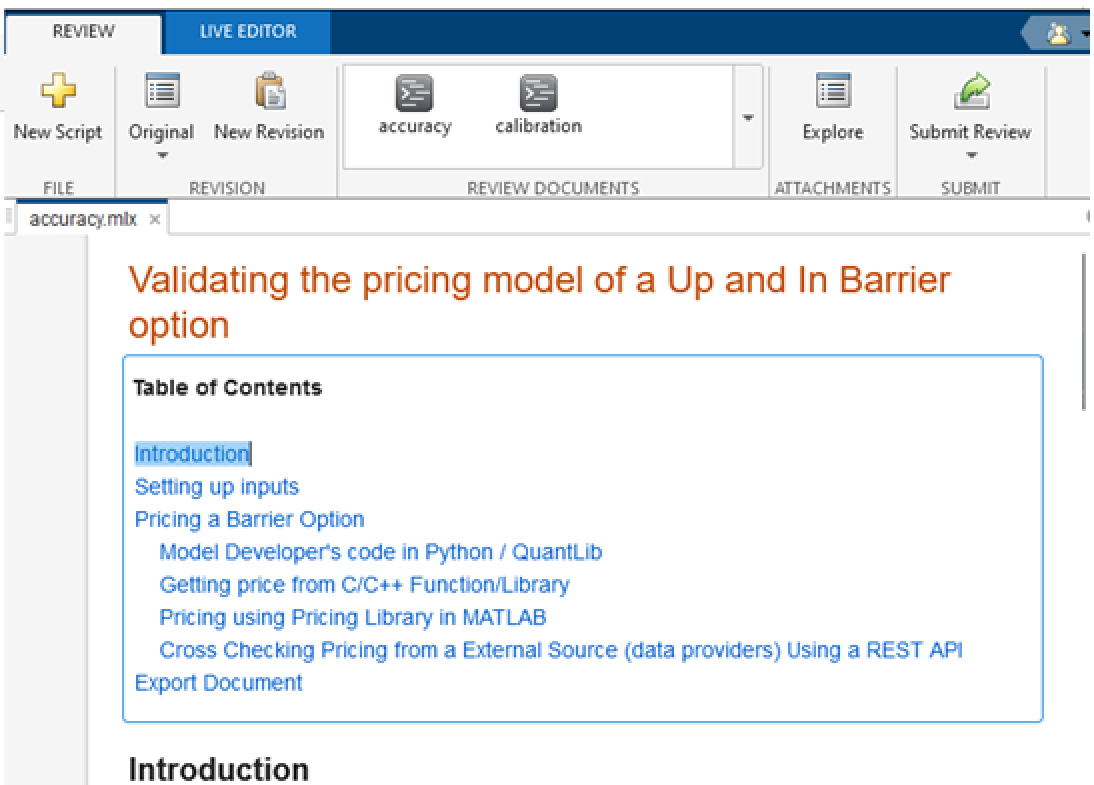
Validate Modelscape models and submit reviews

Description

Use the **Review Editor** app to validate and add reviews to Modelscape models.

After a model developer in your organization develops a model, that model can be proposed to be deployed for production. However, you must validate the model version first. You can validate the model version using the **Review Editor** app. Use the app to explore data and analyze the model version. You can record anomalies or observations related to models, make an appropriate closure measure, and export the findings to a PDF file. Use the app to perform any of these tasks:

- Open, view, and run models.
- Attach supporting documents.
- Make a review decision.



Open the Review Editor App

- To launch the **Review Editor** app, open the Modelscape Home page. To host Modelscape for your organization, contact MathWorks Consulting Services. On the Home page, click **Models** on the navigation bar and open the version of the model you want to review. Open the **Review Editor** app to review a specific model version by clicking **Open Review** next to the model version.

Examples

Analyze Model Version

Create a new live script by clicking the **New Script** button on the menu.

Use the live scripts to explore data and analyze the model versions. For instance, you can perform what-if analyses, which include changing the model parameters, using different fitting methods, and observing changes in the model performance.

To perform these analyses, you can use the suite of metrics in Modelscape. For more details, see “Fairness Metrics in Modelscape” on page 5-26. You can use these metrics to analyze the bias in your models.

You can also write scripts to implement model validation suites. For an example that shows how to validate a probability-of-default model, see “Credit Scorecard Validation Metrics” on page 5-48. This example shows how to use the techniques in the BCBS Working Paper 14.

If your model version uses a programming language other than MATLAB, you can still use the validation tools from Modelscape. For more information, see “Validation of External Models” on page 5-60.

For an example that shows how to validate credit models using the European Central Bank template, see “Validation of Credit Models in ECB Templates” on page 5-57.

After you perform your analyses, save your work by clicking **New Revision** and describing your changes.

Attach Documents to Review

To attach documents, click **Explore** on the menu, and upload file attachments using the **Upload** button. You can attach recommendations for improvement, evidence of model performance reports you generate outside the app, detailed model validation documents, and supporting scripts.

By default, the **Review Editor** attaches files to the model repository. The location to which you save documents is customizable at an organizational level. See “Extensibility” on page 5-15 for more details.

Make Review Decision

Finish your review by using the submit button and sign-off your form in the app. You can customize the **Submit Review** dropdown and sign-off forms based on the needs of your organization. For more details, see “Extensibility” on page 5-15.

By default, the Review Editor app provides a full review form and a reduced review form. Use the reduced review form to approve the model if the changes are trivial. If the changes are not trivial and have a significant material impact, for example on the value of trade, use the full review form.

You can choose to approve or reject the model version. The next stages depend on the model lifecycle. If you do not approve the model version, it could be sent back to the developer. If you approve the model version, it could be sent to production and deployment.

- “File Attachments in Modelscape Review Editor” on page 5-68
- “Customization of Signoff Forms in Review Editor” on page 5-70
- “Fairness Metrics in Modelscape” on page 5-26
- “Credit Scorecard Validation Metrics” on page 5-48
- “Metrics Handlers” on page 5-45
- “Validation of External Models” on page 5-60
- “Validation of Credit Models in ECB Templates” on page 5-57

Version History

Introduced in R2023a

See Also

Topics

“File Attachments in Modelscape Review Editor” on page 5-68
“Customization of Signoff Forms in Review Editor” on page 5-70
“Fairness Metrics in Modelscape” on page 5-26
“Credit Scorecard Validation Metrics” on page 5-48
“Metrics Handlers” on page 5-45
“Validation of External Models” on page 5-60
“Validation of Credit Models in ECB Templates” on page 5-57
“Modelscape Validate” on page 5-9

Remove Risk Factors

Remove or include data and record reasons in Modelscape

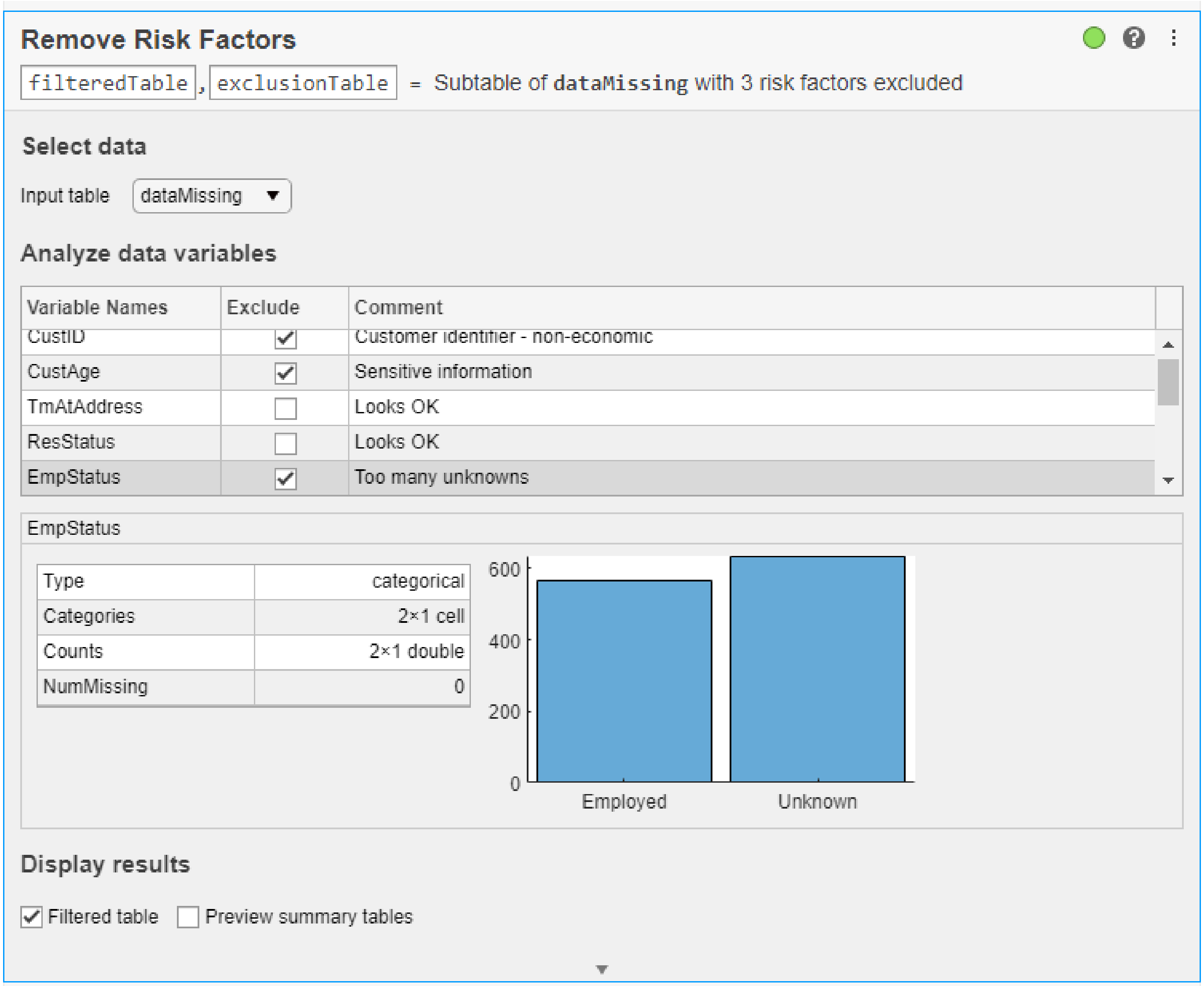
Description

Use the Modelscape **Remove Risk Factors** task to remove or include variables from a data table and record the corresponding reasons. Not all the data in the table is necessarily usable for a statistical model. For example, randomized user identifiers (IDs) are often irrelevant, legally sensitive data such as ethnic origin or religious beliefs cannot be used, and some data can be of poor quality. The task automatically generates MATLAB code for your live script. This task requires the Modelscape for MATLAB support package.

Using this task, you can:

- Inspect summary statistics and histograms for variables in a data table.
- Remove variables from a data table and record the corresponding reason for exclusion.
- Record reasons for including variables in a data table.
- Export the resulting subtables to MATLAB desktop.

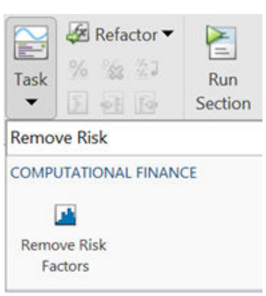
For general information about Live Editor tasks, see “Add Interactive Tasks to a Live Script”.



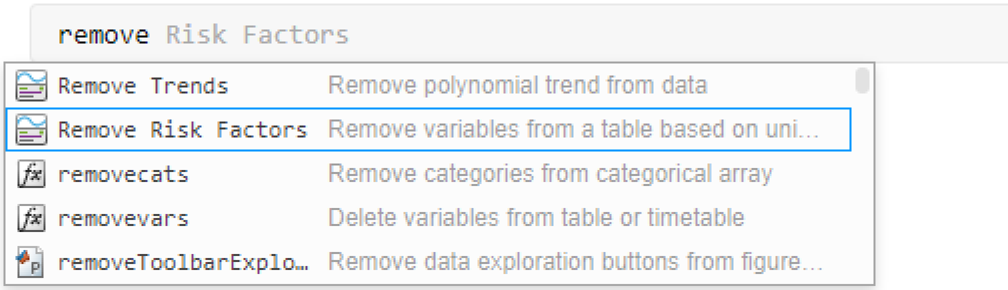
Open the Remove Risk Factors

To add the **Threshold Predictors** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Remove Risk Factors**.



- In a code block in the script, type a relevant keyword, such as `remove`. Select `Remove Risk Factors` from the suggested command completions.



Examples

- "Remove Risk Factors" on page 5-22

Parameters

`Input table` — Table of input data to inspect
table of input data containing variables to inspect

Input table must be a MATLAB table or a timetable. The columns of **Input table** contain the variables for different data points, for example, Residence Status or Customer ID.

`Filtered table` — Display table of filtered variables
check box to display subtable with excluded variables

Check the **Filtered table** check box to display the subtable after excluding the removed variables. The filtered table contains the columns from the **Input table** without the variables that you mark for exclusion.

`Preview summary tables` — Display tables of summary
check box to display two tables with summaries of variables and progress

Check the **Preview summary tables** check box to display two tables of additional information about the feature selection process. The **exclusionSummaryPreview** table includes all the data of the input table together with the exclusion flags and comments that you record in the task. The **progressSummaryPreview** table shows the total number of variables that are present, excluded, included, and commented against.

Version History

Introduced in R2021b

Topics

"Remove Risk Factors" on page 5-22

Screen Risk Factors

Remove risk factors from data in Modelscape

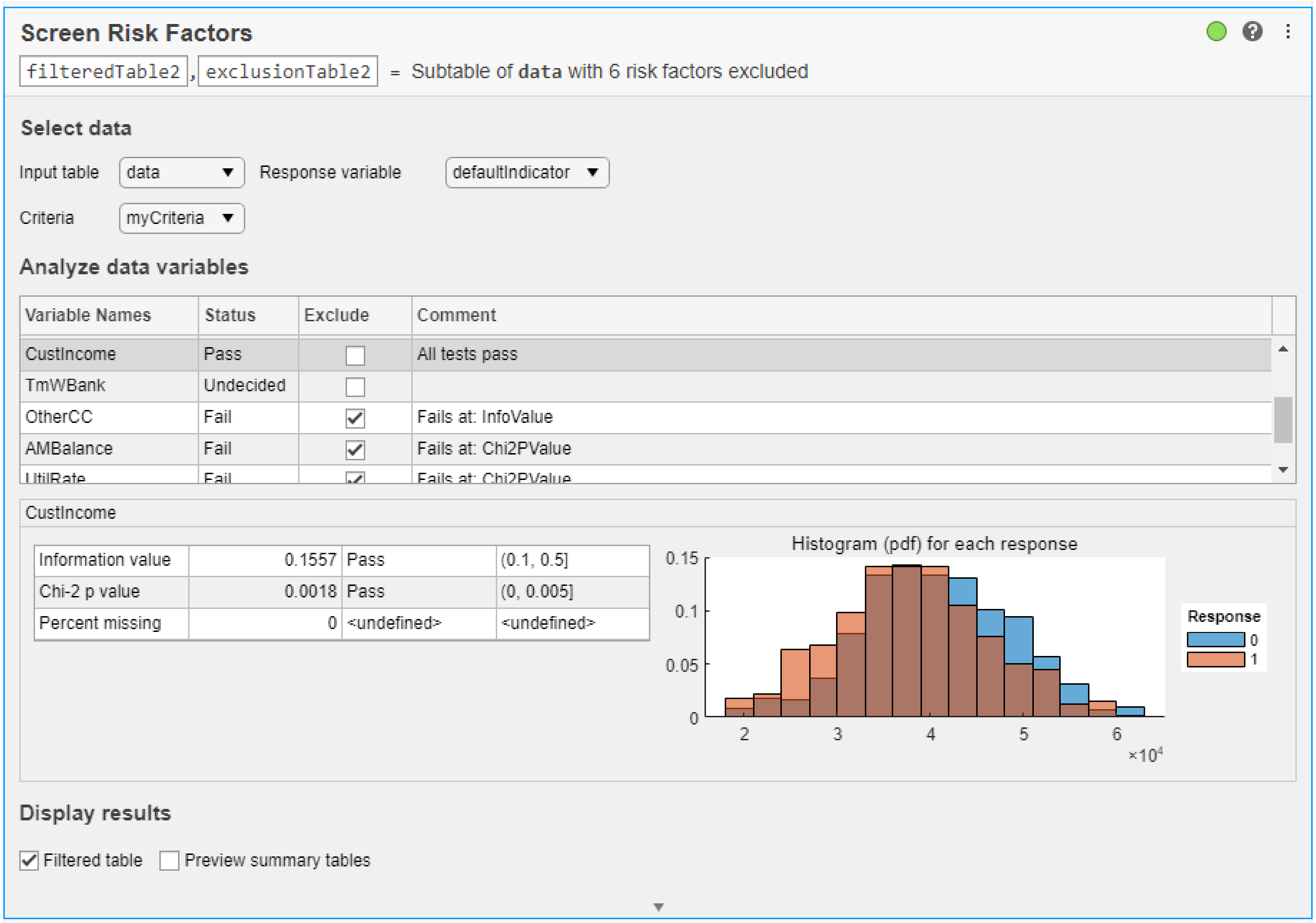
Description

Use the Modelscape **Screen Risk Factors** task to automatically remove risk factors from a data table based on their predictive power relative to a binary response variable. Feature selection is an important step in the development of a statistical model. Input data can have hundreds or thousands of variables, and discarding some variables often improves model interpretability, training times, and other important attributes. The task automatically generates MATLAB code for your live script. This task requires the Modelscape for MATLAB support package.

Using this task, you can:

- Inspect summary statistics and histograms for variables in a data table.
- Use customizable screening criteria to analyze the predictive power of variables.
- Remove variables from a data table and record the corresponding reason for exclusion.
- Record reasons for including variables in a data table.
- Export the resulting subtables to MATLAB desktop.

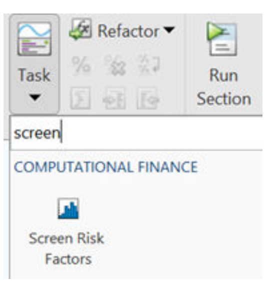
For general information about Live Editor tasks, see “Add Interactive Tasks to a Live Script”.



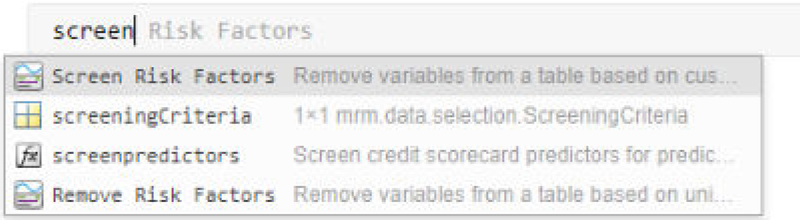
Open the Screen Risk Factors

To add the **Threshold Predictors** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Screen Risk Factors**.



- In a code block in the script, type a relevant keyword, such as `screen`. Select `Screen Risk Factors` from the suggested command completions.



Examples

- "Screen Risk Factors by Custom Criteria" on page 5-30

Parameters

`Input table` — Table of input data to inspect
table of input data containing variables to inspect

Input table must be a MATLAB table or a timetable. The columns of **Input table** contain the variables for different data points, for example, Residence Status or Customer ID.

`Response variable` — Binary variable in table
binary variable to use for prediction

Response variable must be a binary variable in the input table. The task evaluates the risk factors in the input data table based on their power to predict this response variable.

`Criteria` — Screening criteria to apply to input variables
`screeningCriteria` object

Criteria must be an object containing the criteria against which to screen the input variables. You can use the predefined criteria or customize your own screening criteria. For more details, see "Screen Risk Factors by Custom Criteria" on page 5-30.

`Filtered table` — Display table of filtered variables
check box to display subtable with excluded variables

Check the **Filtered table** check box to display the subtable after excluding the removed variables. The filtered table contains the columns from the **Input table** without the variables that you mark for exclusion.

`Preview summary tables` — Display tables of summary
check box to display two tables with summaries of variables and progress

Check the **Preview summary tables** check box to display two tables of additional information about the feature selection process. The **exclusionSummaryPreview** table includes all the data of the input table together with the exclusion flags and comments that you record in the task. The **progressSummaryPreview** table shows the total number of variables that are present, excluded, included, and commented against.

Version History

Introduced in R2021b

Topics

"Screen Risk Factors by Custom Criteria" on page 5-30

checkModel

Check Modelscape model validity

Syntax

```
result = mrm.execution.checkModel(model,inputData)
result = mrm.execution.checkModel(model,inputData,parameterData)
```

Description

`result = mrm.execution.checkModel(model,inputData)` checks the validity of a Modelscape model by creating several test methods. The function determines whether the Modelscape model has the right shape for the input of the original model. The function returns the result of the tests in `result`. This function requires the Modelscape for MATLAB support package.

`result = mrm.execution.checkModel(model,inputData,parameterData)` specifies additional parameters for the model.

Examples

Check Validity of Modelscape implementation of Generalized Linear Model (GLM)

Create a Modelscape model as a subclass of the `mrm.execution.Model` class.

The model takes the inputs `X1`, and `X2`, as well as an optional parameter `intercept`. The function computes the output `Y` as the value of `exp(intercept + alpha1*X1 + alpha2*X2)`.

The Modelscape model must also implement three methods for specifying the inputs and the outputs of the model: `getInputs`, `getParameters`, and `getOutputs`. For more details, see the “Model Implementation for Modelscape Deploy” on page 5-74

```
classdef glmModel < mrm.execution.Model
    %   implement simple GLM with two inputs X1 and X2 that computes
    %   Y = exp(intercept + alpha1*X1 + alpha2*X2)
    properties
        alpha1 = 1
        alpha2 = 2
    end


    methods

         function parameters = getInputs(~)
            doubleDatatype = struct( ...
                "name", "double");
            parameters = struct( ...
                "name", {"X1","X2"}, ...
                "dataType", {doubleDatatype, doubleDatatype});
        end
```

```
        function parameters = getParameters(~)
            doubleDatatype = struct( ...
                "name", "double");
            parameters = struct( ...
                "name", {"intercept"}, ...
                "dataType", {doubleDatatype});
        end

        function parameters = getOutputs(~)
            doubleDatatype = struct( ...
                "name", "double");
            parameters = struct( ...
                "name", {"Y"}, ...
                "dataType", {doubleDatatype});
        end

        function [outputs,diagnostics,batchDiagnostics] = evaluate(this,inputs,parameters)
            outputs = table( ...
                exp(parameters.intercept +  this.alpha1*inputs.X1 + this.alpha2*inputs.X2), ...
                'VariableNames', {'Y'}, ...
                'RowNames', inputs.Properties.RowNames);
            rawDiagnostics = [inputs.Properties.RowNames repmat({struct()} numel(inputs.Propertie
            diagnostics = struct(rawDiagnostics{:});
            batchDiagnostics = struct();
        end

    end
end
```

Validate the custom model.

```
result = mrm.execution.checkModel(model, inputData, parameterData);
```

Input Arguments

`model` — Modelscape model
subclass of `mrm.execution.Model`

Modelscape model, specified as a subclass of the `mrm.execution.Model` class. This model must also implement three methods for specifying the inputs and the outputs of the model: `getInputs`, `getParameters`, and `getOutputs`. For more details, see “Model Implementation for Modelscape Deploy” on page 5-74.

`inputData` — Original model inputs
table

Original model inputs, specified as a table.

Data Types: `table`

`parameterData` — Original model parameters
structure

Original model parameters, specified as a structure.

Data Types: `struct`

Output Arguments

`result` — Test results
`matlab.unittest.TestResult` class

Test results, returned as a `matlab.unittest.TestResult` class. The results include information describing whether the test pass, fail, or run to completion, as well as the duration of each test.

Version History

Introduced in R2023a

See Also

"Modelscape Deploy" on page 5-13

Topics

"Model Implementation for Modelscape Deploy" on page 5-74

packageModel

Create Docker image of Modelscape model

Syntax

```
imageName = mrm.execution.compiler.packageModel(model)
imageName = mrm.execution.compiler.packageModel(model,Name=Value)
```

Description

`imageName = mrm.execution.compiler.packageModel(model)` creates a Docker image of the model `model`. This function requires the Modelscape for MATLAB support package.

`imageName = mrm.execution.compiler.packageModel(model,Name=Value)` specifies options using one or more name-value arguments in addition to the input argument in the previous syntax.

Examples

Create Docker Image of Modelscape Model

Create a Modelscape model.

Specify the model name and the location and tag of the image.

Create a Docker image of the model, specifying the model name and version.

```
imageName = mrm.execution.compiler.packageModel(modelInstance,
    Name="weighted-sum",
    Tag="v1");
```

Input Arguments

`model` — Modelscape model
subclass of `mrm.execution.Model`

Modelscape model, specified as a subclass of the `mrm.execution.Model` class. This model must also implement three methods for specifying the inputs and the outputs of the model: `getInputs`, `getParameters`, and `getOutputs`. For more details, see "Model Implementation for Modelscape Deploy" on page 5-74.

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

`Registry` — Registry of Docker Image
`''` (default) | character vector | string scalar

Registry of the Docker image name, specified as a character vector or string scalar. The function creates a Docker image with a name in the format `Registry/Repository/Name:Tag`.

Data Types: `char` | `string`

`Repository` — Repository of Docker Image
`''` (default) | character vector | string scalar

Repository of the Docker image name, specified as a character vector or string scalar. The function creates a Docker image with a name in the format `Registry/Repository/Name:Tag`.

Data Types: `char` | `string`

`Name` — Name of Docker Image
`Modelscape model class name` (default) | character vector | string scalar

Name of the Docker image, specified as a character vector or string scalar. The function creates a Docker image with a name in the format `Registry/Repository/Name:Tag`.

Data Types: `char` | `string`

`Tag` — Tag of Docker Image
`''` (default) | character vector | string scalar

Tag of the Docker image name, specified as a character vector or string scalar. The function creates a Docker image with a name in the format `Registry/Repository/Name:Tag`.

Data Types: `char` | `string`

Output Arguments

`imageName` — Name of Docker image
string scalar

Name of the Docker image in the local Docker registry, returned as a string scalar.

Version History
Introduced in R2023a

See Also
“Modelscape Deploy” on page 5-13

Topics
“Model Implementation for Modelscape Deploy” on page 5-74

Functions

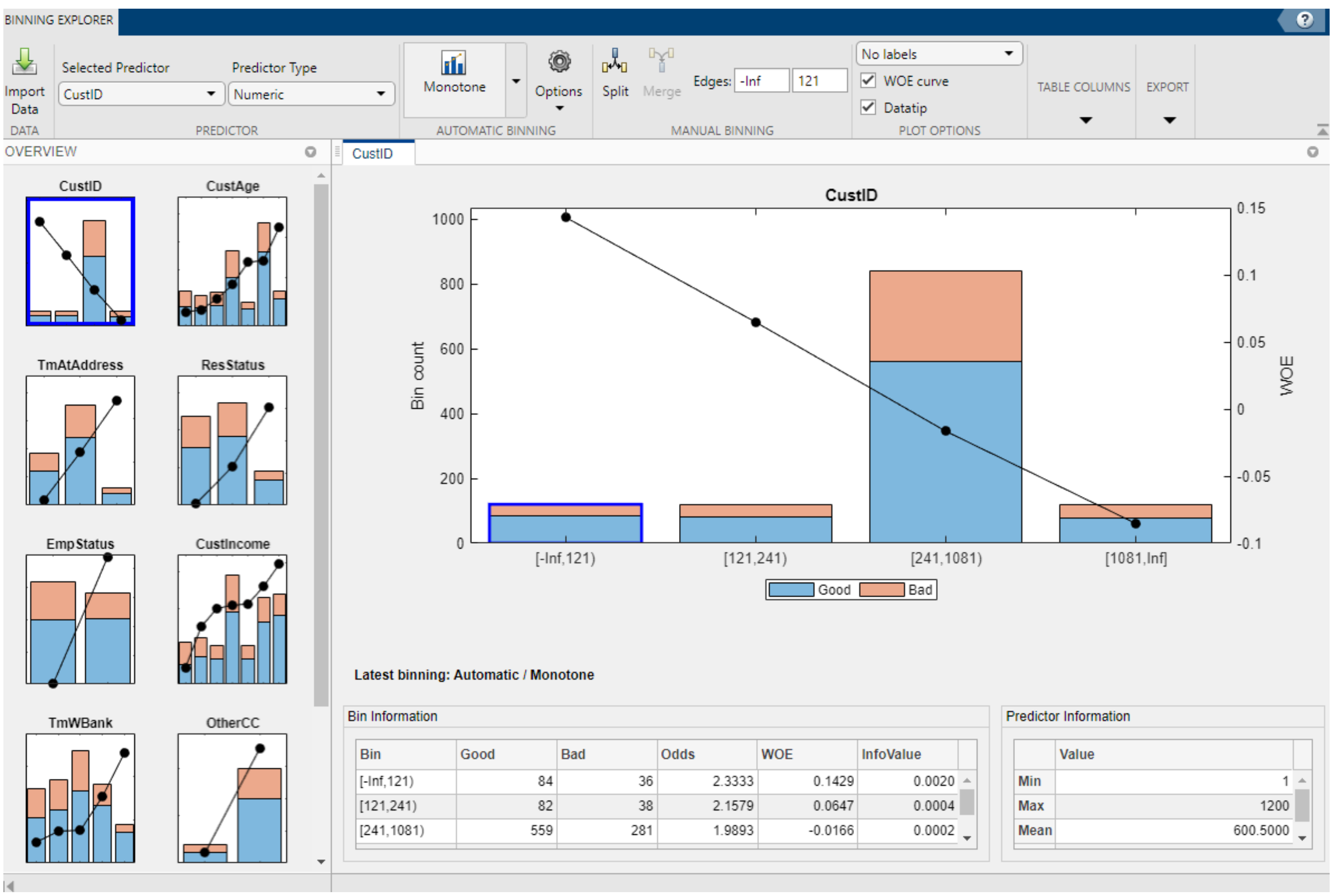
Binning Explorer

Bin data and export into a `creditscorecard` object

Description

The **Binning Explorer** app enables you to manage binning categories for a `creditscorecard` object. Use `screenpredictors` to pare down a potentially large set of predictors to a subset that is most predictive of the credit score card response variable. You can then use this subset of predictors when creating a MATLAB table of data. After creating a table of data in your MATLAB workspace, or after using `creditscorecard` to create a `creditscorecard` object, use the **Binning Explorer** to:

- Select an automatic binning algorithm with an option to bin missing data. (For more information on algorithms for automatic binning, see `autobinning`.)
- Shift bin boundaries.
- Split bins.
- Merge bins.
- Save and export a `creditscorecard` object.



Open the Binning Explorer App

- MATLAB toolstrip: On the **Apps** tab, under **Computational Finance**, click the app icon.
- MATLAB command prompt:
  - Enter `binningExplorer` to open the **Binning Explorer** app.
  - Enter `binningExplorer(data)` or `binningExplorer(data,Name,Value)` to open a table in the **Binning Explorer** app by specifying a table (`data`) as input.
  - Enter `binningExplorer(sc)` to open a `creditscorecard` object in the **Binning Explorer** app by specifying a `creditscorecard` object (`sc`) as input.

To access Help for the App, click the Help icon on the toolbar.

Examples

- “Overview of Binning Explorer” on page 3-2
- “Feature Screening with screenpredictors” on page 3-64
- “Common Binning Explorer Tasks” on page 3-4
- “Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23
- “Case Study for Credit Scorecard Analysis”
- “Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36

Version History

Introduced in R2016b

See Also

Functions
`screenpredictors` | `creditscorecard` | `autobinning`

Topics
“Overview of Binning Explorer” on page 3-2
“Feature Screening with screenpredictors” on page 3-64
“Common Binning Explorer Tasks” on page 3-4
“Bin Data to Create Credit Scorecards Using Binning Explorer” on page 3-23
“Case Study for Credit Scorecard Analysis”
“Stress Testing of Consumer Credit Default Probabilities Using Panel Data” on page 3-36
“Overview of Binning Explorer” on page 3-2
“Credit Scorecard Modeling Workflow”

External Websites
Credit Scorecard Modeling Using the Binning Explorer App (6 min 17 sec)

asrf

Asymptotic Single Risk Factor (ASRF) capital

Syntax

```
[capital,VaR] = asrf(PD,LGD,R)
[capital,VaR] = asrf( ___ ,Name,Value)
```

Description

`[capital,VaR] = asrf(PD,LGD,R)` computes regulatory capital and value-at-risk using an ASRF model.

The ASRF model is useful because the Basel II documents propose this model as the standard for certain types of capital requirements. ASRF is not a Monte-Carlo model, so you can quickly compute the capital requirements for large credit portfolios. You can use the ASRF model to perform a quick sensitivity analysis and exploring "what-if" scenarios more easily than rerunning large simulations.

`[capital,VaR] = asrf( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute Necessary Capital Using an ASRF Model

Load saved portfolio data.

```
load CreditPortfolioData.mat
```

Compute asset correlation for corporate, sovereign, and bank exposures.

```
R = 0.12 * (1-exp(-50*PD)) / (1-exp(-50)) +...
    0.24 * (1 - (1-exp(-50*PD)) / (1-exp(-50)));
```

Compute the asymptotic single risk factor capital. By specifying the name-value pair argument for `EAD`, the `capital` is returned in terms of currency.

```
capital = asrf(PD,LGD,R,'EAD',EAD);
```

Apply a maturity adjustment.

```
b = (0.11852 - 0.05478 * log(PD)).^2;
matAdj = (1 + (Maturity - 2.5) .* b) ./ (1 - 1.5 * b);
adjustedCapital = capital .* matAdj;

portfolioCapital = sum(adjustedCapital)
```

```
portfolioCapital = 175.7865
```

Input Arguments

PD — Probability of default
numeric vector with elements from 0 to 1

Probability of default, specified as a NumCounterparties-by-1 numeric vector with elements from 0 to 1, representing the default probabilities for the counterparties.

Data Types: double

LGD — Loss given default
numeric vector with elements from 0 to 1

Loss given default, specified as a NumCounterparties-by-1 numeric vector with elements from 0 to 1, representing the fraction of exposure that is lost when a counterparty defaults. LGD is defined as (1 - *Recovery*). For example, an LGD of 0.6 implies a 40% recovery rate in the event of a default.

Data Types: double

R — Asset correlation
numeric vector

Asset correlation, specified as a NumCounterparties-by-1 numeric vector.

The asset correlations, R, have values from 0 to 1 and specify the correlation between assets in the same asset class.

Note The correlation between an asset value and the underlying single risk factor is sqrt(R). This value, sqrt(R), corresponds to the Weights input argument to the creditDefaultCopula and creditMigrationCopula classes for one-factor models.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: capital = asrf(PD,LGD,R,'EAD',EAD)

EAD — Exposure at default
1 (default) | numeric vector

Exposure at default, specified as the comma-separated pair consisting of 'EAD' and a NumCounterparties-by-1 numeric vector of credit exposures.

If EAD is not specified, the default EAD is 1, meaning that capital and VaR results are reported as a percentage of the counterparty's exposure. If EAD is specified, then capital and VaR are returned in units of currency.

Data Types: double

VaRLevel — Value at risk level
0.999 (99.9%) (default) | decimal value between 0 and 1

Value at risk level used when calculating the capital requirement, specified as the comma-separated pair consisting of 'VaRLevel' and a decimal value between 0 and 1.

Data Types: double

Output Arguments

capital — Capital for each element in portfolio
vector

Capital for each element in the portfolio, returned as a NumCounterparties-by-1 vector. If the optional input EAD is specified, then capital is in units of currency. Otherwise, capital is reported as a percentage of each exposure.

VaR — Value-at-risk for each exposure
vector

Value-at-risk for each exposure, returned as a NumCounterparties-by-1 vector. If the optional input EAD is specified, then VaR is in units of currency. Otherwise, VaR is reported as a percentage of each exposure.

More About

ASRF Model Capital

In the ASRF model, capital is defined as the loss in excess of the expected loss (EL) at a high confidence level.

The formula for capital is

```
capital = VaR - EL
```

Algorithms

The capital requirement formula for exposures is defined as

$$VaR = EAD * LGD * \Phi\left(\frac{\Phi^{-1}(PD) - \sqrt{R}\Phi^{-1}(1 - VaRLevel)}{\sqrt{1 - R}}\right)$$
$$capital = VaR - EAD * LGD * PD$$

where

ϕ is the normal CDF.

ϕ^{-1} is the inverse normal CDF.

R is asset correlation.

EAD is exposure at default.

PD is probability of default.

LGD is loss given default.

Version History

Introduced in R2017b

References

[1] Basel Committee on Banking Supervision. "*International Convergence of Capital Measurement and Capital Standards*." June, 2006 (https://www.bis.org/publ/bcbs128.pdf).

[2] Basel Committee on Banking Supervision. "*An Explanatory Note on the Basel II IRB Risk Weight Functions*." July, 2005 (https://www.bis.org/bcbs/irbriskweight.pdf).

[3] Gordy, M.B. "A Risk-Factor Model Foundation for Ratings-Based Bank Capital Rules." *Journal of Financial Intermediation.* Vol. 12, pp. 199-232, 2003.

See Also

`creditDefaultCopula` | `creditMigrationCopula`

Topics

"Calculating Regulatory Capital with the ASRF Model" on page 4-59

bin

Binomial test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = bin(vbt)
TestResults = bin(vbt,Name,Value)
```

Description

`TestResults = bin(vbt)` generates the binomial test results for value-at-risk (VaR) backtesting.

`TestResults = bin(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Generate Bin Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `bin` test results.

```
TestResults = bin(vbt)
```

```
TestResults=1×9 table
    PortfolioID    VaRID    VaRLevel     Bin      ZScoreBin    PValueBin    Observations    Failu
    ___________    _____    ________    ______    _________    _________    ____________    _____

    "Portfolio"    "VaR"      0.95      accept     0.68905      0.49079         1043          57
```

Run Bin Test for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `bin` test results using the `TestLevel` optional argument.

```
TestResults = bin(vbt,'TestLevel',0.90)
```

```
TestResults=6×9 table
    PortfolioID        VaRID          VaRLevel     Bin       ZScoreBin     PValueBin     Observation
    ___________    ______________    ________    ______    _________    _________    ___________

     "Equity"      "Normal95"          0.95      accept     0.68905       0.49079         1043
     "Equity"      "Normal99"          0.99      reject      2.0446      0.040896         1043
     "Equity"      "Historical95"      0.95      accept      0.9732       0.33045         1043
     "Equity"      "Historical99"      0.99      accept     0.48858       0.62514         1043
     "Equity"      "EWMA95"            0.95      accept      0.9732       0.33045         1043
     "Equity"      "EWMA99"            0.99      reject      3.6006     0.0003175         1043
```

Input Arguments

`vbt` — `varbacktest` object

object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = bin(vbt,'TestLevel',0.99)`

`TestLevel` — Test confidence level

`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — Bin test results
table

Bin test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for corresponding VaR data column
- `'Bin'` — Categorical array with categories `accept` and `reject` that indicate the result of the `bin` test
- `'ZScoreBin'` — Z-score of the number of failures
- `'PValueBin'` — P-value of the `bin` test
- `'Observations'` — Number of observations
- `'Failures'` — Number of failures.
- `'TestLevel'` — Test confidence level.

Note For `bin` test results, the terms `accept` and `reject` are used for convenience, technically a `bin` test does not accept a model. Rather, the test fails to reject it.

More About

Binomial Test (Bin)

The `bin` function performs a binomial test to assess if the number of failures is consistent with the VaR confidence level.

The binomial test is based on a normal approximation to the binomial distribution.

Algorithms

The result of the binomial test is based on a normal approximation to a binomial distribution. Suppose:

- N is the number of observations.
- p = `1 - VaRLevel` is the probability of observing a failure if the model is correct.
- x is the number of failures.

If the failures are independent, then the number of failures is distributed as a binomial distribution with parameters N and p. The expected number of failures is $N*p$, and the standard deviation of the number of failures is

$$\sqrt{Np(1-p)}$$

The test statistic for the `bin` test is the z-score, defined as:

$$ZScoreBin = \frac{(x - Np)}{\sqrt{Np(1 - p)}}$$

The z-score approximately follows a standard normal distribution. This approximation is not reliable for small values of N or small values of p, but for typical uses in VaR backtesting analyses (N = `250` or much larger,p in the range 1 -10%) the approximation gives results in line with other tests.

The tail probability of the `bin` test is the probability that a standard normal distribution exceeds the absolute value of the z-score

$$TailProbability = 1 - F(|ZScoreBin|)$$

where F is the standard normal cumulative distribution. When too few failures are observed, relative to the expected failures, *PValueBin* is (approximately) the probability of observing that many failures or fewer. For too many failures, this is (approximately) the probability of observing that many failures or more.

The p-value of the `bin` test is defined as two times the tail probability. This is because the binomial test is a two-sided test. If *alpha* is defined as 1 minus the test confidence level, the test rejects if the tail probability is less than one half of *alpha*, or equivalently if

$$PValueBin = 2 * \ TailProbability < alpha$$

Version History

Introduced in R2016b

References

[1] Jorion, P. *Financial Risk Manager Handbook*. 6th Edition. Wiley Finance, 2011.

See Also

`varbacktest` | `tl` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Topics

“VaR Backtesting Workflow” on page 2-6
“Value-at-Risk Estimation and Backtesting” on page 2-10
“Overview of VaR Backtesting” on page 2-2
“Binomial Test” on page 2-2
“Comparison of ES Backtesting Methods” on page 2-26

cc

Conditional coverage mixed test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = cc(vbt)
TestResults = cc(vbt,Name,Value)
```

Description

`TestResults = cc(vbt)` generates the conditional coverage (CC) mixed test for value-at-risk (VaR) backtesting.

`TestResults = cc(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Generate CC Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt =
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `cc` test results.

```
TestResults = cc(vbt)
```

```
TestResults=1×19 table
    PortfolioID    VaRID    VaRLevel     CC      LRatioCC    PValueCC     POF      LRatioPOF
    ___________    _____    ________    ______    ________    ________    ______    _________

    "Portfolio"    "VaR"      0.95      accept    0.72013     0.69763     accept     0.46147
```

Run the CC Test for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `cc` test results using the `TestLevel` optional input.

```
TestResults = cc(vbt,'TestLevel',0.90)
```

```
TestResults=6×19 table
    PortfolioID        VaRID          VaRLevel      CC       LRatioCC     PValueCC       POF      LRa
    ___________    ______________    ________    ______    ________    _________    ______    ___

     "Equity"      "Normal95"          0.95      accept    0.72013       0.69763    accept     0
     "Equity"      "Normal99"          0.99      accept     4.0757       0.13031    reject     3
     "Equity"      "Historical95"      0.95      accept     1.0487       0.59194    accept     0
     "Equity"      "Historical99"      0.99      accept     0.5073       0.77597    accept     0
     "Equity"      "EWMA95"            0.95      accept    0.95051       0.62173    accept     0
     "Equity"      "EWMA99"            0.99      reject     10.779     0.0045645    reject     9
```

Input Arguments

vbt — varbacktest object
object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = cc(vbt,'TestLevel',0.99)`

TestLevel — Test confidence level
`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — `cc` test results
table

`cc` test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for corresponding VaR data column
- `'CC'` — Categorical array with the categories `accept` and `reject` that indicate the result of the cc test
- `'LRatioCC'` — Likelihood ratio of the `cc` test
- `'PValueCC'` — P-value of the `cc` test
- `'POF'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `pof` test
- `'LRatioPOF'` — Likelihood ratio of the `pof` test
- `'PValuePOF'` — P-value of the `pof` test
- `'CCI'` — Categorical array with categories `'accept'` and `'reject'` that indicate the result of the `cci` test
- `'LRatioCCI'` — Likelihood ratio of the `cci` test
- `'PValueCCI'` — P-value of the `cci` test
- `'Observations'` — Number of observations
- `'Failures'` — Number of failures
- `'N00'` — Number of periods with no failures followed by a period with no failures
- `'N10'` — Number of periods with failures followed by a period with no failures
- `'N01'` — Number of periods with no failures followed by a period with failures
- `'N11'` — Number of periods with failures followed by a period with failures
- `'TestLevel'` — Test confidence level

Note For `cc` test results, the terms `accept` and `reject` are used for convenience, technically a `cc` test does not accept a model. Rather, the test fails to reject it.

More About

Conditional Coverage (CC) Mixed Test

The `cc` function performs the conditional coverage mixed test, also known as Christoffersen's interval forecasts method.

'Mixed' means that it combines a frequency and an independence test. The frequency test is Kupiec's proportion of failures test, implemented by the `pof` function. The independence test is the conditional coverage independence test implemented by the `cci` function. This is a likelihood ratio test proposed by Christoffersen (1998) to assess the independence of failures on consecutive time periods. The CC test combines the POF test and the CCI test.

Algorithms

The likelihood ratio (test statistic) of the `cc` test is the sum of the likelihood ratios of the `pof` and `cci` tests,

$$LRatioCC = LRatioPOF + LRatioCCI$$

which is asymptotically distributed as a chi-square distribution with 2 degrees of freedom. See the Algorithms section in `pof` and `cci` for the definition of their likelihood ratios.

The *p*-value of the `cc` test is the probability that a chi-square distribution with 2 degrees of freedom exceeds the likelihood ratio *LRatioCC*,

$$PValueCC = 1 - F(LRatioCC)$$

where F is the cumulative distribution of a chi-square variable with 2 degrees of freedom.

The result of the `cc` test is to accept if

$$F(LRatioCC) < F(TestLevel)$$

and reject otherwise, where F is the cumulative distribution of a chi-square variable with 2 degrees of freedom.

Version History

Introduced in R2016b

References

[1] Christoffersen, P. "Evaluating Interval Forecasts." *International Economic Review.* Vol. 39, 1998, pp. 841 – 862.

See Also

`varbacktest` | `tl` | `tuff` | `bin` | `pof` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Topics

"VaR Backtesting Workflow" on page 2-6
"Value-at-Risk Estimation and Backtesting" on page 2-10
"Overview of VaR Backtesting" on page 2-2
"Christoffersen's Interval Forecast Tests" on page 2-4
"Comparison of ES Backtesting Methods" on page 2-26

cci

Conditional coverage independence test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = cci(vbt)
TestResults = cci(vbt,Name,Value)
```

Description

`TestResults = cci(vbt)` generates the conditional coverage independence (CCI) for value-at-risk (VaR) backtesting.

`TestResults = cci(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Generate CCI Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `cci` test results.

```
TestResults = cci(vbt)
```

```
TestResults=1×13 table
    PortfolioID    VaRID    VaRLevel     CCI      LRatioCCI    PValueCCI    Observations    Failu
    ___________    _____    ________    ______    _________    _________    ____________    _____

    "Portfolio"    "VaR"      0.95      accept     0.25866      0.61104         1043           57
```

Run the CCI Test for VaR Backtests for Multiple VaR's at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `cci` test results using the `TestLevel` optional input.

```
TestResults = cci(vbt,'TestLevel',0.90)
```

```
TestResults=6×13 table
    PortfolioID        VaRID         VaRLevel     CCI      LRatioCCI    PValueCCI    Observation
    ___________    ______________    ________    ______    _________    _________    ___________

     "Equity"      "Normal95"          0.95      accept     0.25866      0.61104         1043
     "Equity"      "Normal99"          0.99      accept     0.56393      0.45268         1043
     "Equity"      "Historical95"      0.95      accept     0.13847      0.70981         1043
     "Equity"      "Historical99"      0.99      accept     0.27962      0.59695         1043
     "Equity"      "EWMA95"            0.95      accept    0.040277      0.84094         1043
     "Equity"      "EWMA99"            0.99      accept     0.94909      0.32995         1043
```

Input Arguments

vbt — varbacktest object
object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = cci(vbt,'TestLevel',0.99)`

TestLevel — Test confidence level
`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — `cci` test results
table

`cci` test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'CCI'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `cci` test
- `'LRatioCCI'` — Likelihood ratio of the `cci` test
- `'PValueCCI'` — P-value of the `cci` test
- `'Observations'` — Number of observations
- `'Failures'` — Number of failures
- `'N00'` — Number of periods with no failures followed by a period with no failures
- `'N10'` — Number of periods with failures followed by a period with no failures
- `'N01'` — Number of periods with no failures followed by a period with failures
- `'N11'` — Number of periods with failures followed by a period with failures
- `'TestLevel'` — Test confidence level

Note For `cci` test results, the terms `accept` and `reject` are used for convenience, technically a `cci` test does not accept a model. Rather, the test fails to reject it.

More About

Conditional Coverage Independence (CCI) Test

The `cci` function performs the conditional coverage independence test.

This is a likelihood ratio test proposed by Christoffersen (1998) to assess the independence of failures on consecutive time periods. For the conditional coverage mixed test, see the `cc` function.

Algorithms

To define the likelihood ratio (test statistic) of the `cc` test, first define the following quantities:

- `'N00'` — Number of periods with no failures followed by a period with no failures
- `'N10'` — Number of periods with failures followed by a period with no failures
- `'N01'` — Number of periods with no failures followed by a period with failures
- `'N11'` — Number of periods with failures followed by a period with failures

Then define the following conditional probability estimates:

- `p01` = Probability of having a failure on period t, given that there was no failure on period t - `1`

$$p01 = \frac{\mathrm{N01}}{(\mathrm{N00} + \mathrm{N01})}$$

- `p11` = Probability of having a failure on period t, given that there was a failure on period t - `1`

$$p11 = \frac{\mathrm{N11}}{(\mathrm{N10} + \mathrm{N11})}$$

Define also the unconditional probability estimate of observing a failure:

pUC = Probability of having a failure on period *t*

$$pUC = \frac{(\mathrm{N01} + \mathrm{N11})}{(\mathrm{N00} + \mathrm{N01} + \mathrm{N10} + \mathrm{N11})}$$

The likelihood ratio of the CCI test is then given by

$$LRatioCCI = -2\log\left(\frac{(1 - pUC)^{N00 + N10}pUC^{N01 + N11}}{(1 - p01)^{N00}p01^{N01}(1 - p11)^{N10}p11^{N11}}\right)$$
$$= -2((\mathrm{N00}{+}\mathrm{N10})\log(1 - pUC) + (\mathrm{N01}{+}\mathrm{N11})\log(pUC) - \mathrm{N00}\log(1 - p01) - \mathrm{N01}\log(p01) - \mathrm{N10}\log(1 - p11)$$

which is asymptotically distributed as a chi-square distribution with 1 degree of freedom.

The *p*-value of the CCI test is the probability that a chi-square distribution with 1 degree of freedom exceeds the likelihood ratio `LRatioCCI`,

$$PValueCCI = 1 - F(LRatioCCI)$$

where *F* is the cumulative distribution of a chi-square variable with 1 degree of freedom.

The result of the test is to accept if

$$F(LRatioCCI) < F(TestLevel)$$

and reject otherwise, where *F* is the cumulative distribution of a chi-square variable with 1 degree of freedom.

If one or more of the quantities `N00`, `N10`, `N01`, or `N11` are zero, the likelihood ratio is handled differently. The likelihood ratio as defined above is composed of three likelihood functions of the form

$$L = (1 - p)^{n1} \times p^{n2}$$

For example, in the numerator of the likelihood ratio, there is a likelihood function of the form *L* with *p* = *pUC*, *n1* = `N00` + `N10`, and *n2* = `N01` + `N11`. There are two such likelihood functions in the denominator of the likelihood ratio.

It can be shown that whenever *n1* = `0` or *n2* = `0`, the likelihood function *L* is replaced by the constant value `1`. Therefore, whenever `N00`, `N10`, `N01`, or `N11` is zero, replace the corresponding likelihood functions by `1` in the likelihood ratio, and the likelihood ratio is well-defined.

Version History

Introduced in R2016b

References

[1] Christoffersen, P. "Evaluating Interval Forecasts." *International Economic Review.* Vol. 39, 1998, pp. 841 – 862.

See Also

`varbacktest` | `tl` | `tuff` | `bin` | `pof` | `cc` | `tbf` | `tbfi` | `summary` | `runtests`

Topics

"VaR Backtesting Workflow" on page 2-6
"Value-at-Risk Estimation and Backtesting" on page 2-10
"Overview of VaR Backtesting" on page 2-2
"Christoffersen's Interval Forecast Tests" on page 2-4
"Comparison of ES Backtesting Methods" on page 2-26

cdfSummary

Compute CDFs to ultimate claims for `developmentTriangle` object

Syntax

```
selectedLinkRatiosTable = cdfSummary(developmentTriangle)
```

Description

`selectedLinkRatiosTable = cdfSummary(developmentTriangle)` calculates the cumulative development factors (CDFs) and the percentage of total claims.

Examples

Calculate CDFs and Percentage of Total Claims for Development Triangle

Calculate the CDFs and the percentage of total claims for a `developmentTriangle` object using simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

Use `linkRatioAverages` function to calculate the different link ratio averages.

```
LinkRatioAveragesTable = linkRatioAverages(dT)
```

```
LinkRatioAveragesTable=8×9 table
                                         12-24     24-36     36-48     48-60     60-72     72-84
                                         ______    ______    ______    ______    ______    _____

    Simple Average                       1.1767    1.0563    1.0249    1.0107    1.0054    1.003
    Simple Average - Latest 5             1.172     1.056    1.0268    1.0108    1.0054    1.003
    Simple Average - Latest 3              1.17    1.0533     1.027    1.0117    1.0057    1.003
    Medial Average - Latest 5x1          1.1733    1.0567    1.0267    1.0103     1.005     1.00
    Volume-weighted Average              1.1766    1.0563     1.025    1.0107    1.0054    1.003
    Volume-weighted Average - Latest 5    1.172     1.056    1.0268    1.0108    1.0054    1.003
    Volume-weighted Average - Latest 3   1.1701    1.0534     1.027    1.0117    1.0057    1.003
    Geometric Average - Latest 4           1.17     1.055    1.0267     1.011    1.0055    1.003
```

Use the `cdfSummary` function to calculate CDFs and the percentage of total claims and return a table with the selected link ratios, CDFs, and percent of total claims.

```
dT.SelectedLinkRatio = [1.1755, 1.0577, 1.0273, 1.0104, 1.0044, 1.0026, 1.0016, 1.0006, 1.0004];
selectedLinkRatiosTable = cdfSummary(dT)
```

```
selectedLinkRatiosTable=3×10 table
                                12-24      24-36      36-48      48-60      60-72      72-84
                               _______    _______    _______    _______    _______    _______

    Selected                    1.1755     1.0577     1.0273     1.0104     1.0044     1.0026
    CDF to Ultimate              1.303     1.1084      1.048     1.0201     1.0096     1.0052
    Percent of Total Claims    0.76747    0.90216    0.95422    0.98027    0.99046    0.99482
```

Input Arguments

`developmentTriangle` — Development triangle

`developmentTriangle` object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

Output Arguments

`selectedLinkRatiosTable` — CDF to ultimate claims

table

CDF to ultimate claims, returned as a table. The table shows the selected ratios, CDFs, and percentage of total claims.

More About

Cumulative Development Factors

Calculating the cumulative development factors (CDFs) of a random variable is a method to describe the distribution of random variables.

The CDF of a real-valued random variable X, or just distribution function of X, evaluated at x, is the probability that X takes a value less than or equal to x.

Ultimate Claims

Ultimate claims are the total sum the insured, its insurer(s), and/or its reinsurer(s) pay for a fully developed loss. A fully developed loss is the paid losses plus outstanding and reported losses and incurred-but-not-reported (IBNR) losses.

Version History

Introduced in R2020b

See Also

`view` | `linkRatios` | `linkRatioAverages` | `ultimateClaims` | `fullTriangle` | `linkRatiosPlot` | `claimsPlot`

Topics

"Mean Square Error of Prediction for Estimated Ultimate Claims" on page 4-161
"Bootstrap Using Chain Ladder Method" on page 4-168
"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

claimsPlot

Plot claims for development triangle

Syntax

```
claimsPlot(dT)
claimsPlot(dT,Name,Value)
h = claimsPlot(ax, ___ )
```

Description

`claimsPlot(dT)` plots one line for each origin period for all development periods.

`claimsPlot(dT,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

`h = claimsPlot(ax, ___ )` additionally returns the figure handle `h`. Use this syntax with any of the input arguments in previous syntaxes.

Examples

Generate Line Plot of Cumulative Claims for Each Development Period

Generate a line plot of cumulative claims for each of the development periods using a `developmentTriangle` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

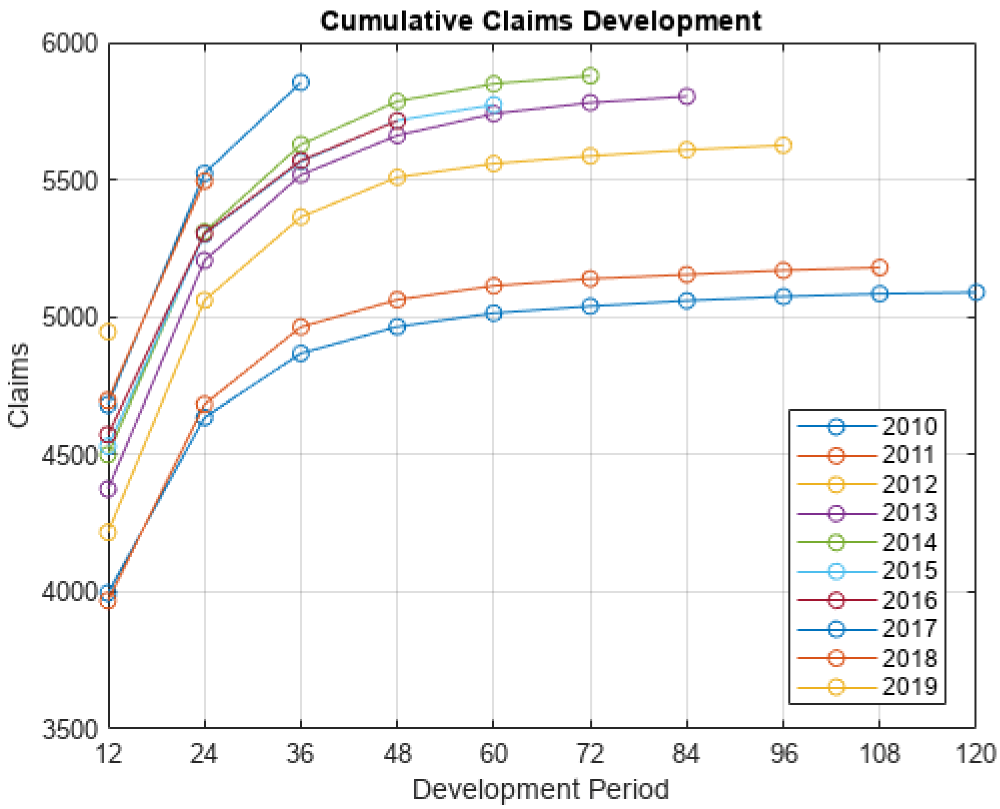
```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
```

```
                 Claims: [10x10 double]
         LatestDiagonal: [10x1 double]
            Description: ""
             TailFactor: 1
CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
      SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

Use the `claimsPlot` function to generate a line plot of cumulative claims.

```
claimsPlot(dT)
```



Input Arguments

dT — Development triangle
`developmentTriangle` object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

ax — Valid axis object
object

(Optional) Valid axis object, specified as an `ax` object created using `axes`. The function creates the plot on the axes specified by the optional `ax` argument instead of on the current axes (`gca`). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `claimsPlot(dT,'Cumulative',false)`

`Cumulative` — Cumulative claims
`true` (default) | logical with value `true` or `false`

Cumulative claims, specified as the comma-separated pair consisting of `'Cumulative'` and a logical value.

Data Types: `logical`

Output Arguments

h — Figure handle
handle object

Figure handle for line objects, returned as a handle object.

Version History

Introduced in R2021a

See Also

`view` | `linkRatios` | `linkRatioAverages` | `cdfSummary` | `ultimateClaims` | `fullTriangle` | `linkRatiosPlot`

Topics

"Mean Square Error of Prediction for Estimated Ultimate Claims" on page 4-161
"Bootstrap Using Chain Ladder Method" on page 4-168
"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

concentrationIndices

Compute ad-hoc concentration indices for a portfolio

Syntax

```
ci = concentrationIndices(PortfolioData)
[ci,Lorenz] = concentrationIndices( ___ ,Name,Value)
```

Description

`ci = concentrationIndices(PortfolioData)` computes multiple ad-hoc concentration indices for a given portfolio. The `concentrationIndices` function supports the following indices:

- CR — Concentration ratio
- Deciles — Deciles of the portfolio weights distribution
- Gini — Gini coefficient
- HH — Herfindahl-Hirschman index
- HK — Hannah-Kay index
- HT — Hall-Tideman index
- TE — Theil entropy index

`[ci,Lorenz] = concentrationIndices( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute Concentration Indices for a Credit Portfolio

Compute the concentration indices for a credit portfolio using a portfolio that is described by its exposures. The exposures at default are stored in the `EAD` array.

Load the `CreditPortfolioData.mat` file that contains `EAD` used for the `PortfolioData` input argument.

```
load CreditPortfolioData.mat
ci = concentrationIndices(EAD)
```

```
ci=1×8 table
         ID              CR          Deciles        Gini         HH           HK           HT          TI
    ___________    ________    ___________    _______    _________    _________    _________    ____

    "Portfolio"    0.058745    1x11 double    0.55751    0.023919     0.013363     0.022599     0.53
```

Compute Multiple Concentration Ratios

Use the CRIndex optional input to obtain the concentration ratios for the tenth and twentieth largest exposures. In the output, the CR column becomes a vector, with one value for each requested index.

Load the CreditPortfolioData.mat file that contains the EAD used for the PortfolioData input argument.

```
load CreditPortfolioData.mat
ci = concentrationIndices(EAD,'CRIndex',[10 20])
```

```
ci=1×8 table
        ID                 CR                Deciles        Gini         HH          HK          HT
    ___________    __________________    ___________    _______    ________    ________    ______

    "Portfolio"    0.38942    0.58836    1x11 double    0.55751    0.023919    0.013363    0.0225
```

Modify the Alpha Parameter of the Hannah-Kay Index

Use the HKAlpha optional input to set the alpha parameter for the Hannah-Kay (HK) index. Use a vector of alpha values to compute the HK index for multiple parameter values. In the output, the HK column becomes a vector, with one value for each requested alpha value.

Load the CreditPortfolioData.mat file that contains EAD used for the PortfolioData input argument.

```
load CreditPortfolioData.mat
ci = concentrationIndices(EAD,'HKAlpha',[0.5 3])
```

```
ci=1×8 table
        ID            CR           Deciles        Gini         HH                 HK
    ___________    ________    ___________    _______    ________    ____________________    ____

    "Portfolio"    0.058745    1x11 double    0.55751    0.023919    0.013363    0.029344    0.02
```

Create an ID to Compare Concentration Index Results

Compare the concentration measures using an ID optional argument for a fully diversified portfolio and a fully concentrated portfolio.

```
ciD = concentrationIndices([1 1 1 1 1],'ID','Fully diversified');
ciC = concentrationIndices([0 0 0 0 5],'ID','Fully concentrated');
disp([ciD;ciC])
```

```
             ID              CR       Deciles       Gini     HH     HK     HT         TE
    ____________________    ___    ___________    ____    ___    ___    ___    ___________

    "Fully diversified"     0.2    1x11 double       0    0.2    0.2    0.2    -2.2204e-16
    "Fully concentrated"      1    1x11 double     0.8      1      1      1         1.6094
```

Apply Scaling to Concentration Indices

Use the `ScaleIndices` optional input to scale the index values of `Gini`, `HH`, `HK`, `HT`, and `TE`. The range of `ScaleIndices` is from `0` through `1`, independent of the number of loans.

```
ciDU = concentrationIndices([1 1 1 1 1],'ID','Diversified, unscaled');
ciDS = concentrationIndices([1 1 1 1 1],'ID','Diversified, scaled','ScaleIndices',true);
ciCU = concentrationIndices([0 0 0 0 5],'ID','Concentrated, unscaled');
ciCS = concentrationIndices([0 0 0 0 5],'ID','Concentrated, scaled','ScaleIndices',true);
disp([ciDU;ciDS;ciCU;ciCS])
```

```
              ID                 CR       Deciles       Gini        HH            HK            HT
    ________________________    ___    ___________    ____    __________    ___________    ______

    "Diversified, unscaled"     0.2    1x11 double      0            0.2            0.2
    "Diversified, scaled"       0.2    1x11 double      0     3.4694e-17    -3.4694e-17    -6.938
    "Concentrated, unscaled"      1    1x11 double    0.8              1              1
    "Concentrated, scaled"        1    1x11 double      1              1              1
```

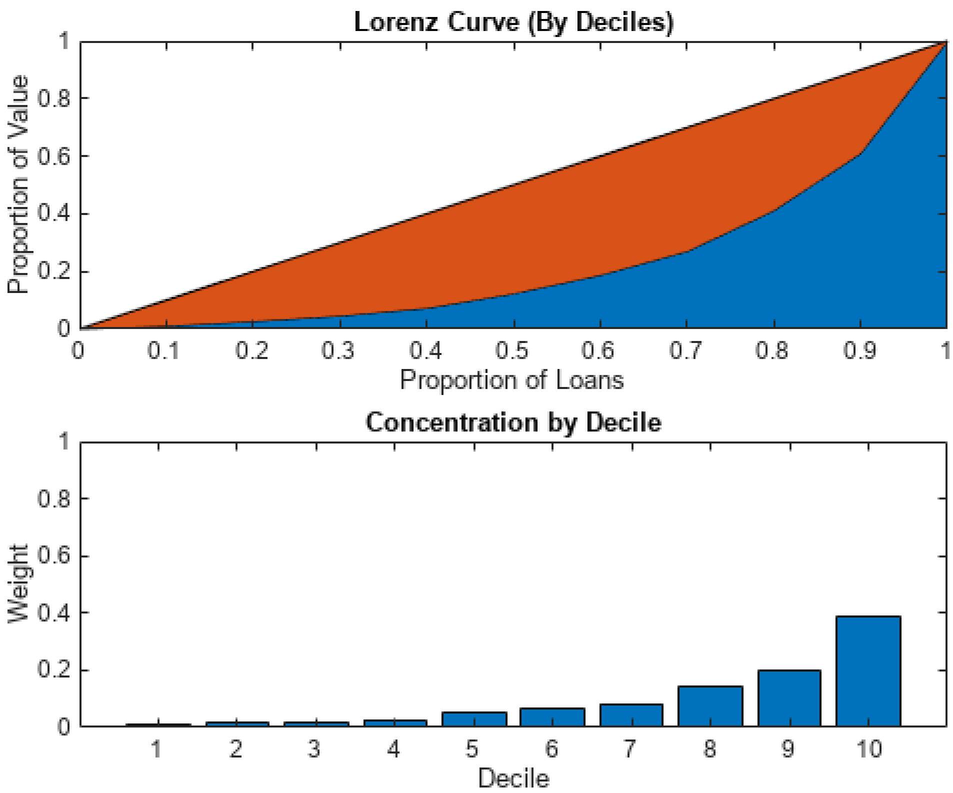
Plot an Approximate Lorenz Curve Using Deciles Information

Load the `CreditPortfolioData.mat` file that contains EAD used for the `PortfolioData` input argument.

```
load CreditPortfolioData.mat
P = EAD;
ci = concentrationIndices(P);
```

Visualize an approximate Lorenz curve using the deciles information and also the concentration at the decile level.

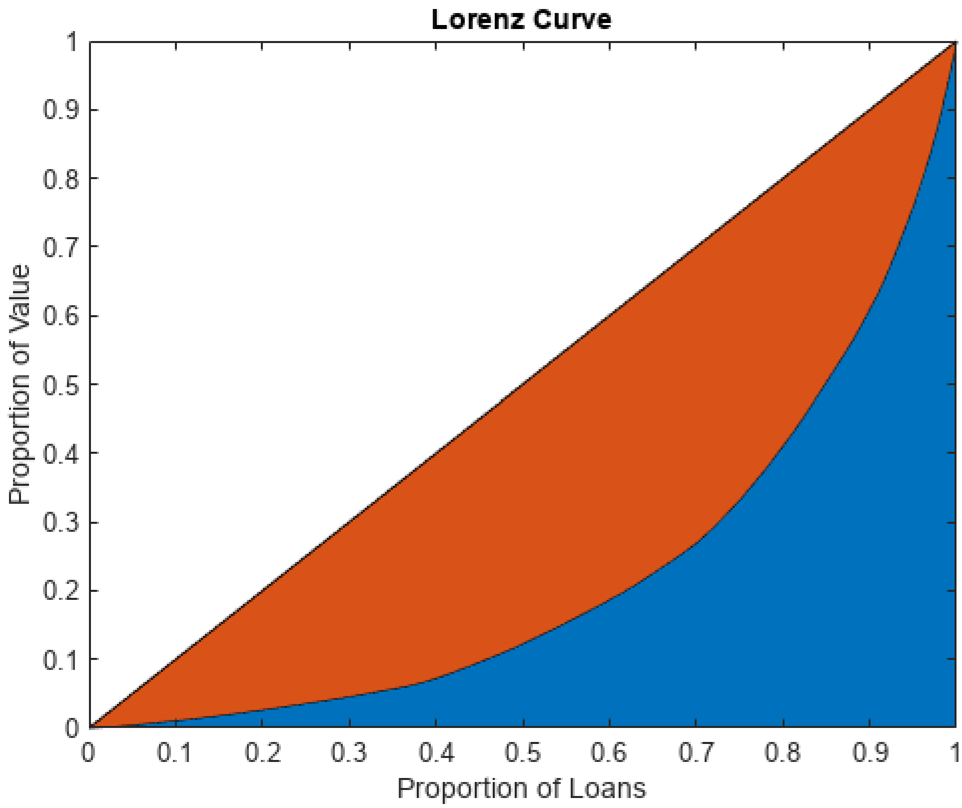
```
Proportion = 0:0.1:1;

figure;
subplot(2,1,1)
area(Proportion',[ci.Deciles' Proportion'-ci.Deciles'])
axis([0 1 0 1])
title('Lorenz Curve (By Deciles)')
xlabel('Proportion of Loans')
ylabel('Proportion of Value')

subplot(2,1,2)
bar(diff(ci.Deciles))
axis([0 11 0 1])
title('Concentration by Decile')
xlabel('Decile')
ylabel('Weight')
```



Plot an Exact Lorenz Curve Using the Optional Lorenz Output

Load the `CreditPortfolioData.mat` file that contains the `EAD` used for the `PortfolioData` input argument. The optional output `Lorenz` contains the data for the exact Lorenz curve.

```
load CreditPortfolioData.mat
P = EAD;
[~,Lorenz] = concentrationIndices(P);

figure;
area(Lorenz.ProportionLoans,[Lorenz.ProportionValue Lorenz.ProportionLoans-Lorenz.ProportionValue
axis([0 1 0 1])
title('Lorenz Curve')
xlabel('Proportion of Loans')
ylabel('Proportion of Value')
```



Input Arguments

`PortfolioData` — Nonnegative portfolio positions in *N* assets
numeric array

Nonnegative portfolio positions in *N* assets, specified as an `N`-by-`1` (or `1`-by-`N`) numeric array.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[ci,Lorenz] = concentrationIndices(PortfolioData,'CRIndex',100)`

`CRIndex` — Index of interest for concentration ratio
`1` (default) | nonnegative integer

Index of interest for the concentration ratio, specified as the comma-separated pair consisting of `'CRIndex'` and an integer value between `1` and *N*, where *N* is the number of assets in the portfolio. The default value for `CRIndex` is `1` (the default `CR` is the largest portfolio weight). If `CRIndex` is a vector, the concentration ratio is computed for the index value in the given order.

Data Types: `double`

`HKAlpha` — Alpha parameter for Hannah-Kay index
`0.5` (default) | nonnegative numeric

Alpha parameter for Hannah-Kay index, specified as the comma-separated pair consisting of `'HKAlpha'`, and a positive number that cannot be equal to `1`. If `HKAlpha` is a vector, the Hannah-Kay index is computed for each alpha value in the given order.

Data Types: `double`

`ID` — User-defined ID for portfolio
`"Portfolio"` (default) | character vector | string object

User-defined ID for the portfolio, specified as the comma-separated pair consisting of `'ID'` and a scalar string object or character vector.

Data Types: `char` | `string`

`ScaleIndices` — Flag to indicate whether to scale concentration indices
`false` (no scaling) (default) | logical

Flag to indicate whether to scale concentration indices, specified as the comma-separated pair consisting of `'ScaleIndices'` and a logical scalar. When the `ScaleIndices` is set to true, the value of the `Gini`, `HH`, `HK`, `HT`, and `TE` indices are scaled so that all these indices have a minimum value of `0` (full diversification) and a maximum value of `1` (full concentration).

Note Scaling is applied only for portfolios with at least two assets. Otherwise, the scaling capability is undefined.

Data Types: `logical`

Output Arguments

`ci` — Concentration indices information for given portfolio
table

Concentration indices information for the given portfolio, returned as a table with the following columns:

- `ID` — Portfolio ID string. Use the `ID` name-value pair argument to set it.
- `CR` — Concentration ratio. By default, the concentration ratio for the first index (largest portfolio weight) is reported. Use the `CRIndex` name-value pair argument to choose a different index. If `CRIndex` is a vector of length *m*, then `CR` is a row vector of size `1`-by-*m*. For more information, see "More About" on page 6-33.
- `Deciles` — Deciles of the portfolio weights distribution is a `1`-by-`11` row vector containing the values `0`, the nine decile cut points, and `1`. For more information, see "More About" on page 6-33.
- `Gini` — Gini coefficient. For more information, see "More About" on page 6-33.
- `HH` — Herfindahl-Hirschman index. For more information, see "More About" on page 6-33.
- `HK` — Hannah-Kay index (reciprocal). By default, the `'alpha'` parameter is set to `0.5`. Use the `HKAlpha` name-value pair argument to choose a different value. If `HKAlpha` is a vector of length*m*, then `HK` is a row vector of size `1`-by-*m*. For more information, see "More About" on page 6-33.

- HT — Hall-Tideman index. For more information, see “More About” on page 6-33.
- TE — Theil entropy index. For more information, see “More About” on page 6-33.

Lorenz — Lorenz curve data
table

Lorenz curve data, returned as a table with the following columns:

- ProportionLoans — (N+1)-by-1 numeric array containing the values 0, 1/N, 2/N, ... N/N = 1. This is the data for the horizontal axis of the Lorenz curve.
- ProportionValue — (N+1)-by-1 numeric array containing the proportion of portfolio value accumulated up to the corresponding proportion of loans in the ProportionLoans column. This is the data for the vertical axis of the Lorenz curve.

More About

Portfolio Notation

All the concentration indices for concentrationIndices assume a credit portfolio with an exposure to counterparties.

Let P be a given credit portfolio with exposure to N counterparties. Let $x_1,\ldots x_N$ represent the exposures to each counterparty, with $x_i >= 0$ for all $i = 1,\ldots N$. And, let x be the total portfolio exposure

$$x = \sum_{i=1}^{N} x_i$$

Assume that $x > 0$, that is, at least one exposure is nonzero. The portfolio weights are given by $w_1,\ldots,w_N$ with

$$w_i = \frac{x_i}{x}$$

The weights are sorted in non-decreasing order. The following standard notation uses brackets around the indices to denote ordered values.

$$w_{[1]} \leq w_{[2]} \leq \ldots \leq w_{[N]}$$

Concentration Ratio

The concentration ratio (CR) answers the question “what proportion of the total exposure is accumulated in the largest k loans?”

The formula for the concentration ratio (CR) is:

$$CR_k = \sum_{i=1}^{k} w_{[N-i+1]}$$

For example, if $k=1$, CR_1 is a sum of the one term $w_{[N-1+1]} = w_{[N]}$, that is, it is the largest weight. For any k, the CR index takes values from 0 through 1.

Lorenz Curve

The Lorenz curve is a visualization of the cumulative proportion of portfolio value (or cumulative portfolio weights) against the cumulative proportion of loans.

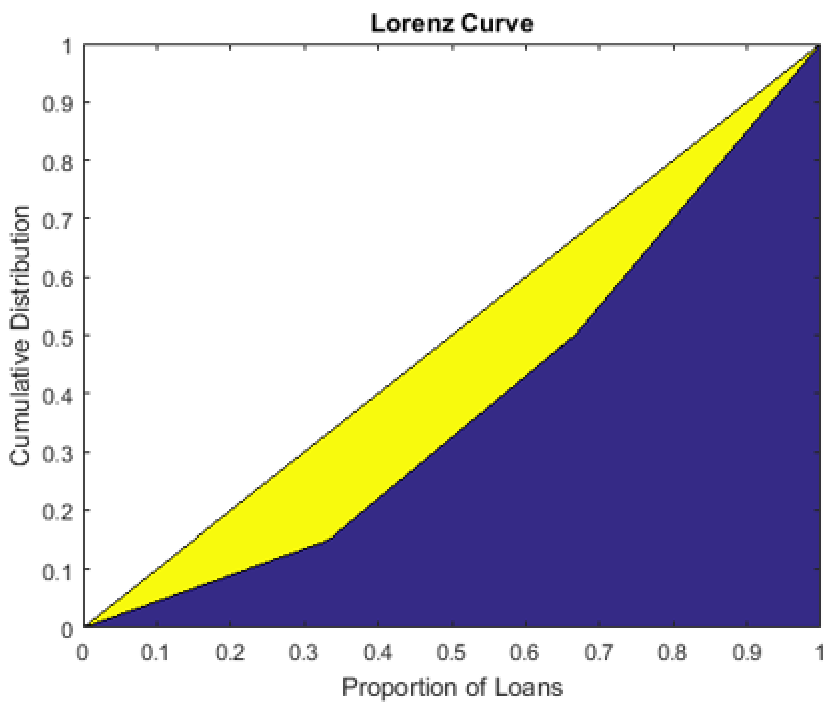
The cumulative proportion of loans (p) is defined by:

$$p_0 = 0, p_1 = \frac{1}{N}, p_2 = \frac{2}{N}, \ldots, p_N = \frac{N}{N} = 1$$

The cumulative proportion of portfolio value L is defined as:

$$L_0 = 0, L_k = \sum_{i=1}^{k} w_{[i]}$$

The Lorenz curve is a plot of L versus p, or the cumulative proportion of portfolio value versus cumulative proportion of the number of loans (sorted from smallest to largest).



The diagonal line is indicated in the same plot because it represents the curve for the portfolio with the least possible concentration (all loans with the same weight). The area between the diagonal and the Lorenz curve is a visual representation of the Gini coefficient, which is another concentration measure.

Deciles

Deciles are commonly used in the context of income inequality.

If you sort individuals by their income level, what proportion of the total income is earned by the lowest 10% and the lowest 20% of the population? In a credit portfolio, loans can be sorted by exposure. The first decile corresponds to the proportion of the portfolio value that is accumulated by the smallest 10% loans, and so on. Deciles are proportions, therefore they always take values from `0` through `1`.

Defining the cumulative proportion of loans (p) and the cumulative proportion of values L as in “Lorenz Curve” on page 6-34, the deciles are a subset of the proportion of value array. Given indices *d1*, *d2*,..., *d9* such that the proportion of loans matches exactly these values:

$$p_{d1} = 0.1, p_{d2} = 0.2, \ldots, p_{d9} = 0.9$$

The deciles $D_0, D_1, \ldots, D_9, D_{10}$ are defined as the corresponding proportion of values:

$$D_0 = L_0 = 0, D_1 = L_{d1}, D_2 = L_{d2}, \ldots, D_9 = L_{d9}, D_{10} = L_N = 1$$

When the total number of loans N is not divisible by 10, no indices match the exact proportion of loans 0.1, 0.2, and so on. In that case, the decile values are linearly interpolated from the Lorenz curve data (that is, from the p and L arrays). With this definition, there are 11 values in the deciles information because the end points 0% and 100% are included.

Gini Index

The Gini index (or coefficient) is visualized on a Lorenz curve plot as the area between the diagonal and the Lorenz curve.

Technically, the Gini index is the ratio of that area to the area of the full triangle under the diagonal on the Lorenz curve plot. The Gini index is also defined equivalently as the average absolute difference between all the weights in the portfolio normalized by the average weight.

Using the proportion of values that array L defined in the Lorenz curve section, the Gini index is given by the formula:

$$Gini = 1 - \frac{1}{N}\sum_{i=1}^{N}(L_{i-1} + L_i)$$

Equivalently, the Gini index can be computed from the sorted weights directly with the formula:

$$Gini = \frac{1}{N}\sum_{i=1}^{N}(2i - 1)w_{[i]} - 1$$

The Gini coefficient values are always between `0` (full diversification) and `1 - 1`/N (full concentration).

Herfindahl-Hirschman Index

The Herfindahl-Hirschman index is commonly used as a measure of market concentration.

The formula for the Herfindahl-Hirschman index is:

$$HH = \sum_{i=1}^{N} w_i^2$$

The Herfindahl-Hirschman index takes values between `1`/N (full diversification) and `1` (full concentration).

Hannah-Kay Index

The Hannah-Kay index is a generalization of the Herfindahl-Hirschman index.

The formula for the Hannah-Kay depends on a parameter α `> 0`, α ≄ `1`, as follows:

$$HK_\alpha = \left(\sum_{i=1}^{N} w_i{}^\alpha\right)^{1/(\alpha-1)}$$

This formula is the reciprocal of the original Hannah-Kay index, which is defined with 1/(1 - α) in the exponent. For concentration analysis, the reciprocal formula is the standard because it increases as the concentration increases. This is the formula implemented in `concentrationIndices`. The Hannah-Kay index takes values between 1/*N* (full diversification) and 1 (full concentration).

Hall-Tideman Index

The Hall-Tideman index is a measure commonly used for market concentration.

The formula for the Hall-Tideman index is:

$$HT = \frac{1}{2\sum_{i=1}^{N}(N-i+1)w_{[i]} - 1}$$

The Hall-Tideman index takes values between 1/*N* (full diversification) and 1 (full concentration).

Theil Entropy Index

The Theil entropy index, based on a traditional entropy measure (for example, Shannon entropy), is adjusted so that it increases as concentration increases (entropy moves in the opposite direction), and shifted to make it positive.

The formula for the Theil entropy index is:

$$TE = \sum_{i=1}^{N} w_i \log(w_i) + \log(N)$$

The Theil entropy index takes values between 0 (full diversification) and `log`(*N*) (full concentration).

Version History

Introduced in R2017a

References

[1] Basel Committee on Banking Supervision. "*Studies on Credit Risk Concentration*". Working paper no. 15. November, 2006.

[2] Calabrese, R., and F. Porro. "Single-name concentration risk in credit portfolios: a comparison of concentration indices." working paper 201214, Geary Institute, University College, Dublin, May, 2012.

[3] Lütkebohmert, E. *Concentration Risk in Credit Portfolios.* Springer, 2009.

See Also

Topics

“Analyze the Sensitivity of Concentration to a Given Exposure” on page 4-49
“Compare Concentration Indices for Random Portfolios” on page 4-51
“Concentration Indices” on page 1-15

conditional

Conditional expected shortfall (ES) backtest by Acerbi and Szekely

Syntax

```
TestResults = conditional(ebts)
[TestResults,SimTestStatistic] = conditional(ebts,Name,Value)
```

Description

`TestResults = conditional(ebts)` runs the conditional ES backtest of Acerbi-Szekely (2014). The conditional test has two underlying tests, a preliminary Value-at-Risk (VaR) backtest that is specified using the name-value pair argument `VaRTest`, and the standalone conditional ES backtest. A `'reject'` result on either underlying test produces a `'reject'` result on the conditional test.

`[TestResults,SimTestStatistic] = conditional(ebts,Name,Value)` adds optional name-value pair arguments for `TestLevel` and `VaRTest`.

Examples

Run an ES Conditional Test

Create an `esbacktestbysim` object.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

Generate the ES conditional test report.

```
TestResults = conditional(ebts)
```

```
TestResults=3×14 table
    PortfolioID       VaRID        VaRLevel    Conditional    ConditionalOnly    PValue     TestS
    ___________    _____________    ________    ___________    _______________    ______    _____

       "S&P"       "t(10) 95%"        0.95        reject           reject            0       -0
       "S&P"       "t(10) 97.5%"     0.975        reject           reject        0.001       -(
       "S&P"       "t(10) 99%"        0.99        reject           reject        0.003       -(
```

Input Arguments

`ebts` — `esbacktestbysim` object
object

`esbacktestbysim` (`ebts`) object, which contains a copy of the given data (the `PortfolioData`, `VarData`, `ESData`, and `Distribution` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktestbysim` object, see `esbacktestbysim`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[TestResults,SimTestStatistic] = conditional(ebts,'TestLevel',0.99)`

`TestLevel` — Test confidence level
`0.95` (default) | numeric value between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric value between `0` and `1`.

Data Types: `double`

`VaRTest` — Indicator for VaR back test
`'pof'` (default) | character vector with a value of `'tl'`, `'bin'`, `'pof'`, `'tuff'`, `'cc'`, `'cci'`, `'tbf'`, or `'tbfi'` | string array with a value of `'tl'`, `'bin'`, `'pof'`, `'tuff'`, `'cc'`, `'cci'`, `'tbf'`, or `'tbfi'`

Indicator for VaR back test, specified as the comma-separated pair consisting of `'VaRTest'` and a character vector or string array with a value of `'tl'`, `'bin'`, `'pof'`, `'tuff'`, `'cc'`, `'cci'`, `'tbf'`, or `'tbfi'`. For more information on these VaR backtests, see `varbacktest`.

Note The specified `VaRTest` is run using the same `TestLevel` value that is specified with the `TestLevel` name-value pair argument in the `conditional` function.

Data Types: `char` | `string`

Output Arguments

`TestResults` — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data.
- `'VaRID'` — VaR ID for each of the VaR data columns provided.

- `'VaRLevel'` — VaR level for the corresponding VaR data column.
- `'Conditional'`— Categorical array with categories 'accept' and 'reject' indicating the result of the conditional test. This result combines the outcome of the `'ConditionalOnly'` column and the VaR test.
- `'ConditionalOnly'`— Categorical array with categories 'accept' and 'reject' indicating the result of the standalone conditional test, independent of the VaR test outcome.
- `'PValue'`— *P*-value of the standalone conditional test (for the`'ConditionalOnly'` column).
- `'TestStatistic'`— Conditional test statistic (for the`'ConditionalOnly'` column).
- `'CriticalValue'`— Critical value for the conditional test.
- `'VaRTest'`— String array indicating the selected VaR test as specified by the `VaRTest` argument.
- `'VaRTestResult'`— Categorical array with categories `'accept'` and `'reject'` indicating the result of the VaR test selected with the `'VaRTest'` argument.
- `'VaRTestPValue'`— P-value for the VaR backtest. If the traffic-light test (`tl`) is used, this is 1 minus the traffic-light test's `'Probability'` column value.
- `'Observations'`— Number of observations.
- `'Scenarios'`— Number of scenarios simulated to get the *p*-values.
- `'TestLevel'`— Test confidence level.

Note For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

`SimTestStatistic` — Simulated values of test statistic
numeric array

Simulated values of the test statistic, returned as a `NumVaRs`-by-`NumScenarios` numeric array.

More About

Conditional Test by Acerbi and Szekely

The conditional test is also known as the first Acerbi-Szekely test.

The conditional test statistic is based on the conditional relationship

$$ES_t = -E_t[X_t | X_t < -VaR_t]$$

where

X_t is the portfolio outcome, that is the portfolio return or portfolio profit and loss for period *t*.

VaR_t is the estimated VaR for period *t*.

ES_t is the estimated expected shortfall for period *t*.

The number of failures is defined as

$$NumFailures = \sum_{t=1}^{N} I_t$$

where

N is the number of periods in the test window ($t = 1,...,N$).

I_t is the VaR failure indicator on period t with a value of 1 if X_t < -VaR, and 0 otherwise.

The conditional test statistic is defined as:

$$Z_{cond} = \frac{1}{NumFailures} \sum_{t=1}^{N} \frac{X_t I_t}{ES_t} + 1$$

The conditional test has two parts. A VaR backtest, specified by the `VaRTest` name-value pair argument, must be run for the number of failures (`NumFailures`), and a standalone conditional test is performed for the conditional test statistic Z_{cond}. The conditional test accepts the model only when both the VaR test and the standalone conditional test accept the model.

Significance of the Test

Under the assumption that the distributional assumptions are correct, the expected value of the test statistic Z_{cond}, assuming at least one VaR failure, is 0.

This is expressed as:

$$E[Z_{cond} | NumFailures > 0] = 0$$

Negative values of the test statistic indicate risk underestimation. The conditional test is a one-sided test that rejects when there is evidence that the model underestimates risk (for technical details on the null and alternative hypotheses, see Acerbi-Szekely, 2014). The conditional test rejects the model when the *p*-value is less than 1 minus the test confidence level.

For more information on the steps to simulate the test statistics and the details for the computation of the *p*-values and critical values, see `simulate`.

Edge Cases

The conditional test statistic is undefined (`NaN`) when there are no VaR failures in the data (`NumFailures = 0`).

The *p*-value is set to `NaN` in these cases, and test result is to `'accept'`, because there is no evidence of risk underestimation.

Likewise, the simulated conditional test statistic is undefined (`NaN`) for scenarios with no VaR failures. These scenarios are discarded for the estimation of the significance of the test. Under the assumption that the distributional assumptions are correct, $E[Z_{cond} | NumFailures > 0] = 0$, so the significance is computed over scenarios with at least one failure (`NumFailures > 0`). The number of scenarios reported by the `conditional` test function is the number of scenarios with at least one VaR failure. The number of scenarios reported can be smaller than the total number of scenarios simulated. The critical value is estimated over the scenarios with at least one VaR failure. If the simulated test statistic is `NaN` for all scenarios, the critical value is set to `NaN`. Scenarios with no failures are more likely as the expected number of failures Np_{VaR} gets smaller.

Version History

Introduced in R2017b

References

[1] Acerbi, C. and Szekely, B. *Backtesting Expected Shortfall.* MSCI Inc. December, 2014.

See Also

`summary` | `runtests` | `unconditional` | `quantile` | `simulate` | `minBiasRelative` | `minBiasAbsolute` | `esbacktestbysim` | `tl` | `bin` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `esbacktestbyde`

Topics

"Expected Shortfall (ES) Backtesting Workflow Using Simulation" on page 2-34
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

conditionalDE

Conditional Du-Escanciano (DE) expected shortfall (ES) backtest

Syntax

```
TestResults = conditionalDE(ebtde)
[TestResults,SimTestStatistic] = conditionalDE( ___ ,Name,Value)
```

Description

`TestResults = conditionalDE(ebtde)` runs the conditional expected shortfall (ES) backtest by Du and Escanciano [1]. The conditional test supports critical values by large-scale approximation and by finite-sample simulation.

`[TestResults,SimTestStatistic] = conditionalDE( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input argument in the previous syntax.

Examples

Create an esbacktestbyde Object and Run a ConditionalDE Test

Create an `esbacktestbyde` object for a *t* model with 10 degrees of freedom and 2 lags, and then run a `conditionalDE` test.

```
load ESBacktestDistributionData.mat
    rng('default'); % For reproducibility
    ebtde = esbacktestbyde(Returns,"t",...
       'DegreesOfFreedom',T10DoF,...
       'Location',T10Location,...
       'Scale',T10Scale,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
    conditionalDE(ebtde,'NumLags',2)
```

```
ans=3×13 table
    PortfolioID        VaRID         VaRLevel     ConditionalDE       PValue      TestStatistic     (
    ___________    _____________     ________     _____________     __________    _____________     _

       "S&P"       "t(10) 95%"         0.95          reject         3.2121e-09       39.113
       "S&P"       "t(10) 97.5%"      0.975          reject         1.6979e-07       31.177
       "S&P"       "t(10) 99%"         0.99          reject         9.1526e-05       18.598
```

Input Arguments

ebtde — esbacktestbyde object
object

`esbacktestbyde` object, which contains a copy of the data (the `PortfolioData`, `VarData`, and `ESData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktestbyde` object, see `esbacktestbyde`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = conditionalDE(ebtde,'CriticalValueMethod','simulation','NumLags',10,'TestLevel',0.99)`

`CriticalValueMethod` — Method to compute critical values, confidence intervals, and *p*-values
`'large-sample'` (default) | character vector with values of `'large-sample'` or `'simulation'` | string with values of `"large-sample"` or `"simulation"`

Method to compute critical values, confidence intervals, and *p*-values, specified as the comma-separated pair consisting of `'CriticalValueMethod'` and a character vector or string with a value of `'large-sample'` or `'simulation'`.

Data Types: `char` | `string`

`NumLags` — Number of lags in `conditionalDE` test
`1` (default) | positive integer

Number of lags in the `conditionalDE` test, specified as the comma-separated pair consisting of `'NumLags'` and a positive integer.

Data Types: `double`

`TestLevel` — Test confidence level
`0.95` (default) | numeric value between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric value between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR levels
- `'VaRLevel'` — VaR level

- `'ConditionalDE'`— Categorical array with the categories `'accept'` and `'reject'`, which indicate the result of the conditional DE test
- `'PValue'`— *P*-value of the conditional DE test
- `'TestStatistic'`— Conditional DE test statistic
- `'CriticalValue'`— Critical value for the conditional DE test
- `'AutoCorrelation'`— Autocorrelation for the reported number of lags
- `'Observations'`— Number of observations
- `'CriticalValueMethod'`— Method used to compute confidence intervals and *p*-values
- `'NumLags'`— Number of lags
- `'Scenarios'`— Number of scenarios simulated to get the *p*-values
- `'TestLevel'`— Test confidence level

Note If you specify `CriticalValueMethod` as `'large-sample'`, the function reports the number of `'Scenarios'` as `NaN`.

For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

`SimTestStatistic` — Simulated values of the test statistics
numeric array

Simulated values of the test statistics, returned as an `NumVaRs`-by-`NumScenarios` numeric array.

More About

Conditional DE Test

The conditional DE test is a one-sided test to check if the test statistic is much larger than zero.

The test statistic for the conditional DE test is derived in several steps. First, define the autocovariance for lag *j*:

$$\gamma_j = \frac{1}{N - j}\sum_{t = j + 1}^{N} (H_t - \alpha/2)(H_{t - j} - \alpha/2)$$

where

- α = 1- *VaRLevel*.
- H_t is the cumulative failures or violations process: $H_t = (\alpha - U_t)I(U_t < \alpha) / \alpha$, where $I(x)$ is the indicator function.
- U_t are the ranks or mapped returns $U_t = P_t(X_t)$, where $P_t(X_t) = P(X_t \mid \theta_t)$ is the cumulative distribution of the portfolio outcomes or returns X_t over a given test window $t = 1,\ldots N$ and θ_t are the parameters of the distribution. For simplicity, the subindex *t* is both the return and the parameters, understanding that the parameters are those used on date *t*, even though those parameters are estimated on the previous date *t*-1, or even prior to that.

The exact theoretical mean α/2, as opposed to the sample mean, is used in the autocovariance formula, as suggested in the paper by Du and Escanciano [1].

The autocorrelation for lag j is then

$$\rho_j = \frac{\gamma_j}{\gamma_0}$$

The test statistic for m lags is

$$C_{ES}(m) = N\sum_{j=1}^{m} \rho_j^2$$

Significance of the Test

The test statistic C_{ES} is a random variable and a function of random return sequences or portfolio outcomes $X_1,\ldots,X_N$:

$$C_{ES} = C_{ES}(X_1, \ldots, X_N) .$$

For returns observed in the test window 1,...,N, the test statistic attains a fixed value:

$$C_{ES}^{obs} = C_{ES}(X_{obs1}, \ldots, X_{obsN}) .$$

In general, for unknown returns that follow a distribution of P_t, the value of C_{ES} is uncertain and it follows a cumulative distribution function:

$$P_C(x) = P[C_{ES} \le x] .$$

This distribution function computes a confidence interval and a p-value. To determine the distribution P_C, the `esbacktestbyde` class supports the large-sample approximation and simulation methods. You can specify one of these methods by using the optional name-value pair argument `CriticalValueMethod`.

For the large sample approximation method, the distribution P_C is derived from an asymptotic analysis. If the number of observations N is large, the test statistic is approximately distributed as a chi-square distribution with m degrees of freedom:

$$C_{ES}(m) \xrightarrow[dist]{} \chi_m^2 = P_C$$

Note that the limiting distribution is independent of α.

If α_{test} = 1 - *test confidence level*, then the critical value CV is the value that satisfies the equation

$$1 - P_C(CV) = \alpha_{test} .$$

The p-value is determined as

$$P_{value} 1 - P_C(C_{ES}^{obs}) .$$

The test rejects if $p_{value} < \alpha_{test}$.

For the simulation method, the distribution $P_C$is estimated as follows

1 Simulate M scenarios of returns as

$$X^s = (X_1^s, \ldots, X_N^s),\ s = 1, \ldots, M .$$

2 Compute the corresponding test statistic as

$$C_{ES}^{s} = C_{ES}(X_1^s, \ldots, X_N^s),\ s = 1, \ldots, M\,.$$

3 Define P_C as the empirical distribution of the simulated test statistic values as

$$P_C = P[C_{ES} \leq x] = \frac{1}{M} I(C_{ES}^{s} \leq x),$$

where $I(.)$ is the indicator function.

In practice, simulating ranks is more efficient than simulating returns and then transforming the returns into ranks. `simulate`.

For the empirical distribution, the value of 1-$P_C(x)$ may be different than $P[C_{ES} \geq x]$ because the distribution may have nontrivial jumps (simulated tied values). Use the latter probability for the estimation of confidence levels and *p*-values.

If α_{test} = 1 - *test confidence level*, then the critical value of levels *CV* is the value that satisfies the equation

$$P[C_{ES} \geq CV] = \alpha_{test}\,.$$

The reported critical value *CV* is one of the simulated test statistic values C^{s}_{ES} that approximately solves the preceding equation.

The *p*-value is determined as

$$p_{value} = P\left[C_{ES} \geq C_{ES}^{obs}\right].$$

The test rejects if $p_{value} < \alpha_{test}$.

Version History

Introduced in R2019b

References

[1] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science.* Vol. 63, Issue 4, April 2017.

[2] Basel Committee on Banking Supervision. "*Minimum Capital Requirements for Market Risk*". January 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

`esbacktestbyde` | `summary` | `runtests` | `unconditionalDE` | `simulate` | `esbacktestbysim`

Topics

“Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63
“Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-72
“Expected Shortfall Estimation and Backtesting” on page 2-44
“Overview of Expected Shortfall Backtesting” on page 2-20

"ES Backtest Using Du-Escanciano Method" on page 2-24
"Comparison of ES Backtesting Methods" on page 2-26

confidenceBands

Confidence interval bands

Syntax

```
cbTable = confidenceBands(cdc)
cbTable = confidenceBands(cdc,Name,Value)
```

Description

`cbTable = confidenceBands(cdc)` returns a table of the requested risk measure and its associated confidence bands. `confidenceBands` is used to investigate how the values of a risk measure and its associated confidence interval converge as the number of scenarios increases. The `simulate` function must be run before `confidenceBands` is used. For more information on using a `creditDefaultCopula` object, see `creditDefaultCopula`.

`cbTable = confidenceBands(cdc,Name,Value)` adds optional name-value pair arguments.

Examples

Generate a Table of the Associated Confidence Bands for a Requested Risk Measure for a `creditDefaultCopula` Object

Load saved portfolio data.

```
load CreditPortfolioData.mat;
```

Create a `creditDefaultCopula` object with a two-factor model.

```
cdc = creditDefaultCopula(EAD,PD,LGD,Weights2F,'FactorCorrelation',FactorCorr2F)
```

```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioLosses: []
```

Set the `VaRLevel` to 99%.

```
cdc.VaRLevel = 0.99;
```

Use the `simulate` function before running `confidenceBands`. Use `confidenceBands` with the `creditDefaultCopula` object to generate the `cbTable`.

```
cdc = simulate(cdc,1e5);
cbTable = confidenceBands(cdc,'RiskMeasure','Std','ConfidenceIntervalLevel',0.9);
cbTable(1:10,:)
```

```
ans=10×4 table
    NumScenarios    Lower      Std      Upper 
    ____________    ______    ______    ______

        1000         23.38    24.237    25.166
        2000        23.255    23.859    24.497
        3000        23.617    24.117    24.642
        4000         23.44    23.871    24.319
        5000        23.504    23.891    24.291
        6000        23.582    23.935    24.301
        7000        23.756    24.086    24.426
        8000        23.587    23.893    24.208
        9000        23.582    23.871    24.167
       10000        23.525    23.799    24.079
```

Input Arguments

`cdc` — `creditDefaultCopula` object
object

`creditDefaultCopula` object obtained after running the `simulate` function.

For more information on `creditDefaultCopula` objects, see `creditDefaultCopula`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cbTable = confidenceBands(cdc,'RiskMeasure','Std','ConfidenceIntervalLevel',0.9,'NumPoints',50)`

`RiskMeasure` — Risk measure to investigate
`'CVaR'` (default) | character vector or string with values `'EL'`, `'Std'`, `'VaR'`, or `'CVaR'`

Risk measure to investigate, specified as the comma-separated pair consisting of `'RiskMeasure'` and a character vector or string. Possible values are:

- `'EL'` — Expected loss, the mean of portfolio losses
- `'Std'` — Standard deviation of the losses
- `'VaR'` — Value at risk at the threshold specified by the `VaRLevel` property of the `creditDefaultCopula` object
- `'CVaR'` — Conditional VaR at the threshold specified by the `VaRLevel` property of the `creditDefaultCopula` object

Data Types: `char` | `string`

`ConfidenceIntervalLevel` — Confidence interval level
`0.95` (default) | numeric between `0` and `1`

Confidence interval level, specified as the comma-separated pair consisting of 'ConfidenceIntervalLevel' and a numeric between 0 and 1. For example, if you specify 0.95, a 95% confidence interval is reported in the output table (cbTable).

Data Types: double

NumPoints — Number of scenario samples to report
100 (default) | nonnegative integer

Number of scenario samples to report, specified as the comma-separated pair consisting of 'NumPoints' and a nonnegative integer. The default is 100, meaning confidence bands are reported at 100 evenly spaced points of increasing sample size ranging from 0 to the total number of simulated scenarios.

Note NumPoints must be a numeric scalar greater than 1, and is typically much smaller than total number of scenarios simulated. confidenceBands can be used to obtain a qualitative idea of how fast a risk measure and its confidence interval are converging. Specifying a large value for NumPoints is not recommended and could cause performance issues with confidenceBands.

Data Types: double

Output Arguments

cbTable — Requested risk measure and associated confidence bands
table

Requested risk measure and associated confidence bands at each of the NumPoints scenario sample sizes, returned as a table containing the following columns:

- NumScenarios — Number of scenarios at the sample point
- Lower — Lower confidence band
- *RiskMeasure* — Requested risk measure where the column takes its name from whatever risk measure is requested with the optional input RiskMeasure
- Upper — Upper confidence band

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`creditDefaultCopula` | `table` | `simulate` | `portfolioRisk` | `riskContribution` | `getScenarios`

Topics

“Credit Simulation Using Copulas” on page 4-2
“creditDefaultCopula Simulation Workflow” on page 4-5
“Modeling Correlated Defaults with Copulas” on page 4-18
“One-Factor Model Calibration” on page 4-64
“Corporate Credit Risk” on page 1-3
“Credit Simulation Using Copulas” on page 4-2

getScenarios

Counterparty scenarios

Syntax

`scenarios = getScenarios(cdc,scenarioIndices)`

Description

`scenarios = getScenarios(cdc,scenarioIndices)` returns counterparty scenario details as a matrix of individual losses for each counterparty for the scenarios requested in `scenarioIndices`.

The `simulate` function must be run before `getScenarios` is used. For more information on using a `creditDefaultCopula` object, see `creditDefaultCopula`.

Examples

Compute Individual Losses for Each Counterparty

Load saved portfolio data.

```
load CreditPortfolioData.mat;
```

Create a `creditDefaultCopula` object with a two-factor model.

```
cdc = creditDefaultCopula(EAD,PD,LGD,Weights2F,'FactorCorrelation',FactorCorr2F)
```

```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioLosses: []
```

Set the `VaRLevel` to 99%.

```
cdc.VaRLevel = 0.99;
```

Use the `simulate` function before running `getScenarios`. Use the `getSenarios` function with the `creditDefaultCopula` object to generate the `scenarios` matrix.

```
cdc = simulate(cdc,1e5);
scenarios = getScenarios(cdc,[2,3]);
% expected loss for each scenario
mean(scenarios)
```

```
ans = 1×2
```

```
    0.0369    0.0329
```

Input Arguments

`cdc` — `creditDefaultCopula` object
object

`creditDefaultCopula` object obtained after running the `simulate` function.

For more information on `creditDefaultCopula` objects, see `creditDefaultCopula`.

`scenarioIndices` — Specifies which scenarios are returned
vector

Specifies which scenarios are returned, entered as a vector.

Output Arguments

`scenarios` — Counterparty losses
matrix

Counterparty losses, returned as `NumCounterparties`-by-`N` matrix where `N` is the number of elements in `scenarioIndices`.

Note If the number of scenarios requested is large, then the output matrix, `scenarios`, could be large and potentially limited by the available machine memory.

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also
`simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `creditDefaultCopula`

Topics
"Credit Simulation Using Copulas" on page 4-2
"Modeling Correlated Defaults with Copulas" on page 4-18
"One-Factor Model Calibration" on page 4-64
"Corporate Credit Risk" on page 1-3
"Credit Simulation Using Copulas" on page 4-2

portfolioRisk

Generate portfolio-level risk measurements

Syntax

```
[riskMeasures,confidenceIntervals] = portfolioRisk(cdc)
[riskMeasures,confidenceIntervals] = portfolioRisk(cdc,Name,Value)
```

Description

`[riskMeasures,confidenceIntervals] = portfolioRisk(cdc)` returns tables of risk measurements for the portfolio losses. The `simulate` function must be run before `portfolioRisk` is used. For more information on using a `creditDefaultCopula` object, see `creditDefaultCopula`.

`[riskMeasures,confidenceIntervals] = portfolioRisk(cdc,Name,Value)` adds an optional name-value pair argument for `ConfidenceIntervalLevel`. The `simulate` function must be run before `portfolioRisk` is used.

Examples

Generate Tables for Risk Measure and Confidence Intervals for a `creditDefaultCopula` Object

Load saved portfolio data.

```
load CreditPortfolioData.mat;
```

Create a `creditDefaultCopula` object with a two-factor model.

```
cdc = creditDefaultCopula(EAD,PD,LGD,Weights2F,'FactorCorrelation',FactorCorr2F)
```

```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioLosses: []
```

Set the `VaRLevel` to 99%.

```
cdc.VaRLevel = 0.99;
```

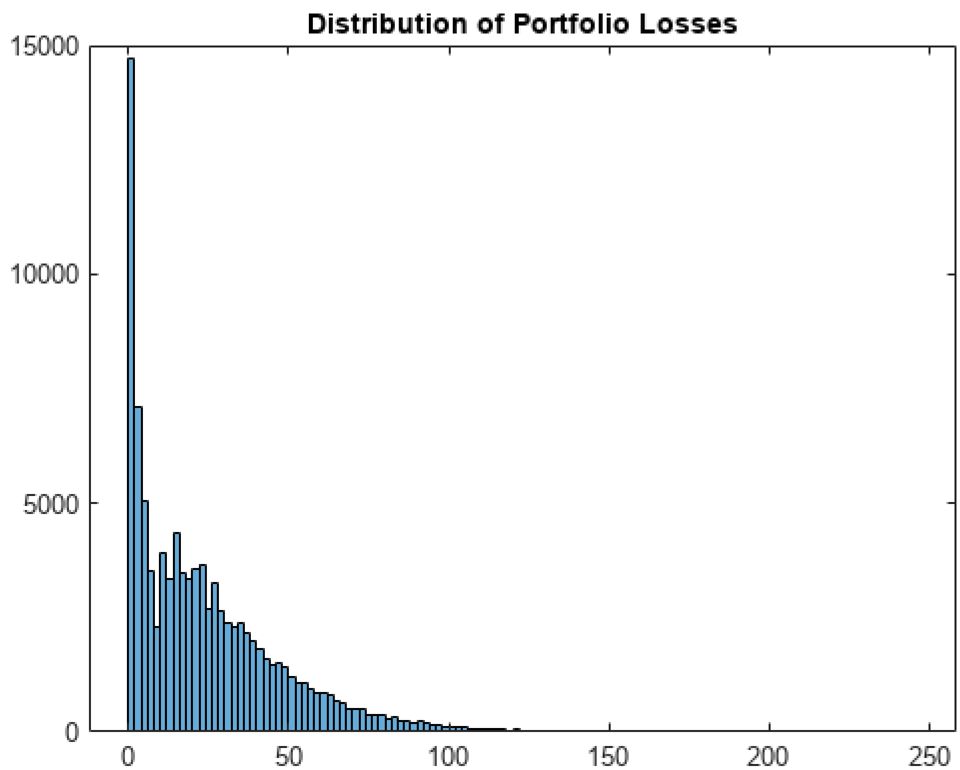
Use the `simulate` function before running `portfolioRisk`. Then use `portfolioRisk` with the `creditDefaultCopula` object to generate the `riskMeasure` and `ConfidenceIntervals` tables.

```
cdc = simulate(cdc,1e5);
[riskMeasure,confidenceIntervals] = portfolioRisk(cdc,'ConfidenceIntervalLevel',0.9)
```

```
riskMeasure=1×4 table
      EL        Std       VaR      CVaR  
    ______    ______    _____    ______

    24.876    23.778    102.4    121.28

confidenceIntervals=1×4 table
            EL                 Std                 VaR                CVaR      
    ________________    ________________    ________________    ________________

    24.752        25    23.691    23.866    101.35    103.35    120.32    122.24
```

View a histogram of the portfolio losses.

```
histogram(cdc.PortfolioLosses);
title('Distribution of Portfolio Losses');
```



For further analysis, use the `simulate`, `portfolioRisk`, `riskContribution`, `confidenceBands`, and `getScenarios` functions with the `creditDefaultCopula` object.

Input Arguments

cdc — `creditDefaultCopula` object
object

`creditDefaultCopula` object obtained after running the `simulate` function.

For more information on `creditDefaultCopula` objects, see `creditDefaultCopula`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[riskMeasure,confidenceIntervals] = portfolioRisk(cdc,'ConfidenceIntervalLevel',0.9)`

`ConfidenceIntervalLevel` — Confidence interval level
`0.95` (default) | numeric between `0` and `1`

Confidence interval level, specified as the comma-separated pair consisting of `'ConfidenceIntervalLevel'` and a numeric between `0` and `1`. For example, if you specify `0.95`, a 95% confidence interval is reported in the output table (`riskMeasures`).

Data Types: `double`

Output Arguments

`riskMeasures` — Risk measures
table

Risk measures, returned as a table containing the following columns:

- `EL` — Expected loss, the mean of portfolio losses
- `Std` — Standard deviation of the losses
- `VaR` — Value at risk at the threshold specified by the `VaRLevel` property of the `creditDefaultCopula` object
- `CVaR` — Conditional VaR at the threshold specified by the `VaRLevel` property of the `creditDefaultCopula` object

`confidenceIntervals` — Confidence intervals
table

Confidence intervals, returned as a table of confidence intervals corresponding to the portfolio risk measures reported in the `riskMeasures` table. Confidence intervals are reported at the level specified by the `ConfidenceIntervalLevel` parameter.

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. “A Comparative Analysis of Current Credit Risk Models.” *Journal of Banking and Finance*. Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. “A Comparative Anatomy of Credit Risk Models.” *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *“CreditMetrics – Technical Document.”* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`table` | `creditDefaultCopula` | `simulate` | `riskContribution` | `confidenceBands` | `getScenarios`

Topics

“Credit Simulation Using Copulas” on page 4-2
“creditDefaultCopula Simulation Workflow” on page 4-5
“Modeling Correlated Defaults with Copulas” on page 4-18
“One-Factor Model Calibration” on page 4-64
“Corporate Credit Risk” on page 1-3
“Credit Simulation Using Copulas” on page 4-2

riskContribution

Generate risk contributions for each counterparty in portfolio

Syntax

```
Contributions = riskContribution(cdc)
Contributions = riskContribution(cdc,Name,Value)
```

Description

`Contributions = riskContribution(cdc)` returns a table of risk contributions for each counterparty in the portfolio. The risk `Contributions` table allocates the full portfolio risk measures to each counterparty, such that the counterparty risk contributions sum to the portfolio risks reported by `portfolioRisk`.

Note When creating a `creditDefaultCopula` object, you can set the `'UseParallel'` property if you have Parallel Computing Toolbox™. Once the `'UseParallel'` property is set, parallel processing is used to compute `riskContribution`.

The `simulate` function must be run before `riskContribution` is used. For more information on using a `creditDefaultCopula` object, see `creditDefaultCopula`.

`Contributions = riskContribution(cdc,Name,Value)` adds an optional name-value pair argument for `VaRWindow`.

Examples

Determine the Risk Contribution for Each Counterparty for a `creditDefaultCopula` Object

Load saved portfolio data.

```
load CreditPortfolioData.mat;
```

Create a `creditDefaultCopula` object with a two-factor model.

```
cdc = creditDefaultCopula(EAD,PD,LGD,Weights2F,'FactorCorrelation',FactorCorr2F)
```

```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioLosses: []
```

Set the `VaRLevel` to 99%.

```
cdc.VaRLevel = 0.99;
```

Use the `simulate` function before running `riskContribution`. Then use `riskContribution` with the `creditDefaultCopula` object to generate the risk `Contributions` table.

```
cdc = simulate(cdc,1e5);
Contributions = riskContribution(cdc);
Contributions(1:10,:)
```

```
ans=10×5 table
    ID        EL            Std           VaR          CVaR
    __    __________    __________    _________    __________

     1      0.036031      0.022762     0.083828       0.13625
     2      0.068357      0.039295      0.23373       0.24984
     3        1.2228       0.60699       2.3184        2.3775
     4      0.002877    0.00079014    0.0024248     0.0013137
     5       0.12127      0.037144      0.18474       0.24622
     6       0.12638      0.078506      0.39779       0.48334
     7       0.84284        0.3541       1.6221        1.8183
     8    0.00090088    0.00011379    0.0016463    0.00089197
     9       0.93117       0.87638       3.3868        3.9936
    10       0.26054       0.37918       1.7399        2.3042
```

Note: Due to simulation noise or numerical error, the `VaR` contribution can sometimes be greater than the `CVaR` contribution.

Input Arguments

`cdc` — `creditDefaultCopula` object
object

`creditDefaultCopula` object obtained after running the `simulate` function.

For more information on `creditDefaultCopula` objects, see `creditDefaultCopula`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Contributions = riskContribution(cdc,'VaRWindow',0.3)`

`VaRWindow` — Size of the window used to compute VaR contributions
`0.05` (default) | numeric between `0` and `1`

Size of the window used to compute VaR contributions, specified as the comma-separated pair consisting of `'VaRWindow'` and a scalar numeric with a percent value. Scenarios in the VaR scenario set are used to calculate the individual counterparty VaR contributions.

The default is `0.05`, meaning that all scenarios with portfolio losses within 5 percent of the VaR are included when computing counterparty VaR contributions.

Data Types: `double`

Output Arguments

`Contributions` — Risk contributions

table

Risk contributions, returned as a table containing the following risk contributions for each counterparty:

- `EL` — Expected loss for the particular counterparty over the scenarios
- `Std` — Standard deviation of loss for the particular counterparty over the scenarios
- `VaR` — Value at risk for the particular counterparty over the scenarios
- `CVaR` — Conditional value at risk for the particular counterparty over the scenarios

The risk `Contributions` table allocates the full portfolio risk measures to each counterparty, such that the counterparty risk contributions sum to the portfolio risks reported by `portfolioRisk`.

More About

Risk Contributions

The `riskContribution` function reports the individual counterparty contributions to the total portfolio risk measures using four risk measures: expected loss (EL), standard deviation (Std), VaR, and CVaR.

- `EL` is the expected loss for each counterparty and is the mean of the counterparty's losses across all scenarios.
- `Std` is the standard deviation for counterparty *i*:

  $$StdCont_i = Std_i \frac{\sum_j Std_j \rho_{ij}}{Std_{\rho}}$$

  where

  Std_{i} is the standard deviation of losses from counterparty *i*.

  $Std_{ï}$ is the standard deviation of portfolio losses.

  ρ_{ij} is the correlation of the losses between counterparties *i* and *j*.
- `VaR` contribution is the mean of a counterparty's losses across all scenarios in which the total portfolio loss is within some small neighborhood around the Portfolio VaR. The default of the `'VaRWindow'` parameter is `0.05` meaning that all scenarios in which the total portfolio loss is within 5% of the portfolio VaR are included in VaR neighborhood.
- `CVaR` is the mean of the counterparty's losses in the set of scenarios in which the total portfolio losses exceed the portfolio VaR.

Version History

Introduced in R2017a

References

[1] Glasserman, P. "Measuring Marginal Risk Contributions in Credit Portfolios." *Journal of Computational Finance*. Vol. 9, No. 2, Winter 2005/2006.

[2] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

See Also

`table` | `creditDefaultCopula` | `simulate` | `portfolioRisk` | `confidenceBands` | `getScenarios`

Topics

"Credit Simulation Using Copulas" on page 4-2
"creditDefaultCopula Simulation Workflow" on page 4-5
"Modeling Correlated Defaults with Copulas" on page 4-18
"One-Factor Model Calibration" on page 4-64
"Corporate Credit Risk" on page 1-3
"Credit Simulation Using Copulas" on page 4-2

External Websites

Parallel Computing with MATLAB (53 min 27 sec)

simulate

Simulate credit defaults using a `creditDefaultCopula` object

Syntax

```
cdc = simulate(cdc,NumScenarios)
cdc = simulate( ___ ,Name,Value)
```

Description

`cdc = simulate(cdc,NumScenarios)` performs the full simulation of credit scenarios and computes defaults and losses for the portfolio defined in the `creditDefaultCopula` object. For more information on using a `creditDefaultCopula` object, see `creditDefaultCopula`.

Note When creating a `creditDefaultCopula` object, you can set the `'UseParallel'` property if you have Parallel Computing Toolbox. Once the `'UseParallel'` property is set, parallel processing is used to compute `simulate`.

`cdc = simulate( ___ ,Name,Value)` adds optional name-value pair arguments for (`Copula`, `DegreesOfFreedom`, and `BlockSize`).

Examples

Run a Simulation Using a `creditDefaultCopula` Object

Load saved portfolio data.

```
load CreditPortfolioData.mat;
```

Create a `creditDefaultCopula` object with a two-factor model.

```
cdc = creditDefaultCopula(EAD,PD,LGD,Weights2F,'FactorCorrelation',FactorCorr2F)
```

```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioLosses: []
```

Set the `VaRLevel` to 99%.

```
cdc.VaRLevel = 0.99;
```

Use the `simulate` function with the `creditDefaultCopula` object. After using `simulate`, you can then use the `portfolioRisk`, `riskContribution`, `confidenceBands`, and `getScenarios` functions with the updated `creditDefaultCopula` object.

```
cdc = simulate(cdc,1e5)
```

```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9900
          UseParallel: 0
       PortfolioLosses: [30.1008 3.6910 3.2895 19.2151 7.5761 44.5088 19.5419 1.7909 72.1443 12.69
```

You can use `riskContribution` with the `creditDefaultCopula` object to generate the risk `Contributions` table.

```
Contributions = riskContribution(cdc);
Contributions(1:10,:)
```

ans=*10×5 table*

ID	EL	Std	VaR	CVaR
1	0.036031	0.022762	0.083828	0.13625
2	0.068357	0.039295	0.23373	0.24984
3	1.2228	0.60699	2.3184	2.3775
4	0.002877	0.00079014	0.0024248	0.0013137
5	0.12127	0.037144	0.18474	0.24622
6	0.12638	0.078506	0.39779	0.48334
7	0.84284	0.3541	1.6221	1.8183
8	0.00090088	0.00011379	0.0016463	0.00089197
9	0.93117	0.87638	3.3868	3.9936
10	0.26054	0.37918	1.7399	2.3042

Input Arguments

`cdc` — `creditDefaultCopula` object

object

`creditDefaultCopula` object, obtained from `creditDefaultCopula`.

For more information on a `creditDefaultCopula` object, see `creditDefaultCopula`.

`NumScenarios` — Number of scenarios to simulate

nonnegative integer

Number of scenarios to simulate, specified as a nonnegative integer. Scenarios are processed in blocks to conserve machine resources.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cdc = simulate(cdc,NumScenarios,'Copula','t','DegreesOfFreedom',5)`

`Copula` — Type of copula
`'Gaussian'` (default) | character vector or string with values `'Gaussian'` or `'t'`

Type of copula, specified as the comma-separated pair consisting of `'Copula'` and a character vector or string. Possible values are:

- `'Gaussian'` — A Gaussian copula
- `'t'` — A *t* copula with degrees of freedom specified using `DegreesOfFreedom`.

Data Types: `char` | `string`

`DegreesOfFreedom` — Degrees of freedom for t copula
`5` (default) | nonnegative numeric value

Degrees of freedom for a *t* copula, specified as the comma-separated pair consisting of `'DegreesOfFreedom'` and a nonnegative numeric value. If `Copula` is set to `'Gaussian'`, the `DegreesOfFreedom` parameter is ignored.

Data Types: `double`

`BlockSize` — Number of scenarios to process in each iteration
nonnegative numeric value

Number of scenarios to process in each iteration, specified as the comma-separated pair consisting of `'BlockSize'` and a nonnegative numeric value.

If unspecified, `BlockSize` defaults to a value of approximately 1,000,000 / (Number-of-counterparties). For example, if there are 100 counterparties, the default `BlockSize` is 10,000 scenarios.

Data Types: `double`

Output Arguments

`cdc` — Updated `creditDefaultCopula` object
object

Updated `creditDefaultCopula` object. The object is populated with the simulated `PortfolioLosses`.

For more information on a `creditDefaultCopula` object, see `creditDefaultCopula`.

Note In the `simulate` function, the `Weights` (specified when using `creditDefaultCopula`) are transformed to ensure that the latent variables have a mean of `0` and a variance of `1`.

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`table` | `creditDefaultCopula` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios`

Topics

"Credit Simulation Using Copulas" on page 4-2
"creditDefaultCopula Simulation Workflow" on page 4-5
"Modeling Correlated Defaults with Copulas" on page 4-18
"One-Factor Model Calibration" on page 4-64
"Corporate Credit Risk" on page 1-3
"Credit Simulation Using Copulas" on page 4-2

External Websites

Parallel Computing with MATLAB (53 min 27 sec)

confidenceBands

Confidence interval bands

Syntax

```
cbTable = confidenceBands(cmc)
cbTable = confidenceBands(cmc,Name,Value)
```

Description

`cbTable = confidenceBands(cmc)` returns a table of the requested risk measure and its associated confidence bands. Use `confidenceBands` to investigate how the values of a risk measure and its associated confidence interval converge as the number of scenarios increases. Before you run the `confidenceBands` function, you must run the `simulate` function. For more information on using a `creditMigrationCopula` object, see `creditMigrationCopula`.

`cbTable = confidenceBands(cmc,Name,Value)` adds optional name-value pair arguments.

Examples

Generate a Table of the Associated Confidence Bands for a Requested Risk Measure for a `creditMigrationCopula` Object

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a `creditMigrationCopula` object with a four-factor model using `creditMigrationCopula`.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioValues: []
```

Set the `VaRLevel` to 99%.

```
cmc.VaRLevel = 0.99;
```

Use the `simulate` function to simulate 100,000 scenarios, and then use the `confidenceBands` function to generate the `cbTable`.

```
cmc = simulate(cmc,1e5);
cbTable = confidenceBands(cmc,'RiskMeasure','Std','ConfidenceIntervalLevel',0.9,'NumPoints',50);
cbTable(1:10,:)
```

```
ans=10×4 table
    NumScenarios    Lower     Std      Upper
    ____________    _____    _____    _____

        2000        11996    12308    12637
        4000        12871    13108    13354
        6000        12556    12744    12939
        8000        12830    12997    13168
       10000        12702    12850    13001
       12000        12784    12920    13059
       14000        12895    13022    13151
       16000        12747    12864    12983
       18000        12948    13060    13174
       20000        12971    13077    13186
```

Input Arguments

`cmc` — `creditMigrationCopula` object
object

`creditMigrationCopula` object obtained after running the `simulate` function.

For more information on `creditMigrationCopula` objects, see `creditMigrationCopula`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cbTable = confidenceBands(cmc,'RiskMeasure','Std','ConfidenceIntervalLevel',0.9,'NumPoints',50)`

`RiskMeasure` — Risk measure to investigate
`'CVaR'` (default) | character vector or string with values `'EL'`, `'Std'`, `'VaR'`, or `'CVaR'`

Risk measure to investigate, specified as the comma-separated pair consisting of `'RiskMeasure'` and a character vector or string. Possible values are:

- `'EL'` — Expected loss, the mean of portfolio losses
- `'Std'` — Standard deviation of the losses

- 'VaR' — Value at risk at the threshold specified by the VaRLevel property of the creditMigrationCopula object
- 'CVaR' — Conditional VaR at the threshold specified by the VaRLevel property of the creditMigrationCopula object

Data Types: char | string

ConfidenceIntervalLevel — Confidence interval level
0.95 (default) | numeric between 0 and 1

Confidence interval level, specified as the comma-separated pair consisting of 'ConfidenceIntervalLevel' and a numeric between 0 and 1. For example, if you specify 0.95, a 95% confidence interval is reported in the output table (cbTable).

Data Types: double

NumPoints — Number of scenario samples to report
100 (default) | nonnegative integer

Number of scenario samples to report, specified as the comma-separated pair consisting of 'NumPoints' and a nonnegative integer. The default is 100, meaning that confidence bands are reported at 100 evenly spaced points of increasing sample size ranging from 0 to the total number of simulated scenarios.

Note NumPoints must be a numeric scalar greater than 1. NumPoints is typically much smaller than total number of scenarios simulated. You can use confidenceBands to obtain a qualitative idea of how fast a risk measure and its confidence interval are converging. Specifying a large value for NumPoints is not recommended and can potentially cause performance issues with confidenceBands.

Data Types: double

Output Arguments

cbTable — Requested risk measure and associated confidence bands
table

Requested risk measure and associated confidence bands at each of the NumPoints scenario sample sizes, returned as a table containing the following columns:

- NumScenarios — Number of scenarios at the sample point
- Lower — Lower confidence band
- *RiskMeasure* — Requested risk measure, where the column takes its name from whatever risk measure is requested with the optional input RiskMeasure
- Upper — Upper confidence band

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`table` | `creditMigrationCopula` | `simulate` | `portfolioRisk` | `riskContribution` | `getScenarios`

Topics

"creditMigrationCopula Simulation Workflow" on page 4-10
"One-Factor Model Calibration" on page 4-64
"Credit Rating Migration Risk" on page 1-10

getScenarios

Counterparty scenarios

Syntax

```
scenarios = getScenarios(cmc,scenarioIndices)
```

Description

`scenarios = getScenarios(cmc,scenarioIndices)` returns counterparty scenario details as a matrix of individual values for each counterparty for the scenarios requested in `scenarioIndices`.

Before you use the `getScenarios` function, you must run the `simulate` function. For more information on using a `creditMigrationCopula` object, see `creditMigrationCopula`.

Examples

Compute Individual Values for Each Counterparty

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a `creditMigrationCopula` object with a four-factor model using `creditMigrationCopula`.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioValues: []
```

Set the `VaRLevel` to 99%.

```
cmc.VaRLevel = 0.99;
```

Use the `simulate` function to simulate 100,000 scenarios, and then use the `getScenarios` function to generate the `scenarios` matrix.

```
cmc = simulate(cmc,1e5);
scenarios = getScenarios(cmc,[2,3]);
scenarios(1:10,:)
```

```
ans = 10×2
10^4 ×

    1.3082    1.3216
    0.2893    0.2893
    0.9788    0.9754
    0.4503    0.4503
    1.0376    1.0376
    0.5795    0.5795
    0.5350    0.5350
    0.4956    0.4956
    0.3537    0.3537
    2.3492    2.3492
```

Input Arguments

cmc — creditMigrationCopula object

object

creditMigrationCopula object obtained after running the simulate function.

For more information on creditMigrationCopula objects, see creditMigrationCopula.

scenarioIndices — Specifies which scenarios are returned

vector

Specifies which scenarios are returned, entered as a vector.

Output Arguments

scenarios — Counterparty values

matrix

Counterparty values, returned as NumCounterparties-by-N matrix, where N is the number of elements in scenarioIndices.

Note If the number of scenarios requested is very large, then the output matrix, scenarios, could be very large, and potentially limited by the available machine memory.

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance*. Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`creditMigrationCopula` | `simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands`

Topics

"creditMigrationCopula Simulation Workflow" on page 4-10
"One-Factor Model Calibration" on page 4-64
"Credit Rating Migration Risk" on page 1-10

portfolioRisk

Generate portfolio-level risk measurements

Syntax

```
[riskMeasures,confidenceIntervals] = portfolioRisk(cmc)
[riskMeasures,confidenceIntervals] = portfolioRisk(cmc,Name,Value)
```

Description

`[riskMeasures,confidenceIntervals] = portfolioRisk(cmc)` returns tables of risk measurements for the portfolio losses. Before you use the `portfolioRisk` function, run the `simulate` function. For more information on using a `creditMigrationCopula` object, see `creditMigrationCopula`.

`[riskMeasures,confidenceIntervals] = portfolioRisk(cmc,Name,Value)` adds an optional name-value pair argument for `ConfidenceIntervalLevel`.

Examples

Generate Tables for Risk Measure and Confidence Intervals for a creditMigrationCopula Object

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a `creditMigrationCopula` object with a four-factor model using `creditMigrationCopula`.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioValues: []
```

Set the `VaRLevel` to 99%.

```
cmc.VaRLevel = 0.99;
```

Use the `simulate` function to simulate 100,000 scenarios, and then use the `portfolioRisk` function to generate the `riskMeasure` and `ConfidenceIntervals` tables.

```
cmc = simulate(cmc,1e5);
[riskMeasure,confidenceIntervals] = portfolioRisk(cmc,'ConfidenceIntervalLevel',0.9)
```

```
riskMeasure=1×4 table
      EL        Std       VaR      CVaR 
    ______     _____     _____     _____

    4515.9     12963     57176     83975

confidenceIntervals=1×4 table
           EL                  Std                 VaR                 CVaR     
    ________________     ______________     ______________     ______________

    4448.5    4583.4     12916     13011     56012     58278     82433     85517
```

Input Arguments

cmc — creditMigrationCopula object
object

`creditMigrationCopula` object obtained after running the `simulate` function.

For more information on `creditMigrationCopula` objects, see `creditMigrationCopula`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[riskMeasure,confidenceIntervals] = portfolioRisk(cmc,'ConfidenceIntervalLevel',0.9)`

ConfidenceIntervalLevel — Confidence interval level
`0.95` (default) | numeric between `0` and `1`

Confidence interval level, specified as the comma-separated pair consisting of `'ConfidenceIntervalLevel'` and a numeric between `0` and `1`. For example, if you specify `0.95`, a 95% confidence interval is reported in the output table (`riskMeasures`).

Data Types: `double`

Output Arguments

riskMeasures — Risk measures
table

Risk measures, returned as a table containing the following columns:

- `EL` — Expected loss, the mean of portfolio losses
- `Std` — Standard deviation of the losses
- `VaR` — Value at risk at the threshold specified by the `VaRLevel` property of the `creditMigrationCopula` object
- `CVaR` — Conditional VaR at the threshold specified by the `VaRLevel` property of the `creditMigrationCopula` object

`confidenceIntervals` — Confidence intervals
table

Confidence intervals, returned as a table of confidence intervals corresponding to the portfolio risk measures reported in the `riskMeasures` table. Confidence intervals are reported at the level specified by the `ConfidenceIntervalLevel` parameter.

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`table` | `creditMigrationCopula` | `simulate` | `riskContribution` | `confidenceBands` | `getScenarios`

Topics

"creditMigrationCopula Simulation Workflow" on page 4-10
"One-Factor Model Calibration" on page 4-64
"Credit Rating Migration Risk" on page 1-10

riskContribution

Generate risk contributions for each counterparty in portfolio

Syntax

```
Contributions = riskContribution(cmc)
Contributions = riskContribution(cmc,Name,Value)
```

Description

`Contributions = riskContribution(cmc)` returns a table of risk contributions for each counterparty in the portfolio. The risk `Contributions` table allocates the full portfolio risk measures to each counterparty, such that the counterparty risk contributions sum to the portfolio risks reported by `portfolioRisk`.

Note When creating a `creditMigrationCopula` object, you can set the `'UseParallel'` property if you have Parallel Computing Toolbox. Once the `'UseParallel'` property is set, parallel processing is used to compute `riskContribution`.

Before you use the `riskContribution` function, you must run the `simulate` function. For more information on using a `creditMigrationCopula` object, see `creditMigrationCopula`.

`Contributions = riskContribution(cmc,Name,Value)` adds an optional name-value pair argument for `VaRWindow`.

Examples

Determine the Risk Contribution for Each Counterparty for a `creditMigrationCopula` Object

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a `creditMigrationCopula` object with a four-factor model using `creditMigrationCopula`.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
```

```
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioValues: []
```

Set the `VaRLevel` to 99%.

```
cmc.VaRLevel = 0.99;
```

Use the `simulate` function to simulate 100,000 scenarios, and then use the `riskContribution` function to generate the `Contributions` table.

```
cmc = simulate(cmc,1e5);
Contributions = riskContribution(cmc);
Contributions(1:10,:)
```

```
ans=10×5 table
    ID      EL       Std       VaR       CVaR 
    __    ______    ______    ______    ______

     1    15.521    41.153    238.72    279.18
     2      8.49    18.838    92.074    122.19
     3    6.0937    20.069    113.22    181.53
     4    6.6964    55.885    272.23    313.25
     5    23.583    73.905    360.32    573.39
     6    10.722    114.97    445.94    728.38
     7    1.8393    84.754    262.32    490.39
     8    11.711    39.768    175.84    253.29
     9    2.2154    4.4038    22.797    31.039
    10    1.7453    2.5545    9.8801    17.603
```

Input Arguments

`cmc` — `creditMigrationCopula` object
object

`creditMigrationCopula` object obtained after running the `simulate` function.

For more information on `creditMigrationCopula` objects, see `creditMigrationCopula`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `Contributions = riskContribution(cmc,'VaRWindow',0.3)`

`VaRWindow` — Size of the window used to compute VaR contributions
`0.05` (default) | numeric between `0` and `1`

Size of the window used to compute VaR contributions, specified as the comma-separated pair consisting of `'VaRWindow'` and a scalar numeric with a percent value. Scenarios in the VaR scenario set are used to calculate the individual counterparty VaR contributions.

The default is `0.05`, meaning that all scenarios with portfolio losses within 5 percent of the VaR are included when computing counterparty VaR contributions.

Data Types: `double`

Output Arguments

`Contributions` — Risk contributions

table

Risk contributions, returned as a table containing the following risk contributions for each counterparty:

- `EL` — Expected loss for the particular counterparty over the scenarios
- `Std` — Standard deviation of loss for the particular counterparty over the scenarios
- `VaR` — Value at risk for the particular counterparty over the scenarios
- `CVaR` — Conditional value at risk for the particular counterparty over the scenarios

The risk `Contributions` table allocates the full portfolio risk measures to each counterparty, such that the counterparty risk contributions sum to the portfolio risks reported by `portfolioRisk`.

More About

Risk Contributions

The `riskContribution` function reports the individual counterparty contributions to the total portfolio risk measures using four risk measures: expected loss (EL), standard deviation (Std), VaR, and CVaR.

- `EL` is the expected loss for each counterparty and is the mean of the counterparty's losses across all scenarios.
- `Std` is the standard deviation for counterparty *i*:

  $$StdCont_i = Std_i \frac{\sum_j Std_j \rho_{ij}}{Std_\rho}$$

  where

  Std_{i} is the standard deviation of losses from counterparty *i*.

  $Std_{\ddot{i}}$ is the standard deviation of portfolio losses.

  ρ_{ij} is the correlation of the losses between counterparties *i* and *j*.
- `VaR` contribution is the mean of a counterparty's losses across all scenarios in which the total portfolio loss is within some small neighborhood around the Portfolio VaR. The default of the `'VaRWindow'` parameter is `0.05` meaning that all scenarios in which the total portfolio loss is within 5% of the portfolio VaR are included in VaR neighborhood.

- `CVaR` is the mean of the counterparty's losses in the set of scenarios in which the total portfolio losses exceed the portfolio VaR.

Version History

Introduced in R2017a

References

[1] Glasserman, P. “Measuring Marginal Risk Contributions in Credit Portfolios.” *Journal of Computational Finance*. Vol. 9, No. 2, Winter 2005/2006.

[2] Gupton, G., Finger, C., and Bhatia, M. *“CreditMetrics – Technical Document.”* J. P. Morgan, New York, 1997.

See Also

`table` | `creditMigrationCopula` | `simulate` | `portfolioRisk` | `confidenceBands` | `getScenarios`

Topics

“creditMigrationCopula Simulation Workflow” on page 4-10
“One-Factor Model Calibration” on page 4-64
“Credit Rating Migration Risk” on page 1-10

External Websites

Parallel Computing with MATLAB (53 min 27 sec)

simulate

Simulate credit migrations using `creditMigrationCopula` object

Syntax

```
cmc = simulate(cmc,NumScenarios)
cmc = simulate( ___ ,Name,Value)
```

Description

`cmc = simulate(cmc,NumScenarios)` performs the full simulation of credit scenarios and computes changes in value due to credit rating changes for the portfolio defined in the `creditMigrationCopula` object. For more information on using a `creditMigrationCopula` object, see `creditMigrationCopula`.

Note When creating a `creditMigrationCopula` object, you can set the `'UseParallel'` property if you have Parallel Computing Toolbox. Once the `'UseParallel'` property is set, parallel processing is used to compute `simulate`.

`cmc = simulate( ___ ,Name,Value)` adds optional name-value pair arguments for (`Copula`, `DegreesOfFreedom`, and `BlockSize`).

Examples

Run a Simulation Using a `creditMigrationCopula` Object

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a `creditMigrationCopula` object with a four-factor model using `creditMigrationCopula`.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9500
          UseParallel: 0
```

```
      PortfolioValues: []
```

Set the `VaRLevel` to 99%.

```
cmc.VaRLevel = 0.99;
```

Use the `simulate` function to simulate 100,000 scenarios. After using simulate, you can then use the `portfolioRisk`, `riskContribution`, `confidenceBands`, and `getScenarios` with the updated `creditMigrationCopula` object.

```
cmc = simulate(cmc,1e5)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioValues: [2.0082e+06 1.9950e+06 1.9933e+06 2.0009e+06 1.9819e+06 1.9955e+06 1.9962e
```

You can use the `riskContribution` function with the `creditMigrationCopula` object to generate the risk `Contributions` table.

```
Contributions = riskContribution(cmc);
Contributions(1:10,:)
```

```
ans=10×5 table
    ID      EL       Std       VaR       CVaR 
    __    ______    ______    ______    ______

     1    15.521    41.153    238.72    279.18
     2      8.49    18.838    92.074    122.19
     3    6.0937    20.069    113.22    181.53
     4    6.6964    55.885    272.23    313.25
     5    23.583    73.905    360.32    573.39
     6    10.722    114.97    445.94    728.38
     7    1.8393    84.754    262.32    490.39
     8    11.711    39.768    175.84    253.29
     9    2.2154    4.4038    22.797    31.039
    10    1.7453    2.5545    9.8801    17.603
```

Input Arguments

`cmc` — `creditMigrationCopula` object
object

`creditMigrationCopula` object, obtained from `creditMigrationCopula`.

For more information on a `creditMigrationCopula` object, see `creditMigrationCopula`.

`NumScenarios` — Number of scenarios to simulate

nonnegative integer

Number of scenarios to simulate, specified as a nonnegative integer. Scenarios are processed in blocks to conserve machine resources.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cmc = simulate(cmc,NumScenarios,'Copula','t','DegreesOfFreedom',5,'BlockSize',1000)`

`Copula` — Type of copula

`'Gaussian'` (default) | character vector or string with values `'Gaussian'` or `'t'`

Type of copula, specified as the comma-separated pair consisting of `'Copula'` and a character vector or string. Possible values are:

- `'Gaussian'` — Gaussian copula
- `'t'` — *t* copula with degrees of freedom specified by using `DegreesOfFreedom`.

Data Types: `char` | `string`

`DegreesOfFreedom` — Degrees of freedom for t copula

`5` (default) | nonnegative numeric value

Degrees of freedom for a *t* copula, specified as the comma-separated pair consisting of `'DegreesOfFreedom'` and a nonnegative numeric value. If `Copula` is set to `'Gaussian'`, the `DegreesOfFreedom` parameter is ignored.

Data Types: `double`

`BlockSize` — Number of scenarios to process in each iteration

nonnegative numeric value

Number of scenarios to process in each iteration, specified as the comma-separated pair consisting of `'BlockSize'` and a nonnegative numeric value. Adjust `BlockSize` for performance, especially when executing large simulations.

If unspecified, `BlockSize` defaults to a value of approximately 1,000,000 / (Number-of-counterparties). For example, if there are 100 counterparties, the default `BlockSize` is 10,000 scenarios.

Data Types: `double`

Output Arguments

`cmc` — Updated `creditMigrationCopula` object

object

`creditMigrationCopula` object, returned as an updated object that is populated with the simulated `PortfolioValues`.

For more information on a `creditMigrationCopula` object, see `creditMigrationCopula`.

Note In the `simulate` function, the `Weights` (specified when using `creditMigrationCopula`) are transformed to ensure that the latent variables have a mean of `0` and a variance of `1`.

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`table` | `creditMigrationCopula` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios`

Topics

"creditMigrationCopula Simulation Workflow" on page 4-10
"One-Factor Model Calibration" on page 4-64
"Credit Rating Migration Risk" on page 1-10

External Websites

Parallel Computing with MATLAB (53 min 27 sec)

displaypoints

Return points per predictor per bin for a `compactCreditScorecard` object

Syntax

```
PointsInfo = displaypoints(csc)
[PointsInfo,MinScore,MaxScore] = displaypoints(csc)
[PointsInfo,MinScore,MaxScore] = displaypoints(___,Name,Value)
```

Description

`PointsInfo = displaypoints(csc)` returns a table of points for all bins of all predictor variables used in the `compactCreditScorecard` object. The `PointsInfo` table displays information on the predictor name, bin labels, and the corresponding points per bin.

`[PointsInfo,MinScore,MaxScore] = displaypoints(csc)` returns a table of points for all bins of all predictor variables used in the `compactCreditScorecard` object. The `PointsInfo` table displays information on the predictor name, bin labels, and the corresponding points per bin and `displaypoints`. In addition, the optional `MinScore` and `MaxScore` values are returned.

`[PointsInfo,MinScore,MaxScore] = displaypoints(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Display Points for a `compactCreditScorecard` Object

To create a `compactCreditScorecard` object, first create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData.mat
sc = creditscorecard(data)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                     Data: [1200x11 table]
```

Before creating a `compactCreditScorecard` object, you must use `autobinning` and `fitmodel` with the `creditscorecard` object.

```
sc = autobinning(sc);
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Use the `creditscorecard` object with `compactCreditScorecard` to create a `compactCreditScorecard` object.

```
csc = compactCreditScorecard(sc)
```

```
csc = 
  compactCreditScorecard with properties:

              Description: ''
                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
        NumericPredictors: {'CustAge'  'CustIncome'  'TmWBank'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
            PredictorVars: {'CustAge'  'ResStatus'  'EmpStatus'  'CustIncome'  'TmWBank'  'Other(
```

Then use `displaypoints` with the `compactCreditScorecard` object to return a table of points for all bins of all predictor variables used in the `compactCreditScorecard` object.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(csc)
```

```
PointsInfo=37×3 table
      Predictors            Bin            Points
    ______________    ________________    _________
```

```
    {'CustAge'   }    {'[-Inf,33)'   }     -0.15894
    {'CustAge'   }    {'[33,37)'     }     -0.14036
    {'CustAge'   }    {'[37,40)'     }    -0.060323
    {'CustAge'   }    {'[40,46)'     }     0.046408
    {'CustAge'   }    {'[46,48)'     }      0.21445
    {'CustAge'   }    {'[48,58)'     }      0.23039
    {'CustAge'   }    {'[58,Inf]'    }        0.479
    {'CustAge'   }    {'<missing>'   }          NaN
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }          NaN
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }          NaN
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮

MinScore = -1.3100

MaxScore = 3.0726
```

`displaypoints` always displays a `'<missing>'` bin for each predictor. The value of the `'<missing>'` bin comes from the initial `creditscorecard` object, and the `'<missing>'` bin is set to `NaN` whenever the scorecard model has no information on how to assign points to missing data.

To configure the points for the `'<missing>'` bin, you must use the initial `creditscorecard` object. For predictors that have missing values in the training set, the points for the `'<missing>'` bin are estimated from the data if the `'BinMissingData'` name-value pair argument is set to `true` using `creditscorecard`. When the `'BinMissingData'` parameter is set to `false`, or when the data contains no missing values in the training set, use the `'Missing'` name-value pair argument in `formatpoints` to indicate how to assign points to the missing data. Then, rebuild the `compactCreditScorecard` object and rerun `displaypoints`. Here is an example of this workflow:

```
sc = formatpoints(sc,'Missing','minpoints');
csc = compactCreditScorecard(sc);
[PointsInfo,MinScore,MaxScore] = displaypoints(csc)
```

```
PointsInfo=37×3 table
      Predictors           Bin            Points  
    ______________    ________________    _________

    {'CustAge'   }    {'[-Inf,33)'   }     -0.15894
    {'CustAge'   }    {'[33,37)'     }     -0.14036
    {'CustAge'   }    {'[37,40)'     }    -0.060323
    {'CustAge'   }    {'[40,46)'     }     0.046408
    {'CustAge'   }    {'[46,48)'     }      0.21445
    {'CustAge'   }    {'[48,58)'     }      0.23039
    {'CustAge'   }    {'[58,Inf]'    }        0.479
    {'CustAge'   }    {'<missing>'   }     -0.15894
    {'ResStatus' }    {'Tenant'      }    -0.031252
    {'ResStatus' }    {'Home Owner'  }      0.12696
    {'ResStatus' }    {'Other'       }      0.37641
    {'ResStatus' }    {'<missing>'   }    -0.031252
    {'EmpStatus' }    {'Unknown'     }    -0.076317
    {'EmpStatus' }    {'Employed'    }      0.31449
    {'EmpStatus' }    {'<missing>'   }    -0.076317
```

```
    {'CustIncome'}    {'[-Inf,29000)'}     -0.45716
      ⋮

MinScore = -1.3100

MaxScore = 3.0726
```

Display Points for a compactCreditScorecard Object That Contains Missing Data

To create a `compactCreditScorecard` object, first create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011). Using the `dataMissing` dataset, set the `'BinMissingData'` indicator to `true`.

```
load CreditCardData.mat
sc = creditscorecard(dataMissing,'BinMissingData',true);
```

Before creating a `compactCreditScorecard` object, you must use `autobinning` and `fitmodel` with the `creditscorecard` object. First, use `autobinning` with the `creditscorecard` object.

```
sc = autobinning(sc);
```

The binning map or rules for categorical data are summarized in a "category grouping" table, returned as an optional output. By default, each category is placed in a separate bin. Here is the information for the predictor `ResStatus`.

```
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi=5×6 table
          Bin           Good    Bad     Odds        WOE         InfoValue 
    ______________    ____    ___    ______    _________    __________

    {'Tenant'    }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'     }     128      52    2.4615      0.19637     0.0055808
    {'<missing>' }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'    }     803     397    2.0227          NaN     0.0092627

cg=3×2 table
       Category       BinNumber
    ______________    _________

    {'Tenant'    }        1    
    {'Home Owner'}        2    
    {'Other'     }        3    
```

To group categories `'Tenant'` and `'Other'`, modify the category grouping table `cg`, so the bin number for `'Other'` is the same as the bin number for `'Tenant'`. Then use `modifybins` to update the `creditscorecard` object.

```
cg.BinNumber(3) = 2;
sc = modifybins(sc,'ResStatus','Catg',cg);
```

Display the updated bin information using `bininfo`. Note that the bin labels has been updated and that the bin membership information is contained in the category grouping `cg`.

```
[bi,cg] = bininfo(sc,'ResStatus')
```

```
bi=4×6 table
         Bin          Good    Bad     Odds        WOE        InfoValue 
    ______________    ____    ___    ______    _________    __________

    {'Group1'   }     296     161    1.8385    -0.095463     0.0035249
    {'Group2'   }     480     223    2.1525     0.062196     0.0022419
    {'<missing>'}      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'   }     803     397    2.0227          NaN       0.00579

cg=3×2 table
       Category       BinNumber
    ______________    _________

    {'Tenant'    }        1    
    {'Home Owner'}        2    
    {'Other'     }        2    
```

Use `formatpoints` with the `'Missing'` name-value pair argument to indicate that missing data is assigned `'maxpoints'`.

```
sc = formatpoints(sc,'BasePoints',true,'Missing','maxpoints','WorstAndBest',[300 800]);
```

Use `fitmodel` to fit the model.

```
sc = fitmodel(sc,'VariableSelection','fullmodel','Display','Off');
```

Use the `creditscorecard` object with `compactCreditScorecard` to create a `compactCreditScorecard` object.

```
csc = compactCreditScorecard(sc)
```

```
csc = 
  compactCreditScorecard with properties:

              Description: ''
                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalar
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
```

Then use `displaypoints` with the `compactCreditScorecard` object to return a table of points for all bins of all predictor variables used in the `compactCreditScorecard` object. By setting the `displaypoints` name-value pair argument for '`ShowCategoricalMembers'` to `true`, all the members contained in each individual group are displayed.

```
[PointsInfo,MinScore,MaxScore] = displaypoints(csc,'ShowCategoricalMembers',true)
```

```
PointsInfo=51×3 table
      Predictors            Bin          Points 
```

```
    _______________    ______________    _______

    {'BasePoints' }    {'BasePoints'}     535.25
    {'CustID'     }    {'[-Inf,121)'}     12.085
    {'CustID'     }    {'[121,241)' }     5.4738
    {'CustID'     }    {'[241,1081)'}    -1.4061
    {'CustID'     }    {'[1081,Inf]'}    -7.2217
    {'CustID'     }    {'<missing>' }     12.085
    {'CustAge'    }    {'[-Inf,33)' }    -25.973
    {'CustAge'    }    {'[33,37)'   }     -22.67
    {'CustAge'    }    {'[37,40)'   }    -17.122
    {'CustAge'    }    {'[40,46)'   }    -2.8071
    {'CustAge'    }    {'[46,48)'   }     9.5034
    {'CustAge'    }    {'[48,51)'   }     10.913
    {'CustAge'    }    {'[51,58)'   }     13.844
    {'CustAge'    }    {'[58,Inf]'  }     37.541
    {'CustAge'    }    {'<missing>' }    -9.7271
    {'TmAtAddress'}    {'[-Inf,23)' }    -9.3683
          :

MinScore = 300.0000

MaxScore = 800.0000
```

Input Arguments

`csc` — Compact credit scorecard model
`compactCreditScorecard` object

Compact credit scorecard model, specified as a `compactCreditScorecard` object.

To create a `compactCreditScorecard` object, use `compactCreditScorecard` or `compact` from Financial Toolbox.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PointsInfo,MinScore,MaxScore] = displaypoints(csc,'ShowCategoricalMembers',true)`

`ShowCategoricalMembers` — Indicator for how to display bins labels of categories that were grouped together
`false` (default) | `true` or `false`

Indicator for how to display bins labels of categories that were grouped together, specified as the comma-separated pair consisting of `'ShowCategoricalMembers'` and a logical scalar with a value of `true` or `false`.

By default, when 'ShowCategoricalMembers' is false, bin labels are displayed as Group1, Group2,...,Group*n*, or if the bin labels were modified in creditscorecard, then the user-defined bin label names are displayed.

If 'ShowCategoricalMembers' is true, all the members contained in each individual group are displayed.

Data Types: logical

Output Arguments

PointsInfo — One row per bin, per predictor, with the corresponding points
table

One row per bin, per predictor, with the corresponding points, returned as a table. For example:

Predictors	Bin	Points
Predictor_1	Bin_11	Points_11
Predictor_1	Bin_12	Points_12
Predictor_1	Bin_13	Points_13
	...	...
Predictor_1	'<missing>'	NaN (Default)
Predictor_2	Bin_21	Points_21
Predictor_2	Bin_22	Points_22
Predictor_2	Bin_23	Points_23
	...	...
Predictor_2	'<missing>'	NaN (Default)
Predictor_*j*	Bin_*ji*	Points_*ji*
	...	...
Predictor_*j*	'<missing>'	NaN (Default)

displaypoints always displays a '<missing>' bin for each predictor. The value of the '<missing>' bin comes from the initial creditscorecard object, and the '<missing>' bin is set to NaN whenever the scorecard model has no information on how to assign points to missing data.

To configure the points for the '<missing>' bin, you must use the initial creditscorecard object. For predictors that have missing values in the training set, the points for the '<missing>' bin are estimated from the data if the 'BinMissingData' name-value pair argument for is set to true using creditscorecard. When the 'BinMissingData' parameter is set to false, or when the data contains no missing values in the training set, use the 'Missing' name-value pair argument in formatpoints to indicate how to assign points to the missing data. Then rebuild the compactCreditScorecard object and rerun displaypoints.

When base points are reported separately (see formatpoints), the first row of the returned PointsInfo table contains the base points.

MinScore — Minimum possible total score
scalar

Minimum possible total score, returned as a scalar.

Note Minimum score is the lowest possible total score in the mathematical sense, independently of whether a low score means high risk or low risk.

MaxScore — Maximum possible total score

scalar

Maximum possible total score, returned as a scalar.

Note Maximum score is the highest possible total score in the mathematical sense, independently of whether a high score means high risk or low risk.

Algorithms

The points for predictor *j* and bin *i* are, by default, given by

```
Points_ji = (Shift + Slope*b0)/p + Slope*(bj*WOEj(i))
```

where *bj* is the model coefficient of predictor *j*, *p* is the number of predictors in the model, and WOE*j*(*i*) is the Weight of Evidence (WOE) value for the *i*-th bin corresponding to the j-th model predictor. `Shift` and `Slope` are scaling constants.

When the base points are reported separately (see the `formatpoints` name-value pair argument `BasePoints`), the base points are given by

```
Base Points = Shift + Slope*b0,
```

and the points for the *j*-th predictor, *i*-th row are given by

```
Points_ji = Slope*(bj*WOEj(i))).
```

By default, the base points are not reported separately.

The minimum and maximum scores are:

```
MinScore = Shift + Slope*b0 + min(Slope*b1*WOE1) + ... +min(Slope*bp*WOEp)),
MaxScore = Shift + Slope*b0 + max(Slope*b1*WOE1) + ... +max(Slope*bp*WOEp)).
```

Use `formatpoints` to control the way points are scaled, rounded, and whether the base points are reported separately. See `formatpoints` for more information on format parameters and for details and formulas on these formatting options.

Version History

Introduced in R2019a

References

[1] Anderson, R. *The Credit Scoring Toolkit*. Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS*. lulu.com, 2011.

See Also

`compactCreditScorecard` | `score` | `probdefault` | `validatemodel`

Topics

“compactCreditScorecard Object Workflow” on page 3-57
“Case Study for Credit Scorecard Analysis”
“Credit Scorecard Modeling with Missing Values”
“Credit Scorecard Modeling Workflow”
“About Credit Scorecards”

fitEADModel

Create specified EAD model object type

Syntax

```
eadModel = fitEADModel(data,ModelType)
eadModel = fitEADModel( ___ ,Name=Value)
```

Description

`eadModel = fitEADModel(data,ModelType)` creates an exposure at default (EAD) model object specified by `data` and `ModelType`. `fitEADModel` takes in credit data in table form and fits an EAD model. `ModelType` is supported for a `Regression`, `Tobit`, or `Beta` model.

`eadModel = fitEADModel( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax. The available optional name-value arguments depend on the specified `ModelType`.

Examples

Create Regression EAD Model

This example shows how to use `fitEADModel` to create a `Regression` model for exposure at default (EAD).

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

UtilizationRate	Age	Marriage	Limit	Drawn	EAD
0.24359	25	not married	44776	10907	44740
0.96946	44	not married	2.1405e+05	2.0751e+05	40678
0	40	married	1.6581e+05	0	1.6567e+05
0.53242	38	not married	1.7375e+05	92506	1593.5
0.2583	30	not married	26258	6782.5	54.175
0.17039	54	married	1.7357e+05	29575	576.69
0.18586	27	not married	19590	3641	998.49
0.85372	42	not married	2.0712e+05	1.7682e+05	1.6454e+05

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Regression` or `Tobit`.

```
ModelType = Regression ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Regression EAD Model

Use `fitEADModel` to create a `Regression` model using `EADData`.

```
eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'}, ...
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Regression with properties:

    ConversionTransform: "logit"
      BoundaryTolerance: 1.0000e-07
                ModelID: "Regression"
            Description: ""
        UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
          PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
            ResponseVar: "EAD"
               LimitVar: "Limit"
               DrawnVar: "Drawn"
      ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'BoundaryTolerance'`, `'LimitVar'`, and `'DrawnVar'` name-value arguments to modify the transformation.

```
eadModel.UnderlyingModel
```

```
ans = 
Compact linear regression model:
    EAD_lcf_logit ~ 1 + UtilizationRate + Age + Marriage

Estimated Coefficients:
                            Estimate        SE         tStat       pValue   
                           _________    _________    _______    __________

    (Intercept)              -2.4745      0.29892    -8.2781    1.6448e-16
    UtilizationRate           6.0045      0.19901     30.172    7.703e-182
    Age                    -0.020095    0.0073019     -2.752     0.0059471
    Marriage_not married    -0.03509      0.13935    -0.2518        0.8012


Number of observations: 4378, Error degrees of freedom: 4374
Root Mean Squared Error: 4.48
R-squared: 0.173,  Adjusted R-Squared: 0.173
F-statistic vs. constant model: 305, p-value = 5.7e-180
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-vale argument.

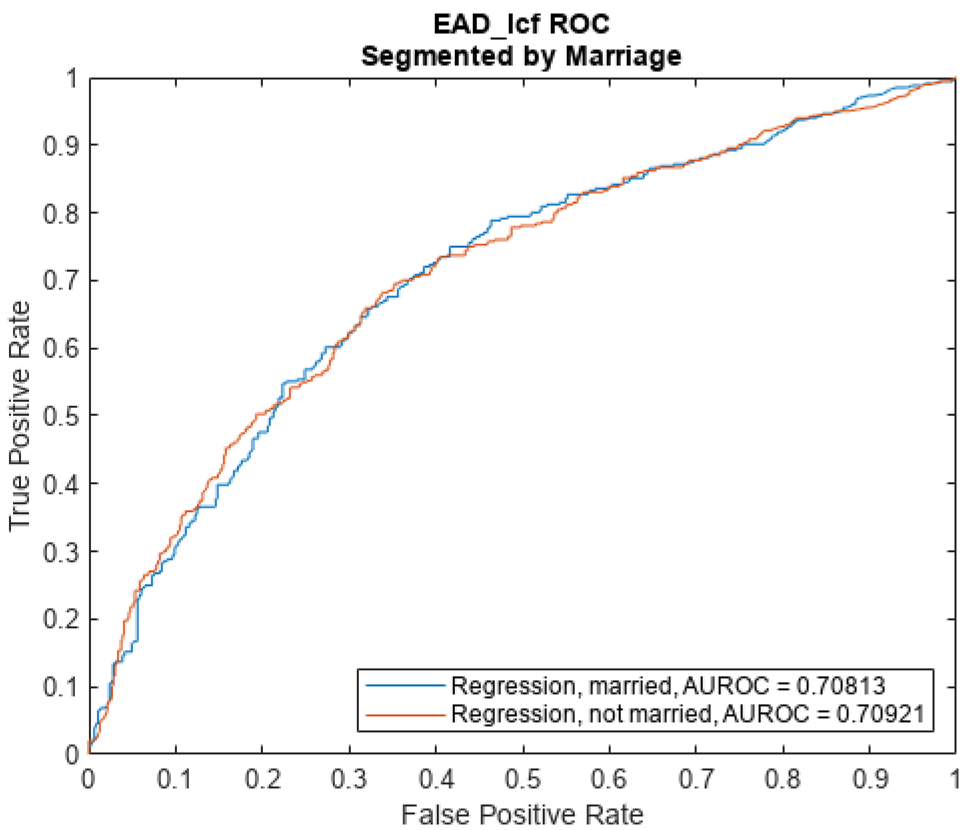
```
predictedEAD = predict(eadModel, EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel, EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

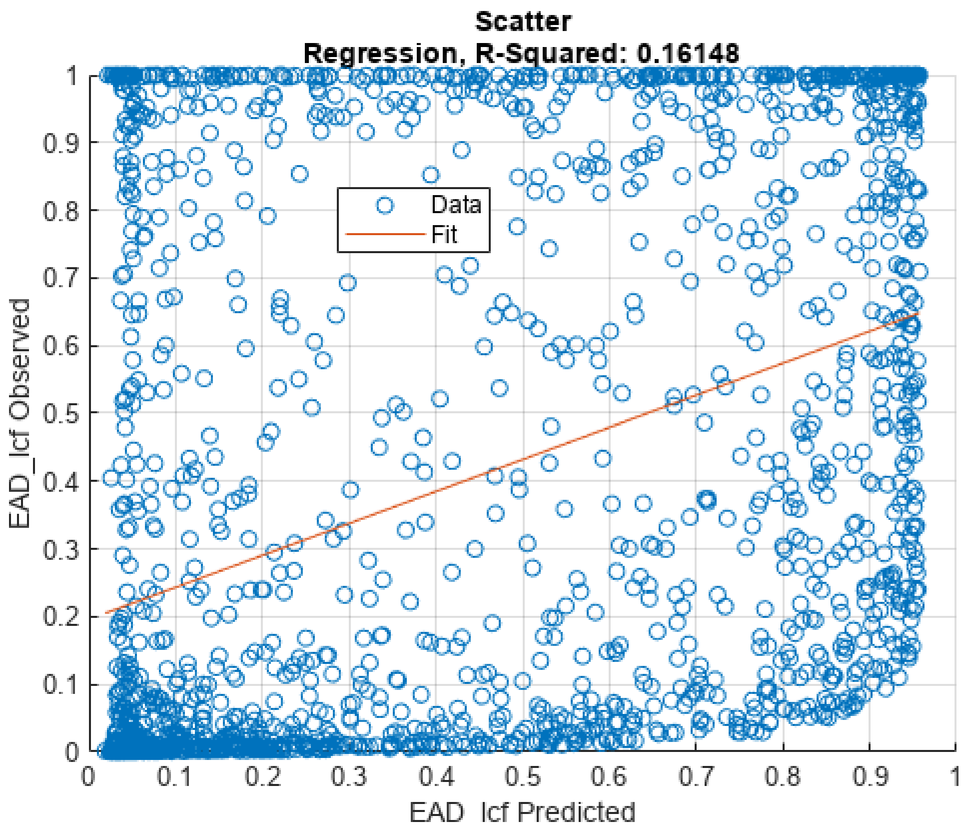
```
ModelLevel = ConversionMeasure ;
[DiscMeasure1, DiscData1] = modelDiscrimination(eadModel, EADData(TestInd,:),ModelLevel=ModelLeve
modelDiscriminationPlot(eadModel, EADData(TestInd, :), ModelLevel=ModelLevel,SegmentBy="Marriage'
```



Use `modelCalibration` and then `modelCalibrationPlot` to show a scatter plot of the predictions.

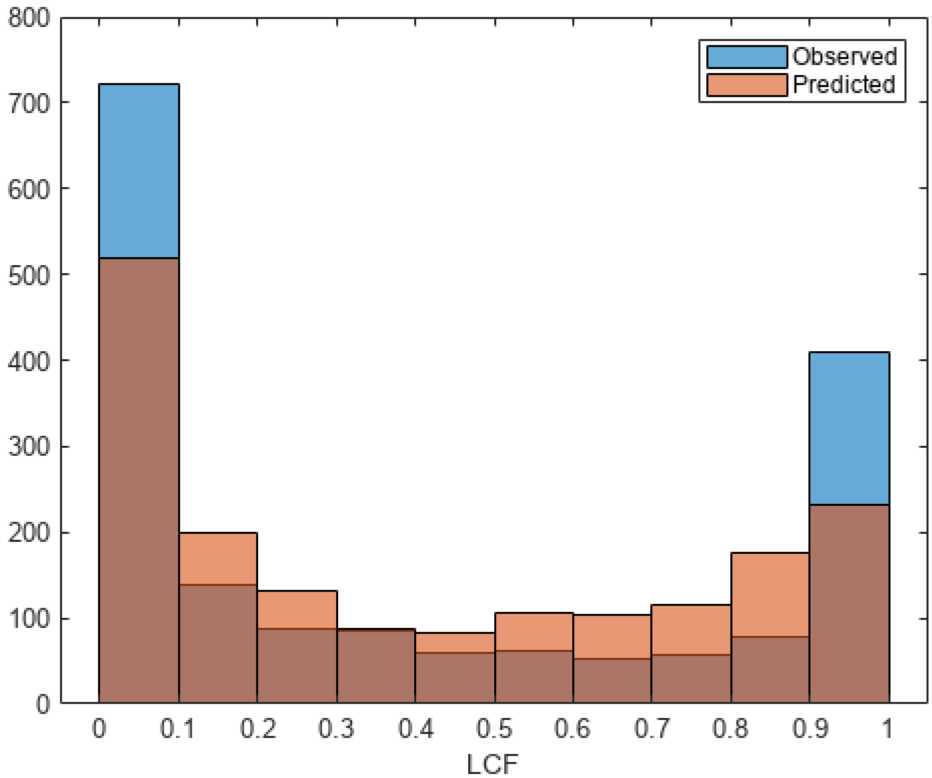
```
YData = Observed ;
```

```
[CalMeasure1, CalData1] = modelCalibration(eadModel, EADData(TestInd,:), ModelLevel=ModelLevel);
modelCalibrationPlot(eadModel, EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Plot a histogram of observed with respect to the predicted EAD.

```
figure;
histogram(CalData1.Observed);
hold on;
histogram(CalData1.(('Predicted_' + ModelType)));
xlabel(ConversionMeasure);
legend('Observed', 'Predicted');
```



Input Arguments

data — Data for loss given default
table

Data for loss given default, specified as a table.

Data Types: `table`

`ModelType` — Type of EAD model
character vector with values `'Regression'`, `'Tobit'`, or `'Beta'` | string with values `"Regression"`, `"Tobit"`, or `"Beta"`

Type of EAD model, specified as a scalar string or character vector. Use one of following values:

- `Regression` —Transform the EAD response variable and fit a linear regression model. For more information, see "Exposure at Default Regression Models" on page 6-645.
- `Tobit` — Fit a Tobit "censored" regression model. For more information, see "Exposure at Default Tobit Models" on page 6-656.
- `Beta` — Fit a Beta regression model. For more information, see "Beta Regression Models" on page 6-669.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'},ConversionMeasure="ccf",DrawnVar='Drawn',LimitVar='Limit',ResponseVar='EAD')`

The available name-value arguments depend on the value you specify for `ModelType`.

- `Regression` — See “Regression Name-Value Arguments” on page 6-638.
- `Tobit` — See “Tobit Name-Value Arguments” on page 6-648.
- `Beta` — See “Beta Name-Value Arguments” on page 6-661.

Output Arguments

`eadModel` — Exposure at default model
`eadModel` object

Loss given default model, returned as an `eadModel` object for a `Regression`, `Tobit`, or `Beta` model.

Version History

Introduced in R2021b

R2022b: Support Beta regression for EAD model
Behavior changed in R2022b

To create a Beta model use a `ModelType` of `"Beta"`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

`Regression` | `Tobit` | `Beta`

Topics

“Compare Results for Regression and Tobit EAD Models” on page 4-151

"Overview of Exposure at Default Models" on page 1-34

fitLGDModel

Create specified LGD model object type

Syntax

```
lgdModel = fitLGDModel(data,ModelType)
lgdModel = fitLGDModel( ___ ,Name,Value)
```

Description

`lgdModel = fitLGDModel(data,ModelType)` creates a loss given default (LGD) model object specified by `data` and `ModelType`. `fitLGDModel` takes in credit data in table form and fits a LGD model. `ModelType` is supported for `Regression`, `Tobit`, or `Beta`.

`lgdModel = fitLGDModel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The available optional name-value pair arguments depend on the specified `ModelType`.

Examples

Create Regression LGD Model

This example shows how to use `fitLGDModel` to create a `Regression` model for loss given default (LGD).

Load LGD Data

Load the LGD data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Create Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using the `data`.

```
lgdModel = fitLGDModel(data,'regression',...
        'ModelID','Example',...
        'Description','Example LGD regression model.',...
```

```
        'PredictorVars',{'LTV' 'Age' 'Type'},...
        'ResponseVar','LGD');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Example"
          Description: "Example LGD regression model."
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model. The underlying model's response variable is the logit transformation of the LGD response data. Use the `'ResponseTransform'` and `'BoundaryTolerance'` arguments to modify the transformation.

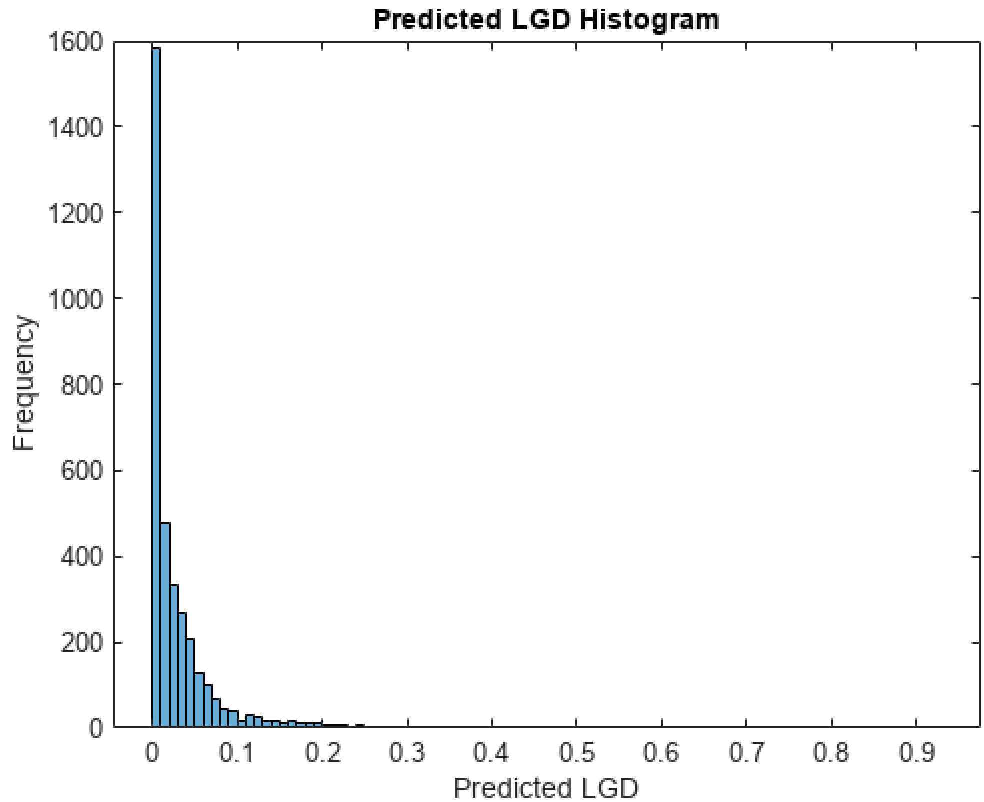
```
lgdModel.UnderlyingModel
```

```
ans = 
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -5.1939      0.28351     -18.32     1.203e-71
    LTV                 3.3217      0.33058     10.048    1.9484e-23
    Age                -1.4953     0.068658    -21.779    1.0596e-98
    Type_investment     1.3813      0.19406     7.1178    1.3259e-12


Number of observations: 3487, Error degrees of freedom: 3483
Root Mean Squared Error: 4.3
R-squared: 0.195,  Adjusted R-Squared: 0.194
F-statistic vs. constant model: 281, p-value = 2.32e-163
```

Predict LGD

For LGD prediction, the LGD model applies the inverse transformation so the predictions are in the LGD scale, not in the transformed scale used to fit the underlying model.

```
predictedLGD = predict(lgdModel,data);
histogram(predictedLGD)
title('Predicted LGD Histogram')
xlabel('Predicted LGD')
ylabel('Frequency')
```

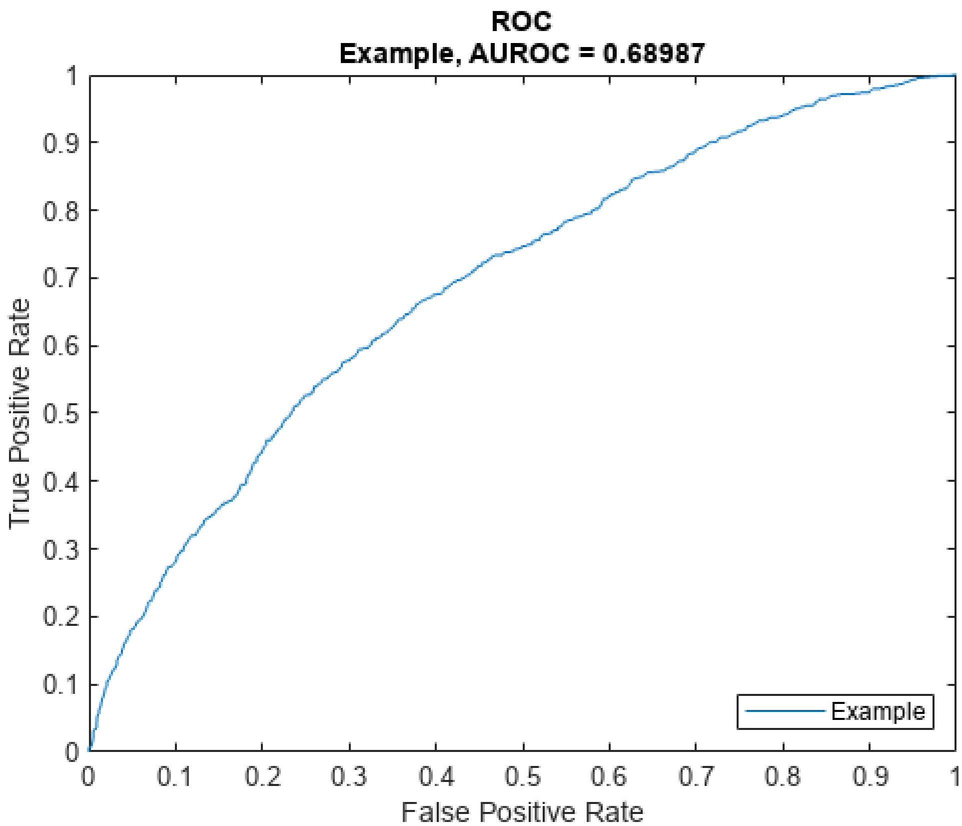


Validate LGD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

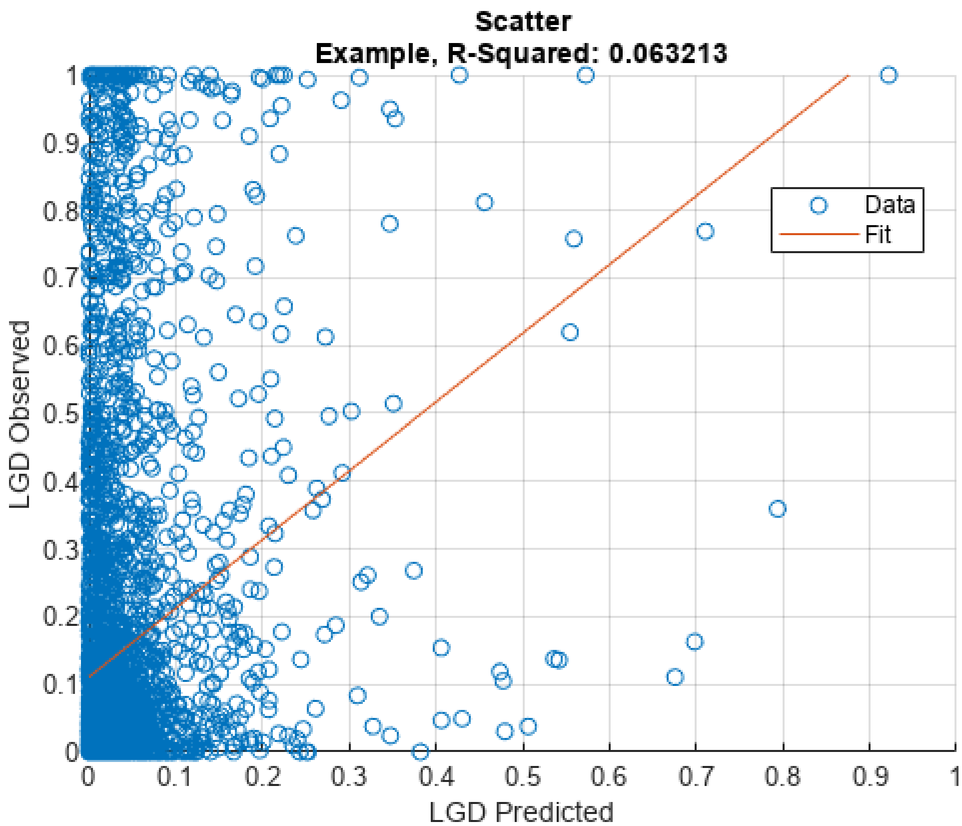
For example, use `modelDiscriminationPlot` to plot the ROC curve.

```
modelDiscriminationPlot(lgdModel,data)
```



Use `modelCalibrationPlot` to show a scatter plot of the predictions.

```
modelCalibrationPlot(lgdModel,data)
```



Input Arguments

data — Data for loss given default

table

Data for loss given default, specified as a table.

Data Types: `table`

`ModelType` — Type of LGD model

character vector with value `'Regression'`, `'Tobit'`, or `'Beta'` | string with value `"Regression"`, `"Tobit"`, or `"Beta"`

Type of LGD model, specified as a scalar string or character vector. Use one of following values:

- `Regression` —Transform the LGD response variable and fit a linear regression model. For more information, see "Loss Given Default Regression Models" on page 6-677.
- `Tobit` — Fit a Tobit regression model. For more information, see "Loss Given Default Tobit Models" on page 6-694.
- `Beta` — Fit a Beta regression model. For more information, see "Beta Regression Models" on page 6-685.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `lgdModel = fitLGDModel(data,'regression','PredictorVars',{'LTV' 'Age','Type'},'ResponseVar','LGD','ResponseTransform','probit','BoundaryTolerance',1e-6)`

The available name-value pair arguments depend on the value you specify for `ModelType`.

- `Regression` — See “Regression Name-Value Pair Arguments” on page 6-672.
- `Tobit` — See “Tobit Name-Value Pair Arguments” on page 6-688.
- `Beta` — See “Beta Name-Value Pair Arguments” on page 6-680.

Output Arguments

`lgdModel` — Loss given default model

`lgdModel` object

Loss given default model, returned as an `lgdModel` object. Supported classes are `Regression`, `Tobit`, and `Beta`.

Version History

Introduced in R2021a

R2022b: Support Beta regression for LGD model

Behavior changed in R2022b

To create a Beta model use a `ModelType` of `"Beta"`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

`Regression` | `Tobit` | `Beta`

Topics

“Model Loss Given Default” on page 4-90
“Basic Loss Given Default Model Validation” on page 4-131
“Compare Tobit LGD Model to Benchmark Model” on page 4-133
“Compare Loss Given Default Models Using Cross-Validation” on page 4-140

"Overview of Loss Given Default Models" on page 1-31

fitLifetimePDModel

Create specified lifetime PD model object type

Syntax

```
pdModel = fitLifetimePDModel(data,ModelType)
pdModel = fitLifetimePDModel( ___ ,Name,Value)
```

Description

`pdModel = fitLifetimePDModel(data,ModelType)` creates a lifetime probability of default (PD) model object specified by `data` and `ModelType`. `fitLifetimePDModel` takes in credit data in panel data form and fits a lifetime PD model. `ModelType` is supported for `Logistic`, `Probit`, or `Cox`.

`pdModel = fitLifetimePDModel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The available optional name-value pair arguments depend on the specified `ModelType`.

Examples

Create Logistic Lifetime PD Model

This example shows how to use `fitLifetimePDModel` to create a `Logistic` model using credit and macroeconomic data.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72       7.61
```

```
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create Logistic Lifetime PD Model

Use `fitLifetimePDModel` to create a `Logistic` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Logistic",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
disp(pdModel)
```

```
  Logistic with properties:

            ModelID: "Logistic"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
```

```
            AgeVar: "YOB"
          LoanVars: "ScoreGroup"
         MacroVars: ["GDP"    "Market"]
       ResponseVar: "Default"
```

Display the underlying model.

```
pdModel.UnderlyingModel
```

```
ans = 
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue   
                              __________    _________    _______    ___________

    (Intercept)                  -2.7422      0.10136    -27.054      3.408e-161
    ScoreGroup_Medium Risk      -0.68968     0.037286    -18.497      2.1894e-76
    ScoreGroup_Low Risk          -1.2587     0.045451    -27.693     8.4736e-169
    YOB                         -0.30894     0.013587    -22.738     1.8738e-114
    GDP                         -0.11111     0.039673    -2.8006       0.0051008
    Market                    -0.0083659    0.0028358    -2.9502       0.0031761


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Predict Conditional and Lifetime PD

Use the `predict` function to predict conditional PD values. The prediction is a row-by-row prediction.

```
dataCustomer1 = data(1:8,:);
CondPD = predict(pdModel,dataCustomer1)
```

```
CondPD = 8×1

    0.0092
    0.0053
    0.0045
    0.0039
    0.0037
    0.0037
    0.0019
    0.0012
```

Use `predictLifetime` to predict the lifetime cumulative PD values (computing marginal and survival PD values is also supported). The `predictLifetime` function uses the ID variable (see the `'IDVar'` property for the `Logistic` object) to transform conditional PDs to cumulative PDs for each ID.

```
LifetimePD = predictLifetime(pdModel,dataCustomer1)
```

```
LifetimePD = 8×1

    0.0092
```

```
    0.0145
    0.0189
    0.0228
    0.0264
    0.0300
    0.0319
    0.0330
```

Validate Model

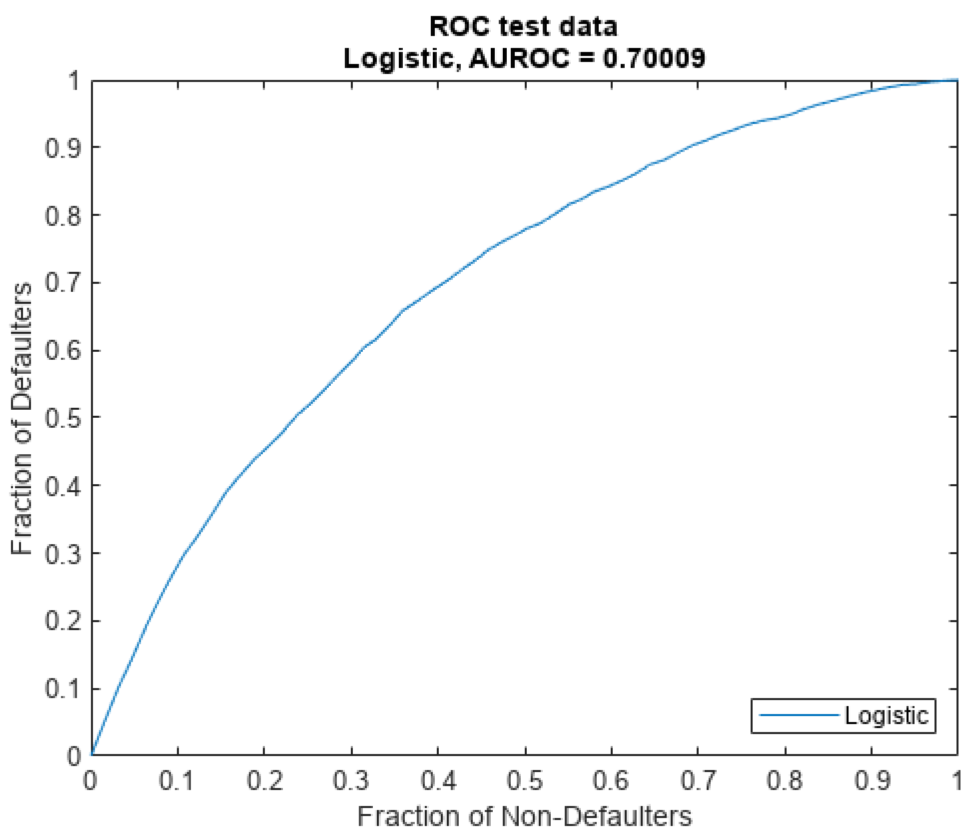
Use `modelDiscrimination` to measure the ranking of customers by PD.

```
DiscMeasure = modelDiscrimination(pdModel,data(TestDataInd,:),DataID='test data');
disp(DiscMeasure)
```

```
                            AUROC 
                           _______

    Logistic, test data    0.70009
```

Use `modelDiscriminationPlot` to visualize the ROC curve.

```
modelDiscriminationPlot(pdModel,data(TestDataInd,:),DataID='test data');
```



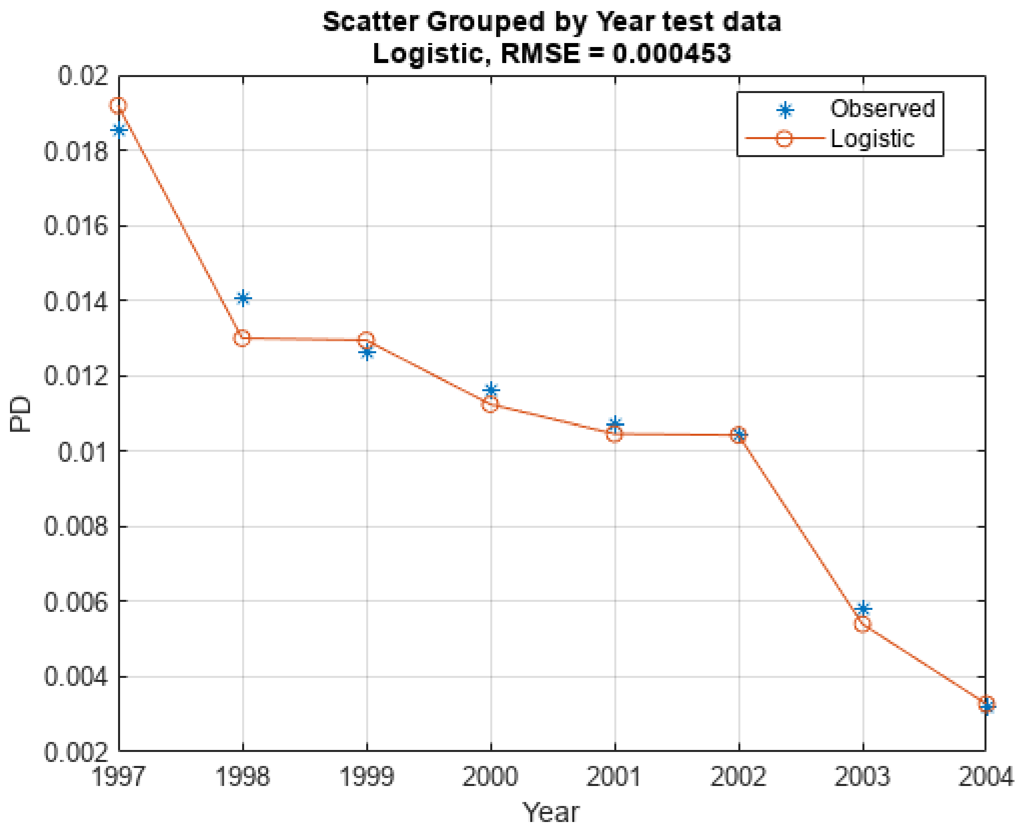
Use `modelCalibration` to measure the calibration of the predicted PD values. The `modelCalibration` function requires a grouping variable and compares the accuracy of the observed default rate in the group with the average predicted PD for the group. For example, you can group by calendar year using the 'Year' variable.

```
CalMeasure = modelCalibration(pdModel,data(TestDataInd,:),'Year',DataID='test data');
disp(CalMeasure)
```

```
                                          RMSE
                                        ________

    Logistic, grouped by Year, test data    0.000453
```

Use modelCalibrationPlot to visualize the observed default rates compared to the predicted probabilities of default (PD).

```
modelCalibrationPlot(pdModel,data(TestDataInd,:),'Year',DataID='test data');
```



Input Arguments

data — Data
table

Data, specified as a table, in panel data form. The data must contain an ID column. The response variable must be a binary variable with the value 0 or 1, with 1 indicating default.

Data Types: table

`ModelType` — Type of PD model
character vector with value `'Logistic'`, `'Probit'`, or `'Cox'` | string with value `"Logistic"`, `"Probit"`, or `"Cox"`

Type of PD model, specified as a scalar string or character vector. Use one of following values:

- `Logistic` —Fit a Logistic model for lifetime probability. For more information, see “Time Interval for Logistic Models” on page 6-623.
- `Probit` — Fit a Probit model for lifetime probability. For more information, see “Time Interval for Probit Models” on page 6-634.
- `Cox` — Fit a Cox model for lifetime probability. For more information, see “Cox Proportional Hazards Models” on page 6-550.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `pdModel = fitLifetimePDModel(data(TrainDataInd,:),ModelType,'AgeVar',"YOB",'IDVar',"ID",'LoanVars',"ScoreGroup",'MacroVars',{'GDP','Market'},'ResponseVar',"Default")`

The available name-value pair arguments depend on the value you specify for `ModelType`.

- `Logistic` — See “Logistic Name-Value Pair Arguments” on page 6-616.
- `Probit` — See “Probit Name-Value Pair Arguments” on page 6-627.
- `Cox` — See “Cox Name-Value Arguments” on page 6-537.

Output Arguments

`pdModel` — Probability of default model
`pdModel` object

Probability of default model, returned as a `pdModel` object. Supported classes are `Logistic`, `Probit`, or `Cox`.

Version History

Introduced in R2020b

References

Independently published

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`Logistic` | `Probit` | `Cox`

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

fullTriangle

Display full development triangle including ultimate claims

Syntax

```
fullTriangleTable = fullTriangle(developmentTriangle)
```

Description

`fullTriangleTable = fullTriangle(developmentTriangle)` calculates the projected claims for every origin and development period in the lower half of the development triangle.

Examples

Creates Filled Development Triangles

Calculate the projected claims for every origin and development period in the lower half of the development triangle for a `developmentTriangle` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.001
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.001
```

Use the `ultimateClaims` function to calculate CDFs and the percentage of total claims and return a table with the selected link ratios, CDFs, and percentage of total claims.

```
dT.SelectedLinkRatio = [1.1755, 1.0577, 1.0273, 1.0104, 1.0044, 1.0026, 1.0016, 1.0006, 1.0004];
selectedLinkRatiosTable = cdfSummary(dT)
```

```
selectedLinkRatiosTable=3×10 table
                                12-24      24-36      36-48      48-60      60-72      72-84
                               _______    _______    _______    _______    _______    _______

    Selected                    1.1755     1.0577     1.0273     1.0104     1.0044     1.0026
    CDF to Ultimate              1.303     1.1084      1.048     1.0201     1.0096     1.0052
    Percent of Total Claims    0.76747    0.90216    0.95422    0.98027    0.99046    0.99482
```

Use the `fullTriangle` function to create a table containing the filled development triangle.

```
fullTriangleTable = fullTriangle(dT)
```

```
fullTriangleTable=10×11 table
              12        24        36        48        60        72        84        96       108
            ______    ______    ______    ______    ______    ______    ______    ______    _____

    2010    3995.7      4635    4866.8    4964.1    5013.7    5038.8      5059    5074.1    5084
    2011      3968    4682.3    4963.2    5062.5    5113.1    5138.7    5154.1    5169.6    5179
    2012      4217    5060.4      5364    5508.9    5558.4    5586.2    5608.6    5625.4    5628
    2013    4374.2    5205.3    5517.7    5661.1    5740.4    5780.6    5803.7      5813    5816
    2014    4499.7    5309.6    5628.2    5785.8    5849.4    5878.7      5894    5903.4    5906
    2015    4530.2    5300.4    5565.4    5715.7    5772.8    5798.2    5813.3    5822.6    5826
    2016    4572.6    5304.2    5569.5    5714.3    5773.7    5799.1    5814.2    5823.5     582
    2017    4680.6    5523.1    5854.4    6014.3    6076.8    6103.6    6119.4    6129.2    6132
    2018    4696.7    5495.1    5812.2    5970.9    6032.9    6059.5    6075.2      6085    6088
    2019    4945.9    5813.9    6149.4    6317.2    6382.9      6411    6427.7      6438    6441
```

Input Arguments

`developmentTriangle` — Development triangle
`developmentTriangle` object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

Output Arguments

`fullTriangleTable` — Filled development triangle
table

Filled development triangle, returned as a table.

Version History

Introduced in R2020b

See Also

`view` | `linkRatios` | `linkRatioAverages` | `cdfSummary` | `ultimateClaims` | `linkRatiosPlot` | `claimsPlot`

Topics

“Mean Square Error of Prediction for Estimated Ultimate Claims” on page 4-161
“Bootstrap Using Chain Ladder Method” on page 4-168
“Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

ibnr

Compute IBNR claims for `bornhuetterFerguson` object

Syntax

```
ibnrClaims = ibnr(bf)
ibnrClaims = ibnr( ___ ,referenceClaimsType)
```

Description

`ibnrClaims = ibnr(bf)` computes incurred-but-not-reported (IBNR) claims for reported or paid claims for a `bornhuetterFerguson` object.

`ibnrClaims = ibnr( ___ ,referenceClaimsType)` additionally specifies the type of claims data. Specify this argument after the input argument in the previous syntax.

Examples

Compute IBNR Claims for bornhuetterFerguson Object

Compute IBNR for either reported or paid claims for a `bornhuetterFerguson` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
```

```
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

          ReportedTriangle: [1x1 developmentTriangle]
              PaidTriangle: [1x1 developmentTriangle]
             EarnedPremium: [10x1 double]
             InitialClaims: [10x1 double]
           CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Create a `bornhuetterFerguson` object with reported claims, paid claims, and expected claims to calculate ultimate claims, cases outstanding, IBNR claims, and unpaid claims estimates.

```
bf = bornhuetterFerguson(dT_reported, dT_paid, ec.ultimateClaims)
```

```
bf = 
  bornhuetterFerguson with properties:

      ReportedTriangle: [1x1 developmentTriangle]
          PaidTriangle: [1x1 developmentTriangle]
        ExpectedClaims: [10x1 double]
    PercentUnreported: [10x1 double]
         PercentUnpaid: [10x1 double]
       CaseOutstanding: [10x1 double]
```

Use `ibnr` to compute IBNR reported claims for a `bornhuetterFerguson` object.

```
ibnrClaims = ibnr(bf,"reported")
```

```
ibnrClaims = 10×1
10³ ×

         0
    0.0052
    0.0167
    0.0347
    0.0572
    0.0889
    0.1496
    0.3006
    0.6118
    1.5509
```

Input Arguments

bf — Bornhuetter-Ferguson

`bornhuetterFerguson` object

Bornhuetter-Ferguson object, specified as a previously created `bornhuetterFerguson` object.

Data Types: `object`

referenceClaimsType — Type of claims data

`'reported'` (default) | character vector with value `'reported'` or `'paid'` | string with value `"reported"` or `"paid"`

Type of claims data, specified as a character vector or a string.

Data Types: `char` | `string`

Output Arguments

ibnrClaims — IBNR claims

array

IBNR claims, returned as an array.

More About

IBNR

Incurred-but-not-reported (IBNR) claims are the claims amount owed by an insurer to all valid claimants who have had a covered loss but have not yet reported it.

Since the insurer knows neither how many of these losses have occurred nor the severity of each loss, IBNR is necessarily an estimate.

Version History

Introduced in R2020b

See Also

`ultimateClaims` | `unpaidClaims` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

ibnr

Compute IBNR claims for `capeCod` object

Syntax

```
ibnrClaims = ibnr(cc)
```

Description

`ibnrClaims = ibnr(cc)` computes incurred-but-not-reported (IBNR) claims for reported or paid claims for a `capeCod` object.

Examples

Compute IBNR Claims for capeCod Object

This example shows how to compute the incurred-but-not-reported (IBNR) claims for a `capeCod` object for simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create a `capeCod` object where the first input argument is the reported development triangle, the second input argument is the paid development triangle, and the third input is the earned premium.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
cc = capeCod(dT_reported, dT_paid,earnedPremium)
```

```
cc = 
  capeCod with properties:

           ReportedTriangle: [1x1 developmentTriangle]
               PaidTriangle: [1x1 developmentTriangle]
              EarnedPremium: [10x1 double]
              UsedUpPremium: [10x1 double]
       EstimatedClaimRatios: [10x1 double]
         ExpectedClaimRatio: 0.4258
    EstimatedExpectedClaims: [10x1 double]
         PercentUnreported: [10x1 double]
            CaseOutstanding: [10x1 double]
```

Use `ibnr` to compute the IBNR claims.

```
ibnrClaims = ibnr(cc)
```

```
ibnrClaims = 10×1

         0
    7.6650
   12.7454
   48.3382
   66.0055
   63.9011
  118.9799
  208.8065
  594.2093
  999.9805
```

Input Arguments

cc — Cape Cod object
`capeCod` object

Cape Cod object, specified as a previously created `capeCod` object.

Data Types: `object`

Output Arguments

`ibnrClaims` — IBNR claims
array

IBNR claims, returned as an array.

More About

IBNR

Incurred-but-not-reported (IBNR) claims are the claims amount owed by an insurer to all valid claimants who have had a covered loss but have not yet reported it.

Since the insurer knows neither how many of these losses have occurred nor the severity of each loss, IBNR is necessarily an estimate.

Version History

Introduced in R2021a

See Also

`unpaidClaims` | `ultimateClaims` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

ibnr

Compute IBNR claims for `chainLadder` object

Syntax

```
ibnrClaims = ibnr(cl)
ibnrClaims = ibnr( ___ ,referenceClaimsType)
```

Description

`ibnrClaims = ibnr(cl)` computes incurred-but-not-reported (IBNR) claims for a `chainLadder` object.

`ibnrClaims = ibnr( ___ ,referenceClaimsType)` additionally specifies the type of claims data. Specify this argument after the input argument in the previous syntax.

Examples

Calculate IBNR Claims for chainLadder

Calculate the IBNR claims for a `chainLadder` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                  Origin: {10x1 cell}
             Development: {10x1 cell}
                  Claims: [10x10 double]
          LatestDiagonal: [10x1 double]
             Description: ""
```

```
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create a `chainLadder` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
cl = chainLadder(dT_reported, dT_paid)
```

```
cl = 
  chainLadder with properties:

    ReportedTriangle: [1x1 developmentTriangle]
        PaidTriangle: [1x1 developmentTriangle]
     CaseOutstanding: [10x1 double]
```

Use `ibnr` to compute the IBNR claims.

```
ibnrClaims = ibnr(cl, 'reported')
```

```
ibnrClaims = 10×1
10^3 ×

         0
    0.0052
    0.0169
    0.0349
    0.0575
    0.0880
    0.1489
    0.3019
    0.6084
    1.5181
```

Input Arguments

cl — Chain Ladder
`chainLadder` object

Chain ladder, specified as a previously created `chainLadder` object.

Data Types: `object`

`referenceClaimsType` — Type of claims data
`'reported'` (default) | character vector with value `'reported'` or `'paid'` | string with value `"reported"` or `"paid"`

Type of claims data, specified as a character vector or string.

Data Types: `char` | `string`

Output Arguments

`ibnrClaims` — IBNR claims
array

IBNR claims, returned as an array.

.

More About

IBNR

Incurred-but-not-reported (IBNR) claims are the claims amount owed by an insurer to all valid claimants who have had a covered loss but have not yet reported it.

Since the insurer knows neither how many of these losses have occurred nor the severity of each loss, IBNR is necessarily an estimate.

Version History

Introduced in R2020b

See Also

`unpaidClaims` | `summary`

ibnr

Compute IBNR claims for `expectedClaims` object

Syntax

```
ibnrClaims = ibnr(ec)
```

Description

`ibnrClaims = ibnr(ec)` computes the incurred-but-not-reported (IBNR) claims for an `expectedClaims` object.

Examples

Compute IBNR Claims for expectedClaims Object

Compute the IBNR claims for an `expectedClaims` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.001
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.001
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                        Origin: {10x1 cell}
                   Development: {10x1 cell}
                        Claims: [10x10 double]
                LatestDiagonal: [10x1 double]
                   Description: ""
                    TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
             SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

         ReportedTriangle: [1x1 developmentTriangle]
             PaidTriangle: [1x1 developmentTriangle]
            EarnedPremium: [10x1 double]
            InitialClaims: [10x1 double]
          CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Use `ibnr` to compute the IBNR claims.

```
ibnrClaims = ibnr(ec)
```

```
ibnrClaims = 10×1
10^3 ×

   -0.0984
   -0.0176
   -0.0399
    0.0030
    0.0204
    0.1483
    0.1753
    0.2744
    0.6423
    1.6575
```

Input Arguments

ec — Expected claims
`expectedClaims` object

Expected claims, specified as a previously created `expectedClaims` object.

Data Types: `object`

Output Arguments

`ibnrClaims` — IBNR claims
array

IBNR claims, returned as an array.

More About

IBNR

Incurred-but-not-reported (IBNR) claims are the claims amount owed by an insurer to all valid claimants who have had a covered loss but have not yet reported it.

Since the insurer knows neither how many of these losses have occurred nor the severity of each loss, IBNR is necessarily an estimate.

Version History

Introduced in R2020b

See Also

`ultimateClaims` | `unpaidClaims` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

linkRatioAverages

Compute link ratio averages for `developmentTriangle` object

Syntax

`LinkRatioAveragesTable = linkRatioAverages(developmentTriangle)`

Description

`LinkRatioAveragesTable = linkRatioAverages(developmentTriangle)` calculates different link ratio averages.

Examples

Calculate Link Ratio Averages for a Development Triangle

Calculate different link ratio averages for a `developmentTriangle` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                         Origin: {10x1 cell}
                    Development: {10x1 cell}
                         Claims: [10x10 double]
                 LatestDiagonal: [10x1 double]
                    Description: ""
                     TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.001
              SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.001
```

Use the `linkRatioAverages` function to calculate different link ratio averages.

```
LinkRatioAveragesTable = linkRatioAverages(dT)
```

```
LinkRatioAveragesTable=8×9 table
                                            12-24     24-36     36-48     48-60     60-72     72-8
                                           ______    ______    ______    ______    ______    ____

    Simple Average                         1.1767    1.0563    1.0249    1.0107    1.0054    1.003
    Simple Average - Latest 5               1.172     1.056    1.0268    1.0108    1.0054    1.003
    Simple Average - Latest 3                1.17    1.0533     1.027    1.0117    1.0057    1.003
    Medial Average - Latest 5x1            1.1733    1.0567    1.0267    1.0103     1.005     1.00
    Volume-weighted Average                1.1766    1.0563     1.025    1.0107    1.0054    1.003
    Volume-weighted Average - Latest 5      1.172     1.056    1.0268    1.0108    1.0054    1.003
    Volume-weighted Average - Latest 3     1.1701    1.0534     1.027    1.0117    1.0057    1.003
    Geometric Average - Latest 4             1.17     1.055    1.0267     1.011    1.0055    1.003
```

Input Arguments

`developmentTriangle` — Development triangle
`developmentTriangle` object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

Output Arguments

`LinkRatioAveragesTable` — Link ratio averages
table

Link ratio averages, returned as a table. The table shows `Simple Average`, `Medial Average`, `Geometric Average`, and `Volume-weighted-average`.

More About

Link Ratio Averages

The link ratio average is the average of the link ratios or the age-to-age factors.

Version History

Introduced in R2020b

See Also

`view` | `linkRatios` | `cdfSummary` | `ultimateClaims` | `fullTriangle` | `linkRatiosPlot` | `claimsPlot`

Topics

"Mean Square Error of Prediction for Estimated Ultimate Claims" on page 4-161
"Bootstrap Using Chain Ladder Method" on page 4-168

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

linkRatios

Compute link ratios for `developmentTriangle` object

Syntax

`LinkRatiosTable = linkRatios(developmentTriangle)`

Description

`LinkRatiosTable = linkRatios(developmentTriangle)` calculates the link ratios between the current development period and the next for each origin period. You can plot the link ratios using `linkRatiosPlot`.

Examples

Calculate Link Ratios for Development Triangle

Calculate the link ratios (age-to-age factors) for a `developmentTriangle` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

Use the `linkRatios` function to calculate link ratios between the current development period and the next period.

```
LinkRatiosTable = linkRatios(dT)
```

```
LinkRatiosTable=10×9 table
            12-24    24-36    36-48    48-60    60-72    72-84    84-96    96-108    108-120
            _____    _____    _____    _____    _____    _____    _____    ______    _______

    2010    1.16     1.05      1.02     1.01    1.005    1.004    1.003    1.002      1.001
    2011    1.18     1.06      1.02     1.01    1.005    1.003    1.003    1.002        NaN
    2012     1.2     1.06     1.027    1.009    1.005    1.004    1.003      NaN        NaN
    2013    1.19     1.06     1.026    1.014    1.007    1.004      NaN      NaN        NaN
    2014    1.18     1.06     1.028    1.011    1.005      NaN      NaN      NaN        NaN
    2015    1.17     1.05     1.027     1.01      NaN      NaN      NaN      NaN        NaN
    2016    1.16     1.05     1.026      NaN      NaN      NaN      NaN      NaN        NaN
    2017    1.18     1.06       NaN      NaN      NaN      NaN      NaN      NaN        NaN
    2018    1.17      NaN       NaN      NaN      NaN      NaN      NaN      NaN        NaN
    2019     NaN      NaN       NaN      NaN      NaN      NaN      NaN      NaN        NaN
```

Input Arguments

`developmentTriangle` — Development triangle
`developmentTriangle` object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

Output Arguments

`LinkRatiosTable` — Link ratios
table

Link ratios, returned as a table.

More About

Link Ratios

Link ratios, also called age-to-age factors or loss development factors (LDFs), represent the ratio of loss amounts from one valuation date to another, and they are intended to capture growth patterns of losses over time.

Version History

Introduced in R2020b

See Also

`view` | `linkRatioAverages` | `cdfSummary` | `ultimateClaims` | `fullTriangle` | `linkRatiosPlot` | `claimsPlot`

Topics

"Mean Square Error of Prediction for Estimated Ultimate Claims" on page 4-161
"Bootstrap Using Chain Ladder Method" on page 4-168
"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

linkRatiosPlot

Plot link ratios for development triangle

Syntax

```
linkRatiosPlot(dT)
h = linkRatiosPlot(ax, ___ )
```

Description

`linkRatiosPlot(dT)` plots the link ratios for the development triangle.

`h = linkRatiosPlot(ax, ___ )` additionally specifies the axes and returns the figure handle `h`. Use this syntax with the required input argument in the previous syntax.

Examples

Generate Plot for Link Ratios

Generate a plot for the link ratios for a `developmentTriangle` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

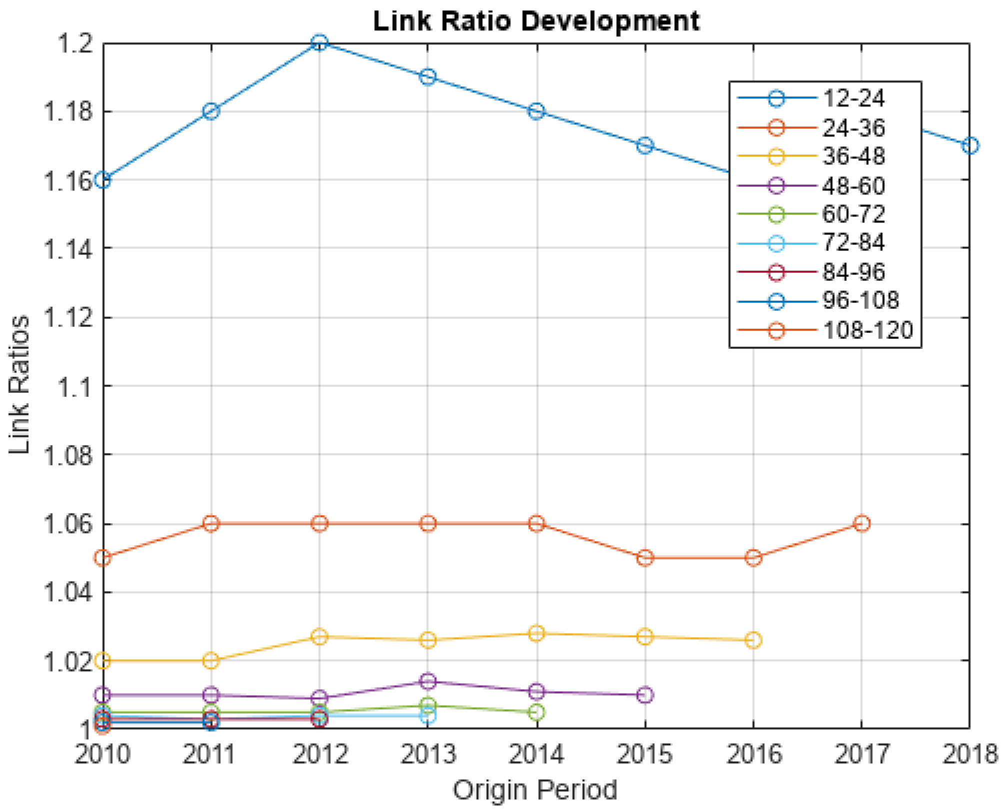
```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
```

```
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
              SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

Use the `linkRatiosPlot` function to plot a series of lines of the link ratios for the development triangle.

```
linkRatiosPlot(dT)
```



Input Arguments

dT — Development triangle
`developmentTriangle` object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

ax — Valid axis object
object

(Optional) Valid axis object, specified as an `ax` object created using `axes`. The function creates the plot on the axes specified by the optional `ax` argument instead of on the current axes (`gca`). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Output Arguments

h — Figure handle
handle object

Figure handle for the line objects, returned as a handle object.

Version History

Introduced in R2021a

See Also

`view` | `linkRatios` | `linkRatioAverages` | `cdfSummary` | `ultimateClaims` | `fullTriangle` | `claimsPlot`

Topics

“Mean Square Error of Prediction for Estimated Ultimate Claims” on page 4-161
“Bootstrap Using Chain Ladder Method” on page 4-168
“Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

mertonByTimeSeries

Estimate default probability using time-series version of Merton model

Syntax

```
[PD,DD,A,Sa] = mertonByTimeSeries(Equity,Liability,Rate)
[PD,DD,A,Sa] = mertonByTimeSeries( ___ ,Name,Value)
```

Description

`[PD,DD,A,Sa] = mertonByTimeSeries(Equity,Liability,Rate)` estimates the default probability of a firm by using the Merton model.

`[PD,DD,A,Sa] = mertonByTimeSeries( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

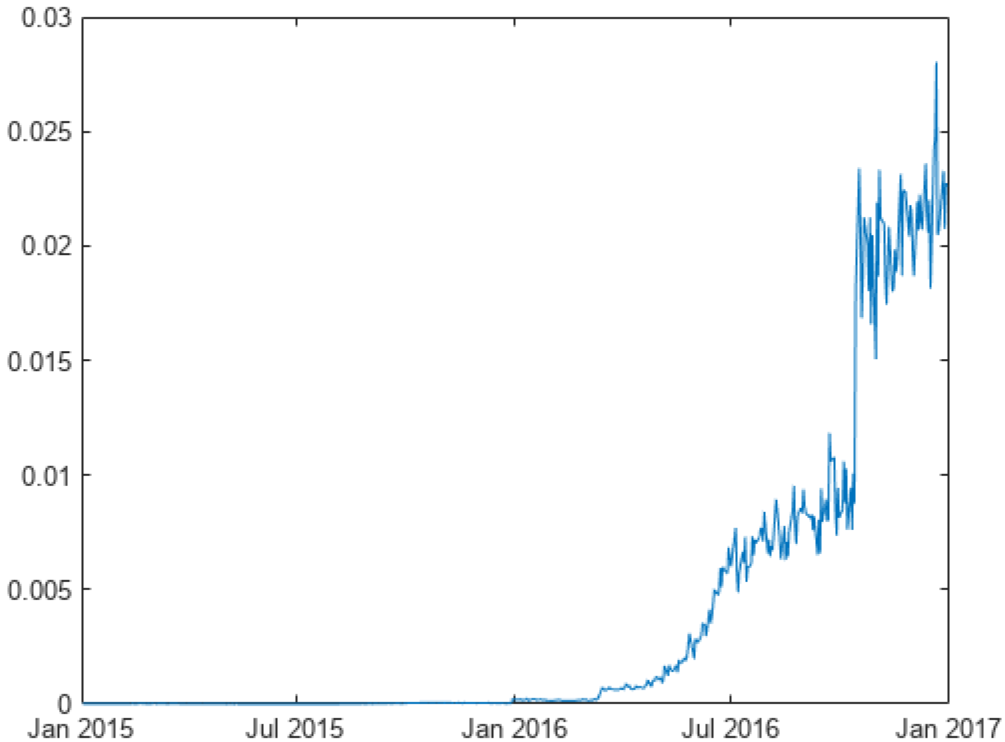
Compute Probability of Default Using the Time-Series Approach to the Merton Model

Load the data from `MertonData.mat`.

```
load MertonData.mat
Dates     = MertonDataTS.Dates;
Equity    = MertonDataTS.Equity;
Liability = MertonDataTS.Liability;
Rate      = MertonDataTS.Rate;
```

Compute the default probability by using the time-series approach of Merton's model.

```
[PD,DD,A,Sa] = mertonByTimeSeries(Equity,Liability,Rate);
plot(Dates,PD)
```



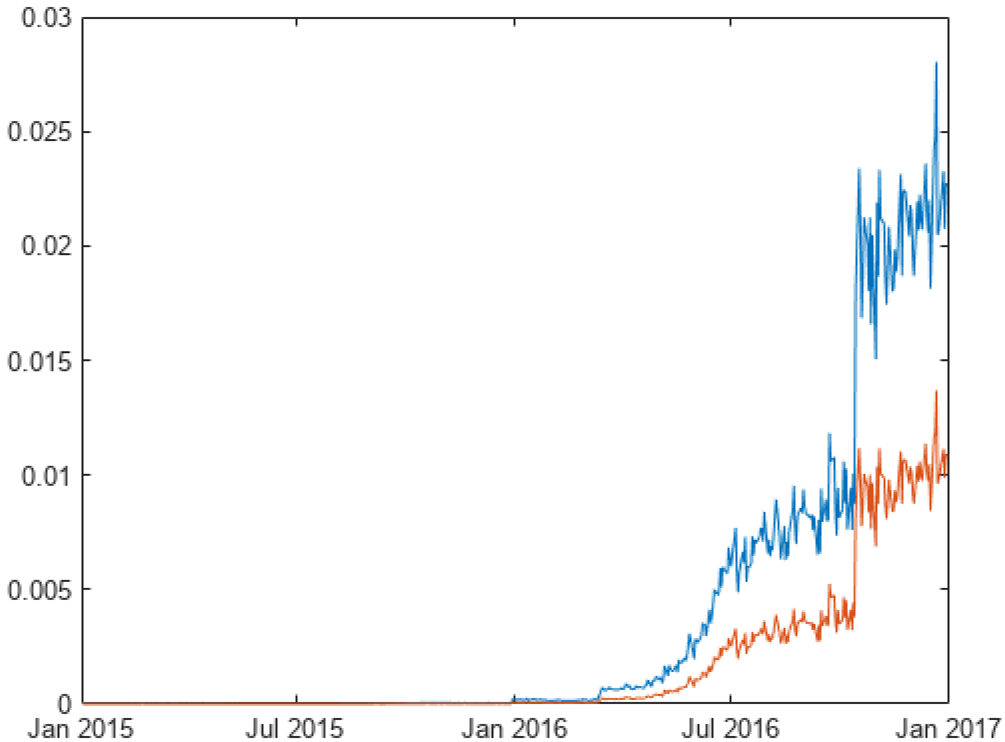
Compute Probability of Default Using the Time-Series Approach to the Merton Model With Drift

Load the data from `MertonData.mat`.

```
load MertonData.mat
Dates     = MertonDataTS.Dates;
Equity    = MertonDataTS.Equity;
Liability = MertonDataTS.Liability;
Rate      = MertonDataTS.Rate;
```

Compute the plot for the default probability values by using the time-series approach of Merton's model. You compute the `PD0` (blue line) by using the default values. You compute the `PD1` (red line) by specifying an optional `Drift` value.

```
PD0 = mertonByTimeSeries(Equity,Liability,Rate);
PD1 = mertonByTimeSeries(Equity,Liability,Rate,'Drift',0.10);
plot(Dates, PD0, Dates, PD1)
```



Input Arguments

`Equity` — Market value of firm's equity
positive numeric value

Market value of the firm's equity, specified as a positive value.

Data Types: `double`

`Liability` — Liability threshold of firm
positive numeric value

Liability threshold of the firm, specified as a positive value. The liability threshold is often referred to as the default point.

Data Types: `double`

`Rate` — Annualized risk-free interest rate
numeric value

Annualized risk-free interest rate, specified as a numeric value.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PD,DD,A,Sa] = mertonByTimeSeries(Equity,Liability,Rate,'Maturity',4,'Drift',0.22,'Tolerance',1e-5,'NumPeriods',12)`

`Maturity` — Time to maturity corresponding to liability threshold
`1` year (default) | positive numeric value

Time to maturity corresponding to the liability threshold, specified as the comma-separated pair consisting of `'Maturity'` and a positive value.

Data Types: `double`

`Drift` — Annualized drift rate
risk-free interest rate defined in `Rate` (default) | numeric value

Annualized drift rate, expected rate of return of the firm's assets, specified as the comma-separated pair consisting of `'Drift'` and a numeric value.

Data Types: `double`

`NumPeriods` — Number of periods per year
`250` trading days per year (default) | positive integer

Number of periods per year, specified as the comma-separated pair consisting of `'NumPeriods'` and a positive integer. Typical values are `250` (yearly), `12` (monthly), or `4` (quarterly).

Data Types: `double`

`Tolerance` — Tolerance for convergence of the solver
`1e-6` (default) | positive scalar

Tolerance for convergence of the solver, specified as the comma-separated pair consisting of `'Tolerance'` and a positive scalar value.

Data Types: `double`

`MaxIterations` — Maximum number of iterations allowed
`500` (default) | positive integer

Maximum number of iterations allowed, specified as the comma-separated pair consisting of `'MaxIterations'` and a positive integer.

Data Types: `double`

Output Arguments

`PD` — Probability of default of firm at maturity
numeric value

Probability of default of the firm at maturity, returned as a numeric.

DD — Distance-to-default
numeric value

Distance-to-default, defined as the number of standard deviations between the mean of the asset distribution at maturity and the liability threshold (default point), returned as a numeric.

A — Value of firm's assets
numeric value

Value of firm's assets, returned as a numeric value.

Sa — Annualized firm's asset volatility
numeric value

Annualized firm's asset volatility, returned as a numeric value.

More About

Merton Model for Time Series

In the Merton model, the value of a company's equity is treated as a call option on its assets, and the liability is taken as a strike price.

Given a time series of observed equity values and liability thresholds for a company, `mertonByTimeSeries` calibrates corresponding asset values, the volatility of the assets in the sample's time span, and computes the probability of default for each observation. Unlike `mertonmodel`, no equity volatility input is required for the time-series version of the Merton model. You compute the probability of default and distance-to-default by using the formulae in “Algorithms” on page 6-144.

Algorithms

Given the time series for equity (E), liability (L), risk-free interest rate (r), asset drift (μA), and maturity (T), `mertonByTimeSeries` sets up the following system of nonlinear equations and solves for a time series asset values (A), and a single asset volatility (σ_A). At each time period t, where t = `1`...n:

$$A_1 = \left(\frac{E_1 + L_1 e^{-r_1 T_1} N(d_2)}{N(d_1)} \right)$$

$$A_t = \left(\frac{E_t + L_t e^{-r_t T_t} N(d_2)}{N(d_1)} \right)$$

...

$$A_n = \left(\frac{E_n + L_n e^{-r_n T_n} N(d_2)}{N(d_1)} \right)$$

where N is the cumulative normal distribution. To simplify the notation, the time subscript is omitted for d_1 and d_2. At each time period, d_1, and d_2 are defined as:

$$d_1 = \frac{\ln\left(\frac{A}{L}\right) + (r + 0.5\sigma_A^2)T}{\sigma_A\sqrt{T}}$$

$$d_2 = d_1 - \sigma_A\sqrt{T}$$

The formulae for the distance-to-default (*DD*) and default probability (*PD*) at each time period are:

$$DD = \frac{\ln\left(\frac{A}{L}\right) + \left(\mu_A - 0.5\sigma_A^2\right)T}{\sigma_A\sqrt{T}}$$

$$PD = 1 - N(DD)$$

Version History

Introduced in R2017a

References

[1] Zielinski, T. *Merton's and KMV Models In Credit Risk Management.*

[2] Loeffler, G. and Posch, P.N. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2011.

[3] Kim, I.J., Byun, S.J, Hwang, S.Y. *An Iterative Method for Implementing Merton.*

[4] Merton, R. C. “On the Pricing of Corporate Debt: The Risk Structure of Interest Rates.” *Journal of Finance.* Vol. 29. pp. 449 – 470.

See Also

`mertonmodel` | `asrf`

Topics

“Comparison of the Merton Model Single-Point Approach to the Time-Series Approach” on page 4-54
“Default Probability by Using the Merton Model for Structural Credit Risk” on page 1-13

mertonmodel

Estimates probability of default using Merton model

Syntax

```
[PD,DD,A,Sa] = mertonmodel(Equity,EquityVol,Liability,Rate)
[PD,DD,A,Sa] = mertonmodel( ___ ,Name,Value)
```

Description

`[PD,DD,A,Sa] = mertonmodel(Equity,EquityVol,Liability,Rate)` estimates the default probability of a firm by using the Merton model.

`[PD,DD,A,Sa] = mertonmodel( ___ ,Name,Value)` adds optional name-value pair arguments.

Examples

Compute the Probability of Default Using the Single-Point Approach to the Merton Model

Load the data from `MertonData.mat`.

```
load MertonData.mat
Equity    = MertonData.Equity;
EquityVol = MertonData.EquityVol;
Liability = MertonData.Liability;
Drift     = MertonData.Drift;
Rate      = MertonData.Rate;
MertonData
```

```
MertonData=5×6 table
        ID            Equity       EquityVol    Liability    Rate    Drift 
    ___________    __________    _________    _________    ____    ______

    {'Firm 1'}     2.6406e+07     0.7103         4e+07     0.05    0.0306
    {'Firm 2'}     2.6817e+07     0.3929       3.5e+07     0.05      0.03
    {'Firm 3'}      3.977e+07     0.3121       3.5e+07     0.05     0.031
    {'Firm 4'}      2.947e+07     0.4595       3.2e+07     0.05    0.0302
    {'Firm 5'}      2.528e+07     0.6181         4e+07     0.05    0.0305
```

Compute the default probability using the single-point approach to the Merton model.

```
[PD,DD,A,Sa] = mertonmodel(Equity,EquityVol,Liability,Rate,'Drift',Drift)
```

```
PD = 5×1

    0.0638
    0.0008
    0.0000
    0.0026
    0.0344
```

```
DD = 5×1

    1.5237
    3.1679
    4.4298
    2.7916
    1.8196


A = 5×1
10^7 ×

    6.4210
    6.0109
    7.3063
    5.9906
    6.3231


Sa = 5×1

    0.3010
    0.1753
    0.1699
    0.2263
    0.2511
```

Input Arguments

`Equity` — Current market value of firm's equity
positive numeric value

Current market value of firm's equity, specified as a positive value.

Data Types: `double`

`EquityVol` — Volatility of firm's equity
positive numeric value

Volatility of the firm's equity, specified as a positive annualized standard deviation.

Data Types: `double`

`Liability` — Liability threshold of firm
positive numeric value

Liability threshold of firm, specified as a positive value. The liability threshold is often referred to as the default point.

Data Types: `double`

`Rate` — Annualized risk-free interest rate
numeric value

Annualized risk-free interest rate, specified as a numeric value.

Data Types: double

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [PD,DD,A,Sa] = mertonmodel(Equity,EquityVol,Liability,Rate,'Maturity',4,'Drift',0.22)

Maturity — Time to maturity corresponding to liability threshold
1 year (default) | positive numeric value

Time to maturity corresponding to the liability threshold, specified as the comma-separated pair consisting of 'Maturity' and a positive value.

Data Types: double

Drift — Annualized drift rate
risk-free interest rate defined in Rate (default) | numeric value

Annualized drift rate (expected rate of return of the firm's assets), specified as the comma-separated pair consisting of 'Drift' and a numeric value.

Data Types: double

Tolerance — Tolerance for convergence of the solver
1e-6 (default) | positive scalar

Tolerance for convergence of the solver, specified as the comma-separated pair consisting of 'Tolerance' and a positive scalar value.

Data Types: double

MaxIterations — Maximum number of iterations allowed
500 (default) | positive integer

Maximum number of iterations allowed, specified as the comma-separated pair consisting of 'MaxIterations' and a positive integer.

Data Types: double

Output Arguments

PD — Probability of default of firm at maturity
numeric value

Probability of default of the firm at maturity, returned as a numeric value.

DD — Distance-to-default
numeric value

Distance-to-default, defined as the number of standard deviations between the mean of the asset distribution at maturity and the liability threshold (default point), returned as a numeric value.

A — Current value of firm's assets
numeric value

Current value of firm's assets, returned as a numeric value.

Sa — Annualized firm's asset volatility
numeric value

Annualized firm's asset volatility, returned as a numeric value.

More About

Merton Model Using Single-Point Calibration

In the Merton model, the value of a company's equity is treated as a call option on its assets and the liability is taken as a strike price.

`mertonmodel` accepts inputs for the firm's equity, equity volatility, liability threshold, and risk-free interest rate. The `mertonmodel` function solves a 2-by-2 nonlinear system of equations whose unknowns are the firm's assets and asset volatility. You compute the probability of default and distance-to-default by using the formulae in “Algorithms” on page 6-149.

Algorithms

Unlike the time series method (see `mertonByTimeSeries`), when using `mertonmodel`, the equity volatility (σ_E) is provided. Given equity (E), liability (L), risk-free interest rate (r), asset drift (μ_A), and maturity (T), you use a 2-by-2 nonlinear system of equations. `mertonmodel` solves for the asset value (A) and asset volatility (σ_A) as follows:

$$E = AN(d_1) - Le^{-rT}N(d_2)$$

$$\sigma_E = \frac{A}{E}N(d_1)\sigma_A$$

where N is the cumulative normal distribution, d_1 and d_2 are defined as:

$$d_1 = \frac{\ln\left(\frac{A}{L}\right) + (r + 0.5\sigma_A^2)T}{\sigma_A\sqrt{T}}$$

$$d_2 = d_1 - \sigma_A\sqrt{T}$$

The formulae for the distance-to-default (DD) and default probability (PD) are:

$$DD = \frac{\ln\left(\frac{A}{L}\right) + \left(\mu_A - 0.5\sigma_A^2\right)T}{\sigma_A\sqrt{T}}$$

$$PD = 1 - N(DD)$$

Version History

Introduced in R2017a

References

[1] Zielinski, T. *Merton's and KMV Models In Credit Risk Management.*

[2] Löffler, G. and Posch, P.N. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2011.

[3] Kim, I.J., Byun, S.J, Hwang, S.Y. *An Iterative Method for Implementing Merton.*

[4] Merton, R. C. "On the Pricing of Corporate Debt: The Risk Structure of Interest Rates." *Journal of Finance.* Vol. 29. pp. 449 – 470.

See Also

`mertonByTimeSeries` | `asrf`

Topics

"Comparison of the Merton Model Single-Point Approach to the Time-Series Approach" on page 4-54
"Default Probability by Using the Merton Model for Structural Credit Risk" on page 1-13

minBiasAbsolute

Minimally biased absolute test for Expected Shortfall (ES) backtest by Acerbi-Szekely

Syntax

```
TestResults = minBiasAbsolute(ebts)
[TestResults,SimTestStatistic] = minBiasAbsolute(ebts,Name,Value)
```

Description

`TestResults = minBiasAbsolute(ebts)` runs the absolute version of the minimally biased Expected Shortfall (ES) backtest by Acerbi-Szekely (2017) using the `esbacktestbysim` object.

`[TestResults,SimTestStatistic] = minBiasAbsolute(ebts,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Run minBiasAbsolute ES Backtest

Create an `esbacktestbysim` object.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

Generate the `TestResults` and `SimTestStatistic` reports for the `minBiasAbsolute` ES backtest.

```
[TestResults,SimTestStatistic] = minBiasAbsolute(ebts)
```

```
TestResults=3×10 table
    PortfolioID        VaRID         VaRLevel    MinBiasAbsolute    PValue    TestStatistic    Cr
    ___________    _____________    ________    _______________    ______    _____________    __

       "S&P"       "t(10) 95%"        0.95          accept          0.062      -0.0014247       -(
       "S&P"       "t(10) 97.5%"     0.975          reject          0.029      -0.0026674       -(
       "S&P"       "t(10) 99%"        0.99          reject          0.005      -0.0060982       -(


SimTestStatistic = 3×1000

    0.0023    0.0008   -0.0018    0.0004    0.0009    0.0003   -0.0003    0.0008   -0.0001    0.(
    0.0036    0.0005   -0.0032    0.0009    0.0017    0.0002   -0.0003    0.0011   -0.0001   -0.(
```

```
    0.0052   -0.0008   -0.0048    0.0014    0.0027    0.0007    0.0005    0.0007    0.0001   -0.0
```

Input Arguments

ebts — esbacktestbysim object
object

esbacktestbysim (ebts) object, which contains a copy of the given data (the PortfolioData, VarData, ESData, and Distribution properties) and all combinations of portfolio IDs, VaR IDs, and VaR levels to be tested. For more information on creating an esbacktestbysim object, see esbacktestbysim.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: TestResults = minBiasAbsolute(ebts)

TestLevel — Test confidence level
0.95 (default) | numeric value between 0 and 1

Test confidence level, specified as the comma-separated pair consisting of 'TestLevel' and a numeric value between 0 and 1.

Data Types: double

Output Arguments

TestResults — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio IDs, VaR IDs, and VaR levels to be tested. The columns correspond to the following information:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR data columns provided
- 'VaRLevel' — VaR level for the corresponding VaR data column
- 'MinBiasAbsolute' — Categorical array with categories'accept' and 'reject' that indicate the result of the minBiasAbsolute test
- 'PValue'— *p*-value for the minBiasAbsolute test
- 'TestStatistic'— minBiasAbsolute test statistic
- 'CriticalValue'— Critical value for minBiasAbsolute test
- 'Observations'— Number of observations
- 'Scenarios' — Number of scenarios simulated to obtain *p*-values
- 'TestLevel' — Test confidence level

Note For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

`SimTestStatistic` — Simulated values of test statistic
numeric array

Simulated values of the test statistic, returned as a `NumVaRs`-by-`NumScenarios` numeric array.

More About

Minimally Biased Test, Absolute Version by Acerbi and Szekely

The absolute version of the Acerbi-Szekely test [1] computes the `TestStatistic` in the units of data.

The absolute version of the minimally biased test statistic is given by

$$Z_{minbias}^{abs} = \frac{1}{N}\sum_{t=1}^{N}(ES_t - VaR_t - \frac{1}{p_{VaR}}(X_t + VaR_t)\_)$$

where

X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period t.

VaR_t is the essential VaR for period t.

ES_t is the expected shortfall for period t.

p_{VaR} is the probability of VaR failure, defined as 1 - VaR level.

N is the number of periods in the test window (t = 1,...N).

(x)_ is the negative part function defined as $(x)\_ = max(0,-x)$.

Significance of the Test

Negative values of the test statistic indicate risk underestimation.

The minimally biased test is a one-sided test that rejects the model when there is evidence that the model underestimates risk (for technical details, see Acerbi-Szekely [1] and [2]). The test rejects the model when the *p*-value is less than `1` minus the test confidence level. For more information on the steps to simulate the test statistics and details on the computation of the *p*-values and critical values, see `simulate`.

ES backtests are necessarily approximated in that they are sensitive to errors in the predicted VaR. However, the minimally biased test has only a small sensitivity to VaR errors and the sensitivity is prudential, in the sense that VaR errors lead to a more punitive ES test. For details, see Acerbi-Szekely ([1] and [2]). When distribution information is available using the minimally biased test is recommended.

Version History

Introduced in R2020b

References

[1] Acerbi, Carlo, and Balazs Szekely. "General Properties of Backtestable Statistics." *SSRN Electronic Journal.* (January, 2017).

[2] Acerbi, Carlo, and Balazs Szekely. "The Minimally Biased Backtest for ES." *Risk.* (September, 2019).

[3] Acerbi, C. and D. Tasche. "On the Coherence of Expected Shortfall." *Journal of Banking and Finance.* Vol. 26, 2002, pp. 1487-1503.

[4] Rockafellar, R. T. and S. Uryasev. "Conditional Value-at-Risk for General Loss Distributions." *Journal of Banking and Finance.* Vol. 26, 2002, pp. 1443-1471.

See Also

`summary` | `conditional` | `unconditional` | `quantile` | `simulate` | `minBiasRelative` | `esbacktestbysim` | `esbacktestbyde`

Topics

"Expected Shortfall (ES) Backtesting Workflow Using Simulation" on page 2-34
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

minBiasRelative

Minimally biased relative test for Expected Shortfall (ES) backtest by Acerbi-Szekely

Syntax

```
TestResults = minBiasRelative(ebts)
[TestResults,SimTestStatistic] = minBiasRelative(ebts,Name,Value)
```

Description

`TestResults = minBiasRelative(ebts)` runs the relative version of the minimally biased Expected Shortfall (ES) back test by Acerbi-Szekely (2017) using the `esbacktestbysim` object.

`[TestResults,SimTestStatistic] = minBiasRelative(ebts,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Run minBiasRelative ES Backtest

Create an `esbacktestbysim` object.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

Generate the `TestResults` and the `SimTestStatistic` reports for the `minBiasRelative` ES backtest.

```
[TestResults,SimTestStatistic] = minBiasRelative(ebts)
```

```
TestResults=3×10 table
    PortfolioID        VaRID         VaRLevel     MinBiasRelative     PValue     TestStatistic     Cr
    ___________    _____________    ________    _______________    ______    _____________    __

       "S&P"       "t(10) 95%"        0.95          reject         0.003       -0.10509
       "S&P"       "t(10) 97.5%"     0.975          reject             0       -0.15603
       "S&P"       "t(10) 99%"        0.99          reject             0       -0.26716


SimTestStatistic = 3×1000

    0.0860    0.0284   -0.0480    0.0176    0.0262    0.0309   -0.0107    0.0361   -0.0171   -0.0
    0.1145    0.0177   -0.0741    0.0357    0.0505    0.0275   -0.0136    0.0421   -0.0190   -0.0
```

```
    0.1435   -0.0195   -0.0915    0.0478    0.0796    0.0397   -0.0022    0.0282   -0.0055   -0.0
```

Input Arguments

ebts — esbacktestbysim object
object

esbacktestbysim (ebts) object, which contains a copy of the given data (the PortfolioData, VarData, ESData, and Distribution properties) and all combinations of portfolio IDs, VaR IDs, and VaR levels to be tested. For more information on creating an esbacktestbysim object, see esbacktestbysim.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: minBiasRelative(ebts,'TestLevel',0.99)

TestLevel — Test confidence level
0.95 (default) | numeric value between 0 and 1

Test confidence level, specified as the comma-separated pair consisting of 'TestLevel' and a numeric value between 0 and 1.

Data Types: double

Output Arguments

TestResults — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio IDs, VaR IDs, and VaR levels to be tested. The columns correspond to the following information:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR data columns provided
- 'VaRLevel' — VaR level for the corresponding VaR data column
- 'MinBiasRelative' — Categorical array with categories'accept' and 'reject' that indicate the result of the minBiasRelative test
- 'PValue'— *p*-value for the minBiasRelative test
- 'TestStatistic'— minBiasRelative test statistic
- 'CriticalValue'— Critical value for minBiasRelative test
- 'Observations'— Number of observations
- 'Scenarios' — Number of scenarios simulated to obtain *p*-values
- 'TestLevel' — Test confidence level

Note For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

`SimTestStatistic` — Simulated values of test statistic
numeric array

Simulated values of the test statistic, returned as a `NumVaRs`-by-`NumScenarios` numeric array.

More About

Minimally Biased Test, Relative Version by Acerbi and Szekely

The relative version of the Acerbi-Szekely test ([1]) computes the `TestStatistic` in the units of data.

The absolute version of the minimally biased test statistic is given by

$$Z_{minbias}^{rel} = \frac{1}{N}\sum_{t=1}^{N}\frac{1}{ES_t}(ES_t - VaR_t - \frac{1}{p_{VaR}}(X_t + VaR_t)\_)$$

where

X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period t.

VaR_t is the essential VaR for period t.

ES_t is the expected shortfall for period t.

p_{VaR} is the probability of VaR failure, defined as 1 - VaR level.

N is the number of periods in the test window (t = 1,...N).

(x)_ is the negative part function defined as $(x)\_ = max(0,-x)$.

Significance of the Test

Negative values of the test statistic indicate risk underestimation.

The minimally biased test is a one-sided test that rejects the model when there is evidence that the model underestimates risk (for technical details, see Acerbi-Szekely [1] and [2]). The test rejects the model when the *p*-value is less than `1` minus the test confidence level. For more information on the steps to simulate the test statistics and details on the computation of the *p*-values and critical values, see `simulate`.

ES backtests are necessarily approximated in that they are sensitive to errors in the predicted VaR. However, the minimally biased test has only a small sensitivity to VaR errors and the sensitivity is prudential, in the sense that VaR errors lead to a more punitive ES test. For details, see Acerbi-Szekely ([1] and [2]). When distribution information is available using the minimally biased test is recommended.

Version History

Introduced in R2020b

References

[1] Acerbi, Carlo, and Balazs Szekely. "General Properties of Backtestable Statistics." *SSRN Electronic Journal.* (January, 2017).

[2] Acerbi, Carlo, and Balazs Szekely. "The Minimally Biased Backtest for ES." *Risk.* (September, 2019).

[3] Acerbi, C. and D. Tasche. "On the Coherence of Expected Shortfall." *Journal of Banking and Finance.* Vol. 26, 2002, pp. 1487-1503.

[4] Rockafellar, R. T. and S. Uryasev. "Conditional Value-at-Risk for General Loss Distributions." *Journal of Banking and Finance.* Vol. 26, 2002, pp. 1443-1471.

See Also

`summary` | `conditional` | `unconditional` | `quantile` | `simulate` | `esbacktestbysim` | `esbacktestbyde`

Topics

"Expected Shortfall (ES) Backtesting Workflow Using Simulation" on page 2-34
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

modelAccuracy

Compute R-square, RMSE, correlation, and sample mean error of predicted and observed EADs

Note `modelAccuracy` is renamed to `modelCalibration`. `modelAccuracy` is not recommended. Use `modelCalibration` instead.

Syntax

```
AccMeasure = modelAccuracy(eadModel,data)
[AccMeasure,AccData] = modelAccuracy( ___ ,Name=Value)
```

Description

`AccMeasure = modelAccuracy(eadModel,data)` computes the R-square, root mean square error (RMSE), correlation, and sample mean error of observed vs. predicted exposure at default (EAD) data. `modelAccuracy` supports comparison against a reference model and also supports different correlation types. By default, `modelAccuracy` computes the metrics in the EAD scale. You can use the `ModelLevel` name-value argument to compute metrics using the underlying model's transformed scale.

`[AccMeasure,AccData] = modelAccuracy( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax.

Input Arguments

`eadModel` — Exposure at default model
`Regression` object | `Tobit` object | `Beta` object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitEADModel`.

Data Types: `object`

`data` — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[AccMeasure,AccData] = modelAccuracy(eadModel,data(TestInd,:),DataID='Testing',CorrelationType='spearman')`

CorrelationType — Correlation type
"pearson" (default) | character vector with value of 'pearson', 'spearman', or 'kendall' | string with value of "pearson", "spearman", or "kendall"

Correlation type, specified as CorrelationType and a character vector or string.

Data Types: char | string

DataID — Data set identifier
"" (default) | character vector | string

Data set identifier, specified as DataID and a character vector or string. The DataID is included in the output for reporting purposes.

Data Types: char | string

ModelLevel — Model level
'ead' (default) | character vector with value 'ead', 'conversionMeasure', or 'conversionTransform' | string with value "ead", "conversionMeasure", or "conversionTransform"

Model level, specified as ModelLevel and a character vector or string.

Note Regression models support all three model levels, but a Tobit or Beta model supports model levels only for "ead" and "conversionMeasure".

Data Types: char | string

ReferenceEAD — EAD values predicted for data by reference model
[] (default) | numeric vector

EAD values predicted for data by the reference model, specified as ReferenceEAD and a NumRows-by-1 numeric vector. The modelAccuracy output information is reported for both the eadModel object and the reference model.

Data Types: double

ReferenceID — Identifier for the reference model
'Reference' (default) | character vector | string

Identifier for the reference model, specified as ReferenceID and a character vector or string. ReferenceID is used in the modelAccuracy output for reporting purposes.

Data Types: char | string

Output Arguments

AccMeasure — Accuracy measure
table

Accuracy measure, returned as a table with columns 'RSquared', 'RMSE', 'Correlation', and 'SampleMeanError'. AccMeasure has one row if only the eadModel accuracy is measured and it has two rows if reference model information is given. The row names of AccMeasure report the model ID and data ID (if provided).

`AccData` — Accuracy data
table

Accuracy data, returned as a table with observed EAD values, predicted EAD values, and residuals (observed minus predicted). Additional columns for predicted and residual values are included for the reference model, if provided. The `ModelID` and `ReferenceID` labels are appended in the column names.

More About

Model Accuracy

Model accuracy measures the accuracy of the predicted probability of EAD values using different metrics.

- R-squared — To compute the R-squared metric, `modelAccuracy` fits a linear regression of the observed EAD values against the predicted EAD values:

  $$EAD_{obs} = a + b * EAD_{pred} + \varepsilon$$

  The R-square of this regression is reported. For more information, see “Coefficient of Determination (R-Squared)”.
- RMSE — To compute the root mean square error (RMSE), `modelAccuracy` uses the following formula where N is the number of observations:

  $$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(EAD_i^{obs} - EAD_i^{pred}\right)^2}$$
- Correlation — This metric is the correlation between the observed and predicted EAD:

  $$corr(EAD_{obs}, EAD_{pred})$$

  For more information and details about the different correlation types, see `corr`.
- Sample mean error — This metric is the difference between the mean observed EAD and the mean predicted EAD or, equivalently, the mean of the residuals:

  $$SampleMeanError = \frac{1}{N}\sum_{i=1}^{N}\left(EAD_i^{obs} - EAD_i^{pred}\right)$$

Version History

Introduced in R2021b

R2023a: `modelAccuracy` function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` function is renamed to `modelCalibration` function. The use of `modelAccuracy` is not recommended, use `modelCalibration` instead.

R2022b: Support for `Beta` model

Behavior changed in R2022b

The `eadModel` input supports an option for a `Beta` model object that you can create using `fitEADModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

`Regression` | `Tobit` | `Beta` | `fitEADModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelAccuracyPlot`

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Exposure at Default Models" on page 1-34

modelAccuracy

Compute R-square, RMSE, correlation, and sample mean error of predicted and observed LGDs

Note modelAccuracy is renamed to modelCalibration. modelAccuracy is not recommended. Use modelCalibration instead.

Syntax

```
AccMeasure = modelAccuracy(lgdModel,data)
[AccMeasure,AccData] = modelAccuracy( ___ ,Name,Value)
```

Description

AccMeasure = modelAccuracy(lgdModel,data) computes the R-square, root mean square error (RMSE), correlation, and sample mean error of observed vs. predicted loss given default (LGD) data. modelAccuracy supports comparison against a reference model and also supports different correlation types. By default, modelAccuracy computes the metrics in the LGD scale. You can use the ModelLevel name-value pair argument to compute metrics using the underlying model's transformed scale.

[AccMeasure,AccData] = modelAccuracy(___ ,Name,Value) specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Input Arguments

lgdModel — Loss given default model
Regression object | Tobit object | Beta object

Loss given default model, specified as a previously created Regression, Tobit, or Beta object using fitLGDModel.

Data Types: object

data — Data
table

Data, specified as a NumRows-by-NumCols table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: table

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: `[AccMeasure,AccData] = modelAccuracy(lgdModel,data(TestInd,:),'DataID','Testing','CorrelationType','spearman')`

`CorrelationType` — Correlation type
`"pearson"` (default) | character vector with value of `'pearson'`, `'spearman'`, or `'kendall'` | string with value of `"pearson"`, `"spearman"`, or `"kendall'"`

Correlation type, specified as the comma-separated pair consisting of `'CorrelationType'` and a character vector or string.

Data Types: `char` | `string`

`DataID` — Data set identifier
`""` (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of `'DataID'` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`ModelLevel` — Model level
`'top'` (default) | character vector with value `'top'` or `'underlying'` | string with value `"top"` or `"underlying"`

Model level, specified as the comma-separated pair consisting of `'ModelLevel'` and a character vector or string.

- `'top'` — The accuracy metrics are computed in the LGD scale at the top model level.
- `'underlying'` — For a `Regression` model only, the metrics are computed in the underlying model's transformed scale. The metrics are computed on the transformed LGD data.

Note `ModelLevel` has no effect for a `Tobit` or `Beta` model because there is no response transformation.

Data Types: `char` | `string`

`ReferenceLGD` — LGD values predicted for `data` by reference model
`[ ]` (default) | numeric vector

LGD values predicted for `data` by the reference model, specified as the comma-separated pair consisting of `'ReferenceLGD'` and a `NumRows`-by-`1` numeric vector. The `modelAccuracy` output information is reported for both the `lgdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of `'ReferenceID'` and a character vector or string. `'ReferenceID'` is used in the `modelAccuracy` output for reporting purposes.

Data Types: `char` | `string`

Output Arguments

`AccMeasure` — Accuracy measure

table

Accuracy measure, returned as a table with columns `'RSquared'`, `'RMSE'`, `'Correlation'`, and `'SampleMeanError'`. `AccMeasure` has one row if only the `lgdModel` accuracy is measured and it has two rows if reference model information is given. The row names of `AccMeasure` report the model ID and data ID (if provided).

`AccData` — Accuracy data

table

Accuracy data, returned as a table with observed LGD values, predicted LGD values, and residuals (observed minus predicted). Additional columns for predicted and residual values are included for the reference model, if provided. The `ModelID` and `ReferenceID` labels are appended in the column names.

More About

Model Accuracy

Model accuracy measures the accuracy of the predicted probability of LGD values using different metrics.

- R-squared — To compute the R-squared metric, `modelAccuracy` fits a linear regression of the observed LGD values against the predicted LGD values

  $$LGD_{obs} = a + b * LGD_{pred} + \varepsilon$$

  The R-square of this regression is reported. For more information, see "Coefficient of Determination (R-Squared)".
- RMSE — To compute the root mean square error (RMSE), `modelAccuracy` uses the following formula where N is the number of observations:

  $$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(LGD_i^{obs} - LGD_i^{pred})^2}$$
- Correlation — This is the correlation between the observed and predicted LGD:

  $$corr(LGD_{obs}, LGD_{pred})$$

  For more information and details about the different correlation types, see `corr`.
- Sample mean error — This is the difference between the mean observed LGD and the mean predicted LGD or, equivalently, the mean of the residuals:

  $$SampleMeanError = \frac{1}{N}\sum_{i=1}^{N}(LGD_i^{obs} - LGD_i^{pred})$$

Version History

Introduced in R2021a

R2023a: `modelAccuracy` function is renamed to `modelCalibration` function
Not recommended starting in R2023a

The `modelAccuracy` function is renamed to `modelCalibration` function. The use of `modelAccuracy` is not recommended, use `modelCalibration` instead.

R2022b: Support for `Beta` model
Behavior changed in R2022b

The `lgdModel` input supports an option for a `Beta` model object that you can create using `fitLGDModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

`Tobit` | `Regression` | `Beta` | `modelAccuracyPlot` | `modelDiscriminationPlot` | `modelDiscrimination` | `predict` | `fitLGDModel`

Topics

"Model Loss Given Default" on page 4-90
"Basic Loss Given Default Model Validation" on page 4-131
"Compare Tobit LGD Model to Benchmark Model" on page 4-133
"Compare Loss Given Default Models Using Cross-Validation" on page 4-140
"Overview of Loss Given Default Models" on page 1-31

modelAccuracy

Compute RMSE of predicted and observed PDs on grouped data

Note `modelAccuracy` is renamed to `modelCalibration`. `modelAccuracy` is not recommended. Use `modelCalibration` instead.

Syntax

```
AccMeasure = modelAccuracy(pdModel,data,GroupBy)
[AccMeasure,AccData] = modelAccuracy( ___ ,Name,Value)
```

Description

`AccMeasure = modelAccuracy(pdModel,data,GroupBy)` computes the root mean squared error (RMSE) of the observed compared to the predicted probabilities of default (PD). `GroupBy` is required and can be any column in the `data` input (not necessarily a model variable). The `modelAccuracy` function computes the observed PD as the default rate of each group and the predicted PD as the average PD for each group. `modelAccuracy` supports comparison against a reference model.

`[AccMeasure,AccData] = modelAccuracy( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Input Arguments

`pdModel` — Probability of default model

`Logistic` object | `Probit` object | `Cox` object | `customLifetimePDModel` object

Probability of default model, specified as a previously created `Logistic`, `Probit`, or `Cox` object using `fitLifetimePDModel`. Alternatively, you can create a custom probability of default model using `customLifetimePDModel`.

Data Types: `object`

`data` — Data

table

Data, specified as a `NumRows`-by-`NumCols` table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model.

Data Types: `table`

`GroupBy` — Name of column in `data` input used to group the data

string | character vector

Name of column in the `data` input used to group the data, specified as a string or character vector. `GroupBy` does not have to be a model variable name. For each group designated by `GroupBy`, the `modelAccuracy` function computes the observed default rates and average predicted PDs are computed to measure the RMSE.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[AccMeasure,AccData] = modelAccuracy(pdModel,data(Ind,:),'GroupBy',["YOB","ScoreGroup"],'DataID',"DataSetChoice")`

`DataID` — Data set identifier

`""` (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of `'DataID'` and a character vector or string. `DataID` is included in the `modelAccuracy` output for reporting purposes.

Data Types: `char` | `string`

`ReferencePD` — Conditional PD values predicted for `data` by reference model

`[]` (default) | numeric vector

Conditional PD values predicted for `data` by the reference model, specified as the comma-separated pair consisting of `'ReferencePD'` and a `NumRows`-by-`1` numeric vector. The functions reports the `modelAccuracy` output information for both the `pdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for reference model

`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of `'ReferenceID'` and a character vector or string. `ReferenceID` is used in the `modelAccuracy` output for reporting purposes.

Data Types: `char` | `string`

Output Arguments

`AccMeasure` — RMSE values

table

Accuracy measure, returned as a table.

RMSE values, returned as a single-column `'RMSE'` table. The table has one row if only the `pdModel` accuracy is measured and it has two rows if reference model information is given. The row names of `AccMeasure` report the model IDs, grouping variables, and data ID.

Note The reported RMSE values depend on the grouping variable for the required `GroupBy` argument.

`AccData` — Observed and predicted PD values for each group

table

Accuracy data, returned as a table.

Observed and predicted PD values for each group, returned as a table. The reported observed PD values correspond to the observed default rate for each group. The reported predicted PD values are the average PD values predicted by the `pdModel` object for each group, and similarly for the reference model. The `modelAccuracy` function stacks the PD data, placing the observed values for all groups first, then the predicted PDs for the `pdModel`, and then the predicted PDs for the reference model, if given.

The column `'ModelID'` identifies which rows correspond to the observed PD, `pdModel`, or reference model. The table also has one column for each grouping variable showing the unique combinations of grouping values. The `'PD'` column of `AccData` is a the PD data. The last column of `AccData` is a `'GroupCount'` column with the group counts data.

More About

Model Accuracy

Model accuracy measures the accuracy of the predicted probability of default (PD) values.

To measure model accuracy, also called model calibration, you must compare the predicted PD values to the observed default rates. For example, if a group of customers is predicted to have an average PD of 5%, then is the observed default rate for that group close to 5%?

The `modelAccuracy` function requires a grouping variable to compute average predicted PD values within each group and the average observed default rate also within each group. `modelAccuracy` uses the root mean squared error (RMSE) to measure the deviations between the observed and predicted values across groups. For example, the grouping variable could be the calendar year, so that rows corresponding to the same calendar year are grouped together. Then, for each year the software computes the observed default rate and the average predicted PD. The `modelAccuracy` function then applies the RMSE formula to obtain a single measure of the prediction error across all years in the sample.

Suppose there are N observations in the data set, and there are M groups $G_1,\ldots,G_M$. The default rate for group G_i is

$$DR_i = \frac{D_i}{N_i}$$

where:

D_i is the number of defaults observed in group G_i.

N_i is the number of observations in group G_i.

The average predicted probability of default PD_i for group G_i is

$$PD_i = \frac{1}{N_i}\sum_{j \in G_i} PD(j)$$

where $PD(j)$ is the probability of default for observation j. In other words, this is the average of the predicted PDs within group G_i.

Therefore, the RMSE is computed as

$$RMSE = \sqrt{\sum_{i=1}^{M} \left(\frac{N_i}{N}\right)(DR_i - PD_i)^2}$$

The RMSE, as defined, depends on the selected grouping variable. For example, grouping by calendar year and grouping by years-on-books might result in different RSME values.

Use `modelAccuracyPlot` to visualize observed default rates and predicted PD values on grouped data.

Version History

Introduced in R2020b

R2023a: `modelAccuracy` function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` function is renamed to `modelCalibration` function. The use of `modelAccuracy` is not recommended, use `modelCalibration` instead.

R2022b: Support for `customLifetimePDModel` model

The `pdModel` input supports an option for a `customLifetimePDModel` model object that you can create using `customLifetimePDModel`.

R2022a: Additional column for `AccData` for `GroupCount`

There is an additional column for `AccData` for `GroupCount` for PD models.

R2022a: `GroupCount` column automatically included in `AccData` outputs

Behavior changed in R2022a

Starting in R2022a, the `AccData` output of `modelAccuracy` contains an additional column for `GroupCount` with the group counts data.

If you extract the end column from the `AccData` output using `AccData{:,end}`, the `end` column is different than previous releases of `modelAccuracy`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`modelDiscrimination` | `modelDiscriminationPlot` | `modelAccuracyPlot` | `predictLifetime` | `predict` | `fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

modelAccuracyPlot

Scatter plot of predicted and observed EADs

Note modelAccuracyPlot is renamed to modelCalibrationPlot. modelAccuracyPlot is not recommended. Use modelCalibrationPlot instead.

Syntax

```
modelAccuracyPlot(eadModel,data)
modelAccuracyPlot( ___ ,Name=Value)
h = modelAccuracyPlot(ax, ___ ,Name=Value)
```

Description

modelAccuracyPlot(eadModel,data) returns a scatter plot of observed vs. predicted exposure at default (EAD) data with a linear fit. modelAccuracyPlot supports comparison against a reference model. By default, modelAccuracyPlot plots in the EAD scale.

modelAccuracyPlot(___ ,Name=Value) specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax. You can use the ModelLevel name-value argument to compute metrics using the underlying model's transformed scale.

h = modelAccuracyPlot(ax, ___ ,Name=Value) specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax and returns the figure handle h.

Input Arguments

eadModel — Exposure at default model
Regression object | Tobit | Beta object

Exposure at default model, specified as a previously created Regression, Tobit, or Beta object using fitEADModel.

Data Types: object

data — Data
table

Data, specified as a NumRows-by-NumCols table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: table

ax — Valid axis object
object

(Optional) Valid axis object, specified as an ax object that is created using axes. The plot will be created in the axes specified by the optional ax argument instead of in the current axes (gca). The optional argument ax must precede any of the input argument combinations.

Data Types: `object`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:
`modelAccuracyPlot(eadModel,data(TestInd,:),DataID=Testing,XData='residuals',YData='residuals')`

`DataID` — Data set identifier

`""` (default) | character vector | string

Data set identifier, specified `DataID` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`ModelLevel` — Model level

`'ead'` (default) | character vector with value `'ead'`, `'conversionMeasure'`, or `'conversionTransform'` | string with value `"ead"`, `"conversionMeasure"`, or `"conversionTransform"`

Model level, specified as `ModelLevel` and a character vector or string.

Note `Regression` models support all three model levels, but a `Tobit` or `Beta` model supports model levels only for `"ead"` and `"conversionMeasure"`.

Data Types: `char` | `string`

`ReferenceEAD` — EAD values predicted for `data` by reference model

`[]` (default) | numeric vector

EAD values predicted for `data` by the reference model, specified as `ReferenceEAD` and a `NumRows`-by-`1` numeric vector. The scatter plot output is plotted for both the `eadModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model

`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as `ReferenceID` and a character vector or string. `ReferenceID` is used in the scatter plot output for reporting purposes.

Data Types: `char` | `string`

`XData` — Data to plot on *x*-axis

`'predicted'` (default) | character vector with value `'predicted'`, `'observed'`, `'residuals'`, or *VariableName* | string with value | `"predicted"`, `"observed"`, `"residuals"`, or *VariableName*

Data to plot on *x*-axis, specified as `XData` and a character vector or string for one of the following:

- `'predicted'` — Plot the predicted EAD values in the *x*-axis.
- `'observed'` — Plot the observed EAD values in the *x*-axis.
- `'residuals'` — Plot the residuals in the *x*-axis.
- *VariableName* — Use the name of the variable in the `data` input, not necessarily a model variable, to plot in the *x*-axis.

Data Types: `char` | `string`

`YData` — Data to plot on *y*-axis
`'predicted'` (default) | character vector with value `'predicted'`, `'observed'`, or `'residuals'` | string with value | `"predicted"`, `"observed"`, or `"residuals"`

Data to plot on *y*-axis, specified as `YData` and a character vector or string for one of the following:

- `'predicted'` — Plot the predicted EAD values in the *y*-axis.
- `'observed'` — Plot the observed EAD values in the *y*-axis.
- `'residuals'` — Plot the residuals in the *y*-axis.

Data Types: `char` | `string`

Output Arguments

h — Figure handle
handle object

Figure handle for the scatter and line objects, returned as handle object.

More About

Model Accuracy Plot

The `modelAccuracyPlot` function returns a scatter plot of observed vs. predicted loss given default (EAD) data with a linear fit and reports the R-square of the linear fit.

The `XData` name-value pair argument allows you to change the *x* values on the plot. By default, predicted EAD values are plotted in the *x*-axis, but predicted EAD values, residuals, or any variable in the `data` input, not necessarily a model variable, can be used as *x* values. If the selected `XData` is a categorical variable, a swarm chart is used. For more information, see `swarmchart`.

The `YData` name-value pair argument allows users to change the *y* values on the plot. By default, observed EAD values are plotted in the *y*-axis, but predicted EAD values or residuals can also be used as *y* values. `YData` does not support table variables.

The linear fit and reported R-squared value always correspond to the linear regression model with the plotted *y* values as response and the plotted *x* values as the only predictor.

Version History

Introduced in R2021b

R2023a: `modelAccuracyPlot` function is renamed to `modelCalibrationPlot` function
Not recommended starting in R2023a

The `modelAccuracyPlot` function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is not recommended, use `modelCalibrationPlot` instead.

R2022b: Support for Beta model

Behavior changed in R2022b

The `eadModel` input supports an option for a `Beta` model object that you can create using `fitEADModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

`Regression` | `Tobit` | `Beta` | `fitEADModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelAccuracy`

Topics

“Compare Results for Regression and Tobit EAD Models” on page 4-151
“Overview of Exposure at Default Models” on page 1-34

modelAccuracyPlot

Scatter plot of predicted and observed LGDs

Note `modelAccuracyPlot` is renamed to `modelCalibrationPlot`. `modelAccuracyPlot` is not recommended. Use `modelCalibrationPlot` instead.

Syntax

```
modelAccuracyPlot(lgdModel,data)
modelAccuracyPlot( ___ ,Name,Value)
h = modelAccuracyPlot(ax, ___ ,Name,Value)
```

Description

`modelAccuracyPlot(lgdModel,data)` returns a scatter plot of observed vs. predicted loss given default (LGD) data with a linear fit. `modelAccuracyPlot` supports comparison against a reference model. By default, `modelAccuracyPlot` plots in the LGD scale.

`modelAccuracyPlot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. You can use the `ModelLevel` name-value pair argument to compute metrics using the underlying model's transformed scale.

`h = modelAccuracyPlot(ax, ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Input Arguments

`lgdModel` — Loss given default model
`Regression` object | `Tobit` object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitLGDModel`.

Data Types: `object`

`data` — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

ax — Valid axis object
object

(Optional) Valid axis object, specified as an `ax` object that is created using `axes`. The plot will be created in the axes specified by the optional `ax` argument instead of in the current axes (gca). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`modelAccuracyPlot(lgdModel,data(TestInd,:),'DataID','Testing','YData','residuals','XData','LTV')`

`DataID` — Data set identifier

`""` (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of `'DataID'` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`ModelLevel` — Model level

`'top'` (default) | character vector with value `'top'` or `'underlying'` | string with value `"top"` or `"underlying"`

Model level, specified as the comma-separated pair consisting of `'ModelLevel'` and a character vector or string.

- `'top'` — The accuracy metrics are computed in the LGD scale at the top model level.
- `'underlying'` — For a `Regression` model only, the metrics are computed in the underlying model's transformed scale. The metrics are computed on the transformed LGD data.

Note `ModelLevel` has no effect for a `Tobit` or `Beta` model because there is no response transformation.

Data Types: `char` | `string`

`ReferenceLGD` — LGD values predicted for `data` by reference model

`[ ]` (default) | numeric vector

LGD values predicted for `data` by the reference model, specified as the comma-separated pair consisting of `'ReferenceLGD'` and a `NumRows`-by-`1` numeric vector. The scatter plot output is plotted for both the `lgdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model

`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of `'ReferenceID'` and a character vector or string. `'ReferenceID'` is used in the scatter plot output for reporting purposes.

Data Types: `char` | `string`

XData — Data to plot on x-axis
'predicted' (default) | character vector with value 'predicted', 'observed', 'residuals', or *VariableName* | string with value | "predicted", "observed", "residuals", or *VariableName*

Data to plot on x-axis, specified as the comma-separated pair consisting of 'XData' and a character vector or string for one of the following:

- 'predicted' — Plot the predicted LGD values in the x-axis.
- 'observed' — Plot the observed LGD values in the x-axis.
- 'residuals' — Plot the residuals in the x-axis.
- *VariableName* — Use the name of the variable in the data input, not necessarily a model variable, to plot in the x-axis.

Data Types: char | string

YData — Data to plot on y-axis
'predicted' (default) | character vector with value 'predicted', 'observed', or 'residuals' | string with value | "predicted", "observed", or "residuals"

Data to plot on y-axis, specified as the comma-separated pair consisting of 'YData' and a character vector or string for one of the following:

- 'predicted' — Plot the predicted LGD values in the y-axis.
- 'observed' — Plot the observed LGD values in the y-axis.
- 'residuals' — Plot the residuals in the y-axis.

Data Types: char | string

Output Arguments

h — Figure handle
handle object

Figure handle for the scatter and line objects, returned as handle object.

More About

Model Accuracy Plot

The modelAccuracyPlot function returns a scatter plot of observed vs. predicted loss given default (LGD) data with a linear fit and reports the R-square of the linear fit.

The XData name-value pair argument allows you to change the *x* values on the plot. By default, predicted LGD values are plotted in the *x*-axis, but predicted LGD values, residuals, or any variable in the data input, not necessarily a model variable, can be used as *x* values. If the selected XData is a categorical variable, a swarm chart is used. For more information, see swarmchart.

The YData name-value pair argument allows users to change the *y* values on the plot. By default, observed LGD values are plotted in the *y*-axis, but predicted LGD values or residuals can also be used as *y* values. YData does not support table variables.

For `Regression` models, if `ModelLevel` is set to `'underlying'`, the LGD data is transformed into the underlying model's scale. The transformed data is shown on the plot. The `ModelLevel` name-value pair argument has no effect for `Tobit` models.

The linear fit and reported R-squared value always correspond to the linear regression model with the plotted *y* values as response and the plotted *x* values as the only predictor.

Version History

Introduced in R2021a

R2023a: `modelAccuracyPlot` function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is not recommended, use `modelCalibrationPlot` instead.

R2022b: Support for `Beta` model

Behavior changed in R2022b

The `lgdModel` input supports an option for a `Beta` model object that you can create using `fitLGDModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

`Tobit` | `Regression` | `Beta` | `modelAccuracy` | `modelDiscriminationPlot` | `modelDiscrimination` | `predict` | `fitLGDModel`

Topics

"Model Loss Given Default" on page 4-90
"Basic Loss Given Default Model Validation" on page 4-131
"Compare Tobit LGD Model to Benchmark Model" on page 4-133
"Compare Loss Given Default Models Using Cross-Validation" on page 4-140
"Overview of Loss Given Default Models" on page 1-31

modelCalibration

Compute R-square, RMSE, correlation, and sample mean error of predicted and observed EADs

Syntax

```
CalMeasure = modelCalibration(eadModel,data)
[CalMeasure,CalData] = modelCalibration( ___ ,Name=Value)
```

Description

`CalMeasure = modelCalibration(eadModel,data)` computes the R-square, root mean square error (RMSE), correlation, and sample mean error of observed vs. predicted exposure at default (EAD) data. `modelCalibration` supports comparison against a reference model and also supports different correlation types. By default, `modelCalibration` computes the metrics in the EAD scale. You can use the `ModelLevel` name-value argument to compute metrics using the underlying model's transformed scale.

`[CalMeasure,CalData] = modelCalibration( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax.

Examples

Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed Using a Tobit EAD Model

This example shows how to use `fitEADModel` to create a `Tobit` model and then use `modelCalibration` to compute the R-Square, RMSE, correlation, and sample mean error of predicted and observed EAD.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
```

```
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Tobit` or `Regression`.

```
ModelType = Tobit ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Tobit EAD Model

Use `fitEADModel` to create a `Tobit` model using `EADData`.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','M
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Tobit with properties:

        CensoringSide: "both"
            LeftLimit: 0
           RightLimit: 1
              ModelID: "Tobit"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrwanVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Tobit regression model:
     EAD_lcf = max(0,min(Y*,1))
     Y* ~ 1 + UtilizationRate + Age + Marriage

Estimated coefficients:
                             Estimate        SE         tStat       pValue  
                            __________    _________    ________    __________

    (Intercept)                0.22467      0.03134      7.1689    9.7855e-13
    UtilizationRate             0.4714     0.020722      22.749             0
    Age                     -0.0014209   0.00076326     -1.8616      0.062771
    Marriage_not married     -0.010542      0.01578    -0.66807       0.50415
    (Sigma)                     0.3618    0.0050022       72.33             0
```

```
Number of observations: 2627
Number of left-censored observations: 0
Number of uncensored observations: 2626
Number of right-censored observations: 1
Log-likelihood: -1057.9
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

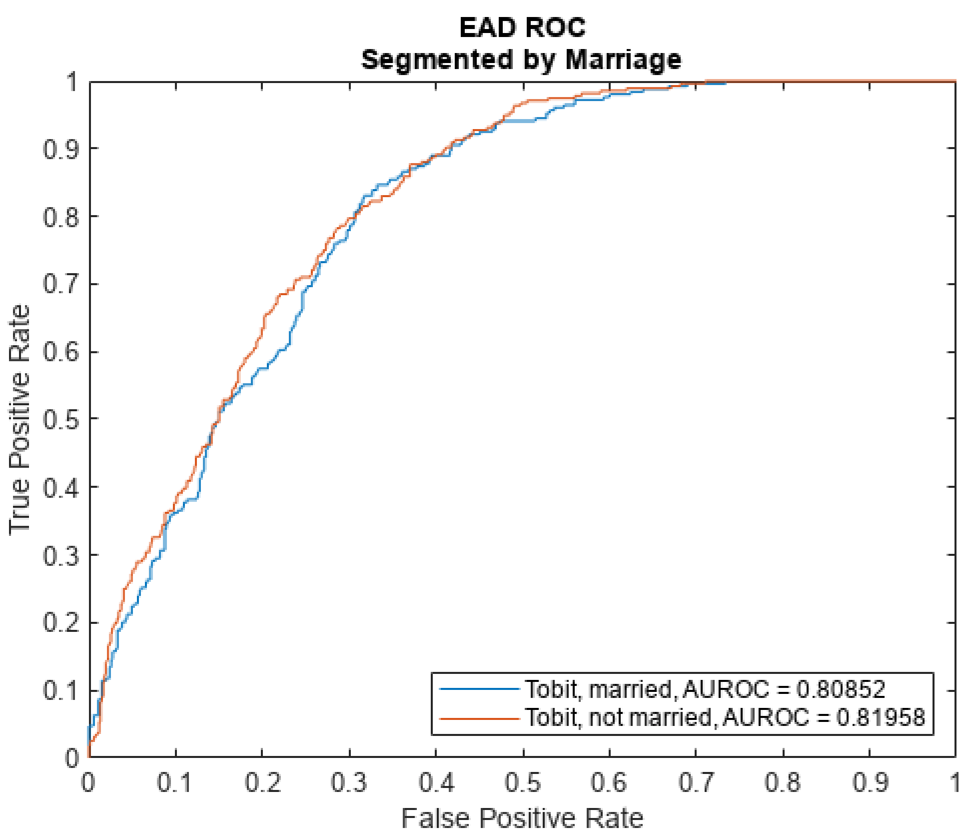
Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ead ;
```

```
[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel
modelDiscriminationPlot(eadModel,EADData(TestInd, :),ModelLevel=ModelLevel,SegmentBy="Marriage")
```



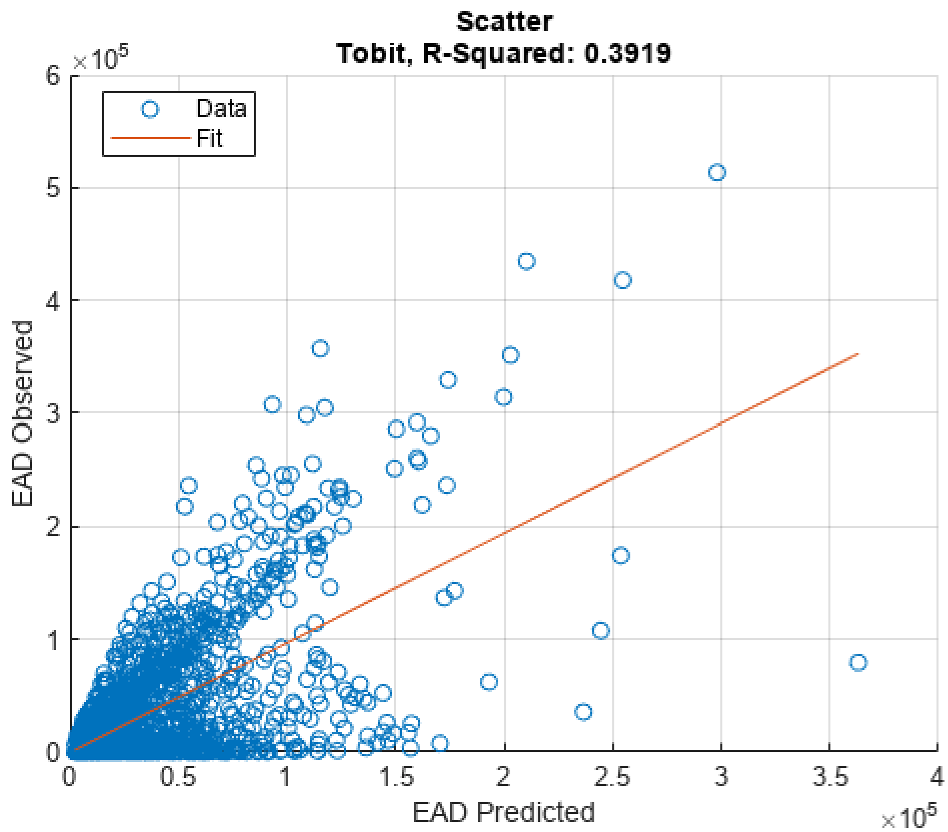
Use `modelCalibration`, and `modelCalibrationPlot` to show a scatter plot of the predictions.

```
YData = Observed ;

[CalMeasure1,CalData1] = modelCalibration(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel)
```

```
CalMeasure1=1×4 table
             RSquared    RMSE     Correlation    SampleMeanError
             ________    _____    ___________    _______________

    Tobit     0.3919     42494      0.62602          -1240.7
```

```
CalData1=1751×3 table
     Observed     Predicted_Tobit    Residuals_Tobit
    __________    _______________    _______________

         44740         14893               29847
        54.175        8730.2               -8676
        987.39         13244              -12257
        9606.4        7367.5              2238.9
        83.809         27501              -27417
         73538         45726               27812
        96.949        5522.5             -5425.5
        873.21        4426.3             -3553.1
        328.35        5952.4             -5624.1
         55237         28040               27198
         30359         19047               11312
         39211         28368               10843
    2.0885e+05    1.0539e+05          1.0346e+05
        1921.7         19939              -18017
         15230        5427.4              9802.5
         20063        9359.6               10703
          ⋮
```

```
modelCalibrationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed Using a Beta EAD Model

This example shows how to use `fitEADModel` to create a `Beta` model and then use `modelCalibration` to compute the R-Square, RMSE, correlation, and sample mean error of predicted and observed EAD.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Beta`.

```
ModelType = Beta ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Beta EAD Model

Use `fitEADModel` to create a `Beta` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-07
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrwanVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Beta regression model:
     logit(EAD_lcf) ~ 1_mu + UtilizationRate_mu + Age_mu + Marriage_mu
     log(EAD_lcf) ~ 1_phi + UtilizationRate_phi + Age_phi + Marriage_phi

Estimated coefficients:
                                 Estimate        SE        tStat       pValue  
                                _________    ________    ________    __________

    (Intercept)_mu               -0.65566     0.11484     -5.7093    1.2616e-08
    UtilizationRate_mu             1.7014    0.078094      21.787             0
    Age_mu                     -0.0055901   0.0027603     -2.0252      0.042949
    Marriage_not married_mu     -0.012577    0.052098    -0.24141       0.80926
    (Intercept)_phi              -0.50131    0.094625     -5.2979    1.2686e-07
```

```
    UtilizationRate_phi           0.39731     0.066707       5.956    2.9303e-09
    Age_phi                      -0.001167    0.0023161    -0.50387       0.6144
    Marriage_not married_phi     -0.013275    0.042627     -0.31143       0.7555

Number of observations: 2627
Log-likelihood: -3140.21
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

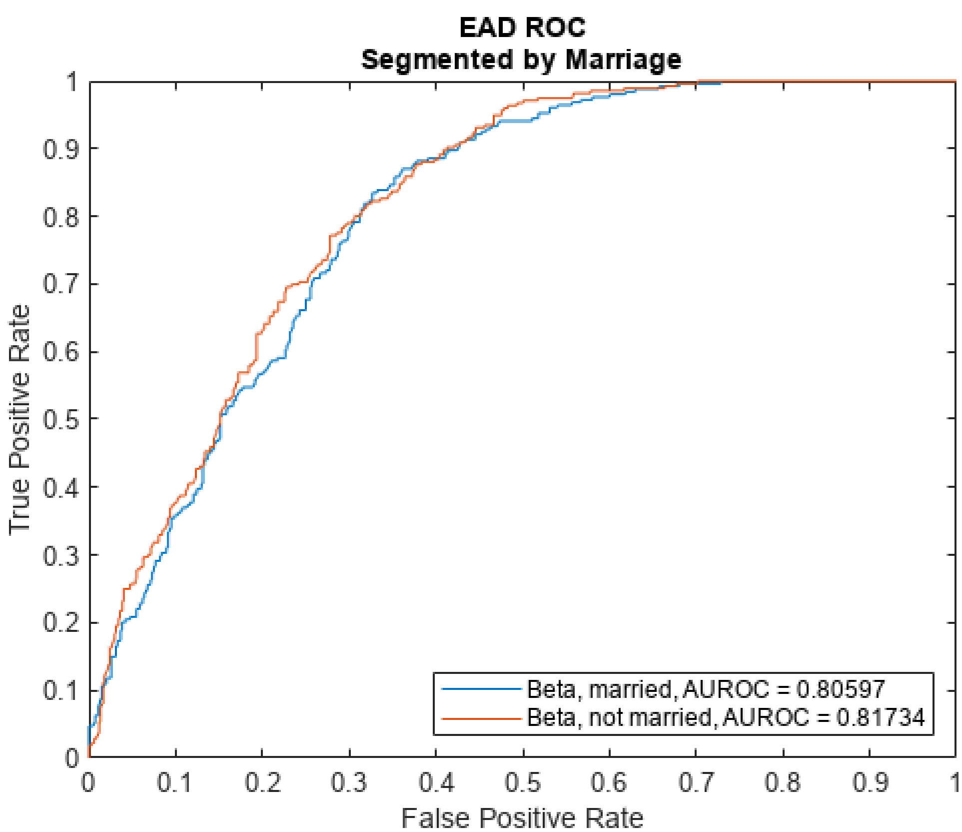
Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ead ;
```

```
[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel
modelDiscriminationPlot(eadModel,EADData(TestInd, :),ModelLevel=ModelLevel,SegmentBy="Marriage")
```



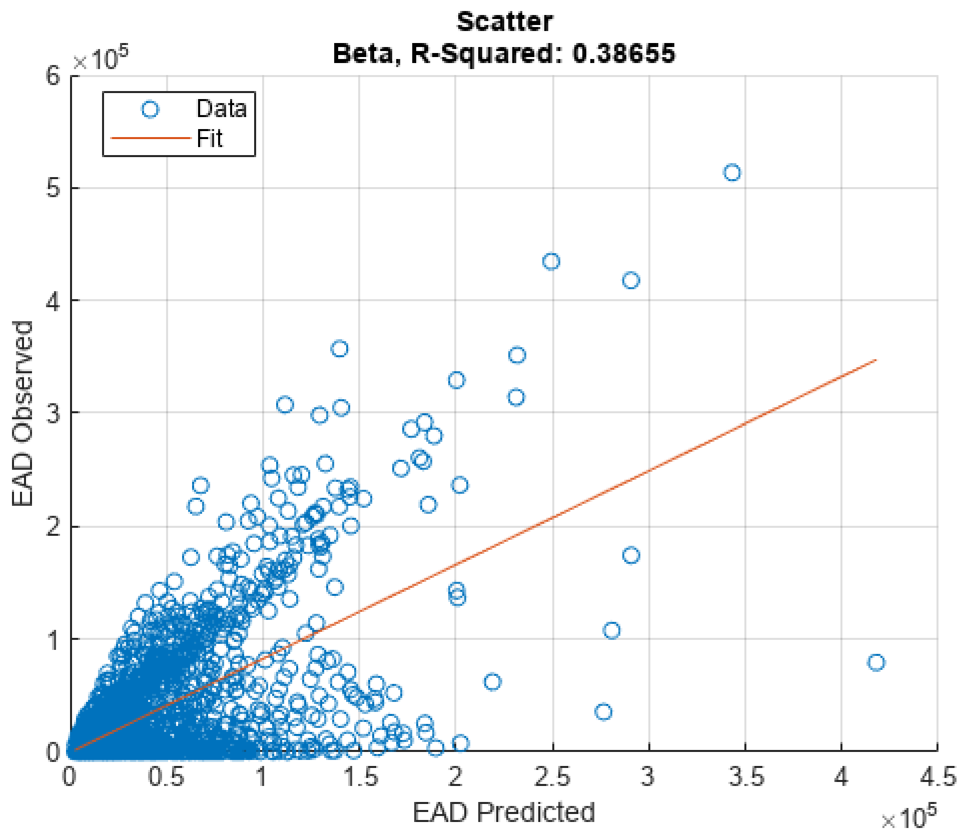
Use `modelCalibration`, and `modelCalibrationPlot` to show a scatter plot of the predictions.

```
YData = Observed ;
```

```
[CalMeasure1,CalData1] = modelCalibration(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel)
```

```
CalMeasure1=1×4 table
            RSquared    RMSE     Correlation    SampleMeanError
            ________    _____    ___________    _______________

    Beta    0.38655     43817      0.62173          -7393.4


CalData1=1751×3 table
     Observed     Predicted_Beta    Residuals_Beta
    __________    ______________    ______________

         44740           18039            26701
        54.175           10560           -10506
        987.39           15551           -14564
        9606.4          8407.7           1198.8
        83.809           33318           -33234
         73538           52120            21418
        96.949          6598.1          -6501.2
        873.21          5471.1          -4597.9
        328.35            7335          -7006.6
         55237           32580            22658
         30359           21563           8796.4
         39211           33177           6033.6
    2.0885e+05      1.2586e+05            82987
        1921.7           23319           -21397
         15230          6565.9             8664
         20063           11075           8987.5
          ⋮
```

```
modelCalibrationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Input Arguments

eadModel — Exposure at default model
Regression object | Tobit object | Beta object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitEADModel`.

Data Types: `object`

data — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[CalMeasure,CalData] = modelCalibration(eadModel,data(TestInd,:),DataID='Testing',CorrelationType='spearman')`

`CorrelationType` — Correlation type
`"pearson"` (default) | character vector with value of `'pearson'`, `'spearman'`, or `'kendall'` | string with value of `"pearson"`, `"spearman"`, or `"kendall"`

Correlation type, specified as `CorrelationType` and a character vector or string.

Data Types: `char` | `string`

`DataID` — Data set identifier
`""` (default) | character vector | string

Data set identifier, specified as `DataID` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`ModelLevel` — Model level
`'ead'` (default) | character vector with value `'ead'`, `'conversionMeasure'`, or `'conversionTransform'` | string with value `"ead"`, `"conversionMeasure"`, or `"conversionTransform"`

Model level, specified as `ModelLevel` and a character vector or string.

Note `Regression` models support all three model levels, but a `Tobit` or `Beta` model supports model levels only for `"ead"` and `"conversionMeasure"`.

Data Types: `char` | `string`

`ReferenceEAD` — EAD values predicted for `data` by reference model
`[]` (default) | numeric vector

EAD values predicted for `data` by the reference model, specified as `ReferenceEAD` and a `NumRows`-by-`1` numeric vector. The `modelCalibration` output information is reported for both the `eadModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as `ReferenceID` and a character vector or string. `ReferenceID` is used in the `modelCalibration` output for reporting purposes.

Data Types: `char` | `string`

Output Arguments

`CalMeasure` — Calibration measure
table

Calibration measure, returned as a table with columns `'RSquared'`, `'RMSE'`, `'Correlation'`, and `'SampleMeanError'`. `CalMeasure` has one row if only the `eadModel` accuracy is measured and it has two rows if reference model information is given. The row names of `CalMeasure` report the model ID and data ID (if provided).

`CalData` — Calibration data
table

Calibration data, returned as a table with observed EAD values, predicted EAD values, and residuals (observed minus predicted). Additional columns for predicted and residual values are included for the reference model, if provided. The `ModelID` and `ReferenceID` labels are appended in the column names.

More About

Model Calibration

Model calibration measures the accuracy of the predicted probability of EAD values using different metrics.

- R-squared — To compute the R-squared metric, `modelCalibration` fits a linear regression of the observed EAD values against the predicted EAD values:

  $$EAD_{obs} = a + b * EAD_{pred} + \varepsilon$$

  The R-square of this regression is reported. For more information, see “Coefficient of Determination (R-Squared)”.
- RMSE — To compute the root mean square error (RMSE), `modelCalibration` uses the following formula where N is the number of observations:

  $$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(EAD_i^{obs} - EAD_i^{pred})^2}$$
- Correlation — This metric is the correlation between the observed and predicted EAD:

  $$corr(EAD_{obs}, EAD_{pred})$$

  For more information and details about the different correlation types, see `corr`.
- Sample mean error — This metric is the difference between the mean observed EAD and the mean predicted EAD or, equivalently, the mean of the residuals:

  $$SampleMeanError = \frac{1}{N}\sum_{i=1}^{N}(EAD_i^{obs} - EAD_i^{pred})$$

Version History

Introduced in R2023a

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk*. Independently published, 2020.

See Also

`Regression` | `Tobit` | `Beta` | `fitEADModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibrationPlot`

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Exposure at Default Models" on page 1-34

modelCalibration

Compute R-square, RMSE, correlation, and sample mean error of predicted and observed LGDs

Syntax

```
CalMeasure = modelCalibration(lgdModel,data)
[CalMeasure,CalData] = modelCalibration( ___ ,Name,Value)
```

Description

`CalMeasure = modelCalibration(lgdModel,data)` computes the R-square, root mean square error (RMSE), correlation, and sample mean error of observed vs. predicted loss given default (LGD) data. `modelCalibration` supports comparison against a reference model and also supports different correlation types. By default, `modelCalibration` computes the metrics in the LGD scale. You can use the `ModelLevel` name-value pair argument to compute metrics using the underlying model's transformed scale.

`[CalMeasure,CalData] = modelCalibration( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed LGDs Using Regression LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Regression` model and then use `modelCalibration` to compute the R-Square, RMSE, correlation, and sample mean error of predicted and observed LGDs.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD   
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'regression');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Regression"
          Description: ""
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

```
lgdModel.UnderlyingModel
```

```
ans = 
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed LGDs

Use `modelCalibration` to compute the `RSquared`, `RMSE`, `Correlation`, and `SampleMeanError` of the predicted and observed LGDs for the test data set.

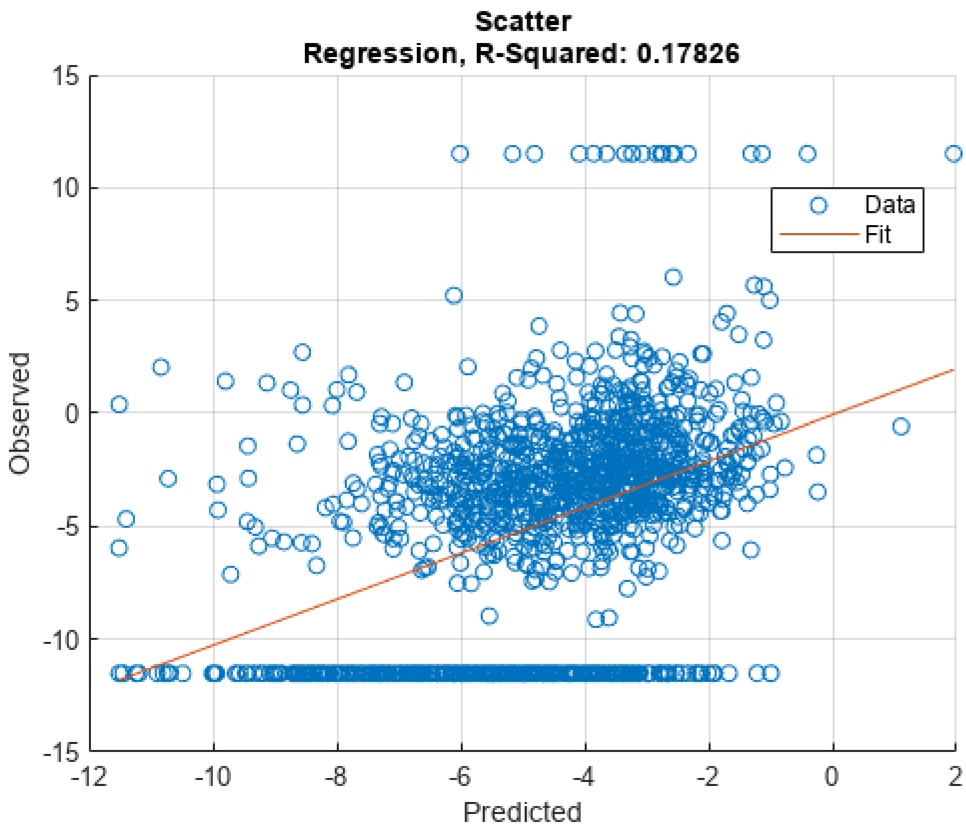
```
[CalMeasure,CalData] = modelCalibration(lgdModel,data(TestInd,:))
```

```
CalMeasure=1×4 table
                  RSquared     RMSE      Correlation    SampleMeanError
                  ________    _______    ___________    _______________
```

```
    Regression    0.070867    0.25988      0.26621          0.10759

CalData=1394×3 table
    Observed     Predicted_Regression    Residuals_Regression
    _________    ____________________    ____________________

    0.0064766         0.00091169               0.0055649
     0.007947          0.0036758               0.0042713
     0.063182            0.18774                -0.12456
            0          0.0010877              -0.0010877
      0.10904           0.011213                0.097823
            0           0.041992               -0.041992
      0.89463           0.052947                 0.84168
            0         3.7188e-06             -3.7188e-06
     0.072437          0.0090124                0.063425
     0.036006           0.023928                0.012078
            0          0.0034833              -0.0034833
      0.39549          0.0065253                 0.38896
     0.057675           0.071956               -0.014281
     0.014439          0.0061499                0.008289
            0          0.0012183              -0.0012183
            0          0.0019828              -0.0019828
      ⋮
```

Generate a scatter plot of predicted and observed LGDs using `modelCalibrationPlot`.

```
modelCalibrationPlot(lgdModel,data(TestInd,:),ModelLevel="underlying")
```



Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed LGDs Using Tobit LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Tobit` model and then use `modelCalibration` to compute R-Square, RMSE, correlation, and sample mean error of predicted and observed LGDs.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD   
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Tobit LGD Model

Use `fitLGDModel` to create a `Tobit` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'tobit');
disp(lgdModel)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Tobit regression model:
     LGD = max(0,min(Y*,1))
     Y* ~ 1 + LTV + Age + Type

Estimated coefficients:
                       Estimate        SE         tStat       pValue  
                       _________    _________    _______    __________

    (Intercept)         0.058257     0.027265     2.1367      0.032737
    LTV                  0.20126     0.031354     6.4189    1.6932e-10
    Age                -0.095407    0.0072653    -13.132             0
    Type_investment      0.10208     0.018058     5.6531    1.7915e-08
    (Sigma)              0.29288    0.0057036      51.35             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1521
Number of right-censored observations: 25
Log-likelihood: -698.383
```

Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed LGDs

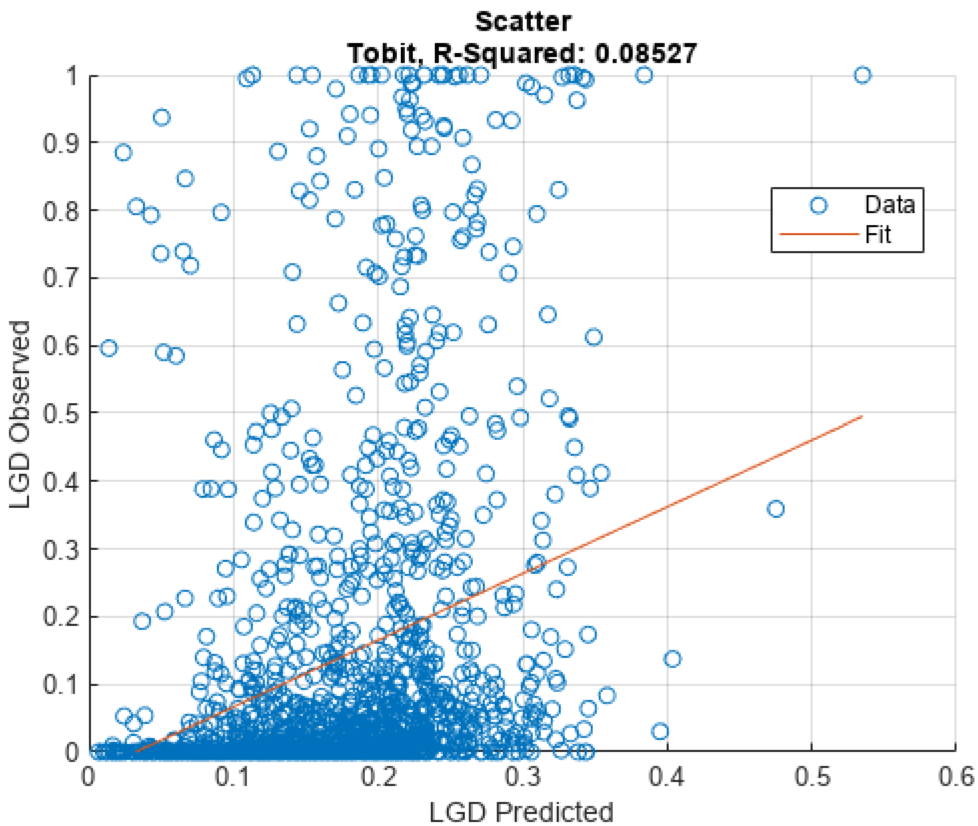
Use `modelCalibration` to compute `RSquared`, `RMSE`, `Correlation`, and `SampleMeanError` of predicted and observed LGDs for the test data set.

```
[CalMeasure,CalData] = modelCalibration(lgdModel,data(TestInd,:),CorrelationType="kendall")
```

```
CalMeasure=1×4 table
             RSquared     RMSE      Correlation    SampleMeanError
             ________    _______    ___________    _______________

    Tobit    0.08527     0.23712      0.29964         -0.034412


CalData=1394×3 table
    Observed     Predicted_Tobit    Residuals_Tobit
    _________    _______________    _______________

    0.0064766       0.087889           -0.081412
     0.007947        0.12432            -0.11638
     0.063182        0.32043            -0.25724
            0       0.093354           -0.093354
      0.10904        0.16718           -0.058144
            0        0.22382            -0.22382
      0.89463        0.23695             0.65768
            0       0.010234           -0.010234
     0.072437         0.1592           -0.086761
     0.036006        0.19893            -0.16292
            0        0.12764            -0.12764
      0.39549        0.14568              0.2498
     0.057675        0.26181            -0.20413
     0.014439        0.14483            -0.13039
            0       0.094123           -0.094123
            0        0.10944            -0.10944
      ⋮
```

Generate a scatter plot of the predicted and observed LGDs using `modelCalibrationPlot`.

```
modelCalibrationPlot(lgdModel,data(TestInd,:))
```



Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed LGDs Using Beta LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Beta` model and then use `modelCalibration` to compute R-Square, RMSE, correlation, and sample mean error of predicted and observed LGDs.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

LTV	Age	Type	LGD
0.89101	0.39716	residential	0.032659
0.70176	2.0939	residential	0.43564
0.72078	2.7948	residential	0.0064766
0.37013	1.237	residential	0.007947
0.36492	2.5818	residential	0
0.796	1.5957	residential	0.14572
0.60203	1.1599	residential	0.025688
0.92005	0.50253	investment	0.063182

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Beta LGD Model

Use `fitLGDModel` to create a `Beta` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'Beta');
disp(lgdModel)
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-05
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Beta regression model:
     logit(LGD) ~ 1_mu + LTV_mu + Age_mu + Type_mu
     log(LGD) ~ 1_phi + LTV_phi + Age_phi + Type_phi

Estimated coefficients:
                           Estimate       SE        tStat       pValue  
                           ________    ________    _______    __________

    (Intercept)_mu          -1.3772     0.13201    -10.433             0
    LTV_mu                  0.60269     0.15087     3.9947    6.7023e-05
    Age_mu                 -0.47464    0.040264    -11.788             0
    Type_investment_mu      0.45372    0.085143     5.3289     1.094e-07
    (Intercept)_phi        -0.16337     0.12591    -1.2975       0.19462
    LTV_phi                0.055892     0.14719    0.37973       0.70419
    Age_phi                 0.22887    0.040335     5.6743    1.5863e-08
    Type_investment_phi    -0.14102    0.078155    -1.8044      0.071311

Number of observations: 2093
Log-likelihood: -5291.04
```

Compute R-Square, RMSE, Correlation, and Sample Mean Error of Predicted and Observed LGDs

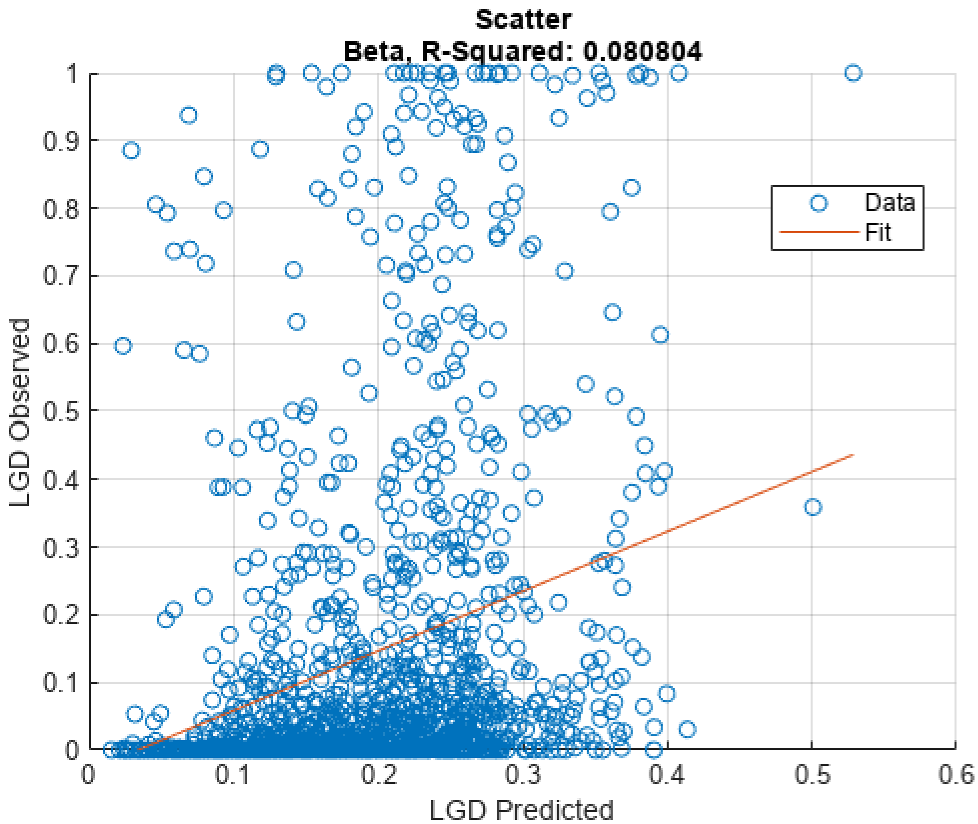
Use `modelCalibration` to compute `RSquared`, `RMSE`, `Correlation`, and `SampleMeanError` of predicted and observed LGDs for the test data set.

```
[CalMeasure,CalData] = modelCalibration(lgdModel,data(TestInd,:),CorrelationType="kendall")
```

```
CalMeasure=1×4 table
            RSquared     RMSE      Correlation    SampleMeanError
            ________    _______    ___________    _______________

    Beta    0.080804    0.24112      0.29448         -0.052396

CalData=1394×3 table
    Observed     Predicted_Beta    Residuals_Beta
    _________    ______________    ______________

    0.0064766       0.093695          -0.087218
     0.007947        0.14915            -0.1412
     0.063182        0.35263           -0.28944
            0       0.096434          -0.096434
      0.10904        0.18858          -0.079542
            0         0.2595            -0.2595
      0.89463        0.26767            0.62696
            0       0.021315          -0.021315
     0.072437        0.17736           -0.10492
     0.036006        0.22556           -0.18955
            0        0.13369           -0.13369
      0.39549        0.16768             0.2278
     0.057675        0.29159           -0.23392
     0.014439         0.1617           -0.14726
            0        0.10506           -0.10506
            0         0.1161            -0.1161
      ⋮
```

Generate a scatter plot of the predicted and observed LGDs using `modelCalibrationPlot`.

```
modelCalibrationPlot(lgdModel,data(TestInd,:))
```



Input Arguments

`lgdModel` — Loss given default model
`Regression` object | `Tobit` object | `Beta` object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitLGDModel`.

Data Types: `object`

`data` — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[CalMeasure,CalData] = modelCalibration(lgdModel,data(TestInd,:),DataID='Testing',CorrelationType='spearman')`

CorrelationType — Correlation type
"pearson" (default) | character vector with value of 'pearson', 'spearman', or 'kendall' | string with value of "pearson", "spearman", or "kendall'"

Correlation type, specified as CorrelationType and a character vector or string.

Data Types: char | string

DataID — Data set identifier
"" (default) | character vector | string

Data set identifier, specified as DataID and a character vector or string. The DataID is included in the output for reporting purposes.

Data Types: char | string

ModelLevel — Model level
'top' (default) | character vector with value 'top' or 'underlying' | string with value "top" or "underlying"

Model level, specified as ModelLevel and a character vector or string.

- 'top' — The accuracy metrics are computed in the LGD scale at the top model level.
- 'underlying' — For a Regression model only, the metrics are computed in the underlying model's transformed scale. The metrics are computed on the transformed LGD data.

Note ModelLevel has no effect for a Tobit or Beta model because there is no response transformation.

Data Types: char | string

ReferenceLGD — LGD values predicted for data by reference model
[ ] (default) | numeric vector

LGD values predicted for data by the reference model, specified as ReferenceLGD and a NumRows-by-1 numeric vector. The modelCalibration output information is reported for both the lgdModel object and the reference model.

Data Types: double

ReferenceID — Identifier for the reference model
'Reference' (default) | character vector | string

Identifier for the reference model, specified as ReferenceID and a character vector or string. 'ReferenceID' is used in the modelCalibration output for reporting purposes.

Data Types: char | string

Output Arguments

CalMeasure — Calibration measure
table

Calibration measure, returned as a table with columns 'RSquared', 'RMSE', 'Correlation', and 'SampleMeanError'. CalMeasure has one row if only the lgdModel accuracy is measured and it has two rows if reference model information is given. The row names of `CalMeasure` report the model ID and data ID (if provided).

`CalData` — Calibration data
table

Calibration data, returned as a table with observed LGD values, predicted LGD values, and residuals (observed minus predicted). Additional columns for predicted and residual values are included for the reference model, if provided. The `ModelID` and `ReferenceID` labels are appended in the column names.

More About

Model Calibration

Model calibration measures the accuracy of the predicted probability of LGD values using different metrics.

- R-squared — To compute the R-squared metric, `modelCalibration` fits a linear regression of the observed LGD values against the predicted LGD values

  $$LGD_{obs} = a + b * LGD_{pred} + \varepsilon$$

  The R-square of this regression is reported. For more information, see "Coefficient of Determination (R-Squared)".
- RMSE — To compute the root mean square error (RMSE), `modelCalibration` uses the following formula where N is the number of observations:

  $$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(LGD_i^{obs} - LGD_i^{pred})^2}$$
- Correlation — This is the correlation between the observed and predicted LGD:

  $$corr(LGD_{obs}, LGD_{pred})$$

  For more information and details about the different correlation types, see `corr`.
- Sample mean error — This is the difference between the mean observed LGD and the mean predicted LGD or, equivalently, the mean of the residuals:

  $$SampleMeanError = \frac{1}{N}\sum_{i=1}^{N}(LGD_i^{obs} - LGD_i^{pred})$$

Version History

Introduced in R2023a

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

`Tobit` | `Regression` | `Beta` | `modelCalibrationPlot` | `modelDiscriminationPlot` | `modelDiscrimination` | `predict` | `fitLGDModel`

Topics

“Model Loss Given Default” on page 4-90
“Basic Loss Given Default Model Validation” on page 4-131
“Compare Tobit LGD Model to Benchmark Model” on page 4-133
“Compare Loss Given Default Models Using Cross-Validation” on page 4-140
“Overview of Loss Given Default Models” on page 1-31

modelCalibration

Compute RMSE of predicted and observed PDs on grouped data

Syntax

```
CalMeasure = modelCalibration(pdModel,data,GroupBy)
[CalMeasure,CalData] = modelCalibration( ___ ,Name,Value)
```

Description

`CalMeasure = modelCalibration(pdModel,data,GroupBy)` computes the root mean squared error (RMSE) of the observed compared to the predicted probabilities of default (PD). `GroupBy` is required and can be any column in the `data` input (not necessarily a model variable). The `modelCalibration` function computes the observed PD as the default rate of each group and the predicted PD as the average PD for each group. `modelCalibration` supports comparison against a reference model.

`[CalMeasure,CalData] = modelCalibration( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute Model Calibration for Logistic Lifetime PD Model

This example shows how to use `fitLifetimePDModel` to fit data with a `Logistic` model and then use `modelCalibration` to compute the root mean squared error (RMSE) of the observed probabilities of default (PDs) with respect to the predicted PDs.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______
```

```
    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % For reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create Logistic Lifetime PD Model

Use `fitLifetimePDModel` to create a `Logistic` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Logistic",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
 disp(pdModel)
```

```
  Logistic with properties:

            ModelID: "Logistic"
        Description: ""
```

```
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Display the underlying model.

```
pdModel.UnderlyingModel
```

```
ans = 
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue  
                              __________    _________    _______    ___________

    (Intercept)                  -2.7422      0.10136    -27.054      3.408e-161
    ScoreGroup_Medium Risk      -0.68968     0.037286    -18.497      2.1894e-76
    ScoreGroup_Low Risk          -1.2587     0.045451    -27.693     8.4736e-169
    YOB                         -0.30894     0.013587    -22.738     1.8738e-114
    GDP                         -0.11111     0.039673    -2.8006       0.0051008
    Market                    -0.0083659    0.0028358    -2.9502       0.0031761


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Compute Model Calibration

Model calibration measures the predicted probabilities of default. For example, if the model predicts a 10% PD for a group, does the group end up showing an approximate 10% default rate, or is the eventual rate much higher or lower? While model discrimination measures the risk ranking only, model calibration measures the the predicted risk levels.

`modelCalibration` computes the root mean squared error (RMSE) of the observed PDs with respect to the predicted PDs. A grouping variable is required and it can be any column in the data input (not necessarily a model variable). The `modelCalibration` function computes the observed PD as the default rate of each group and the predicted PD as the average PD for each group.

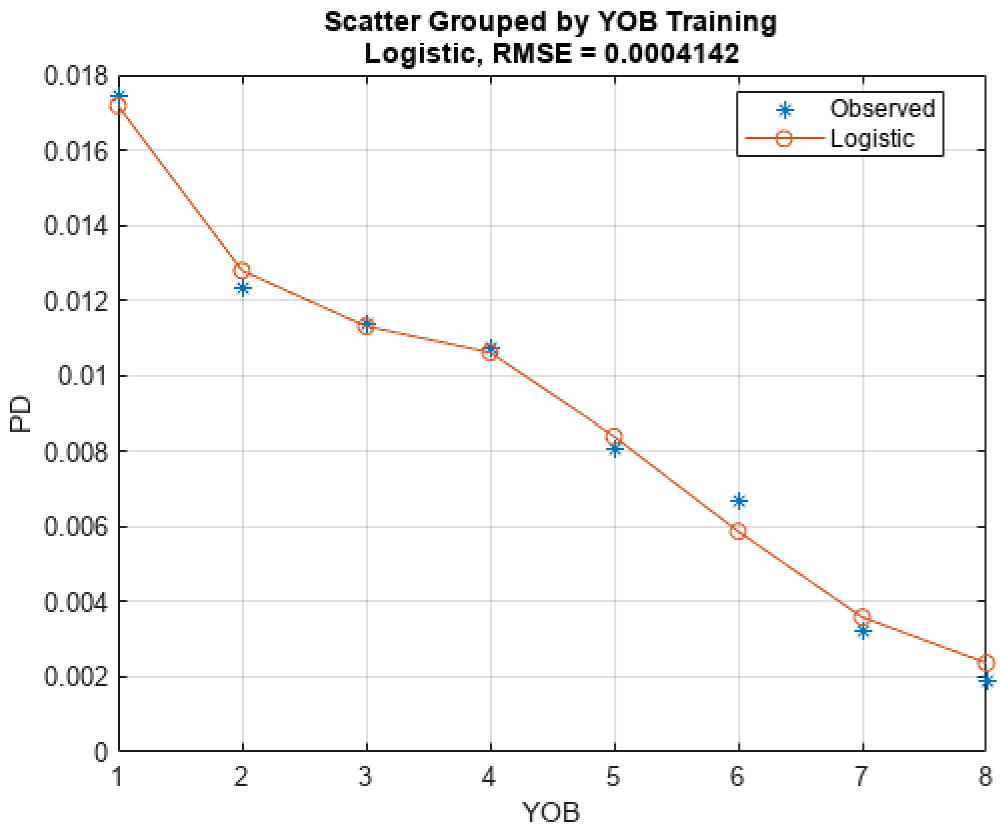
```
DataSetChoice = Training ;
 if DataSetChoice=="Training"
    Ind = TrainDataInd;
else
   Ind = TestDataInd;
 end

GroupingVar = YOB ;
[CalMeasure,CalData] = modelCalibration(pdModel,data(Ind,:),GroupingVar,DataID=DataSetChoice)
```

```
CalMeasure=table
                                           RMSE
```

```
                                         _________

    Logistic, grouped by YOB, Training    0.0004142

CalData=16×4 table
     ModelID      YOB       PD        GroupCount
    __________    ___    _________    __________

    "Observed"     1      0.017421      58092
    "Observed"     2      0.012305      56723
    "Observed"     3      0.011382      55524
    "Observed"     4      0.010741      54650
    "Observed"     5       0.00809      53770
    "Observed"     6     0.0066747      53186
    "Observed"     7     0.0032198      36959
    "Observed"     8     0.0018757      19193
    "Logistic"     1      0.017185      58092
    "Logistic"     2      0.012791      56723
    "Logistic"     3       0.01131      55524
    "Logistic"     4      0.010615      54650
    "Logistic"     5     0.0083982      53770
    "Logistic"     6     0.0058744      53186
    "Logistic"     7     0.0035872      36959
    "Logistic"     8     0.0023689      19193
```

Visualize the model calibration using `modelCalibrationPlot`.

```
modelCalibrationPlot(pdModel,data(Ind,:),GroupingVar,DataID=DataSetChoice);
```



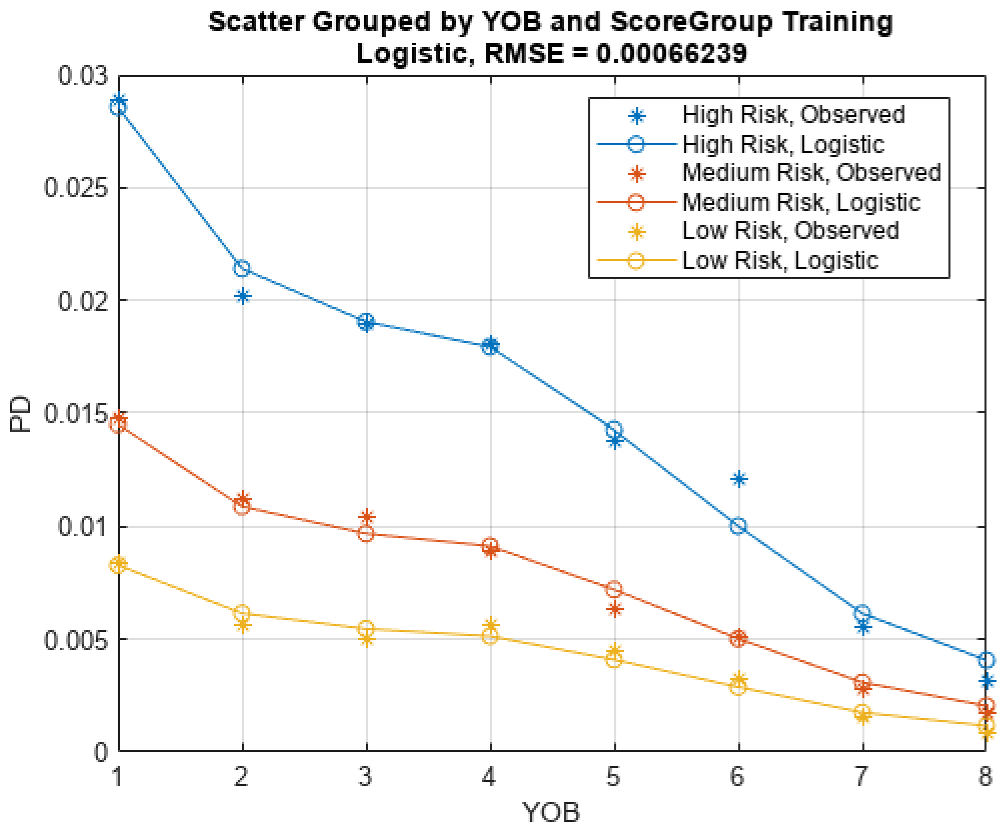
You can use more than one variable for grouping. For this example, group by the variables `YOB` and `ScoreGroup`.

```
CalMeasure = modelCalibration(pdModel,data(Ind,:),["YOB","ScoreGroup"],DataID=DataSetChoice);
disp(CalMeasure)
```

```
                                                  RMSE
                                               __________

    Logistic, grouped by YOB, ScoreGroup, Training    0.00066239
```

Now visualize the two grouping variables using `modelCalibrationPlot`.

```
modelCalibrationPlot(pdModel,data(Ind,:),["YOB","ScoreGroup"],DataID=DataSetChoice);
```



Input Arguments

pdModel — Probability of default model
Logistic object | Probit object | Cox object | customLifetimePDModel object

Probability of default model, specified as a previously created Logistic, Probit, or Cox object using fitLifetimePDModel. Alternatively, you can create a custom probability of default model using customLifetimePDModel.

Data Types: object

data — Data
table

Data, specified as a NumRows-by-NumCols table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model.

Data Types: table

GroupBy — Name of column in data input used to group the data
string | character vector

Name of column in the data input used to group the data, specified as a string or character vector. GroupBy does not have to be a model variable name. For each group designated by GroupBy, the modelCalibration function computes the observed default rates and average predicted PDs are computed to measure the RMSE.

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[CalMeasure,CalData] = modelCalibration(pdModel,data(Ind,:),GroupBy=["YOB","ScoreGroup"],DataID="DataSetChoice")`

`DataID` — Data set identifier
`""` (default) | character vector | string

Data set identifier, specified as `DataID` and a character vector or string. `DataID` is included in the `modelCalibration` output for reporting purposes.

Data Types: `char` | `string`

`ReferencePD` — Conditional PD values predicted for `data` by reference model
`[]` (default) | numeric vector

Conditional PD values predicted for `data` by the reference model, specified as `ReferencePD` and a `NumRows`-by-`1` numeric vector. The function reports the `modelCalibration` output information for both the `pdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as `ReferenceID` and a character vector or string. `ReferenceID` is used in the `modelCalibration` output for reporting purposes.

Data Types: `char` | `string`

Output Arguments

`CalMeasure` — Calibration measure
table

Calibration measure, returned as a single-column table of RMSE values.

This table has one row if only the `pdModel` accuracy is measured and it has two rows if reference model information is given. The row names of `CalMeasure` report the model IDs, grouping variables, and data ID.

Note The reported RMSE values depend on the grouping variable for the required `GroupBy` argument.

`CalData` — Calibration data
table

Calibration data, returned as a table of observed and predicted PD values for each group.

The reported observed PD values correspond to the observed default rate for each group. The reported predicted PD values are the average PD values predicted by the `pdModel` object for each group, and similarly for the reference model. The `modelCalibration` function stacks the PD data, placing the observed values for all groups first, then the predicted PDs for the `pdModel`, and then the predicted PDs for the reference model, if given.

The column `'ModelID'` identifies which rows correspond to the observed PD, `pdModel`, or reference model. The table also has one column for each grouping variable showing the unique combinations of grouping values. The `'PD'` column of `CalData` is a the PD data. The last column of `CalData` is a `'GroupCount'` column with the group counts data.

More About

Model Calibration

Model calibration measures the accuracy of the predicted probability of default (PD) values.

To measure model calibration, you must compare the predicted PD values to the observed default rates. For example, if a group of customers is predicted to have an average PD of 5%, then is the observed default rate for that group close to 5%?

The `modelCalibration` function requires a grouping variable to compute average predicted PD values within each group and the average observed default rate also within each group. `modelCalibration` uses the root mean squared error (RMSE) to measure the deviations between the observed and predicted values across groups. For example, the grouping variable could be the calendar year, so that rows corresponding to the same calendar year are grouped together. Then, for each year the software computes the observed default rate and the average predicted PD. The `modelCalibration` function then applies the RMSE formula to obtain a single measure of the prediction error across all years in the sample.

Suppose there are N observations in the data set, and there are M groups $G_1,\ldots,G_M$. The default rate for group G_i is

$$DR_i = \frac{D_i}{N_i}$$

where:

D_i is the number of defaults observed in group G_i.

N_i is the number of observations in group G_i.

The average predicted probability of default PD_i for group G_i is

$$PD_i = \frac{1}{N_i}\sum_{j \in G_i} PD(j)$$

where $PD(j)$ is the probability of default for observation j. In other words, this is the average of the predicted PDs within group G_i.

Therefore, the RMSE is computed as

$$RMSE = \sqrt{\sum_{i=1}^{M}\left(\frac{N_i}{N}\right)(DR_i - PD_i)^2}$$

The RMSE, as defined, depends on the selected grouping variable. For example, grouping by calendar year and grouping by years-on-books might result in different RSME values.

Use `modelCalibrationPlot` to visualize observed default rates and predicted PD values on grouped data.

Version History

Introduced in R2023a

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibrationPlot` | `predictLifetime` | `predict` | `fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

modelCalibrationPlot

Scatter plot of predicted and observed EADs

Syntax

```
modelCalibrationPlot(eadModel,data)
modelCalibrationPlot( ___ ,Name=Value)
h = modelCalibrationPlot(ax, ___ ,Name=Value)
```

Description

`modelCalibrationPlot(eadModel,data)` returns a scatter plot of observed vs. predicted exposure at default (EAD) data with a linear fit. `modelCalibrationPlot` supports comparison against a reference model. By default, `modelCalibrationPlot` plots in the EAD scale.

`modelCalibrationPlot( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax. You can use the `ModelLevel` name-value argument to compute metrics using the underlying model's transformed scale.

`h = modelCalibrationPlot(ax, ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Examples

Generate Scatter Plot of Predicted and Observed EADs Using a Tobit EAD Model

This example shows how to use `fitEADModel` to create a `Tobit` model and then use `modelCalibrationPlot` to generate a scatter plot for predicted and observed EADs.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
```

```
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Tobit` or `Regression`.

```
ModelType = Tobit ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = CCF ;
```

Create Tobit EAD Model

Use `fitEADModel` to create a `Tobit` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Tobit with properties:

        CensoringSide: "right"
            LeftLimit: NaN
           RightLimit: 1
              ModelID: "Tobit"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "ccf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Tobit regression model, right-censored:
     EAD_ccf = min(Y*,1)
     Y* ~ 1 + UtilizationRate + Age + Marriage

Estimated coefficients:
                            Estimate       SE        tStat       pValue 
                            _________   ________   ________   ________

    (Intercept)               0.51514    0.88065    0.58496    0.55863
    UtilizationRate           -1.5872     0.7043    -2.2535   0.024307
    Age                     -0.005808   0.017758   -0.32706    0.74365
    Marriage_not married     0.007784    0.11996   0.064888    0.94827
    (Sigma)                    1.4689   0.025508     57.588          0
```

```
Number of observations: 2627
Number of left-censored observations: 0
Number of uncensored observations: 2626
Number of right-censored observations: 1
Log-likelihood: -6311.87
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

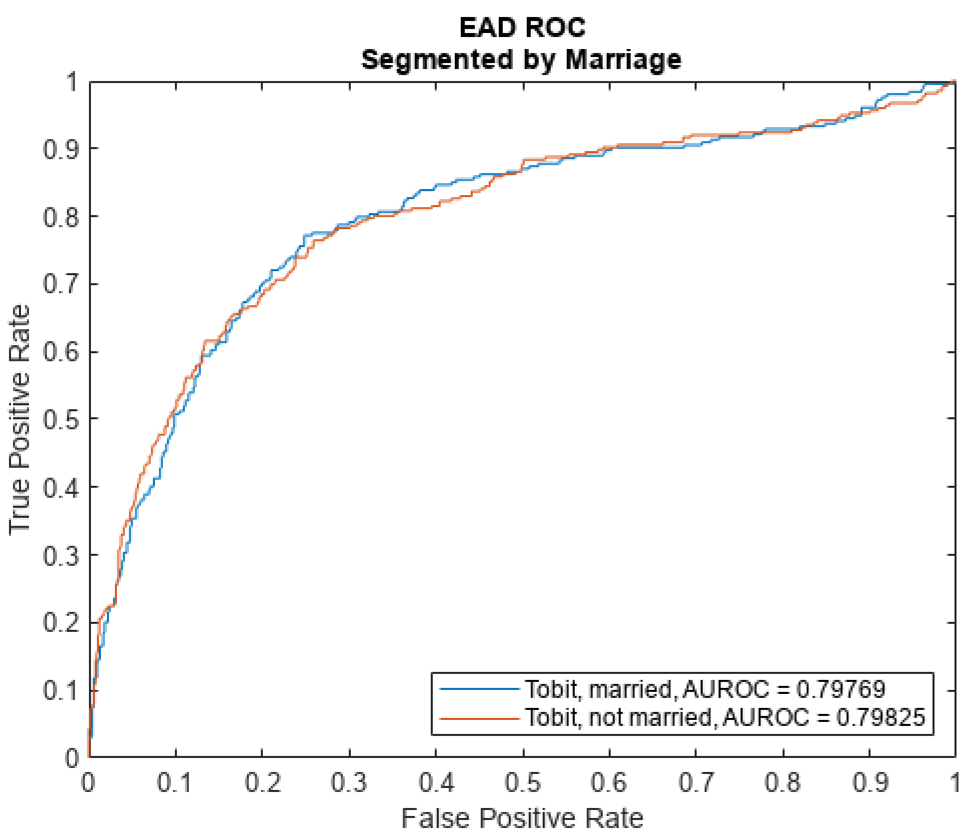
```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ead ;

[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel
modelDiscriminationPlot(eadModel,EADData(TestInd, :),ModelLevel=ModelLevel,SegmentBy="Marriage")
```



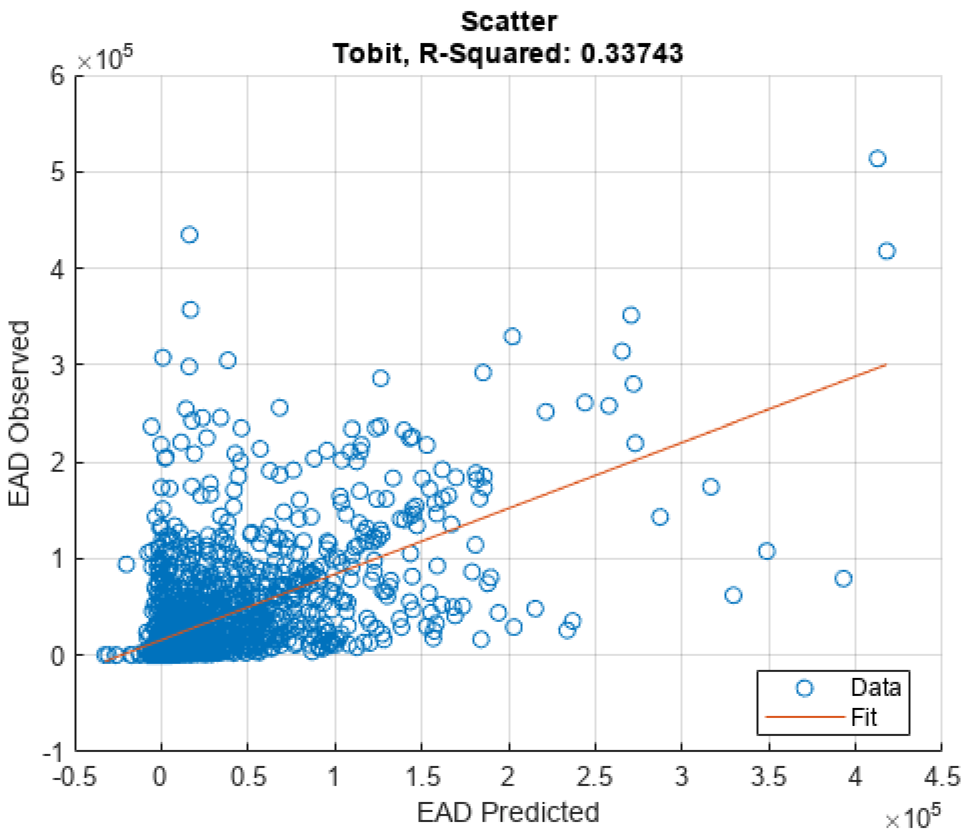
Use `modelCalibration` and then `modelCalibrationPlot` to show a scatter plot of the predictions.

```
YData = Observed ;

[CalMeasure1,CalData1] = modelCalibration(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel)
```

```
CalMeasure1=1×4 table
             RSquared    RMSE     Correlation    SampleMeanError
             ________    _____    ___________    _______________

    Tobit    0.33743     47299      0.58089          7285.1
```

```
CalData1=1751×3 table
     Observed     Predicted_Tobit    Residuals_Tobit
    __________    _______________    _______________

         44740        3337.3              41403
        54.175        1654.8            -1600.6
        987.39         11361             -10374
        9606.4        9702.4            -95.943
        83.809        2044.5            -1960.7
         73538         70963             2575.3
        96.949        846.22            -749.27
        873.21        640.46             232.75
        328.35        1357.3            -1028.9
         55237         21710              33527
         30359         29015             1344.2
         39211         27048              12163
    2.0885e+05         42571         1.6628e+05
        1921.7         11162            -9240.8
         15230          1542              13688
         20063        6298.9              13764
      ⋮
```

```
modelCalibrationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Generate Scatter Plot of Predicted and Observed EADs Using a Beta EAD Model

This example shows how to use `fitEADModel` to create a `Beta` model and then use `modelCalibrationPlot` to generate a scatter plot for predicted and observed EADs.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

UtilizationRate	Age	Marriage	Limit	Drawn	EAD
0.24359	25	not married	44776	10907	44740
0.96946	44	not married	2.1405e+05	2.0751e+05	40678
0	40	married	1.6581e+05	0	1.6567e+05
0.53242	38	not married	1.7375e+05	92506	1593.5
0.2583	30	not married	26258	6782.5	54.175
0.17039	54	married	1.7357e+05	29575	576.69
0.18586	27	not married	19590	3641	998.49
0.85372	42	not married	2.0712e+05	1.7682e+05	1.6454e+05

```
rng('default');
NumObs = height(EADData);
```

```
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Beta`.

```
ModelType = Beta ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Beta EAD Model

Use `fitEADModel` to create a `Beta` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-07
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Beta regression model:
     logit(EAD_lcf) ~ 1_mu + UtilizationRate_mu + Age_mu + Marriage_mu
     log(EAD_lcf) ~ 1_phi + UtilizationRate_phi + Age_phi + Marriage_phi

Estimated coefficients:
                                 Estimate        SE        tStat       pValue  
                                __________    _________    ________    __________

    (Intercept)_mu                -0.65566      0.11484     -5.7093    1.2616e-08
    UtilizationRate_mu              1.7014     0.078094      21.787             0
    Age_mu                      -0.0055901    0.0027603     -2.0252      0.042949
    Marriage_not married_mu      -0.012577     0.052098    -0.24141       0.80926
    (Intercept)_phi               -0.50131     0.094625     -5.2979    1.2686e-07
    UtilizationRate_phi            0.39731     0.066707       5.956    2.9303e-09
    Age_phi                      -0.001167    0.0023161    -0.50387        0.6144
```

```
    Marriage_not married_phi     -0.013275     0.042627     -0.31143        0.7555

Number of observations: 2627
Log-likelihood: -3140.21
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

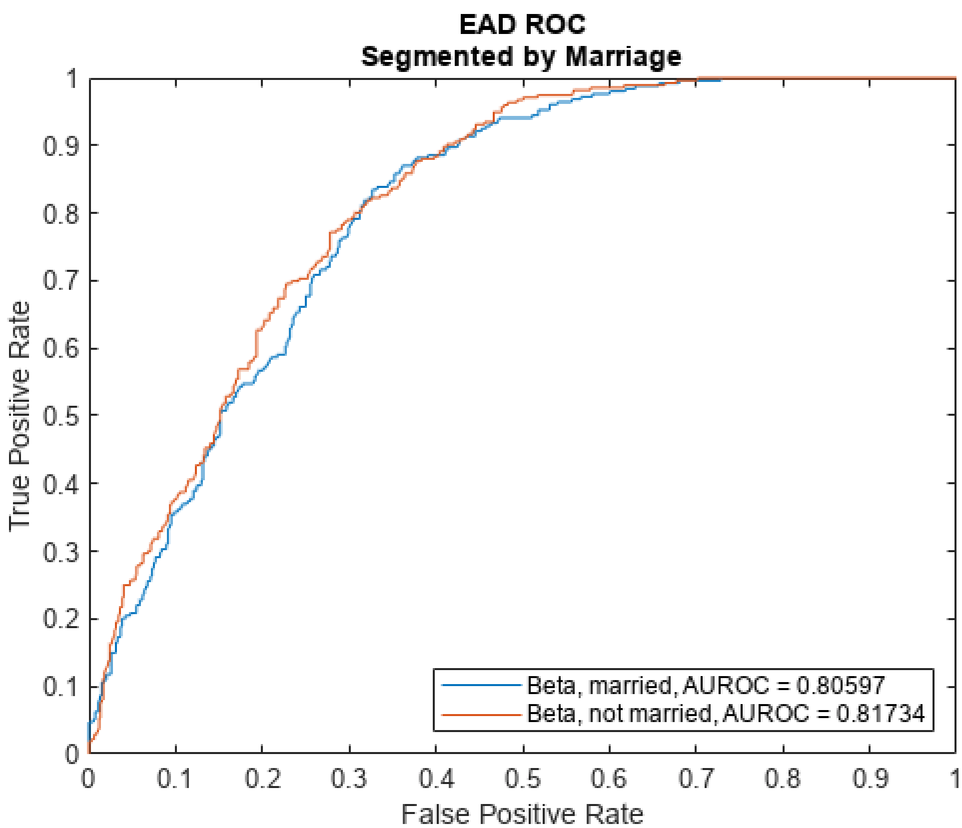
```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ead ;

[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel
modelDiscriminationPlot(eadModel,EADData(TestInd, :),ModelLevel=ModelLevel,SegmentBy="Marriage")
```

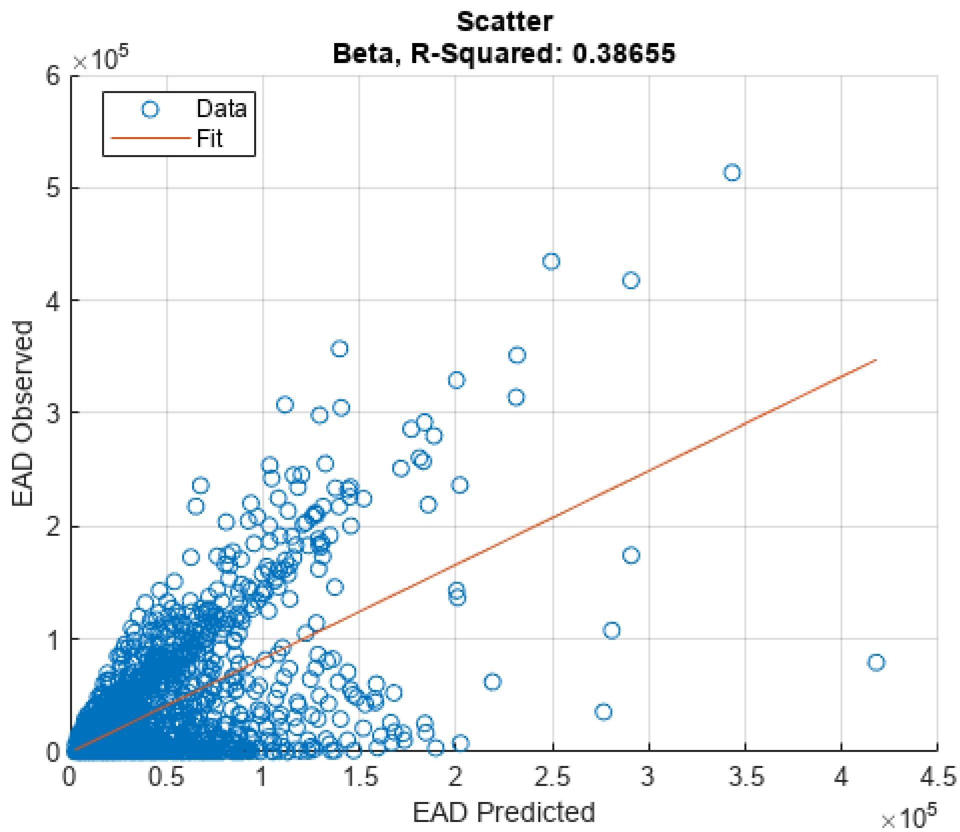


Use `modelCalibration` and then `modelCalibrationPlot` to show a scatter plot of the predictions.

```
YData = Observed ;

[CalMeasure1,CalData1] = modelCalibration(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel)
```

```
CalMeasure1=1×4 table
            RSquared    RMSE     Correlation    SampleMeanError
            ________    _____    ___________    _______________

    Beta    0.38655     43817      0.62173          -7393.4


CalData1=1751×3 table
     Observed     Predicted_Beta    Residuals_Beta
    __________    ______________    ______________

         44740          18039             26701
        54.175          10560            -10506
        987.39          15551            -14564
        9606.4         8407.7            1198.8
        83.809          33318            -33234
         73538          52120             21418
        96.949         6598.1           -6501.2
        873.21         5471.1           -4597.9
        328.35           7335           -7006.6
         55237          32580             22658
         30359          21563            8796.4
         39211          33177            6033.6
    2.0885e+05     1.2586e+05             82987
        1921.7          23319            -21397
         15230         6565.9              8664
         20063          11075            8987.5
      ⋮
```

```
modelCalibrationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Input Arguments

eadModel — Exposure at default model

Regression object | Tobit | Beta object

Exposure at default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitEADModel`.

Data Types: `object`

data — Data

table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

ax — Valid axis object

object

(Optional) Valid axis object, specified as an `ax` object that is created using `axes`. The plot will be created in the axes specified by the optional `ax` argument instead of in the current axes (gca). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `modelCalibrationPlot(eadModel,data(TestInd,:),DataID=Testing,XData='residuals',YData='residuals')`

`DataID` — Data set identifier
`""` (default) | character vector | string

Data set identifier, specified `DataID` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`ModelLevel` — Model level
`'ead'` (default) | character vector with value `'ead'`, `'conversionMeasure'`, or `'conversionTransform'` | string with value `"ead"`, `"conversionMeasure"`, or `"conversionTransform"`

Model level, specified as `ModelLevel` and a character vector or string.

Note `Regression` models support all three model levels, but a `Tobit` or `Beta` model supports model levels only for `"ead"` and `"conversionMeasure"`.

Data Types: `char` | `string`

`ReferenceEAD` — EAD values predicted for `data` by reference model
`[]` (default) | numeric vector

EAD values predicted for `data` by the reference model, specified as `ReferenceEAD` and a `NumRows`-by-`1` numeric vector. The scatter plot output is plotted for both the `eadModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as `ReferenceID` and a character vector or string. `ReferenceID` is used in the scatter plot output for reporting purposes.

Data Types: `char` | `string`

`XData` — Data to plot on *x*-axis
`'predicted'` (default) | character vector with value `'predicted'`, `'observed'`, `'residuals'`, or *VariableName* | string with value | `"predicted"`, `"observed"`, `"residuals"`, or *VariableName*

Data to plot on *x*-axis, specified as `XData` and a character vector or string for one of the following:

- `'predicted'` — Plot the predicted EAD values in the *x*-axis.

- `'observed'` — Plot the observed EAD values in the *x*-axis.
- `'residuals'` — Plot the residuals in the *x*-axis.
- *VariableName* — Use the name of the variable in the `data` input, not necessarily a model variable, to plot in the *x*-axis.

Data Types: `char` | `string`

`YData` — Data to plot on *y*-axis
`'predicted'` (default) | character vector with value `'predicted'`, `'observed'`, or `'residuals'` | string with value | `"predicted"`, `"observed"`, or `"residuals"`

Data to plot on *y*-axis, specified as `YData` and a character vector or string for one of the following:

- `'predicted'` — Plot the predicted EAD values in the *y*-axis.
- `'observed'` — Plot the observed EAD values in the *y*-axis.
- `'residuals'` — Plot the residuals in the *y*-axis.

Data Types: `char` | `string`

Output Arguments

`h` — Figure handle
handle object

Figure handle for the scatter and line objects, returned as handle object.

More About

Model Calibration Plot

The `modelCalibrationPlot` function returns a scatter plot of observed vs. predicted loss given default (EAD) data with a linear fit and reports the R-square of the linear fit.

The `XData` name-value pair argument allows you to change the *x* values on the plot. By default, predicted EAD values are plotted in the *x*-axis, but predicted EAD values, residuals, or any variable in the `data` input, not necessarily a model variable, can be used as *x* values. If the selected `XData` is a categorical variable, a swarm chart is used. For more information, see `swarmchart`.

The `YData` name-value pair argument allows users to change the *y* values on the plot. By default, observed EAD values are plotted in the *y*-axis, but predicted EAD values or residuals can also be used as *y* values. `YData` does not support table variables.

The linear fit and reported R-squared value always correspond to the linear regression model with the plotted *y* values as response and the plotted *x* values as the only predictor.

Version History

Introduced in R2023a

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

`Regression` | `Tobit` | `Beta` | `fitEADModel` | `predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration`

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Exposure at Default Models" on page 1-34

modelCalibrationPlot

Scatter plot of predicted and observed LGDs

Syntax

```
modelCalibrationPlot(lgdModel,data)
modelCalibrationPlot( ___ ,Name,Value)
h = modelCalibrationPlot(ax, ___ ,Name,Value)
```

Description

`modelCalibrationPlot(lgdModel,data)` returns a scatter plot of observed vs. predicted loss given default (LGD) data with a linear fit. `modelCalibrationPlot` supports comparison against a reference model. By default, `modelCalibrationPlot` plots in the LGD scale.

`modelCalibrationPlot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. You can use the `ModelLevel` name-value pair argument to compute metrics using the underlying model's transformed scale.

`h = modelCalibrationPlot(ax, ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Examples

Generate a Scatter Plot of Predicted and Observed LGDs Using Regression LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Regression` model and then use `modelCalibrationPlot` to generate a scatter plot for predicted and observed LGDs.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD   
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'regression');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Regression"
          Description: ""
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

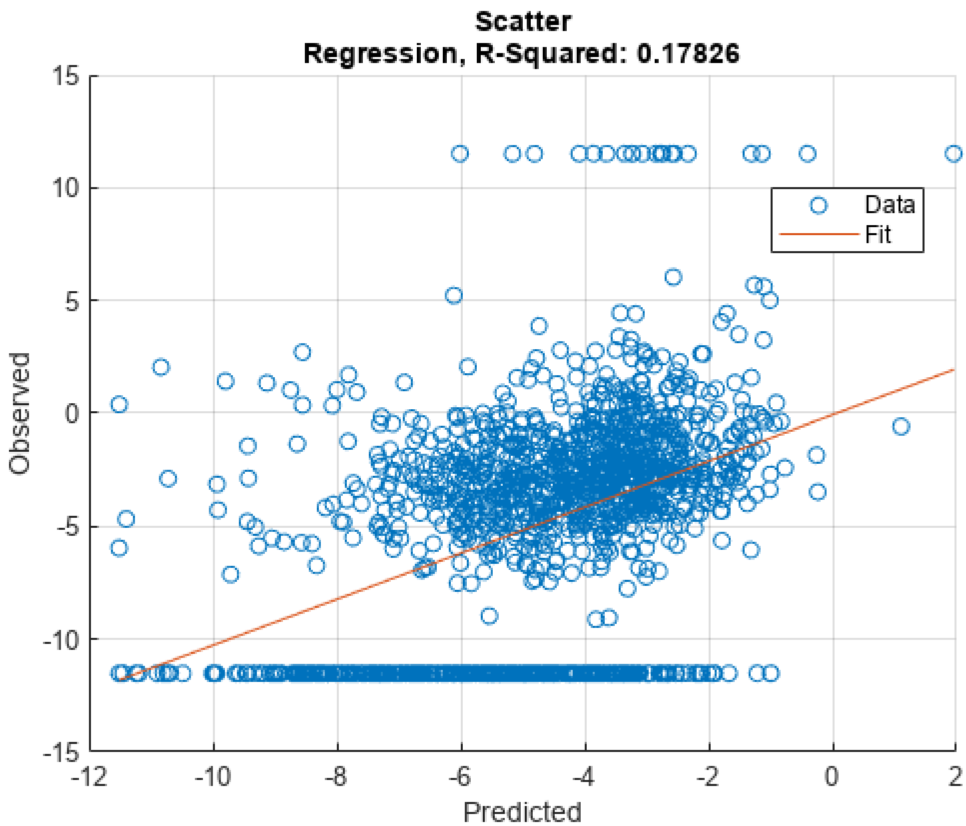
```
lgdModel.UnderlyingModel
```

```
ans = 
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

Generate Scatter Plot of Predicted and Observed LGDs

Use `modelCalibrationPlot` to generate a scatter plot of predicted and observed LGDs for the test data set. The `ModelLevel` name-value pair argument modifies the output only for `Regression` models, not `Tobit` models, because there are no response transformations for the `Tobit` model.

```
modelCalibrationPlot(lgdModel,data(TestInd,:),ModelLevel="underlying")
```



Generate Scatter Plot of Predicted and Observed LGDs Using Tobit LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Tobit` model and then use `modelCalibrationPlot` to generate a scatter plot of predicted and observed LGDs.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD   
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Tobit LGD Model

Use `fitLGDModel` to create a `Tobit` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'tobit');
disp(lgdModel)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

Display the underlying model.

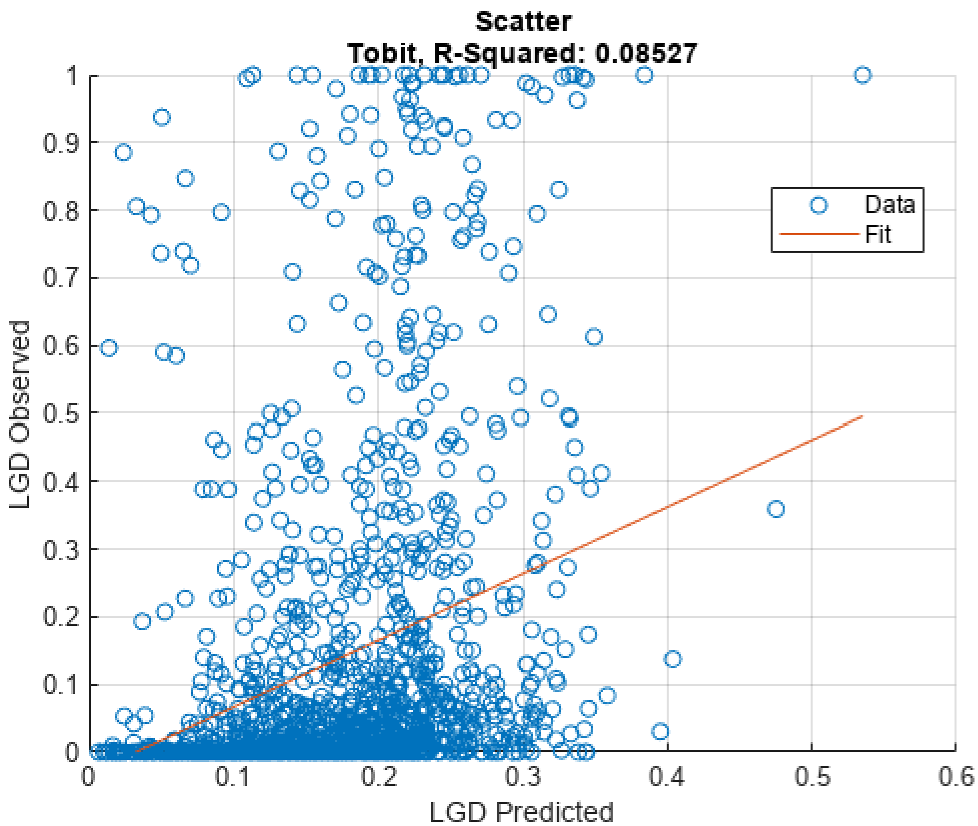
```
disp(lgdModel.UnderlyingModel)
```

```
Tobit regression model:
     LGD = max(0,min(Y*,1))
     Y* ~ 1 + LTV + Age + Type

Estimated coefficients:
                       Estimate        SE         tStat       pValue  
                       _________    _________    _______    __________

    (Intercept)         0.058257     0.027265     2.1367      0.032737
    LTV                  0.20126     0.031354     6.4189    1.6932e-10
    Age                -0.095407    0.0072653    -13.132             0
    Type_investment      0.10208     0.018058     5.6531    1.7915e-08
    (Sigma)              0.29288    0.0057036      51.35             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1521
Number of right-censored observations: 25
Log-likelihood: -698.383
```

Generate Scatter Plot of Predicted and Observed LGDs

Use `modelCalibrationPlot` to generate a scatter plot of predicted and observed LGDs for the test data set.

```
modelCalibrationPlot(lgdModel,data(TestInd,:))
```



Generate Scatter Plot of Predicted and Observed LGDs Using Beta LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Beta` model and then use `modelCalibrationPlot` to generate a scatter plot of predicted and observed LGDs.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV         Age          Type           LGD
    _______     _______     ___________     _________

    0.89101     0.39716     residential      0.032659
    0.70176      2.0939     residential       0.43564
    0.72078      2.7948     residential     0.0064766
    0.37013       1.237     residential      0.007947
    0.36492      2.5818     residential             0
      0.796      1.5957     residential       0.14572
    0.60203      1.1599     residential      0.025688
    0.92005     0.50253     investment       0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Beta LGD Model

Use `fitLGDModel` to create a `Beta` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'Beta');
disp(lgdModel)
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-05
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

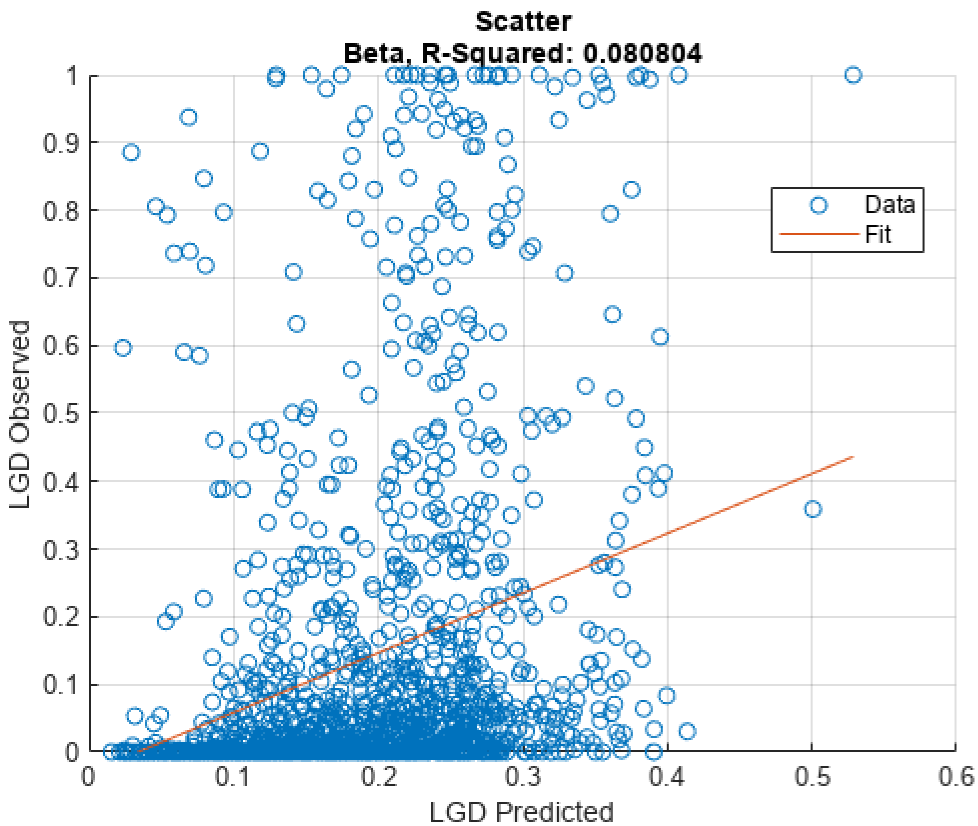
```
disp(lgdModel.UnderlyingModel)
```

```
Beta regression model:
     logit(LGD) ~ 1_mu + LTV_mu + Age_mu + Type_mu
     log(LGD) ~ 1_phi + LTV_phi + Age_phi + Type_phi

Estimated coefficients:
                           Estimate       SE        tStat       pValue  
                           ________    ________    _______    __________

    (Intercept)_mu          -1.3772     0.13201    -10.433             0
    LTV_mu                  0.60269     0.15087     3.9947    6.7023e-05
    Age_mu                 -0.47464    0.040264    -11.788             0
    Type_investment_mu      0.45372    0.085143     5.3289     1.094e-07
    (Intercept)_phi        -0.16337     0.12591    -1.2975       0.19462
    LTV_phi                0.055892     0.14719    0.37973       0.70419
    Age_phi                 0.22887    0.040335     5.6743    1.5863e-08
    Type_investment_phi    -0.14102    0.078155    -1.8044      0.071311

Number of observations: 2093
Log-likelihood: -5291.04
```

Generate Scatter Plot of Predicted and Observed LGDs

Use `modelCalibrationPlot` to generate a scatter plot of predicted and observed LGDs for the test data set.

```
modelCalibrationPlot(lgdModel,data(TestInd,:))
```



Visualize Calibration for Residuals or Other Variables

`modelCalibrationPlot` generates a scatter plot of observed vs. predicted LGD values. The `'XData'` and `'YData'` name-value arguments allow you to visualize the residuals or generate a scatter plot against a variable of interest.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'regression');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Regression"
          Description: ""
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

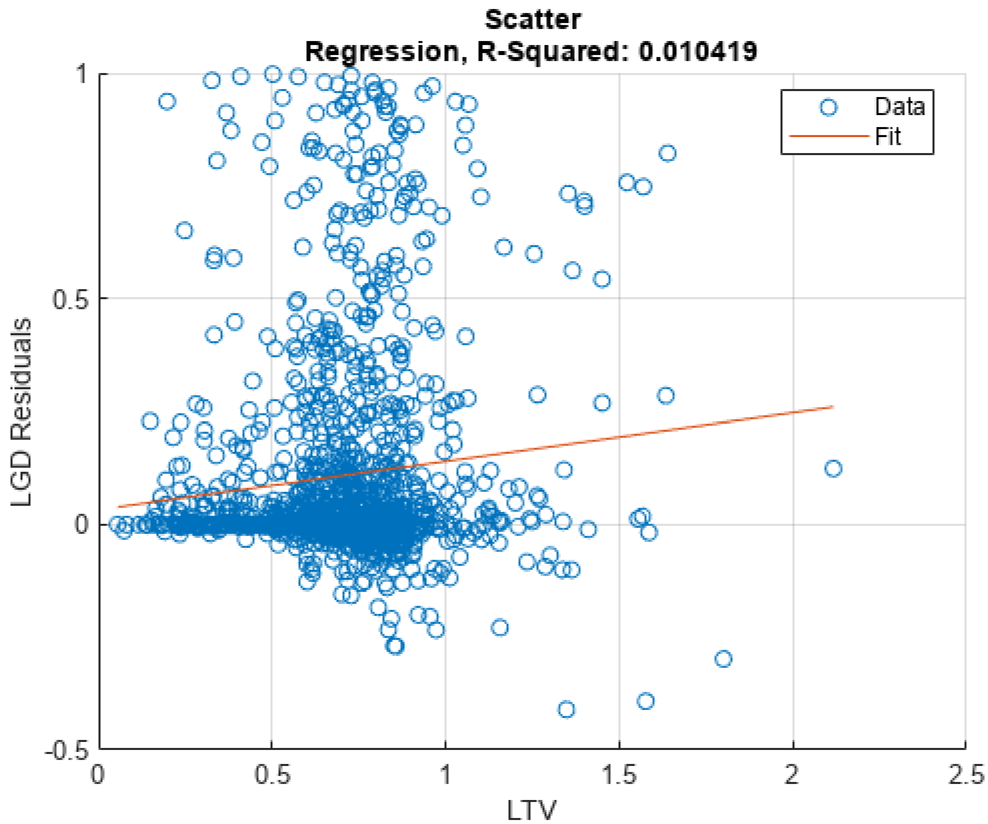
```
disp(lgdModel.UnderlyingModel)
```

```
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE         tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

Generate Scatter Plot of Predicted and Observed LGDs

Use `modelCalibrationPlot` to generate a scatter plot of residuals against LTV values.

```
modelCalibrationPlot(lgdModel,data(TestInd,:),XData='LTV',YData='residuals')
```

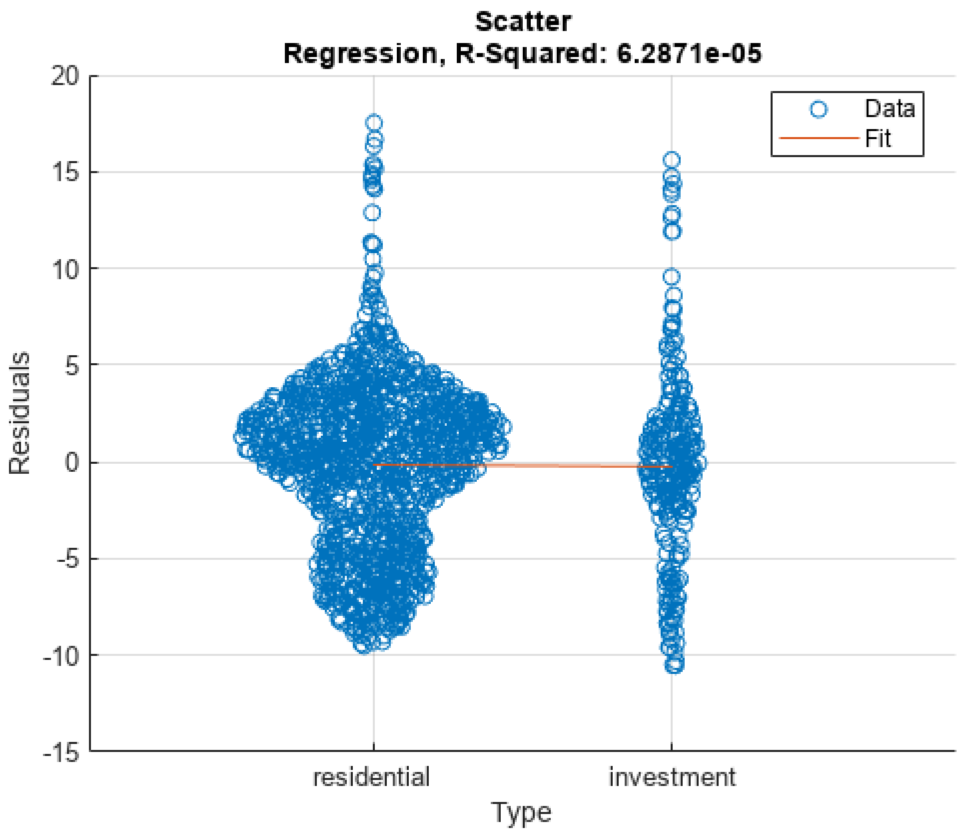


For `Regression` models, the `'ModelLevel'` name-value argument allows you to visualize the plot using the underlying model scale.

```
modelCalibrationPlot(lgdModel,data(TestInd,:),XData='LTV',YData='residuals',ModelLevel='underlyi
```

For categorical variables, `modelCalibrationPlot` uses a swarm chart. For more information, see `swarmchart`.

```
modelCalibrationPlot(lgdModel,data(TestInd,:),XData='Type',YData='residuals',ModelLevel='underly
```



Input Arguments

lgdModel — Loss given default model
Regression object | Tobit object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitLGDModel`.

Data Types: `object`

data — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

ax — Valid axis object
object

(Optional) Valid axis object, specified as an `ax` object that is created using `axes`. The plot will be created in the axes specified by the optional `ax` argument instead of in the current axes (gca). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `modelCalibrationPlot(lgdModel,data(TestInd,:),DataID='Testing',YData=residuals,XData='LTV')`

`DataID` — Data set identifier
`""` (default) | character vector | string

Data set identifier, specified as `DataID` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`ModelLevel` — Model level
`'top'` (default) | character vector with value `'top'` or `'underlying'` | string with value `"top"` or `"underlying"`

Model level, specified as `ModelLevel` and a character vector or string.

- `'top'` — The accuracy metrics are computed in the LGD scale at the top model level.
- `'underlying'` — For a `Regression` model only, the metrics are computed in the underlying model's transformed scale. The metrics are computed on the transformed LGD data.

Note `ModelLevel` has no effect for a `Tobit` or `Beta` model because there is no response transformation.

Data Types: `char` | `string`

`ReferenceLGD` — LGD values predicted for `data` by reference model
`[ ]` (default) | numeric vector

LGD values predicted for `data` by the reference model, specified as `ReferenceLGD` and a `NumRows`-by-`1` numeric vector. The scatter plot output is plotted for both the `lgdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as `ReferenceID` and a character vector or string. `'ReferenceID'` is used in the scatter plot output for reporting purposes.

Data Types: `char` | `string`

`XData` — Data to plot on x-axis
`'predicted'` (default) | character vector with value `'predicted'`, `'observed'`, `'residuals'`, or *VariableName* | string with value | `"predicted"`, `"observed"`, `"residuals"`, or *VariableName*

Data to plot on x-axis, specified as `XData` and a character vector or string for one of the following:

- `'predicted'` — Plot the predicted LGD values in the x-axis.
- `'observed'` — Plot the observed LGD values in the x-axis.
- `'residuals'` — Plot the residuals in the x-axis.
- *VariableName* — Use the name of the variable in the `data` input, not necessarily a model variable, to plot in the x-axis.

Data Types: `char` | `string`

`YData` — Data to plot on y-axis
`'predicted'` (default) | character vector with value `'predicted'`, `'observed'`, or `'residuals'` | string with value | `"predicted"`, `"observed"`, or `"residuals"`

Data to plot on y-axis, specified as `YData` and a character vector or string for one of the following:

- `'predicted'` — Plot the predicted LGD values in the y-axis.
- `'observed'` — Plot the observed LGD values in the y-axis.
- `'residuals'` — Plot the residuals in the y-axis.

Data Types: `char` | `string`

Output Arguments

h — Figure handle
handle object

Figure handle for the scatter and line objects, returned as handle object.

More About

Model Calibration Plot

The `modelCalibrationPlot` function returns a scatter plot of observed vs. predicted loss given default (LGD) data with a linear fit and reports the R-square of the linear fit.

The `XData` name-value pair argument allows you to change the *x* values on the plot. By default, predicted LGD values are plotted in the *x*-axis, but predicted LGD values, residuals, or any variable in the `data` input, not necessarily a model variable, can be used as *x* values. If the selected `XData` is a categorical variable, a swarm chart is used. For more information, see `swarmchart`.

The `YData` name-value pair argument allows users to change the *y* values on the plot. By default, observed LGD values are plotted in the *y*-axis, but predicted LGD values or residuals can also be used as *y* values. `YData` does not support table variables.

For `Regression` models, if `ModelLevel` is set to `'underlying'`, the LGD data is transformed into the underlying model's scale. The transformed data is shown on the plot. The `ModelLevel` name-value pair argument has no effect for `Tobit` models.

The linear fit and reported R-squared value always correspond to the linear regression model with the plotted *y* values as response and the plotted *x* values as the only predictor.

Version History

Introduced in R2023a

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

`Tobit` | `Regression` | `Beta` | `modelCalibration` | `modelDiscriminationPlot` | `modelDiscrimination` | `predict` | `fitLGDModel`

Topics

"Model Loss Given Default" on page 4-90
"Basic Loss Given Default Model Validation" on page 4-131
"Compare Tobit LGD Model to Benchmark Model" on page 4-133
"Compare Loss Given Default Models Using Cross-Validation" on page 4-140
"Overview of Loss Given Default Models" on page 1-31

modelCalibrationPlot

Plot observed default rates compared to predicted PDs on grouped data

Syntax

```
modelCalibrationPlot(pdModel,data,GroupBy)
modelCalibrationPlot( ___ ,Name,Value)
h = modelCalibrationPlot(ax, ___ ,Name,Value)
```

Description

`modelCalibrationPlot(pdModel,data,GroupBy)` plots the observed default rates compared to the predicted probabilities of default (PD). `GroupBy` is required and can be any column in the `data` input (not necessarily a model variable). The `modelCalibrationPlot` function computes the observed PD as the default rate of each group and the predicted PD as the average PD for each group. `modelCalibrationPlot` supports comparison against a reference model.

`modelCalibrationPlot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

`h = modelCalibrationPlot(ax, ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Examples

Plot RMSE of Observed Compared to Predicted Probabilities of Default

This example shows how to use `modelCalibrationPlot` to plot the root mean squared error (RMSE) of the observed probabilities of default (PDs) with respect to the predicted PDs.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % For reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create Logistic Lifetime PD Model

Use `fitLifetimePDModel` to create a `Logistic` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),'logistic',...
        'ModelID','Example',...
        'Description','Lifetime PD model using RetailCreditPanelData.',...
        'IDVar','ID',...
        'AgeVar','YOB',...
        'LoanVars','ScoreGroup',...
        'MacroVars',{'GDP' 'Market'},...
        'ResponseVar','Default');
 disp(pdModel)
```

```
  Logistic with properties:

            ModelID: "Example"
        Description: "Lifetime PD model using RetailCreditPanelData."
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

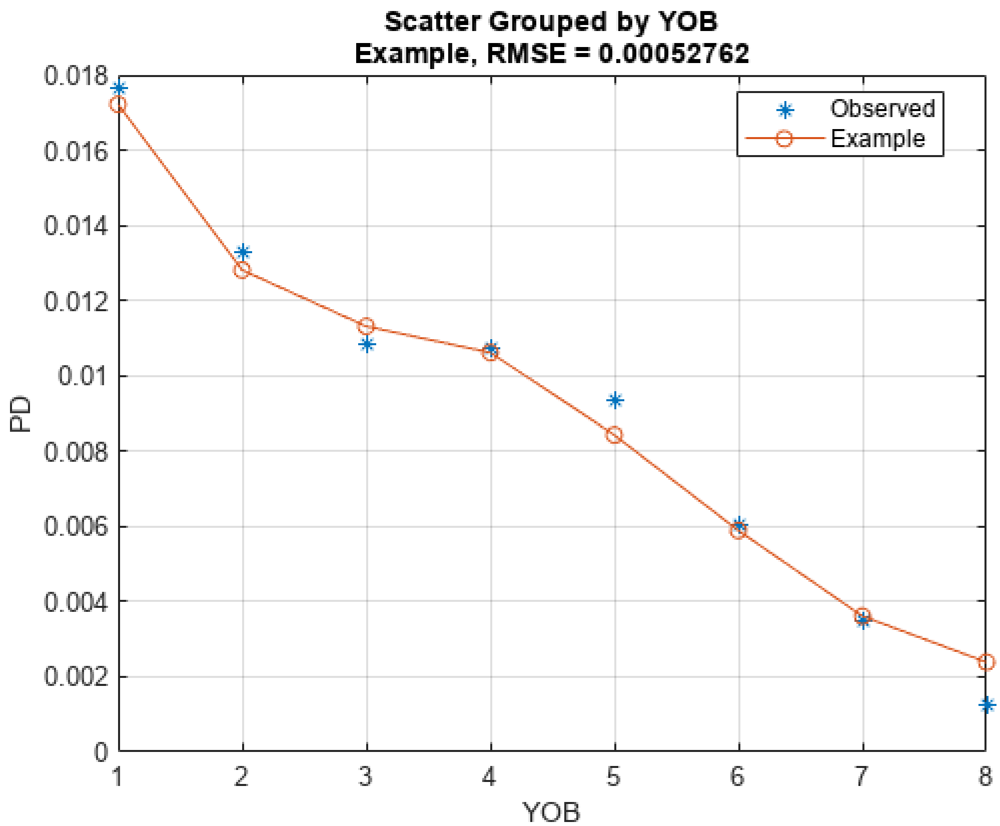
```
pdModel.UnderlyingModel
```

```
ans = 
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue   
                              __________    _________    _______    ___________

    (Intercept)                  -2.7422      0.10136    -27.054      3.408e-161
    ScoreGroup_Medium Risk      -0.68968     0.037286    -18.497      2.1894e-76
    ScoreGroup_Low Risk          -1.2587     0.045451    -27.693     8.4736e-169
    YOB                         -0.30894     0.013587    -22.738     1.8738e-114
    GDP                         -0.11111     0.039673    -2.8006       0.0051008
    Market                    -0.0083659    0.0028358    -2.9502       0.0031761


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Visualize Model Calibration

Use `modelCalibrationPlot` to visualize the model calibration on test data, grouping by age.

```
modelCalibrationPlot(pdModel,data(TestDataInd,:),'YOB')
```



Input Arguments

`pdModel` — Probability of default model
`Logistic` object | `Probit` object | `Cox` object | `customLifetimePDModel` object

Probability of default model, specified as a `Logistic`, `Probit`, or `Cox` object previously created using `fitLifetimePDModel`. Alternatively, you can create a custom probability of default model using `customLifetimePDModel`.

Note The `'ModelID'` property of the `pdModel` object is used as the identifier or tag for `pdModel`.

Data Types: `object`

`data` — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model.

Data Types: `table`

`GroupBy` — Name of column in `data` input used to group the data
string | character vector

Name of column in the `data` input used to group the data, specified as a string or character vector. `GroupBy` does not have to be a model variable name. For each group designated by `GroupBy`, the `modelCalibrationPlot` function computes the observed default rates and average predicted PDs are computed to measure the RMSE. `modelCalibrationPlot` supports up to two grouping variables.

Data Types: `string` | `char`

ax — Valid axis object
object

(Optional) Valid axis object, specified as an `ax` object that is created using `axes`. The plot will be created in the axes specified by the optional `ax` argument instead of in the current axes (gca). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:
`modelCalibrationPlot(pdModel,data(Ind,:),GroupBy=["YOB","ScoreGroup"],DataID="DataSetChoice")`

`DataID` — Data set identifier
`""` (default) | character vector | string

Data set identifier, specified as `DataID` and a character vector or string. `DataID` is included in the plot title for reporting purposes.

Data Types: `char` | `string`

`ReferencePD` — Conditional PD values predicted for `data` by reference model
`[]` (default) | numeric vector

Conditional PD values predicted for `data` by the reference model, specified as `ReferencePD` and a `NumRows`-by-`1` numeric vector. The predicted PD is plotted for both the `pdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as `ReferenceID` and a character vector or string. `ReferenceID` is used in the plot for reporting purposes.

Data Types: `char` | `string`

Output Arguments

h — Figure handle
handle object

Figure handle for the line objects, returned as handle object.

More About

Model Calibration

Model calibration measures the accuracy of the predicted probability of default (PD) values.

The `modelCalibrationPlot` function allows you to visually compare the predicted PD values to the observed default rates. The `modelCalibrationPlot` function requires a grouping variable to compute average predicted PD values within each group and the average observed default rate also within each group. The predicted PD values and the observed default rates by group are plotted against the grouping variable values.

Up to two grouping variables are supported in `modelCalibrationPlot`. When two grouping variables are specified, the average predicted PD and default rates are computed for all the groups defined by the combination of the two grouping variables. The data is plotted against the first grouping variable, and the second variable is used to differentiate the data on the plot with different colors.

The root mean square error (RMSE) of the grouped data is reported on the title of the plot. To get the RMSE metric programmatically, use `modelCalibration`.

Version History

Introduced in R2023a

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `predictLifetime` | `predict` | `fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

“Basic Lifetime PD Model Validation” on page 4-129
“Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
“Compare Lifetime PD Models Using Cross-Validation” on page 4-121
“Expected Credit Loss Computation” on page 4-124
“Compare Model Discrimination and Model Calibration to Validate of Probability of Default” on page 4-144

"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

modelAccuracyPlot

Plot observed default rates compared to predicted PDs on grouped data

Note `modelAccuracyPlot` is renamed to `modelCalibrationPlot`. `modelAccuracyPlot` is not recommended. Use `modelCalibrationPlot` instead.

Syntax

```
modelAccuracyPlot(pdModel,data,GroupBy)
modelAccuracyPlot( ___ ,Name,Value)
h = modelAccuracyPlot(ax, ___ ,Name,Value)
```

Description

`modelAccuracyPlot(pdModel,data,GroupBy)` plots the observed default rates compared to the predicted probabilities of default (PD). `GroupBy` is required and can be any column in the `data` input (not necessarily a model variable). The `modelAccuracyPlot` function computes the observed PD as the default rate of each group and the predicted PD as the average PD for each group. `modelAccuracyPlot` supports comparison against a reference model.

`modelAccuracyPlot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

`h = modelAccuracyPlot(ax, ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Input Arguments

`pdModel` — Probability of default model
`Logistic` object | `Probit` object | `Cox` object | `customLifetimePDModel` object

Probability of default model, specified as a `Logistic`, `Probit`, or `Cox` object previously created using `fitLifetimePDModel`. Alternatively, you can create a custom probability of default model using `customLifetimePDModel`.

Note The `'ModelID'` property of the `pdModel` object is used as the identifier or tag for `pdModel`.

Data Types: `object`

`data` — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model.

Data Types: `table`

GroupBy — Name of column in data input used to group the data
string | character vector

Name of column in the data input used to group the data, specified as a string or character vector. GroupBy does not have to be a model variable name. For each group designated by GroupBy, the modelAccuracyPlot function computes the observed default rates and average predicted PDs are computed to measure the RMSE. modelAccuracyPlot supports up to two grouping variables.

Data Types: string | char

ax — Valid axis object
object

(Optional) Valid axis object, specified as an ax object that is created using axes. The plot will be created in the axes specified by the optional ax argument instead of in the current axes (gca). The optional argument ax must precede any of the input argument combinations.

Data Types: object

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: modelAccuracyPlot(pdModel,data(Ind,:),'GroupBy',["YOB","ScoreGroup"],'DataID',"DataSetChoice")

DataID — Data set identifier
"" (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of 'DataID' and a character vector or string. DataID is included in the plot title for reporting purposes.

Data Types: char | string

ReferencePD — Conditional PD values predicted for data by reference model
[] (default) | numeric vector

Conditional PD values predicted for data by the reference model, specified as the comma-separated pair consisting of 'ReferencePD' and a NumRows-by-1 numeric vector. The predicted PD is plotted for both the pdModel object and the reference model.

Data Types: double

ReferenceID — Identifier for reference model
'Reference' (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of 'ReferenceID' and a character vector or string. ReferenceID is used in the plot for reporting purposes.

Data Types: char | string

Output Arguments

h — Figure handle
handle object

Figure handle for the line objects, returned as handle object.

More About

Model Accuracy

Model accuracy measures the accuracy of the predicted probability of default (PD) values.

The `modelAccuracyPlot` function allows you to visually compare the predicted PD values to the observed default rates. The `modelAccuracyPlot` function requires a grouping variable to compute average predicted PD values within each group and the average observed default rate also within each group. The predicted PD values and the observed default rates by group are plotted against the grouping variable values.

Up to two grouping variables are supported in `modelAccuracyPlot`. When two grouping variables are specified, the average predicted PD and default rates are computed for all the groups defined by the combination of the two grouping variables. The data is plotted against the first grouping variable, and the second variable is used to differentiate the data on the plot with different colors.

The root mean square error (RMSE) of the grouped data is reported on the title of the plot. To get the RMSE metric programmatically, use `modelAccuracy`.

Version History

Introduced in R2021a

R2023a: `modelAccuracyPlot` function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is not recommended, use `modelCalibrationPlot` instead.

R2022b: Support for `customLifetimePDModel` model

The `pdModel` input supports an option for a `customLifetimePDModel` model object that you can create using `customLifetimePDModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`modelDiscrimination` | `modelDiscriminationPlot` | `modelAccuracy` | `predictLifetime` | `predict` | `fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

modelDiscrimination

Compute AUROC and ROC data

Syntax

```
DiscMeasure = modelDiscrimination(eadModel,data)
[DiscMeasure,DiscData] = modelDiscrimination( ___ ,Name=Value)
```

Description

`DiscMeasure = modelDiscrimination(eadModel,data)` computes the area under the receiver operating characteristic curve (AUROC). `modelDiscrimination` supports segmentation and comparison against a reference model and alternative methods to discretize the EAD response into a binary variable.

`[DiscMeasure,DiscData] = modelDiscrimination( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax.

Examples

Compute AUROC and ROC Using Tobit EAD Model

This example shows how to use `fitEADModel` to create a `Tobit` model and then use `modelDiscrimination` to compute AUROC and ROC.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Tobit` or `Regression`.

```
ModelType = Tobit ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Tobit EAD Model

Use `fitEADModel` to create a `Tobit` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','M
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Tobit with properties:

        CensoringSide: "both"
            LeftLimit: 0
           RightLimit: 1
              ModelID: "Tobit"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LiimitVar'` and `'DrwanVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Tobit regression model:
     EAD_lcf = max(0,min(Y*,1))
     Y* ~ 1 + UtilizationRate + Age + Marriage

Estimated coefficients:
                             Estimate        SE         tStat       pValue  
                            __________    _________    ________    __________

    (Intercept)               0.22467       0.03134      7.1689    9.7855e-13
    UtilizationRate            0.4714      0.020722      22.749             0
    Age                    -0.0014209    0.00076326     -1.8616      0.062771
    Marriage_not married    -0.010542       0.01578    -0.66807       0.50415
    (Sigma)                    0.3618     0.0050022       72.33             0

Number of observations: 2627
Number of left-censored observations: 0
Number of uncensored observations: 2626
```

```
Number of right-censored observations: 1
Log-likelihood: -1057.9
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

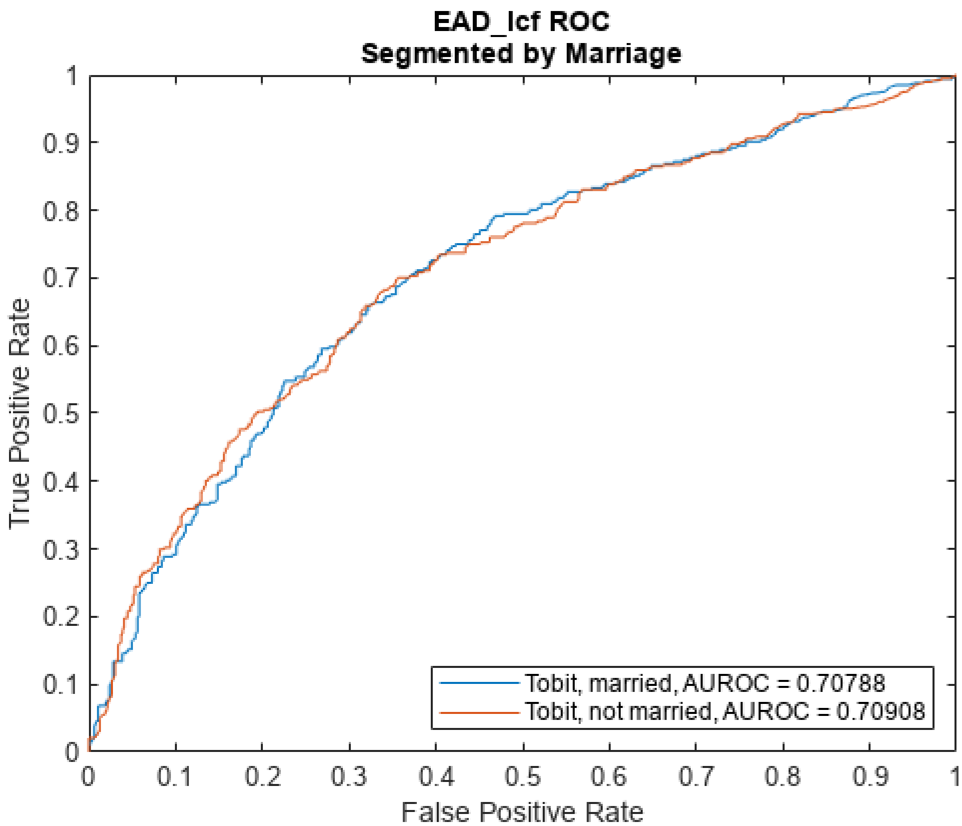
```
ModelLevel = ConversionMeasure ;
```

```
[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ShowDetails=true,Mode
```

```
DiscMeasure1=1×3 table
              AUROC      Segment      SegmentCount
             _______    __________    ____________

    Tobit    0.70893    "all_data"        1751
```

```
DiscData1=1534×3 table
        X             Y            T
    __________    _________    _______

             0            0    0.63602
             0    0.0027778    0.63602
             0    0.0041667    0.63489
    0.00096993    0.0055556    0.63377
    0.00096993    0.0069444    0.63265
     0.0019399    0.0083333    0.63152
     0.0029098    0.0097222     0.6304
     0.0029098     0.015278    0.62927
     0.0029098     0.016667    0.62922
     0.0029098     0.018056     0.6288
     0.0029098     0.019444    0.62864
     0.0038797     0.022222    0.62814
     0.0038797        0.025    0.62767
     0.0048497     0.026389    0.62701
     0.0048497     0.033333    0.62654
     0.0058196     0.033333    0.62618
      ⋮
```

```
modelDiscriminationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,SegmentBy="Marriage");
```



Compute AUROC and ROC Using Beta EAD Model

This example shows how to use `fitEADModel` to create a `Beta` model and then use `modelDiscrimination` to compute AUROC and ROC.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
```

```
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for a `Beta` model.

```
ModelType = Beta ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Beta EAD Model

Use `fitEADModel` to create a `Beta` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','M
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-07
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LiimitVar'` and `'DrwanVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Beta regression model:
     logit(EAD_lcf) ~ 1_mu + UtilizationRate_mu + Age_mu + Marriage_mu
     log(EAD_lcf) ~ 1_phi + UtilizationRate_phi + Age_phi + Marriage_phi

Estimated coefficients:
                                 Estimate        SE         tStat       pValue  
                                __________    _________    ________    __________

    (Intercept)_mu                -0.65566      0.11484     -5.7093    1.2616e-08
    UtilizationRate_mu              1.7014     0.078094      21.787             0
    Age_mu                      -0.0055901    0.0027603     -2.0252      0.042949
    Marriage_not married_mu      -0.012577     0.052098    -0.24141       0.80926
    (Intercept)_phi               -0.50131     0.094625     -5.2979    1.2686e-07
    UtilizationRate_phi            0.39731     0.066707       5.956    2.9303e-09
    Age_phi                      -0.001167    0.0023161    -0.50387        0.6144
```

```
    Marriage_not married_phi     -0.013275     0.042627    -0.31143        0.7555

Number of observations: 2627
Log-likelihood: -3140.21
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

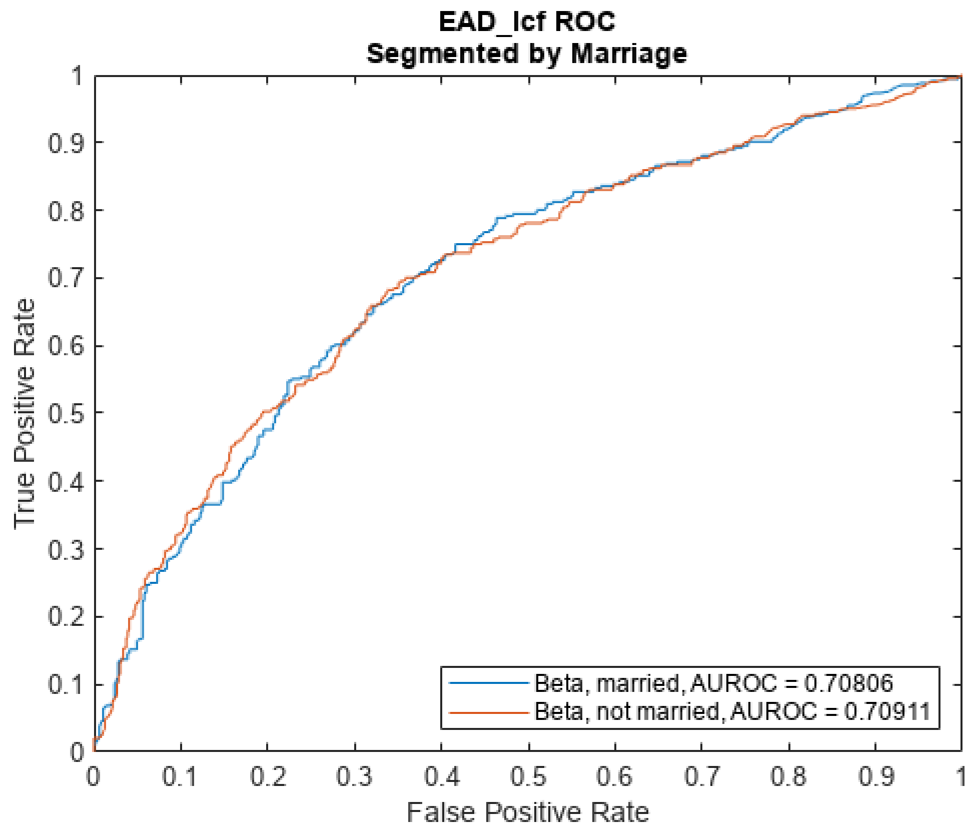
For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ConversionMeasure ;

[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ShowDetails=true,Mode
```

```
DiscMeasure1=1×3 table
             AUROC       Segment       SegmentCount
            _______    ___________    ____________

    Beta    0.70895    "all_data"         1751


DiscData1=1534×3 table
         X              Y            T
    __________    _________    _______

             0            0    0.71675
             0    0.0027778    0.71675
             0    0.0041667    0.71561
             0    0.0055556    0.71533
    0.00096993    0.0069444    0.71447
    0.00096993    0.0097222    0.71419
    0.00096993     0.011111    0.71333
    0.00096993     0.018056    0.71304
     0.0019399     0.018056     0.7128
     0.0029098     0.019444    0.71218
     0.0048497     0.019444     0.7119
     0.0058196     0.020833    0.71104
     0.0067895     0.020833    0.71075
     0.0067895     0.022222    0.71022
     0.0067895     0.027778    0.70989
     0.0067895     0.029167    0.70968
      ⋮
```

```
modelDiscriminationPlot(eadModel,EADData(TestInd, :),ModelLevel=ModelLevel,SegmentBy="Marriage")
```



Input Arguments

eadModel — Exposure at default model
Regression object | Tobit object | Beta object

Exposure at default model, specified as a previously created Regression, Tobit, or Beta object using fitEADModel.

Data Types: object

data — Data
table

Data, specified as a NumRows-by-NumCols table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: table

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: [DiscMeasure,DiscData] = modelDiscrimination(eadModel,data(TestInd,:),DataID='Testing',DiscretizeBy='median')

DataID — Data set identifier
"" (default) | character vector | string

Data set identifier, specified as DataID and a character vector or string. The DataID is included in the output for reporting purposes.

Data Types: char | string

DiscretizeBy — Discretization method for EAD data at defined ModelLevel
'mean' (default) | character vector with value 'mean' or 'median' | string with value "mean" or "median"

Discretization method for EAD data at the defined ModelLevel, specified as DiscretizeBy and a character vector or string.

- 'mean' — Discretized response is 1 if observed EAD is greater than or equal to the mean EAD, 0 otherwise.
- 'median' — Discretized response is 1 if observed EAD is greater than or equal to the median EAD, 0 otherwise.

Data Types: char | string

SegmentBy — Name of column in data input used to segment data set
"" (default) | character vector | string

Name of a column in the data input, not necessarily a model variable, to be used to segment the data set, specified as SegmentBy and a character vector or string. One AUROC is reported for each segment, and the corresponding ROC data for each segment is returned in the optional output.

Data Types: char | string

ShowDetails — Indicates if output includes columns showing segment value and segment count
false (default) | logical

Indicates if the output includes columns showing segment value and segment count, specified as the comma-separated pair consisting of 'ShowDetails' and a scalar logical.

Data Types: logical

ModelLevel — Model level
"ead" (default) | character vector with value 'ead', 'conversionMeasure', or 'conversionTransform' | string with value "ead", "conversionMeasure", or "conversionTransform"

Model level, specified as ModelLevel and a character vector or string.

Note Regression models support all three model levels, but a Tobit or Beta model supports only a ModelLevel for "ead" and "conversionMeasure".

Data Types: char | string

ReferenceEAD — EAD values predicted for data by reference model
[] (default) | numeric vector

EAD values predicted for data by the reference model, specified as ReferenceEAD and a NumRows-by-1 numeric vector. The modelDiscrimination output information is reported for both the eadModel object and the reference model.

Data Types: double

ReferenceID — Identifier for the reference model
'Reference' (default) | character vector | string

Identifier for the reference model, specified as ReferenceID and a character vector or string. 'ReferenceID' is used in the modelDiscrimination output for reporting purposes.

Data Types: char | string

Output Arguments

DiscMeasure — AUROC information for each model and each segment
table

AUROC information for each model and each segment, returned as a table. DiscMeasure has a single column named 'AUROC' and the number of rows depends on the number of segments and whether you use a ReferenceID for a reference model . The row names of DiscMeasure report the model IDs, segment, and data ID. If the optional ShowDetails name-value argument is true, the DiscMeasure output displays Segment and SegmentCount columns.

Note If you do not specify SegmentBy and use ShowDetails to request the segment details, the two columns are added and show the Segment column as "all_data" and the sample size (minus missing values) for the SegmentCount column.

DiscData — ROC data for each model and each segment
table

ROC data for each model and each segment, returned as a table. There are three columns for the ROC data, with column names 'X', 'Y', and 'T', where the first two are the X and Y coordinates of the ROC curve, and T contains the corresponding thresholds. For more information, see “Model Discrimination” on page 6-259 or perfcurve.

If you use SegmentBy, the function stacks the ROC data for all segments and DiscData has a column with the segmentation values to indicate where each segment starts and ends.

If reference model data is given, the DiscData outputs for the main and reference models are stacked, with an extra column 'ModelID' indicating where each model starts and ends.

More About

Model Discrimination

Model discrimination measures the risk ranking.

The modelDiscrimination function computes the area under the receiver operator characteristic (AUROC) curve, sometimes called simply the area under the curve (AUC). This metric is between 0 and 1 and higher values indicate better discrimination.

To compute the AUROC, you need a numeric prediction and a binary response. For EAD models, the predicted EAD is used directly as the prediction. However, the observed EAD must be discretized into a binary variable. By default, observed EAD values greater than or equal to the mean observed EAD are assigned a value of 1, and values below the mean are assigned a value of 0. This discretized response is interpreted as "high EAD" vs. "low EAD." Therefore, the `modelDiscrimination` function measures how well the predicted EAD separates the "high EAD" vs. the "low EAD" observations. You can change the level to compute the model discrimination with the `ModelLevel` name-value pair argument and the discretization criterion with the `DiscretizeBy` name-value pair argument.

To plot the receiver operator characteristic (ROC) curve, use the `modelDiscriminationPlot` function. However, if you need the ROC curve data, use the optional `DiscData` output argument from the `modelDiscrimination` function.

The ROC curve is a parametric curve that plots the proportion of

- High EAD cases with predicted EAD greater than or equal to a parameter *t*, or true positive rate (TPR)
- Low EAD cases with predicted EAD greater than or equal to the same parameter *t*, or false positive rate (FPR)

The parameter *t* sweeps through all the observed predicted EAD values for the given data. The `DiscData` optional output contains the TPR in the `'X'` column, the FPR in the `'Y'` column, and the corresponding parameters *t* in the `'T'` column. For more information about ROC curves, see “ROC Curve and Performance Metrics”.

Version History

Introduced in R2021b

R2022b: Support for `Beta` model

Behavior changed in R2022b

The `eadModel` input supports an option for a `Beta` model object that you can create using `fitEADModel`.

R2022a: Additional option for `ShowDetails`

Behavior changed in R2022a

There is an additional name-value pair for `ShowDetails` to indicate if the `DiscMeasure` output includes columns for `Segment` value and the `SegmentCount`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

`Regression` | `Tobit` | `fitEADModel` | `predict` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot`

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Exposure at Default Models" on page 1-34

modelDiscrimination

Compute AUROC and ROC data

Syntax

```
DiscMeasure = modelDiscrimination(lgdModel,data)
[DiscMeasure,DiscData] = modelDiscrimination( ___ ,Name,Value)
```

Description

`DiscMeasure = modelDiscrimination(lgdModel,data)` computes the area under the receiver operating characteristic curve (AUROC). `modelDiscrimination` supports segmentation and comparison against a reference model and also alternative methods to discretize the LGD response into a binary variable.

`[DiscMeasure,DiscData] = modelDiscrimination( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Compute AUROC and ROC Using a Regression LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Regression` model and then use `modelDiscrimination` to compute AUROC and ROC.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV         Age          Type           LGD   
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);
```

```
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create a Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using training data. You can also use `fitLGDModel` to create a `Tobit` model by changing the `lgdModel` input argument to `'Tobit'`.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'Regression');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Regression"
          Description: ""
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

Compute AUROC and ROC Data

Use `modelDiscrimination` to compute the AUROC and ROC for the test data set.

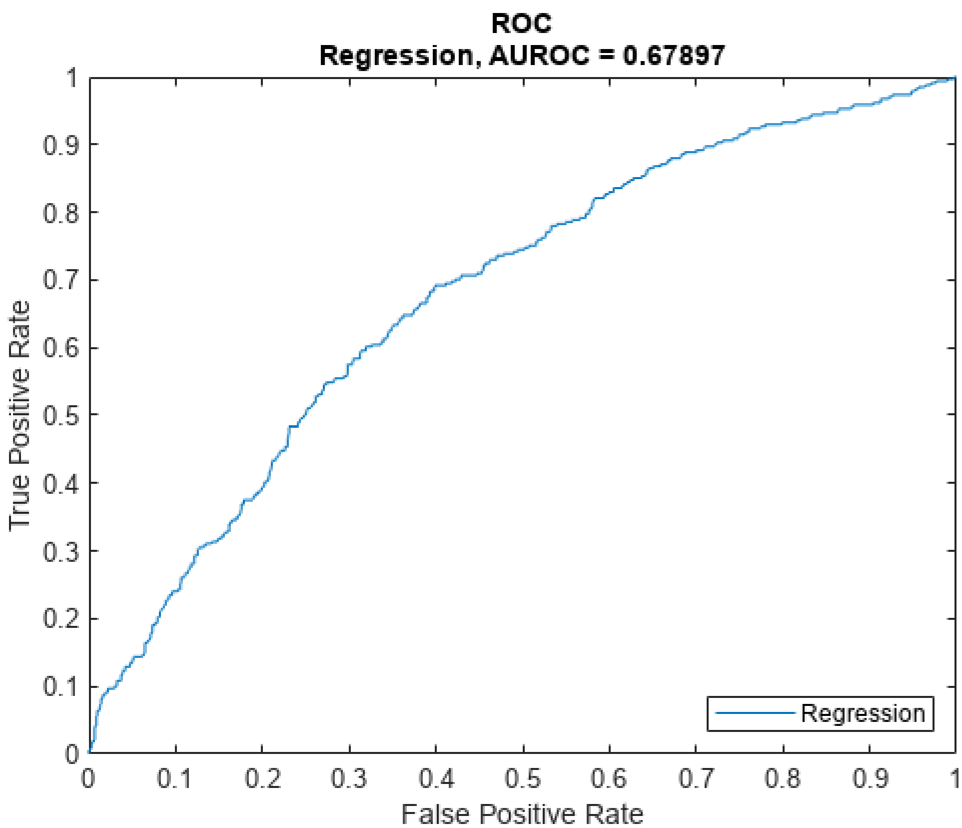
```
[DiscMeasure,DiscData] = modelDiscrimination(lgdModel,data(TestInd,:),'ShowDetails',true)
```

```
DiscMeasure=1×3 table
                   AUROC      Segment      SegmentCount
                  _______    __________    ____________

    Regression    0.67897    "all_data"        1394    


DiscData=1395×3 table
        X             Y           T
```

```
    ___________    __________    ________

              0             0     0.87604
              0     0.0029326     0.87604
              0     0.0058651      0.7515
     0.00094967     0.0058651     0.44074
      0.0018993     0.0058651     0.43569
      0.0018993     0.0087977     0.40058
       0.002849     0.0087977     0.31703
       0.002849       0.01173     0.30375
       0.002849      0.014663     0.28789
       0.002849      0.017595     0.27996
      0.0037987      0.017595     0.27026
      0.0047483      0.017595     0.26868
       0.005698      0.017595     0.26854
       0.005698      0.020528     0.26682
      0.0066477      0.020528     0.26668
      0.0066477       0.02346     0.24923
        :
```

You can visualize the ROC data using `modelDiscriminationPlot`.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:))
```



Compute AUROC and ROC Using Tobit LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Tobit` model and then use `modelDiscrimination` to compute AUROC and ROC.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD   
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create a Tobit LGD Model

Use `fitLGDModel` to create a `Tobit` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'tobit');
disp(lgdModel)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Tobit regression model:
     LGD = max(0,min(Y*,1))
```

```
     Y* ~ 1 + LTV + Age + Type

Estimated coefficients:
                       Estimate        SE         tStat       pValue  
                       _________    _________    _______    __________

    (Intercept)         0.058257     0.027265     2.1367      0.032737
    LTV                  0.20126     0.031354     6.4189    1.6932e-10
    Age                -0.095407    0.0072653    -13.132             0
    Type_investment      0.10208     0.018058     5.6531    1.7915e-08
    (Sigma)              0.29288    0.0057036      51.35             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1521
Number of right-censored observations: 25
Log-likelihood: -698.383
```

Compute AUROC and ROC Data

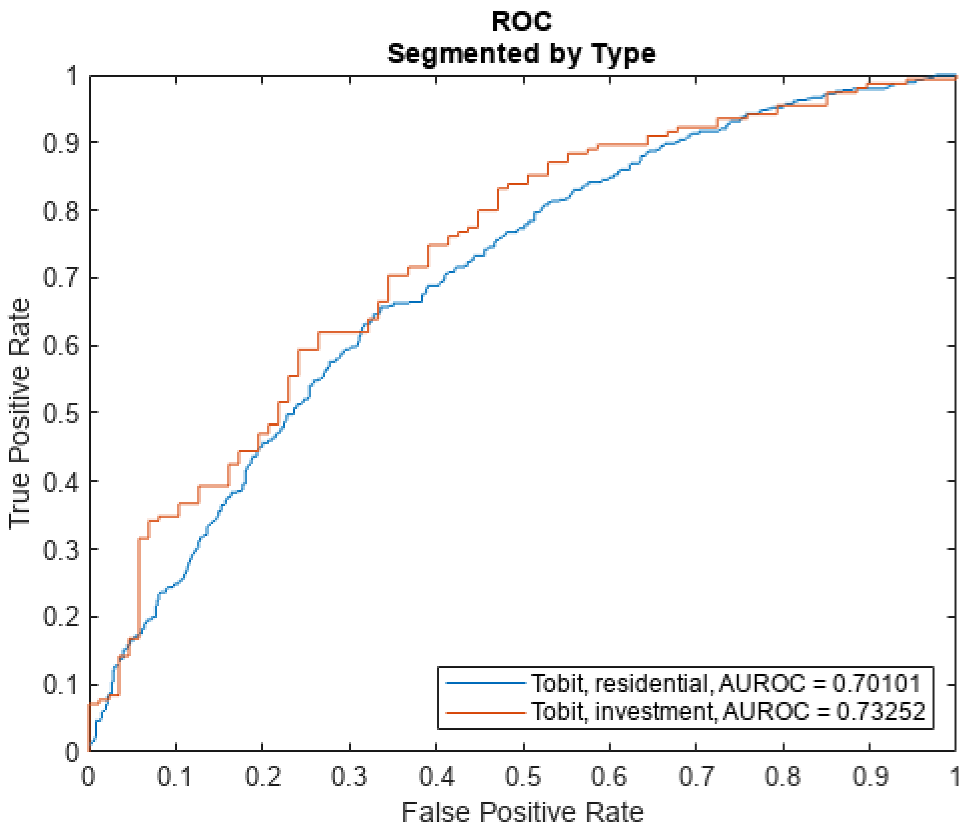
Use `modelDiscrimination` to compute the AUROC and ROC for the test data set.

```
DiscMeasure = modelDiscrimination(lgdModel,data(TestInd,:),'ShowDetails',true,'SegmentBy',"Type"
```

```
DiscMeasure=2×3 table
                                 AUROC        Segment        SegmentCount
                                _______    _____________    ____________

    Tobit, Type=residential     0.70101    "residential"        1152    
    Tobit, Type=investment      0.73252    "investment"          242    
```

You can visualize the ROC using `modelDiscriminationPlot`.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:),'SegmentBy',"Type",'DiscretizeBy',"median")
```



Compute AUROC and ROC Using Beta LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Beta` model and then use `modelDiscrimination` to compute AUROC and ROC.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create a Beta LGD Model

Use `fitLGDModel` to create a risk_ug#object_model_beta_lgd model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'Beta');
disp(lgdModel)
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-05
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Beta regression model:
     logit(LGD) ~ 1_mu + LTV_mu + Age_mu + Type_mu
     log(LGD) ~ 1_phi + LTV_phi + Age_phi + Type_phi

Estimated coefficients:
                           Estimate       SE        tStat       pValue  
                           ________    ________    _______    __________

    (Intercept)_mu          -1.3772     0.13201    -10.433             0
    LTV_mu                  0.60269     0.15087     3.9947    6.7023e-05
    Age_mu                 -0.47464    0.040264    -11.788             0
    Type_investment_mu      0.45372    0.085143     5.3289     1.094e-07
    (Intercept)_phi        -0.16337     0.12591    -1.2975       0.19462
    LTV_phi                0.055892     0.14719    0.37973       0.70419
    Age_phi                 0.22887    0.040335     5.6743    1.5863e-08
    Type_investment_phi    -0.14102    0.078155    -1.8044      0.071311

Number of observations: 2093
Log-likelihood: -5291.04
```

Compute AUROC and ROC Data

Use `modelDiscrimination` to compute the AUROC and ROC for the test data set.

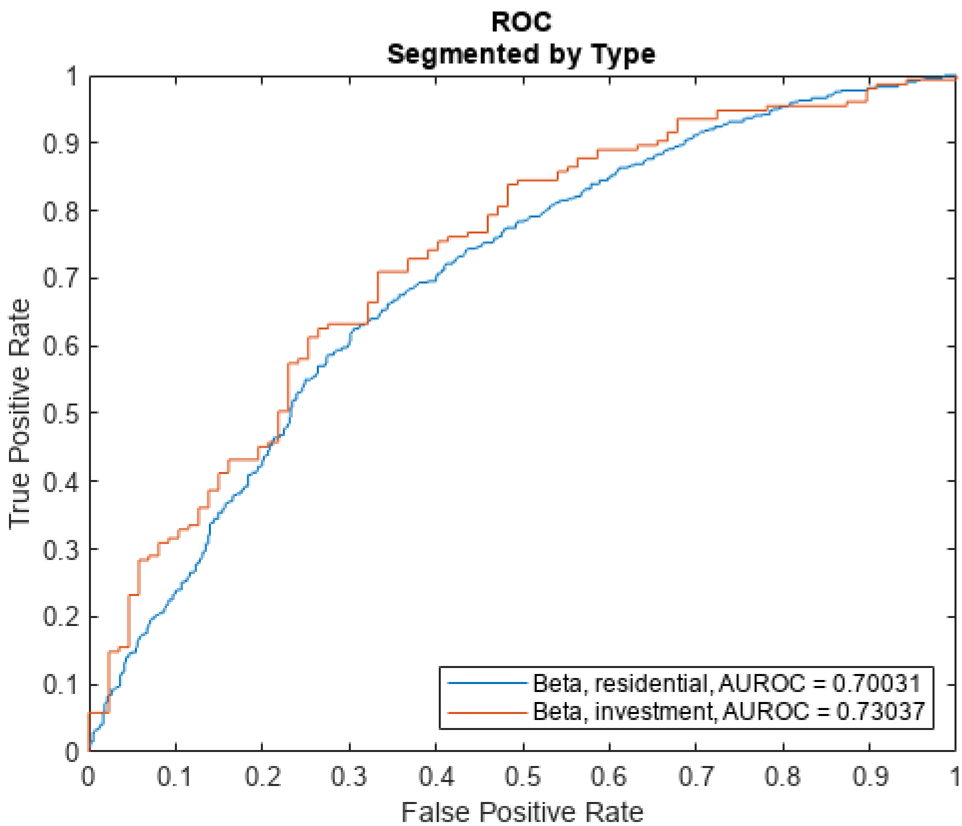
```
DiscMeasure = modelDiscrimination(lgdModel,data(TestInd,:),'ShowDetails',true,'SegmentBy',"Type"
```

```
DiscMeasure=2×3 table
                              AUROC         Segment        SegmentCount
```

```
                              _______    _____________    ____________

    Beta, Type=residential    0.70031    "residential"        1152
    Beta, Type=investment     0.73037    "investment"          242
```

You can visualize the ROC using `modelDiscriminationPlot`.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:),'SegmentBy',"Type",'DiscretizeBy',"median")
```



Input Arguments

`lgdModel` — Loss given default model
`Regression` object | `Tobit` object | `Beta` object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitLGDModel`.

Data Types: `object`

`data` — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[DiscMeasure,DiscData] = modelDiscrimination(lgdModel,data(TestInd,:),'DataID','Testing','DiscretizeBy','median')`

`DataID` — Data set identifier

`""` (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of `'DataID'` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`DiscretizeBy` — Discretization method for LGD `data`

`'mean'` (default) | character vector with value `'mean'`, `'median'`, `'positive'`, or `'total'` | string with value `"mean"`, `"median"`, `"positive"`, or `"total"`

Discretization method for LGD `data`, specified as the comma-separated pair consisting of `'DiscretizeBy'` and a character vector or string.

- `'mean'` — Discretized response is `1` if observed LGD is greater than or equal to the mean LGD, `0` otherwise.
- `'median'` — Discretized response is `1` if observed LGD is greater than or equal to the median LGD, `0` otherwise.
- `'positive'` — Discretized response is `1` if observed LGD is positive, `0` otherwise (full recovery).
- `'total'` — Discretized response is `1` if observed LGD is greater than or equal to `1` (total loss), `0` otherwise.

Data Types: `char` | `string`

`SegmentBy` — Name of column in `data` input used to segment data set

`""` (default) | character vector | string

Name of a column in the `data` input, not necessarily a model variable, to be used to segment the data set, specified as the comma-separated pair consisting of `'SegmentBy'` and a character vector or string. One AUROC is reported for each segment, and the corresponding ROC data for each segment is returned in the optional output.

Data Types: `char` | `string`

`ShowDetails` — Indicates if output includes columns showing segment value and segment count

`false` (default) | logical

Indicates if the output includes columns showing segment value and segment count, specified as the comma-separated pair consisting of `'ShowDetails'` and a scalar logical.

Data Types: `logical`

ReferenceLGD — LGD values predicted for data by reference model
[ ] (default) | numeric vector

LGD values predicted for data by the reference model, specified as the comma-separated pair consisting of 'ReferenceLGD' and a NumRows-by-1 numeric vector. The modelDiscrimination output information is reported for both the lgdModel object and the reference model.

Data Types: double

ReferenceID — Identifier for the reference model
'Reference' (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of 'ReferenceID' and a character vector or string. 'ReferenceID' is used in the modelDiscrimination output for reporting purposes.

Data Types: char | string

Output Arguments

DiscMeasure — AUROC information for each model and each segment
table

AUROC information for each model and each segment, returned as a table. DiscMeasure has a single column named 'AUROC' and the number of rows depends on the number of segments and whether you use a ReferenceID for a reference model . The row names of DiscMeasure report the model IDs, segment, and data ID. If the optional ShowDetails name-value argument is true, the DiscMeasure output displays Segment and SegmentCount columns.

Note If you do not specify SegmentBy and use ShowDetails to request the segment details, the two columns are added and show the Segment column as "all_data" and the sample size (minus missing values) for the SegmentCount column.

DiscData — ROC data for each model and each segment
table

ROC data for each model and each segment, returned as a table. There are three columns for the ROC data, with column names 'X', 'Y', and 'T', where the first two are the X and Y coordinates of the ROC curve, and T contains the corresponding thresholds. For more information, see “Model Discrimination” on page 6-271 or perfcurve.

If you use SegmentBy, the function stacks the ROC data for all segments and DiscData has a column with the segmentation values to indicate where each segment starts and ends.

If reference model data is given, the DiscData outputs for the main and reference models are stacked, with an extra column 'ModelID' indicating where each model starts and ends.

More About

Model Discrimination

Model discrimination measures the risk ranking.

The `modelDiscrimination` function computes the area under the receiver operator characteristic (AUROC) curve, sometimes called simply the area under the curve (AUC). This metric is between 0 and 1 and higher values indicate better discrimination.

To compute the AUROC, you need a numeric prediction and a binary response. For loss given default (LGD) models, the predicted LGD is used directly as the prediction. However, the observed LGD must be discretized into a binary variable. By default, observed LGD values greater than or equal to the mean observed LGD are assigned a value of 1, and values below the mean are assigned a value of 0. This discretized response is interpreted as "high LGD" vs. "low LGD." Therefore, the `modelDiscrimination` function measures how well the predicted LGD separates the "high LGD" vs. the "low LGD" observations. You can change the discretization criterion with the `DiscretizeBy` name-value pair argument.

To plot the receiver operator characteristic (ROC) curve, use the `modelDiscriminationPlot` function. However, if the ROC curve data is needed, use the optional `DiscData` output argument from the `modelDiscrimination` function.

The ROC curve is a parametric curve that plots the proportion of

- High LGD cases with predicted LGD greater than or equal to a parameter *t*, or true positive rate (TPR)
- Low LGD cases with predicted LGD greater than or equal to the same parameter *t*, or false positive rate (FPR)

The parameter *t* sweeps through all the observed predicted LGD values for the given data. The `DiscData` optional output contains the TPR in the `'X'` column, the FPR in the `'Y'` column, and the corresponding parameters *t* in the `'T'` column. For more information about ROC curves, see “ROC Curve and Performance Metrics”.

Version History

Introduced in R2021a

R2022b: Support for `Beta` model

Behavior changed in R2022b

The `lgdModel` input supports an option for a `Beta` model object that you can create using `fitLGDModel`.

R2022a: Additional option for `ShowDetails`

Behavior changed in R2022a

There is an additional name-value pair for `ShowDetails` to indicate if the `DiscMeasure` output includes columns for `Segment` value and the `SegmentCount`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

`Tobit` | `Regression` | `modelCalibration` | `modelCalibartionPlot` | `modelDiscriminationPlot` | `predict` | `fitLGDModel`

Topics

"Model Loss Given Default" on page 4-90
"Basic Loss Given Default Model Validation" on page 4-131
"Compare Tobit LGD Model to Benchmark Model" on page 4-133
"Compare Loss Given Default Models Using Cross-Validation" on page 4-140
"Overview of Loss Given Default Models" on page 1-31

modelDiscriminationPlot

Plot ROC curve

Syntax

```
modelDiscriminationPlot(pdModel,data)
modelDiscriminationPlot( ___ ,Name,Value)
h = modelDiscriminationPlot(ax, ___ ,Name,Value)
```

Description

`modelDiscriminationPlot(pdModel,data)` plots the receiver operating characteristic curve (ROC). `modelDiscriminationPlot` supports segmentation and comparison against a reference model.

`modelDiscriminationPlot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

`h = modelDiscriminationPlot(ax, ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Examples

Plot ROC Curve

This example shows how to use `modelDiscriminationPlot` to plot the ROC curve.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______
```

```
    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % For reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create Logistic Lifetime PD Model

Use `fitLifetimePDModel` to create a `Logistic` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),'logistic',...
        'ModelID','Example',...
        'Description','Lifetime PD model using RetailCreditPanelData.',...
        'IDVar','ID',...
        'AgeVar','YOB',...
        'LoanVars','ScoreGroup',...
        'MacroVars',{'GDP' 'Market'},...
        'ResponseVar','Default');
 disp(pdModel)
```

```
  Logistic with properties:
```

```
            ModelID: "Example"
        Description: "Lifetime PD model using RetailCreditPanelData."
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

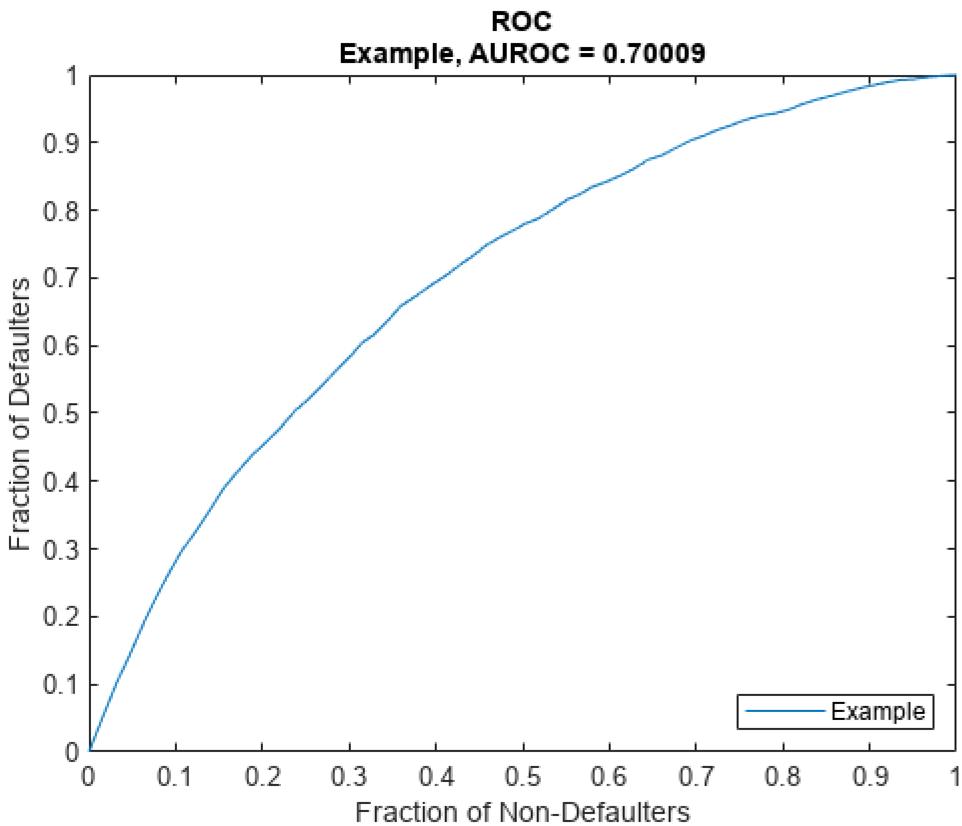
```
disp(pdModel.UnderlyingModel)
```

```
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue
                              __________    _________    _______    __________

    (Intercept)                  -2.7422      0.10136    -27.054     3.408e-161
    ScoreGroup_Medium Risk      -0.68968     0.037286    -18.497     2.1894e-76
    ScoreGroup_Low Risk          -1.2587     0.045451    -27.693    8.4736e-169
    YOB                         -0.30894     0.013587    -22.738    1.8738e-114
    GDP                         -0.11111     0.039673    -2.8006      0.0051008
    Market                    -0.0083659    0.0028358    -2.9502      0.0031761


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Visualize Model Discrimination

Use `modelDiscriminationPlot` to plot the ROC for the test data.

```
modelDiscriminationPlot(pdModel,data(TestDataInd,:))
```



Input Arguments

pdModel — Probability of default model
Logistic object | Probit object | Cox object | customLifetimePDModel object

Probability of default model, specified as a Logistic, Probit, or Cox object previously created using fitLifetimePDModel. Alternatively, you can create a custom probability of default model using customLifetimePDModel.

Note The 'ModelID' property of the pdModel object is used as the identifier or tag for pdModel.

Data Types: object

data — Data
table

Data, specified as a NumRows-by-NumCols table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model.

Data Types: table

ax — Valid axis object
object

(Optional) Valid axis object, specified as an `ax` object that is created using `axes`. The plot will be created in the axes specified by the optional `ax` argument instead of in the current axes (gca). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `modelDiscriminationPlot(pdModel,data(Ind,:),'DataID',"DataSetChoice")`

`DataID` — Data set identifier

`""` (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of `'DataID'` and a character vector or string. The `DataID` is included in the plot title for reporting purposes.

Data Types: `char` | `string`

`SegmentBy` — Name of column in `data` input used to segment data set

`""` (default) | character vector | string

Name of a column in the data input, not necessarily a model variable, to be used to segment the data set, specified as the comma-separated pair consisting of `'SegmentBy'` and a character vector or string. `modelDiscriminationPlot` plots one ROC for each segment.

Data Types: `char` | `string`

`ReferencePD` — Conditional PD values predicted for `data` by reference model

`[ ]` (default) | numeric vector

Conditional PD values predicted for `data` by the reference model, specified as the comma-separated pair consisting of `'ReferencePD'` and a `NumRows`-by-`1` numeric vector. The ROC curve output information is plotted for both the `pdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for reference model

`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of `'ReferenceID'` and a character vector or string. `'ReferenceID'` is used in the plot for reporting purposes.

Data Types: `char` | `string`

Output Arguments

`h` — Figure handle

handle object

Figure handle for the line objects, returned as handle object.

More About

Model Discrimination

Model discrimination measures the risk ranking.

Higher-risk loans should get higher predicted probability of default (PD) than lower-risk loans. The `modelDiscrimination` function computes the area under the receiver operator characteristic curve (AUROC), sometimes called simply the area under the curve (AUC). This metric is between 0 and 1 and higher values indicate better discrimination.

The receiver operator characteristic (ROC) curve is a parametric curve that plots the proportion of

- Defaulters with PD higher than or equal to a reference PD value *p*
- Nondefaulters with PD higher than or equal to the same reference PD value *p*

The reference PD value *p* parametizes the curve, and the software sweeps through the unique predicted PD values observed in a data set. The proportion of actual defaulters are assigned a PD higher than or equal to *p* is the true positive rate. The proportion of actual nondefaulters that are assigned a PD higher than or equal to *p* is the false positive rate." For more information about ROC curves, see “ROC Curve and Performance Metrics”.

The AUROC is reported on the plot created by `modelDiscriminationPlot`. To get the AUROC metric programmatically, use `modelDiscrimination`.

Version History

Introduced in R2021a

R2022b: Support for `customLifetimePDModel` model

The `pdModel` input supports an option for a `customLifetimePDModel` model object that you can create using `customLifetimePDModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`predictLifetime` | `predict` | `modelDiscrimination` | `modelCalibration` | `modelCalibrationPlot` | `fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

modelDiscrimination

Compute AUROC and ROC data

Syntax

```
DiscMeasure = modelDiscrimination(pdModel,data)
[DiscMeasure,DiscData] = modelDiscrimination( ___ ,Name,Value)
```

Description

`DiscMeasure = modelDiscrimination(pdModel,data)` computes the area under the receiver operating characteristic curve (AUROC). `modelDiscrimination` supports segmentation and comparison against a reference model.

`[DiscMeasure,DiscData] = modelDiscrimination( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Generate AUROC and ROC for Logistic Lifetime PD Model

This example shows how to use `fitLifetimePDModel` to fit data with a `Logistic` model and then generate the area under the receiver operating characteristic curve (AUROC) and ROC curve.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
```

```
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

ID	ScoreGroup	YOB	Default	Year	GDP	Market
1	Low Risk	1	0	1997	2.72	7.61
1	Low Risk	2	0	1998	3.57	26.24
1	Low Risk	3	0	1999	2.86	18.1
1	Low Risk	4	0	2000	2.43	3.19
1	Low Risk	5	0	2001	1.26	-10.51
1	Low Risk	6	0	2002	-0.59	-22.95
1	Low Risk	7	0	2003	0.63	2.78
1	Low Risk	8	0	2004	1.85	9.48

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Logistic Lifetime PD Model

Use `fitLifetimePDModel` to create a `Logistic` model.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Logistic",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
 disp(pdModel)
```

```
  Logistic with properties:

            ModelID: "Logistic"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
```

```
          LoanVars: "ScoreGroup"
         MacroVars: ["GDP"    "Market"]
       ResponseVar: "Default"
```

Display the underlying model.

```
pdModel.UnderlyingModel
```

```
ans = 
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue   
                              __________    _________    _______    ___________

    (Intercept)                  -2.7422      0.10136    -27.054      3.408e-161
    ScoreGroup_Medium Risk      -0.68968     0.037286    -18.497      2.1894e-76
    ScoreGroup_Low Risk          -1.2587     0.045451    -27.693     8.4736e-169
    YOB                         -0.30894     0.013587    -22.738     1.8738e-114
    GDP                         -0.11111     0.039673    -2.8006       0.0051008
    Market                    -0.0083659    0.0028358    -2.9502       0.0031761


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

```
pdModel.UnderlyingModel.Coefficients
```

```
ans=6×4 table
                               Estimate        SE         tStat       pValue   
                              __________    _________    _______    ___________

    (Intercept)                  -2.7422      0.10136    -27.054      3.408e-161
    ScoreGroup_Medium Risk      -0.68968     0.037286    -18.497      2.1894e-76
    ScoreGroup_Low Risk          -1.2587     0.045451    -27.693     8.4736e-169
    YOB                         -0.30894     0.013587    -22.738     1.8738e-114
    GDP                         -0.11111     0.039673    -2.8006       0.0051008
    Market                    -0.0083659    0.0028358    -2.9502       0.0031761
```

Model Discrimination to Generate AUROC and ROC

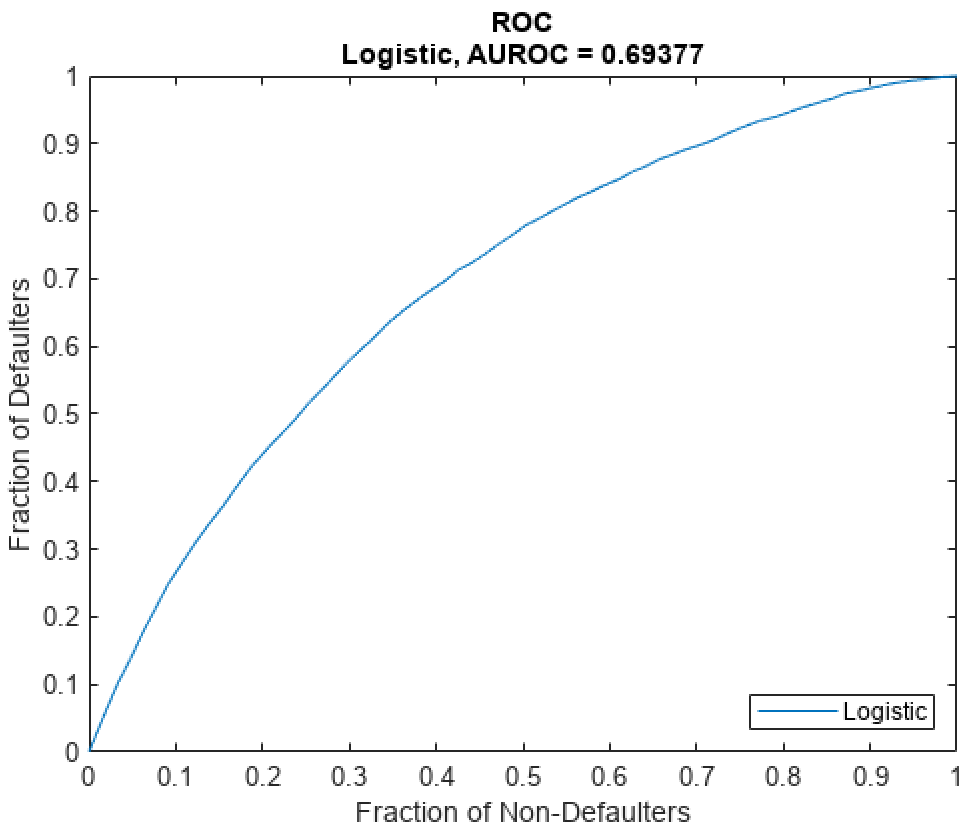
Model "discrimination" measures how effectively a model ranks customers by risk. You can use the AUROC and ROC outputs to determine whether customers with higher predicted PDs actually have higher risk in the observed data.

```
DataSetChoice = Training;
if DataSetChoice=="Training"
    Ind = TrainDataInd;
 else
    Ind = TestDataInd;
 end

DiscMeasure = modelDiscrimination(pdModel,data(TrainDataInd,:),'ShowDetails',true,'DataID',DataSe
disp(DiscMeasure)
```

	AUROC	Segment	SegmentCount
Logistic, Training	0.69377	"all_data"	3.881e+05

Visualize the ROC for the `Logistic` model using `modelDiscriminationPlot`.

```
modelDiscriminationPlot(pdModel,data(TrainDataInd,:));
```



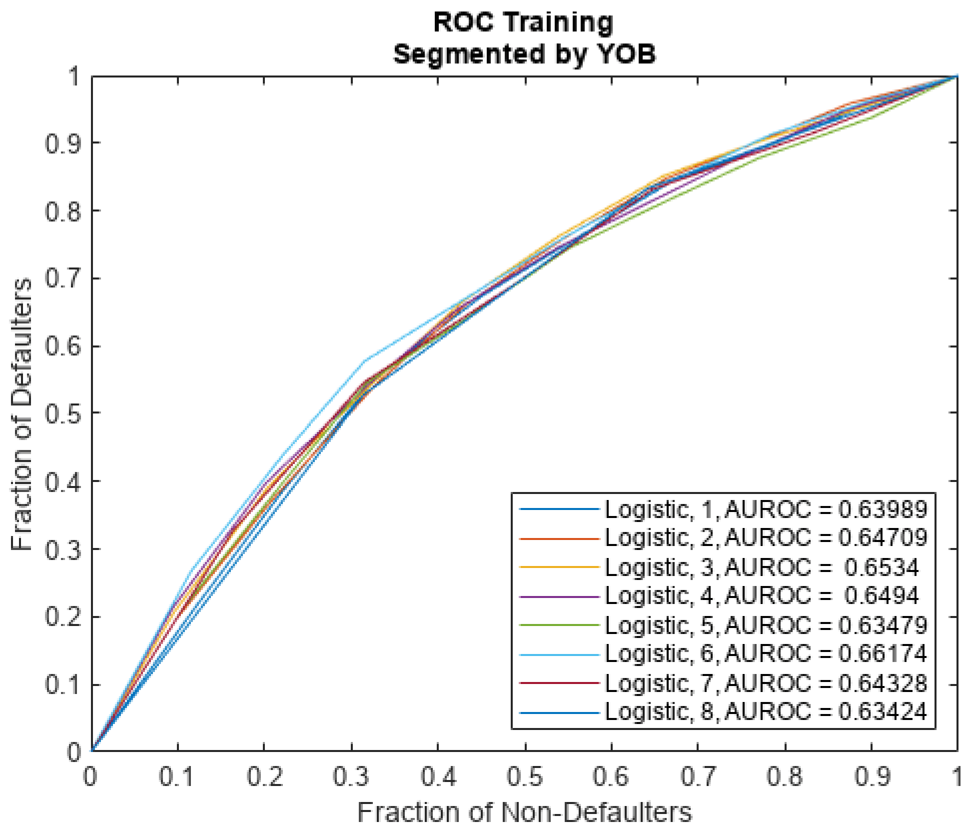
Data can be segmented to get the AUROC per segment and the corresponding ROC data.

```
SegmentVar = YOB ;
DiscMeasure = modelDiscrimination(pdModel,data(Ind,:),'ShowDetails',true,'SegmentBy',SegmentVar,
disp(DiscMeasure)
```

	AUROC	Segment	SegmentCount
Logistic, YOB=1, Training	0.63989	1	58092
Logistic, YOB=2, Training	0.64709	2	56723
Logistic, YOB=3, Training	0.6534	3	55524
Logistic, YOB=4, Training	0.6494	4	54650
Logistic, YOB=5, Training	0.63479	5	53770
Logistic, YOB=6, Training	0.66174	6	53186
Logistic, YOB=7, Training	0.64328	7	36959
Logistic, YOB=8, Training	0.63424	8	19193

Visualize the ROC segmented by `YOB`, `ScoreGroup`, or `Year` using `modelDiscriminationPlot`.

```
modelDiscriminationPlot(pdModel,data(Ind,:),'SegmentBy',SegmentVar,'DataID',DataSetChoice);
```



Input Arguments

pdModel — Probability of default model
`Logistic` object | `Probit` object | `Cox` object | `customLifetimePDModel` object

Probability of default model, specified as a `Logistic`, `Probit`, or `Cox` object previously created using `fitLifetimePDModel`. Alternatively, you can create a custom probability of default model using `customLifetimePDModel`.

Note The `'ModelID'` property of the `pdModel` object is used as the identifier or tag for `pdModel`.

Data Types: `object`

data — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[PerfMeasure,PerfData] = modelDiscrimination(pdModel,data(Ind,:),'DataID',"DataSetChoice")`

`DataID` — Data set identifier

`""` (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of `'DataID'` and a character vector or string.

Data Types: `char` | `string`

`SegmentBy` — Name of column in `data` input used to segment data set

`""` (default) | character vector | string

Name of a column in the data input, not necessarily a model variable, to be used to segment the data set, specified as the comma-separated pair consisting of `'SegmentBy'` and a character vector or string.

One AUROC value is reported for each segment and the corresponding ROC data for each segment is returned in the `PerfData` optional output.

Data Types: `char` | `string`

`ShowDetails` — Indicates if output includes columns showing segment value and segment count

`false` (default) | logical

Indicates if the output includes columns showing segment value and segment count, specified as the comma-separated pair consisting of `'ShowDetails'` and a scalar logical.

Data Types: `logical`

`ReferencePD` — Conditional PD values predicted for `data` by reference model

`[ ]` (default) | numeric vector

Conditional PD values predicted for `data` by the reference model, specified as the comma-separated pair consisting of `'ReferencePD'` and a `NumRows`-by-`1` numeric vector. The `modelDiscrimination` output information is reported for both the `pdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for reference model

`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of `'ReferenceID'` and a character vector or string. `'ReferenceID'` is used in the `modelDiscrimination` output for reporting purposes.

Data Types: `char` | `string`

Output Arguments

`DiscMeasure` — AUROC information for each model and each segment
table

AUROC information for each model and each segment., returned as a table. `DiscMeasure` has a single column named `'AUROC'` and the number of rows depends on the number of segments and whether you use a `ReferenceID` for a reference model and `ReferencePD` for reference data. The row names of `DiscMeasure` report the model IDs, segment, and data ID. If the optional `ShowDetails` name-value argument is `true`, the `DiscMeasure` output displays `Segment` and `SegmentCount` columns.

Note If you do not specify `SegmentBy` and use `ShowDetails` to request the segment details, the two columns are added and show the `Segment` column as `"all_data"` and the sample size (minus missing values) for the `SegmentCount` column.

`DiscData` — ROC data for each model and each segment
table

ROC data for each model and each segment, returned as a table. There are three columns for the ROC data, with column names `'X'`, `'Y'`, and `'T'`, where the first two are the X and Y coordinates of the ROC curve, and T contains the corresponding thresholds.

If you use `SegmentBy`, the function stacks the ROC data for all segments and `DiscData` has a column with the segmentation values to indicate where each segment starts and ends.

If reference model data is given using `ReferenceID` and `ReferencePD`, the `DiscData` outputs for the main and reference models are stacked, with an extra column `'ModelID'` indicating where each model starts and ends.

More About

Model Discrimination

Model discrimination measures the risk ranking.

Higher-risk loans should get higher predicted probability of default (PD) than lower-risk loans. The `modelDiscrimination` function computes the Area Under the Receiver Operator Characteristic curve (AUROC), sometimes called simply the Area Under the Curve (AUC). This metric is between 0 and 1 and higher values indicate better discrimination.

For more information about the Receiver Operator Characteristic (ROC) curve, see “Model Discrimination” on page 6-279 and “ROC Curve and Performance Metrics”.

Version History

Introduced in R2020b

R2022b: Support for `customLifetimePDModel` model

The pdModel input supports an option for a customLifetimePDModel model object that you can create using customLifetimePDModel.

R2022a: Additional option for ShowDetails

There is an additional name-value pair for ShowDetails to indicate if the DiscMeasure output includes columns for Segment value and the SegmentCount.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

predictLifetime | predict | modelDiscriminationPlot | modelCalibration | modelCalibrationPlot | fitLifetimePDModel | Logistic | Probit | Cox | customLifetimePDModel

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

modelDiscriminationPlot

Plot ROC curve

Syntax

```
modelDiscriminationPlot(eadModel,data)
modelDiscriminationPlot( ___ ,Name=Value)
h = modelDiscriminationPlot(ax, ___ ,Name=Value)
```

Description

`modelDiscriminationPlot(eadModel,data)` generates the receiver operating characteristic (ROC) curve. `modelDiscriminationPlot` supports segmentation and comparison against a reference model.

`modelDiscriminationPlot( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax.

`h = modelDiscriminationPlot(ax, ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Examples

Plot ROC Using a Tobit EAD Model

This example shows how to use `fitEADModel` to create a `Tobit` model and then use `modelDiscriminationPlot` to plot the ROC.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
```

```
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Tobit` or `Regression`.

```
ModelType = Tobit ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Tobit EAD Model

Use `fitEADModel` to create a `Tobit` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','M
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Tobit with properties:

        CensoringSide: "both"
            LeftLimit: 0
           RightLimit: 1
              ModelID: "Tobit"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Tobit regression model:
     EAD_lcf = max(0,min(Y*,1))
     Y* ~ 1 + UtilizationRate + Age + Marriage

Estimated coefficients:
                             Estimate        SE         tStat       pValue   
                            __________    _________    ________    __________

    (Intercept)               0.22467       0.03134      7.1689    9.7855e-13
    UtilizationRate            0.4714      0.020722      22.749             0
    Age                    -0.0014209    0.00076326     -1.8616      0.062771
    Marriage_not married    -0.010542       0.01578    -0.66807       0.50415
    (Sigma)                    0.3618     0.0050022       72.33             0
```

```
Number of observations: 2627
Number of left-censored observations: 0
Number of uncensored observations: 2626
Number of right-censored observations: 1
Log-likelihood: -1057.9
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

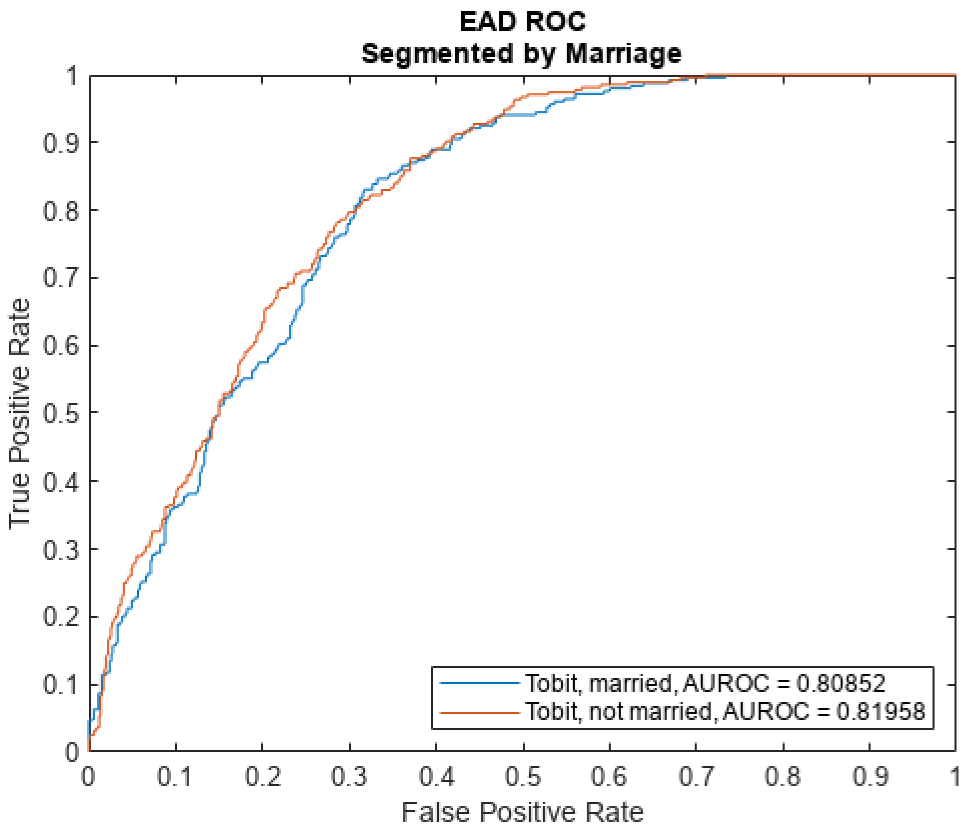
Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ead;
```

```
[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel
modelDiscriminationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,SegmentBy="Marriage");
```



Plot ROC Using a Beta EAD Model

This example shows how to use `fitEADModel` to create a `Beta` model and then use `modelDiscriminationPlot` to plot the ROC.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
```

```
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Beta`.

```
ModelType = Beta ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Beta EAD Model

Use `fitEADModel` to create a `Beta` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-07
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Beta regression model:
     logit(EAD_lcf) ~ 1_mu + UtilizationRate_mu + Age_mu + Marriage_mu
     log(EAD_lcf) ~ 1_phi + UtilizationRate_phi + Age_phi + Marriage_phi

Estimated coefficients:
                                Estimate       SE        tStat       pValue  
                               __________    _________    ________    __________

    (Intercept)_mu               -0.65566      0.11484     -5.7093    1.2616e-08
    UtilizationRate_mu             1.7014     0.078094      21.787             0
    Age_mu                     -0.0055901    0.0027603     -2.0252      0.042949
    Marriage_not married_mu     -0.012577     0.052098    -0.24141       0.80926
    (Intercept)_phi              -0.50131     0.094625     -5.2979    1.2686e-07
    UtilizationRate_phi           0.39731     0.066707       5.956    2.9303e-09
    Age_phi                     -0.001167    0.0023161    -0.50387        0.6144
```

```
    Marriage_not married_phi     -0.013275     0.042627    -0.31143        0.7555

Number of observations: 2627
Log-likelihood: -3140.21
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

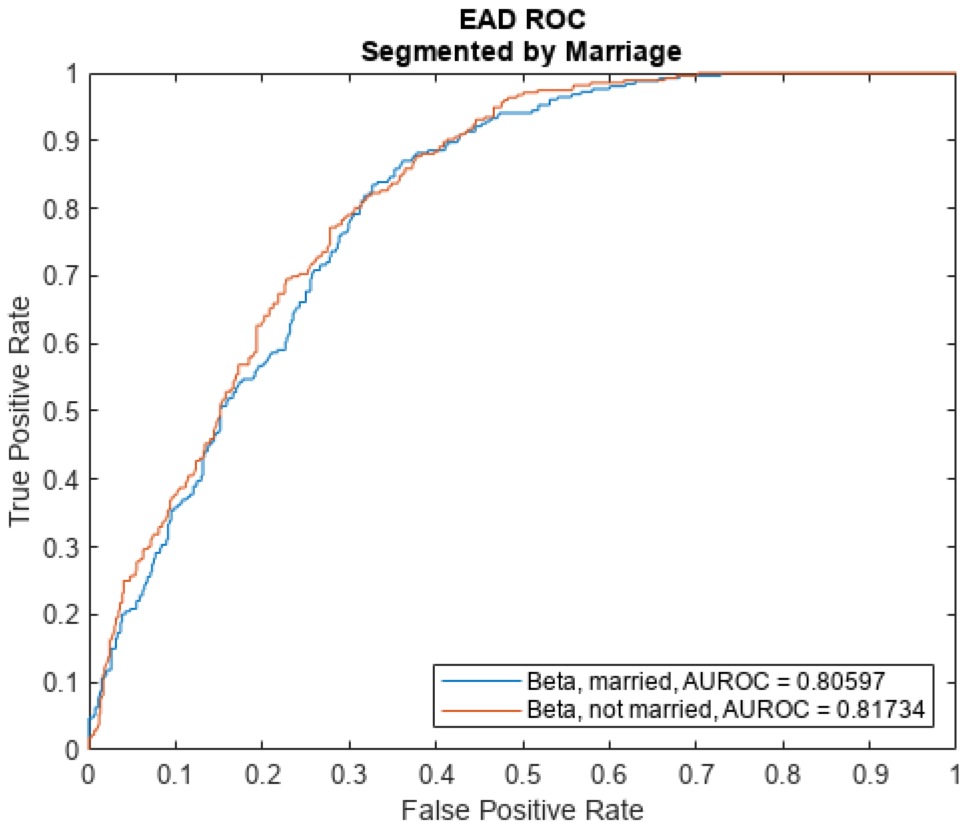
```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ead ;

[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel
modelDiscriminationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,SegmentBy="Marriage");
```



Input Arguments

eadModel — Exposure at model
Regression object | Tobit object | Beta object

Exposure at default model, specified as a previously created Regression, Tobit, or Beta object using fitEADModel.

Data Types: object

data — Data
table

Data, specified as a NumRows-by-NumCols table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: table

ax — Valid axis object
object

(Optional) Valid axis object, specified as an ax object that is created using axes. The plot will be created in the axes specified by the optional ax argument instead of in the current axes (gca). The optional argument ax must precede any of the input argument combinations.

Data Types: object

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example:
`modelDiscriminationPlot(eadModel,data(TestInd,:),DataID='Testing',DiscretizeBy='median')`

`DataID` — Data set identifier
`""` (default) | character vector | string

Data set identifier, specified as `DataID` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`DiscretizeBy` — Discretization method for EAD `data` at defined `ModelLevel`
`'mean'` (default) | character vector with value `'mean'` or `'median'` | string with value `"mean"` or `"median"`

Discretization method for EAD `data` at the defined `ModelLevel`, specified as `DiscretizeBy` and a character vector or string.

- `'mean'` — Discretized response is `1` if observed EAD is greater than or equal to the mean EAD, `0` otherwise.
- `'median'` — Discretized response is `1` if observed EAD is greater than or equal to the median EAD, `0` otherwise.

Data Types: `char` | `string`

`SegmentBy` — Name of column in `data` input used to segment data set
`""` (default) | character vector | string

Name of a column in the `data` input, not necessarily a model variable, to be used to segment the data set, specified as `SegmentBy` and a character vector or string. One AUROC is reported for each segment, and the corresponding ROC data for each segment is returned in the optional output.

Data Types: `char` | `string`

`ModelLevel` — Model level
`"ead"` (default) | character vector with value `'ead'`, `'conversionMeasure'`, or `'conversionTransform'` | string with value `"ead"`, `"conversionMeasure"`, or `"conversionTransform"`

Model level, specified as `ModelLevel` and a character vector or string.

Note `Regression` models support all three model levels, but a `Tobit` or `Beta` model supports model levels only for `"ead"` and `"conversionMeasure"`.

Data Types: `char` | `string`

`ReferenceEAD` — EAD values predicted for `data` by reference model
`[]` (default) | numeric vector

EAD values predicted for `data` by the reference model, specified as `ReferenceEAD` and a `NumRows`-by-`1` numeric vector. The ROC curve is plotted for both the `eadModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as `ReferenceID` and a character vector or string. `'ReferenceID'` is used in the plot for reporting purposes.

Data Types: `char` | `string`

Output Arguments

h — Figure handle
handle object

Figure handle for the line objects, returned as handle object.

More About

Model Discrimination Plot

The `modelDiscriminationPlot` function plots the receiver operator characteristic (ROC) curve.

The `modelDiscriminationPlot` function also shows the area under the receiver operator characteristic (AUROC) curve, sometimes called simply the area under the curve (AUC). This metric is between 0 and 1 and higher values indicate better discrimination.

A numeric prediction and a binary response are needed to plot the ROC and compute the AUROC. For EAD models, the predicted EAD is used directly as the prediction. However, the observed EAD must be discretized into a binary variable. By default, observed EAD values greater than or equal to the mean observed EAD are assigned a value of 1, and values below the mean are assigned a value of 0. This discretized response is interpreted as "high EAD" vs. "low EAD." The ROC curve and the AUROC curve measure how well the predicted EAD separates the "high EAD" vs. the "low EAD" observations. You can change the level to compute the model discrimination with the `ModelLevel` name-value pair argument and the discretization criterion with the `DiscretizeBy` name-value pair argument.

The ROC curve is a parametric curve that plots the proportion of

- High EAD cases with predicted EAD greater than or equal to a parameter *t*, or true positive rate (TPR)
- Low EAD cases with predicted EAD greater than or equal to the same parameter *t*, or false positive rate (FPR)

The parameter *t* sweeps through all the observed predicted EAD values for the given data. If the AUROC value or the ROC curve data are needed programmatically, use the `modelDiscrimination` function. For more information about ROC curves, see “ROC Curve and Performance Metrics”.

Version History

Introduced in R2021b

R2022b: Support for Beta model

Behavior changed in R2022b

The eadModel input supports an option for a Beta model object that you can create using fitEADModel.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

Regression | Tobit | fitEADModel | predict | modelDiscrimination | modelCalibration | modelCalibrationPlot

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Exposure at Default Models" on page 1-34

modelDiscriminationPlot

Plot ROC curve

Syntax

```
modelDiscriminationPlot(lgdModel,data)
modelDiscriminationPlot( ___ ,Name,Value)
h = modelDiscriminationPlot(ax, ___ ,Name,Value)
```

Description

`modelDiscriminationPlot(lgdModel,data)` generates the receiver operating characteristic (ROC) curve. `modelDiscriminationPlot` supports segmentation and comparison against a reference model.

`modelDiscriminationPlot( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

`h = modelDiscriminationPlot(ax, ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the figure handle `h`.

Examples

Plot ROC Using Regression LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Regression` model and then use `modelDiscriminationPlot` to plot the ROC.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV         Age          Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create a Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'regression');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Regression"
          Description: ""
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

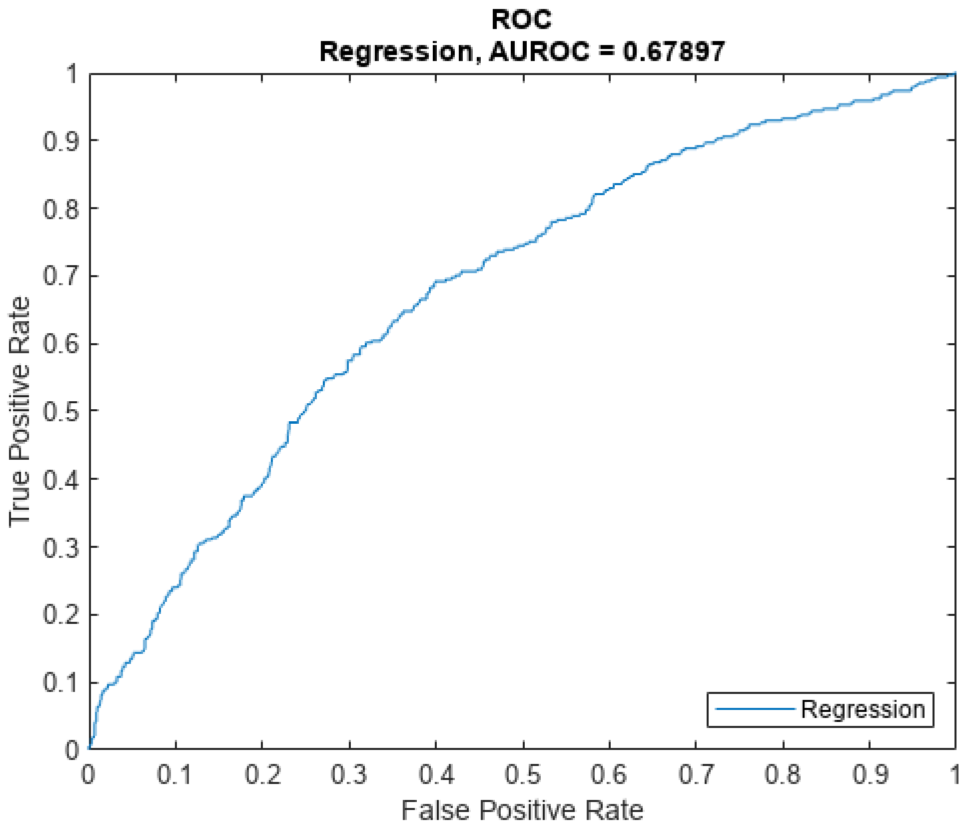
```
disp(lgdModel.UnderlyingModel)
```

```
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

Plot ROC Data

Use `modelDiscriminationPlot` to plot the ROC for the test data set.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:))
```



Plot ROC Using Tobit LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Tobit` model and then use `modelDiscriminationPlot` to plot the ROC.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create a Tobit LGD Model

Use `fitLGDModel` to create a `Tobit` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'tobit');
disp(lgdModel)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

Display the underlying model.

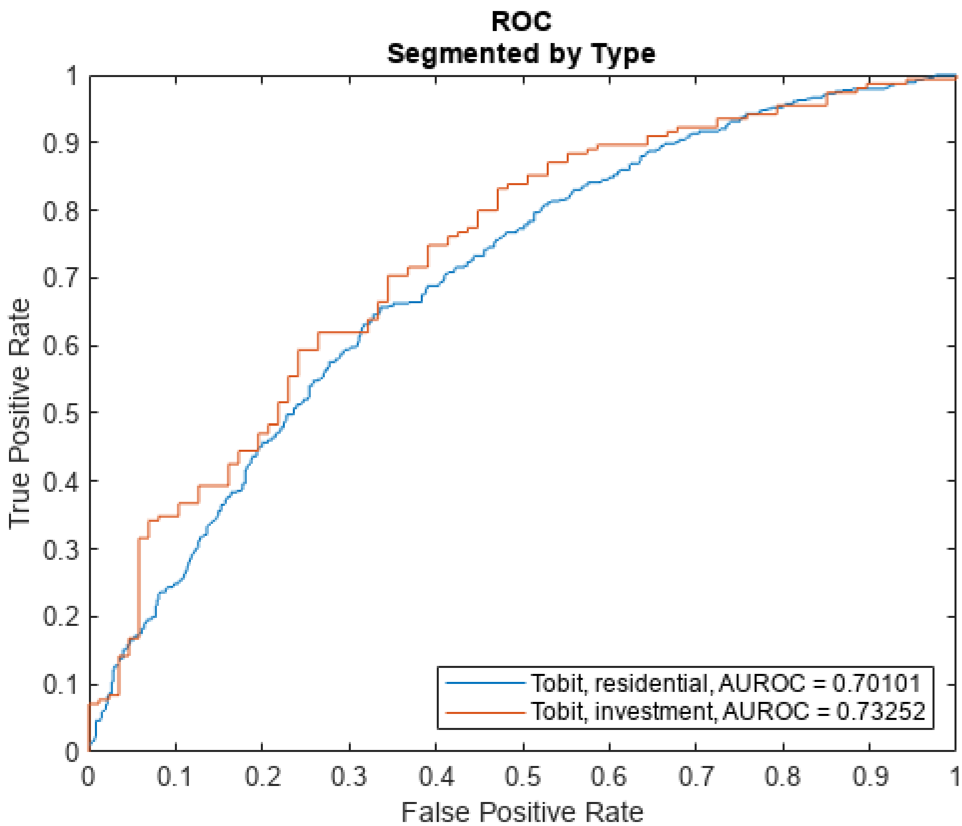
```
disp(lgdModel.UnderlyingModel)
```

```
Tobit regression model:
     LGD = max(0,min(Y*,1))
     Y* ~ 1 + LTV + Age + Type

Estimated coefficients:
                       Estimate        SE         tStat       pValue  
                       _________    _________    _______    __________

    (Intercept)         0.058257     0.027265     2.1367      0.032737
    LTV                  0.20126     0.031354     6.4189    1.6932e-10
    Age                -0.095407    0.0072653    -13.132             0
    Type_investment      0.10208     0.018058     5.6531    1.7915e-08
    (Sigma)              0.29288    0.0057036      51.35             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1521
Number of right-censored observations: 25
Log-likelihood: -698.383
```

Plot ROC Data

Use `modelDiscriminationPlot` to plot the ROC for the test data set.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:),"SegmentBy","Type","DiscretizeBy","median")
```



Plot ROC Using Beta LGD Model

This example shows how to use `fitLGDModel` to fit data with a `Beta` model and then use `modelDiscriminationPlot` to plot the ROC.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create a Beta LGD Model

Use `fitLGDModel` to create a `Beta` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'Beta');
disp(lgdModel)
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-05
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

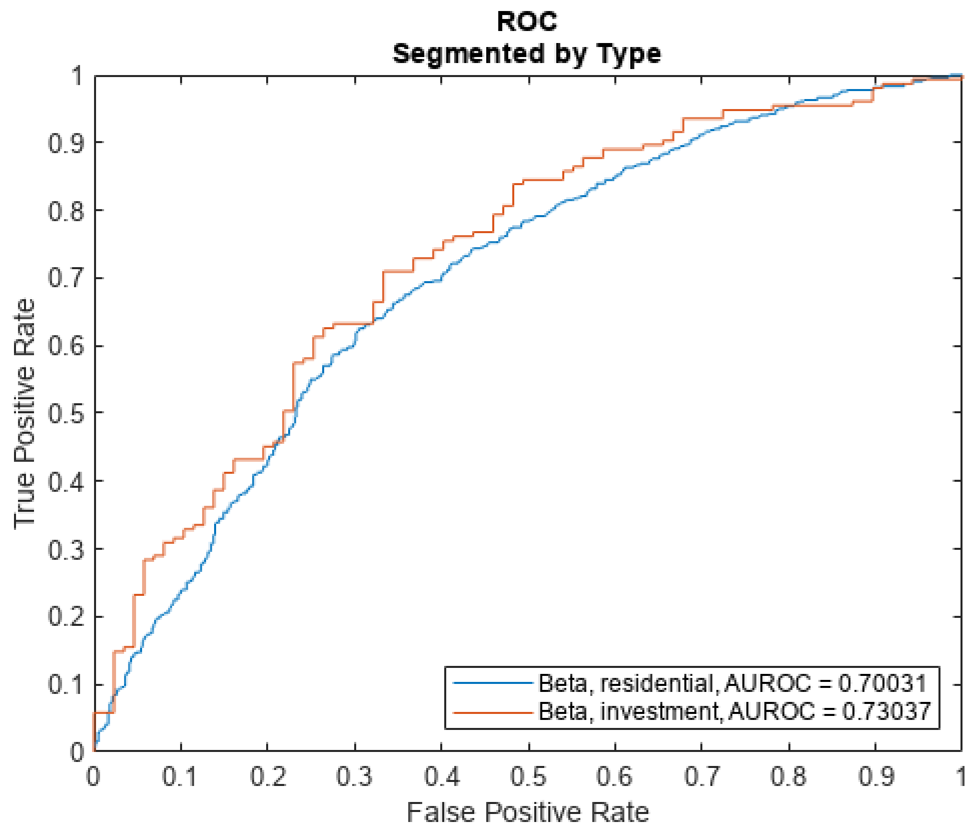
```
disp(lgdModel.UnderlyingModel)
```

```
Beta regression model:
     logit(LGD) ~ 1_mu + LTV_mu + Age_mu + Type_mu
     log(LGD) ~ 1_phi + LTV_phi + Age_phi + Type_phi

Estimated coefficients:
                           Estimate       SE        tStat       pValue  
                           ________    ________    ________    __________

    (Intercept)_mu          -1.3772     0.13201     -10.433             0
    LTV_mu                  0.60269     0.15087      3.9947    6.7023e-05
    Age_mu                 -0.47464    0.040264     -11.788             0
    Type_investment_mu      0.45372    0.085143      5.3289     1.094e-07
    (Intercept)_phi        -0.16337     0.12591     -1.2975       0.19462
    LTV_phi                0.055892     0.14719     0.37973       0.70419
    Age_phi                 0.22887    0.040335      5.6743    1.5863e-08
    Type_investment_phi    -0.14102    0.078155     -1.8044      0.071311

Number of observations: 2093
Log-likelihood: -5291.04
```

Plot ROC Data

Use `modelDiscriminationPlot` to plot the ROC for the test data set.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:),"SegmentBy","Type","DiscretizeBy","median")
```



Input Arguments

`lgdModel` — Loss given default model

`Regression` object | `Tobit` object | `Beta` object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitLGDModel`.

Data Types: `object`

`data` — Data

table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

`ax` — Valid axis object

object

(Optional) Valid axis object, specified as an `ax` object that is created using `axes`. The plot will be created in the axes specified by the optional `ax` argument instead of in the current axes (gca). The optional argument `ax` must precede any of the input argument combinations.

Data Types: `object`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
`modelDiscriminationPlot(lgdModel,data(TestInd,:),'DataID','Testing','DiscretizeBy','median')`

`DataID` — Data set identifier

`""` (default) | character vector | string

Data set identifier, specified as the comma-separated pair consisting of `'DataID'` and a character vector or string. The `DataID` is included in the output for reporting purposes.

Data Types: `char` | `string`

`DiscretizeBy` — Discretization method for LGD `data`

`'mean'` (default) | character vector with value `'mean'`, `'median'`, `'positive'`, or `'total'` | string with value `"mean"`, `"median"`, `"positive"`, or `"total"`

Discretization method for LGD `data`, specified as the comma-separated pair consisting of `'DiscretizeBy'` and a character vector or string.

- `'mean'` — Discretized response is `1` if observed LGD is greater than or equal to the mean LGD, `0` otherwise.
- `'median'` — Discretized response is `1` if observed LGD is greater than or equal to the median LGD, `0` otherwise.
- `'positive'` — Discretized response is `1` if observed LGD is positive, `0` otherwise (full recovery).
- `'total'` — Discretized response is `1` if observed LGD is greater than or equal to `1` (total loss), `0` otherwise.

Data Types: `char` | `string`

`SegmentBy` — Name of column in `data` input used to segment data set

`""` (default) | character vector | string

Name of a column in the `data` input, not necessarily a model variable, to be used to segment the data set, specified as the comma-separated pair consisting of `'SegmentBy'` and a character vector or string. One AUROC is reported for each segment, and the corresponding ROC data for each segment is returned in the optional output.

Data Types: `char` | `string`

`ReferenceLGD` — LGD values predicted for `data` by reference model

`[ ]` (default) | numeric vector

LGD values predicted for `data` by the reference model, specified as the comma-separated pair consisting of `'ReferenceLGD'` and a `NumRows`-by-`1` numeric vector. The ROC curve is plotted for both the `lgdModel` object and the reference model.

Data Types: `double`

`ReferenceID` — Identifier for the reference model
`'Reference'` (default) | character vector | string

Identifier for the reference model, specified as the comma-separated pair consisting of `'ReferenceID'` and a character vector or string. `'ReferenceID'` is used in the plot for reporting purposes.

Data Types: `char` | `string`

Output Arguments

h — Figure handle
handle object

Figure handle for the line objects, returned as handle object.

More About

Model Discrimination Plot

The `modelDiscriminationPlot` function plots the receiver operator characteristic (ROC) curve.

The `modelDiscriminationPlot` function also shows the area under the receiver operator characteristic (AUROC) curve, sometimes called simply the area under the curve (AUC). This metric is between 0 and 1 and higher values indicate better discrimination.

A numeric prediction and a binary response are needed to plot the ROC and compute the AUROC. For LGD models, the predicted LGD is used directly as the prediction. However, the observed LGD must be discretized into a binary variable. By default, observed LGD values greater than or equal to the mean observed LGD are assigned a value of 1, and values below the mean are assigned a value of 0. This discretized response is interpreted as "high LGD" vs. "low LGD." The ROC curve and the AUROC curve measure how well the predicted LGD separates the "high LGD" vs. the "low LGD" observations. The discretization criterion can be changed with the `DiscretizeBy` name-value pair argument for `modelDiscriminationPlot`.

The ROC curve is a parametric curve that plots the proportion of

- High LGD cases with predicted LGD greater than or equal to a parameter *t*, or true positive rate (TPR)
- Low LGD cases with predicted LGD greater than or equal to the same parameter *t*, or false positive rate (FPR)

The parameter *t* sweeps through all the observed predicted LGD values for the given data. If the AUROC value or the ROC curve data are needed programmatically, use the `modelDiscrimination` function. For more information about ROC curves, see “ROC Curve and Performance Metrics”.

Version History

Introduced in R2021a

R2022b: Support for `Beta` model
Behavior changed in R2022b

The lgdModel input supports an option for a Beta model object that you can create using fitLGDModel.

R2022a: Support for reference LGD outside of [0,1] range
Behavior changed in R2022a

The Regression and Tobit LGD models support a reference LGD outside of the [0,1] range.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

Tobit | Regression | modelCalibration | modelCalibartionPlot | modelDiscrimination | predict | fitLGDModel

Topics

“Model Loss Given Default” on page 4-90
“Basic Loss Given Default Model Validation” on page 4-131
“Compare Tobit LGD Model to Benchmark Model” on page 4-133
“Compare Loss Given Default Models Using Cross-Validation” on page 4-140
“Overview of Loss Given Default Models” on page 1-31

pof

Proportion of failures test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = pof(vbt)
TestResults = pof(vbt,Name,Value)
```

Description

`TestResults = pof(vbt)` generates the proportion of failures (POF) test for value-at-risk (VaR) backtesting.

`TestResults = pof(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Generate POF Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `pof` test results.

```
TestResults = pof(vbt,'TestLevel',0.99)
```

```
TestResults=1×9 table
    PortfolioID    VaRID    VaRLevel     POF      LRatioPOF    PValuePOF    Observations    Failu
    ___________    _____    ________    ______    _________    _________    ____________    _____

    "Portfolio"    "VaR"      0.95      accept     0.46147      0.49694         1043           57
```

Run the POF Test for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `pof` test results using the `TestLevel` optional input.

```
TestResults = pof(vbt,'TestLevel',0.90)
```

```
TestResults=6×9 table
    PortfolioID        VaRID          VaRLevel     POF      LRatioPOF    PValuePOF    Observations
    ___________    ______________    ________    ______    _________    _________    ____________

     "Equity"      "Normal95"          0.95      accept     0.46147       0.49694        1043
     "Equity"      "Normal99"          0.99      reject      3.5118      0.060933        1043
     "Equity"      "Historical95"      0.95      accept     0.91023       0.34005        1043
     "Equity"      "Historical99"      0.99      accept     0.22768       0.63325        1043
     "Equity"      "EWMA95"            0.95      accept     0.91023       0.34005        1043
     "Equity"      "EWMA99"            0.99      reject      9.8298     0.0017171        1043
```

Input Arguments

vbt — `varbacktest` object
object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = pof(vbt,'TestLevel',0.99)`

`TestLevel` — Test confidence level
`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — `pof` test results
table

`pof` test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR level to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'POF'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `pof` test
- `'LRatioPOF'` — Likelihood ratio of the `pof` test
- `'PValuePOF'` — P-value of the `pof` test
- `'Observations'` — Number of observations
- `'Failures'` — Number of failures
- `'TestLevel'` — Test confidence level

Note For `pof` test results, the terms `accept` and `reject` are used for convenience, technically a `pof` test does not accept a model. Rather, the test fails to reject it.

More About

Proportion of Failures (POF) Test

The `pof` function performs Kupiec's proportion of failures test.

The POF test is a likelihood ratio test proposed by Kupiec (1995) to assess if the proportion of failures (number of failures divided by number of observations) is consistent with the VaR confidence level.

Algorithms

The likelihood ratio (test statistic) of the `pof` test is given by

$$LRatioPOF = -2\log\left(\frac{(1-pVaR)^{N-x}pVaR^{x}}{\left(1-\frac{x}{N}\right)^{N-x}\left(\frac{x}{N}\right)^{x}}\right) = -2\left[(N-x)\log\left(\frac{N(1-pVaR)}{N-x}\right) + x\log\left(\frac{NpVaR}{x}\right)\right]$$

where N is the number of observations, x is the number of failures, and $pVaR$ = 1 - $VaRLevel$. This test statistic is asymptotically distributed as a chi-square distribution with 1 degree of freedom. By the properties of the logarithm,

$$LRatioPOF = -2N\log(1-pVar) \quad \text{if } x = 0.$$

and

$$LRatioPOF = -2N\log(pVar) \quad \text{if } x = N.$$

The *p*-value of the POF test is the probability that a chi-square distribution with 1 degree of freedom exceeds the likelihood ratio *LRatioPOF*

$$PValuePOF = 1 - F(LRatioPOF)$$

where *F* is the cumulative distribution of a chi-square variable with 1 degree of freedom.

The result of the test is to accept if

$$PValuePOF < F(TestLevel)$$

and reject otherwise, where *F* is the cumulative distribution of a chi-square variable with 1 degree of freedom.

Version History

Introduced in R2016b

References

[1] Kupiec, P. "Techniques for Verifying the Accuracy of Risk Management Models." *Journal of Derivatives.* Vol. 3, 1995, pp. 73 – 84.

See Also

`varbacktest` | `tl` | `tuff` | `bin` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Topics

“VaR Backtesting Workflow” on page 2-6
“Value-at-Risk Estimation and Backtesting” on page 2-10
“Overview of VaR Backtesting” on page 2-2
“Kupiec’s POF and TUFF Tests” on page 2-3
“Comparison of ES Backtesting Methods” on page 2-26

predict

Predict exposure at default

Syntax

```
predictedEAD = predict(eadModel,data)
predictedEAD = predict(___,Name=Value)
```

Description

`predictedEAD = predict(eadModel,data)` computes the exposure at default (EAD).

When using a `Regression` model, the `predict` function operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale.

`predictedEAD = predict(___,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax.

Examples

Use Tobit EAD Model to Predict EAD

This example shows how to use `fitEADModel` to create a `Tobit` model and then predict exposure at default (EAD) values.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Tobit` or `Regression`.

```
ModelType = Tobit ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Tobit EAD Model

Use `fitEADModel` to create a `Tobit` model using the `TrainingInd` data.

```
eadModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age','M
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Tobit with properties:

        CensoringSide: "both"
            LeftLimit: 0
           RightLimit: 1
              ModelID: "Tobit"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Tobit regression model:
     EAD_lcf = max(0,min(Y*,1))
     Y* ~ 1 + UtilizationRate + Age + Marriage

Estimated coefficients:
                             Estimate        SE         tStat       pValue
                            __________    __________    ________    __________

    (Intercept)               0.22467       0.03134      7.1689     9.7855e-13
    UtilizationRate            0.4714      0.020722      22.749              0
    Age                    -0.0014209    0.00076326     -1.8616       0.062771
    Marriage_not married    -0.010542       0.01578    -0.66807        0.50415
    (Sigma)                    0.3618     0.0050022       72.33              0

Number of observations: 2627
Number of left-censored observations: 0
Number of uncensored observations: 2626
```

```
Number of right-censored observations: 1
Log-likelihood: -1057.9
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

```
predictedEAD = predict(eadModel, EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel, EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Use Beta EAD Model to Predict EAD

This example shows how to use `fitEADModel` to create a `Beta` model and then predict exposure at default (EAD) values.

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age      Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Beta`.

```
ModelType = Beta;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF;
```

Create Beta EAD Model

Use `fitEADModel` to create a `Beta` model using `EADData`.

```
eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'}, ...
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-07
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Beta regression model:
     logit(EAD_lcf) ~ 1_mu + UtilizationRate_mu + Age_mu + Marriage_mu
     log(EAD_lcf) ~ 1_phi + UtilizationRate_phi + Age_phi + Marriage_phi

Estimated coefficients:
                                 Estimate        SE          tStat        pValue
                                _________    _________    _________    __________

    (Intercept)_mu                -0.6741     0.087775      -7.6799     1.954e-14
    UtilizationRate_mu             1.6974     0.060621           28             0
    Age_mu                     -0.0046006    0.0021317      -2.1582      0.030965
    Marriage_not married_mu    -0.0020533     0.040397    -0.050827       0.95947
    (Intercept)_phi              -0.43364     0.071602      -6.0562    1.5101e-09
    UtilizationRate_phi           0.42461     0.051852       8.1889    4.4409e-16
    Age_phi                    -0.0036089    0.0017592      -2.0514      0.040285
    Marriage_not married_phi    -0.016663     0.032976     -0.50531       0.61337

Number of observations: 4378
Log-likelihood: -5255.12
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-value argument.

```
predictedEAD = predict(eadModel, EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel, EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Input Arguments

eadModel — Exposure at default model

Regression object | Tobit object | Beta object

Exposure at default model, specified as a previously created Regression, Tobit, or Beta object using fitEADModel.

Data Types: object

data — Data

table

Data, specified as a NumRows-by-NumCols table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: table

Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel='ead')

ModelLevel — Model level

"ead" (default) | character vector with value 'ead', 'conversionMeasure', or 'conversionTransform' | string with value "ead", "conversionMeasure", or "conversionTransform"

Model level, specified as ModelLevel and a character vector or string.

Note Regression models support all three model levels, but a Tobit or Beta model supports model levels only for 'ead' and 'conversionMeasure'.

Data Types: char | string

Output Arguments

predictedEAD — Exposure at default predicted values

vector

Exposure at default predicted values, returned as a NumRows-by-1 numeric vector.

More About

Prediction with EAD Models

Use a Regression, Tobit, or Beta model to predict EAD.

`Regression`, `Tobit`, or `Beta` EAD models first predict on the transformed space using the underlying linear regression model, and then apply the inverse transformation to return predictions on the EAD scale.

Version History

Introduced in R2021b

R2022b: Support for Beta model

Behavior changed in R2022b

The `eadModel` input supports an option for a `Beta` model object that you can create using `fitEADModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

`Regression` | `Tobit` | `Beta` | `fitEADModel` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot`

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Exposure at Default Models" on page 1-34

predict

Predict loss given default

Syntax

```
LGD = predict(lgdModel,data)
```

Description

`LGD = predict(lgdModel,data)` computes the loss given default (LGD).

When using a `Regression` model, the `predict` function operates on the underlying compact statistical model and then transforms the predicted values back to the LGD scale.

When using a `Tobit` model, the `predict` function operates on the underlying `Tobit` regression model and returns the unconditional expected value of the response, given the predictor values.

Examples

Use Regression LGD Model to Predict LGD

This example shows how to use `fitLGDModel` to fit data with a `Regression` model and then predict the loss given default (LGD) values.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV         Age          Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
```

```
TrainingInd = training(c);
TestInd = test(c);
```

Create Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'regression');
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "logit"
    BoundaryTolerance: 1.0000e-05
              ModelID: "Regression"
          Description: ""
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Compact linear regression model:
    LGD_logit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue
                       ________    ________    _______    __________

    (Intercept)        -4.7549      0.36041    -13.193    3.0997e-38
    LTV                 2.8565      0.41777     6.8377    1.0531e-11
    Age                -1.5397     0.085716    -17.963    3.3172e-67
    Type_investment     1.4358       0.2475     5.8012     7.587e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 4.24
R-squared: 0.206,  Adjusted R-Squared: 0.205
F-statistic vs. constant model: 181, p-value = 2.42e-104
```

Predict LGD on Test Data

Use `predict` to predict the LGD for the test data set.

```
predictedLGD = predict(lgdModel,data(TestInd,:))
```

```
predictedLGD = 1394×1

    0.0009
    0.0037
    0.1877
    0.0011
    0.0112
    0.0420
    0.0529
    0.0000
    0.0090
```

```
    0.0239
      ⋮
```

You can analyze and validate these predictions using `modelDiscrimination` and `modelCalibration`.

Use Tobit LGD Model to Predict LGD

This example shows how to use `fitLGDModel` to fit data with a `Tobit` model and then predict the loss given default (LGD) values.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD    
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Tobit LGD Model

Use `fitLGDModel` to create a `Tobit` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'tobit');
disp(lgdModel)
```

```
  Tobit with properties:

      CensoringSide: "both"
          LeftLimit: 0
         RightLimit: 1
            ModelID: "Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
```

```
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Tobit regression model:
     LGD = max(0,min(Y*,1))
     Y* ~ 1 + LTV + Age + Type

Estimated coefficients:
                       Estimate        SE         tStat       pValue  
                       _________    _________    _______    __________

    (Intercept)         0.058257     0.027265     2.1367      0.032737
    LTV                  0.20126     0.031354     6.4189    1.6932e-10
    Age                -0.095407    0.0072653    -13.132             0
    Type_investment      0.10208     0.018058     5.6531    1.7915e-08
    (Sigma)              0.29288    0.0057036      51.35             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1521
Number of right-censored observations: 25
Log-likelihood: -698.383
```

Predict LGD on Test Data

Use `predict` to predict the LGD for the test data set.

```
predictedLGD = predict(lgdModel,data(TestInd,:))
```

```
predictedLGD = 1394×1

    0.0879
    0.1243
    0.3204
    0.0934
    0.1672
    0.2238
    0.2370
    0.0102
    0.1592
    0.1989
      ⋮
```

You can analyze and validate these predictions using `modelDiscrimination` and `modelCalibration`.

Use Beta LGD Model to Predict LGD

This example shows how to use `fitLGDModel` to fit data with a `Beta` model and then predict the loss given default (LGD) values.

Load Data

Load the loss given default data.

```
load LGDData.mat
head(data)
```

```
      LTV         Age          Type           LGD
    _______     _______    ___________    _________

    0.89101     0.39716    residential     0.032659
    0.70176      2.0939    residential      0.43564
    0.72078      2.7948    residential    0.0064766
    0.37013       1.237    residential     0.007947
    0.36492      2.5818    residential            0
      0.796      1.5957    residential      0.14572
    0.60203      1.1599    residential     0.025688
    0.92005     0.50253    investment       0.063182
```

Partition Data

Separate the data into training and test partitions.

```
rng('default'); % for reproducibility
NumObs = height(data);

c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Beta LGD Model

Use `fitLGDModel` to create a `Beta` model using training data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'Beta');
disp(lgdModel)
```

```
  Beta with properties:

    BoundaryTolerance: 1.0000e-05
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

```
disp(lgdModel.UnderlyingModel)
```

```
Beta regression model:
     logit(LGD) ~ 1_mu + LTV_mu + Age_mu + Type_mu
     log(LGD) ~ 1_phi + LTV_phi + Age_phi + Type_phi

Estimated coefficients:
                           Estimate       SE        tStat       pValue  
                           ________    ________    _______    __________

    (Intercept)_mu          -1.3772     0.13201    -10.433             0
```

```
    LTV_mu                  0.60269     0.15087      3.9947    6.7023e-05
    Age_mu                 -0.47464    0.040264     -11.788             0
    Type_investment_mu      0.45372    0.085143      5.3289     1.094e-07
    (Intercept)_phi        -0.16337     0.12591     -1.2975       0.19462
    LTV_phi                0.055892     0.14719     0.37973       0.70419
    Age_phi                 0.22887    0.040335      5.6743    1.5863e-08
    Type_investment_phi    -0.14102    0.078155     -1.8044      0.071311

Number of observations: 2093
Log-likelihood: -5291.04
```

Predict LGD on Test Data

Use `predict` to predict the LGD for the test data set.

```
predictedLGD = predict(lgdModel,data(TestInd,:))
```

```
predictedLGD = 1394×1

    0.0937
    0.1492
    0.3526
    0.0964
    0.1886
    0.2595
    0.2677
    0.0213
    0.1774
    0.2256
      ⋮
```

You can analyze and validate these predictions using `modelDiscrimination` and `modelCalibration`.

Input Arguments

`lgdModel` — Loss given default model
`Regression` object | `Tobit` object | `Beta` object

Loss given default model, specified as a previously created `Regression`, `Tobit`, or `Beta` object using `fitLGDModel`.

Data Types: `object`

data — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with predictor and response values. The variable names and data types must be consistent with the underlying model.

Data Types: `table`

Output Arguments

LGD — Loss given default values
vector

Loss given default values, returned as a `NumRows`-by-`1` numeric vector.

More About

Prediction with LGD Models

Use a Regression, Tobit, or Beta model to predict LGD.

`Regression` LGD models first predict on the transformed space using the underlying linear regression model, and then apply the inverse transformation to return predictions on the LGD scale. For more information on the supported transformations and their inverses, see “Loss Given Default Regression Models” on page 6-677.

`Tobit` LGD models return the unconditional expected value of the response, given the predictor values. For more information, see “Loss Given Default Tobit Models” on page 6-694.

`Beta` LGD models return the mean of the beta distribution, given the predictor values. For more information, see “Beta Regression Models” on page 6-685.

Version History

Introduced in R2021a

R2022b: Support for `Beta` model

Behavior changed in R2022b

The `lgdModel` input supports an option for a `Beta` model object that you can create using `fitLGDModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

`Tobit` | `Regression` | `Beta` | `modelCalibration` | `modelCalibartionPlot` | `modelDiscriminationPlot` | `modelDiscrimination` | `fitLGDModel`

Topics

“Model Loss Given Default” on page 4-90
“Basic Loss Given Default Model Validation” on page 4-131
“Compare Tobit LGD Model to Benchmark Model” on page 4-133
“Compare Loss Given Default Models Using Cross-Validation” on page 4-140
“Overview of Loss Given Default Models” on page 1-31

predict

Compute conditional PD

Syntax

```
conditionalPD = predict(pdModel,data)
```

Description

`conditionalPD = predict(pdModel,data)` computes the conditional probability of default (PD).

Examples

Use Probit Lifetime PD Model to Predict Conditional PD

This example shows how to use `fitLifetimePDModel` to fit data with a `Probit` model and then predict the conditional probability of default (PD).

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Probit Lifetime PD Model

Use `fitLifetimePDModel` to create a `Probit` model.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Probit",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
disp(pdModel)
```

```
  Probit with properties:

            ModelID: "Probit"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Display the underlying model.

```
pdModel.UnderlyingModel
```

```
ans = 
Compact generalized linear regression model:
```

```
    probit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate         SE          tStat        pValue
                              __________    _________    _______    ___________

    (Intercept)                 -1.6267       0.03811    -42.685              0
    ScoreGroup_Medium Risk     -0.26542       0.01419    -18.704     4.5503e-78
    ScoreGroup_Low Risk        -0.46794      0.016364    -28.595    7.775e-180
    YOB                        -0.11421     0.0049724    -22.969    9.6208e-117
    GDP                       -0.041537      0.014807    -2.8052      0.0050291
    Market                   -0.0029609     0.0010618    -2.7885      0.0052954


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Predict on Training and Test Data

Predict the PD for training or test data sets.

```
DataSetChoice = Training ;
if DataSetChoice=="Training"
    Ind = TrainDataInd;
 else
    Ind = TestDataInd;
 end

% Predict conditional PD
PD = predict(pdModel,data(Ind,:));
head(data(Ind,:))
```

```
    ID     ScoreGroup     YOB    Default    Year     GDP     Market
    __    ___________    ___    _______    ____    _____    ______

    1      Low Risk       1        0       1997     2.72      7.61
    1      Low Risk       2        0       1998     3.57     26.24
    1      Low Risk       3        0       1999     2.86      18.1
    1      Low Risk       4        0       2000     2.43      3.19
    1      Low Risk       5        0       2001     1.26    -10.51
    1      Low Risk       6        0       2002    -0.59    -22.95
    1      Low Risk       7        0       2003     0.63      2.78
    1      Low Risk       8        0       2004     1.85      9.48
```

```
disp(PD(1:8))
```

```
    0.0095
    0.0054
    0.0045
    0.0039
    0.0036
    0.0036
    0.0017
    0.0009
```

You can analyze and validate these predictions using modelDiscrimination and modelCalibration.

Use Cox Lifetime PD Model to Predict Conditional PD

This example shows how to use fitLifetimePDModel to fit data with a Cox model and then predict the conditional probability of default (PD).

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Cox Lifetime PD Model

Use `fitLifetimePDModel` to create a `Cox` model.

```
ModelType = cox ;

pdModel = fitLifetimePDModel(data(TrainDataInd,:),ModelType,...
   'IDVar','ID','AgeVar','YOB',...
   'LoanVars','ScoreGroup','MacroVars',{'GDP' 'Market'},...
   'ResponseVar','Default');
disp(pdModel)
```

```
  Cox with properties:

            TimeInterval: 1
     ExtrapolationFactor: 1
                 ModelID: "Cox"
             Description: ""
         UnderlyingModel: [1x1 CoxModel]
                   IDVar: "ID"
                  AgeVar: "YOB"
                LoanVars: "ScoreGroup"
               MacroVars: ["GDP"    "Market"]
             ResponseVar: "Default"
```

Display the underlying model.

```
disp(pdModel.UnderlyingModel)
```

```
Cox Proportional Hazards regression model

                                Beta          SE          zStat       pValue
                             __________    _________    _______    ___________

    ScoreGroup_Medium Risk      -0.6794     0.037029    -18.348     3.4442e-75
    ScoreGroup_Low Risk         -1.2442     0.045244    -27.501    1.7116e-166
    GDP                       -0.084533     0.043687     -1.935       0.052995
    Market                   -0.0084411    0.0032221    -2.6198      0.0087991


Log-likelihood: -41742.871
```

Predict on Age Values not Observed in the Training Data

Cox models make predictions for the range of age values observed in the training data. To extrapolate for ages larger than the maximum age in the training data, an extrapolation rule is needed.

When using `predict` with a `Cox` model, you can set the `ExtrapolationFactor` property of the `Cox` model. By default, the `ExtrapolationFactor` is set to `1`. For age values (`AgeVar`) greater than the maximum age observed in the training data, `predict` computes the conditional PD using the maximum age observed in the training data. In particular, the predicted PD value is constant if the predictor values do not change and only the age values change when the `ExtrapolationFactor` is `1`.

To illustrate this, select the rows corresponding to a single ID and add new rows with new, incremental age values beyond the maximum observed age in the training data. The maximum age observed in the training data is 8; for illustration purposes, add rows with ages `9`, `10`, `11`, and `12`.

```
% Select rows corresponding to one ID
% ID 1 goes from row 1 through 8
% Only the ID, Age (YOB) and predictor variables are needed
dataNewAge = data(1:8,{'ID' 'YOB' 'ScoreGroup' 'GDP' 'Market'});
% Allocate more rows
% This line copies the same predictor values going forward
dataNewAge(9:12,:) = repmat(dataNewAge(8,:),4,1);
% Reset age values to 9, 10, 11, 12
dataNewAge.YOB(9:12) = (9:12)';
% Show the new dataset
disp(dataNewAge)
```

```
    ID    YOB    ScoreGroup     GDP     Market
    __    ___    __________    _____    ______

    1      1      Low Risk      2.72      7.61
    1      2      Low Risk      3.57     26.24
    1      3      Low Risk      2.86      18.1
    1      4      Low Risk      2.43      3.19
    1      5      Low Risk      1.26    -10.51
    1      6      Low Risk     -0.59    -22.95
    1      7      Low Risk      0.63      2.78
    1      8      Low Risk      1.85      9.48
    1      9      Low Risk      1.85      9.48
    1     10      Low Risk      1.85      9.48
    1     11      Low Risk      1.85      9.48
    1     12      Low Risk      1.85      9.48
```

When the predictor values are constant in the rows with new age values and the extrapolation factor is `1`, the predicted PD values are constant. If the extrapolation factor is set to a value smaller than `1`, then the predicted PD values decrease more and more for larger age values and decrease towards zero exponentially.

```
% Extrapolation factor can be adjusted

pdModel.ExtrapolationFactor =  1 ;
% Store predicted conditional PD in the same table
dataNewAge.PD = predict(pdModel,dataNewAge);
disp(dataNewAge)
```

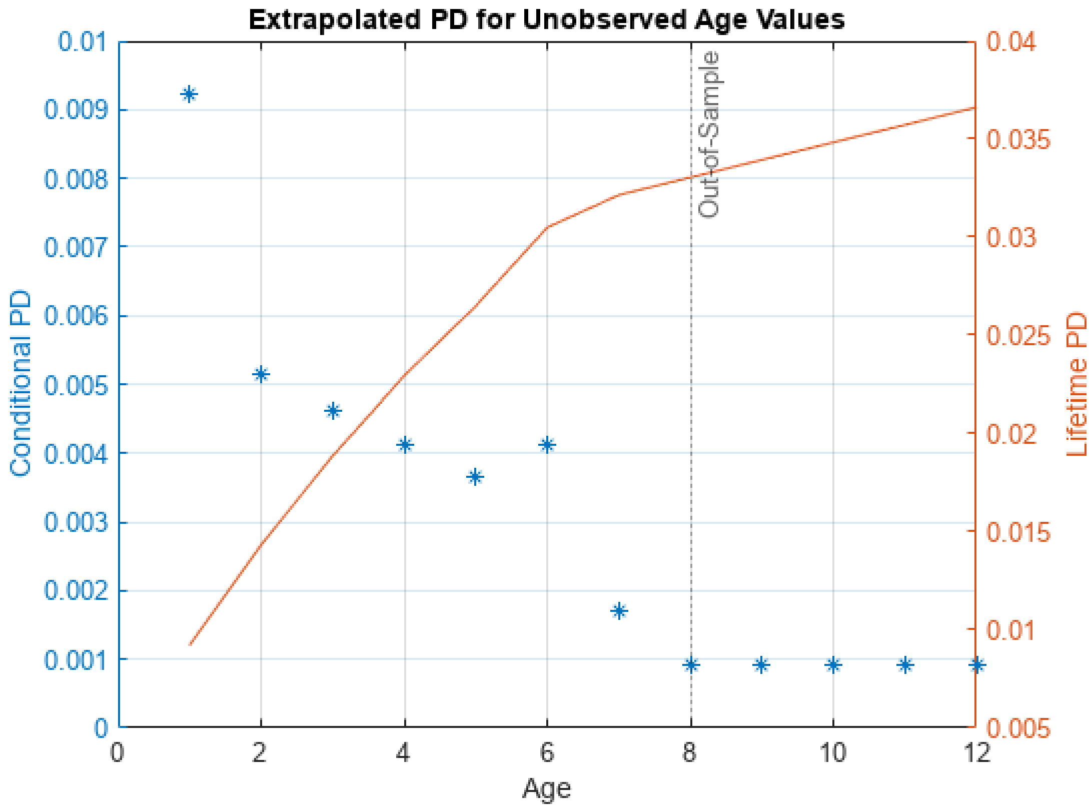
```
    ID    YOB    ScoreGroup     GDP     Market        PD
    __    ___    __________    _____    ______    __________
```

```
    1       1       Low Risk      2.72       7.61      0.0092197
    1       2       Low Risk      3.57      26.24       0.005158
    1       3       Low Risk      2.86       18.1      0.0046079
    1       4       Low Risk      2.43       3.19      0.0041351
    1       5       Low Risk      1.26     -10.51       0.003645
    1       6       Low Risk     -0.59     -22.95      0.0041128
    1       7       Low Risk      0.63       2.78      0.0017034
    1       8       Low Risk      1.85       9.48     0.00092551
    1       9       Low Risk      1.85       9.48     0.00092551
    1      10       Low Risk      1.85       9.48     0.00092551
    1      11       Low Risk      1.85       9.48     0.00092551
    1      12       Low Risk      1.85       9.48     0.00092551
```

Also, it is useful to see the effect of the extrapolation factor on the lifetime prediction.

Plot the predicted conditional PD values and the lifetime PD values to see the effect of the extrapolation factor on both probabilities. The vertical dotted line separates the known age values (up to, and including, the age value `8`), from the age values not observed in the training data (anything greater than `8`). If the extrapolation factor is `1`, the lifetime PD has a steady upward trend and the conditional PDs are constant. If the extrapolation factor is set to a smaller value like `0.5`, the lifetime PD flattens quickly, as the conditional PD quickly drops towards zero.

```
dataNewAge.LifetimePD = predictLifetime(pdModel,dataNewAge);

figure;
yyaxis left
plot(dataNewAge.YOB,dataNewAge.PD,'*')
ylabel('Conditional PD')
yyaxis right
plot(dataNewAge.YOB,dataNewAge.LifetimePD)
ylabel('Lifetime PD')
title('Extrapolated PD for Unobserved Age Values')
xlabel('Age')
xline(8,':','Out-of-Sample')
grid on
```



Input Arguments

pdModel — Probability of default model
Logistic object | Probit object | Cox object | customLifetimePDModel object

Probability of default model, specified as a previously created `Logistic`, `Probit`, or `Cox` object using `fitLifetimePDModel`. Alternatively, you can create a custom probability of default model using `customLifetimePDModel`.

Data Types: object

data — Data
table

Data, specified as a `NumRows`-by-`NumCols` table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model.

Data Types: table

Output Arguments

conditionalPD — Predicted conditional probability of default values
vector

Predicted conditional probability of default values, returned as a `NumRows`-by-`1` numeric vector.

More About

Conditional PD

Conditional PD is the probability of defaulting, given no default yet.

For example, the predicted conditional PD for the second year is the probability that the borrower defaults in the second year, given that the borrower did not default in the first year.

The formula for conditional PD is

$$PD(t) = P\{t - \Delta t < T \leq t | T > t - \Delta t\}$$

where

- T is the time to default.
- Δt is the "time interval" consistent with the periodicity of the panel training `data` (for example, one row per year) and the definition of the default indicator values.

The default indicator is `1` if there is a default over a 1-year period. For more information on time intervals, see “Time Interval for Logistic Models” on page 6-623, “Time Interval for Probit Models” on page 6-634, and “Time Interval for Cox Models” on page 6-551.

In the formulas that follow for `Logistic`, `Probit`, and `Cox` models, the notation is:

- $X(t)$ is the predictor data for the row corresponding to time t.
- β is the vector of coefficients of the underlying model.

For `Logistic` models, the conditional PD is computed as:

$$PD_{cond}(t) = \frac{1}{1 + \exp(-X(t)\beta)}$$

For `Probit` models, the conditional PD is computed as:

$$PD_{cond}(t) = \phi(X(t)\beta)$$

For `Cox` models, the conditional PD is computed as

$$PD_{cond}(t) = 1 - \frac{S(t)}{S(t - \Delta t)}$$

where S is the survival function. The survival function depends on the predictor values through the hazard ratio. For more information, see “Cox Proportional Hazards Models” on page 6-550. There are different ways to represent the dependence of the PD on the predictors explicitly. The implementation in the `predict` function uses the baseline cumulative hazard rate function given by

$$H_0(t) = \int_0^t h_0(u)du$$

where h_0 is the baseline hazard rate. For more information, see “Cox Proportional Hazards Models” on page 6-550. Using the baseline cumulative hazard rate, the PD formula for the `Cox` model is written as:

$$PD_{cond}(t) = 1 - \exp(-(H_0(t) - H_0(t - \Delta t))\exp(X(t)\beta))$$

Extrapolation for Cox Models

The baseline cumulative hazard function H_0 for `Cox` models is fitted to the observed age values (that is, the observed "times-to-event") in a nonparametric way.

Therefore, some form of interpolation or extrapolation is needed to make predictions for age values not observed in the training `data`. In the `predict` function, linear interpolation is used as follows:

- If the known age values are t_1, t_2,...,t_N, with t_i - t_{i-1} = Δt, and if $t_0 = t_1 - \Delta t$, then:
  - $H_0(t) = 0$, for all $t \leq t_0$.
  - $H_0(t)$ is interpolated linearly for $t_{i-1} \leq t \leq t_i$, for $i = 0,...N$.
  - $H_0(t)$ is extrapolated linearly for $t > t_N$, following the slope defined by the last two known values $H_0(t_{N-1})$ and $H_0(t_N)$.

This implies the baseline hazard rate h_0 is piecewise constant and remains constant after the last fitted value. By default, after the last known age value, the PD is evaluated as follows

$$PD_{cond}(t|X(t)) = PD_{cond}(t_N|X(t))$$

for $t > t_N$. This behavior is adjusted with the `ExtrapolationFactor` property of the `Cox` model. For more information, see “Use Cox Lifetime PD Model to Predict Conditional PD” on page 6-329.

Extrapolation Factor for Cox Models

The extrapolation formula implemented in the `predict` function includes the `ExtrapolationFactor` property value

$$PD_{cond}(t_{N+k}\Big|X(t_{N+k})) = (ExtrapolationFactor)^k PD_{cond}(t_N\Big|X(t_{N+k}))$$

where t_{N+k} is the time value k periods after the largest age observed in the training data t_N, that is, $t_{N+k} = t_N + k^* \Delta t$.

By default, the extrapolation factor is `1`, resulting in the formula in the “Extrapolation for Cox Models” on page 6-335 section, where the PD values remain constant as the age increases — if the predictor values do not change. If the extrapolation factor is set to a value smaller than `1`, the predicted PD values decrease exponentially towards `0`. The smaller the factor, the faster the conditional PD values decrease, and the faster the lifetime PD values flatten out.

In general, PD values tend to go down towards the end of the life of a loan, since the pool of borrowers gets cured earlier on. How fast this happens depends on the product and must be calibrated on a case-by-case basis.

Note that `Logistic` and `Probit` models need no special considerations regarding interpolation or extrapolation. These models are fully parametric models and predict the conditional PD for any values, in between, or beyond the numeric values observed in the dataset.

Version History

Introduced in R2020b

R2022b: Support for `customLifetimePDModel` model

The `pdModel` input supports an option for a `customLifetimePDModel` model object that you can create using `customLifetimePDModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot` | `predictLifetime` | `fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

predictLifetime

Compute cumulative lifetime PD, marginal PD, and survival probability

Syntax

```
LifeTimePredictedPD = predictLifetime(pdModel,data)
LifeTimePredictedPD = predictLifetime( ___ ,Name,Value)
```

Description

`LifeTimePredictedPD = predictLifetime(pdModel,data)` computes the cumulative lifetime probability of default (PD), marginal PD, and survival probability.

`LifeTimePredictedPD = predictLifetime( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Use Probit Lifetime PD Model to Predict Lifetime PD

This example shows how to use `fitLifetimePDModel` to fit data with a `Probit` model and then predict the lifetime probability of default (PD).

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
```

```
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Probit Lifetime PD Model

Use `fitLifetimePDModel` to create a `Probit` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Probit",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
disp(pdModel)
```

```
  Probit with properties:

            ModelID: "Probit"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
```

```
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Display the underlying model.

```
disp(pdModel.Model)
```

```
Compact generalized linear regression model:
    probit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate         SE         tStat       pValue   
                              __________    _________    ________    ___________

    (Intercept)                  -1.6267      0.03811     -42.685              0
    ScoreGroup_Medium Risk      -0.26542      0.01419     -18.704     4.5503e-78
    ScoreGroup_Low Risk         -0.46794     0.016364     -28.595     7.775e-180
    YOB                         -0.11421    0.0049724     -22.969    9.6208e-117
    GDP                        -0.041537     0.014807     -2.8052      0.0050291
    Market                    -0.0029609    0.0010618     -2.7885      0.0052954


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Predict Lifetime PD on Training and Test Data

Use the `predictLifetime` function to get lifetime PDs on the training or the test data. To get conditional PDs, use the `predict` function. For model validation, use the `modelDiscrimination` and `modelCalibration` functions on the training or test data.

```
DataSetChoice = Testing ;
if DataSetChoice=="Training"
    Ind = TrainDataInd;
else
    Ind = TestDataInd;
end

% Predict lifetime PD
PD = predictLifetime(pdModel,data(Ind,:));
head(data(Ind,:))
```

```
    ID     ScoreGroup      YOB    Default    Year     GDP     Market
    __    ___________    ___    _______    ____    _____    ______

    2     Medium Risk     1        0       1997     2.72      7.61
    2     Medium Risk     2        0       1998     3.57     26.24
    2     Medium Risk     3        0       1999     2.86      18.1
    2     Medium Risk     4        0       2000     2.43      3.19
    2     Medium Risk     5        0       2001     1.26    -10.51
    2     Medium Risk     6        0       2002    -0.59    -22.95
    2     Medium Risk     7        0       2003     0.63      2.78
    2     Medium Risk     8        0       2004     1.85      9.48
```

Predict Lifetime PD on New Data

Lifetime PD models are used to make predictions on existing loans. The `predictLifetime` function requires projected values for both the loan and macro predictors for the remainder of the life of the loan.

The `DataPredictLifetime.mat` file contains projections for two loans and also for the macro variables. One loan is three years old at the end of 2019, with a lifetime of 10 years, and the other loan is six years old with a lifetime of 10 years. The `ScoreGroup` is constant and the age values are incremental. For the macro variables, the forecasts for the macro predictors must span the longest lifetime in the portfolio.

```
load DataPredictLifetime.mat
disp(LoanData)
```

```
     ID       ScoreGroup       YOB     Year
    ____    _____________    ___    ____

    1304    "Medium Risk"     4     2020
    1304    "Medium Risk"     5     2021
    1304    "Medium Risk"     6     2022
    1304    "Medium Risk"     7     2023
    1304    "Medium Risk"     8     2024
    1304    "Medium Risk"     9     2025
    1304    "Medium Risk"    10     2026
    2067    "Low Risk"        7     2020
    2067    "Low Risk"        8     2021
    2067    "Low Risk"        9     2022
    2067    "Low Risk"       10     2023
```

```
disp(MacroScenario)
```

```
    Year    GDP    Market
    ____    ___    ______

    2020    1.1     4.5
    2021    0.9     1.5
    2022    1.2       5
    2023    1.4     5.5
    2024    1.6       6
    2025    1.8     6.5
    2026    1.8     6.5
    2027    1.8     6.5
```

```
LifetimeData = join(LoanData,MacroScenario);
disp(LifetimeData)
```

```
     ID       ScoreGroup       YOB     Year    GDP    Market
    ____    _____________    ___    ____    ___    ______

    1304    "Medium Risk"     4     2020    1.1     4.5
    1304    "Medium Risk"     5     2021    0.9     1.5
    1304    "Medium Risk"     6     2022    1.2       5
    1304    "Medium Risk"     7     2023    1.4     5.5
    1304    "Medium Risk"     8     2024    1.6       6
    1304    "Medium Risk"     9     2025    1.8     6.5
    1304    "Medium Risk"    10     2026    1.8     6.5
    2067    "Low Risk"        7     2020    1.1     4.5
```

```
    2067    "Low Risk"        8     2021    0.9     1.5
    2067    "Low Risk"        9     2022    1.2       5
    2067    "Low Risk"       10     2023    1.4     5.5
```

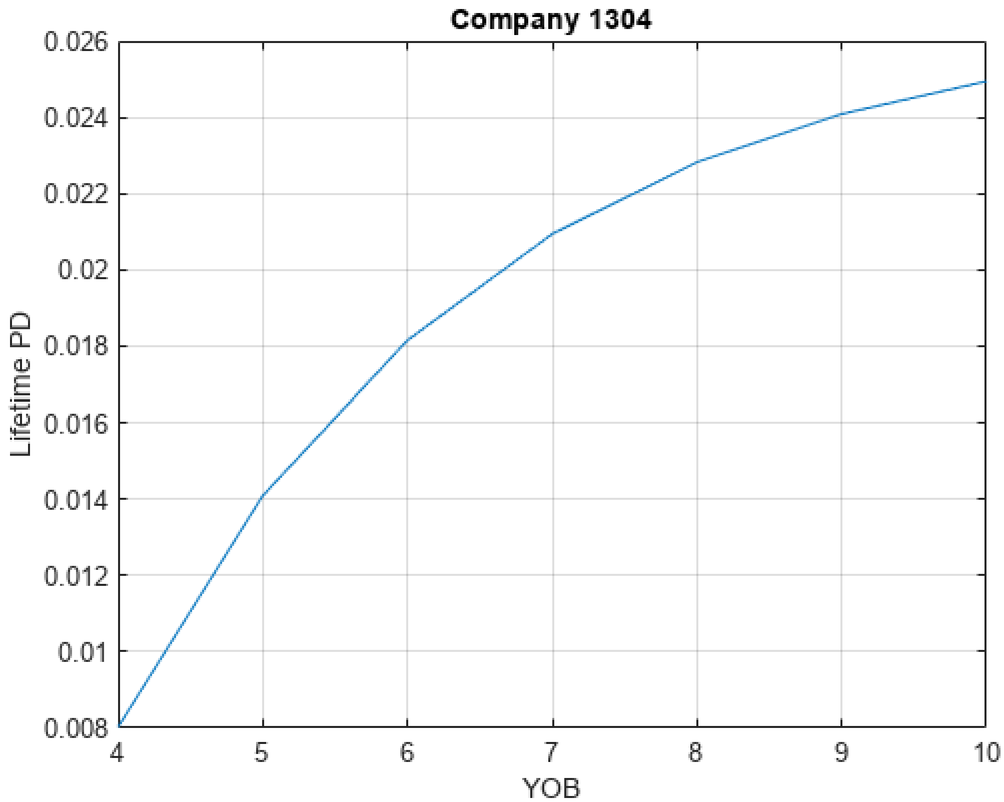
Predict lifetime PDs and store the output as a new table column for convenience.

```
LifetimeData.PredictedPD = predictLifetime(pdModel,LifetimeData);
disp(LifetimeData)
```

```
     ID       ScoreGroup       YOB    Year    GDP    Market    PredictedPD
    ____    _____________     ___    ____    ___    ______    ___________

    1304    "Medium Risk"      4     2020    1.1     4.5       0.0080202
    1304    "Medium Risk"      5     2021    0.9     1.5        0.014093
    1304    "Medium Risk"      6     2022    1.2       5        0.018156
    1304    "Medium Risk"      7     2023    1.4     5.5        0.020941
    1304    "Medium Risk"      8     2024    1.6       6        0.022827
    1304    "Medium Risk"      9     2025    1.8     6.5        0.024086
    1304    "Medium Risk"     10     2026    1.8     6.5        0.024945
    2067    "Low Risk"         7     2020    1.1     4.5       0.0015728
    2067    "Low Risk"         8     2021    0.9     1.5       0.0027146
    2067    "Low Risk"         9     2022    1.2       5        0.003431
    2067    "Low Risk"        10     2023    1.4     5.5       0.0038939
```

Visualize the predicted lifetime PD for a company.

```
CompanyIDChoice = 1304;
CompanyID = str2double(CompanyIDChoice);
IndPlot = LifetimeData.ID==CompanyID;
plot(LifetimeData.YOB(IndPlot),LifetimeData.PredictedPD(IndPlot))
grid on
xlabel('YOB')
xticks(LifetimeData.YOB(IndPlot))
ylabel('Lifetime PD')
title(strcat("Company ",CompanyIDChoice))
```



Lifetime Prediction and Time Interval

This example shows how time interval plays an important role for lifetime prediction when using a `Logistic`, `Probit`, or `Cox` model for probability of default (PD). Each PD value is a probability of default for the given "time interval" (for example, a time interval of 1 year), The data rows passed in for lifetime prediction must have the same periodicity as the time interval (that is, you can't pass a row that represents a quarter, and then a row that represents a year, and then one that represents 5 years. You must pass data for periods 1, 2, 3, 4,..., but not 1, 3, 7, 10, 20. Or if the time interval is 3, you must pass periods 3, 6, 9,... or 2, 5, 8,..., but not 3, 7, 15, 30.

Fit and Validate Model

```
load RetailCreditPanelData.mat
data = join(data,dataMacro);
head(data)
```

ID	ScoreGroup	YOB	Default	Year	GDP	Market
1	Low Risk	1	0	1997	2.72	7.61
1	Low Risk	2	0	1998	3.57	26.24
1	Low Risk	3	0	1999	2.86	18.1
1	Low Risk	4	0	2000	2.43	3.19
1	Low Risk	5	0	2001	1.26	-10.51
1	Low Risk	6	0	2002	-0.59	-22.95

```
    1      Low Risk      7        0       2003      0.63       2.78
    1      Low Risk      8        0       2004      1.85       9.48
```

Select a model type. The behavior of the data validation in `predictLifetime` depends on the model type. For more information, see "Validation of Data Input for Lifetime Prediction" on page 6-349.

The time interval in this example is `1`. This value is stored in `Cox` models as the `TimeInterval` property and it is used for fitting and prediction.`Logistic` and `Probit` models do not store the time interval information.

```
ModelType = Cox;

pdModel = fitLifetimePDModel(data,ModelType,...
   'IDVar','ID','AgeVar','YOB',...
   'LoanVars','ScoreGroup','MacroVars',{'GDP' 'Market'},...
   'ResponseVar','Default');
disp(pdModel)
```

```
  Cox with properties:

            TimeInterval: 1
     ExtrapolationFactor: 1
                 ModelID: "Cox"
             Description: ""
         UnderlyingModel: [1x1 CoxModel]
                   IDVar: "ID"
                  AgeVar: "YOB"
                LoanVars: "ScoreGroup"
               MacroVars: ["GDP"    "Market"]
             ResponseVar: "Default"
```

Conditional PD and Model Validation

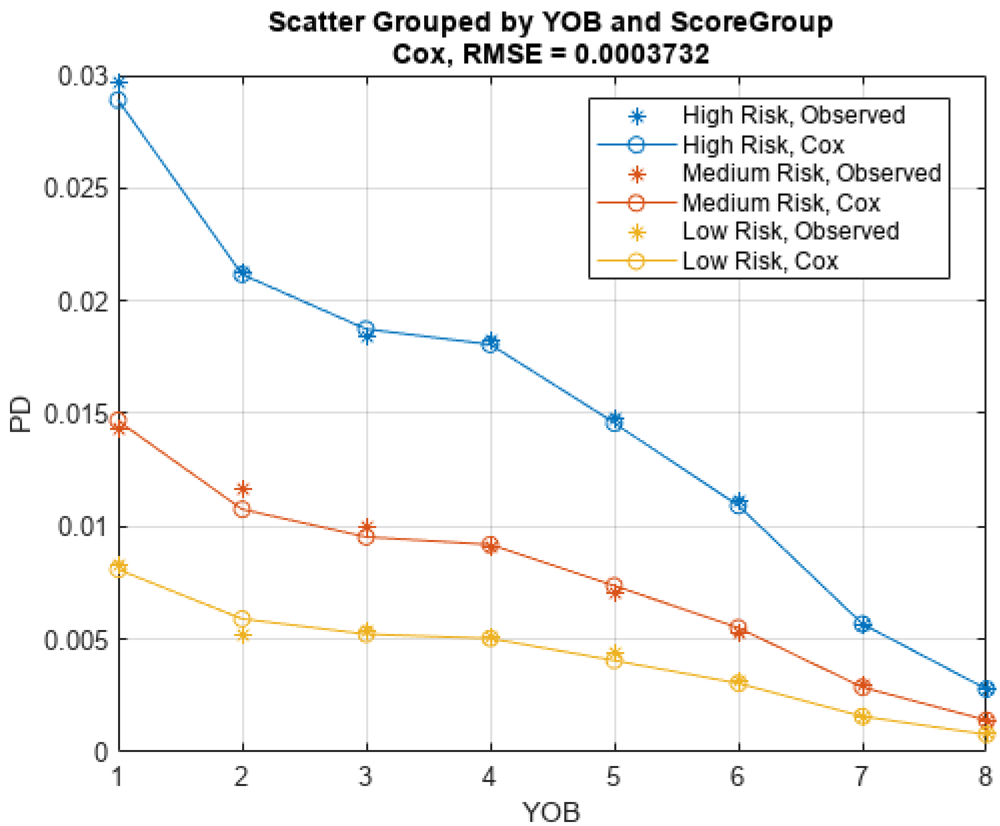
The conditional PD values returned by `predict` are consistent with the time interval used for training the model. In this example, all PD values returned by `predict` are 1-year probabilities of default. There is no validation of the periodicity in the data input for `predict`.

```
dataPredictExample = data([1 2 6 10 15],:);
pdExample = predict(pdModel,dataPredictExample)
```

```
pdExample = 5×1

    0.0089
    0.0052
    0.0038
    0.0094
    0.0031
```

Model validation is done using the conditional PD returned by `predict`. Therefore, there is no row periodicity validation in `modelDiscrimination` or `modelCalibration`. However, model validation requires observed values of the response variable, and the definition of default used for the validation response values must be consistent with the training `data`. In other words, if the training `data` uses a time interval of `1`, the validation response data cannot be defined with quarterly default data. There are no row-periodicity checks for `modelDiscrimination` or `modelCalibration`, it is assumed that the default definition in the validation data is consistent with the training data.

```
modelCalibrationPlot(pdModel,data,{'YOB','ScoreGroup'})
```



Lifetime PD

The `predictLifetime` function is used to compute lifetime PD. When making lifetime predictions:

- A different data set is likely used, not the data you used for training and validation, but a new data set with forward-looking projections for different loans.
- The projected values in the lifetime prediction data set span several periods ahead, potentially several years ahead.

Load the `DataPredictLifetime.mat` data for lifetime prediction. Note that for prediction, you don't need to pass the response data, you only pass predictors. You only pass response values for fitting or validation, not for prediction.

```
load DataPredictLifetime.mat
LifetimeData = join(LoanData,MacroScenario);
disp(LifetimeData)
```

```
     ID       ScoreGroup      YOB    Year    GDP    Market
    ____    _____________    ___    ____    ___    ______

    1304    "Medium Risk"     4     2020    1.1     4.5  
    1304    "Medium Risk"     5     2021    0.9     1.5  
    1304    "Medium Risk"     6     2022    1.2       5  
    1304    "Medium Risk"     7     2023    1.4     5.5  
    1304    "Medium Risk"     8     2024    1.6       6  
```

```
    1304    "Medium Risk"     9     2025    1.8     6.5
    1304    "Medium Risk"    10     2026    1.8     6.5
    2067    "Low Risk"        7     2020    1.1     4.5
    2067    "Low Risk"        8     2021    0.9     1.5
    2067    "Low Risk"        9     2022    1.2       5
    2067    "Low Risk"       10     2023    1.4     5.5
```

The rows have yearly data, consistent with the time interval used for training. You can see this in both the `Year` variable and the `YOB` variable. There are no flags in this data set for lifetime predictions.

```
LifetimeData.PD = predict(pdModel,LifetimeData);
LifetimeData.LifetimePD = predictLifetime(pdModel,LifetimeData)
```

```
LifetimeData=11×8 table
     ID       ScoreGroup      YOB    Year    GDP    Market        PD          LifetimePD
    ____    _____________    ___    ____    ___    ______    __________    __________

    1304    "Medium Risk"     4     2020    1.1     4.5       0.0081336     0.0081336
    1304    "Medium Risk"     5     2021    0.9     1.5       0.0063861      0.014468
    1304    "Medium Risk"     6     2022    1.2       5       0.0047416      0.019141
    1304    "Medium Risk"     7     2023    1.4     5.5       0.0028262      0.021913
    1304    "Medium Risk"     8     2024    1.6       6       0.0014844      0.023365
    1304    "Medium Risk"     9     2025    1.8     6.5       0.0014517      0.024783
    1304    "Medium Risk"    10     2026    1.8     6.5       0.0014517      0.026198
    2067    "Low Risk"        7     2020    1.1     4.5       0.0016091     0.0016091
    2067    "Low Risk"        8     2021    0.9     1.5       0.0009006     0.0025082
    2067    "Low Risk"        9     2022    1.2       5      0.00085273     0.0033588
    2067    "Low Risk"       10     2023    1.4     5.5      0.00083391     0.0041899
```

When the periodicity of the rows does not match the periodicity in the training data, the lifetime PD values cannot be correctly computed.

Modify the selected rows using the `SelectedRows` variable in the code to see the behavior of `predictLifetime` as the periodicity of the data changes. (Alternatively, the `YOB` values can be manually modified to enter age increments inconsistent with the time interval of 1 year.)

```
RowSelection = All rows ;
switch RowSelection
   case "All rows"
      SelectedRows = 1:11; % Selecting all rows 1:11 is the same as the output above, no warnings
   case "Every other row"
      SelectedRows = 1:2:11; % Regular age increments, but skipping one year
   case "Irregular"
      SelectedRows = [1 2 7 8 11]; % Irregular age increments
end
LifetimeData2 = LifetimeData(SelectedRows,{'ID','ScoreGroup','YOB','Year','GDP','Market'});
disp(LifetimeData2)
```

```
     ID       ScoreGroup      YOB    Year    GDP    Market
    ____    _____________    ___    ____    ___    ______

    1304    "Medium Risk"     4     2020    1.1     4.5
    1304    "Medium Risk"     5     2021    0.9     1.5
    1304    "Medium Risk"     6     2022    1.2       5
    1304    "Medium Risk"     7     2023    1.4     5.5
    1304    "Medium Risk"     8     2024    1.6       6
```

```
    1304    "Medium Risk"     9     2025    1.8     6.5
    1304    "Medium Risk"    10     2026    1.8     6.5
    2067    "Low Risk"        7     2020    1.1     4.5
    2067    "Low Risk"        8     2021    0.9     1.5
    2067    "Low Risk"        9     2022    1.2       5
    2067    "Low Risk"       10     2023    1.4     5.5
```

```
LifetimeData2.PD = predict(pdModel,LifetimeData2);
LifetimeData2.LifetimePD = predictLifetime(pdModel,LifetimeData2);
disp(LifetimeData2)
```

```
     ID      ScoreGroup      YOB    Year    GDP    Market        PD         LifetimePD
    ____    _____________    ___    ____    ___    ______    __________    __________

    1304    "Medium Risk"     4     2020    1.1     4.5       0.0081336    0.0081336
    1304    "Medium Risk"     5     2021    0.9     1.5       0.0063861     0.014468
    1304    "Medium Risk"     6     2022    1.2       5       0.0047416     0.019141
    1304    "Medium Risk"     7     2023    1.4     5.5       0.0028262     0.021913
    1304    "Medium Risk"     8     2024    1.6       6       0.0014844     0.023365
    1304    "Medium Risk"     9     2025    1.8     6.5       0.0014517     0.024783
    1304    "Medium Risk"    10     2026    1.8     6.5       0.0014517     0.026198
    2067    "Low Risk"        7     2020    1.1     4.5       0.0016091    0.0016091
    2067    "Low Risk"        8     2021    0.9     1.5       0.0009006    0.0025082
    2067    "Low Risk"        9     2022    1.2       5      0.00085273    0.0033588
    2067    "Low Risk"       10     2023    1.4     5.5      0.00083391    0.0041899
```

The differences in behavior depend on the model type and whether the age variable is part of the model. You can change the model type in the fitting step to see the behavior for different model types. Remove the age variable (`AgeVar`) for `Logistic` and `Probit` models to observe the behavior when an age input argument is not part of the model. Note that an age input (`AgeVar`) argument is required for a `Cox` model. For more information, see “Time Interval and Data Input for Lifetime Prediction” on page 6-348.

Input Arguments

`pdModel` — Probability of default model

`Logistic` object | `Probit` object | `Cox` object | `customLifetimePDModel` object

Probability of default model, specified as a previously created `Logistic`, `Probit`, or `Cox` object using `fitLifetimePDModel`. Alternatively, you can create a custom probability of default model using `customLifetimePDModel`.

Data Types: `object`

`data` — Lifetime data

table

Lifetime data, specified as a `NumRows`-by-`NumCols` table with projected predictor values to make lifetime predictions. The predictor names and data types must be consistent with the underlying model. The `IDVar` property of the `pdModel` input is used to identify the column containing the ID values in the table, and the IDs are used to identify rows corresponding to the different IDs and to make lifetime predictions for each ID.

Note

- Rows passed in data for lifetime prediction must have the same periodicity as the time interval used to fit the model. For example, if the time interval used for training was one year, the data input for lifetime prediction cannot have quarterly data, or data for every five years.
- Consecutive rows for the same ID *must* correspond to consecutive periods. For example, if the time interval used for training was one year, you cannot skip years and pass data for years 1, 2, 5, and 10.

For more information, see “Data Input for Lifetime Prediction” on page 6-348 and “Time Interval and Data Input for Lifetime Prediction” on page 6-348.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `LifetimeData = predictLifetime(pdModel,Data,'ProbabilityType','survival')`

`ProbabilityType` — Probability type
`'cumulative'` (default) | character vector with value `'cumulative'`, `'marginal'`, or `'survival'` | string with value `"cumulative"`, `"marginal"`, or `"survival"`

Probability type, specified as the comma-separated pair consisting of `'ProbabilityType'` and a character vector or string.

Data Types: `char` | `string`

Output Arguments

`LifeTimePredictedPD` — Predicted lifetime PD values
vector

Predicted lifetime PD values, returned as a `NumRows`-by-`1` numeric vector.

More About

Lifetime PD

Lifetime PD is the probability of a default event over the lifetime of a financial asset.

Lifetime PD typically refers to the cumulative default probability, given by

$$PD_{cumulative}(t) = P\{T \leq t\}$$

where T is the time to default.

For example, the predicted lifetime, cumulative PD for the second year is the probability that the borrower defaults any time between now and two years from now.

A closely related concept used for the computation of the lifetime Expected Credit Loss (ECL) is the marginal PD, given by

$$PD_{marginal} = PD_{cumulative}(t) - PD_{cumulative}(t-1)$$

A closely related probability is the survival probability, which is the complement of the cumulative probability and is reported as

$$S(t) = P\{T > t\} = 1 - PD_{cumulative}(t)$$

The following recursive formula shows the relationship between the conditional PDs and the survival probability:

$$S(t_0) = 1$$

$$S(t_1) = S(t_0)(1 - PD(t_1))$$

...

$$S(t_n) = S(t_{n-1})(1 - PD(t_n))$$

Where t_i - t_i-1 = Δt for all i = 1,...,n, and Δt is the time interval used to fit the model. For more information, see "Time Interval for Logistic Models" on page 6-623 and "Time Interval for Probit Models" on page 6-634. In other words, because the PD values on the right-hand side of the formulas are probabilities of default for a period of length Δt, the increments between consecutive times in the recursion must always be of length Δt for all periods i = 1, 2,..., n.

The `predictLifetime` function calls the `predict` function to get the conditional PD and then converts it to survival, marginal, or lifetime cumulative PD using the previous formulas.

Data Input for Lifetime Prediction

Lifetime PD is the cumulative probability of default over multiple periods.

The input for the `predictLifetime` function should contain multiple rows per ID, where rows represent sequential time periods regularly spaced. In other words, the data should be in panel data form. The time interval between adjacent rows must be consistent with the time interval used to define the default binary variable in the training data. For more information, see "Time Interval and Data Input for Lifetime Prediction" on page 6-348.

If a dataset with one row per ID is passed, the output of `predictLifetime` is the same as the output of `predict` because the PD is predicted for one period only (see formulas in `predict` section). A dataset with multiple rows per ID allows `predictLifetime` to aggregate the default probability over multiple periods to get the cumulative PD.

The `predictLifetime` function is typically used for predictions on outstanding loans, where the predictor variable values must be projected, period by period, for several periods into the future. Although historical (training or testing) data sets in panel data form can be passed to `predictLifetime`, the typical workflow requires data preparation. It starts out with outstanding loans, where only the most recent values of the predictor variables are known. The data preparation then projects the predictor variable values into the future for multiple time periods, typically until the maturity of the loan for a lifetime analysis. For example, see "Create Custom Lifetime PD Model for Decision Tree Model with Function Handle" on page 4-224.

Time Interval and Data Input for Lifetime Prediction

The time interval used for fitting the model plays an important role for lifetime prediction.

The data input for `predictLifetime` is in panel data form, with multiple rows for each ID. There is an implicit or explicit time stamp for each row, and the time increments between consecutive rows must be the same as the time interval used to fit the model. For more information on time intervals, see "Time Interval for Cox Models" on page 6-551, "Time Interval for Logistic Models" on page 6-623, and "Time Interval for Probit Models" on page 6-634.

Following the notation of the lifetime PD recursive formulas described in "Lifetime PD" on page 6-347, the time stamps t_1, t_2,...,t_n between consecutive rows must satisfy $t_i - t_{i-1} = \Delta t$ for all $i = 1,...,n$, where Δt is the time interval used to fit the model. In other words:

- Rows passed in the `data` input for lifetime prediction must have the same periodicity as the time interval used to fit the model. For example, if the time interval used for training was 1 year, the `data` input for lifetime prediction cannot have quarterly data, or data for every 5 years.
- consecutive rows for the same ID *must* correspond to consecutive periods. For example, if the time interval used for training was 1 year, you cannot skip years and pass data for years 1, 2, 5, and 10.

Suppose, for concreteness, that the time interval Δt used to fit the model is 1 year. Then the PD values on the right-hand side of the formulas in "Lifetime PD" on page 6-347 are 1-year PDs. Therefore:

- Lifetime PD for quarterly data cannot be computed because S(`1.25`) ≠ S(*1*)(1 - PD(`1.25`)), since PD(`1.25`) is a 1-year PD spanning the default over the interval going from `0.25` to `1.25`.
- Lifetime PD for data every 5 years cannot be computed because S(`10`) ≠ S(`5`)(1 - PD(`10`)), since PD(`10`) is a 1-year PD spanning the default over the interval going from `9` to `10`.
- Lifetime PD for non-consecutive rows cannot be computed. For example, if the `data` input has rows corresponding to years 1, 2, 5 and 10, then S(`1`) and S(`2`) can be computed correctly, however S(`5`) ≠ S(`2`)(1-PD(`5`)) because PD(`5`) is a 1-year PD spanning the default over the interval going from `4` to `5`, and similarly for S(`10`).

Validation of Data Input for Lifetime Prediction

The validation of the row periodicity in the data input for `predictLifetime` depends on the model type (`ModelType`) and whether the model contains an age variable (`AgeVar`).

`Cox` models can validate the periodicity of the data because the age variable (`AgeVar`) is a required input argument and `Cox` models store the time interval (`TimeInterval`) used to fit the model. The `TimeInterval` is used both to fit the model and to predict PD values. For more information on time intervals for a `Cox` model, see "Time Interval for Cox Models" on page 6-551. The age variable (`AgeVar`) is used as the time dimension. For each ID, if the periodicity of the `data` input, measured by the increments in the age variable, does not match the time interval used to train the model, a warning is displayed and the lifetime PD values are filled with `NaN`s.

`Logistic` and `Probit` models do not store the time interval value. However the predicted PD values are still consistent with the (explicit or implicit) time interval in the training data. For more information, see "Time Interval for Logistic Models" on page 6-623 and "Time Interval for Probit Models" on page 6-634. Moreover, for `Logistic` and `Probit` models, the age variable (`AgeVar`) is optional, and there is no other way to specify a time dimension in the model. Therefore:

- If the `Logistic` or `Probit` model has no age variable information, there is no way to validate the periodicity of the data. The lifetime PD is computed using the recursion in "Lifetime PD" on page 6-347, assuming that the periodicity is correct. It is the responsibility of the caller to ensure that the periodicity of the data rows is consistent with the time interval in the training `data`.

- If the `Logistic` or `Probit` model has an age variable (`AgeVar`), this is used as a time dimension. However, because the time interval used to train the data is unknown for `Logistic` and `Probit` models, these models can only validate that the age increments are regular as follows, but cannot compare against a reference time interval.
  - For each ID, when the age shows irregular age increments, there is a warning and the lifetime PD values are set to `NaN`s.
  - When the age increments are regular within each ID, but some IDs have different age increments than others, a warning is displayed, but it is unknown which ID has the wrong increments. The lifetime PD values are computed using the recursion in “Lifetime PD” on page 6-347 for all IDs. It is the responsibility of the caller to ensure that the periodicity of the data rows for all IDs is consistent with the time interval in the training `data`.

For an example, see “Lifetime Prediction and Time Interval” on page 6-342.

Version History

Introduced in R2020b

R2022b: Support for `customLifetimePDModel` model

The `pdModel` input supports an option for a `customLifetimePDModel` model object that you can create using `customLifetimePDModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

`predict` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibration` | `modelCalibrationPlot` | `fitLifetimePDModel` | `Logistic` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

“Basic Lifetime PD Model Validation” on page 4-129
“Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
“Compare Lifetime PD Models Using Cross-Validation” on page 4-121
“Expected Credit Loss Computation” on page 4-124
“Compare Model Discrimination and Model Calibration to Validate of Probability of Default” on page 4-144
“Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models” on page 4-75

“Create Custom Lifetime PD Model for Decision Tree Model with Function Handle” on page 4-224
“Overview of Lifetime Probability of Default Models” on page 1-25

probdefault

Likelihood of default for given dataset for a `compactCreditScorecard` object

Syntax

```
pd = probdefault(csc,data)
```

Description

`pd = probdefault(csc,data)` computes the probability of default for the `compactCreditScorecard` (`csc`) based on the `data`.

Examples

Calculate the Probability of Default for a compactCreditScorecard Object with New Data

To create a `compactCreditScorecard` object, first create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData.mat
sc = creditscorecard(data)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBala
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
                     Data: [1200x11 table]
```

Before creating a `compactCreditScorecard` object, you must use `autobinning` and `fitmodel` with the `creditscorecard` object.

```
sc = autobinning(sc);
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769
```

```
Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Use the `creditscorecard` object with `compactCreditScorecard` to create a `compactCreditScorecard` object.

```
csc = compactCreditScorecard(sc)
```

```
csc = 
  compactCreditScorecard with properties:

              Description: ''
                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
        NumericPredictors: {'CustAge'  'CustIncome'  'TmWBank'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
            PredictorVars: {'CustAge'  'ResStatus'  'EmpStatus'  'CustIncome'  'TmWBank'  'Other
```

Then use `probdefault` with the `compactCreditScorecard` object. For the purpose of illustration, suppose that a few rows from the original data are our "new" data. Use the `data` input argument in the `probdefault` function to obtain the probability of default using the `newdata`.

```
newdata = data(10:20,:);
pd = probdefault(csc,newdata)
```

```
pd = 11×1

    0.3047
    0.3418
    0.2237
    0.2793
    0.3615
    0.1653
    0.3799
    0.4055
    0.4269
    0.1915
```

⋮

Input Arguments

csc — Compact credit scorecard model
compactCreditScorecard object

Credit scorecard model, specified as a compactCreditScorecard object.

To create a compactCreditScorecard object, use compactCreditScorecard or compact from Financial Toolbox.

data — Dataset to apply probability of default rules
table

Dataset to apply probability of default rules, specified as a MATLAB table, where each row corresponds to individual observations. The data must contain columns for each of the predictors in the compactCreditScorecard object.

Data Types: table

Output Arguments

pd — Probability of default
array

Probability of default, returned as a NumObs-by-1 numerical array of default probabilities.

More About

Default Probability

After the unscaled scores are computed (see “Algorithms for Computing and Scaling Scores”), the probability of the points being “Good” is represented by the following formula:

```
ProbGood = 1./(1 + exp(-UnscaledScores))
```

Thus, the probability of default is

```
pd = 1 - ProbGood
```

Version History

Introduced in R2019a

References

[1] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

compactCreditScorecard | score | displaypoints | validatemodel

Topics

“Case Study for Credit Scorecard Analysis”
“Credit Scorecard Modeling with Missing Values”
“Credit Scorecard Modeling Workflow”
“About Credit Scorecards”

quantile

Quantile expected shortfall (ES) backtest by Acerbi and Szekely

Syntax

```
TestResults = quantile(ebts)
[TestResults,SimTestStatistic] = quantile(ebts,Name,Value)
```

Description

`TestResults = quantile(ebts)` runs the quantile ES backtest of Acerbi-Szekely (2014).

`[TestResults,SimTestStatistic] = quantile(ebts,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Run an ES Quantile Test

Create an `esbacktestbysim` object.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

Generate the ES quantile test report.

```
TestResults = quantile(ebts)
```

```
TestResults=3×10 table
    PortfolioID        VaRID         VaRLevel     Quantile     PValue     TestStatistic     CriticalVa
    ___________    _____________    ________     ________     ______     _____________     __________

       "S&P"       "t(10) 95%"        0.95        reject      0.002        -0.10602          -0.05579
       "S&P"       "t(10) 97.5%"     0.975        reject          0        -0.15697          -0.07351
       "S&P"       "t(10) 99%"        0.99        reject          0        -0.26561           -0.1011
```

Input Arguments

`ebts` — `esbacktestbysim` object
object

esbacktestbysim (ebts) object, which contains a copy of the given data (the PortfolioData, VarData, ESData, and Distribution properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an esbacktestbysim object, see esbacktestbysim.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [TestResults,SimTestStatistic] = quantile(ebts,'TestLevel',0.99)

TestLevel — Test confidence level
0.95 (default) | numeric with values between 0 and 1

Test confidence level, specified as the comma-separated pair consisting of 'TestLevel' and a numeric value between 0 and 1.

Data Types: double

Output Arguments

TestResults — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR data columns provided
- 'VaRLevel' — VaR level for the corresponding VaR data column
- 'Quantile'— Categorical array with categories 'accept' and 'reject' indicating the result of the quantile test
- 'PValue'— *P*-value of the quantile test
- 'TestStatistic'— Quantile test statistic
- 'CriticalValue'— Critical value for the quantile test
- 'Observations'— Number of observations
- 'Scenarios'— Number of scenarios simulated to get the *p*-values
- 'TestLevel'— Test confidence level

SimTestStatistic — Simulated values of test statistic
numeric array

Simulated values of the test statistic, returned as a NumVaRs-by-NumScenarios numeric array.

More About

Quantile Test by Acerbi and Szekely

The quantile test (also known as the third Acerbi-Szekely test) uses a sample estimator of the expected shortfall.

The expected shortfall for a sample $Y_1,...,Y_N$ is:

$$\widehat{ES}(Y) = -\frac{1}{\lfloor Np_{VaR} \rfloor} \sum_{i=1}^{\lfloor Np_{VaR} \rfloor} Y_{[i]}$$

where

N is the number of periods in the test window (t = 1,...,N).

P_{VaR} is the probability of VaR failure defined as 1-VaR level.

$Y_{[1]},...,Y_{[N]}$ are the sorted sample values (from smallest to largest), and $\lfloor Np_{VaR} \rfloor$ is the largest integer less than or equal to Np_{VaR}.

To compute the quantile test statistic, a sample of size N is created at each time t as follows. First, convert the portfolio outcomes to X_t to ranks $U_1 = P_1(X_1), ..., U_N = P_N(X_N)$ using the cumulative distribution function P_t. If the distribution assumptions are correct, the rank values $U_1,...,U_N$ are uniformly distributed in the interval (0,1). Then at each time t:

- Invert the ranks U = $(U_1,...,U_N)$ to get N quantiles $P_t^{-1}(U) = (P_t^{-1}(U_1), ..., P_t^{-1}(U_N))$.
- Compute the sample estimator $\widehat{ES}(P_t^{-1}(U))$.
- Compute the expected value of the sample estimator $E\left[\widehat{ES}(P_t^{-1}(V))\right]$

  where V = $(V_1,...,V_N$ is a sample of N independent uniform random variables in the interval (0,1). This value can be computed analytically.

Define the quantile test statistic as

$$Z_{quantile} = -\frac{1}{N} \sum_{t=1}^{N} \frac{\widehat{ES}(P_t^{-1}(U))}{E[\widehat{ES}(P_t^{-1}(V))]} + 1$$

The denominator inside the sum can be computed analytically as

$$E[\widehat{ES}(P_t^{-1}(V))] = -\frac{N}{\lfloor N_{pVaR} \rfloor} \int_0^1 I_{1-p}(N - \lfloor N_{pVaR} \rfloor, \lfloor N_{pVaR} \rfloor) P_t^{-1}(p) dp$$

where $I_x(z,w)$ is the regularized incomplete beta function. For more information, see `betainc`.

Significance of the Test

Assuming that the distributional assumptions are correct, the expected value of the test statistic $Z_{quantile}$ is 0.

This is expressed as:

$$E[Z_{quantile}] = 0$$

Negative values of the test statistic indicate risk underestimation. The quantile test is a one-sided test that rejects the model when there is evidence that the model underestimates risk. (For technical details on the null and alternative hypotheses, see Acerbi-Szekely, 2014). The quantile test rejects the model when the *p*-value is less than `1` minus the test confidence level.

For more information on simulating the test statistics and computing the *p*-values and critical values, see `simulate`.

Edge Cases

The quantile test statistic is well-defined when there are no VaR failures in the data.

However, when the expected number of failures Np_{VaR} is small, an adjustment is required. The sample estimator of the expected shortfall takes the average of the smallest N_{tail} observations in the sample, where $N_{tail} = \lfloor N_{pVaR} \rfloor$. If $Np_{VaR} < 1$, then $N_{tail} = 0$, the sample estimator of the expected shortfall becomes an empty sum, and the quantile test statistic is undefined.

To account for this, whenever $Np_{VaR} < 1$, the value of N_{tail} is set to `1`. Thus, the sample estimator of the expected shortfall has a single term and is equal to the minimum value of the sample. With this adjustment, the quantile test statistic is then well-defined and the significance analysis is unchanged.

Version History

Introduced in R2017b

References

[1] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall*. MSCI Inc. December, 2014.

See Also

`summary` | `runtests` | `conditional` | `unconditional` | `simulate` | `minBiasRelative` | `minBiasAbsolute` | `esbacktestbysim` | `esbacktestbyde`

Topics

"Expected Shortfall (ES) Backtesting Workflow Using Simulation" on page 2-34
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

runtests

Run all tests in `varbacktest`

Syntax

```
TestResults = runtests(vbt)
TestResults = runtests(vbt,Name,Value)
```

Description

`TestResults = runtests(vbt)` runs all the tests in the `varbacktest` object. `runtests` reports only the final test result. For test details such as likelihood ratios, run individual tests:

- `tl` — Traffic light test
- `bin` — Binomial test
- `pof` — Proportion of failures
- `tuff` — Time until first failure
- `cc` — Conditional coverage mixed
- `cci` — Conditional coverage independence
- `tbf` — Time between failures mixed
- `tbfi` — Time between failures independence

`TestResults = runtests(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Run All VaR Backtests

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `TestResults` report for all VaR backtests.

```
TestResults = runtests(vbt,'TestLevel',0.99)
```

```
TestResults=1×11 table
    PortfolioID    VaRID    VaRLevel      TL       Bin       POF       TUFF       CC        CCI
    ___________    _____    ________    _____    ______    ______    ______    ______    ______

    "Portfolio"    "VaR"      0.95      green    accept    accept    accept    accept    accept
```

Generate the `TestResults` report for all VaR backtests using the name-value argument for `'ShowDetails'` to display the test confidence level.

```
TestResults = runtests(vbt,'TestLevel',0.99,"ShowDetails",true)
```

```
TestResults=1×12 table
    PortfolioID    VaRID    VaRLevel      TL       Bin       POF       TUFF       CC        CCI
    ___________    _____    ________    _____    ______    ______    ______    ______    ______

    "Portfolio"    "VaR"      0.95      green    accept    accept    accept    accept    accept
```

Run All VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object and run all tests.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99]);
    runtests(vbt)
```

```
ans=6×11 table
    PortfolioID        VaRID         VaRLevel      TL        Bin       POF       TUFF       CC
    ___________    ______________    ________    ______    ______    ______    ______    ______

     "Equity"      "Normal95"          0.95      green     accept    accept    accept    accept
     "Equity"      "Normal99"          0.99      yellow    reject    accept    accept    accept
     "Equity"      "Historical95"      0.95      green     accept    accept    accept    accept
     "Equity"      "Historical99"      0.99      green     accept    accept    accept    accept
     "Equity"      "EWMA95"            0.95      green     accept    accept    accept    accept
     "Equity"      "EWMA99"            0.99      yellow    reject    reject    accept    reject
```

Input Arguments

vbt — varbacktest object
object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = runtests(vbt,'TestLevel',0.99)`

`TestLevel` — Test confidence level
`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

`ShowDetails` — Indicates if the output displays a column showing the test confidence level
`false` (default) | scalar logical with a value of `true` or `false`

Indicates if the output displays a column showing the test confidence level, specified as the comma-separated pair consisting of `'ShowDetails'` and a scalar logical value.

Data Types: `logical`

Output Arguments

`TestResults` — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'TL'` — Categorical (ordinal) array with categories `green`, `yellow`, and `red` that indicate the result of the traffic light (`tl`) test
- `'Bin'` — Categorical array with categories `accept` and `reject` that indicate the result of the `bin` test
- `'POF'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `pof` test.
- `'TUFF'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `tuff` test
- `'CC'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `cc` test
- `'CCI'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `cci` test
- `'TBF'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `tbf` test

- `'TBFI'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `tbfi` test

Note For the test results, the terms `'accept'` and `'reject'` are used for convenience, technically a test does not accept a model. Rather, a test fails to reject it.

Version History

Introduced in R2016b

See Also

`varbacktest` | `tl` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `summary`

Topics

"VaR Backtesting Workflow" on page 2-6
"Value-at-Risk Estimation and Backtesting" on page 2-10
"Overview of VaR Backtesting" on page 2-2
"Comparison of ES Backtesting Methods" on page 2-26

runtests

Run all expected shortfall (ES) backtests for `esbacktest` object

Syntax

```
TestResults = runtests(ebt)
TestResults = runtests(ebt,Name,Value)
```

Description

`TestResults = runtests(ebt)` runs all the tests for the `esbacktest` object. `runtests` reports only the final test result. For test details, such as *p*-values, run the individual tests:

- `unconditionalNormal`
- `unconditionalT`

`TestResults = runtests(ebt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Run All ES Backtests

Create an `esbacktest` object.

```
load ESBacktestData
ebt = esbacktest(Returns,VaRModel1,ESModel1,'VaRLevel',VaRLevel)
```

```
ebt = 
  esbacktest with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x1 double]
           ESData: [1966x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9750
```

Generate the `TestResults` report for all ES backtests.

```
TestResults = runtests(ebt,'TestLevel',0.99)
```

```
TestResults=1×5 table
    PortfolioID    VaRID    VaRLevel    UnconditionalNormal    UnconditionalT
    ___________    _____    ________    ___________________    ______________

    "Portfolio"    "VaR"     0.975            reject               accept
```

Generate the TestResults report for all ES backtests using the name-value argument for 'ShowDetails' to display the test confidence level.

```
TestResults = runtests(ebt,'TestLevel',0.99,'ShowDetails',true)
```

```
TestResults=1×6 table
    PortfolioID    VaRID    VaRLevel    UnconditionalNormal    UnconditionalT    TestLevel
    ___________    _____    ________    ___________________    ______________    _________

    "Portfolio"    "VaR"     0.975            reject               accept           0.99
```

Input Arguments

ebt — esbacktest object
object

esbacktest (ebt) object, which contains a copy of the given data (the PortfolioData, VarData, and ESData properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an esbacktest object, see esbacktest.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: TestResults = runtests(ebt,'TestLevel',0.99)

TestLevel — Test confidence level
0.95 (default) | numeric value between 0.5 and 0.9999

Test confidence level, specified as the comma-separated pair consisting of 'TestLevel' and a numeric value between 0.5 and 0.9999.

Data Types: double

ShowDetails — Indicates if the output displays a column showing the test confidence level
false (default) | scalar logical with a value of true or false

Indicates if the output displays a column showing the test confidence level, specified as the comma-separated pair consisting of 'ShowDetails' and a scalar logical value.

Data Types: logical

Output Arguments

TestResults — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'UnconditionalNormal'`— Categorical array with categories 'accept' and 'reject' that indicate the result of the unconditional normal test
- `'UnconditionalT'` — Categorical array with categories 'accept' and 'reject' that indicate the result of the unconditional *t* test

Note For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

Version History

Introduced in R2017b

See Also

`esbacktest` | `summary` | `unconditionalNormal` | `unconditionalT`

Topics

"Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information" on page 2-30
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

runtests

Run all expected shortfall backtests (ES) for `esbacktestbysim` object

Syntax

```
TestResults = runtests(ebts)
TestResults = runtests(ebts,Name,Value)
```

Description

`TestResults = runtests(ebts)` runs all the tests for the `esbacktestbysim` object. `runtests` reports only the final test result. For test details, such as *p*-values, run the individual tests:

- `conditional`
- `unconditional`
- `quantile`

`TestResults = runtests(ebts,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Run All ES Backtests

Create an `esbacktestbysim` object.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

Generate the `TestResults` report for all ES backtests.

```
TestResults = runtests(ebts,'TestLevel',0.99)
```

```
TestResults=3×8 table
    PortfolioID        VaRID         VaRLevel     Conditional     Unconditional     Quantile     MinBi
    ___________    _____________    ________    ___________    _____________    ________    _____

       "S&P"       "t(10) 95%"         0.95        reject          accept          reject          a
       "S&P"       "t(10) 97.5%"      0.975        reject          accept          reject          a
       "S&P"       "t(10) 99%"         0.99        reject          reject          reject          r
```

Generate the `TestResults` report for all ES backtests using the name-value argument for `'ShowDetails'` to display the test confidence level.

```
TestResults = runtests(ebts,'TestLevel',0.99,'ShowDetails',true)

TestResults=3×9 table
    PortfolioID         VaRID          VaRLevel     Conditional     Unconditional     Quantile     MinBi
    ___________    _____________    ________    ___________    _____________    ________    _____

       "S&P"       "t(10) 95%"        0.95         reject           accept          reject          a
       "S&P"       "t(10) 97.5%"     0.975         reject           accept          reject          a
       "S&P"       "t(10) 99%"        0.99         reject           reject          reject          r
```

Input Arguments

`ebts` — `esbacktestbysim` object
object

`esbacktestbysim` (`ebts`) object, which contains a copy of the given data (the `PortfolioData`, `VarData`, `ESData`, and `Distribution` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktestbysim` object, see `esbacktestbysim`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = runtests(ebts,'TestLevel',0.99)`

`TestLevel` — Test confidence level
`0.95` (default) | numeric value between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric value between `0` and `1`.

Data Types: `double`

`ShowDetails` — Indicates if the output displays a column showing the test confidence level
`false` (default) | scalar logical with a value of `true` or `false`

Indicates if the output displays a column showing the test confidence level, specified as the comma-separated pair consisting of `'ShowDetails'` and a scalar logical value.

Data Types: `logical`

Output Arguments

`TestResults` — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'Conditional'`— Categorical array with categories 'accept' and 'reject' indicating the result of the conditional test
- `'Unconditional'` — Categorical array with categories 'accept' and 'reject' indicating the result of the unconditional test
- `'Quantile'` — Categorical array with categories 'accept' and 'reject' indicating the result of the quantile test
- `'minBiasAbsolute'` — Categorical array with categories 'accept' and 'reject' indicating the result of the `minBiasAbsolute` test
- `'minBiasRelative'` — Categorical array with categories 'accept' and 'reject' indicating the result of the `minBiasRelative` test

---

Note If you request to show additional details by setting the `ShowDetails` optional input to `true`, then the output also contains a TestLevel column for the confidence level.

For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

Version History

Introduced in R2017b

See Also

`summary` | `conditional` | `unconditional` | `quantile` | `simulate` | `minBiasRelative` | `minBiasAbsolute` | `esbacktestbysim` | `esbacktestbyde`

Topics

“Expected Shortfall (ES) Backtesting Workflow Using Simulation” on page 2-34
“Expected Shortfall Estimation and Backtesting” on page 2-44
“Overview of Expected Shortfall Backtesting” on page 2-20
“Comparison of ES Backtesting Methods” on page 2-26

runtests

Run all expected shortfall (ES) backtests for `esbacktestbyde` object

Syntax

```
TestResults = runtests(ebtde)
TestResults = runtests( ___ ,Name,Value)
```

Description

`TestResults = runtests(ebtde)` runs all the tests for the `esbacktestbyde` object. `runtests` reports only the final test result. For test details such as *p*-values, run the individual tests:

- `unconditionalDE`
- `conditionalDE`

`TestResults = runtests( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input argument in the previous syntax.

Examples

Create an esbacktestbyde Object and Run ES Backtests

Create an `esbacktestbyde` object for a *t* model with 10 degrees of freedom, and then run ES backtests.

```
load ESBacktestDistributionData.mat
    rng('default'); % For reproducibility
    ebtde = esbacktestbyde(Returns,"t",...
       'DegreesOfFreedom',T10DoF,...
       'Location',T10Location,...
       'Scale',T10Scale,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
    runtests(ebtde)
```

```
ans=3×5 table
    PortfolioID        VaRID         VaRLevel     ConditionalDE     UnconditionalDE
    ___________    _____________    ________    _____________    _______________

       "S&P"       "t(10) 95%"        0.95         reject             accept
       "S&P"       "t(10) 97.5%"     0.975         reject             accept
       "S&P"       "t(10) 99%"        0.99         reject             reject
```

To view complete details for the tests, use the name-value pair argument `'ShowDetails'`.

```
runtests(ebtde,'ShowDetails',true)
```

```
ans=3×8 table
    PortfolioID        VaRID         VaRLevel     ConditionalDE     UnconditionalDE     CriticalValue
```

```
    ___________     _____________     ________     _____________     _______________     _____________

       "S&P"        "t(10) 95%"        0.95          reject              accept            "large-samp
       "S&P"        "t(10) 97.5%"     0.975          reject              accept            "large-samp
       "S&P"        "t(10) 99%"        0.99          reject              reject            "large-samp
```

Input Arguments

ebtde — esbacktestbyde object
object

`esbacktestbyde` object, which contains a copy of the data (the `PortfolioData`, `VarData`, and `ESData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktestbyde` object, see `esbacktestbyde`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = runtests(ebtde,'CriticalValueMethod','simulation','TestLevel',0.99,'ShowDetails',true)`

CriticalValueMethod — Method to compute critical values, confidence intervals, and *p*-values
`'large-sample'` (default) | character vector with values of `'large-sample'` or `'simulation'` | string with values of `"large-sample"` or `"simulation"`

Method to compute critical values, confidence intervals, and *p*-values, specified as the comma-separated pair consisting of `'CriticalValueMethod'` and character vector or string with a value of `'large-sample'` or `'simulation'`.

Data Types: `char` | `string`

NumLags — Number of lags in the conditionalDE test
`1` (default) | positive integer

Number of lags in the `conditionalDE` test, specified as the comma-separated pair consisting of `'NumLags'` and a positive integer.

Data Types: `double`

TestLevel — Test confidence level
`0.95` (default) | numeric value between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric value between `0` and `1`.

Data Types: `double`

ShowDetails — Flag to display all details in output

false (default) | scalar logical with a value of true or false

Flag to display all details in output including the columns for critical-value method, number of lags tested, and test confidence level, specified as the comma-separated pair consisting of 'ShowDetails' and a scalar logical value.

Data Types: logical

Output Arguments

TestResults — Results

table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR levels
- 'VaRLevel' — VaR level
- 'ConditionalDE' — Categorical array with the categories'accept' and 'reject', which indicate the result of the conditionalDE test
- 'UnconditionalDE'— Categorical array with the categories'accept' and 'reject', which indicate the result of the unconditionalDE test

Note For the test results, the terms 'accept' and 'reject' are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

If you set the ShowDetails optional name-value argument to true, the TestResults table also includes 'CriticalValueMethod', 'NumLags', and 'TestLevel' columns.

Version History

Introduced in R2019b

References

[1] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science.* Vol. 63, Issue 4, April 2017.

[2] Basel Committee on Banking Supervision. "*Minimum Capital Requirements for Market Risk*". January 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

esbacktestbyde | summary | unconditionalDE | conditionalDE | simulate | esbacktestbysim

Topics

“Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63

"Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano" on page 2-72
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information" on page 2-30
"Overview of Expected Shortfall Backtesting" on page 2-20
"ES Backtest Using Du-Escanciano Method" on page 2-24
"Comparison of ES Backtesting Methods" on page 2-26

score

Compute credit scores for given dataset for a `compactCreditScorecard` object

Syntax

`[Scores,Points] = score(csc,data)`

Description

`[Scores,Points] = score(csc,data)` computes the credit scores and points for the `compactCreditScorecard` object (`csc`) based on the `data`. Missing data translates into `NaN` values for the corresponding points.

Examples

Obtain a Score for a compactCreditScorecard Object with New Data

To create a `compactCreditScorecard` object, first create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData.mat
sc = creditscorecard(data)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalan
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
                     Data: [1200x11 table]
```

Before creating a `compactCreditScorecard` object, you must use `autobinning` and `fitmodel` with the `creditscorecard` object.

```
sc = autobinning(sc);
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769
```

```
Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Use the `creditscorecard` object with `compactCreditScorecard` to create a `compactCreditScorecard` object.

```
csc = compactCreditScorecard(sc)
```

```
csc = 
  compactCreditScorecard with properties:

              Description: ''
                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
        NumericPredictors: {'CustAge'  'CustIncome'  'TmWBank'  'AMBalance'}
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
            PredictorVars: {'CustAge'  'ResStatus'  'EmpStatus'  'CustIncome'  'TmWBank'  'Other
```

Then use `score` with the `compactCreditScorecard` object. For the purpose of illustration, suppose that a few rows from the original data are our "new" data. Use the `data` input argument in the `score` function to obtain the scores for the `newdata`.

```
newdata = data(10:20,:);
[Scores,Points] = score(csc,newdata)
```

```
Scores = 11×1

    0.8252
    0.6553
    1.2443
    0.9478
    0.5690
    1.6192
    0.4899
    0.3824
    0.2945
```

```
    1.4401
      ⋮

Points=11×7 table
     CustAge      ResStatus    EmpStatus     CustIncome     TmWBank      OtherCC     AMBalance
    _________    _________    _________    __________    _________    ________    _________

      0.23039      0.12696    -0.076317      0.43693     -0.033752     0.15842    -0.017472
      0.23039    -0.031252    -0.076317     0.052329     -0.033752     0.15842      0.35551
      0.23039      0.37641    -0.076317      0.24473     -0.044811     0.15842      0.35551
        0.479      0.12696    -0.076317      0.43693      -0.18257    -0.19168      0.35551
     0.046408      0.37641    -0.076317     0.092433     -0.033752    -0.19168      0.35551
      0.21445      0.37641      0.31449      0.24473     -0.044811     0.15842      0.35551
     -0.14036      0.12696      0.31449     0.081611     -0.033752     0.15842    -0.017472
    -0.060323    -0.031252      0.31449     0.052329     -0.033752     0.15842    -0.017472
     -0.15894      0.12696      0.31449     -0.45716     -0.044811     0.15842      0.35551
      0.23039      0.12696      0.31449      0.43693      -0.18257     0.15842      0.35551
      0.23039      0.37641    -0.076317      0.24473     -0.044811     0.15842    -0.064636
```

Input Arguments

`csc` — Compact credit scorecard model
`compactCreditScorecard` object

Compact credit scorecard model, specified as a `compactCreditScorecard` object.

To create a `compactCreditScorecard` object, use `compactCreditScorecard` or `compact` from Financial Toolbox.

`data` — Dataset to be scored
table

Dataset to be scored, specified as a MATLAB table where each row corresponds to individual observations. The `data` must contain columns for each of the predictors in the `compactCreditScorecard` object.

Output Arguments

`Scores` — Scores for each observation
vector

Scores for each observation, returned as a vector.

`Points` — Points per predictor for each observation
table

Points per predictor for each observation, returned as a table.

Algorithms

The score of an individual *i* is given by the formula

```
Score(i) = Shift + Slope*(b0 + b1*WOE1(i) + b2*WOE2(i)+ ... +bp*WOEp(i))
```

where *bj* is the coefficient of the *j*-th variable in the model, and WOE*j*(*i*) is the Weight of Evidence (WOE) value for the *i*-th individual corresponding to the *j*-th model variable. `Shift` and `Slope` are scaling constants that can be controlled with `formatpoints`.

If the data for individual *i* is in the *i*-th row of a given dataset, to compute a score, the data(*i*,*j*) is binned using existing binning maps, and converted into a corresponding Weight of Evidence value `WOE`*j*(*i*). Using the model coefficients, the unscaled score is computed as

```
s = b0 + b1*WOE1(i) + ... +bp*WOEp(i).
```

For simplicity, assume in the description above that the *j*-th variable in the model is the *j*-th column in the data input, although, in general, the order of variables in a given dataset does not have to match the order of variables in the model, and the dataset could have additional variables that are not used in the model.

The formatting options can be controlled using `formatpoints`.

Version History

Introduced in R2019a

References

[1] Anderson, R. *The Credit Scoring Toolkit.* Oxford University Press, 2007.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

`compactCreditScorecard` | `probdefault` | `displaypoints` | `validatemodel`

Topics

“compactCreditScorecard Object Workflow” on page 3-57
“Case Study for Credit Scorecard Analysis”
“Credit Scorecard Modeling with Missing Values”
“Credit Scorecard Modeling Workflow”
“About Credit Scorecards”

screenpredictors

Screen credit scorecard predictors for predictive value

Syntax

```
metric_table = screenpredictors(data)
metric_table = screenpredictors( ___ ,Name,Value)
```

Description

`metric_table = screenpredictors(data)` returns the output variable, `metric_table`, a MATLAB table containing the calculated values for several measures of predictive power for each predictor variable in the `data`.

Use the `screenpredictors` function as a preprocessing step in the "Credit Scorecard Modeling Workflow" to reduce the number of predictor variables before you create the credit scorecard using the `creditscorecard` function from Financial Toolbox. In addition, you can use **Threshold Predictors** from Risk Management Toolboxto interactively set credit scorecard predictor thresholds using the output from `screenpredictors` before you create the credit scorecard using the `creditscorecard`.

`metric_table = screenpredictors( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax.

Examples

Screen Predictors for a `creditscorecard` Object

Reduce the number of predictor variables by screening predictors before you create a credit scorecard.

Use the `CreditCardData.mat` file to load the data (using a dataset from Refaat 2011).

```
load CreditCardData.mat
```

Define `'IDVar'` and `'ResponseVar'`.

```
idvar = 'CustID';
responsevar = 'status';
```

Use `screenpredictors` to calculate the predictor screening metrics. The function returns a table containing the metrics values. Each table row corresponds to a predictor from the input table data.

```
metric_table = screenpredictors(data,'IDVar', idvar,'ResponseVar', responsevar)
```

```
metric_table=9×7 table
                  InfoValue    AccuracyRatio     AUROC      Entropy      Gini       Chi2PValue
                  _________    _____________    _______    _______    _______    __________

    CustAge        0.18863        0.17095       0.58547    0.88729    0.42626    0.00074524
```

```
    TmWBank          0.15719         0.13612         0.56806    0.89167    0.42864      0.0054591
    CustIncome       0.15572         0.17758         0.58879      0.891    0.42731      0.0018428
    TmAtAddress     0.094574        0.010421         0.50521    0.90089    0.43377          0.182
    UtilRate        0.075086        0.035914         0.51796    0.90405    0.43575        0.45546
    AMBalance        0.07159        0.087142         0.54357    0.90446    0.43592        0.48528
    EmpStatus       0.048038         0.10886         0.55443    0.90814     0.4381     0.00037823
    OtherCC         0.014301        0.044459         0.52223    0.91347    0.44132       0.047616
    ResStatus      0.0097738         0.05039          0.5252    0.91422    0.44182        0.27875
```

```
metric_table = sortrows(metric_table,'AccuracyRatio','descend')
```

```
metric_table=9×7 table
                   InfoValue    AccuracyRatio     AUROC     Entropy     Gini       Chi2PValue    
                   _________    _____________    _______    _______    _______    __________    

    CustIncome       0.15572         0.17758     0.58879      0.891    0.42731      0.0018428
    CustAge          0.18863         0.17095     0.58547    0.88729    0.42626     0.00074524
    TmWBank          0.15719         0.13612     0.56806    0.89167    0.42864      0.0054591
    EmpStatus       0.048038         0.10886     0.55443    0.90814     0.4381     0.00037823
    AMBalance        0.07159        0.087142     0.54357    0.90446    0.43592        0.48528
    ResStatus      0.0097738         0.05039      0.5252    0.91422    0.44182        0.27875
    OtherCC         0.014301        0.044459     0.52223    0.91347    0.44132       0.047616
    UtilRate        0.075086        0.035914     0.51796    0.90405    0.43575        0.45546
    TmAtAddress     0.094574        0.010421     0.50521    0.90089    0.43377          0.182
```

Based on the `AccuracyRatio` metric, select the top predictors to use when you create the `creditscorecard` object.

```
varlist = metric_table.Row(metric_table.AccuracyRatio > 0.09)
```

```
varlist = 4x1 cell
    {'CustIncome'}
    {'CustAge'   }
    {'TmWBank'   }
    {'EmpStatus' }
```

Use `creditscorecard` to create a `createscorecard` object based on only the "screened" predictors.

```
sc = creditscorecard(data,'IDVar', idvar,'ResponseVar', responsevar, 'PredictorVars', varlist)
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'CustIncome'  'TmWBank'}
    CategoricalPredictors: {'EmpStatus'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'EmpStatus'  'CustIncome'  'TmWBank'}
                     Data: [1200x11 table]
```

Input Arguments

`data` — Data for `creditscorecard` object
table | tall table | tall timetable

Data for the `creditscorecard` object, specified as a MATLAB table, tall table, or tall timetable, where each column of data can be any one of the following data types:

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

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `metric_table = screenpredictors(data,'IDVar','CustAge','ResponseVar','status','PredictorVars',{'CustID','CustIncome'})`

`IDVar` — Name of identifier variable
`''` (default) | character vector

Name of identifier variable, specified as the comma-separated pair consisting of `'IDVar'` and a case-sensitive character vector. The `'IDVar'` data can be ordinal numbers or Social Security numbers. By specifying `'IDVar'`, you can omit the identifier variable from the predictor variables easily.

Data Types: `char`

`ResponseVar` — Response variable name for “Good” or “Bad” indicator
last column of the `data` input (default) | character vector

Response variable name for the “Good” or “Bad” indicator, specified as the comma-separated pair consisting of `'ResponseVar'` and a case-sensitive character vector. The response variable data must be binary.

If not specified, `'ResponseVar'` is set to the last column of the input `data` by default.

Data Types: `char`

`PredictorVars` — Names of predictor variables
set difference between `VarNames` and `{IDVar,ResponseVar}` (default) | cell array of character vectors | string array

Names of predictor variables, specified as the comma-separated pair consisting of `'PredictorVars'` and a case-sensitive cell array of character vectors or string array. By default, when you create a `creditscorecard` object, all variables are predictors except for `IDVar` and `ResponseVar`. Any name you specify using `'PredictorVars'` must differ from the `IDVar` and `ResponseVar` names.

Data Types: `cell` | `string`

`WeightsVar` — Name of weights variable
`''` (default) | character vector

Name of weights variable, specified as the comma-separated pair consisting of `'WeightsVar'` and a case-sensitive character vector to indicate which column name in the `data` table contains the row weights.

If you do not specify `'WeightsVar'` when you create a `creditscorecard` object, then the function uses the unit weights as the observation weights.

Data Types: `char`

`NumBins` — Number of (equal frequency) bins for numeric predictors
`20` (default) | scalar numeric

Number of (equal frequency) bins for numeric predictors, specified as the comma-separated pair consisting of `'NumBins'` and a scalar numeric.

Data Types: `double`

`FrequencyShift` — Indicates small shift in frequency tables that contain zero entries
`0.5` (default) | scalar numeric between `0` and `1`

Small shift in frequency tables that contain zero entries, specified as the comma-separated pair consisting of `'FrequencyShift'` and a scalar numeric with a value between `0` and `1`.

If the frequency table of a predictor contains any "pure" bins (containing all goods or all bads) after you bin the data using `autobinning`, then the function adds the `'FrequencyShift'` value to all bins in the table. To avoid any perturbation, set `'FrequencyShift'` to `0`.

Data Types: `double`

Output Arguments

`metric_table` — Calculated values for predictor screening metrics
table

Calculated values for the predictor screening metrics, returned as table. Each table row corresponds to a predictor from the input table data. The table columns contain calculated values for the following metrics:

- `'InfoValue'` — Information value. This metric measures the strength of a predictor in the fitting model by determining the deviation between the distributions of `"Goods"` and `"Bads"`.
- `'AccuracyRatio'` — Accuracy ratio.
- `'AUROC'` — Area under the ROC curve.
- `'Entropy'` — Entropy. This metric measures the level of unpredictability in the bins. You can use the entropy metric to validate a risk model.
- `'Gini'` — Gini. This metric measures the statistical dispersion or inequality within a sample of data.

- `'Chi2PValue'` — Chi-square *p*-value. This metric is computed from the chi-square metric and is a measure of the statistical difference and independence between groups.
- `'PercentMissing'` — Percentage of missing values in the predictor. This metric is expressed in decimal form.

Version History

Introduced in R2019a

Extended Capabilities

Tall Arrays

Calculate with arrays that have more rows than fit in memory.

This function supports input `data` that is specified as a tall column vector, a tall table, or a tall timetable. Note that the output for numeric predictors might be slightly different when using a tall array. Categorical predictors return the same results for tables and tall arrays. For more information, see `tall` and “Tall Arrays”.

See Also

`creditscorecard` | `modifybins` | `modifypredictor` | `bininfo` | **Threshold Predictors**

Topics

“Feature Screening with screenpredictors” on page 3-64

simulate

Simulate Du-Escanciano (DE) expected shortfall (ES) test statistics

Syntax

```
ebtde = simulate(ebtde)
ebtde = simulate( ___ ,Name,Value)
```

Description

`ebtde = simulate(ebtde)` performs a simulation of the Du-Escanciano (DE) [1] expected shortfall (ES) test statistics. `simulate` simulates scenarios and calculates the supported test statistics for each scenario. The function uses the simulated test statistics to estimate the significance of the ES backtests when the `CriticalValueMethod` name-value pair argument for `unconditionalDE` or `conditionalDE` is set to `'simulation'`.

`ebtde = simulate( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input argument in the previous syntax.

Examples

Create an esbacktestbyde Object and Run a Simulation

Create an `esbacktestbyde` object for a *t* model with 10 degrees of freedom. First, run a `conditionalDE` test based on 1000 scenarios and then use the `simulate` function to run a second simulation with 5000 scenarios.

```
load ESBacktestDistributionData.mat
  rng('default'); % For reproducibility
    % Constructor runs simulation with 1000 scenarios
  ebtde = esbacktestbyde(Returns,"t",...
       'DegreesOfFreedom',T10DoF,...
       'Location',T10Location,...
       'Scale',T10Scale,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
% Run conditionalDE tests
conditionalDE(ebtde,'CriticalValueMethod','simulation')
```

```
ans=3×13 table
    PortfolioID       VaRID         VaRLevel    ConditionalDE    PValue    TestStatistic    Crit
    ___________    _____________    ________    _____________    ______    _____________    ____

       "S&P"       "t(10) 95%"        0.95         reject        0.003        15.285           3
       "S&P"       "t(10) 97.5%"     0.975         reject        0.006        16.177           3
       "S&P"       "t(10) 99%"        0.99         reject        0.037        6.9975           4
```

The tests report 1000 scenarios, see the `Scenarios` column.

Run a second simulation with 5000 scenarios

```
ebtde = simulate(ebtde,'NumScenarios',5000);
conditionalDE(ebtde,'CriticalValueMethod','simulation')
```

```
ans=3×13 table
    PortfolioID        VaRID         VaRLevel    ConditionalDE    PValue    TestStatistic    Crit
    ___________    ______________    ________    _____________    ______    _____________    ____

       "S&P"       "t(10) 95%"         0.95         reject        0.0016       15.285           3
       "S&P"       "t(10) 97.5%"      0.975         reject        0.0046       16.177           3
       "S&P"       "t(10) 99%"         0.99         reject        0.0362       6.9975           3
```

The tests show 5000 scenarios and updated *p*-values and critical values.

Input Arguments

ebtde — esbacktestbyde object
object

esbacktestbyde object, which contains a copy of the data (the PortfolioData, VarData, ESData, and Distribution properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an esbacktestbyde object, see esbacktestbyde.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: ebtde = simulate(ebtde,'NumLags',10,'NumScenarios',1000000,'BlockSize',10000,'TestList','conditionalDE')

NumLags — Number of lags in the conditionalDE test statistic
5 (default) | positive integer

Number of lags in the conditionalDE test statistic, specified as the comma-separated pair consisting of 'NumLags' and a positive integer. The simulated test statistics are stored for all lags from 1 to NumLags, so that the conditionalDE test results are available for any number of lags between 1 and NumLags after running the simulate function.

Data Types: double

NumScenarios — Number of scenarios to simulate
1000 (default) | scalar positive integer

Number of scenarios to simulate, specified using the comma-separated pair consisting of 'NumScenarios' and a scalar positive integer.

Data Types: double

BlockSize — Number of scenarios to simulate in single simulation block
1000 (default) | scalar positive integer

Number of scenarios to simulate in a single simulation block, specified using the comma-separated pair consisting of 'BlockSize' and a scalar positive integer.

Data Types: double

TestList — Indicator for which test statistics to simulate
["conditionalDE","unconditionalDE"] (default) | character vector with a value of 'conditionalDE' or 'unconditionalDE' | string with a value of "conditionalDE" or "unconditionalDE"

Indicator for which test statistics to simulate, specified as the comma-separated pair consisting of 'TestList' and a cell array of character vectors or a string array with the value 'conditionalDE', 'unconditionalDE'.

Data Types: cell | string

Output Arguments

ebtde — Updated esbacktestbyde object
object

ebtde is returned as an updated esbacktestbyde object. After you run simulate, the updated esbacktestbyde object stores the simulated test statistics, which unconditionalDE uses to calculate *p*-values and generate test results.

For more information on the esbacktestbyde object, see esbacktestbyde.

More About

Simulation of Test Statistics

The simulation of test statistics requires simulating scenarios of returns, assuming the distribution of returns $X_t \sim P_t$ is correct (null hypothesis), and computing the corresponding tests statistics for each scenario.

More specifically, the following steps describe the simulation process. The description uses the conditional test statistic C_{ES} for concreteness, but the same steps apply to the unconditional test statistic U_{ES}.

1. Simulate M scenarios of returns as

   $$X^s = (X_1^s, \ldots, X_N^s),\ s = 1, \ldots, M\,.$$

2. Compute the corresponding test statistic as

   $$C_{ES}^s = C_{ES}(X_1^s, \ldots, X_N^s),\ s = 1, \ldots, M\,.$$

3. Define P_C as the empirical distribution of the simulated test statistic values as

   $$P_C = P[C_{ES} \le x] = \frac{1}{M} I(C_{ES}^s \le x),$$

   where $I(.)$ is the indicator function.

To compute the test statistic in step 2, the ranks or mapped returns $U_t = P_t(X_t)$ need to be computed (see the definition of the test statistics for `unconditionalDE` and `conditionalDE`). Assuming that the model distribution is correct, the ranks U_t are always uniformly distributed in the unit interval. Therefore, in practice, directly simulating ranks is more efficient than simulating returns and then transforming the returns into ranks.

The `simulate` function implements the simulation process more efficiently as follows:

1. Simulated M scenarios of returns as

   $$U^s = (U_1^s, \ldots, U_N^s), \ s = 1, \ldots, M,$$

   with $U_t^s \sim Uniform(0, 1)$.
2. Compute the corresponding test statistic C_{ES} using the simulated ranks U^s as

   $$C_{ES}^s = C_{ES}(U_1^s, \ldots, U_N^s), \ s = 1, \ldots, M.$$
3. Define P_C as the empirical distribution of the simulated test statistic values as

   $$P_C = P[C_{ES} \leq x] = \frac{1}{M} I(C_{ES}^s \leq x).$$

After you determine the empirical distribution of the test statistic P_C in step 3, the significance of the test follows the descriptions provided for `unconditionalDE` and `conditionalDE`. The same steps apply to the unconditional test statistic U_{ES} and its distribution function P_U.

Version History

Introduced in R2019b

References

[1] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science.* Vol. 63, Issue 4, April 2017.

[2] Basel Committee on Banking Supervision. "*Minimum Capital Requirements for Market Risk*". January 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

`esbacktestbyde` | `summary` | `runtests` | `unconditionalDE` | `conditionalDE` | `esbacktestbysim`

Topics

"Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano" on page 2-63
"Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano" on page 2-72
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"ES Backtest Using Du-Escanciano Method" on page 2-24
"Comparison of ES Backtesting Methods" on page 2-26

simulate

Simulate expected shortfall (ES) test statistics

Syntax

```
ebts = simulate(ebts)
ebts = simulate(ebts,Name,Value)
```

Description

`ebts = simulate(ebts)` performs a simulation of ES test statistics. The `simulate` function simulates portfolio outcomes according to the distribution assumptions indicated in the `esbacktestbysim` object, and calculates all the supported test statistics under each scenario. The simulated test statistics are used to estimate the significance of the ES backtests.

`ebts = simulate(ebts,Name,Value)` adds optional name-value pair arguments.

Examples

Simulate ES Test Statistics

Create an `esbacktestbysim` object and run a simulation of 1000 scenarios.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

The `unconditional` and `minBiasAbsolute` tests report 1000 scenarios (see the `Scenarios` column in the report).

```
unconditional(ebts)
```

```
ans=3×10 table
    PortfolioID        VaRID         VaRLevel     Unconditional     PValue     TestStatistic     Crit
    ___________    _____________    ________     _____________     ______     _____________     ____

       "S&P"       "t(10) 95%"        0.95          accept         0.093        -0.13342          -0
       "S&P"       "t(10) 97.5%"     0.975          reject         0.031        -0.25011          -(
       "S&P"       "t(10) 99%"        0.99          reject         0.008        -0.57396          -0
```

```
minBiasAbsolute(ebts)
```

```
ans=3×10 table
    PortfolioID        VaRID         VaRLevel     MinBiasAbsolute     PValue     TestStatistic     Cr
```

```
    ____________    ______________    ________    ________________    ______    ______________    ___

        "S&P"       "t(10) 95%"         0.95          accept          0.062     -0.0014247        -(
        "S&P"       "t(10) 97.5%"      0.975          reject          0.029     -0.0026674        -(
        "S&P"       "t(10) 99%"         0.99          reject          0.005     -0.0060982        -(
```

Run a second simulation with `5000` scenarios using the `simulate` function. Rerun the `unconditional` and `minBiasAbsolute` tests using the updated `esbacktestbysim` object. Notice that the tests now show 5,000 scenarios along with updated *p*-values and critical values.

```
ebts = simulate(ebts,'BlockSize',10000,'NumScenarios',5000,'TestList',["conditional","unconditio
unconditional(ebts)
```

```
ans=3×10 table
    PortfolioID        VaRID         VaRLevel    Unconditional    PValue    TestStatistic    Crit
    ____________    ______________    ________    ______________    ______    ______________    ____

        "S&P"       "t(10) 95%"         0.95         accept         0.0952      -0.13342         -0
        "S&P"       "t(10) 97.5%"      0.975         reject         0.0456      -0.25011         -0
        "S&P"       "t(10) 99%"         0.99         reject          0.009      -0.57396         -0
```

```
minBiasAbsolute(ebts,"TestLevel",0.99)
```

```
ans=3×10 table
    PortfolioID        VaRID         VaRLevel    MinBiasAbsolute    PValue    TestStatistic    Cri
    ____________    ______________    ________    ________________    ______    ______________    ___

        "S&P"       "t(10) 95%"         0.95          accept          0.0622    -0.0014247        -(
        "S&P"       "t(10) 97.5%"      0.975          accept           0.026    -0.0026674        -(
        "S&P"       "t(10) 99%"         0.99          reject           0.006    -0.0060982        -(
```

Input Arguments

ebts — esbacktestbysim object

object

`esbacktestbysim` (`ebts`) object, which contains a copy of the given data (the `PortfolioData`, `VarData`, `ESData`, and `Distribution` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktestbysim` object, see `esbacktestbysim`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ebts = simulate(ebts,'NumScenarios',1000000,'BlockSize',10000,'TestList','conditional')`

NumScenarios — Number of scenarios to simulate
1000 (default) | positive integer

Number of scenarios to simulate, specified using the comma-separated pair consisting of 'NumScenarios' and a positive integer.

Data Types: double

BlockSize — Number of scenarios to simulate in single simulation block
1000 (default) | positive integer

Number of scenarios to simulate in a single simulation block, specified using the comma-separated pair consisting of 'BlockSize' and a positive integer.

Data Types: double

TestList — Indicator for which test statistics to simulate
["conditional","unconditional","quantile","minBiasAbsolute","minBiasRelative"] (default) | character vector with value of 'conditional', 'unconditional', 'quantile', 'minBiasAbsolute', or 'minBiasRelative' | string with value of "conditional", "unconditional", "quantile", "minBiasAbsolute", or "minBiasRelative"

Indicator for which test statistics to simulate, specified as the comma-separated pair consisting of 'TestList' and a cell array of character vectors or a string array with the value conditional, unconditional, quantile, minBiasAbsolute or minBiasRelative.

Data Types: char | cell | string

Output Arguments

ebts — Updated esbacktestbysim object
object

esbacktestbysim (ebts), returned as an updated object. After running simulate, the updated esbacktestbysim object stores the simulated test statistics, which are used to calculate *p*-values and generate test results.

For more information on an esbacktestbysim object, see esbacktestbysim.

More About

Simulation of Test Statistics and Significance of the Tests

The VaR and ES models assume that for each period *t*, the portfolio outcomes X_t have a cumulative probability distribution P_t.

Under the assumption that the distributions P_t are correct (the null hypothesis), test statistics are simulated by:

- Simulating M scenarios of N observations each, for example, $X^s = (X_1^s, \ldots, X_t^s, \ldots, X_N^s)$, with $X_t^s \sim P_t$, t = 1,…,N, and s = 1,…,M.
- For each simulated scenario X^s, compute the test statistic of interest $Z^s = Z(X^s)$, s = 1,…,M.
- The resulting M simulated test statistic values $Z^1,\ldots,Z^M$ from a distribution of the test statistic assuming the probability distributions P_t are correct.

The *p*-value is defined as the proportion of scenarios for which the simulated test statistic is smaller than the test statistic evaluated at the observed portfolio outcomes: $Z^{obs} = Z(X_1, \ldots X_N)$:

$$P_{value} = \frac{1}{M} \sum_{s=1}^{M} I(Z^s \leq Z^{obs})$$

where $I(Z^s \leq Z^{obs})$ is an indicator function with a value of `1` if $Z^s \leq Z^{obs}$, and `0` otherwise. If P_{test} is `1` minus the test confidence level, the test result is to `'reject'` if $P_{value} < P_{test}$.

The critical value is defined as the minimum simulated test statistic Z^{crit} with a *p*-value greater than or equal to P_{test}.

Version History

Introduced in R2017b

References

[1] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall.* MSCI Inc. December, 2014.

See Also

`summary` | `runtests` | `conditional` | `unconditional` | `quantile` | `minBiasRelative` | `minBiasAbsolute` | `esbacktestbysim` | `esbacktestbyde`

Topics

"Expected Shortfall (ES) Backtesting Workflow Using Simulation" on page 2-34
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

summary

Report on varbacktest data

Syntax

```
S = summary(vbt)
```

Description

`S = summary(vbt)` returns a basic report on the given `varbacktest` data, including the number of observations, the number of failures, the observed confidence level, and so on (see `S` for details).

Examples

Generate a Summary Report

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the summary report.

```
S = summary(vbt)
```

```
S=1×10 table
    PortfolioID    VaRID    VaRLevel    ObservedLevel    Observations    Failures    Expected
    ___________    _____    ________    _____________    ____________    ________    ________

    "Portfolio"    "VaR"      0.95         0.94535           1043           57         52.15 
```

Run a Summary Report for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object and generate a summary report.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
```

```
        [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
        'PortfolioID','Equity',...
        'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
        'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99]);
S = summary(vbt)
```

```
S=6×10 table
    PortfolioID        VaRID         VaRLevel     ObservedLevel     Observations     Failures     Exp
    ___________    ______________    ________    _____________    ____________    ________    ___

     "Equity"      "Normal95"          0.95         0.94535           1043            57         52
     "Equity"      "Normal99"          0.99          0.9837           1043            17         10
     "Equity"      "Historical95"      0.95         0.94343           1043            59         52
     "Equity"      "Historical99"      0.99         0.98849           1043            12         10
     "Equity"      "EWMA95"            0.95         0.94343           1043            59         52
     "Equity"      "EWMA99"            0.99         0.97891           1043            22         10
```

Input Arguments

vbt — varbacktest object

object

varbacktest (vbt) object, contains a copy of the given data (the PortfolioData and VarData properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a varbacktest object, see varbacktest.

Output Arguments

S — Summary report

table

Summary report, returned as a table. The table rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR data columns provided
- 'VaRLevel' — VaR level for the corresponding VaR data column
- 'ObservedLevel' — Observed confidence level, defined as number of periods without failures divided by number of observations
- 'Observations' — Number of observations, where missing values are removed from the data
- 'Failures' — Number of failures, where a failure occurs whenever the loss (negative of portfolio data) exceeds the VaR
- 'Expected' — Expected number of failures, defined as the number of observations multiplied by one minus the VaR level
- 'Ratio' — Ratio of the number of failures to expected number of failures
- 'FirstFailure' — Number of periods until first failure
- 'Missing' — Number of periods with missing values removed from the sample

Version History

Introduced in R2016b

See Also

`varbacktest` | `tl` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `runtests`

Topics

“VaR Backtesting Workflow” on page 2-6
“Value-at-Risk Estimation and Backtesting” on page 2-10
“Overview of VaR Backtesting” on page 2-2
“Comparison of ES Backtesting Methods” on page 2-26

summary

Display summary report for Bornhuetter-Ferguson analysis

Syntax

```
unpaidClaimsEstimateTable = summary(bf)
```

Description

`unpaidClaimsEstimateTable = summary(bf)` displays a summary report of different claims estimates using the Bornhuetter-Ferguson technique. The report displays the latest diagonal of both reported and paid development triangles, projected ultimate claims, cases outstanding, IBNR claims, and total unpaid claims estimates.

Examples

Generate Summary Report for bornhuetterFerguson Object

Generate a summary report for a `bornhuetterFerguson` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                  Origin: {10x1 cell}
             Development: {10x1 cell}
                  Claims: [10x10 double]
          LatestDiagonal: [10x1 double]
             Description: ""
              TailFactor: 1
```

```
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

         ReportedTriangle: [1x1 developmentTriangle]
             PaidTriangle: [1x1 developmentTriangle]
            EarnedPremium: [10x1 double]
            InitialClaims: [10x1 double]
          CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Create a `bornhuetterFerguson` object with reported claims, paid claims, and expected claims to calculate ultimate claims, cases outstanding, IBNR claims, and unpaid claims estimates.

```
bf = bornhuetterFerguson(dT_reported, dT_paid, ec.ultimateClaims)
```

```
bf = 
  bornhuetterFerguson with properties:

     ReportedTriangle: [1x1 developmentTriangle]
         PaidTriangle: [1x1 developmentTriangle]
       ExpectedClaims: [10x1 double]
    PercentUnreported: [10x1 double]
        PercentUnpaid: [10x1 double]
      CaseOutstanding: [10x1 double]
```

Use `summary` to display the latest diagonal of both reported and paid development triangles, projected ultimate claims, cases outstanding, IBNR claims, and total unpaid claims estimates for a `bornhuetterFerguson` object.

```
unpaidClaimsEstimateTable = summary(bf)
```

```
unpaidClaimsEstimateTable=11×9 table
            Reported Claims    Paid Claims    Projected Ultimate Reported Claims    Projected Ul
            _______________    ___________    __________________________________    ____________

    2010        5089.4           4892.6                    5089.4
    2011        5179.9           5134.4                    5185.1
    2012        5625.4           5512.3                    5642.1
    2013        5803.7           5728.9                    5838.4
    2014        5878.7           5759.1                    5935.8
    2015        5772.8           5763.6                    5861.7
    2016        5714.3           5472.4                    5863.9
    2017        5854.4           5171.2                      6155
    2018        5495.1           4386.1                    6106.9
    2019        4945.9           2764.8                    6496.8
    Total        55360            50585                     58175
```

Input Arguments

bf — Bornhuetter-Ferguson
`bornhuetterFerguson` object

Bornhuetter-Ferguson object, specified as a previously created `bornhuetterFerguson` object.

Data Types: `object`

Output Arguments

unpaidClaimsEstimateTable — Report of claims estimates obtained using the Bornhuetter-Ferguson technique
table

Report of claims estimates obtained using the Bornhuetter-Ferguson technique, returned as a table.

Version History

Introduced in R2020b

See Also

`ultimateClaims` | `ibnr` | `unpaidClaims`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

summary

Display summary report for Cape Cod analysis

Syntax

```
unpaidClaimsEstimateTable = summary(cc)
```

Description

`unpaidClaimsEstimateTable = summary(cc)` displays a summary report of different claims estimates using the Cape Cod technique. The report displays the latest diagonal of both reported and paid development triangles, projected ultimate claims, cases outstanding, IBNR claims, and total unpaid claims estimates.

Examples

Generate Summary Report for capeCod Object

This example shows how to generate a summary report for a `capeCod` object for simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                      Origin: {10x1 cell}
                 Development: {10x1 cell}
                      Claims: [10x10 double]
              LatestDiagonal: [10x1 double]
                 Description: ""
                  TailFactor: 1
```

```
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create a `capeCod` object where the first input argument is the reported development triangle, the second input argument is the paid development triangle, and the third input is the earned premium.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
cc = capeCod(dT_reported, dT_paid,earnedPremium)
```

```
cc = 
  capeCod with properties:

           ReportedTriangle: [1x1 developmentTriangle]
               PaidTriangle: [1x1 developmentTriangle]
              EarnedPremium: [10x1 double]
              UsedUpPremium: [10x1 double]
       EstimatedClaimRatios: [10x1 double]
         ExpectedClaimRatio: 0.4258
    EstimatedExpectedClaims: [10x1 double]
         PercentUnreported: [10x1 double]
            CaseOutstanding: [10x1 double]
```

Use `summary` to generate a summary report for the different claims estimates.

```
unpaidClaimsEstimateTable = summary(cc)
```

unpaidClaimsEstimateTable=*11×6 table*

	Reported Claims	Paid Claims	Ultimate Claims	Case Outstanding	IBNR
2010	5089.4	4892.6	5089.4	196.79	0
2011	5179.9	5134.4	5187.6	45.46	7.665
2012	5625.4	5512.3	5638.2	113.15	12.745
2013	5803.7	5728.9	5852	74.83	48.338
2014	5878.7	5759.1	5944.7	119.58	66.006
2015	5772.8	5763.6	5836.7	9.2	63.901
2016	5714.3	5472.4	5833.2	241.88	118.98
2017	5854.4	5171.2	6063.2	683.23	208.81
2018	5495.1	4386.1	6089.3	1109	594.21
2019	4945.9	2764.8	5945.9	2181.1	999.98
Total	55360	50585	57480	4774.2	2120.6

Input Arguments

`cc` — Cape Cod object
`capeCod` object

Cape Cod object, specified as a previously created `capeCod` object.

Data Types: `object`

Output Arguments

`unpaidClaimsEstimateTable` — Report of claims estimates obtained using the Cape Cod technique
table

Report of claims estimates obtained using the Cape Cod technique, returned as a table.

Version History

Introduced in R2021a

See Also

`ibnr` | `unpaidClaims` | `ultimateClaims`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

summary

Display summary report for different claims estimates

Syntax

`unpaidClaimsEstimateTable = summary(cl)`

Description

`unpaidClaimsEstimateTable = summary(cl)` displays the latest diagonal of both reported and paid development triangles, projected ultimate claims, case outstanding, IBNR claims, and the total unpaid claims estimates.

Examples

Generate Summary Report for Different Claims Estimates Using chainLadder

Generate the summary report for a `chainLadder` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create a `chainLadder` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
cl = chainLadder(dT_reported, dT_paid)
```

```
cl = 
  chainLadder with properties:

    ReportedTriangle: [1x1 developmentTriangle]
        PaidTriangle: [1x1 developmentTriangle]
     CaseOutstanding: [10x1 double]
```

Use `ibnr` to compute the incurred-but-not-reported (IBNR).

```
ibnrClaims = ibnr(cl,'reported')
```

```
ibnrClaims = 10×1
10³ ×

         0
    0.0052
    0.0169
    0.0349
    0.0575
    0.0880
    0.1489
    0.3019
    0.6084
    1.5181
```

Use `unpaidClaims` to compute the unpaid claims.

```
unpaidClaimsEstimate = unpaidClaims(cl,'reported')
```

```
unpaidClaimsEstimate = 10×1
10³ ×

    0.1968
    0.0506
    0.1300
    0.1097
    0.1771
```

```
    0.0972
    0.3908
    0.9851
    1.7175
    3.6992
```

Use `summary` to display the latest diagonal of both reported and paid development triangles, projected ultimate claims, cases outstanding, IBNR claims, and total unpaid claims estimates.

```
unpaidClaimsEstimateTable = summary(cl)
```

```
unpaidClaimsEstimateTable=11×9 table
             Reported Claims    Paid Claims    Projected Ultimate Reported Claims    Projected Ul
             _______________    ___________    __________________________________    ____________

    2010         5089.4           4892.6                     5089.4
    2011         5179.9           5134.4                     5185.1
    2012         5625.4           5512.3                     5642.3
    2013         5803.7           5728.9                     5838.6
    2014         5878.7           5759.1                     5936.2
    2015         5772.8           5763.6                     5860.8
    2016         5714.3           5472.4                     5863.2
    2017         5854.4           5171.2                     6156.4
    2018         5495.1           4386.1                     6103.5
    2019         4945.9           2764.8                       6464
    Total         55360            50585                      58139
```

Input Arguments

`cl` — Chain ladder
`chainLadder` object

Chain ladder, specified as a previously created `chainLadder` object.

Data Types: `object`

Output Arguments

`unpaidClaimsEstimateTable` — Different claims estimates using chain ladder technique
table

Different claims estimates obtained using the chain ladder technique, returned as a table.

Version History

Introduced in R2020b

See Also

`ibnr` | `unpaidClaims`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

summary

Basic expected shortfall (ES) report on failures and severity

Syntax

```
S = summary(ebt)
```

Description

`S = summary(ebt)` returns a basic report on the given `esbacktest` data, including the number of observations, number of failures, observed confidence level, and so on (see `S` for details).

Examples

Generate an ES Summary Report

Create an `esbacktest` object.

```
load ESBacktestData
ebt = esbacktest(Returns,VaRModel1,ESModel1,'VaRLevel',VaRLevel)
```

```
ebt = 
  esbacktest with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x1 double]
           ESData: [1966x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9750
```

Generate the ES summary report.

```
S = summary(ebt)
```

```
S=1×11 table
    PortfolioID    VaRID    VaRLevel    ObservedLevel    ExpectedSeverity    ObservedSeverity
    ___________    _____    ________    _____________    ________________    ________________

    "Portfolio"    "VaR"     0.975         0.97101            1.1928              1.4221
```

Input Arguments

`ebt` — `esbacktest` object
object

esbacktest (ebt) object, contains a copy of the given data (the PortfolioData, VarData, and ESData properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an esbacktest object, see esbacktest.

Output Arguments

S — Summary report
table

Summary report, returned as a table. The table rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR data columns provided
- 'VaRLevel' — VaR level for the corresponding VaR data column
- 'ObservedLevel' — Observed confidence level, defined as the number of periods without failures divided by number of observations
- 'ExpectedSeverity' — Expected average severity ratio, that is, the average ratio of ES to VaR over the periods with VaR failures
- 'ObservedSeverity' — Observed average severity ratio, that is, the average ratio of loss to VaR over the periods with VaR failures
- 'Observations' — Number of observations, where missing values are removed from the data
- 'Failures' — Number of failures, where a failure occurs whenever the loss (negative of portfolio data) exceeds the VaR
- 'Expected' — Expected number of failures, defined as the number of observations multiplied by 1 minus the VaR level
- 'Ratio' — Ratio of number of failures to expected number of failures
- 'Missing' — Number of periods with missing values removed from the sample

Note The 'ExpectedSeverity' and 'ObservedSeverity' ratios are undefined (NaN) when there are no VaR failures in the data.

Version History

Introduced in R2017b

See Also

esbacktest | runtests | unconditionalNormal | unconditionalT

Topics

"Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information" on page 2-30
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

summary

Basic expected shortfall (ES) report on failures and severity

Syntax

```
S = summary(ebts)
```

Description

`S = summary(ebts)` returns a basic report on the given `esbacktestbysim` data, including the number of observations, number of failures, observed confidence level, and so on (see `S` for details).

Examples

Generate an ES Summary Report

Create an `esbacktestbysim` object.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

Generate the ES summary report.

```
S = summary(ebts)
```

```
S=3×11 table
    PortfolioID        VaRID         VaRLevel    ObservedLevel    ExpectedSeverity    ObservedSeve
    ___________    _____________    ________    _____________    ________________    ____________

       "S&P"       "t(10) 95%"        0.95         0.94812           1.3288               1.4515
       "S&P"       "t(10) 97.5%"     0.975         0.97202           1.2652               1.4134
       "S&P"       "t(10) 99%"        0.99         0.98627           1.2169               1.3947
```

Input Arguments

ebts — `esbacktestbysim` object
object

`esbacktestbysim` (`ebts`) object, which contains a copy of the given data (the `PortfolioData`, `VarData`, `ESData`, and `Distribution` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktestbysim` object, see `esbacktestbysim`.

Output Arguments

S — Summary report

table

Summary report, returned as a table. The table rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'ObservedLevel'` — Observed confidence level, defined as the number of periods without failures divided by number of observations
- `'ExpectedSeverity'` — Expected average severity ratio, that is, the average ratio of ES to VaR over the periods with VaR failures
- `'ObservedSeverity'` — Observed average severity ratio, that is, the average ratio of loss to VaR over the periods with VaR failures
- `'Observations'` — Number of observations, where missing values are removed from the data
- `'Failures'` — Number of failures, where a failure occurs whenever the loss (negative of portfolio data) exceeds the VaR
- `'Expected'` — Expected number of failures, defined as the number of observations multiplied by 1 minus the VaR level
- `'Ratio'` — Ratio of number of failures to expected number of failures
- `'Missing'` — Number of periods with missing values removed from the sample

Note The `'ExpectedSeverity'` and `'ObservedSeverity'` ratios are undefined (`NaN`) when there are no VaR failures in the data.

Version History

Introduced in R2017b

See Also

`runtests` | `conditional` | `unconditional` | `quantile` | `simulate` | `minBiasRelative` | `minBiasAbsolute` | `esbacktestbysim` | `esbacktestbyde`

Topics

“Expected Shortfall (ES) Backtesting Workflow Using Simulation” on page 2-34
“Expected Shortfall Estimation and Backtesting” on page 2-44
“Overview of Expected Shortfall Backtesting” on page 2-20
“Comparison of ES Backtesting Methods” on page 2-26

summary

Basic expected shortfall (ES) report on failures and severity

Syntax

`S = summary(ebtde)`

Description

`S = summary(ebtde)` returns a basic report on the given `esbacktestbyde` data. The report includes the number of observations, number of failures, observed confidence level, and so on. See `S` for details.

Unlike other ES backtesting classes, the `esbacktestbyde` object does not require VaR data or ES data inputs. `esbacktestbyde` internally computes VaR and ES data based on distribution information to determine the severity information reported by the `summary` function.

Examples

Create an `esbacktestbyde` Object and Run ES Backtest Summary Report

Create an `esbacktestbyde` object for a *t* model with 10 degrees of freedom, and then run a basic ES backtest summary report.

```
load ESBacktestDistributionData.mat
    rng('default'); % For reproducibility
    ebtde = esbacktestbyde(Returns,"t",...
       'DegreesOfFreedom',T10DoF,...
       'Location',T10Location,...
       'Scale',T10Scale,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
    summary(ebtde)
```

```
ans=3×11 table
    PortfolioID       VaRID        VaRLevel     ObservedLevel     ExpectedSeverity     ObservedSeve
    ___________    _____________    ________    _____________    ________________    ____________

       "S&P"       "t(10) 95%"        0.95         0.94812            1.3288              1.4515
       "S&P"       "t(10) 97.5%"     0.975         0.97202            1.2652              1.4134
       "S&P"       "t(10) 99%"        0.99         0.98627            1.2169              1.3947
```

Input Arguments

ebtde — esbacktestbyde object
object

esbacktestbyde object contains a copy of the data (the PortfolioData, VaRData, and ESData properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested.

Note Unlike other ES backtesting classes, esbacktestbyde does not require VaR data or ES data inputs. esbacktestbyde internally computes VaR and ES data based on distribution information to determine the severity information reported by summary. For more information on creating an esbacktestbyde object, see esbacktestbyde.

Output Arguments

S — Summary report

table

Summary report, returned as a table. The table rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR levels
- 'VaRLevel' — VaR level
- 'ObservedLevel' — Observed confidence level, defined as the number of periods without failures divided by number of observations
- 'ExpectedSeverity' — Expected average severity ratio, that is, the average ratio of ES to VaR over the periods with VaR failures
- 'ObservedSeverity' — Observed average severity ratio, that is, the average ratio of loss to VaR over the periods with VaR failures
- 'Observations' — Number of observations, where missing values are removed from the data
- 'Failures' — Number of failures, where a failure occurs whenever the loss (negative of portfolio data) exceeds the VaR
- 'Expected' — Expected number of failures, defined as the number of observations multiplied by 1 minus the VaR level
- 'Ratio' — Ratio of number of failures to expected number of failures
- 'Missing' — Number of periods with missing values removed from the sample

Note The 'ExpectedSeverity' and 'ObservedSeverity' ratios are undefined (NaN) when there are no VaR failures in the data.

Version History

Introduced in R2019b

References

[1] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science.* Vol. 63, Issue 4, April 2017.

[2] Basel Committee on Banking Supervision. "*Minimum Capital Requirements for Market Risk*". January 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

`esbacktestbyde` | `runtests` | `unconditionalDE` | `conditionalDE` | `simulate` | `esbacktestbysim`

Topics

“Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63
“Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-72
“Expected Shortfall Estimation and Backtesting” on page 2-44
“Overview of Expected Shortfall Backtesting” on page 2-20
“ES Backtest Using Du-Escanciano Method” on page 2-24
“Comparison of ES Backtesting Methods” on page 2-26

summary

Display summary report for different claims estimates

Syntax

`unpaidClaimsEstimateTable = summary(ec)`

Description

`unpaidClaimsEstimateTable = summary(ec)` displays the summary report for the latest diagonal of both reported and paid development triangles, projected ultimate claims, cases outstanding, IBNR claims, and total unpaid claims estimates.

Examples

Generate Summary Report for `expectedClaims` Object

Generate the summary report for different claims estimates for an `expectedClaims` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.001
```

```
              SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                         Origin: {10x1 cell}
                    Development: {10x1 cell}
                         Claims: [10x10 double]
                 LatestDiagonal: [10x1 double]
                    Description: ""
                     TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
              SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

          ReportedTriangle: [1x1 developmentTriangle]
              PaidTriangle: [1x1 developmentTriangle]
             EarnedPremium: [10x1 double]
             InitialClaims: [10x1 double]
           CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Use `summary` to display the report for the latest diagonal of both reported and paid development triangles, projected ultimate claims, cases outstanding, IBNR claims, and total unpaid claims estimates.

```
unpaidClaimsEstimateTable = summary(ec)
```

unpaidClaimsEstimateTable=*11×6 table*

	Reported Claims	Paid Claims	Ultimate Claims	Case Outstanding	IBNR
2010	5089.4	4892.6	4991	196.79	-98.395
2011	5179.9	5134.4	5162.3	45.46	-17.574
2012	5625.4	5512.3	5585.6	113.15	-39.856
2013	5803.7	5728.9	5806.7	74.83	2.9912
2014	5878.7	5759.1	5899	119.58	20.351
2015	5772.8	5763.6	5921.1	9.2	148.29
2016	5714.3	5472.4	5889.5	241.88	175.26
2017	5854.4	5171.2	6128.9	683.23	274.42
2018	5495.1	4386.1	6137.4	1109	642.25
2019	4945.9	2764.8	6603.4	2181.1	1657.5
Total	55360	50585	58125	4774.2	2765.3

Input Arguments

ec — Expected claims
`expectedClaims` object

Expected claims, specified as a previously created `expectedClaims` object.

Data Types: `object`

Output Arguments

`unpaidClaimsEstimateTable` — Displays different claims estimates using the expected claims technique
table

Displays different claims estimates using the expected claims technique, returned as a table.

Version History

Introduced in R2020b

See Also

`ultimateClaims` | `ibnr` | `unpaidClaims`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

tbf

Time between failures mixed test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = tbf(vbt)
TestResults = tbf(vbt,Name,Value)
```

Description

`TestResults = tbf(vbt)` generates the time between failures mixed test (TBF) for value-at-risk (VaR) backtesting.

`TestResults = tbf(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Generate TBF Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `tbf` test results.

```
TestResults = tbf(vbt)
```

TestResults=*1×20 table*

PortfolioID	VaRID	VaRLevel	TBF	LRatioTBF	PValueTBF	POF	LRatioPOF
"Portfolio"	"VaR"	0.95	reject	88.952	0.0055565	accept	0.46147

Run the TBF Test for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt =
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `tbf` test results using the `TestLevel` optional input.

```
TestResults = tbf(vbt,'TestLevel',0.90)
```

TestResults=*6×20 table*

PortfolioID	VaRID	VaRLevel	TBF	LRatioTBF	PValueTBF	POF	LI
"Equity"	"Normal95"	0.95	reject	88.952	0.0055565	accept	(
"Equity"	"Normal99"	0.99	reject	26.441	0.090095	reject	
"Equity"	"Historical95"	0.95	reject	83.63	0.023609	accept	(
"Equity"	"Historical99"	0.99	accept	16.456	0.22539	accept	(
"Equity"	"EWMA95"	0.95	accept	72.545	0.12844	accept	(
"Equity"	"EWMA99"	0.99	reject	41.66	0.0099428	reject	

Input Arguments

`vbt` — `varbacktest` object

object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = tbf(vbt,'TestLevel',0.99)`

`TestLevel` — Test confidence level

`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — `tbf` test results
table

`tbf` test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'TBF'` — Categorical array with categories `accept` and `reject` that indicate the result of the `tbf` test
- `'LRatioTBF'` — Likelihood ratio of the `tbf` test
- `'PValueTBF'` — P-value of the `tbf` test
- `'POF'` — Categorical array with the categories `accept` and `reject` that indicate the result of the POF test
- `'LRatioPOF'` — Likelihood ratio of the `pof` test
- `'PValuePOF'` — P-value of the `pof` test
- `'TBFI'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `tbfi` test
- `'LRatioTBFI'` — Likelihood ratio of the `tbfi` test
- `'PValueTBFI'` — P-value of the `tbfi` test
- `'Observations'` — Number of observations
- `'Failures'` — Number of failures
- `'TBFMin'` — Minimum value of observed times between failures
- `'TBFQ1'` — First quartile of observed times between failures
- `'TBFQ2'` — Second quartile of observed times between failures
- `'TBFQ3'` — Third quartile of observed times between failures
- `'TBFMax'` — Maximum value of observed times between failures
- `'TestLevel'` — Test confidence level

Note For `tbf` test results, the terms `accept` and `reject` are used for convenience, technically a `tbf` test does not accept a model. Rather, the test fails to reject it.

More About

Time Between Failures (TBF) Mixed Test

The `tbf` function performs the time between failures mixed test, also known as the Haas mixed Kupiec test.

'Mixed' means that it combines a frequency and an independence test. The frequency test is Kupiec's proportion of failures (POF) test. The independence test is the time between failures independence

(TBFI) test. The TBF test is an extension of Kupiec's time until first failure (TUFF) test, proposed by Haas (2001), to take into account not only the time until the first failure, but also the time between all failures. The `tbf` function combines the `pof` test and the `tbfi` test.

Algorithms

The likelihood ratio (test statistic) of the TBF test is the sum of the likelihood ratios of the POF and TBFI tests

$$LRatioTBF = LRatioPOF + LRatioTBFI$$

which is asymptotically distributed as a chi-square distribution with x+1 degrees of freedom, where$x$ is the number of failures. See the Algorithms sections for `pof` and `tbfi` for the definitions of their likelihood ratios.

The *p*-value of the `tbf` test is the probability that a chi-square distribution with x+1 degrees of freedom exceeds the likelihood ratio *LRatioTBF*

$$PValueTBF = 1 - F(LRatioTBF)$$

where F is the cumulative distribution of a chi-square variable with x+1 degrees of freedom and x is the number of failures.

The result of the test is to accept if

$$F(LRatioTBF) < F(TestLevel)$$

and reject otherwise, where F is the cumulative distribution of a chi-square variable with x+1 degrees of freedom and x is the number of failures. If the likelihood ratio (*LRatioTBF*) is undefined, that is, with no failures yet, the TBF result is to accept only when both POF and TBFI tests accept.

Version History

Introduced in R2016b

References

[1] Haas, M. "*New Methods in Backtesting.*" Financial Engineering, Research Center Caesar, Bonn, 2001.

See Also

`varbacktest` | `tl` | `tuff` | `bin` | `pof` | `cc` | `cci` | `tbfi` | `summary` | `runtests`

Topics

"VaR Backtesting Workflow" on page 2-6
"Value-at-Risk Estimation and Backtesting" on page 2-10
"Overview of VaR Backtesting" on page 2-2
"Haas's Time Between Failures or Mixed Kupiec's Test" on page 2-4
"Comparison of ES Backtesting Methods" on page 2-26

tbfi

Time between failures independence test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = tbfi(vbt)
TestResults = tbfi(vbt,Name,Value)
```

Description

`TestResults = tbfi(vbt)` generates the time between failures independence (TBFI) test for value-at-risk (VaR) backtesting.

`TestResults = tbfi(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Generate TBFI Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt =
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `tbfi` test results.

```
TestResults = tbfi(vbt)
```

```
TestResults=1×14 table
    PortfolioID    VaRID    VaRLevel     TBFI     LRatioTBFI    PValueTBFI    Observations    Fa
    ___________    _____    ________    ______    __________    __________    ____________    __

    "Portfolio"    "VaR"      0.95      reject      88.491      0.0047475         1043
```

Run the TBFI Test for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `tbfi` test results using the `TestLevel` optional input.

```
TestResults = tbfi(vbt,'TestLevel',0.90)
```

TestResults=6×14 table

PortfolioID	VaRID	VaRLevel	TBFI	LRatioTBFI	PValueTBFI	Observatio
"Equity"	"Normal95"	0.95	reject	88.491	0.0047475	1043
"Equity"	"Normal99"	0.99	accept	22.929	0.15157	1043
"Equity"	"Historical95"	0.95	reject	82.719	0.022513	1043
"Equity"	"Historical99"	0.99	accept	16.228	0.18101	1043
"Equity"	"EWMA95"	0.95	accept	71.635	0.12517	1043
"Equity"	"EWMA99"	0.99	reject	31.83	0.080339	1043

Input Arguments

`vbt` — `varbacktest` object

object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = tbfi(vbt,'TestLevel',0.99)`

`TestLevel` — Test confidence level

`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — `tbfi` test results
table

`tbfi` test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'TBFI'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `tbfi` test
- `'LRatioTBFI'` — Likelihood ratio of the `tbfi` test
- `'PValueTBFI'` — P-value of the `tbfi` test
- `'Observations'` — Number of observations
- `'Failures'` — Number of failures
- `'TBFMin'` — Minimum value of observed times between failures
- `'TBFQ1'` — First quartile of observed times between failures
- `'TBFQ2'` — Second quartile of observed times between failures
- `'TBFQ3'` — Third quartile of observed times between failures
- `'TBFMax'` — Maximum value of observed times between failures
- `'TestLevel'` — Test confidence level

Note For `tbfi` test results, the terms `accept` and `reject` are used for convenience, technically a `tbfi` test does not accept a model. Rather, the test fails to reject it.

More About

Time Between Failures Independence (TBIF) Test

The `tbfi` function performs the time between failures independence test. This test is an extension of Kupiec's time until first failure (TUFF) test.

TBFI was proposed by Haas (2001) to test for independence. It takes into account not only the time until the first failure, but also the time between all failures. For the time between failures mixed test, see the `tbf` function.

Algorithms

The likelihood ratio (test statistic) of the TBFI test is the sum of TUFF likelihood ratios for each time between failures. If x is the number of failures, and n_1 is the number of periods until the first failure, n_2 the number of periods between the first and the second failure, and, in general, n_i is the number of periods between failure i - 1 and failure i, then a likelihood ratio $LRatioTBFI_i$ for each n_i is based on the TUFF formula

$$LRatioTBFI_i = LRatioTUFF(n_i) = -2\sum_{i=1}^{x} \log\left(\frac{pVaR(1 - pVaR)^{n_i - 1}}{\left(\frac{1}{n_i}\right)\left(1 - \frac{1}{n_i}\right)^{n_i - 1}}\right)$$

$$= -2(\log(pVaR) + (n_i - 1)\log(1 - pVaR) + n_i\log(n_i) - (n_i - 1)\log(n_i - 1))$$

As with the `tuff` test, $LRatioTBFI_i$ = `-2`$log(pVaR)$ if n_i = `1`.

The TBFI likelihood ratio *LRatioTBFI* is then the sum of the individual likelihood ratios for all times between failures

$$LRatioTBFI = \sum_{i=1}^{x} LRatioTBFI_i$$

which is asymptotically distributed as a chi-square distribution with x degrees of freedom, where x is the number of failures.

The *p*-value of the `tbfi` test is the probability that a chi-square distribution with x degrees of freedom exceeds the likelihood ratio *LRatioTBFI*

$$PValueTBFI = 1 - F(LRatioTBFI)$$

where F is the cumulative distribution of a chi-square variable with x degrees of freedom and x is the number of failures.

The result of the test is to accept if

$$F(LRatioTBFI) < F(TestLevel)$$

and reject otherwise, where F is the cumulative distribution of a chi-square variable with x degrees of freedom and x is the number of failures.

If there are no failures in the sample, the test statistic is not defined. This is handled the same as a TUFF test with no failures. For more information, see `tuff`.

Version History

Introduced in R2016b

References

[1] Haas, M. "*New Methods in Backtesting*." Financial Engineering, Research Center Caesar, Bonn, 2001.

See Also

`varbacktest` | `tl` | `tuff` | `bin` | `pof` | `cc` | `cci` | `tbf` | `summary` | `runtests`

Topics

"VaR Backtesting Workflow" on page 2-6
"Value-at-Risk Estimation and Backtesting" on page 2-10
"Overview of VaR Backtesting" on page 2-2
"Haas's Time Between Failures or Mixed Kupiec's Test" on page 2-4

“Comparison of ES Backtesting Methods” on page 2-26

tl

Traffic light test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = tl(vbt)
```

Description

`TestResults = tl(vbt)` generates the traffic light (TL) test for value-at-risk (VaR) backtesting.

Examples

Generate Traffic Light Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `tl` test results.

```
TestResults = tl(vbt)
```

```
TestResults=1×9 table
    PortfolioID    VaRID    VaRLevel     TL      Probability     TypeI     Increase    Observatio
    ___________    _____    ________    _____    ___________    _______    ________    __________

    "Portfolio"    "VaR"      0.95      green      0.77913      0.26396       0           1043
```

Run the TL Test for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
```

```
        'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
        'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `tl` test results.

```
TestResults = tl(vbt)
```

```
TestResults=6×9 table
    PortfolioID        VaRID          VaRLevel      TL      Probability      TypeI       Increase
    ___________    ______________    ________    ______    ___________    _________    ________

     "Equity"      "Normal95"          0.95      green       0.77913        0.26396           0
     "Equity"      "Normal99"          0.99      yellow      0.97991        0.03686     0.26582
     "Equity"      "Historical95"      0.95      green       0.85155        0.18232           0
     "Equity"      "Historical99"      0.99      green       0.74996        0.35269           0
     "Equity"      "EWMA95"            0.95      green       0.85155        0.18232           0
     "Equity"      "EWMA99"            0.99      yellow      0.99952      0.0011122     0.43511
```

Input Arguments

`vbt` — `varbacktest` object

object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Output Arguments

`TestResults` — `tl` test results

table

`tl` test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'TL'` — Categorical (ordinal) array with the categories `green`, `yellow`, and `red` that indicate the result of the traffic light `tl` test
- `'Probability'` — Cumulative probability of observing up to the corresponding number of failures

- `'TypeI'` — Probability of observing the corresponding number of failures or more if the model is correct
- `'Increase'` — Increase in the scaling factor
- `'Observations'` — Number of observations
- `'Failures'` — Number of failures

More About

Traffic Light Test

The `tl` function performs Basel's traffic light test, also known as three-zone test. Basel's methodology can be applied to any number of time periods and VaR confidence levels, as explained in “Algorithms” on page 6-425.

The Basel Committee reports, as an example, a table of the three zones for 250 time periods and a VaR confidence level of 0.99. The increase in scaling factor in the table reported by Basel has some ad-hoc adjustments (rounding, and so on) not explicitly described in the Basel document. The following table compares the increase in scaling factor reported in the Basel document for the case of 250 periods and 0.99% VaR confidence level, and the increase in the factors reported by the TL test.

Failures	Zone	Increase Basel	Increase TL
0	Green	0	0
1	Green	0	0
2	Green	0	0
3	Green	0	0
4	Green	0	0
5	Yellow	0.40	0.3982
6	Yellow	0.50	0.5295
7	Yellow	0.65	0.6520
8	Yellow	0.75	0.7680
9	Yellow	0.85	0.8791
10	Red	1	1

The `tl` function computes the scaling factor following the methodology described in the Basel document (see “References” on page 6-426) and is explained in the “Algorithms” on page 6-425 section. The `tl` function does not apply any ad-hoc adjustments.

Algorithms

The traffic light test is based on a binomial distribution. Suppose `N` is the number of observations, `p` = 1 - `VaRLevel` is the probability of observing a failure if the model is correct, and x is the number of failures.

The test computes the cumulative probability of observing up to x failures, reported in the `'Probability'` column,

$$Probability = Probability(X \leq x|N, p) = F(x|N, p)$$

where $F(x|N, p)$ is the cumulative distribution of a binomial variable with parameters N and p, with $p = 1 - VaRLevel$. The three zones are defined based on this cumulative probability:

- Green: $F(x|N, p) \leq$ `0.95`
- Yellow: `0.95` < $F(x|N, p) \leq$ `0.9999`
- Red: `0.9999` < $F(x|N, p)$

The probability of a Type-I error, reported in the `'TypeI'` column, is
$TypeI = TypeI(x|N, p) = 1 - F(X \geq x|N, p)$.

This probability corresponds to the probability of mistakenly rejecting the model if the model were correct. *Probability* and *TypeI* do not sum up to 1, they exceed 1 by exactly the probability of having x failures.

The increase in scaling factor, reported in the `'Increase'` column, is always `0` for the `green` zone and always `1` for the `red` zone. For the `yellow` zone, it is an adjustment based on the relative difference between the assumed VaR confidence level (*VaRLevel*) and the observed confidence level (x / N), where `N` is the number of observations and$x$ is the number of failures. To find the increase under the assumption of a normal distribution, compute the critical values *zAssumed* and *zObserved*.

The increase to the baseline scaling factor is given by

$$Increase = Baseline \times \left(\frac{zAssumed}{zObserved} - 1\right)$$

with the restriction that the increase cannot be negative or greater than `1`. The baseline scaling factor in the Basel rules is 3.

The `tl` function computes the scaling factor following this methodology, which is also described in the Basel document (see “References” on page 6-426). The `tl` function does not apply any ad-hoc adjustments.

Version History

Introduced in R2016b

References

[1] Basel Committee on Banking Supervision, *Supervisory Framework for the Use of 'Backtesting' in Conjunction with the Internal Models Approach to Market Risk Capital Requirements.* January, 1996, https://www.bis.org/publ/bcbs22.htm.

See Also

`varbacktest` | `bin` | `pof` | `tuff` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Topics

“VaR Backtesting Workflow” on page 2-6
“Value-at-Risk Estimation and Backtesting” on page 2-10
“Overview of VaR Backtesting” on page 2-2
“Traffic Light Test” on page 2-3
“Comparison of ES Backtesting Methods” on page 2-26

tuff

Time until first failure test for value-at-risk (VaR) backtesting

Syntax

```
TestResults = tuff(vbt)
TestResults = tuff(vbt,Name,Value)
```

Description

`TestResults = tuff(vbt)` generates the time until first failure (TUFF) test for value-at-risk (VaR) backtesting.

`TestResults = tuff(vbt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Generate TUFF Test Results

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

Generate the `tuff` test results.

```
TestResults = tuff(vbt)
```

```
TestResults=1×9 table
    PortfolioID    VaRID    VaRLevel     TUFF     LRatioTUFF    PValueTUFF    FirstFailure    Obs
    ___________    _____    ________    ______    __________    __________    ____________    ___

    "Portfolio"    "VaR"      0.95      accept      1.7354        0.18773          58
```

Run the TUFF Test for VaR Backtests for Multiple VaRs at Different Confidence Levels

Use the `varbacktest` constructor with name-value pair arguments to create a `varbacktest` object.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99])
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x6 double]
      PortfolioID: "Equity"
            VaRID: ["Normal95"    "Normal99"    "Historical95"    "Historical99"    "EWMA95"    "
         VaRLevel: [0.9500 0.9900 0.9500 0.9900 0.9500 0.9900]
```

Generate the `tuff` test results using the `TestLevel` optional input.

```
TestResults = tuff(vbt,'TestLevel',0.90)
```

```
TestResults=6×9 table
    PortfolioID        VaRID          VaRLevel      TUFF       LRatioTUFF     PValueTUFF     FirstFailu
    ___________    ______________    ________    ______    __________    __________    __________

     "Equity"      "Normal95"          0.95      accept      1.7354       0.18773            58
     "Equity"      "Normal99"          0.99      accept     0.36686       0.54472           173
     "Equity"      "Historical95"      0.95      accept      1.5348        0.2154            55
     "Equity"      "Historical99"      0.99      accept     0.36686       0.54472           173
     "Equity"      "EWMA95"            0.95      accept     0.13304        0.7153            28
     "Equity"      "EWMA99"            0.99      accept     0.14596       0.70243           143
```

Input Arguments

vbt — varbacktest object
object

`varbacktest` (`vbt`) object, contains a copy of the given data (the `PortfolioData` and `VarData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating a `varbacktest` object, see `varbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = tuff(vbt,'TestLevel',0.99)`

TestLevel — Test confidence level
`0.95` (default) | numeric between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — `tuff` test results
table

`tuff` test results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'TUFF'` — Categorical array with the categories `accept` and `reject` that indicate the result of the `tuff` test
- `'LRatioTUFF'` — Likelihood ratio of the `tuff` test
- `'PValueTUFF'` — P-value of the `tuff` test
- `'FirstFailure'` — Number of periods until the first failure
- `'Observations'` — Number of observations
- `'TestLevel'` — Test confidence level

Note For `tuff` test results, the terms `accept` and `reject` are used for convenience, technically a `tuff` test does not accept a model. Rather, the test fails to reject it.

More About

Time Until First Failure (TUFF) Test

The `tuff` function performs Kupiec's time until first failure test.

The TUFF test is a likelihood ratio test proposed by Kupiec (1995) to assess if the number of periods until the first failure is consistent with the VaR confidence level.

Algorithms

The likelihood ratio (test statistic) of the `tuff` test is given by

$$LRatioTUFF = -2\log\left(\frac{pVaR(1-pVaR)^{n-1}}{\left(\frac{1}{n}\right)\left(1-\frac{1}{n}\right)^{n-1}}\right) = -2(\log(pVaR) + (n-1)\log(1-pVaR) + n\log(n)$$

$$-(n-1)\log(n-1))$$

where n is the number of periods until the first failure and $pVaR = 1 - VaRLevel$. By the properties of the logarithm (if $n = 1$),

$$LRatioTUFF = -2\log(pVaR)$$

This is asymptotically distributed as a chi-square distribution with 1 degree of freedom.

The *p*-value of the `tuff` test is the probability that a chi-square distribution with 1 degree of freedom exceeds the likelihood ratio *LRatioTUFF*

$$PValueTUFF = 1 - F(LRatioTUFF)$$

where `F` is the cumulative distribution of a chi-square variable with 1 degree of freedom.

The result of the test is to accept if

$$F(LRatioTUFF) < F(TestLevel)$$

and reject otherwise, where *F* is the cumulative distribution of a chi-square variable with 1 degree of freedom.

If the sample has no failures, the test statistic is not defined. However, there are two cases distinguished here:

- If the number of observations is large enough that no matter when the first failure occurred it would be too late to pass the test, then the model is rejected. Technically, this happens if the number of observations N is larger than $1/pVaR$ (large enough relative to the VaR confidence level) and if the test fails when $n = N + 1$ (the earliest observation for the first VaR failure). In this case, the likelihood ratio is reported for $n = N + 1$, and the corresponding *p*-value.
- In all other cases, it is not possible to tell with certainty whether the result of the test would eventually be to accept or reject the model. There are ranges of possible first failure values that would result in accepting or rejecting the model. In these cases, the `tuff` function accepts the model and reports undefined (`NaN`) values for the likelihood ratio and *p*-value.

Version History

Introduced in R2016b

References

[1] Kupiec, P. "Techniques for Verifying the Accuracy of Risk Management Models." *Journal of Derivatives.* Vol. 3, 1995, pp. 73 – 84.

See Also

`varbacktest` | `tl` | `pof` | `bin` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests`

Topics

“VaR Backtesting Workflow” on page 2-6
“Value-at-Risk Estimation and Backtesting” on page 2-10
“Overview of VaR Backtesting” on page 2-2
“Kupiec’s POF and TUFF Tests” on page 2-3
“Comparison of ES Backtesting Methods” on page 2-26

ultimateClaims

Compute projected ultimate claims for `expectedClaims` object

Syntax

```
projectedUltimateClaims = ultimateClaims(ec)
```

Description

`projectedUltimateClaims = ultimateClaims(ec)` computes the projected ultimate claims for each origin period, based on the earned premium and the selected claims ratios for an `expectedClaims` object.

Examples

Compute Ultimate Claims for `expectedClaims` Object

Compute the projected ultimate claims for an `expectedClaims` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                       Origin: {10x1 cell}
                  Development: {10x1 cell}
                       Claims: [10x10 double]
               LatestDiagonal: [10x1 double]
                  Description: ""
                   TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
```

```
              SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                         Origin: {10x1 cell}
                    Development: {10x1 cell}
                         Claims: [10x10 double]
                 LatestDiagonal: [10x1 double]
                    Description: ""
                     TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
              SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

          ReportedTriangle: [1x1 developmentTriangle]
              PaidTriangle: [1x1 developmentTriangle]
             EarnedPremium: [10x1 double]
             InitialClaims: [10x1 double]
           CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Use `ultimateClaims` to compute the projected ultimate claims using Expected Claims Technique.

```
projectedUltimateClaims = ultimateClaims(ec)
```

```
projectedUltimateClaims = 10×1
10³ ×

    4.9910
    5.1623
    5.5856
    5.8067
    5.8990
    5.9211
    5.8895
    6.1289
    6.1374
    6.6034
```

Input Arguments

ec — Expected claims
`expectedClaims` object

Expected claims, specified as a previously created `expectedClaims` object.

Data Types: `object`

Output Arguments

`projectedUltimateClaims` — Projected ultimate claims obtained using expected claims technique
vector

Projected ultimate claims obtained using the expected claims technique, returned as a vector.

Version History

Introduced in R2020b

See Also

`ibnr` | `unpaidClaims` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

ultimateClaims

Compute projected ultimate claims for `bornhuetterFerguson` object

Syntax

```
projectedUltimateClaims = ultimateClaims(bf)
projectedUltimateClaims = ultimateClaims( ___ ,referenceClaimsType)
```

Description

`projectedUltimateClaims = ultimateClaims(bf)` computes the projected ultimate claims for each origin period, based on the earned premium and the selected claims ratios for a `bornhuetterFerguson` object.

`projectedUltimateClaims = ultimateClaims( ___ ,referenceClaimsType)` additionally specifies the type of claims data. Specify this argument after the input argument in the previous syntax.

Examples

Compute Projected Ultimate Claims for `bornhuetterFerguson` Object

This example shows how to compute the projected ultimate claims for a `bornhuetterFerguson` object for simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
```

```
                       Claims: [10x10 double]
               LatestDiagonal: [10x1 double]
                  Description: ""
                   TailFactor: 1
   CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
            SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                       Origin: {10x1 cell}
                  Development: {10x1 cell}
                       Claims: [10x10 double]
               LatestDiagonal: [10x1 double]
                  Description: ""
                   TailFactor: 1
   CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
            SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

          ReportedTriangle: [1x1 developmentTriangle]
              PaidTriangle: [1x1 developmentTriangle]
             EarnedPremium: [10x1 double]
             InitialClaims: [10x1 double]
           CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Create a `bornhuetterFerguson` object with reported claims, paid claims, and expected claims to calculate ultimate claims, case outstanding, IBNR, and unpaid claims estimates.

```
bf = bornhuetterFerguson(dT_reported, dT_paid, ec.ultimateClaims)
```

```
bf = 
  bornhuetterFerguson with properties:

     ReportedTriangle: [1x1 developmentTriangle]
         PaidTriangle: [1x1 developmentTriangle]
       ExpectedClaims: [10x1 double]
    PercentUnreported: [10x1 double]
        PercentUnpaid: [10x1 double]
      CaseOutstanding: [10x1 double]
```

Use `ultimateClaims` to compute the projected ultimate claims for each origin period, based on the earned premium and the selected claims ratios.

```
projectedUltimateClaims = ultimateClaims(bf,"reported")

projectedUltimateClaims = 10×1
10³ ×

    5.0894
    5.1851
    5.6421
    5.8384
    5.9358
    5.8617
    5.8639
    6.1550
    6.1069
    6.4968
```

Input Arguments

bf — Bornhuetter-Ferguson
`bornhuetterFerguson` object

Bornhuetter-Ferguson object, specified as a previously created `bornhuetterFerguson` object.

Data Types: `object`

referenceClaimsType — Type of claims data
`'reported'` (default) | character vector with value `'reported'` or `'paid'` | string with value `"reported"` or `"paid"`

(Optional) Type of claims data, specified as a character vector or a string.

Data Types: `char` | `string`

Output Arguments

projectedUltimateClaims — Projected ultimate claims obtained using Bornhuetter-Ferguson technique
vector

Projected ultimate claims obtained using the Bornhuetter-Ferguson technique, returned as a vector.

More About

Ultimate Claims

Ultimate claims are the total sum the insured, its insurer, and/or its reinsurer pay for a fully developed loss. A fully developed loss is the paid losses plus outstanding reported losses and incurred but not reported (IBNR) losses.

Knowing the exact value of ultimate losses might not be possible for a long time after the end of a policy period. Actuaries assist with these projections for purposes of financial modeling and year-end reserve determinations.

Version History

Introduced in R2020b

See Also

`ibnr` | `unpaidClaims` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

ultimateClaims

Compute projected ultimate claims for capeCod object

Syntax

```
projectedUltimateClaims = ultimateClaims(cc)
```

Description

`projectedUltimateClaims = ultimateClaims(cc)` computes the projected ultimate claims for each origin period, based on the earned premium and the selected claims ratios for a `capeCod` object.

Examples

Compute Projected Ultimate Claims for capeCod Object

This example shows how to compute the projected ultimate claims for a `capeCod` object for simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
```

```
              SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create a `capeCod` object where the first input argument is the reported development triangle, the second input argument is the paid development triangle, and the third input is the earned premium.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
cc = capeCod(dT_reported, dT_paid,earnedPremium)
```

```
cc = 
  capeCod with properties:

           ReportedTriangle: [1x1 developmentTriangle]
               PaidTriangle: [1x1 developmentTriangle]
              EarnedPremium: [10x1 double]
              UsedUpPremium: [10x1 double]
       EstimatedClaimRatios: [10x1 double]
         ExpectedClaimRatio: 0.4258
    EstimatedExpectedClaims: [10x1 double]
         PercentUnreported: [10x1 double]
           CaseOutstanding: [10x1 double]
```

Use `ultimateClaims` to compute the projected ultimate claims.

```
projectedUltimateClaims = ultimateClaims(cc)
```

```
projectedUltimateClaims = 10×1
10³ ×

    5.0894
    5.1876
    5.6382
    5.8520
    5.9447
    5.8367
    5.8332
    6.0632
    6.0893
    5.9459
```

Input Arguments

cc — Cape Cod object
capeCod object

Cape Cod object, specified as a previously created `capeCod` object.

Data Types: `object`

Output Arguments

projectedUltimateClaims — Projected ultimate claims obtained using Cape Cod technique
vector

Projected ultimate claims obtained using the Cape Cod technique, returned as a vector.

More About

Ultimate Claims

Ultimate claims are the total sum the insured, its insurer, and/or its reinsurer pay for a fully developed loss. A fully developed loss is the paid losses plus outstanding reported losses and incurred-but-not-reported (IBNR) losses.

Knowing the exact value of ultimate losses might not be possible for a long time after the end of a policy period. Actuaries assist with these projections for purposes of financial modeling and year-end reserve determinations.

Version History

Introduced in R2021a

See Also

`ibnr` | `unpaidClaims` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

ultimateClaims

Compute ultimate claims for `developmentTriangle` object

Syntax

`projectedUltimateClaims = ultimateClaims(dT)`

Description

`projectedUltimateClaims = ultimateClaims(dT)` calculates the projected ultimate claims for each origin period, based on the observed claims and the cumulative development factors.

Examples

Calculate the Projected Ultimate Claims for Development Triangle

Calculate the projected ultimate claims for a `developmentTriangle` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle which, is the standard form for representing claims data.

```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

Use the `cdfSummary` function to calculate CDFs and the percentage of total claims and return a table with the selected link ratios, CDFs, and percentage of total claims.

```
dT.SelectedLinkRatio = [1.1755, 1.0577, 1.0273, 1.0104, 1.0044, 1.0026, 1.0016, 1.0006, 1.0004];
selectedLinkRatiosTable = cdfSummary(dT)
```

```
selectedLinkRatiosTable=3×10 table
                              12-24      24-36      36-48      48-60      60-72      72-84
                             _______    _______    _______    _______    _______    _______

    Selected                  1.1755     1.0577     1.0273     1.0104     1.0044     1.0026
    CDF to Ultimate            1.303     1.1084      1.048     1.0201     1.0096     1.0052
    Percent of Total Claims  0.76747    0.90216    0.95422    0.98027    0.99046    0.99482
```

Use the `ultimateClaims` function to calculate the projected ultimate claims for each origin period, based on the observed claims and the cumulative development factors.

```
projectedUltimateClaims = ultimateClaims(dT)
```

```
projectedUltimateClaims = 10×1
10^3 ×

    5.0894
    5.1820
    5.6310
    5.8188
    5.9093
    5.8284
    5.8293
    6.1353
    6.0911
    6.4444
```

Input Arguments

dT — Development triangle

`developmentTriangle` object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

Output Arguments

`projectedUltimateClaims` — Projected ultimate claims obtained using development technique

vector

Projected ultimate claims obtained using the development technique, returned as a vector.

Version History

Introduced in R2020b

See Also

`view` | `linkRatios` | `linkRatioAverages` | `cdfSummary` | `fullTriangle` | `linkRatiosPlot` | `claimsPlot`

Topics

"Mean Square Error of Prediction for Estimated Ultimate Claims" on page 4-161
"Bootstrap Using Chain Ladder Method" on page 4-168
"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

unconditional

Unconditional expected shortfall backtest by Acerbi and Szekely

Syntax

```
TestResults = unconditional(ebts)
[TestResults,SimTestStatistic] = unconditional(ebts,Name,Value)
```

Description

`TestResults = unconditional(ebts)` runs the unconditional expected shortfall (ES) backtest of Acerbi-Szekely (2014).

`[TestResults,SimTestStatistic] = unconditional(ebts,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Run an ES Unconditional Test

Create an `esbacktestbysim` object.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
```

Generate the ES unconditional test report.

```
TestResults = unconditional(ebts)
```

```
TestResults=3×10 table
    PortfolioID        VaRID         VaRLevel     Unconditional     PValue     TestStatistic     Crit
    ___________    _____________    ________    _____________    ______    _____________     ____

       "S&P"       "t(10) 95%"        0.95         accept        0.093       -0.13342          -0
       "S&P"       "t(10) 97.5%"     0.975         reject        0.031       -0.25011           -(
       "S&P"       "t(10) 99%"        0.99         reject        0.008       -0.57396          -0
```

Input Arguments

`ebts` — `esbacktestbysim` object
object

`esbacktestbysim` (`ebts`) object, contains a copy of the given data (the `PortfolioData`, `VarData`, `ESData`, and `Distribution` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktestbysim` object, see `esbacktestbysim`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[TestResults,SimTestStatistic] = unconditional(ebts,'TestLevel',0.99)`

`TestLevel` — Test confidence level

`0.95` (default) | numeric with values between `0` and `1`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric value between `0` and `1`.

Data Types: `double`

Output Arguments

`TestResults` — Results

table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data
- `'VaRID'` — VaR ID for each of the VaR data columns provided
- `'VaRLevel'` — VaR level for the corresponding VaR data column
- `'Unconditional'`— Categorical array with categories 'accept' and 'reject' that indicate the result of the unconditional test
- `'PValue'`— P-value of the unconditional test
- `'TestStatistic'`— Unconditional test statistic
- `'CriticalValue'`— Critical value for the unconditional test
- `'Observations'`— Number of observations
- `'Scenarios'`— Number of scenarios simulated to get the *p*-values
- `'TestLevel'`— Test confidence level

`SimTestStatistic` — Simulated values of the test statistic

numeric array

Simulated values of the test statistic, returned as a `NumVaRs`-by-`NumScenarios` numeric array.

More About

Unconditional Test by Acerbi and Szekely

The unconditional test is also known as the second Acerbi-Szekely test.

The unconditional test is based on the unconditional relationship

$$ES_t = -E_t\left[\frac{X_t I_t}{p_{VaR}}\right]$$

where

X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period *t*.

P_{VaR} is the probability of VaR failure defined as 1-VaR level.

ES_t is the estimated expected shortfall for period *t*.

I_t is the VaR failure indicator on period *t* with a value of 1 if X_t < -VaR, and 0 otherwise.

The unconditional test statistic is defined as:

$$Z_{uncond} = \frac{1}{Np_{VaR}}\sum_{t=1}^{N}\frac{X_t I_t}{ES_t} + 1$$

Significance of the Test

Under the assumption that the distributional assumptions are correct, the expected value of the test statistic Z_{uncond} is `0`.

This is expressed as

$$E[Z_{uncond}] = 0$$

Negative values of the test statistic indicate risk underestimation. The unconditional test is a one-sided test that rejects when there is evidence that the model underestimates risk (for technical details on the null and alternative hypotheses, see Acerbi-Szekely, 2014). The unconditional test rejects the model when the *p*-value is less than `1` minus the test confidence level.

For more information on the steps to simulate the test statistics and the details for the computation of the*p*-values and critical values, see `simulate`.

Edge Cases

The unconditional test statistic takes a value of `1` when there are no VaR failures in the data or in a simulated scenario.

`1` is also the maximum possible value for the test statistic. When the expected number of failures Np_{VaR} is small, the distribution of the unconditional test statistic has a discrete probability jump at $Z_{uncond} = 1$, and the probability that $Z_{uncond} \leq 1$ is `1`. The *p*-value is set to `1` in these cases, and the test result is to `'accept'`, because there is no evidence of risk underestimation. Scenarios with no failures are more likely as the expected number of failures Np_{VaR} gets smaller.

Version History

Introduced in R2017b

References

[1] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall*. MSCI Inc. December, 2014.

See Also

`summary` | `runtests` | `conditional` | `quantile` | `minBiasRelative` | `minBiasAbsolute` | `simulate` | `esbacktestbysim` | `esbacktestbyde`

Topics

"Expected Shortfall (ES) Backtesting Workflow Using Simulation" on page 2-34
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

unconditionalDE

Unconditional Du-Escanciano (DE) expected shortfall (ES) backtest

Syntax

```
TestResults = unconditionalDE(ebtde)
[TestResults,SimTestStatistic] = unconditionalDE( ___ ,Name,Value)
```

Description

`TestResults = unconditionalDE(ebtde)` runs the unconditional Du-Escanciano (DE) expected shortfall (ES) backtest [1] . The unconditional test supports critical values by large-scale approximation and by finite-sample simulation.

`[TestResults,SimTestStatistic] = unconditionalDE( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input argument in the previous syntax.

Examples

Create an esbacktestbyde Object and Run an UnconditionalDE Test

Create an `esbacktestbyde` object for a *t* model with 10 degrees of freedom, and then run an `unconditionalDE` test.

```
load ESBacktestDistributionData.mat
    rng('default'); % For reproducibility
    ebtde = esbacktestbyde(Returns,"t",...
       'DegreesOfFreedom',T10DoF,...
       'Location',T10Location,...
       'Scale',T10Scale,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
    unconditionalDE(ebtde)
```

```
ans=3×14 table
    PortfolioID        VaRID         VaRLevel     UnconditionalDE      PValue      TestStatistic
    ___________    _____________    ________    _______________    ________    _____________

       "S&P"       "t(10) 95%"        0.95          accept           0.181         0.028821
       "S&P"       "t(10) 97.5%"     0.975          accept        0.086278         0.015998
       "S&P"       "t(10) 99%"        0.99          reject        0.016871        0.0080997
```

Input Arguments

ebtde — esbacktestbyde object
object

esbacktestbyde (ebtde) object, which contains a copy of the data (the PortfolioData, VarData, and ESData properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an esbacktestbyde object, see esbacktestbyde.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: TestResults = unconditionalDE(ebtde,'CriticalValueMethod','large-sample','TestLevel',0.99)

CriticalValueMethod — Method to compute critical values, confidence intervals, and *p*-values

'large-sample' (default) | character vector with values of 'large-sample' or 'simulation' | string with values of "large-sample" or "simulation"

Method to compute critical values, confidence intervals, and *p*-values, specified as the comma-separated pair consisting of 'CriticalValueMethod' and a character vector or string with a value of 'large-sample' or 'simulation'.

Data Types: char | string

TestLevel — Test confidence level

0.95 (default) | numeric value between 0 and 1

Test confidence level, specified as the comma-separated pair consisting of 'TestLevel' and a numeric value between 0 and 1.

Data Types: double

Output Arguments

TestResults — Results

table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following:

- 'PortfolioID' — Portfolio ID for the given data
- 'VaRID' — VaR ID for each of the VaR levels
- 'VaRLevel' — VaR level
- 'UnconditionalDE'— Categorical array with the categories 'accept' and 'reject', which indicate the result of the unconditional DE test
- 'PValue'— *P*-value of the unconditional DE test
- 'TestStatistic'— Unconditional DE test statistic
- 'LowerCI'— Confidence-interval lower limit for the unconditional DE test statistic
- 'UpperCI'— Confidence-interval upper limit for the unconditional DE test statistic
- 'Observations'— Number of observations

- `'CriticalValueMethod'`— Method for computing confidence intervals and *p*-values
- `'MeanLS'`— Mean of the `large-sample` normal distribution; if `CriticalValueMethod` is `'simulation'`, `'MeanLS'` is reported as `NaN`
- `'StdLS'`— Standard deviation of the `large-sample` normal distribution; if `CriticalValueMethod` is `'simulation'`, `'StdLS'` is reported as `NaN`
- `'Scenarios'`— Number of scenarios simulated to get the *p*-values; if `CriticalValueMethod` is `'large-sample'`, the number of scenarios is reported as `NaN`
- `'TestLevel'`— Test confidence level

Note For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

`SimTestStatistic` — Simulated values of the test statistics
numeric array

Simulated values of the test statistics, returned as a `NumVaRs`-by-`NumScenarios` numeric array.

More About

Unconditional DE Test

The unconditional DE test is a two-sided test to check if the test statistic is close to an expected value of α/2, where α = 1- *VaRLevel*.

The test statistic for the unconditional DE test is

$$U_{ES} = \frac{1}{N}\sum_{t=1}^{N} H_t$$

where

- H_t is the cumulative failures or violations process; $H_t = (\alpha - U_t)I(U_t < \alpha) / \alpha$, where $I(x)$ is the indicator function.
- U_t are the ranks or mapped returns $U_t = P_t(X_t)$, where $P_t(X_t) = P(X_t \mid \theta_t)$ is the cumulative distribution of the portfolio outcomes or returns X_t over a given test window $t = 1,\ldots N$ and θ_t are the parameters of the distribution. For simplicity, the subindex t is both the return and the parameters, understanding that the parameters are those used on date t, even though those parameters are estimated on the previous date t-1, or even prior to that.

Significance of the Test

The test statistic U_{ES} is a random variable and a function of random return sequences:

$$U_{ES} = U_{ES}(X_1, \ldots, X_N)\,.$$

For returns observed in the test window 1,...,N, the test statistic attains a fixed value:

$$U_{ES}^{obs} = U_{ES}(X_1^{obs}, \ldots, X_N^{obs})\,.$$

In general, for unknown returns that follow a distribution of P_t, the value of U_{ES} is uncertain and follows a cumulative distribution function:

$$P_U(x) = P[U_{ES} \le x]\,.$$

This distribution function computes a confidence interval and a *p*-value. To determine the distribution P_U, the `esbacktestbyde` class supports the large-sample approximation and simulation methods. You can specify one of these methods by using the optional name-value pair argument `CriticalValueMethod`.

For the large-sample approximation method, the distribution P_U is derived from an asymptotic analysis. If the number of observations N is large, the test statistic U_{ES} is distributed as

$$U_{ES} \xrightarrow[dist]{} N\left(\frac{\alpha}{2}, \frac{\alpha(1/3 - \alpha/4)}{N}\right) = P_U$$

where $N(\mu,\sigma^2)$ is the normal distribution with mean μ and variance σ^2.

Because the test statistic cannot be smaller than 0 or greater than 1, the analytical confidence interval limits are clipped to the interval [0,1]. Therefore, if the analytical value is negative, the test statistic is reset to 0, and if the analytical value is greater than 1, it is reset to 1.

The *p*-value is

$$p_{value} = 2 * \min\left\{P_U(U_{ES}^{obs}), 1 - P_U(U_{ES}^{obs})\right\}.$$

The test rejects if $p_{value} < \alpha_{test}$.

For the simulation method, the distribution $P_U$is estimated as follows

1 Simulate M scenarios of returns as

$$X^s = (X_1^s, \ldots, X_N^s),\ s = 1, \ldots, M\,.$$

2 Compute the corresponding test statistic as

$$U_{ES}^s = U_{ES}^s(X_1^s, \ldots, X_N^s),\ s = 1, \ldots, M\,.$$

3 Define P_U as the empirical distribution of the simulated test statistic values as

$$P_U = P[U_{ES} \le x] = \frac{1}{M} I(U_{ES}^s \le x),$$

where $I(.)$ is the indicator function.

In practice, simulating ranks is more efficient than simulating returns and then transforming the returns into ranks. For more information, see `simulate`.

For the empirical distribution, the value of 1-$P_U(x)$ can differ from the value of $P[U_{ES} \ge x]$ because the distribution may have nontrivial jumps (simulated tied values). Use the latter probability for the estimation of confidence levels and *p*-values.

If α_{test} = 1 - *test confidence level*, then the confidence intervals levels CI_{lower} and CI_{upper} are the values that satisfy equations:

$$P_U(CI_{lower}) = P[CI_{lower} \le U_{ES}] = \frac{\alpha_{test}}{2},$$

$$P\left[U_{ES} \ge CI_{upper}\right] = \frac{\alpha_{test}}{2}\,.$$

The reported confidence interval limits CI_{lower} and CI_{upper} are simulated test statistic values U^{s}_{ES} that approximately solve the preceding equations.

The *p*-value is determined as

$$p_{value} = 2 * \min\left\{P\left[U_{ES} \le U_{ES}^{obs}\right], P\left[U_{ES} \ge U_{ES}^{obs}\right]\right\}.$$

The test rejects if $p_{value} < \alpha_{test}$.

Version History

Introduced in R2019b

References

[1] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science.* Vol. 63, Issue 4, April 2017.

[2] Basel Committee on Banking Supervision. "*Minimum Capital Requirements for Market Risk*". January 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

`esbacktestbyde` | `summary` | `runtests` | `conditionalDE` | `simulate` | `esbacktestbysim`

Topics

"Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano" on page 2-63
"Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano" on page 2-72
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"ES Backtest Using Du-Escanciano Method" on page 2-24
"Comparison of ES Backtesting Methods" on page 2-26

unconditionalNormal

Unconditional expected shortfall (ES) backtest by Acerbi-Szekely with critical values for normal distributions

Syntax

```
TestResults = unconditionalNormal(ebt)
TestResults = unconditionalNormal(ebt,Name,Value)
```

Description

`TestResults = unconditionalNormal(ebt)` runs the unconditional expected shortfall (ES) backtest by Acerbi-Szekely (2014) using precomputed critical values and assuming that the returns distribution is standard normal.

`TestResults = unconditionalNormal(ebt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Run an Unconditional ES Backtest

Create an `esbacktest` object.

```
load ESBacktestData
ebt = esbacktest(Returns,VaRModel1,ESModel1,'VaRLevel',VaRLevel)
```

```
ebt = 
  esbacktest with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x1 double]
           ESData: [1966x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9750
```

Generate the `TestResults` report for the unconditional ES backtest that assumes the returns distribution is standard normal.

```
TestResults = unconditionalNormal(ebt,'TestLevel',0.99)
```

TestResults=*1×9 table*

PortfolioID	VaRID	VaRLevel	UnconditionalNormal	PValue	TestStatistic	Cri
"Portfolio"	"VaR"	0.975	reject	0.0054099	-0.38265	-(

Input Arguments

`ebt` — `esbacktest` object
object

`esbacktest` (`ebt`) object, which contains a copy of the given data (the `PortfolioData`, `VarData`, and `ESData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktest` object, see `esbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = unconditionalNormal(ebt,'TestLevel',0.99)`

`TestLevel` — Test confidence level
`0.95` (default) | numeric value between `0.5` and `0.9999`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric value between `0.5` and `0.9999`.

Data Types: `double`

Output Arguments

`TestResults` — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data.
- `'VaRID'` — VaR ID for each of the VaR data columns provided.
- `'VaRLevel'` — VaR level for the corresponding VaR data column.
- `'UnconditionalNormal'`— Categorical array with categories 'accept' and 'reject' that indicate the result of the unconditional normal test.
- `'PValue'`— P-value of the unconditional normal test, interpolated from the precomputed critical values under the assumption that the returns follow a standard normal distribution.

  Note *p*-values < `0.0001` are truncated to the minimum (`0.0001`) and *p*-values > `0.5` are displayed as a maximum (`0.5`).
- `'TestStatistic'`— Unconditional normal test statistic.
- `'CriticalValue'`— Precomputed critical value for the corresponding test level and number of observations. Critical values are obtained under the assumption that the returns follow a standard normal distribution.
- `'Observations'`— Number of observations.

- 'TestLevel'— Test confidence level.

Note For the test results, the terms 'accept' and 'reject' are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

More About

Unconditional Test by Acerbi and Szekely

The unconditional test (also known as the second Acerbi-Szekely test) scales the losses by the corresponding ES value.

The unconditional test statistic is based on the unconditional relationship

$$ES_t = -E_t\left[\frac{X_t I_t}{p_{VaR}}\right]$$

where

X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period *t*.

P_{VaR} is the probability of VaR failure defined as 1-VaR level.

ES_t is the estimated expected shortfall for period *t*.

I_t is the VaR failure indicator on period *t* with a value of 1 if X_t < -VaR, and 0 otherwise.

The unconditional test statistic is defined as

$$Z_{uncond} = \frac{1}{Np_{VaR}}\sum_{t=1}^{N}\frac{X_t I_t}{ES_t} + 1$$

The critical values for the unconditional test statistic, which form the basis for table-based tests, are stable across a range of distributions. The esbacktest class runs the unconditional test against precomputed critical values under two distributional assumptions: normal distribution (thin tails) using unconditionalNormal and *t* distribution with 3 degrees of freedom (heavy tails) using unconditionalT).

Version History

Introduced in R2017b

References

[1] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall*. MSCI Inc. December, 2014.

See Also

esbacktest | summary | runtests | unconditionalT

Topics

“Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30

“Expected Shortfall Estimation and Backtesting” on page 2-44
“Overview of Expected Shortfall Backtesting” on page 2-20
“Comparison of ES Backtesting Methods” on page 2-26

unconditionalT

Unconditional expected shortfall (ES) backtest by Acerbi-Szekely with critical values for *t* distributions

Syntax

```
TestResults = unconditionalT(ebt)
TestResults = unconditionalT(ebt,Name,Value)
```

Description

`TestResults = unconditionalT(ebt)` runs the unconditional expected shortfall (ES) backtest by Acerbi-Szekely (2014) using precomputed critical values and assuming that the returns distribution is *t* with 3 degrees of freedom.

`TestResults = unconditionalT(ebt,Name,Value)` adds an optional name-value pair argument for `TestLevel`.

Examples

Run an Unconditional `t` ES Backtest

Create an `esbacktest` object.

```
load ESBacktestData
ebt = esbacktest(Returns,VaRModel1,ESModel1,'VaRLevel',VaRLevel)
```

```
ebt = 
  esbacktest with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x1 double]
           ESData: [1966x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9750
```

Generate the `TestResults` report for the unconditional `t` ES backtest that assumes the returns distribution is `t` with 3 degrees of freedom.

```
TestResults = unconditionalT(ebt,'TestLevel',0.99)
```

```
TestResults=1×9 table
    PortfolioID    VaRID    VaRLevel    UnconditionalT     PValue     TestStatistic    CriticalVa
    ___________    _____    ________    ______________    ________    _____________    __________

    "Portfolio"    "VaR"     0.975          accept        0.018566      -0.38265          -0.4298
```

Input Arguments

`ebt` — `esbacktest` object
object

`esbacktest` (`ebt`) object, which contains a copy of the given data (the `PortfolioData`, `VarData`, and `ESData` properties) and all combinations of portfolio ID, VaR ID, and VaR levels to be tested. For more information on creating an `esbacktest` object, see `esbacktest`.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `TestResults = unconditionalT(ebt,'TestLevel',0.99)`

`TestLevel` — Test confidence level
`0.95` (default) | numeric value between `0.5` and `0.9999`

Test confidence level, specified as the comma-separated pair consisting of `'TestLevel'` and a numeric value between `0.5` and `0.9999`.

Data Types: `double`

Output Arguments

`TestResults` — Results
table

Results, returned as a table where the rows correspond to all combinations of portfolio ID, VaR ID, and VaR levels to be tested. The columns correspond to the following information:

- `'PortfolioID'` — Portfolio ID for the given data.
- `'VaRID'` — VaR ID for each of the VaR data columns provided.
- `'VaRLevel'` — VaR level for the corresponding VaR data column.
- `'UnconditionalT'`— Categorical array with categories 'accept' and 'reject' indicating the result of the unconditional *t* test.
- `'PValue'`— P-value of the unconditional *t* test, interpolated from the precomputed critical values under the assumption that the returns follow a standard normal distribution.

  Note *p*-values < `0.0001` are truncated to the minimum (`0.0001`) and *p*-values > `0.5` are displayed as a maximum (`0.5`).

- `'TestStatistic'`— Unconditional *t* test statistic.
- `'CriticalValue'`— Precomputed critical value for the corresponding test level and number of observations. Critical values are obtained under the assumption that the returns follow a *t* distribution with 3 degrees of freedom.
- `'Observations'`— Number of observations.

- `'TestLevel'`— Test confidence level.

Note For the test results, the terms `'accept'` and `'reject'` are used for convenience. Technically, a test does not accept a model; rather, a test fails to reject it.

More About

Unconditional Test by Acerbi and Szekely

The unconditional test (also known as the second Acerbi-Szekely test) scales the losses by the corresponding ES value.

The unconditional test statistic is based on the unconditional relationship

$$ES_t = -E_t\left[\frac{X_t I_t}{p_{VaR}}\right]$$

where

X_t is the portfolio outcome, that is, the portfolio return or portfolio profit and loss for period *t*.

P_{VaR} is the probability of VaR failure defined as 1-VaR level.

ES_t is the estimated expected shortfall for period *t*.

I_t is the VaR failure indicator on period *t* with a value of 1 if X_t < -VaR, and 0 otherwise.

The unconditional test statistic is defined as:

$$Z_{uncond} = \frac{1}{Np_{VaR}}\sum_{t=1}^{N}\frac{X_t I_t}{ES_t} + 1$$

The critical values for the unconditional test statistic, which form the basis for table-based tests, are stable across a range of distributions. The `esbacktest` class runs the unconditional test against precomputed critical values under two distributional assumptions: normal distribution (thin tails) using `unconditionalNormal` and *t* distribution with 3 degrees of freedom (heavy tails) using `unconditionalT`.

Version History

Introduced in R2017b

References

[1] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall*. MSCI Inc. December, 2014.

See Also

`esbacktest` | `summary` | `runtests` | `unconditionalNormal`

Topics

“Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information” on page 2-30

“Expected Shortfall Estimation and Backtesting” on page 2-44
“Overview of Expected Shortfall Backtesting” on page 2-20
“Comparison of ES Backtesting Methods” on page 2-26

unpaidClaims

Compute unpaid claims estimates for `bornhuetterFerguson` object

Syntax

```
unpaidClaimsEstimate = unpaidClaims(bf)
unpaidClaimsEstimate = unpaidClaims( ___ ,referenceClaimsType)
```

Description

`unpaidClaimsEstimate = unpaidClaims(bf)` computes unpaid claims estimates for a `bornhuetterFerguson` object.

`unpaidClaimsEstimate = unpaidClaims( ___ ,referenceClaimsType)` additionally specifies the type of claims data. Specify this argument after the input argument in the previous syntax.

Examples

Compute Unpaid Claims Estimates for `bornhuetterFerguson` Object

Compute unpaid claims estimates for a `bornhuetterFerguson` object for simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                   Origin: {10x1 cell}
              Development: {10x1 cell}
                   Claims: [10x10 double]
           LatestDiagonal: [10x1 double]
```

```
                    Description: ""
                     TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                         Origin: {10x1 cell}
                    Development: {10x1 cell}
                         Claims: [10x10 double]
                 LatestDiagonal: [10x1 double]
                    Description: ""
                     TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

          ReportedTriangle: [1x1 developmentTriangle]
              PaidTriangle: [1x1 developmentTriangle]
             EarnedPremium: [10x1 double]
             InitialClaims: [10x1 double]
           CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Create a `bornhuetterFerguson` object with reported claims, paid claims, and expected claims to calculate the ultimate claims, cases outstanding, IBNR claims, and unpaid claims estimates.

```
bf = bornhuetterFerguson(dT_reported, dT_paid, ec.ultimateClaims)
```

```
bf = 
  bornhuetterFerguson with properties:

     ReportedTriangle: [1x1 developmentTriangle]
         PaidTriangle: [1x1 developmentTriangle]
       ExpectedClaims: [10x1 double]
    PercentUnreported: [10x1 double]
        PercentUnpaid: [10x1 double]
      CaseOutstanding: [10x1 double]
```

Use `unpaidClaims` to to compute the unpaid claims estimates for the `bornhuetterFerguson` object.

```
unpaidClaimsEstimate = unpaidClaims(bf,"reported")
```

```
unpaidClaimsEstimate = 10×1
10³ ×

    0.1968
    0.0506
    0.1299
    0.1095
    0.1767
    0.0981
    0.3915
    0.9838
    1.7208
    3.7320
```

Input Arguments

bf — Bornhuetter-Ferguson
`bornhuetterFerguson` object

Bornhuetter-Ferguson object, specified as a previously created `bornhuetterFerguson` object.

Data Types: `object`

referenceClaimsType — Type of claims data
`'reported'` (default) | character vector with value `'reported'` or `'paid'` | string with value `"reported"` or `"paid"`

Type of claims data, specified as a character vector or a string.

Data Types: `char` | `string`

Output Arguments

unpaidClaimsEstimate — Unpaid claims estimates
array

Unpaid claims estimates, returned as an array.

More About

Unpaid Claims

Unpaid claims are claims reserves for events that have occurred, including both reported and incurred-but-not-reported (IBNR) reserves, as well as the expenses of settling such claims.

Version History

Introduced in R2020b

See Also

`ultimateClaims` | `ibnr` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

unpaidClaims

Compute unpaid claims estimates for `capeCod` object

Syntax

```
unpaidClaimsEstimate = unpaidClaims(cc)
```

Description

`unpaidClaimsEstimate = unpaidClaims(cc)` computes unpaid claims estimates for a `capeCod` object.

Examples

Compute Unpaid Claims Estimate for capeCod Object

This example shows how to compute the unpaid claims estimates for a `capeCod` object for simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.001
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.001
```

Create a `capeCod` object where the first input argument is the reported development triangle, the second input argument is the paid development triangle, and the third input is the earned premium.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
cc = capeCod(dT_reported, dT_paid,earnedPremium)
```

```
cc = 
  capeCod with properties:

            ReportedTriangle: [1x1 developmentTriangle]
                PaidTriangle: [1x1 developmentTriangle]
               EarnedPremium: [10x1 double]
               UsedUpPremium: [10x1 double]
        EstimatedClaimRatios: [10x1 double]
          ExpectedClaimRatio: 0.4258
    EstimatedExpectedClaims: [10x1 double]
           PercentUnreported: [10x1 double]
             CaseOutstanding: [10x1 double]
```

Use `unpaidClaims` to compute the unpaid claims estimates.

```
unpaidClaimsEstimate = unpaidClaims(cc)
```

```
unpaidClaimsEstimate = 10×1
10^3 ×

    0.1968
    0.0531
    0.1259
    0.1232
    0.1856
    0.0731
    0.3609
    0.8920
    1.7032
    3.1811
```

Input Arguments

`cc` — Cape Cod object
`capeCod` object

Cape Cod object, specified as a previously created `capeCod` object.

Data Types: `object`

Output Arguments

`unpaidClaimsEstimate` — Unpaid claims estimates
array

Unpaid claims estimates, returned as an array.

More About

Unpaid Claims

Unpaid claims are claims reserves for events that have occurred, including both reported and incurred-but-not-reported (IBNR) reserves, as well as the expenses of settling such claims.

Version History

Introduced in R2021a

See Also

`ibnr` | `ultimateClaims` | `summary`

Topics
“Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

unpaidClaims

Compute unpaid claims for `chainLadder` object

Syntax

```
unpaidClaimsEstimate = unpaidClaims(cl)
unpaidClaimsEstimate = unpaidClaims( ___ ,referenceClaimsType)
```

Description

`unpaidClaimsEstimate = unpaidClaims(cl)` computes unpaid claims for the `chainLadder` object.

`unpaidClaimsEstimate = unpaidClaims( ___ ,referenceClaimsType)` specifies options using one or more optional arguments in addition to the input argument in the previous syntax.

Examples

Calculate the Unpaid Claims for chainLadder

Calculate the unpaid claims for a `chainLadder` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                  Origin: {10x1 cell}
             Development: {10x1 cell}
                  Claims: [10x10 double]
          LatestDiagonal: [10x1 double]
             Description: ""
```

```
                  TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
           SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid =
  developmentTriangle with properties:

                      Origin: {10x1 cell}
                 Development: {10x1 cell}
                      Claims: [10x10 double]
              LatestDiagonal: [10x1 double]
                 Description: ""
                  TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
           SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create a `chainLadder` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
cl = chainLadder(dT_reported, dT_paid)
```

```
cl =
  chainLadder with properties:

    ReportedTriangle: [1x1 developmentTriangle]
        PaidTriangle: [1x1 developmentTriangle]
     CaseOutstanding: [10x1 double]
```

Use `ibnr` to compute the incurred-but-not-reported (IBNR).

```
ibnrClaims = ibnr(cl,'reported')
```

```
ibnrClaims = 10×1
10^3 ×

         0
    0.0052
    0.0169
    0.0349
    0.0575
    0.0880
    0.1489
    0.3019
    0.6084
    1.5181
```

Use `unpaidClaims` to compute the unpaid claims.

```
unpaidClaimsEstimate = unpaidClaims(cl,'reported')
```

```
unpaidClaimsEstimate = 10×1
10^3 ×
```

```
    0.1968
    0.0506
    0.1300
    0.1097
    0.1771
    0.0972
    0.3908
    0.9851
    1.7175
    3.6992
```

Input Arguments

`cl` — Chain ladder
`chainLadder` object

Chain ladder, specified as a previously created `chainLadder` object.

Data Types: `object`

`referenceClaimsType` — Type of claims data
`'reported'` (default) | character vector with value `'reported'` or `'paid'` | string with value `"reported"` or `"paid"`

(Optional) Type of claims data, specified as a character vector or string.

Data Types: `char` | `string`

Output Arguments

`unpaidClaimsEstimate` — Unpaid claims estimates
array

Unpaid claims estimates, returned as an array.

More About

Unpaid Claims

Unpaid claims are claims reserves for events that have occurred, including both reported and incurred-but-not-reported (IBNR) reserves, as well as the expenses of settling such claims.

Version History

Introduced in R2020b

See Also

`ibnr` | `summary`

Topics
"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

unpaidClaims

Compute unpaid claims estimates for `expectedClaims` object

Syntax

`unpaidClaimsEstimate = unpaidClaims(ec)`

Description

`unpaidClaimsEstimate = unpaidClaims(ec)` computes unpaid claims estimates for an `expectedClaims` object.

Examples

Compute Unpaid Claims Estimates for expectedClaims Object

Compute unpaid claims estimates for an `expectedClaims` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                        Origin: {10x1 cell}
                   Development: {10x1 cell}
                        Claims: [10x10 double]
                LatestDiagonal: [10x1 double]
                   Description: ""
                    TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
             SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

         ReportedTriangle: [1x1 developmentTriangle]
             PaidTriangle: [1x1 developmentTriangle]
            EarnedPremium: [10x1 double]
            InitialClaims: [10x1 double]
          CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Use `unpaidClaims` to compute the unpaid claims estimates.

```
unpaidClaimsEstimate = unpaidClaims(ec)
```

```
unpaidClaimsEstimate = 10×1
10^3 ×

    0.0984
    0.0279
    0.0733
    0.0778
    0.1399
    0.1575
    0.4171
    0.9577
    1.7513
    3.8386
```

Input Arguments

ec — Expected claims
`expectedClaims` object

Expected claims, specified as a previously created `expectedClaims` object.

Data Types: `object`

Output Arguments

`unpaidClaimsEstimate` — Unpaid claims estimates
array

Unpaid claims estimates, returned as an array.

More About

Unpaid Claims

Unpaid claims are claims reserves for events that have occurred, including both reported and incurred-but-not-reported (IBNR) reserves, as well as the expenses of settling such claims.

Version History

Introduced in R2020b

See Also

`ultimateClaims` | `ibnr` | `summary`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

validatemodel

Validate quality of compact credit scorecard model

Syntax

```
Stats = validatemodel(csc,data)
[Stats,T] = validatemodel( ___ ,Name,Value)
[Stats,T,hf] = validatemodel( ___ ,Name,Value)
```

Description

`Stats = validatemodel(csc,data)` validates the quality of the `compactCreditScorecard` model for the data set specified using the argument `data`.

`[Stats,T] = validatemodel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the outputs `Stats` and `T`.

`[Stats,T,hf] = validatemodel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax and returns the outputs `Stats` and `T` and the figure handle `hf` to the CAP, ROC, and KS plots.

Examples

Validate a Compact Credit Scorecard Model

Compute model validation statistics for a compact credit scorecard model.

To create a `compactCreditScorecard` object, you must first develop a credit scorecard model using a `creditscorecard` object.

Create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData.mat
sc = creditscorecard(data, 'IDVar','CustID')
```

```
sc = 
  creditscorecard with properties:

                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
        NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: 'CustID'
            PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                     Data: [1200x11 table]
```

Perform automatic binning using the default options. By default, `autobinning` uses the `Monotone` algorithm.

```
sc = autobinning(sc);
```

Fit the model.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    _____     __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Format the unscaled points.

```
sc = formatpoints(sc, 'PointsOddsAndPDO',[500,2,50]);
```

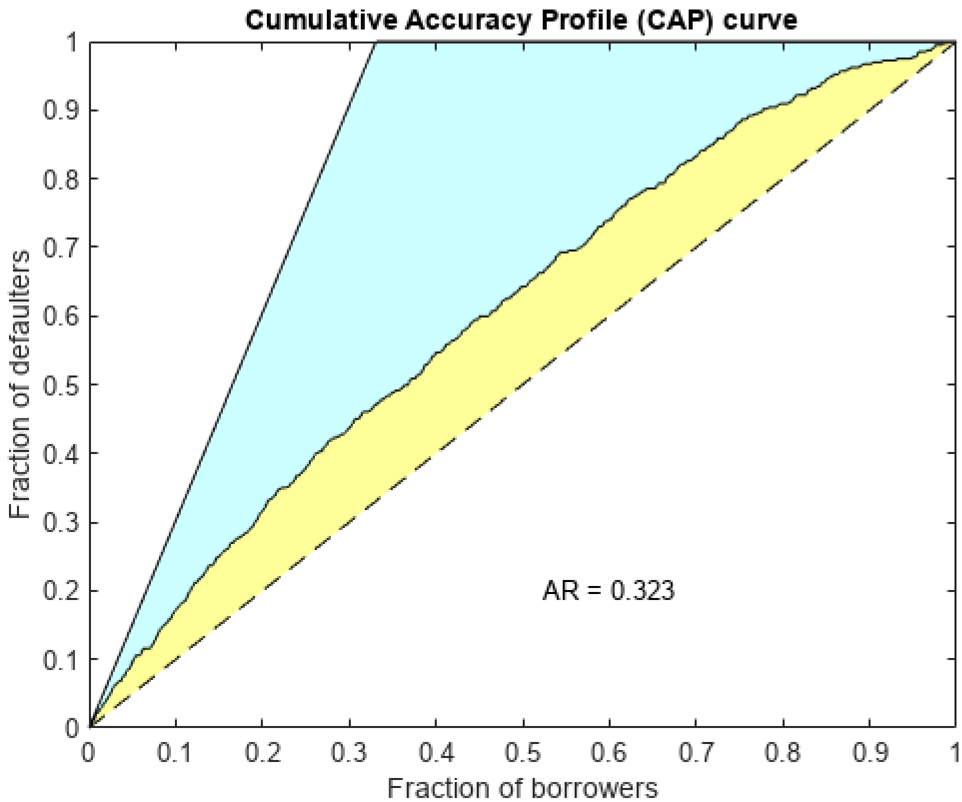
Convert the `creditscorecard` object into a `compactCreditScorecard` object. A `compactCreditScorecard` object is a lightweight version of a `creditscorecard` object that is used for deployment purposes.

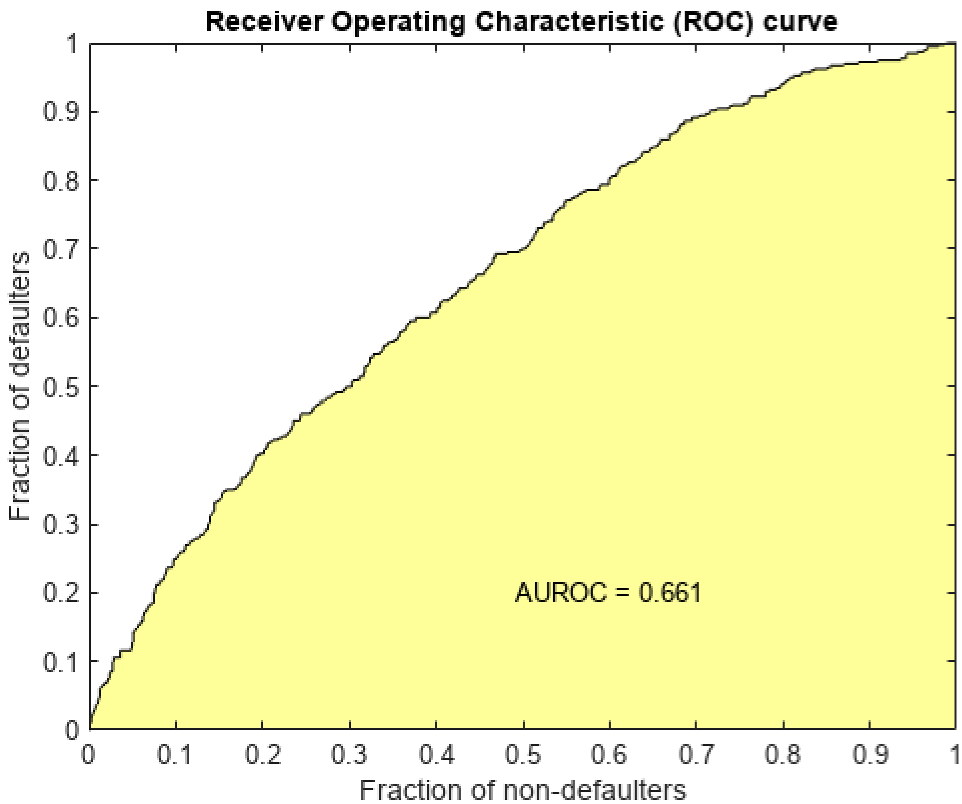
```
csc = compactCreditScorecard(sc);
```

Validate the compact credit scorecard model by generating the CAP, ROC, and KS plots. This example uses the training `data`. However, you can use any validation data, as long as:

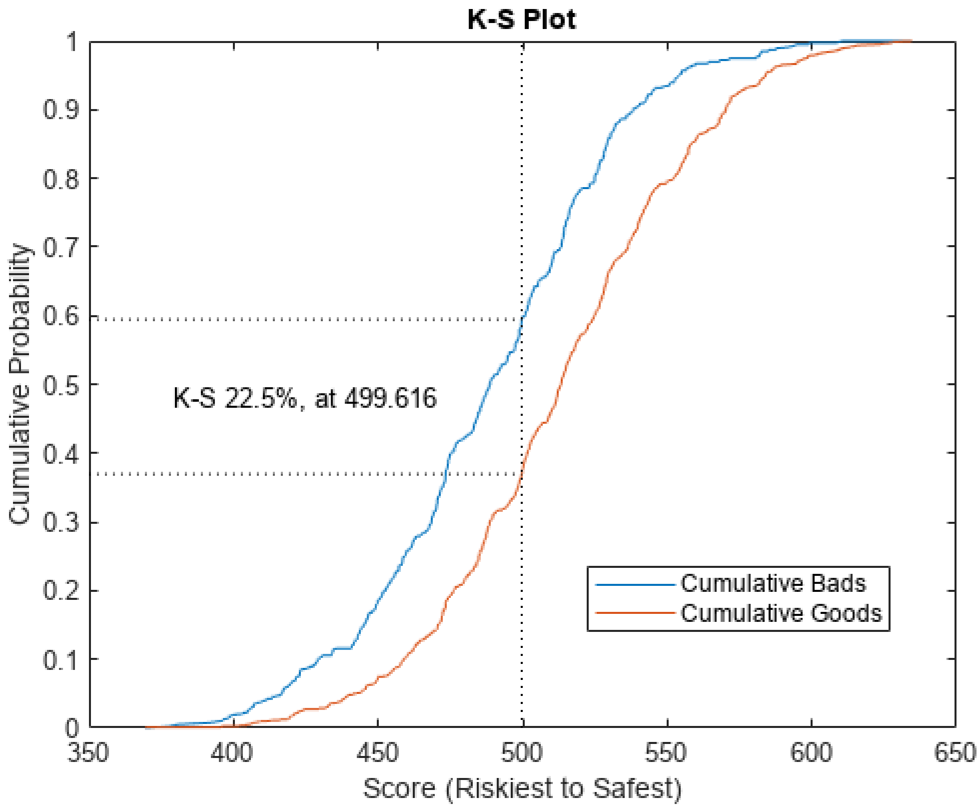
- The data has the same predictor names and predictor types as the data used to create the initial `creditscorecard` object.
- The data has a response column with the same name as the `'ResponseVar'` property in the initial `creditscorecard` object.
- The data has a weights column (if weights were used to train the model) with the same name as `'WeightsVar'` property in the initial `creditscorecard` object.

```
[Stats,T] = validatemodel(csc,data,'Plot',{'CAP','ROC','KS'});
```





Receiver Operating Characteristic (ROC) curve
Fraction of defaulters
Fraction of non-defaulters
AUROC = 0.661
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1




```
disp(Stats)
```

Measure	Value
{'Accuracy Ratio' }	0.32258
{'Area under ROC curve'}	0.66129
{'KS statistic' }	0.2246
{'KS score' }	499.62

```
disp(T(1:15,:))
```

Scores	ProbDefault	TrueBads	FalseBads	TrueGoods	FalseGoods	Sensitivity
369.54	0.75313	0	1	802	397	0
378.19	0.73016	1	1	802	396	0.0025189
380.28	0.72444	2	1	802	395	0.0050378
391.49	0.69234	3	1	802	394	0.0075567
395.57	0.68017	4	1	802	393	0.010076
396.14	0.67846	4	2	801	393	0.010076
396.45	0.67752	5	2	801	392	0.012594
398.61	0.67094	6	2	801	391	0.015113
398.68	0.67072	7	2	801	390	0.017632
401.33	0.66255	8	2	801	389	0.020151
402.66	0.65842	8	3	800	389	0.020151
404.25	0.65346	9	3	800	388	0.02267
404.73	0.65193	9	4	799	388	0.02267

```
    405.53      0.64941         11            4           799           386           0.027708      (
     405.7      0.64887         11            5           798           386           0.027708      (
```

Validate a Compact Credit Scorecard Model with Weights

Compute model validation statistics for a compact credit scorecard model with weights.

To create a `compactCreditScorecard` object, you must first develop a credit scorecard model using a `creditscorecard` object.

Use the `CreditCardData.mat` file to load the data (`dataWeights`) that contains a column (`RowWeights`) for the weights (using a dataset from Refaat 2011).

```
load CreditCardData.mat
```

Create a `creditscorecard` object using the optional name-value pair argument `'WeightsVar'`.

```
sc = creditscorecard(dataWeights,'IDVar','CustID','WeightsVar','RowWeights')
```

```
sc =
  creditscorecard with properties:

              GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: 'RowWeights'
               VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
      NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
         BinMissingData: 0
                  IDVar: 'CustID'
          PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tn
                   Data: [1200x12 table]
```

Perform automatic binning. By default, `autobinning` uses the `Monotone` algorithm.

```
sc = autobinning(sc)
```

```
sc =
  creditscorecard with properties:

              GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: 'RowWeights'
               VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
      NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
         BinMissingData: 0
                  IDVar: 'CustID'
          PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'Tn
                   Data: [1200x12 table]
```

Fit the model.

```
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 764.3187, Chi2Stat = 15.81927, PValue = 6.968927e-05
2. Adding TmWBank, Deviance = 751.0215, Chi2Stat = 13.29726, PValue = 0.0002657942
3. Adding AMBalance, Deviance = 743.7581, Chi2Stat = 7.263384, PValue = 0.007037455

Generalized linear regression model:
    logit(status) ~ 1 + CustIncome + TmWBank + AMBalance
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE        tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70642     0.088702     7.964    1.6653e-15
    CustIncome      1.0268      0.25758    3.9862    6.7132e-05
    TmWBank         1.0973      0.31294    3.5063     0.0004543
    AMBalance       1.0039      0.37576    2.6717     0.0075464


1200 observations, 1196 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 36.4, p-value = 6.22e-08
```

Format the unscaled points.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500,2,50]);
```

Convert the `creditscorecard` object into a `compactCreditScorecard` object. A `compactCreditScorecard` object is a lightweight version of a `creditscorecard` object that is used for deployment purposes.

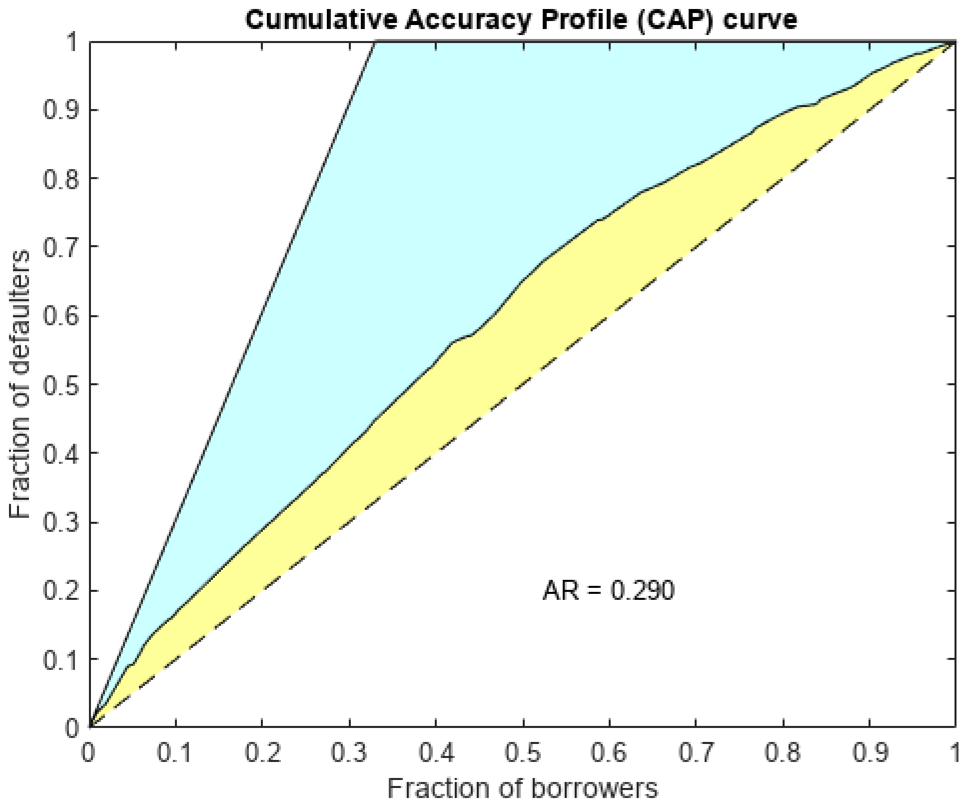
```
csc = compactCreditScorecard(sc);
```

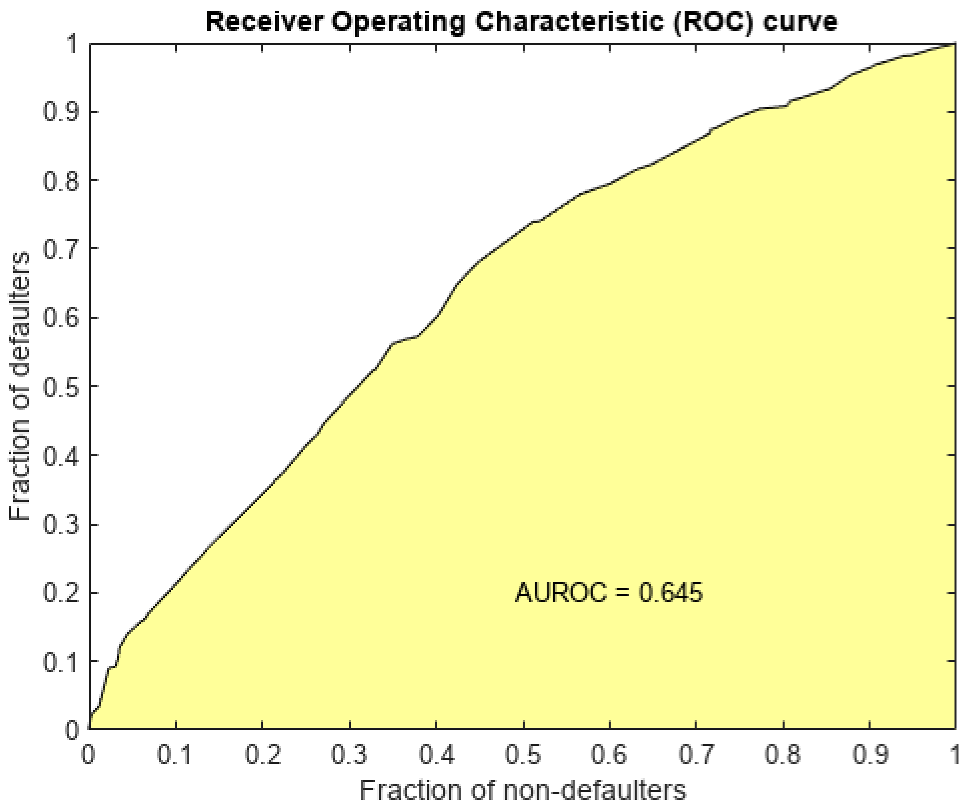
Validate the compact credit scorecard model by generating the CAP, ROC, and KS plots. When you use the optional name-value pair argument `'WeightsVar'` to specify observation (sample) weights in the original `creditscorecard` object, the `T` table for `validatemodel` uses statistics, sums, and cumulative sums that are weighted counts.

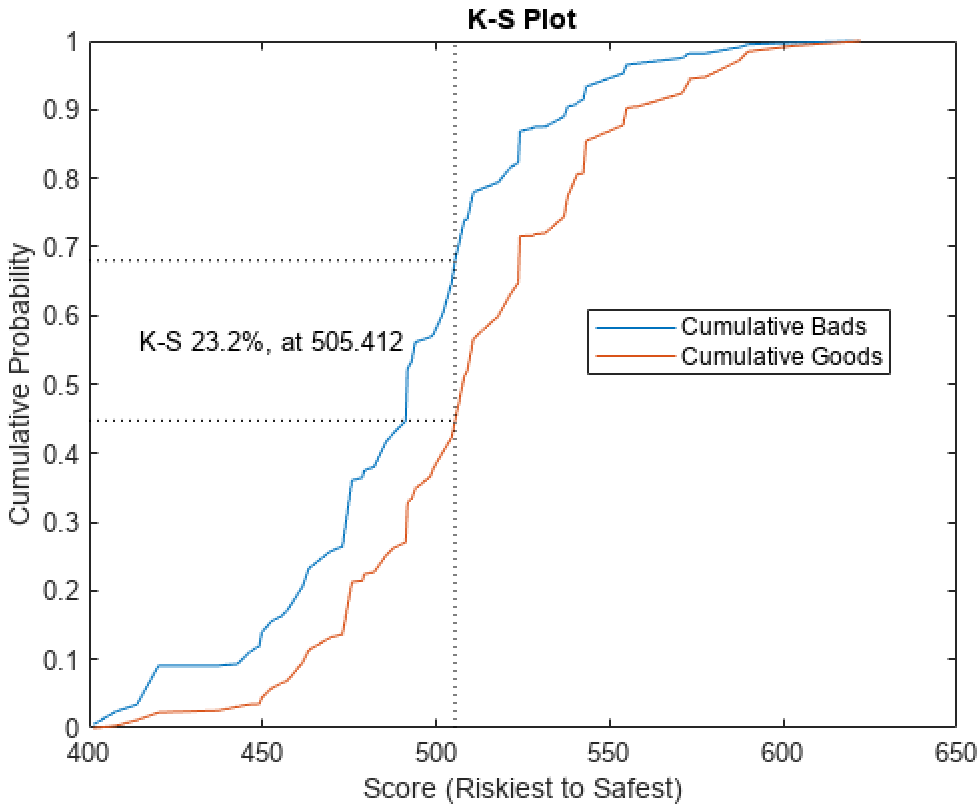
This example uses the training data (`dataWeights`). However, you can use any validation data, as long as:

- The data has the same predictor names and predictor types as the data used to create the initial `creditscorecard` object.
- The data has a response column with the same name as the `'ResponseVar'` property in the initial `creditscorecard` object.
- The data has a weights column (if weights were used to train the model) with the same name as the `'WeightsVar'` property in the initial `creditscorecard` object.

```
[Stats,T] = validatemodel(csc,dataWeights,'Plot',{'CAP','ROC','KS'});
```


Cumulative Accuracy Profile (CAP) curve
Fraction of defaulters
Fraction of borrowers
AR = 0.290
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Receiver Operating Characteristic (ROC) curve
Fraction of defaulters
Fraction of non-defaulters
AUROC = 0.645
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1




```
Stats
```

Stats=*4×2 table*

Measure	Value
{'Accuracy Ratio' }	0.28972
{'Area under ROC curve'}	0.64486
{'KS statistic' }	0.23215
{'KS score' }	505.41

```
T(1:10,:)
```

ans=*10×9 table*

Scores	ProbDefault	TrueBads	FalseBads	TrueGoods	FalseGoods	Sensitivity
401.34	0.66253	1.0788	0	411.95	201.95	0.0053135
407.59	0.64289	4.8363	1.2768	410.67	198.19	0.023821
413.79	0.62292	6.9469	4.6942	407.25	196.08	0.034216
420.04	0.60236	18.459	9.3899	402.56	184.57	0.090918
437.27	0.544	18.459	10.514	401.43	184.57	0.090918
442.83	0.52481	18.973	12.794	399.15	184.06	0.093448
446.19	0.51319	22.396	14.15	397.8	180.64	0.11031
449.08	0.50317	24.325	14.405	397.54	178.71	0.11981
449.73	0.50095	28.246	18.049	393.9	174.78	0.13912

```
    452.44      0.49153       31.511       23.565       388.38       171.52          0.1552
```

Validate a Compact Credit Score Card Model When Using the 'BinMissingData' Option

Compute model validation statistics and assign points for missing data when using the 'BinMissingData' option.

- Predictors in a creditscorecard object that have missing data in the training set have an explicit bin for <missing> with corresponding points in the final scorecard. These points are computed from the Weight-of-Evidence (WOE) value for the <missing> bin and the logistic model coefficients. For scoring purposes, these points are assigned to missing values and to out-of-range values, and after you convert the creditscorecard object to a compactCreditScorecard object, you can use the final score to compute model validation statistics with validatemodel.
- Predictors in a creditscorecard object with no missing data in the training set have no <missing> bin, so no WOE can be estimated from the training data. By default, the points for missing and out-of-range values are set to NaN resulting in a score of NaN when running score. For predictors in a creditscorecard object that have no explicit <missing> bin, use the name-value argument 'Missing' in formatpoints to specify how the function treats missing data for scoring purposes. After converting the creditscorecard object to a compactCreditScorecard object, you can use the final score to compute model validation statistics with validatemodel.

To create a compactCreditScorecard object, you must first develop a credit scorecard model using a creditscorecard object.

Create a creditscorecard object using the CreditCardData.mat file to load dataMissing, a table that contains missing values.

```
load CreditCardData.mat
head(dataMissing,5)
```

```
    CustID     CustAge    TmAtAddress     ResStatus      EmpStatus     CustIncome     TmWBank     Othe
    ______     _______    ___________    ___________    __________    ___________    _______     ____

      1          53           62         <undefined>    Unknown          50000          55          Ye
      2          61           22         Home Owner     Employed         52000          25          Ye
      3          47           30         Tenant         Employed         37000          61          No
      4         NaN           75         Home Owner     Employed         53000          20          Ye
      5          68           56         Home Owner     Employed         53000          14          Ye
```

Use creditscorecard with the name-value argument 'BinMissingData' set to true to bin the missing numeric or categorical data in a separate bin. Apply automatic binning.

```
sc = creditscorecard(dataMissing,'IDVar','CustID','BinMissingData',true);
sc = autobinning(sc);

disp(sc)
```

```
  creditscorecard with properties:

              GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: ''
```

```
                VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustI
       NumericPredictors: {'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalance'  'Uti
   CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
          BinMissingData: 1
                   IDVar: 'CustID'
           PredictorVars: {'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIncome'  'T
                    Data: [1200x11 table]
```

To make any negative age or income information invalid or "out of range," set a minimum value of zero for `'CustAge'` and `'CustIncome'`. For scoring and probability-of-default computations, out-of-range values are given the same points as missing values.

```
sc = modifybins(sc,'CustAge','MinValue',0);
sc = modifybins(sc,'CustIncome','MinValue',0);
```

Display bin information for numeric data for `'CustAge'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'CustAge');
disp(bi)
```

```
         Bin         Good    Bad     Odds       WOE       InfoValue
    _____________    ____    ___    ______    ________    __________

    {'[0,33)'   }     69      52    1.3269    -0.42156      0.018993
    {'[33,37)'  }     63      45       1.4    -0.36795      0.012839
    {'[37,40)'  }     72      47    1.5319     -0.2779     0.0079824
    {'[40,46)'  }    172      89    1.9326    -0.04556     0.0004549
    {'[46,48)'  }     59      25      2.36     0.15424     0.0016199
    {'[48,51)'  }     99      41    2.4146     0.17713     0.0035449
    {'[51,58)'  }    157      62    2.5323     0.22469     0.0088407
    {'[58,Inf]' }     93      25      3.72     0.60931      0.032198
    {'<missing>'}     19      11    1.7273    -0.15787    0.00063885
    {'Totals'   }    803     397    2.0227         NaN      0.087112
```

Display bin information for categorical data for `'ResStatus'` that includes missing data in a separate bin labelled `<missing>`.

```
bi = bininfo(sc,'ResStatus');
disp(bi)
```

```
          Bin           Good    Bad     Odds        WOE        InfoValue
    ______________    ____    ___    ______    _________    __________

    {'Tenant'    }     296     161    1.8385    -0.095463     0.0035249
    {'Home Owner'}     352     171    2.0585     0.017549    0.00013382
    {'Other'     }     128      52    2.4615      0.19637     0.0055808
    {'<missing>' }      27      13    2.0769     0.026469    2.3248e-05
    {'Totals'    }     803     397    2.0227          NaN     0.0092627
```

For the `'CustAge'` and `'ResStatus'` predictors, the training data contains missing data (`NaN`s and `<undefined>` values. For missing data in these predictors, the binning process estimates WOE values of `-0.15787` and `0.026469`, respectively.

Because the training data contains no missing values for the `'EmpStatus'` and `'CustIncome'` predictors, neither predictor has an explicit bin for missing values.

```
bi = bininfo(sc,'EmpStatus');
disp(bi)
```

```
         Bin          Good    Bad     Odds       WOE       InfoValue
    ____________      ____    ___    ______    ________    _________

    {'Unknown' }      396     239    1.6569    -0.19947    0.021715 
    {'Employed'}      407     158    2.5759      0.2418    0.026323 
    {'Totals'  }      803     397    2.0227         NaN    0.048038 
```

```
bi = bininfo(sc,'CustIncome');
disp(bi)
```

```
           Bin             Good    Bad     Odds        WOE       InfoValue 
    _________________      ____    ___    _______    _________    __________

    {'[0,29000)'    }       53      58    0.91379     -0.79457       0.06364
    {'[29000,33000)'}       74      49     1.5102     -0.29217     0.0091366
    {'[33000,35000)'}       68      36     1.8889     -0.06843    0.00041042
    {'[35000,40000)'}      193      98     1.9694    -0.026696    0.00017359
    {'[40000,42000)'}       68      34          2    -0.011271    1.0819e-05
    {'[42000,47000)'}      164      66     2.4848      0.20579     0.0078175
    {'[47000,Inf]'  }      183      56     3.2679      0.47972      0.041657
    {'Totals'       }      803     397     2.0227          NaN       0.12285
```

Use `fitmodel` to fit a logistic regression model using Weight of Evidence (WOE) data. `fitmodel` internally transforms all the predictor variables into WOE values by using the bins found in the automatic binning process. `fitmodel` then fits a logistic regression model using a stepwise method (by default). For predictors that have missing data, there is an explicit `<missing>` bin, with a corresponding WOE value computed from the data. When you use `fitmodel`, the function applies the corresponding WOE value for the `<missing>` bin when performing the WOE transformation.

```
[sc,mdl] = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1442.8477, Chi2Stat = 4.4974731, PValue = 0.033944979
6. Adding ResStatus, Deviance = 1438.9783, Chi2Stat = 3.86941, PValue = 0.049173805
7. Adding OtherCC, Deviance = 1434.9751, Chi2Stat = 4.0031966, PValue = 0.045414057

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70229     0.063959     10.98    4.7498e-28
    CustAge        0.57421      0.25708    2.2335      0.025513
    ResStatus       1.3629      0.66952    2.0356       0.04179
    EmpStatus      0.88373       0.2929    3.0172      0.002551
    CustIncome     0.73535       0.2159     3.406    0.00065929
    TmWBank         1.1065      0.23267    4.7556    1.9783e-06
    OtherCC         1.0648      0.52826    2.0156      0.043841
    AMBalance       1.0446      0.32197    3.2443     0.0011775


1200 observations, 1192 error degrees of freedom
```

```
Dispersion: 1
Chi^2-statistic vs. constant model: 88.5, p-value = 2.55e-16
```

Scale the scorecard points by the "points, odds, and points to double the odds (PDO)" method using the `'PointsOddsAndPDO'` argument of `formatpoints`. Suppose that you want a score of 500 points to have odds of 2 (twice as likely to be good than to be bad) and that the odds double every 50 points (so that 550 points would have odds of 4).

Display the scorecard showing the scaled points for predictors retained in the fitting model.

```
sc = formatpoints(sc,'PointsOddsAndPDO',[500 2 50]);
PointsInfo = displaypoints(sc)
```

```
PointsInfo=38×3 table
     Predictors           Bin           Points
    ______________    _______________   ______

    {'CustAge'   }    {'[0,33)'     }   54.062
    {'CustAge'   }    {'[33,37)'    }   56.282
    {'CustAge'   }    {'[37,40)'    }   60.012
    {'CustAge'   }    {'[40,46)'    }   69.636
    {'CustAge'   }    {'[46,48)'    }   77.912
    {'CustAge'   }    {'[48,51)'    }    78.86
    {'CustAge'   }    {'[51,58)'    }    80.83
    {'CustAge'   }    {'[58,Inf]'   }    96.76
    {'CustAge'   }    {'<missing>'  }   64.984
    {'ResStatus'}     {'Tenant'     }   62.138
    {'ResStatus'}     {'Home Owner'}    73.248
    {'ResStatus'}     {'Other'      }   90.828
    {'ResStatus'}     {'<missing>'  }   74.125
    {'EmpStatus'}     {'Unknown'    }   58.807
    {'EmpStatus'}     {'Employed'   }   86.937
    {'EmpStatus'}     {'<missing>'  }      NaN
       ⋮
```

Notice that points for the `<missing>` bin for `'CustAge'` and `'ResStatus'` are explicitly shown (as `64.9836` and `74.1250`, respectively). The function computes these points from the WOE value for the `<missing>` bin and the logistic model coefficients.

For predictors that have no missing data in the training set, there is no explicit `<missing>` bin during the training of the model. By default, `displaypoints` reports the points as `NaN` for missing data resulting in a score of `NaN` when you use `score`. For these predictors, use the name-value pair argument `'Missing'` in `formatpoints` to indicate how missing data should be treated for scoring purposes.

Use `compactCreditScorecard` to convert the `creditscorecard` object into a `compactCreditScorecard` object. A `compactCreditScorecard` object is a lightweight version of a `creditscorecard` object that is used for deployment purposes.

```
csc = compactCreditScorecard(sc);
```

For the purpose of illustration, take a few rows from the original data as test data and introduce some missing data. Also introduce some invalid, or out-of-range, values. For numeric data, values below the minimum (or above the maximum) are considered invalid, such as a negative value for age (recall that in a previous step, you set `'MinValue'` to `0` for `'CustAge'` and `'CustIncome'`). For categorical data, invalid values are categories not explicitly included in the scorecard, for example, a residential status not previously mapped to scorecard categories, such as "House", or a meaningless string such as "abc123."

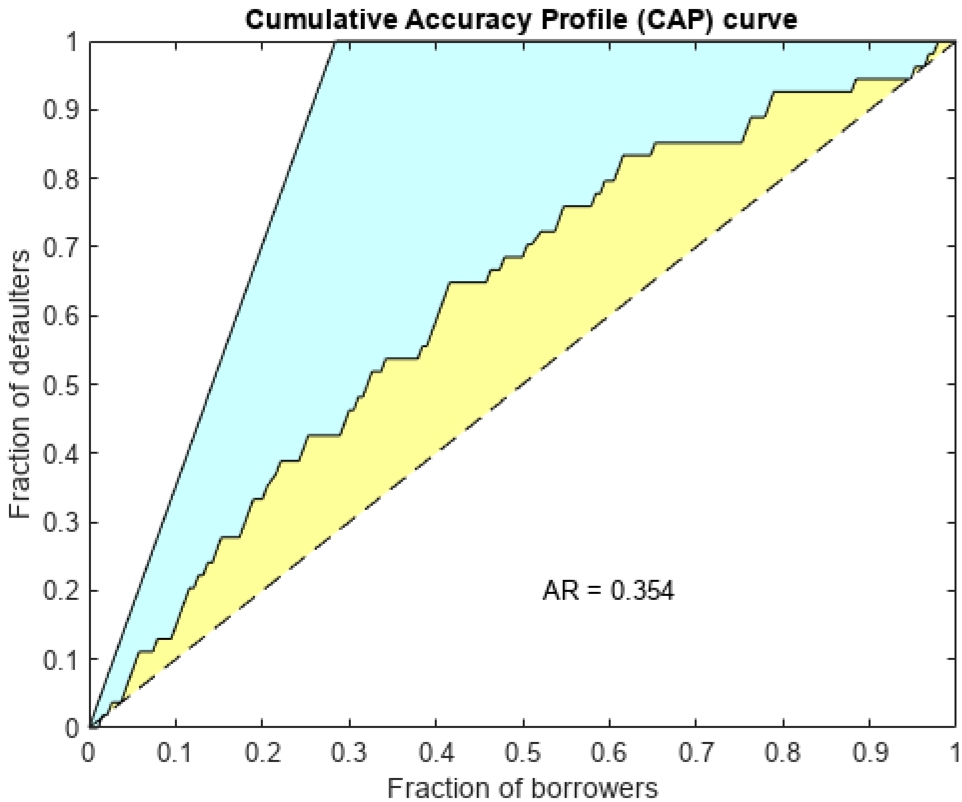
This example uses a very small validation data set only to illustrate the scoring of rows with missing and out-of-range values and the relationship between scoring and model validation.

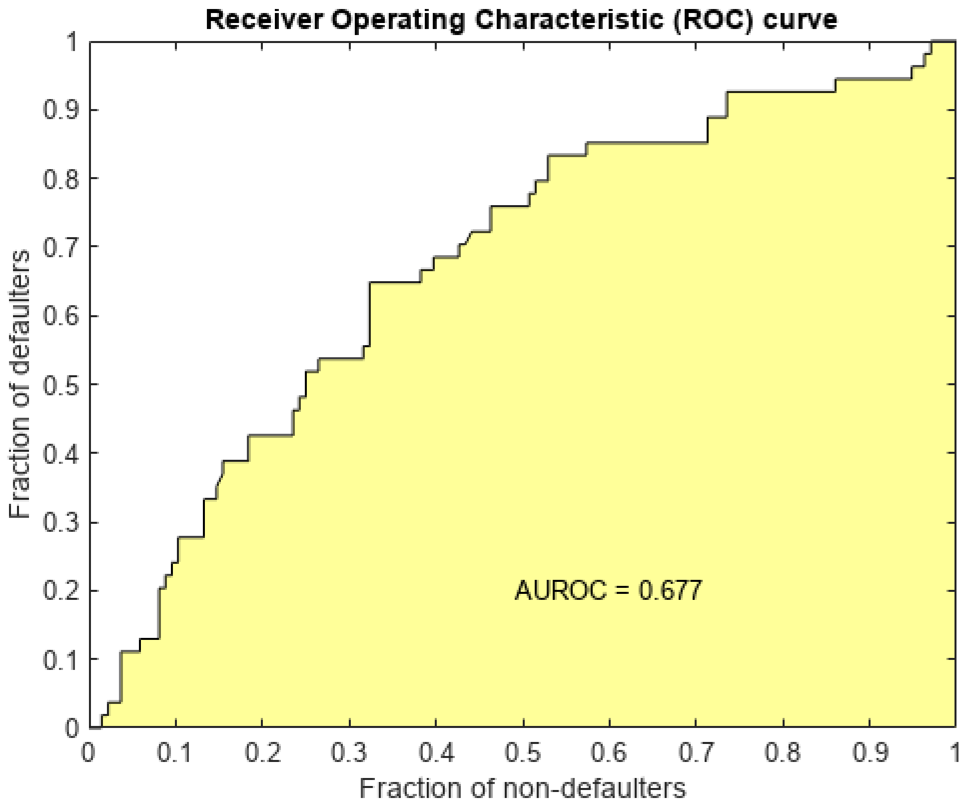
```
tdata = dataMissing(11:200,mdl.PredictorNames); % Keep only the predictors retained in the model
tdata.status = dataMissing.status(11:200); % Copy the response variable value, needed for valida
% Set some missing values
tdata.CustAge(1) = NaN;
tdata.ResStatus(2) = '<undefined>';
tdata.EmpStatus(3) = '<undefined>';
tdata.CustIncome(4) = NaN;
% Set some invalid values
tdata.CustAge(5) = -100;
tdata.ResStatus(6) = 'House';
tdata.EmpStatus(7) = 'Freelancer';
tdata.CustIncome(8) = -1;
disp(tdata(1:10,:))
```

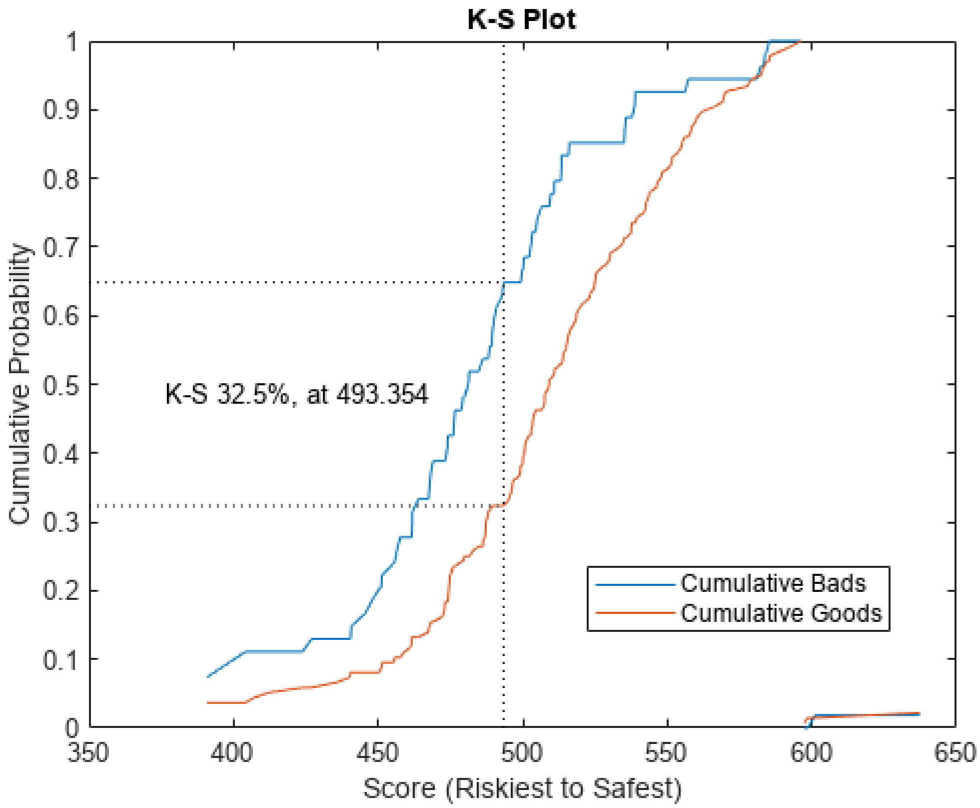
```
    CustAge     ResStatus      EmpStatus      CustIncome    TmWBank    OtherCC    AMBalance    sta
    _______    ___________    ___________    __________    _______    _______    _________    ___

      NaN      Tenant         Unknown          34000         44         Yes        119.8       1
       48      <undefined>    Unknown          44000         14         Yes       403.62       0
       65      Home Owner     <undefined>      48000          6         No        111.88       0
       44      Other          Unknown            NaN         35         No        436.41       0
     -100      Other          Employed         46000         16         Yes       162.21       0
       33      House          Employed         36000         36         Yes       845.02       0
       39      Tenant         Freelancer       34000         40         Yes       756.26       1
       24      Home Owner     Employed            -1         19         Yes       449.61       0
      NaN      Home Owner     Employed         51000         11         Yes       519.46       1
       52      Other          Unknown          42000         12         Yes       1269.2       0
```

Use `validatemodel` for a `compactCreditScorecard` object with the validation data set (`tdata`).

```
[ValStats,ValTable] = validatemodel(csc,tdata,'Plot',{'CAP','ROC','KS'});
```


Cumulative Accuracy Profile (CAP) curve
Fraction of defaulters
Fraction of borrowers
AR = 0.354
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Receiver Operating Characteristic (ROC) curve
Fraction of defaulters
Fraction of non-defaulters
AUROC = 0.677
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1




```
disp(ValStats)
```

```
             Measure                Value 
    _________________________      _______

    {'Accuracy Ratio'       }      0.35376
    {'Area under ROC curve'}       0.67688
    {'KS statistic'         }      0.32462
    {'KS score'             }       493.35
```

```
disp(ValTable(1:10,:))
```

```
    Scores    ProbDefault    TrueBads    FalseBads    TrueGoods    FalseGoods    Sensitivity
    ______    ___________    ________    _________    _________    __________    ___________

    597.33          NaN          0           1           135           54               0
    598.54          NaN          0           2           134           54               0
    601.18          NaN          1           2           134           53        0.018519
     637.3          NaN          1           3           133           53        0.018519
       NaN      0.69421          2           3           133           52        0.037037
       NaN      0.65394          2           4           132           52        0.037037
       NaN      0.64441          2           5           131           52        0.037037
       NaN      0.62799          3           5           131           51        0.055556
    390.86      0.58964          4           5           131           50        0.074074
    404.09      0.57902          6           5           131           48         0.11111
```

Input Arguments

`csc` — Compact credit scorecard model
`compactCreditScorecard` object

Compact credit scorecard model, specified as a `compactCreditScorecard` object.

To create a `compactCreditScorecard` object, use `compactCreditScorecard` or `compact` from Financial Toolbox.

`data` — Validation data
table

Validation data, specified as a MATLAB table, where each table row corresponds to individual observations. The `data` must contain columns for each of the predictors in the credit scorecard model. The columns of data can be any one of the following data types:

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

In addition, the table must contain a binary response variable and the name of this column must match the name of the `ResponseVar` property in the `compactCreditScorecard` object. (The `ResponseVar` property in the `compactCreditScorecard` is copied from the `ResponseVar` property of the original `creditscorecard` object.)

Note If a different validation data set is provided using the optional `data` input, observation weights for the validation data must be included in a column whose name matches `WeightsVar` from the original `creditscorecard` object, otherwise unit weights are used for the validation data. For more information, see “Using validatemodel with Weights”.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `csc = validatemodel(csc,data,'Plot','CAP')`

`Plot` — Type of plot
`'None'` (default) | character vector with values `'None'`, `'CAP'`, `'ROC'`,`'KS'` | cell array of character vectors with values `'None'`, `'CAP'`, `'ROC'`,`'KS'`

Type of plot, specified as the comma-separated pair consisting of `'Plot'` and a character vector with one of the following values:

- `'None'` — No plot is displayed.
- `'CAP'` — Cumulative Accuracy Profile. Plots the fraction of borrowers up to score “s” against the fraction of defaulters up to score “s” (`'PctObs'` against `'Sensitivity'` columns of `T` optional output argument). For details, see “Cumulative Accuracy Profile (CAP)”.
- `'ROC'` — Receiver Operating Characteristic. Plots the fraction of non-defaulters up to score “s” against the fraction of defaulters up to score “s” (`'FalseAlarm'` against `'Sensitivity'` columns of `T` optional output argument). For details, see “Receiver Operating Characteristic (ROC)”.
- `'KS'` — Kolmogorov-Smirnov. Plots each score “s” against the fraction of defaulters up to score “s,” and also against the fraction of nondefaulters up to score “s” (`'Scores'` against both `'Sensitivity'` and `'FalseAlarm'` columns of the optional output argument `T`). For details, see “Kolmogorov-Smirnov statistic (KS)”.

Tip For the Kolmogorov-Smirnov statistic option, you can enter either `'KS'` or `'K-S'`.

Data Types: `char` | `cell`

Output Arguments

`Stats` — Validation measures
table

Validation measures, returned as a `4`-by-`2` table. The first column, `'Measure'`, contains the names of the following measures:

- Accuracy ratio (AR)
- Area under the ROC curve (AUROC)
- The KS statistic
- KS score

The second column, `'Value'`, contains the values corresponding to these measures.

`T` — Validation statistics data
array

Validation statistics data, returned as an `N`-by-`9` table of validation statistics data, sorted by score from riskiest to safest. `N` is equal to the total number of unique scores, that is, scores without duplicates.

The table `T` contains the following nine columns, in this order:

- `'Scores'` — Scores sorted from riskiest to safest. The data in this row corresponds to all observations up to and including the score in this row.
- `'ProbDefault'` — Probability of default for observations in this row. For deciles, the average probability of default for all observations in the given decile is reported.
- `'TrueBads'` — Cumulative number of “bads” up to and including the corresponding score.

- 'FalseBads' — Cumulative number of “goods” up to and including the corresponding score.
- 'TrueGoods' — Cumulative number of “goods” above the corresponding score.
- 'FalseGoods' — Cumulative number of “bads” above the corresponding score.
- 'Sensitivity' — Fraction of defaulters (or the cumulative number of “bads” divided by total number of “bads”). This is the distribution of “bads” up to and including the corresponding score.
- 'FalseAlarm' — Fraction of nondefaulters (or the cumulative number of “goods” divided by total number of “goods”). This is the distribution of “goods” up to and including the corresponding score.
- 'PctObs' — Fraction of borrowers, or the cumulative number of observations, divided by total number of observations up to and including the corresponding score.

Note When creating the creditscorecard object with creditscorecard, if the optional name-value pair argument WeightsVar was used to specify observation (sample) weights, then the T table uses statistics, sums, and cumulative sums that are weighted counts.

hf — Handle to the plotted measures

figure handle

Figure handle to plotted measures, returned as a figure handle or array of handles. When Plot is set to 'None', hf is an empty array.

More About

Cumulative Accuracy Profile (CAP)

CAP is generally a concave curve and is also known as the Gini curve, Power curve, or Lorenz curve.

The scores of given observations are sorted from riskiest to safest. For a given fraction M (0% to 100%) of the total borrowers, the height of the CAP curve is the fraction of defaulters whose scores are less than or equal to the maximum score of the fraction M. This fraction of defaulters is also known as the “Sensitivity.”.

The area under the CAP curve, known as the AUCAP, is then compared to that of the perfect or “ideal” model, leading to the definition of a summary index known as the accuracy ratio (*AR*) or the Gini coefficient:

$$AR = \frac{A_R}{A_P}$$

where A_R is the area between the CAP curve and the diagonal, and A_P is the area between the perfect model and the diagonal. This represents a “random” model, where scores are assigned randomly and therefore the proportion of defaulters and nondefaulters is independent of the score. The perfect model is the model for which all defaulters are assigned the lowest scores, and therefore perfectly discriminates between defaulters and nondefaulters. Thus, the closer to unity *AR* is, the better the scoring model.

Receiver Operating Characteristic (ROC)

To find the receiver operating characteristic (ROC) curve, the proportion of defaulters up to a given score “s,” or “Sensitivity,” is computed.

This proportion is known as the true positive rate (TPR). Also, the proportion of nondefaulters up to score “s,“ or “False Alarm Rate,” is also computed. This proportion is also known as the false positive rate (FPR). The ROC curve is the plot of the “Sensitivity” vs. the “False Alarm Rate.” Computing the ROC curve is similar to computing the equivalent of a confusion matrix at each score level.

Similar to the CAP, the ROC has a summary statistic known as the area under the ROC curve (AUROC). The closer to unity, the better the scoring model. The accuracy ratio (*AR*) is related to the area under the curve by the following formula:

$$AR = 2(AUROC) - 1$$

Kolmogorov-Smirnov Statistic (KS)

The Kolmogorov-Smirnov (KS) plot, also known as the fish-eye graph, is a common statistic for measuring the predictive power of scorecards.

The KS plot shows the distribution of defaulters and the distribution of nondefaulters on the same plot. For the distribution of defaulters, each score “s” is plotted against the proportion of defaulters up to “s," or “Sensitivity." For the distribution of non-defaulters, each score “s” is plotted against the proportion of nondefaulters up to "s," or "False Alarm." The statistic of interest is called the KS statistic and is the maximum difference between these two distributions (“Sensitivity” minus “False Alarm”). The score at which this maximum is attained is also of interest.

Use `validatemodel` with Weights

If you provide observation weights, the `validatemodel` function incorporates the observation weights when calculating model validation statistics.

If you do not provide weights, the validation statistics are based on how many good and bad observations fall below a particular score. If you do provide weights, the weight (not the count) is accumulated for the good and the bad observations that fall below a particular score.

When you define observation weights using the optional `WeightsVar` name-value pair argument when creating a `creditscorecard` object, the weights stored in the `WeightsVar` column are used when validating the model on the training data. When a different validation data set is provided using the optional `data` input, observation weights for the validation data must be included in a column whose name matches `WeightsVar`. Otherwise, the unit weights are used for the validation data set.

The observation weights of the training data affect not only the validation statistics but also the credit scorecard scores themselves. For more information, see “Using fitmodel with Weights” and “Credit Scorecard Modeling Using Observation Weights”.

Version History

Introduced in R2019b

References

[1] *“Basel Committee on Banking Supervision: Studies on the Validation of Internal Rating Systems.”* Working Paper No. 14, February 2005.

[2] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

[3] Loeffler, G. and P. N. Posch. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

See Also

`compactCreditScorecard` | `probdefault` | `displaypoints` | `score`

Topics

"compactCreditScorecard Object Workflow" on page 3-57
"Case Study for Credit Scorecard Analysis"
"Credit Scorecard Modeling with Missing Values"
"Credit Scorecard Modeling Workflow"
"About Credit Scorecards"

portfolioECL

Compute the lifetime ECL at individual or portfolio level

Syntax

```
[totalECL,ECLByID,ECLByPeriod] = portfolioECL(MarginalPD,LGD,EAD)
[totalECL,ECLByID,ECLByPeriod] = portfolioECL( ___ ,Name=Value)
```

Description

`[totalECL,ECLByID,ECLByPeriod] = portfolioECL(MarginalPD,LGD,EAD)`, given the `MarginalPD`, `LGD`, and `EAD` values for a portfolio of loans, computes the lifetime expected credit loss (ECL) at the individual or portfolio level.

`[totalECL,ECLByID,ECLByPeriod] = portfolioECL( ___ ,Name=Value)` adds optional name-value pair arguments for `ScenarioProbabilities`, `InterestRate`, `Periodicity`, `IDVar`, and `ScenarioNames`.

Examples

Calculate ECL Based on Marginal PD, LGD, and EAD Predictions

This example shows how to calculate the expected credit loss (ECL) based on marginal probability of default (PD), loss given default (LGD), and exposure at default (EAD).

- Marginal PD — Expectation of a credit default event over a given time frame.
- LGD — Portion of a nonrecovered credit in the case of default.
- EAD — Balance at the time of default.

IFRS 9 requires multiple economic scenarios to be modeled while computing ECL. This example considers five macroeconomic scenarios: severe, adverse, baseline, favorable, and excellent.

Load Data

Load the credit data for company IDs `1304` and `2067` and the associated macroeconomic scenarios.

```
load DataPredictLifetime.mat
disp(LoanData)
```

```
     ID       ScoreGroup       YOB    Year
    ____    _____________    ___    ____

    1304    "Medium Risk"     4     2020
    1304    "Medium Risk"     5     2021
    1304    "Medium Risk"     6     2022
    1304    "Medium Risk"     7     2023
    1304    "Medium Risk"     8     2024
    1304    "Medium Risk"     9     2025
    1304    "Medium Risk"    10     2026
    2067    "Low Risk"        7     2020
```

```
    2067    "Low Risk"        8     2021
    2067    "Low Risk"        9     2022
    2067    "Low Risk"       10     2023
```

```
disp(head(MultipleScenarios,10))
```

```
    ScenarioID    Year    GDP     Market
    __________    ____    ____    ______

    "Severe"      2020    -0.9     -5.5 
    "Severe"      2021    -0.5     -6.5 
    "Severe"      2022     0.2       -1 
    "Severe"      2023     0.8      1.5 
    "Severe"      2024     1.4        4 
    "Severe"      2025     1.8      6.5 
    "Severe"      2026     1.8      6.5 
    "Severe"      2027     1.8      6.5 
    "Adverse"     2020     0.1     -0.5 
    "Adverse"     2021     0.2     -2.5 
```

```
disp(ScenarioProbabilities)
```

```
                 Probability
                 ___________

    Severe           0.1    
    Adverse          0.2    
    Baseline         0.3    
    Favorable        0.2    
    Excellent        0.2    
```

Load the `pdModel` that was created using `fitLifetimePDModel` with a `Probit` model.

```
load LifetimeChampionModel.mat
disp(pdModel)
```

```
  Probit with properties:

            ModelID: "Champion"
        Description: "A sample model used as champion model for illustration purposes."
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Define the interest rate to discount future losses back to present.

```
EffRate = 0.045;
```

Create Scenarios

Compute marginal lifetime PDs for the two companies.

```
CompanyID = 1304;
IndCompany = LoanData.ID == CompanyID;
Years = LoanData.Year(IndCompany);
NumYears = length(Years);
```

```
ScenarioID = unique(MultipleScenarios.ScenarioID,'stable');
NumScenarios = length(ScenarioID);

PD1 = zeros(NumYears,NumScenarios);
for ii=1:NumScenarios
   IndScenario = MultipleScenarios.ScenarioID==ScenarioID(ii);
   data = join(LoanData(IndCompany,:),MultipleScenarios(IndScenario,:));
   PD1(:,ii) = predictLifetime(pdModel,data,ProbabilityType="marginal");
end

DiscTimes = Years-Years(1)+1;
DiscFactors = 1./(1+EffRate).^DiscTimes;

ProbScenario = ScenarioProbabilities.Probability;


CompanyID = 2067;
IndCompany = LoanData.ID == CompanyID;
Years = LoanData.Year(IndCompany);
NumYears = length(Years);


PD4 = zeros(NumYears,NumScenarios);
for ii=1:NumScenarios
   IndScenario = MultipleScenarios.ScenarioID==ScenarioID(ii);
   data = join(LoanData(IndCompany,:),MultipleScenarios(IndScenario,:));
   PD4(:,ii) = predictLifetime(pdModel,data,ProbabilityType="marginal");
end
```

Calculate Marginal PD for Multiple IDs

Create a table for the portfolio PD that contains the PD for the two companies.

```
PD = array2table([PD1; PD4]);
PD.Properties.VariableNames = {'Severe','Adverse','Baseline','Favorable','Excellent'};
PD.ID = [repmat(1304,7,1);repmat(2067,4,1)];
PD = movevars(PD, 'ID', 'Before', 'Severe');
disp(PD)
```

```
     ID       Severe        Adverse       Baseline      Favorable     Excellent 
    ____    __________    __________    __________    __________    __________

    1304      0.011316     0.0096361     0.0081783      0.006918     0.0058324
    1304     0.0078277     0.0069482     0.0061554     0.0054425     0.0048028
    1304     0.0048869     0.0044693     0.0040823     0.0037243     0.0033938
    1304     0.0031017     0.0029321     0.0027698     0.0026147     0.0024668
    1304     0.0019309     0.0018923     0.0018538     0.0018153      0.001777
    1304     0.0012157     0.0012197     0.0012233     0.0012264     0.0012293
    1304    0.00082053    0.00082322    0.00082562    0.00082775    0.00082964
    2067     0.0022199      0.001832     0.0015067      0.001235     0.0010088
    2067     0.0014464     0.0012534     0.0010841    0.00093599    0.00080662
    2067     0.0008343    0.00074897    0.00067168    0.00060175    0.00053857
    2067    0.00049107    0.00045839    0.00042769    0.00039887    0.00037183
```

Calculate LGD for Multiple IDs

Create a table for the portfolio LGD that contains the LGD for the two companies.

```
LGD = array2table([0.25, 0.23, 0.21, 0.19, 0.17; 0.24, 0.22, 0.2, 0.18, 0.16]);
LGD.Properties.VariableNames = {'S1','S2','S3','S4','S5'};
LGD.ID = [1304;2067];
LGD = movevars(LGD, 'ID', 'Before', 'S1');
disp(LGD)
```

```
     ID      S1      S2      S3      S4      S5 
    ____    ____    ____    ____    ____    ____

    1304    0.25    0.23    0.21    0.19    0.17
    2067    0.24    0.22     0.2    0.18    0.16
```

Calculate EAD for Multiple IDs

Create a table for the portfolio EAD that contains the EAD for the two companies 1304 and 2067.

```
EAD = array2table(horzcat([repmat(1304,7,1);repmat(2067,4,1)],vertcat((100000:-10000:40000)',(12
EAD.Properties.VariableNames = {'ID','EAD'};
disp(EAD)
```

```
     ID       EAD   
    ____    _______

    1304      1e+05
    1304      90000
    1304      80000
    1304      70000
    1304      60000
    1304      50000
    1304      40000
    2067    1.2e+05
    2067    1.1e+05
    2067      1e+05
    2067      90000
```

Use portfolioECL with PD, LGD, and EAD Tables

Compute the lifetime ECL using `portfolioECL`.

```
[totalECL, ECLByID, ECLByPeriod] = portfolioECL(PD, LGD, EAD,ScenarioProbabilities=[0.1 0.2 0.3 0
InterestRate = EffRate, Periodicity="monthly",ScenarioNames={'Severe','Adverse','Baseline','Favor
```

Display the total portfolio ECL.

```
disp(totalECL);
```

```
  510.5860
```

Display the scenario weighted ECLs for each individual loan.

```
disp(ECLByID);
```

```
     ID      ECL  
    ____    ______

    1304    430.68
    2067    79.905
```

Display the ECL for each individual loan per time period and per scenario.

```
disp(ECLByPeriod);

     ID     TimePeriod    Severe    Adverse    Baseline    Favorable    Excellent
    ____    __________    ______    _______    ________    _________    _________

    1304        1         281.84     220.8       171.1       130.95       98.781 
    1304        2         174.81    142.76      115.47       92.372       72.935 
    1304        3         96.647    81.317      67.817       55.978        45.64 
    1304        4         53.474    46.505      40.111       34.259       28.918 
    1304        5         28.426     25.63      22.924       20.311        17.79 
    1304        6         14.859    13.715      12.559       11.393       10.217 
    1304        7         7.9931    7.3777      6.7558       6.1282       5.4957 
    2067        1         63.693    48.183      36.026       26.576       19.296 
    2067        2         37.901    30.106      23.673       18.394       14.091 
    2067        3           19.8    16.293      13.284       10.711       8.5209 
    2067        4         10.449    8.9412      7.5839       6.3656       5.2748 
```

Input Arguments

`MarginalPD` — Marginal PD values

table

Marginal PD values, specified as a table with a column for IDs that is defined by `IDVar`.

Note The `MarginalPD` table column name for IDs and the order of IDs must be the same as the ID columns of the `LGD` and `EAD` tables.

You can use `fitLifetimePDModel` to create a PD model and `predict` to create a vector that can be converted to a table using `array2table`.

Data Types: `table`

`LGD` — LGD values

table

LGD value, specified as a table with a column for IDs that is defined by `IDVar`.

Note The `LGD` table column name for lDs and the order of IDs must be the same as the ID columns of the `MarginalPD` and `EAD` tables.

You can use `fitLGDModel` to create a LGD model and `predict` to create a vector that can be converted to a table using `array2table`.

Data Types: `table`

`EAD` — EAD values

table

EAD value, specified as a table with a column for IDs that is defined by `IDVar`.

Note The `EAD` table column name for IDs and the order of IDs must be the same as the ID columns of the `MarginalPD` and `LGD` tables.

You can use `fitEADModel` to create a EAD model and `predict` to create a vector that can be converted to a table using `array2table`.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `[totalECL,ECLByID,ECLByPeriod] = portfolioECL(MarginalPD,LGD,EAD,InterestRate=0.045,Periodicity="annual")`

`ScenarioProbabilities` — Probabilities assigning weights to corresponding scenarios
equal weighted (default) | numeric vector

Probabilities assigning weights to corresponding scenarios, specified as `ScenarioProbabilities` and a numeric vector. The `ScenarioProbabilities` values must be greater than or equal to 0 and sum to 1.

Data Types: `double`

`InterestRate` — Interest rate to discount future losses back to present
`0` (default) | scalar positive or negative decimal | table

Interest rate to discount future losses back to present, specified as `InterestRate` and a scalar positive or negative decimal or a table.

- If you specify a scalar, the interest-rate value applies to the entire portfolio.
- If you specify a table, there must be exactly two columns in the interest-rate table, one for IDs and the other for the interest-rate value for each loan. Each row must have an ID that cannot be repeated on another row in the table. The IDs must match and be in the same order as the IDs used by `MarginalPD`, `LGD`, and `EAD` tables.

Data Types: `double` | `table`

`Periodicity` — Time period of input data
`"quarterly"` (default) | character vector with value of `'quarterly'`, `'monthly'`, `'semiannual'`, or `'annual'` | string with value of `"quarterly"`, `"monthly"`, `"semiannual"`, or `"annual"`

Time period of input data, specified as `Periodicity` and a character vector or string.

Data Types: `char` | `string`

`IDVar` — Column name for ID in `MarginalPD`, `LGD`, `EAD` tables
1st column in `MarginalPD`, `LGD`, `EAD` tables (default) | character vector | string

Column name for ID in `MarginalPD`, `LGD`, and `EAD` tables, specified as `IDVar` and a character vector or string.

Data Types: `char` | `string`

`ScenarioNames` — User-defined scenario names
`Scenario(n)` (where `n=1:numScenarios`) (default) | cell array of character vectors | string array

User-defined scenario names with one name per scenario, specified as `ScenarioNames` and a cell array of character vectors or string array. The `ScenarioNames` must all be unique and nonempty.

Data Types: `cell` | `string`

Output Arguments

`totalECL` — Total portfolio ECL
scalar

Total portfolio ECL, returned as a scalar. The total portfolio ECL is computed as a sum of the ECLs of each loan weighted by the scenario probabilities and discounted to the present.

`ECLByID` — Scenario weighted ECLs for each individual loan
table

Scenario weighted ECLs for each individual loan, returned as a table.

`ECLByPeriod` — ECL for each individual loan per time period and per scenario
table

ECL for each individual loan per time period and per scenario, returned as a table.

More About

Expected Credit Losses

The expected credit losses (ECLs) model adopts a forward-looking approach to estimation of impairment losses.

- The discounted ECL at time t for scenario s is defined as

  $$ECL_i(t;s) = PD_{\mathrm{marginal},\,i}(t;s)LGD_i(t;s)EAD_i(t;s)Disc_i(t)$$

  where

  t denotes a time period.

  s denotes a scenario.

  i denotes a loan.

  $PD_{marginal,i}(t;s)$ is the marginal probability of default (PD) (see `predictLifetime`) for loan i at time period t, given scenario s.

  $LGD_i(t;s)$ is the loss given default (LGD) for loan i at time period t, given scenario s.

  $EAD_i(t;s)$ is the exposure at default (EAD) for loan i at time period t, given scenario s.

  $Disc_i(t)$ is the discount factor for loan i at time period t, based on the loan's effective interest rate.

  The $ECL_i(t;s)$ quantities are computed for each time period in the remaining life of a loan and for each scenario. These quantities are reported in the `ECLByPeriod` output of `portfolioECL` for all loans in the portfolio.

- The lifetime ECL for loan *i* is computed as

  $$ECL_i = \sum_{s=1}^{M} \sum_{t=1}^{N_i} ECL_i(t;s) * P(s)$$

  where

  N_i is the number of periods in the remaining life of loan *i*.

  M is the number of scenarios.

  $P(s)$ denotes the scenario probabilities.

  The ECL_i quantity is reported in the `ECLByID` output of `portfolioECL` for all loans in the portfolio.
- The total portfolio lifetime ECL is

  $$ECL = \sum_{i=1}^{L} ECL_i$$

  where

  `L` is the number of loans in the portfolio.

  The total ECL value for the portfolio is reported in the `totalECL` output of the `portfolioECL` function.

To compute an ECL spanning only 1-year ahead (as opposed to a lifetime ECL), the inputs to `portfolioECL` must only include time periods within the 1-year period of interest. For more information, see "Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations" on page 4-195.

Version History

Introduced in R2022a

See Also

`fitLifetimePDModel` | `fitLGDModel` | `fitEADModel`

Topics

"Expected Credit Loss Computation" on page 4-124
"Modeling Probabilities of Default with Cox Proportional Hazards" on page 4-28
"Incorporate Macroeconomic Scenario Projections in Loan Portfolio ECL Calculations" on page 4-195

view

Display `developmentTriangle` object

Syntax

`developmentTriangleTable = view(developmentTriangle)`

Description

`developmentTriangleTable = view(developmentTriangle)` displays a `developmentTriangle` object in table form. Each row represents an origin period and each column represents a development period.

Examples

Display developmentTriangle Object in Table Form

Display a `developmentTriangle` object using simulated insurance claims data in table form.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

Use the `view` function to display the `developmentTriangle` contents in table form. In the table, each row represents an origin period and each column represents a development period.

```
developmentTriangleTable = view(dT)
```

```
developmentTriangleTable=10×10 table
             12        24        36        48        60        72        84        96       108
           ______    ______    ______    ______    ______    ______    ______    ______    _____

    2010   3995.7      4635    4866.8    4964.1    5013.7    5038.8      5059    5074.1    5084
    2011     3968    4682.3    4963.2    5062.5    5113.1    5138.7    5154.1    5169.6    5179
    2012     4217    5060.4      5364    5508.9    5558.4    5586.2    5608.6    5625.4      Na
    2013   4374.2    5205.3    5517.7    5661.1    5740.4    5780.6    5803.7       NaN      Na
    2014   4499.7    5309.6    5628.2    5785.8    5849.4    5878.7       NaN       NaN      Na
    2015   4530.2    5300.4    5565.4    5715.7    5772.8       NaN       NaN       NaN      Na
    2016   4572.6    5304.2    5569.5    5714.3       NaN       NaN       NaN       NaN      Na
    2017   4680.6    5523.1    5854.4       NaN       NaN       NaN       NaN       NaN      Na
    2018   4696.7    5495.1       NaN       NaN       NaN       NaN       NaN       NaN      Na
    2019   4945.9       NaN       NaN       NaN       NaN       NaN       NaN       NaN      Na
```

Input Arguments

`developmentTriangle` — Development triangle

object

Development triangle, specified as a previously created `developmentTriangle` object.

Data Types: `object`

Output Arguments

`developmentTriangleTable` — Development triangle in table form

table

Development triangle in table form, returned as a table. In the table, each row represents an origin period and each column represents a development period.

Version History

Introduced in R2020b

See Also

`linkRatios` | `linkRatioAverages` | `cdfSummary` | `ultimateClaims` | `fullTriangle` | `linkRatiosPlot` | `claimsPlot`

Topics

"Mean Square Error of Prediction for Estimated Ultimate Claims" on page 4-161
"Bootstrap Using Chain Ladder Method" on page 4-168
"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

Threshold Predictors

Select thresholds for predictor risk metrics in the Live Editor

Description

The **Threshold Predictors** task lets you interactively set credit scorecard predictor thresholds for one or more risk metrics computed for a set of predictors, or features. Risk metric thresholds are part of the feature selection process before building a credit scorecard. The task automatically generates MATLAB code for your live script.

Using this task, you can:

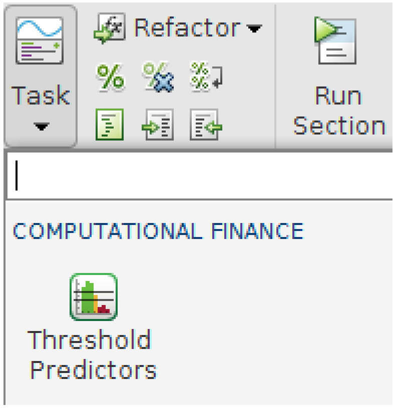
- Select risk metrics from the columns of a table of risk metric data.
- Specify thresholds for the risk metrics, separating the rows of predictors into color-coded **Pass**, **Fail**, or **Undecided** regions.
- Visualize the labeled and color-coded risk metric values for each thresholded metric.

For general information about Live Editor tasks, see “Add Interactive Tasks to a Live Script”.

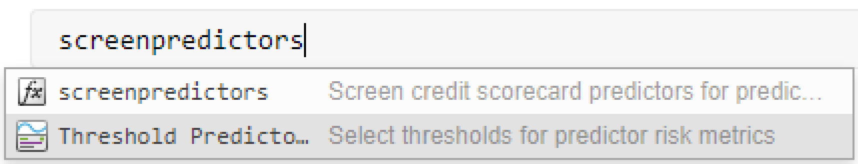
Open the Task

To add the **Threshold Predictors** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task > Threshold Predictors**.



- In a code block in the script, type a relevant keyword, such as `screenpredictors`. Select **Threshold Predictors** from the suggested command completions.



Parameters

`Predictor metrics` — Table of risk metrics
table of risk metrics calculated for a set of predictors

The **Predictor metrics** table must be a numeric MATLAB table. The columns of the **Predictor metrics** table contain the values for a particular risk metric (for example, information value or accuracy ratio) for a set of model predictors. The rows of the table contain the values of each risk metric for a particular predictor. The **Predictor metrics** table must have defined row names.

Typically, you create the **Predictor metrics** table using the `screenpredictors` function. `screenpredictors` takes a `creditscorecard` input data set and calculates the risk metrics table.

Example: `metric_table = screenpredictors(data,'IDVar',idvar,'ResponseVar',responsevar)`

`Select thresholded metrics` — List of metrics with defined thresholds
list box containing metrics with thresholds

The **Select thresholded metrics** list shows which metrics have thresholds specified. The **Select thresholded metrics** drop-down box contains the risk metrics defined in the columns of the **Predictor metrics** table.

To specify a threshold:

1 Select a risk metric from the **Select thresholded metrics** drop-down box and click the plus + button. The metric is added to the **Select thresholded metrics** list box and the **Predictors** plot displays with a default threshold and region labels.

2 To adjust a threshold, drag the associated threshold line or use the **Thresholds and Labels** spinner controls.

3 Set additional thresholds by clicking the **Predictors** plot at the desired value.

4 Select a different metric from the **Select thresholded metrics** list box. The **Predictors** plot updates to show the associated metric bar chart with its overlayed threshold lines.

To remove a threshold:

- Select the threshold line or the associated **Thresholds and Labels** spinner and click the line delete button on the **Predictors** plot. You can remove *all* thresholds for the selected risk metric by clicking the minus button next to the **Select thresholded metrics** list box.

When using a **Predictor metrics** table that is created using the `screenpredictors` function, you can set thresholds for any of the following metrics:

- **InfoValue**
- **Entropy**
- **Accuracy Ratio**
- **AUROC**
- **Gini**
- **Chi2PValue**
- **PercentMissing**

For more information on the metrics for `screenpredictors`, see “metric_table” on page 6-0 .

`Thresholds and Labels` — Thresholds and region labels
drop-down box and spinners specifying labeled regions

The **Thresholds and Labels** controls are composed of spinners for each specified threshold of the currently selected risk metric and drop-down boxes that set the labels for the surrounding regions to **Pass**, **Fail**, or **Undecided**.

The **Thresholds and Labels** spinners are sorted in descending order from top to bottom. The region labels can be set to **Pass**, **Fail**, or **Undecided** where the region label defines the label for all metric values that lie on a particular side of a threshold.

`Display results` — Display table of labeled metrics
check box to toggle display of label table

Check the **Display label table** check box to display the current set of labeled metric values. The label table contains the columns from the **Predictor metrics** table for which there are specified thresholds. The entries in the label table are categorical labels (**Pass**, **Fail**, or **Undecided**) based on which region each metric value is found.

Tips

- To sort the predictors in the **Predictors** plot, click **Sort**.To revert to the original sort order, click **Revert**.
- Each time you add a new threshold by clicking the **Predictors** plot, a new set of controls is added to the **Thresholds and Labels** section. Use the spinner to fine tune the threshold value. Use the

label drop-down box to set the appropriate label (**Pass**, **Fail**, or **Undecided**) for the newly defined region of metric values.

Version History

Introduced in R2021b

R2022b: Support for non-standard column names in Predictor metrics table
Behavior changed in R2022b

The **Predictor metrics** table supports non-standard MATLAB variable names containing spaces or Unicode characters.

See Also

Functions
`screenpredictors`

Topics
"Feature Screening with screenpredictors" on page 3-64

bornhuetterFerguson

Create `bornhuetterFerguson` object

Description

Use this workflow to generate unpaid claims for a `bornhuetterFerguson`:

1. Load or generate the data for the Bornhuetter-Ferguson technique.
2. Create a `developmentTriangle` object.
3. Create an `expectedClaims` object.
4. Create a `bornhuetterFerguson` object.
5. Use the `ultimateClaims` function to calculate the ultimate claims.
6. Use the `ibnr` function to calculate the incurred-but-not-reported (IBNR) claims.
7. Use the `unpaidClaims` function to calculate the unpaid claims.
8. Use the `summary` function to generate a summary report for the Bornhuetter-Ferguson technique.

Creation

Syntax

```
bf = bornhuetterFerguson(dT_reported,dT_paid,expectedClaims)
```

Description

`bf = bornhuetterFerguson(dT_reported,dT_paid,expectedClaims)` creates a `bornhuetterFerguson` object using the `developmentTriangle` objects for reported claims (`dT_reported`) and paid claims (`dT_paid`) and the `expectedClaims`.

Input Arguments

`dT_reported` — Development triangle for reported claims
`developmentTriangle` object

Development triangle for reported claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

`dT_paid` — Development triangle for paid claims
`developmentTriangle` object

Development triangle for paid claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

expectedClaims — Expected claims estimates for each Origin period
array

Expected claims estimates for each Origin period, specified as an array.

Data Types: double

Properties

ReportedTriangle — Development triangle for reported claims
developmentTriangle object

Development triangle for reported claims, returned as a developmentTriangle object containing the origin years, development years, and claims.

Data Types: object

PaidTriangle — Development triangle for paid claims
developmentTriangle object

Development triangle for paid claims, returned as a developmentTriangle object containing the origin years, development years, and claims.

Data Types: object

expectedClaims — Expected claims estimates for each Origin period
array

Expected claims estimates for each Origin period, returned as an array.

Data Types: double

Object Functions

ultimateClaims	Compute projected ultimate claims for bornhuetterFerguson object
ibnr	Compute IBNR claims for bornhuetterFerguson object
unpaidClaims	Compute unpaid claims estimates for bornhuetterFerguson object
summary	Display summary report for Bornhuetter-Ferguson analysis

Examples

Create bornhuetterFerguson Object

Create a bornhuetterFerguson object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
```

```
    2010        60        5013.7      4711.4
    2010        72        5038.8      4805.6
    2010        84          5059      4853.7
    2010        96        5074.1      4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                      Origin: {10x1 cell}
                 Development: {10x1 cell}
                      Claims: [10x10 double]
              LatestDiagonal: [10x1 double]
                 Description: ""
                  TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
           SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                      Origin: {10x1 cell}
                 Development: {10x1 cell}
                      Claims: [10x10 double]
              LatestDiagonal: [10x1 double]
                 Description: ""
                  TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
           SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

          ReportedTriangle: [1x1 developmentTriangle]
              PaidTriangle: [1x1 developmentTriangle]
             EarnedPremium: [10x1 double]
             InitialClaims: [10x1 double]
           CaseOutstanding: [10x1 double]
     EstimatedClaimsRatios: [10x1 double]
      SelectedClaimsRatios: [10x1 double]
```

Create a `bornhuetterFerguson` object with reported claims, paid claims, and expected claims to calculate ultimate claims, case outstanding, IBNR claims, and unpaid claims estimates.

```
bf = bornhuetterFerguson(dT_reported, dT_paid, ec.InitialClaims)
```

```
bf = 
  bornhuetterFerguson with properties:

     ReportedTriangle: [1x1 developmentTriangle]
         PaidTriangle: [1x1 developmentTriangle]
       ExpectedClaims: [10x1 double]
    PercentUnreported: [10x1 double]
        PercentUnpaid: [10x1 double]
      CaseOutstanding: [10x1 double]
```

Version History

Introduced in R2020b

See Also

developmentTriangle | chainLadder | expectedClaims

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

capeCod

Create `capeCod` object

Description

Use this workflow to generate unpaid claims for a `capeCod` object:

1. Load or generate the data for the development triangle.
2. Create two `developmentTriangle` objects — one for the reported development triangle and one for the paid development triangle.
3. Create a `capeCod` object.
4. Use the `ibnr` function to calculate the incurred-but-not-reported (IBNR) claims.
5. Use the `ultimateClaims` function to calculate the ultimate claims.
6. Use the `unpaidClaims` function to calculate the unpaid claims.
7. Use the `summary` function to display the chain ladder summary report.

Creation

Syntax

```
cc = capeCod(dT_reported,dT_paid,earnedPremium)
```

Description

`cc = capeCod(dT_reported,dT_paid,earnedPremium)` creates a `capeCod` object using the `developmentTriangle` objects for reported claims (`dT_reported`) and paid claims (`dT_paid`) and the `earnedPremium`.

Input Arguments

`dT_reported` — Development triangle for reported claims
`developmentTriangle` object

Development triangle for reported claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

`dT_paid` — Development triangle for paid claims
`developmentTriangle` object

Development triangle for paid claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

earnedPremium — Earned premium
vector

Earned premium, specified as a vector.

Data Types: double

Properties

ReportedTriangle — Development triangle for reported claims
developmentTriangle object

Development triangle for reported claims, returned as a developmentTriangle object containing the origin years, development years, and claims.

Data Types: object

PaidTriangle — Development triangle for paid claims
developmentTriangle object

Development triangle for paid claims, returned as a developmentTriangle object containing the origin years, development years, and claims.

Data Types: object

EarnedPremium — Earned premium
vector

Earned premium, returned as a vector.

Data Types: double

UsedUpPremium — Used up premium
vector

This property is read-only.

Used up premium, calculated by multiplying the initial claims with the percent of ultimate claims that are reported, returned as a vector.

Data Types: double

EstimatedClaimsRatio — Estimated claims ratio
vector

This property is read-only.

Estimated claims ratio, calculated by dividing the initial claims by the used up premium, returned as a vector.

Data Types: double

ExpectedClaimRatio — Expected claim ration
vector

This property is read-only.

Expected claim ratio, weighted average claim ratio from all the time periods, returned as a vector.

Data Types: double

EstimatedExpectedClaims — Estimated expected claims
vector

This property is read-only.

Estimated expected claims, that is, the earned premium multiplied by the expected claim ratio, returned as a vector.

Data Types: double

PercentUnreported — Percentage of unreported claims
vector

This property is read-only.

Percentage of unreported claims, returned as a vector.

Data Types: double

CaseOutstanding — Difference of the latest diagonals of the reported and paid development triangles
vector

This property is read-only.

Difference of the latest diagonals of the reported and paid development triangles, returned as a vector.

Data Types: double

Object Functions

ibnr	Compute IBNR claims for capeCod object
unpaidClaims	Compute unpaid claims estimates for capeCod object
ultimateClaims	Compute projected ultimate claims for capeCod object
summary	Display summary report for Cape Cod analysis

Examples

Create capeCod Object

Create a capeCod object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
```

```
      2010              72                 5038.8          4805.6
      2010              84                   5059          4853.7
      2010              96                 5074.1          4877.9
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
```

Create a `capeCod` object where the first input argument is the reported development triangle, the second input argument is the paid development triangle, and the third argument is the earned premium.

```
cc = capeCod(dT_reported, dT_paid, earnedPremium)
```

```
cc = 
  capeCod with properties:

            ReportedTriangle: [1x1 developmentTriangle]
                PaidTriangle: [1x1 developmentTriangle]
               EarnedPremium: [10x1 double]
               UsedUpPremium: [10x1 double]
        EstimatedClaimRatios: [10x1 double]
          ExpectedClaimRatio: 0.4258
    EstimatedExpectedClaims: [10x1 double]
           PercentUnreported: [10x1 double]
             CaseOutstanding: [10x1 double]
```

Version History

Introduced in R2021a

See Also

`developmentTriangle` | `expectedClaims` | `bornhuetterFerguson`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

chainLadder

Create `chainLadder` object

Description

Use this workflow to generate unpaid claims for a `chainLadder`:

1. Load or generate the data for the development triangle.
2. Create two `developmentTriangle` objects — one for the reported development triangle and one for the paid development triangle.
3. Create a `chainLadder` object.
4. Use the `ibnr` function to calculate the incurred-but-not-reported (IBNR) claims.
5. Use the `unpaidClaims` function to calculate the unpaid claims.
6. Use the `summary` function to display the chain ladder summary report.

Creation

Syntax

```
cl = chainladder(dT_reported,dT_paid)
```

Description

`cl = chainladder(dT_reported,dT_paid)` creates a `chainLadder` object using the `developmentTriangle` objects for reported claims (`dT_reported`) and paid claims (`dT_paid`).

Input Arguments

`dT_reported` — Development triangle for reported claims
`developmentTriangle` object

Development triangle for reported claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

`dT_paid` — Development triangle for paid claims
`developmentTriangle` object

Development triangle for paid claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

Properties

`ReportedTriangle` — Development triangle for reported claims
`developmentTriangle` object

Development triangle for reported claims, returned as a `developmentTriangle` object containing the origin years, development years, and claims.

Data Types: `object`

`PaidTriangle` — Development triangle for paid claims
`developmentTriangle` object

Development triangle for paid claims, returned as a `developmentTriangle` object containing the origin years, development years, and claims.

Data Types: `object`

`CaseOutstanding` — Difference of the latest diagonals of the reported and paid development triangles
vector

Difference of the latest diagonals of the reported and paid development triangles, returned as a vector.

Data Types: `double`

Object Functions

ibnr	Compute IBNR claims for chainLadder object
unpaidClaims	Compute unpaid claims for chainLadder object
summary	Display summary report for different claims estimates

Examples

Create `chainLadder` Object

Create a `chainLadder` object containing simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data. Create two `developmentTriangle` objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create a `chainLadder` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
cl = chainLadder(dT_reported, dT_paid)
```

```
cl = 
  chainLadder with properties:

    ReportedTriangle: [1x1 developmentTriangle]
        PaidTriangle: [1x1 developmentTriangle]
     CaseOutstanding: [10x1 double]
```

Version History

Introduced in R2020b

See Also

`developmentTriangle` | `expectedClaims` | `bornhuetterFerguson`

Topics

"Overview of Claims Estimation Methods for Non-Life Insurance" on page 1-16

compactCreditScorecard

Create `compactCreditScorecard` object for a credit scorecard model

Description

Build a compact credit scorecard model by creating a `compactCreditScorecard` object from an existing `creditscorecard` object.

After creating a `compactCreditScorecard` object, you can use the associated object functions to display points (`displaypoints`), calculate the probability of default (`probdefault`), or compute scores (`score`).

Note You cannot directly modify a `compactCreditScorecard` object. To change a `compactCreditScorecard` object, you must modify the existing `creditscorecard` object that you used to create the `compactCreditScorecard` object. You must then use `compactCreditScorecard` to create a new `compactCreditScorecard` object.

Creation

Syntax

```
csc = compactCreditScorecard(sc)
```

Description

`csc = compactCreditScorecard(sc)` creates a `compactCreditScorecard` object from an existing `creditscorecard`. You can then use the `compactCreditScorecard` object with the `displaypoints`, `score`, and `probdefault` functions.

Note You cannot use a `compactCreditScorecard` object with the **Binning Explorer** app.

Input Arguments

`sc` — `creditscorecard` object
object

`creditscorecard` object, specified using an existing `creditscorecard` object.

Note To use a `creditscorecard` object for input, you must first process the object using the `autobinning` and `fitmodel` functions. Optionally, you can also use `formatpoints` for processing.

Data Types: `object`

Properties

`PredictorVars` — Names of predictor variables
cell array of character vectors

Names of the predictor variables used in the input `creditscorecard` object, returned as a cell array of character vectors. The `PredictorVars` property includes only the predictor variable names in the fitted `creditscorecard` object.

Data Types: `cell`

`NumericPredictors` — Numeric predictors
cell array of character vectors

Numeric predictors in the input `creditscorecard` object, returned as a cell array of character vectors. The `NumericPredictors` property includes only the numeric predictors in the fitted `creditscorecard` object.

Data Types: `cell`

`CategoricalPredictors` — Names of categorical predictors
cell array of character vectors

Names of categorical predictors used in the input `creditscorecard` object, returned as a cell array of character vectors. The `CategoricalPredictors` property includes only the categorical predictors in the fitted `creditscorecard` object.

Data Types: `cell`

`Description` — User-defined description
character vector | string

User-defined description, returned as a character vector or string.

Data Types: `char` | `string`

Object Functions

displaypoints	Return points per predictor per bin for a compactCreditScorecard object
score	Compute credit scores for given dataset for a compactCreditScorecard object
probdefault	Likelihood of default for given dataset for a compactCreditScorecard object
validatemodel	Validate quality of compact credit scorecard model

Examples

Create `compactCreditScorecard` Object

To create a `compactCreditScorecard` object, first create a `creditscorecard` object using the `CreditCardData.mat` file to load the `data` (using a dataset from Refaat 2011).

```
load CreditCardData.mat
sc = creditscorecard(data)
```

```
sc = 
  creditscorecard with properties:
```

```
                GoodLabel: 0
              ResponseVar: 'status'
               WeightsVar: ''
                 VarNames: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
        NumericPredictors: {'CustID'  'CustAge'  'TmAtAddress'  'CustIncome'  'TmWBank'  'AMBalan
    CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
           BinMissingData: 0
                    IDVar: ''
            PredictorVars: {'CustID'  'CustAge'  'TmAtAddress'  'ResStatus'  'EmpStatus'  'CustIn
                     Data: [1200x11 table]
```

Before creating a `compactCreditScorecard` object, you must use `autobinning` and `fitmodel` with the `creditscorecard` object.

```
sc = autobinning(sc);
sc = fitmodel(sc);
```

```
1. Adding CustIncome, Deviance = 1490.8527, Chi2Stat = 32.588614, PValue = 1.1387992e-08
2. Adding TmWBank, Deviance = 1467.1415, Chi2Stat = 23.711203, PValue = 1.1192909e-06
3. Adding AMBalance, Deviance = 1455.5715, Chi2Stat = 11.569967, PValue = 0.00067025601
4. Adding EmpStatus, Deviance = 1447.3451, Chi2Stat = 8.2264038, PValue = 0.0041285257
5. Adding CustAge, Deviance = 1441.994, Chi2Stat = 5.3511754, PValue = 0.020708306
6. Adding ResStatus, Deviance = 1437.8756, Chi2Stat = 4.118404, PValue = 0.042419078
7. Adding OtherCC, Deviance = 1433.707, Chi2Stat = 4.1686018, PValue = 0.041179769

Generalized linear regression model:
    logit(status) ~ 1 + CustAge + ResStatus + EmpStatus + CustIncome + TmWBank + OtherCC + AMBala
    Distribution = Binomial

Estimated Coefficients:
                   Estimate       SE       tStat       pValue  
                   ________    ________    ______    __________

    (Intercept)    0.70239     0.064001    10.975    5.0538e-28
    CustAge        0.60833      0.24932      2.44      0.014687
    ResStatus        1.377      0.65272    2.1097      0.034888
    EmpStatus      0.88565        0.293    3.0227     0.0025055
    CustIncome     0.70164      0.21844    3.2121     0.0013179
    TmWBank         1.1074      0.23271    4.7589    1.9464e-06
    OtherCC         1.0883      0.52912    2.0569      0.039696
    AMBalance        1.045      0.32214    3.2439     0.0011792


1200 observations, 1192 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 89.7, p-value = 1.4e-16
```

Use the `creditscorecard` object with `compactCreditScorecard` to create a `compactCreditScorecard` object.

```
csc = compactCreditScorecard(sc)
```

```
csc = 
  compactCreditScorecard with properties:

              Description: ''
```

```
              GoodLabel: 0
            ResponseVar: 'status'
             WeightsVar: ''
      NumericPredictors: {'CustAge'  'CustIncome'  'TmWBank'  'AMBalance'}
  CategoricalPredictors: {'ResStatus'  'EmpStatus'  'OtherCC'}
         PredictorVars: {'CustAge'  'ResStatus'  'EmpStatus'  'CustIncome'  'TmWBank'  'OtherC
```

You can then use `displaypoints`, `score`, and `probdefault` with the `compactCreditScorecard` object.

Version History

Introduced in R2019a

References

[1] Anderson, R. *The Credit Scoring Toolkit.* Oxford University Press, 2007.

[2] Refaat, M. *Data Preparation for Data Mining Using SAS.* Morgan Kaufmann, 2006.

[3] Refaat, M. *Credit Risk Scorecards: Development and Implementation Using SAS.* lulu.com, 2011.

See Also

Functions
`displaypoints` | `score` | `probdefault` | `validatemodel`

Apps
Binning Explorer

Topics
"compactCreditScorecard Object Workflow" on page 3-57
"Case Study for Credit Scorecard Analysis"
"Credit Scorecard Modeling Workflow"
"About Credit Scorecards"

External Websites
Credit Risk Modeling with MATLAB (53 min 10 sec)

creditDefaultCopula

Create `creditDefaultCopula` object to simulate and analyze multifactor credit default model

Description

The `creditDefaultCopula` class simulates portfolio losses due to counterparty defaults using a multifactor model. `creditDefaultCopula` associates each counterparty with a random variable, called a latent variable, which is mapped to default/non-default outcomes for each scenario such that defaults occur with probability `PD`. In the event of default, a loss for that scenario is recorded equal to `EAD` * `LGD` for the counterparty. These latent variables are simulated using a multi-factor model, where systemic credit fluctuations are modeled with a series of risk factors. These factors can be based on industry sectors (such as financial, aerospace), geographical regions (such as USA, Eurozone), or any other underlying driver of credit risk. Each counterparty is assigned a series of weights which determine their sensitivity to each underlying credit factors.

The inputs to the model describe the credit-sensitive portfolio of exposures:

- `EAD` — Exposure at default
- `PD` — Probability of default
- `LGD` — Loss given default (1 âˆ’ *Recovery*)
- `Weights` — Factor and idiosyncratic model weights

After the `creditDefaultCopula` object is created (see “Create creditDefaultCopula” on page 6-528 and “Properties” on page 6-531), use the `simulate` function to simulate credit defaults using the multifactor model. The results are stored in the form of a distribution of losses at the portfolio and counterparty level. Several risk measures at the portfolio level are calculated, and the risk contributions from individual obligors. The model calculates:

- Full simulated distribution of portfolio losses across scenarios
- Losses on each counterparty across scenarios
- Several risk measures (`VaR`, `CVaR`, `EL`, `Std`) with confidence intervals
- Risk contributions per counterparty (for `EL` and `CVaR`)

Creation

Syntax

```
cdc = creditDefaultCopula(EAD,PD,LGD,Weights)
cdc = creditDefaultCopula( ___ ,Name,Value)
```

Description

`cdc = creditDefaultCopula(EAD,PD,LGD,Weights)` creates a `creditDefaultCopula` object. The `creditDefaultCopula` object has the following properties:

- Portfolio on page 6-0 :

  A table with the following variables (each row of the table represents one counterparty):

  - `ID` — ID to identify each counterparty
  - `EAD` — Exposure at default
  - `PD` — Probability of default
  - `LGD` — Loss given default
  - `Weights` — Factor and idiosyncratic weights for counterparties
- FactorCorrelation on page 6-0 :

  Factor correlation matrix, a `NumFactors`-by-`NumFactors` matrix that defines the correlation between the risk factors.
- VaRLevel on page 6-0 :

  The value-at-risk level, used when reporting VaR and CVaR.
- PortfolioLosses on page 6-0

  Portfolio losses, a `NumScenarios`-by-`1` vector of portfolio losses. This property is empty until the `simulate` function is used.

`cdc = creditDefaultCopula( ___ ,Name,Value)` sets Properties on page 6-531 using name-value pairs and any of the arguments in the previous syntax. For example, `cdc = creditDefaultCopula(EAD,PD,LGD,Weights,'VaRLevel',0.99)`. You can specify multiple name-value pairs as optional name-value pair arguments.

Input Arguments

EAD — Exposure at default
numeric vector

Exposure at default, specified as a `NumCounterparties`-by-`1` vector of credit exposures. The `EAD` input sets the Portfolio on page 6-0 property.

Note The `creditDefaultCopula` model simulates defaults and losses over some fixed time period (for example, one year). The counterparty exposures (`EAD`) and default probabilities (`PD`) must both be specific to a particular time.

Data Types: `double`

PD — Probability of default
numeric vector with elements from `0` through `1`

Probability of default, specified as a `NumCounterparties`-by-`1` numeric vector with elements from `0` through `1`, representing the default probabilities for the counterparties. The `PD` input sets the Portfolio on page 6-0 property.

Note The `creditDefaultCopula` model simulates defaults and losses over a fixed time period (for example, one year). The counterparty exposures (`EAD`) and default probabilities (`PD`) must both be specific to a particular time.

Data Types: `double`

LGD — Loss given default
numeric vector with elements from `0` through `1`

Loss given default, specified as a `NumCounterparties`-by-`1` numeric vector with elements from `0` through `1`, representing the fraction of exposure that is lost when a counterparty defaults. `LGD` is defined as (1 âˆ’ *Recovery*). For example, an `LGD` of 0.6 implies a 40% recovery rate in the event of a default. The `LGD` input sets the Portfolio on page 6-0 property.

`LGD` can alternatively be specified as a `NumCounterparties`-by-`2` matrix, where the first column holds the LGD mean values and the 2nd column holds the LGD standard deviations. Valid open intervals for LGD mean and standard deviation are:

- For the first column, the mean values are between `0` and `1`.
- For the second column, the LGD standard deviations are between `0` and `sqrt(m*(1-m))`.

Then, in the case of default, LGD values are drawn randomly from a beta distribution with provided parameters for the defaulting counterparty.

Data Types: `double`

`Weights` — Factor and idiosyncratic weights
array of factor and idiosyncratic weights

Factor and idiosyncratic weights, specified as a `NumCounterparties`-by-(`NumFactors + 1`) array. Each row contains the factor weights for a particular counterparty. Each column contains the weights for an underlying risk factor. The last column in `Weights` contains the idiosyncratic risk weight for each counterparty. The idiosyncratic weight represents the company-specific credit risk. The total of the weights for each counterparty (that is, each row) must sum to `1`. The `Weights` input sets the Portfolio on page 6-0 property.

For example, if a counterparty's creditworthiness is composed of 60% US, 20% European, and 20% idiosyncratic, then the `Weights` vector would be `[0.6 0.2 0.2]`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cdc = creditDefaultCopula(EAD,PD,LGD,Weights,'VaRLevel',0.99)`

`ID` — User-defined IDs for counterparties
`1:NumCounterparties` (default) | vector

User-defined IDs for counterparties, specified as the comma-separated pair consisting of `'ID'` and a `NumCounterparties`-by-`1` vector of `ID`s for each counterparty. `ID` is used to identify exposures in the `Portfolio` table and the risk contribution table. `ID` must be a numeric, a string array, or a cell array of character vectors. The `ID` name-value pair argument sets the Portfolio on page 6-0 property.

If unspecified, `ID` defaults to a numeric vector `1:NumCounterparties`.

Data Types: `double` | `string` | `cell`

`VaRLevel` — Value at risk level
`0.95` (default) | numeric between `0` and `1`

Value at risk level (used for reporting `VaR` and `CVaR`), specified as the comma-separated pair consisting of `'VaRLevel'` and a numeric between `0` and `1`. The `VaRLevel` name-value pair argument sets the VaRLevel on page 6-0 property.

Data Types: `double`

`FactorCorrelation` — Factor correlation matrix
identity matrix (default) | correlation matrix

Factor correlation matrix, specified as the comma-separated pair consisting of `'FactorCorrelation'` and a `NumFactors`-by-`NumFactors` matrix that defines the correlation between the risk factors. The `FactorCorrelation` name-value pair argument sets the FactorCorrelation on page 6-0 property.

If not specified, the factor correlation matrix defaults to an identity matrix, meaning that factors are not correlated.

Data Types: `double`

`UseParallel` — Flag to use parallel processing for simulations
false (default) | logical with value of `true` or `false`

Flag to use parallel processing for simulations, specified as the comma-separated pair consisting of `'UseParallel'` and a scalar value of `true` or `false`. The `UseParallel` name-value pair argument sets the UseParallel on page 6-0 property.

Note The `'UseParallel'` property can only be set when creating a `creditDefaultCopula` object if you have Parallel Computing Toolbox. Once the `'UseParallel'` property is set, parallel processing is used with `riskContribution` or `simulate`.

Data Types: `logical`

Properties

`Portfolio` — Details of credit portfolio
table

Details of credit portfolio, specified as a MATLAB table that contains all the portfolio data that was passed as input into `creditDefaultCopula`.

The `Portfolio` table has a column for each of the constructor inputs (`EAD`, `PD`, `LGD`, `Weights`, and `ID`). Each row of the table represents one counterparty.

For example:

```
ID    EAD         PD          LGD        Weights
__    ______    _________    _______    _________
```

```
    1     122.43     0.064853    0.68024    0.3  0.7
    2     70.386     0.073957    0.59256    0.3  0.7
    3     79.281     0.066235    0.52383    0.3  0.7
    4     113.42     0.01466     0.43977    0.3  0.7
    5     100.46     0.0042036   0.41838    0.3  0.7
```

Data Types: `table`

`FactorCorrelation` — Correlation matrix for credit factors
matrix

Correlation matrix for credit factors, specified as a `NumFactors`-by-`NumFactors` matrix. Specify the correlation matrix using the optional name-value pair argument `'FactorCorrelation'` when you create a `creditDefaultCopula` object.

Data Types: `double`

`VaRLevel` — Value at Risk Level
numeric between `0` and `1`

Value at risk level used when reporting VaR and CVaR, specified using an optional name-value pair argument `'VaRLevel'` when you create a `creditDefaultCopula` object.

Data Types: `double`

`PortfolioLosses` — Total portfolio losses
vector

Total portfolio losses, specified as a `1`-by-`NumScenarios` vector. The `PortfolioLosses` property is empty after you create a `creditDefaultCopula` object. After the `simulate` function is invoked, the `PortfolioLosses` property is populated with the vector of portfolio losses.

Data Types: `double`

`UseParallel` — Flag to use parallel processing for simulations
false (default) | logical with value of `true` or `false`

Flag to use parallel processing for simulations, specified using an optional name-value pair argument `'UseParallel'` when you create a `creditDefaultCopula` object. The `UseParallel` name-value pair argument sets the `UseParallel` property.

Note The `'UseParallel'` property can only be set when creating a `creditDefaultCopula` object if you have Parallel Computing Toolbox. Once the `'UseParallel'` property is set, parallel processing is used with `riskContribution` or `simulate`.

Data Types: `logical`

Object Functions

simulate	Simulate credit defaults using a creditDefaultCopula object
portfolioRisk	Generate portfolio-level risk measurements
riskContribution	Generate risk contributions for each counterparty in portfolio
confidenceBands	Confidence interval bands
getScenarios	Counterparty scenarios

Examples

Create a `creditDefaultCopula` Object and Simulate Credit Portfolio Losses

Load saved portfolio data.

```
load CreditPortfolioData.mat;
```

Create a `creditDefaultCopula` object with a two-factor model.

```
cdc = creditDefaultCopula(EAD,PD,LGD,Weights2F,'FactorCorrelation',FactorCorr2F)
```

```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioLosses: []
```

Set the `VaRLevel` to 99%.

```
cdc.VaRLevel = 0.99;
```

Simulate 100,000 scenarios, and view the portfolio risk measures.

```
 cdc = simulate(cdc,1e5)
```

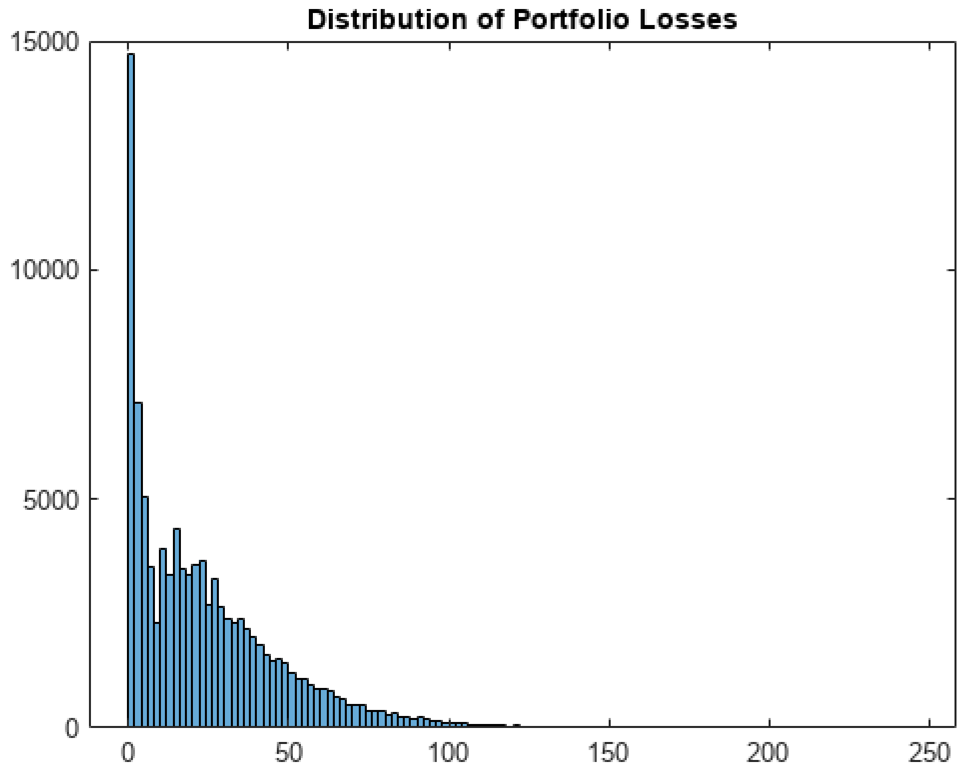
```
cdc = 
  creditDefaultCopula with properties:

            Portfolio: [100x5 table]
    FactorCorrelation: [2x2 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioLosses: [30.1008 3.6910 3.2895 19.2151 7.5761 44.5088 19.5419 1.7909 72.1443 12.69
```

```
 portRisk = portfolioRisk(cdc)
```

```
portRisk=1×4 table
      EL       Std       VaR      CVaR 
    ______    ______    _____    ______

    24.876    23.778    102.4    121.28
```

View a histogram of the portfolio losses.

```
histogram(cdc.PortfolioLosses);
title('Distribution of Portfolio Losses');
```



For further analysis, use the `simulate`, `portfolioRisk`, `riskContribution`, and `getScenarios` functions with the `creditDefaultCopula` object.

Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`table` | `simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios` | `creditMigrationCopula` | `nearcorr`

Topics

“Modeling Correlated Defaults with Copulas” on page 4-18
“creditDefaultCopula Simulation Workflow” on page 4-5
“Modeling Correlated Defaults with Copulas” on page 4-18
“One-Factor Model Calibration” on page 4-64
“Corporate Credit Risk” on page 1-3
“Credit Simulation Using Copulas” on page 4-2

External Websites

Parallel Computing with MATLAB (53 min 27 sec)

Cox

Create `Cox` model object for lifetime probability of default

Description

Create and analyze a `Cox` model object to calculate lifetime probability of default (PD) using this workflow:

1. Use `fitLifetimePDModel` to create a `Cox` model object.
2. Optionally, use `discardResiduals` to remove residual information from the `Cox` model object.
3. Use `predict` to predict the conditional PD and `predictLifetime` to predict the lifetime PD.
4. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
5. Use `modelCalibration` to return the root mean square error (RMSE) of observed and predicted PD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
CoxPDModel = fitLifetimePDModel(data,ModelType,AgeVar=agevar_value)
CoxPDModel = fitLifetimePDModel( ___ ,Name=Value)
```

Description

`CoxPDModel = fitLifetimePDModel(data,ModelType,AgeVar=agevar_value)` creates a `Cox` PD model object.

If you do not specify variable information for `IDVar`, `LoanVars`, `MacroVars`, and `ResponseVar`, then:

- `IDVar` is set to the first column in the `data` input.
- `LoanVars` is set to include all columns from the second to the second-to-last columns of the `data` input.
- `ResponseVar` is set to the last column in the `data` input.

`CoxPDModel = fitLifetimePDModel( ___ ,Name=Value)` sets optional properties on page 6-539 using additional name-value arguments in addition to the required arguments in the previous syntax. For example, `CoxPDModel = fitLifetimePDModel(data(TrainDataInd,:),"Cox",ModelID="Cox_A",Descripion="Cox_model",AgeVar="YOB",IDVar="ID",LoanVars="ScoreGroup",MacroVars={'GDP','Market'},ResponseVar="Default",TimeInterval=1,TieBreakMethod='Efron')` creates a `CoxPDModel` using a `Cox` model type. You can specify multiple name-value arguments.

Input Arguments

data — Data
table

Data, specified as a table, in panel data form. The data must contain an ID column and an Age column. The response variable must be a binary variable with the value 0 or 1, with 1 indicating default.

Data Types: table

ModelType — Model type
string with value "Cox" | character vector with value 'Cox'

Model type, specified as a string with the value "Cox" or a character vector with the value 'Cox'.

Data Types: char | string

Cox Name-Value Arguments

Specify required and optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: CoxPDModel = fitLifetimePDModel(data(TrainDataInd,:),"Cox",ModelID="Cox_A",Descripion="Cox_model",AgeVar="YOB",IDVar="ID",LoanVars="ScoreGroup",MacroVars={'GDP','Market'},ResponseVar="Default",TimeInterval=1)

Required Cox Name-Value Argument

AgeVar — Age variable indicating which column in data contains loan age information
string | character vector

Age variable indicating which column in data contains the loan age information, specified as AgeVar and a string or character vector.

Note The required name-value argument AgeVar is not treated as a predictor in the Cox lifetime PD model. When using a Cox model, you must specify predictor variables using LoanVars or MacroVars. The AgeVar values are the event times for the underlying Cox proportional hazards model.

AgeVar values for each ID should be increasing. If there are nonpositive age increments, fitLifetimePDModel warns when you create a Cox model and removes the IDs with nonpositive age increments. By default, the TimeInterval value is set to the most common age increment in the training data.

Data Types: string | char

Optional Cox Name-Value Arguments

ModelID — User-defined model ID
Cox (default) | string | character vector

User-defined model ID, specified as ModelID and a string or character vector. The software uses the ModelID to format outputs and is expected to be short.

Data Types: `string` | `char`

`Description` — User-defined description for model
`""` (default) | string | character vector

User-defined description for model, specified as `Description` and a string or character vector.

Data Types: `string` | `char`

`IDVar` — ID variable indicating which column in `data` contains loan or borrower ID
1st column of `data` (default) | string | character vector

ID variable indicating which column in `data` contains the loan or borrower ID, specified as `IDVar` and a string or character vector.

Data Types: `string` | `char`

`LoanVars` — Loan variables indicating which column in `data` contains loan-specific information
all columns of `data` that are not the first or last column (default) | string array | cell array of character vectors

Loan variables indicating which column in `data` contains the loan-specific information, such as origination score or loan-to-value ratio, specified as `LoanVars` and a string array or cell array of character vectors.

Data Types: `string` | `cell`

`MacroVars` — Macro variables indicating which column in `data` contains macroeconomic information
`""` (default) | string array | cell array of character vectors

Macro variables indicating which column in `data` contains the macroeconomic information, such as gross domestic product (GDP) growth or unemployment rate, specified as `MacroVars` and a string array or cell array of character vectors.

Data Types: `string` | `cell`

`ResponseVar` — Variable indicating which column in `data` contains response variable
string | character vector

Variable indicating which column in `data` contains the response variable, specified as `ResponseVar` and a logical value.

Note The response variable *values* in the `data` must be a binary variable with `0` or `1` values, with `1` indicating default.

In Cox lifetime PD models, the `ResponseVar` values define the censoring information for the underlying Cox proportional hazards model.

Data Types: `string` | `char`

`TimeInterval` — Distance between age values in panel data input
set to most common `AgeVar` increment in the training `data` (default) | positive numeric

Distance between age values in training data in the panel data input, specified as TimeInterval and a positive numeric scalar.

Use the TimeInterval name-value argument to fit time-dependent models and also as the time interval for the PD computation when you use the predict function. For example, if the age data (AgeVar) is 1, 2, 3, ..., then the TimeInterval is 1; if the age data is 0.25, 0.5, 0.75,..., then the TimeInterval is 0.25. For more information, see “Time Interval for Cox Models” on page 6-551 and “Lifetime Prediction and Time Interval” on page 6-342.

Note Unlike Logistic and Probit models, a Cox model requires an AgeVar variable. By default, if you do not specify a TimeInterval when creating a Cox model, the TimeInterval is inferred from the increments in the AgeVar values in the training data.

Data Types: double

TieBreakMethod — Method to handle tied default times
"breslow" (default) | string with value "breslow" or "efron" | character vector with value 'breslow' or 'efron'

Method to handle tied default times, specified as a string or character vector with one of the following tie-break methods:

- breslow — Breslow's approximation to the partial likelihood
- efron — Efron's approximation to the partial likelihood

For credit applications, the time to default comes discretized and there are many "ties." This means that are multiple borrowers that may default at the same (discretized) time (such as, in the second year of their loan). TieBreakMethod supports the breslow or efron methods to handle this scenario.

Data Types: string | char

Properties

ModelID — User-defined model ID
Probit (default) | string

User-defined model ID, returned as a string.

Data Types: string

Description — User-defined description
"" (default) | string

User-defined description, returned as a string.

Data Types: string

UnderlyingModel — Underlying statistical model
Cox model

Underlying statistical model, returned as a returned as a Cox proportional hazards model object. For more information, see fitcox and CoxModel.

Data Types: `CoxModel`

`IDVar` — ID variable indicating which column in `data` contains loan or borrower ID
1st column of `data` (default) | string

ID variable indicating which column in `data` contains the loan or borrower ID, returned as a string.

Data Types: `string`

`AgeVar` — Age variable indicating which column in `data` contains loan age information
string

Age variable indicating which column in `data` contains the loan age information, returned as a string.

Data Types: `string`

`LoanVars` — Loan variables indicating which column in `data` contains loan-specific information
all columns of `data` that are not the first or last column (default) | string array

Loan variables indicating which column in `data` contains the loan-specific information, returned as a string array.

Data Types: `string`

`MacroVars` — Macro variables indicating which column in `data` contains macroeconomic information
`""` (default) | string array

Macro variables indicating which column in `data` contains the macroeconomic information, returned as a string array.

Data Types: `string`

`ResponseVar` — Variable indicating which column in `data` contains response variable
string

Variable indicating which column in `data` contains the response variable, returned as a string.

Data Types: `string`

`TimeInterval` — Distance between age values in panel data input
set to most common `AgeVar` increment in the training data (default) | positive numeric

This property is read-only.

Distance between age values in panel `data` input, returned as a scalar positive numeric.

Data Types: `double`

`ExtrapolationFactor` — Extrapolation factor
`1` (default) | positive numeric between `0` and `1`

Extrapolation factor, returned as a positive numeric scalar between `0` and `1`.

By default, the `ExtrapolationFactor` is set to `1`. For age values (`AgeVar`) greater than the maximum age observed in the training data, the conditional PD, computed with `predict`, uses the maximum age observed in the training data. In particular, the predicted PD value is constant if the predictor values do not change and only the age values change when the `ExtrapolationFactor` is 1. For more information, see “Extrapolation for Cox Models” on page 6-335, “Extrapolation Factor for Cox Models” on page 6-335, and “Use Cox Lifetime PD Model to Predict Conditional PD” on page 6-329.

Data Types: `double`

`TieBreakMethod` — Method to handle tied default times
`"breslow"` (default) | string with value `"breslow"` or `"efron"`

Method to handle tied default times, returned as a string.

Data Types: `string`

Object Functions

predict	Compute conditional PD
predictLifetime	Compute cumulative lifetime PD, marginal PD, and survival probability
modelDiscrimination	Compute AUROC and ROC data
modelCalibration	Compute RMSE of predicted and observed PDs on grouped data
modelDiscriminationPlot	Plot ROC curve
modelCalibrationPlot	Plot observed default rates compared to predicted PDs on grouped data
discardResiduals	Discard residual information of underlying Cox model

Examples

Create Cox Lifetime PD Model

This example shows how to use `fitLifetimePDModel` to create a `Cox` model using credit and macroeconomic data.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72     7.61
```

```
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % For reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Cox Lifetime PD Model

Use `fitLifetimePDModel` to create a `Cox` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Cox",...
    AgeVar="YOB", ...
    IDVar="ID", ...
    LoanVars="ScoreGroup", ...
    MacroVars={'GDP','Market'}, ...
    ResponseVar="Default");
disp(pdModel)
```

```
  Cox with properties:

            TimeInterval: 1
     ExtrapolationFactor: 1
                 ModelID: "Cox"
             Description: ""
```

```
        UnderlyingModel: [1x1 CoxModel]
                  IDVar: "ID"
                 AgeVar: "YOB"
               LoanVars: "ScoreGroup"
              MacroVars: ["GDP"    "Market"]
            ResponseVar: "Default"
```

Display the underlying model.

```
disp(pdModel.UnderlyingModel)
```

```
Cox Proportional Hazards regression model

                                Beta          SE         zStat       pValue
                             __________    _________    _______    __________

    ScoreGroup_Medium Risk      -0.6794     0.037029    -18.348    3.4442e-75
    ScoreGroup_Low Risk         -1.2442     0.045244    -27.501    1.7116e-166
    GDP                       -0.084533     0.043687     -1.935      0.052995
    Market                   -0.0084411    0.0032221    -2.6198     0.0087991


Log-likelihood: -41742.871
```

Validate Model

Use `modelDiscrimination` to measure the ranking of customers by PD.

```
DataSetChoice = Testing ;
if DataSetChoice=="Training"
    Ind = TrainDataInd;
else
    Ind = TestDataInd;
end

DiscMeasure = modelDiscrimination(pdModel,data(Ind,:),SegmentBy="ScoreGroup")
```

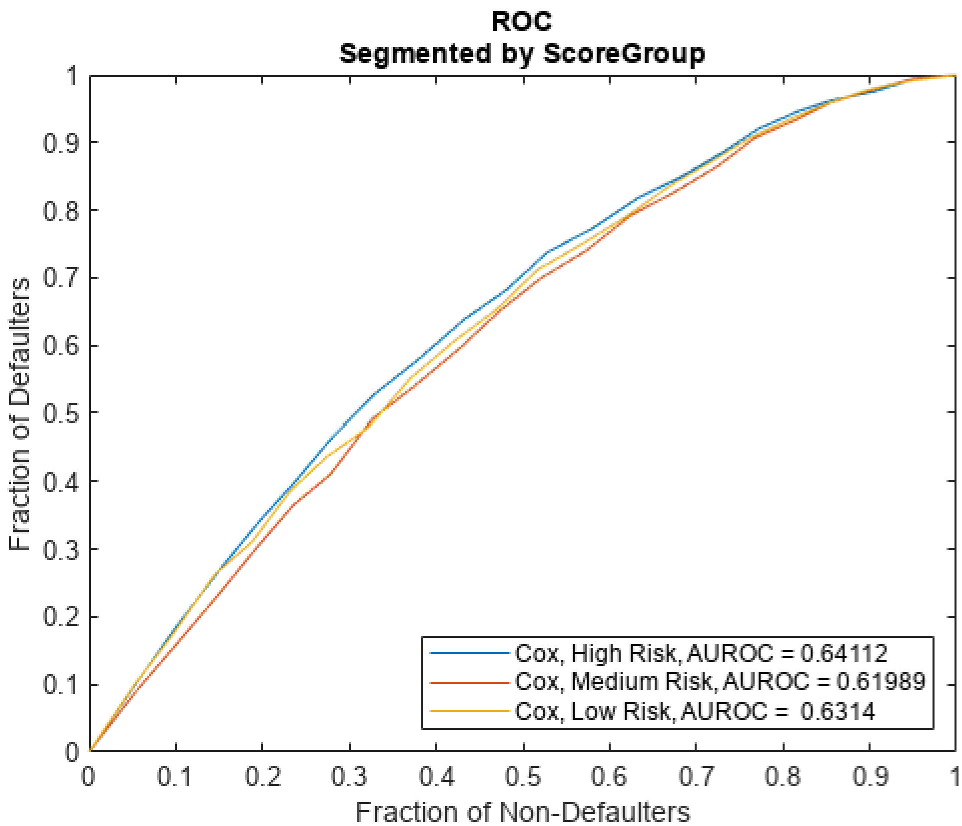
```
DiscMeasure=3×1 table
                                      AUROC
                                     _______

    Cox, ScoreGroup=High Risk        0.64112
    Cox, ScoreGroup=Medium Risk      0.61989
    Cox, ScoreGroup=Low Risk          0.6314
```

```
disp(DiscMeasure)
```

```
                                      AUROC
                                     _______

    Cox, ScoreGroup=High Risk        0.64112
    Cox, ScoreGroup=Medium Risk      0.61989
    Cox, ScoreGroup=Low Risk          0.6314
```

Use `modelDiscriminationPlot` to visualize the ROC curve.

```
modelDiscriminationPlot(pdModel,data(Ind,:),SegmentBy="ScoreGroup")
```



Use `modelCalibration` to measure the calibration of the predicted PD values. The `modelCalibration` function requires a grouping variable and compares the accuracy of the observed default rate in the group with the average predicted PD for the group.

```
CalMeasure = modelCalibration(pdModel,data(Ind,:),{'YOB','ScoreGroup'})
```

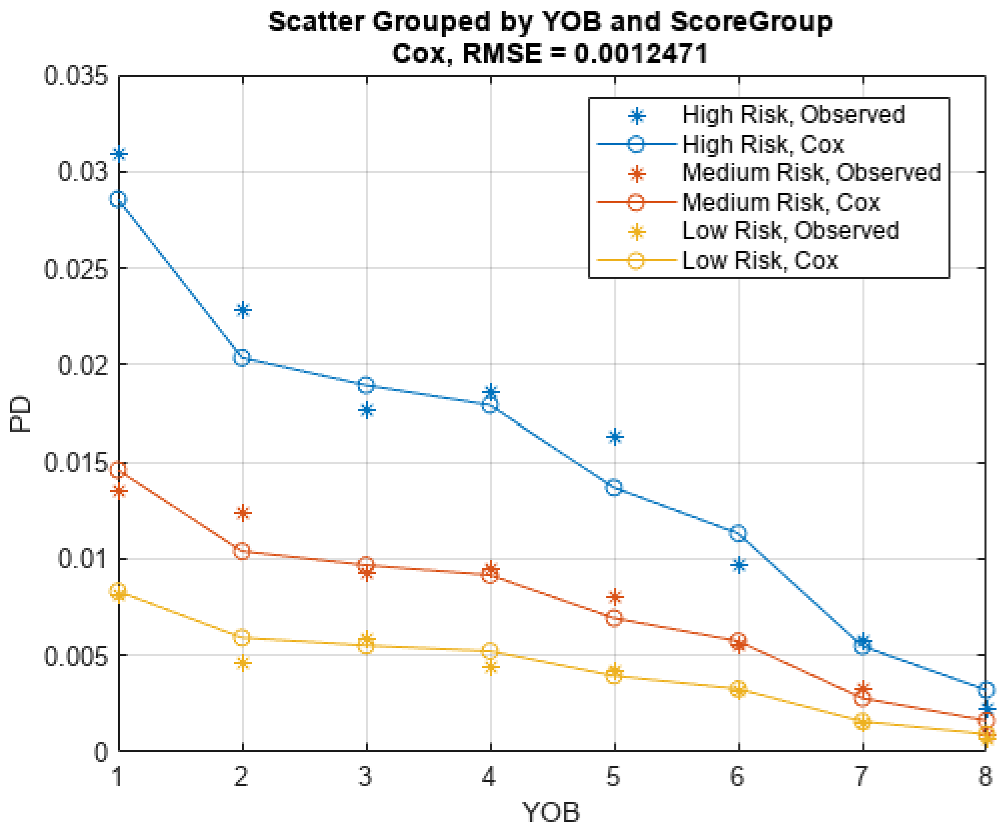
```
CalMeasure=table
                                      RMSE   
                                   _________

    Cox, grouped by YOB, ScoreGroup    0.0012471
```

```
disp(CalMeasure)
```

```
                                      RMSE   
                                   _________

    Cox, grouped by YOB, ScoreGroup    0.0012471
```

Use `modelCalibrationPlot` to visualize the observed default rates compared to the predicted PD.

```
modelCalibrationPlot(pdModel,data(Ind,:),{'YOB','ScoreGroup'})
```



Predict Conditional and Lifetime PD

Use the `predict` function to predict conditional PD values. The prediction is a row-by-row prediction.

```
%dataCustomer1 = data(1:8,:);
CondPD = predict(pdModel,data(Ind,:));
```

Use `predictLifetime` to predict the lifetime cumulative PD values (computing marginal and survival PD values is also supported).

```
LifetimePD = predictLifetime(pdModel,data(Ind,:));
```

Select Tie-Break Method for Cox Lifetime PD Models

This example shows how to create a `Cox` model and select the tie-break method while fitting a Cox lifetime PD model.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____
```

```
    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Join the Data

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition the Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Cox Lifetime PD Model with Breslow's Method

Use `fitLifetimePDModel` to create a `Cox` model using the training data. Use the name-value argument `TieBreakMethod` to set tie-break method to `'breslow'`. This is the default choice for this argument.

```
pdModel1 = fitLifetimePDModel(data(TrainDataInd,:),"Cox",...
ModelID="Cox-Breslow", IDVar="ID", AgeVar="YOB", ...
LoanVars="ScoreGroup", MacroVars={'GDP','Market'}, ...
ResponseVar="Default",TieBreakMethod='breslow');
```

Display the underlying model.

```
disp(pdModel1.Model)
```

```
Cox Proportional Hazards regression model

                                 Beta          SE         zStat        pValue  
                              __________    _________    _______    ___________

    ScoreGroup_Medium Risk       -0.6794     0.037029    -18.348     3.4442e-75
    ScoreGroup_Low Risk          -1.2442     0.045244    -27.501    1.7116e-166
    GDP                        -0.084533     0.043687     -1.935       0.052995
    Market                    -0.0084411    0.0032221    -2.6198      0.0087991


Log-likelihood: -41742.871
```

Use `predict` to predict the conditional PD.

```
pd1 = predict(pdModel1,data(TestDataInd,:));
```

Create a Cox Lifetime PD Model with Efron's Method

Use `fitLifetimePDModel` to create a `Cox` model using the training data. Use the name-value argument `TieBreakMethod` to set tie-break method to `'Efron'`. This is the default choice for this argument.

```
pdModel2 = fitLifetimePDModel(data(TrainDataInd,:),"Cox",...
ModelID="Cox-Efron", IDVar="ID", AgeVar="YOB", ...
LoanVars="ScoreGroup", MacroVars={'GDP','Market'}, ...
ResponseVar="Default",TieBreakMethod='efron');
```

Display the underlying model. The coefficients are only slightly different for this data set.

```
disp(pdModel2.Model)
```

```
Cox Proportional Hazards regression model

                                Beta          SE         zStat       pValue
                             __________    _________    _______    __________

    ScoreGroup_Medium Risk      -0.6844     0.037029    -18.483    2.8461e-76
    ScoreGroup_Low Risk         -1.2515     0.045243    -27.662    2.006e-168
    GDP                       -0.084985     0.043691    -1.9452      0.051756
    Market                   -0.0085126    0.0032223    -2.6418     0.0082469


Log-likelihood: -41713.445
```

Use `predict` to predict the conditional PD for the second `Cox` model.

```
pd2 = predict(pdModel2,data(TestDataInd,:));
```

Compare Cox Models

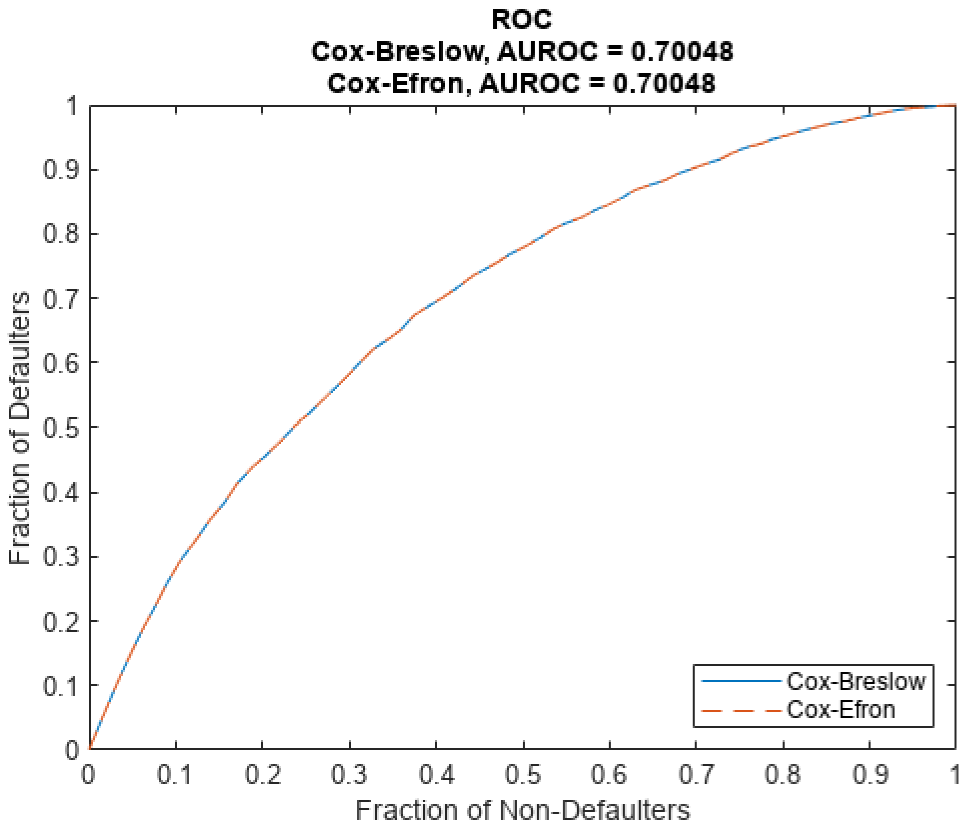
The predictions for the two `Cox` models are almost the same for this data set.

```
[pd1(1:10) pd2(1:10)]
```

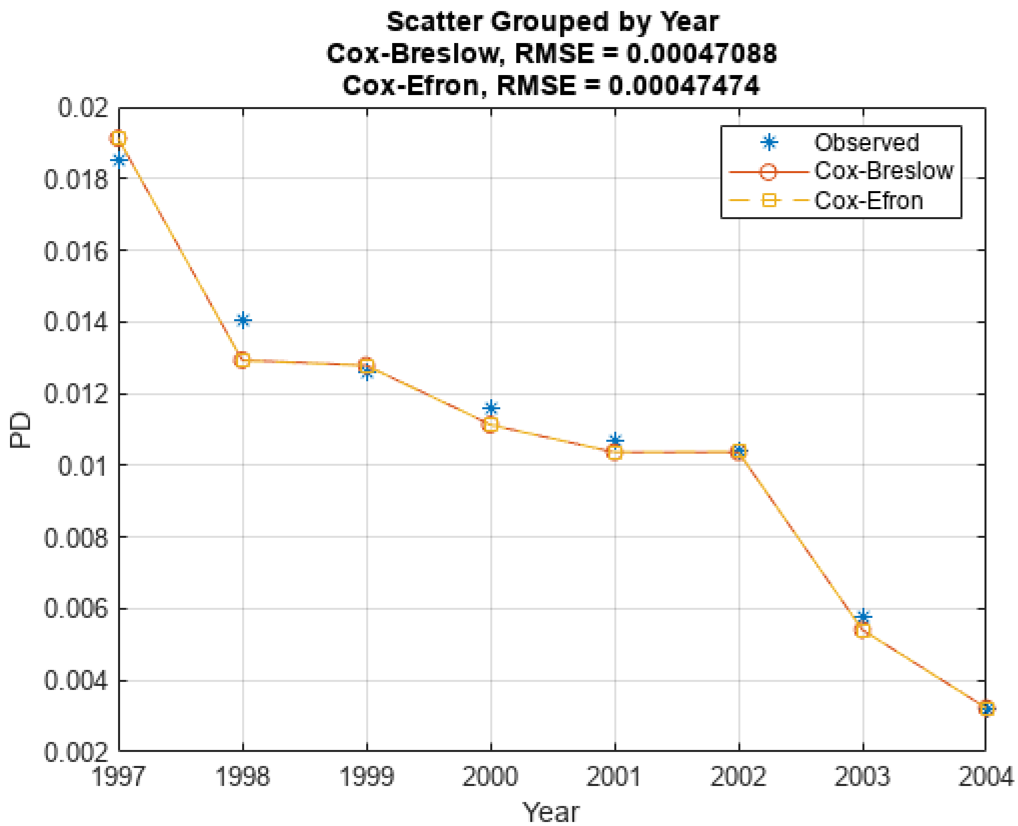
```
ans = 10×2

    0.0162    0.0161
    0.0091    0.0090
    0.0081    0.0081
    0.0073    0.0072
    0.0064    0.0064
    0.0072    0.0072
    0.0030    0.0030
    0.0016    0.0016
    0.0162    0.0161
    0.0091    0.0090
```

For this data set, the model discrimination (`modelDiscrimination`) does not seem to change with the `TieBreakMethod` method and the model accuracy (`modelCalibration`) shows only a negligible difference in RMSE.

```
modelDiscriminationPlot(pdModel1,data(TestDataInd,:),ReferencePD=pd2,ReferenceID=pdModel2.ModelI
```




```
modelCalibrationPlot(pdModel1,data(TestDataInd,:),'Year',ReferencePD=pd2,ReferenceID=pdModel2.Moc
```



More About

Cox Proportional Hazards Models

The Cox proportional hazards (PH) model is a survival model and it models the time until an event of interest occurs.

For probability of default (PD) models, the event of interest is the default on a credit obligation. `Cox` models need information on whether there was a default and when it happened. For other commonly used PD models, a binary variable indicating whether there was a default is enough. `Cox` PD models need that information, plus the age of the loan at the time of default.

The `Cox` proportional hazards (PH) model, also known as a `Cox` regression model, assumes the hazard rate is of the form

$$h(t; X) = h_0(t)\exp(X\beta)$$

where

- $h_0(t)$ is the baseline hazard rate.
- X is the predictor data.
- β is a vector of coefficients of the predictors.
- $\exp(X\hat{I}^2)$ is the hazard ratio.

The baseline hazard rate is a reference hazard level, common to all observations, and it does not depend on the predictor values. The hazard ratio is the factor that scales the baseline hazard value up or down, depending on the predictor values. For lower risk observations, the hazard ratio is less than 1 and this reduces the hazard rate. For higher risk observations, the hazard ratio increases the hazard rate.

In the hazard rate formula, the predictor values in X are fixed, or *independent of time*. This is the basic version of the Cox PH model. For PD models, the basic version of the Cox PH model includes predictors that have constant values, such as the origination score, or whether a property is for residential or commercial purposes.

The *time-dependent* Cox PH model allows predictor values to change over time. For example, the loan-to-value (LTV) ratio changes over the life of a loan, and the macroeconomic variables change from period to period. Therefore, the following hazard rate formula for time-dependent models includes predictor values that can be a function of time:

$$h(t; X) = h_0(t)\exp(X(t)\beta)$$

The data input for fitLifetimePDModel must be in panel data form. For each ID (IDVar), there are multiple rows of data. The panel data input is required for both time-dependent and time - independent models.

For time-independent predictors, the predictor value is constant for each ID. For example, the score at origination for each customer is constant throughout the life of the loan, and this value is repeated for each row corresponding to the same ID in the panel data format.

For time-dependent predictors, the values may change from one row to the next for the same ID. The assumption is that the predictor values in each row are valid in the time interval defined by the age value (AgeVar) in the previous row and the age value in the current row.

Time Interval for Cox Models

Time is discretized into intervals, and predictor values in the training data (data input) are constant for each interval: X_1 from t_0 to t_1; X_2 from t_1 to t_2; and so forth.

The data input must be in panel data form, with multiple observations for each ID, with corresponding age information (the t_k values, the AgeVar column) and the corresponding default indicator values (the ResponseVar column).

Assume that $t_k - t_{k-1} = \Delta t$ for all k and this is the time interval. This time interval is the age increment for consecutive observations in the age data (AgeVar). The assumption is that these increments are regular and that the default indicator (ResponseVar) is defined consistently with this time interval, in the sense that a 1 means there was a default in a time interval of length Δt. The time interval Δt is also used for the computation of the probability of default. For more information, see “Lifetime Prediction and Time Interval” on page 6-342.

Survival and Probability of Default for Cox Models

The survival function $S(t)$ is a function of time, and gives the probability of surviving longer than a given time t.

$$S(t) = P(T > t)$$

where

- T is the failure time, the random variable of interest, and in the `Cox` model case, the time to default.
- t is the specific time of interest, for example, 1 year.

The main relationship between the survival function and the hazard rate is

$$S(t) = \exp\left(-\int_0^t h(u)du\right)$$

Higher values of the hazard rate cause the survival probability to drop faster. Conversely, lower values of the hazard rate cause the survival probability to rise faster.

The probability of default (PD) is the conditional probability of defaulting in a time interval, given that there has been no default prior to that interval. For example, the probability of default between time s and t, with $s < t$, is represented as:

$$\begin{aligned} PD(s,t) &= P(s < T \le t | T > s) \\ &= \frac{S(s) - S(t)}{S(s)} \\ &= 1\text{-}\frac{S(t)}{S(s)} \end{aligned}$$

In credit applications, the time interval of interest, Δt, is consistent with the training data and the definition of default in the response variable. The PD is a function of a single time variable t and the implicit time interval Δt:

$$PD(t) = 1 - \frac{S(t)}{S(t - \Delta t)}$$

Version History

Introduced in R2021b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

R2023a: Added `TieBreakMethod` name-value argument

Behavior changed in R2023a

The `TieBreakMethod` name-value argument enables you to specify the method to handle tied default times.

R2023a: Added `discardResiduals` method for Cox model

Behavior changed in R2023a

Use the `discardResiduals` method to discard residual information of the underlying Cox model.

R2023a: `Model` property renamed to `UnderlyingModel`
Behavior changed in R2023a

The `Model` property is renamed to `UnderlyingModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

Functions
`fitLifetimePDModel` | `Logistic` | `Probit` | `customLifetimePDModel`

Topics
“Basic Lifetime PD Model Validation” on page 4-129
“Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
“Compare Lifetime PD Models Using Cross-Validation” on page 4-121
“Expected Credit Loss Computation” on page 4-124
“Compare Model Discrimination and Model Calibration to Validate of Probability of Default” on page 4-144
“Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models” on page 4-75
“Modeling Probabilities of Default with Cox Proportional Hazards” on page 4-28
“Overview of Lifetime Probability of Default Models” on page 1-25

discardResiduals

Discard residual information of underlying Cox model

Syntax

```
pdModel = discardResiduals(pdModel)
```

Description

`pdModel = discardResiduals(pdModel)` discards residual information of underlying `Cox` model to reduce memory footprint of the `Cox` lifetime PD model.

Examples

Discard Residuals for Cox Lifetime PD Model

This example shows how to create a `Cox` model and then use `discardResiduals` to remove residual information to reduce the model's memory usage.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the Data

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition the Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Cox Lifetime PD Model

Use `fitLifetimePDModel` to create a `Cox` model.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Cox",...
   IDVar="ID", AgeVar="YOB", LoanVars="ScoreGroup", ...
   MacroVars={'GDP','Market'}, ResponseVar="Default");
disp(pdModel)
```

```
  Cox with properties:

            TimeInterval: 1
     ExtrapolationFactor: 1
                 ModelID: "Cox"
             Description: ""
         UnderlyingModel: [1x1 CoxModel]
                   IDVar: "ID"
                  AgeVar: "YOB"
                LoanVars: "ScoreGroup"
               MacroVars: ["GDP"    "Market"]
             ResponseVar: "Default"
```

The Cox `pdModel` object uses a noticeable amount of memory.

```
whos pdModel
```

```
  Name         Size                Bytes  Class                 Attributes

  pdModel      1x1              59000597  risk.credit.pd.Cox
```

This is because the underlying Cox model stores residual information, and multiple residual types are supported.

```
head(pdModel.UnderlyingModel.Residuals)
```

```
    CoxSnell     Deviance    Martingale           Schoenfeld                  ScaledSchoenfeld
    _________    ________    __________    ________________________    ________________________

    0.0092625       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     0.012537       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     0.018878       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     0.026346       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     0.036303       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     0.051269       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     0.038922       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
     0.034104       0            0         NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
```

For additional information on the residuals, see `CoxModel`.

Remove Residual Information

For prediction purposes, the residual information can be discarded using `discardResiduals` without affecting the prediction or validation functionality of the Cox lifetime PD model.

```
pdModel = discardResiduals(pdModel)
```

```
pdModel = 
  Cox with properties:

           TimeInterval: 1
    ExtrapolationFactor: 1
                ModelID: "Cox"
            Description: ""
        UnderlyingModel: [1x1 CoxModel]
                  IDVar: "ID"
                 AgeVar: "YOB"
               LoanVars: "ScoreGroup"
              MacroVars: ["GDP"    "Market"]
            ResponseVar: "Default"
```

The model storage is minimal once the residuals have been discarded and the Residuals property of the underlying model have been emptied.

```
whos pdModel
```

```
  Name         Size             Bytes  Class                 Attributes

  pdModel      1x1               8477  risk.credit.pd.Cox
```

```
pdModel.UnderlyingModel.Residuals
```

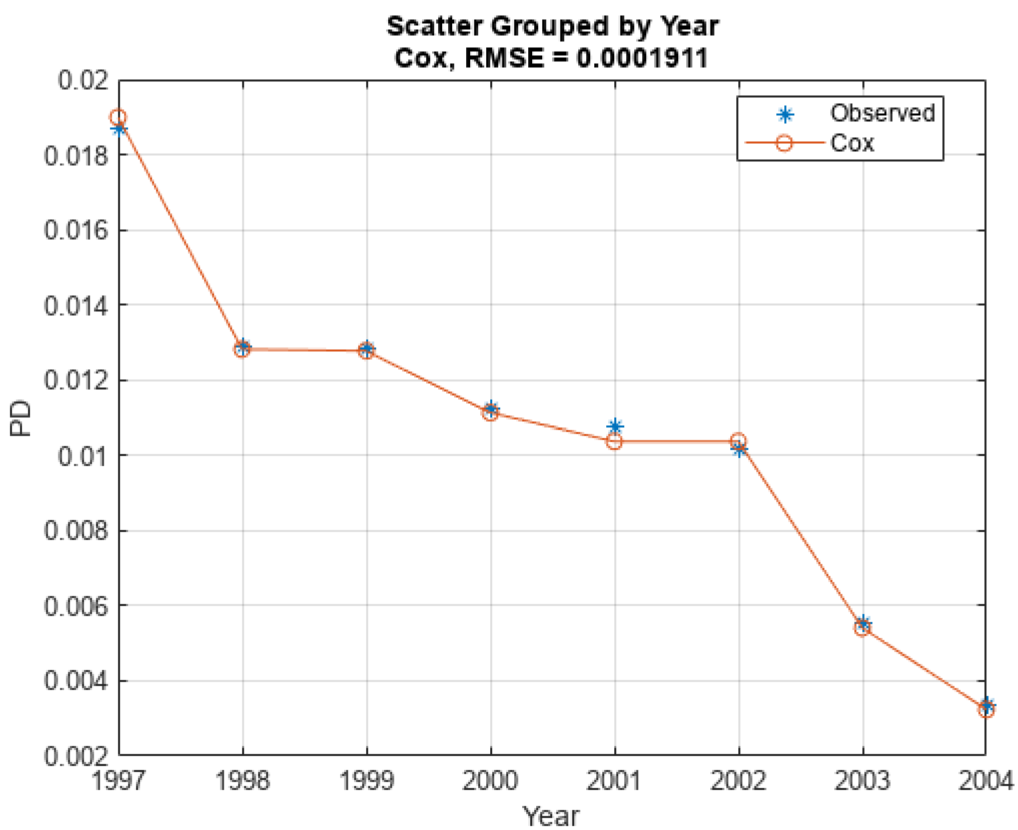
```
ans =

  0x1 empty table
```

The prediction and validation functions are not affected after the residuals have been discarded.

```
pdLifetime = predictLifetime(pdModel,data(1:8,:))
```

```
pdLifetime = 8×1

    0.0092
    0.0143
    0.0189
    0.0229
    0.0265
    0.0305
    0.0321
    0.0330
```

```
modelCalibrationPlot(pdModel,data(TrainDataInd,:),'Year')
```



Copyright 2022 The MathWorks, Inc.

Input Arguments

pdModel — Probability of default model
Cox object

Probability of default model, specified as a previously created `Cox` object using `fitLifetimePDModel`.

Data Types: `object`

Output Arguments

`pdModel` — Updated Cox PD model
object

Updated Cox PD model, returned as a `Cox` model.

Version History

Introduced in R2023a

See Also

`modelCalibration` | `modelDiscrimination` | `modelDiscriminationPlot` | `modelCalibrationPlot` | `predictLifetime` | `fitLifetimePDModel` | `Cox` | `customLifetimePDModel`

Topics

"Basic Lifetime PD Model Validation" on page 4-129
"Compare Logistic Model for Lifetime PD to Champion Model" on page 4-113
"Compare Lifetime PD Models Using Cross-Validation" on page 4-121
"Expected Credit Loss Computation" on page 4-124
"Compare Model Discrimination and Model Calibration to Validate of Probability of Default" on page 4-144
"Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models" on page 4-75
"Overview of Lifetime Probability of Default Models" on page 1-25

creditMigrationCopula

Simulate and analyze multifactor credit migration rating model

Description

The `creditMigrationCopula` takes as input a portfolio of credit-sensitive positions with a set of counterparties and performs a copula-based, multifactor simulation of credit rating migrations. Counterparty credit rating migrations and subsequent changes in portfolio value are calculated for each scenario and several risk measurements are reported.

`creditMigrationCopula` associates each counterparty with a random variable, called a latent variable, which is mapped to credit ratings based on a rating transition matrix. For each scenario, the value of the position with each counterparty is recomputed based on the realized credit rating of the counterparty. These latent variables are simulated by using a multifactor model, where systemic credit fluctuations are modeled with a series of risk factors. These factors can be based on industry sectors (such as financial or aerospace), geographical regions (such as USA or Eurozone), or any other underlying driver of credit risk. Each counterparty is assigned a series of weights which determine their sensitivity to each underlying credit factors.

The inputs to the model are:

- `migrationValues` — Values of the counterparty positions for each credit rating.
- `ratings` — Current credit rating for each counterparty.
- `transitionMatrix` — Matrix of credit rating transition probabilities.
- `LGD` — Loss given default (1 âˆ’ *Recovery*).
- `Weights` — Factor and idiosyncratic model weights

After you create `creditMigrationCopula` object (see "Create creditMigrationCopula" on page 6-559 and "Properties" on page 6-563), use the `simulate` function to simulate credit migration by using the multifactor model. Then, for detailed reports, use the following functions: `portfolioRisk`, `riskContribution`, `confidenceBands`, and `getScenarios`.

Creation

Syntax

```
cmc = creditMigrationCopula(migrationValues,ratings,transitionMatrix,LGD,
Weights)
cmc = creditMigrationCopula( ___ ,Name,Value)
```

Description

`cmc = creditMigrationCopula(migrationValues,ratings,transitionMatrix,LGD, Weights)` creates a `creditMigrationCopula` object. The `creditMigrationCopula` object has the following properties:

- Portfolio on page 6-0 :

  A table with the following variables:

  - `ID` — ID to identify each counterparty
  - `migrationValues` — Values of counterparty positions for each credit rating
  - `ratings` — Current credit rating for each counterparty
  - `LGD` — Loss given default
  - `Weights` — Factor and idiosyncratic weights for counterparties
- FactorCorrelation on page 6-0 :

  Factor correlation matrix, a `NumFactors`-by-`NumFactors` matrix that defines the correlation between the risk factors.
- RatingLabels on page 6-0 :

  The set of all possible credit ratings.
- TransitionMatrix on page 6-0 :

  The matrix of probabilities that a counterparty transitions from a starting credit rating to a final credit rating. The rows represent the starting credit ratings and the columns represent the final ratings. The top row holds the probabilities for a counterparty that starts at the highest rating (for example `AAA`) and the bottom row holds those for a counterparty starting in the default state. The bottom row may be omitted, indicating that a counterparty in default remains in default. Each row must sum to `1`. The order of rows and columns must match the order of credit ratings defined in the `RatingLabels` parameter. The last column holds the probability of default for each of the ratings. If unspecified, the default rating labels are: `"AAA","AA","A","BBB","BB","B","CCC","D"`.
- VaRLevel on page 6-0 :

  The value-at-risk level, used when reporting VaR and CVaR.
- PortfolioValues on page 6-0 :

  A `NumScenarios`-by-`1` vector of portfolio values. This property is empty until you use the `simulate` function.

`cmc = creditMigrationCopula( ___ ,Name,Value)` sets Properties on page 6-563 using name-value pairs and any of the arguments in the previous syntax. For example, `cmc = creditMigrationCopula(migrationValues,ratings,transitionMatrix,LGD,Weights,'VaRLevel',0.99)`. You can specify multiple name-value pairs as optional name-value pair arguments.

Input Arguments

`migrationValues` — Values of counterparty positions for each credit rating

matrix

Values of the counterparty positions for each credit rating, specified as a `NumCounterparties`-by-`NumRatings` matrix. Each row holds the possible values of the counterparty position for each credit rating. The last rating must be the default rating. The `migrationValues` input sets the Portfolio on page 6-0 property.

The migration value for the default rating (the last column of `migrationValues` input) is pre-recovery. This is a reference value (for example, face value, forward value at current rating, or other) that is multiplied by the recovery rate during the simulation to get the value of the asset in the event of default. The recovery rate is defined as `1-LGD`, where `LGD` is specified using the `LGD` input argument. The `LGD` is either a constant or a random number drawn from a beta distribution (see the description of the `LGD` input).

Note The `creditMigrationCopula` model simulates the changes in portfolio value over a fixed time period (for example, one year). The `migrationValues` and `transitionMatrix` must be specific to a particular time period.

Data Types: `double`

`ratings` — Current credit rating for each counterparty

cell array of character vectors | numeric value | string

Current credit rating for each counterparty, specified as a `NumCounterparties`-by-`1` vector that represents the initial credit states. The set of all valid credit ratings and their order is defined by using the optional `RatingLabels` parameter. The `ratings` input sets the Portfolio on page 6-0 property.

If `RatingLabels` are unspecified, the default rating labels are: `"AAA","AA","A","BBB","BB","B","CCC","D"`.

Data Types: `double` | `string` | `cell`

`transitionMatrix` — Credit rating transition probabilities

numeric value

Credit rating transition probabilities, specified as a `NumRatings`-by-`NumRatings` matrix. The matrix contains the probabilities that a counterparty starting at a particular credit rating transitions to every other rating over some fixed time period. Each row holds all the transition probabilities for a particular starting credit rating. The `transitionMatrix` input sets the TransitionMatrix on page 6-0 property.

The top row holds the probabilities for a counterparty that starts at the highest rating (such as `AAA`). The bottom row holds the probabilities for a counterparty starting in the default state. The bottom row may be omitted, indicating that a counterparty in default remains in default. Each row must sum to `1`.

The order of rows and columns must match the order of credit ratings defined in the `RatingLabels` parameter. The last column holds the probability of default for each of the ratings. If `RatingLabels` are unspecified, the default rating labels are: `"AAA","AA","A","BBB","BB","B","CCC","D"`.

Note The `creditMigrationCopula` model simulates the changes in portfolio value over a fixed time period (for example, one year). The `migrationValues` and `transitionMatrix` must be specific to a particular time period.

Data Types: `double`

`LGD` — Loss given default

numeric vector with elements from `0` through `1`

Loss given default, specified as a `NumCounterparties`-by-`1` numeric vector with elements from `0` through `1`, representing the fraction of exposure that is lost when a counterparty defaults. `LGD` is defined as (1 âˆ' *Recovery*). For example, an `LGD` of 0.6 implies a 40% recovery rate in the event of a default. The `LGD` input sets the Portfolio on page 6-0 property.

`LGD` can alternatively be specified as a `NumCounterparties`-by-`2` matrix, where the first column holds the LGD mean values and the 2nd column holds the LGD standard deviations. Then, in the case of default, LGD values are drawn randomly from a beta distribution with provided parameters for the defaulting counterparty.

Valid open intervals for LGD mean and standard deviation are:

- For the first column, the mean values are between `0` and `1`.
- For the second column, the LGD standard deviations are between `0` and `sqrt(m*(1-m))`.

Data Types: `double`

`Weights` — Weights variable name
array of factor and idiosyncratic weights

Factor and idiosyncratic weights, specified as a `NumCounterparties`-by-(`NumFactors + 1`) array. Each row contains the factor weights for a particular counterparty. Each column contains the weights for an underlying risk factor. The last column in `Weights` contains the idiosyncratic risk weight for each counterparty. The idiosyncratic weight represents the company-specific credit risk. The total of the weights for each counterparty (that is, each row) must sum to `1`. The `Weights` input sets the Portfolio on page 6-0 property.

For example, if a counterparty's creditworthiness was composed of 60% US, 20% European, and 20% idiosyncratic, then the `Weights` vector is `[0.6 0.2 0.2]`.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `cmc = creditMigrationCopula(migrationValues,ratings,transitionMatrix,LGD,Weights,'VaRLevel',0.99)`

`ID` — User-defined IDs for counterparties
`1:NumCounterparties` (default) | vector

User-defined IDs for counterparties, specified as the comma-separated pair consisting of `'ID'` and a `NumCounterparties`-by-`1` vector of `ID`s for each counterparty. `ID` is used to identify exposures in the `Portfolio` table and the risk contribution table. `ID` must be a numeric, a string array, or a cell array of character vectors. The `ID` name-value pair argument sets the Portfolio on page 6-0 property.

If unspecified, `ID` defaults to a numeric vector (`1:NumCounterparties`).

Data Types: `double` | `string` | `cell`

VaRLevel — Value at risk level
0.95 (default) | numeric between 0 and 1

Value at risk level (used for reporting VaR and CVaR), specified as the comma-separated pair consisting of 'VaRLevel' and a numeric between 0 and 1. The VaRLevel name-value pair argument sets the VaRLevel on page 6-0 property.

Data Types: double

FactorCorrelation — Factor correlation matrix
identity matrix (default) | correlation matrix

Factor correlation matrix, specified as the comma-separated pair consisting of 'FactorCorrelation' and a NumFactors-by-NumFactors matrix that defines the correlation between the risk factors. The FactorCorrelation name-value pair argument sets the FactorCorrelation on page 6-0 property.

If not specified, the factor correlation matrix defaults to an identity matrix, meaning that the factors are not correlated.

Data Types: double

RatingLabels — Set of all possible credit ratings
["AAA", "AA", "A", "BBB", "BB", "B", "CCC", "D"] (default) | cell array of character vectors | numeric | string

Set of all possible credit ratings, specified as the comma-separated pair consisting of 'RatingLabels' and a NumRatings-by-1 vector, where the first element is the highest credit rating and the last element is the default state. The RatingLabels name-value pair argument sets the RatingLabels on page 6-0 property.

Data Types: cell | double | string

UseParallel — Flag to use parallel processing for simulations
false (default) | logical with value of true or false

Flag to use parallel processing for simulations, specified as the comma-separated pair consisting of 'UseParallel' and a scalar value of true or false. The UseParallel name-value pair argument sets the UseParallel on page 6-0 property.

Note The 'UseParallel' property can only be set when creating a creditMigrationCopula object if you have Parallel Computing Toolbox. Once the 'UseParallel' property is set, parallel processing is used with riskContribution or simulate.

Data Types: logical

Properties

Portfolio — Details of credit portfolio
table

Details of credit portfolio, specified as a MATLAB table that contains all the portfolio data that was passed as input into the creditMigrationCopula object.

The `Portfolio` table has a column for each of the constructor inputs (`MigrationValues`, `Rating`, `LGD`, `Weights`, and `ID`). Each row of the table represents one counterparty.

For example:

```
ID    MigrationValues    Rating     LGD       Weights
__    _______________    ______    ______    ____________

1     [1x8 double]       "A"       0.6509     0.5     0.5
2     [1x8 double]       "BBB"     0.8283     0.55    0.45
3     [1x8 double]       "AA"      0.6041     0.7     0.3
4     [1x8 double]       "BB"      0.6509     0.55    0.45
5     [1x8 double]       "BBB"     0.4966     0.75    0.25
```

Data Types: `table`

`FactorCorrelation` — Correlation matrix for credit factors
matrix

Correlation matrix for credit factors, specified as a `NumFactors`-by-`NumFactors` matrix. Specify the correlation matrix by using the optional name-value pair argument `'FactorCorrelation'` when you create the `creditMigrationCopula` object.

Data Types: `double`

`RatingLabels` — Set of all possible credit ratings
cell array of character vectors, string, or numeric vector representing set of credit ratings

Set of all possible credit ratings, specified using an optional name-value input argument for `'RatingLabels'` when you create the `creditMigrationCopula` object.

Data Types: `double` | `cell` | `string`

`TransitionMatrix` — Probabilities counterparty transitions from starting credit rating to final credit rating
matrix

Probabilities that a counterparty transitions from a starting credit rating to a final credit rating, specified using the input argument `'transitionMatrix'` when you create the `creditMigrationCopula` object. The rows represent the starting credit ratings and the columns represent the final ratings. The top row corresponds to the highest rating.

The top row holds the probabilities for a counterparty that starts at the highest rating (such as `AAA`) and the bottom row holds those for a counterparty starting in the default state. The bottom row may be omitted, indicating that a counterparty in default remains in default. Each row must sum to `1`.

The order of rows and columns must match the order of credit ratings defined in the `RatingLabels` parameter. The last column holds the probability of default for each of the ratings. If `RatingLabels` are unspecified, the default rating labels are: `"AAA","AA","A","BBB","BB","B","CCC","D"`.

Data Types: `double`

`VaRLevel` — Value at Risk Level
numeric value between `0` and `1`

Value at risk level used when reporting VaR and CVaR, specified using an optional name-value pair argument `'VaRLevel'` when you create the `creditMigrationCopula` object.

Data Types: double

PortfolioValues — Portfolio values
vector

Portfolio values, specified as a 1-by-NumScenarios vector. After creating the creditMigrationCopula object, the PortfolioValues property is empty. After you invoke the simulate function, PortfolioValues is populated with the portfolio values over each scenario.

Data Types: double

UseParallel — Flag to use parallel processing for simulations
false (default) | logical with value of true or false

Flag to use parallel processing for simulations, specified using an optional name-value pair argument 'UseParallel' when you create a creditMigrationCopula object. The UseParallel name-value pair argument sets the UseParallel property.

Note The 'UseParallel' property can only be set when creating a creditMigrationCopula object if you have Parallel Computing Toolbox. Once the 'UseParallel' property is set, parallel processing is used with riskContribution or simulate.

Data Types: logical

Object Functions

simulate	Simulate credit migrations using creditMigrationCopula object
portfolioRisk	Generate portfolio-level risk measurements
riskContribution	Generate risk contributions for each counterparty in portfolio
confidenceBands	Confidence interval bands
getScenarios	Counterparty scenarios

Examples

Create a creditMigrationCopula Object Using a Four-Factor Model

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a creditMigrationCopula object with a four-factor model using creditMigrationCopula.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:
```

```
          Portfolio: [250x5 table]
  FactorCorrelation: [4x4 double]
       RatingLabels: [8x1 string]
   TransitionMatrix: [8x8 double]
           VaRLevel: 0.9500
        UseParallel: 0
    PortfolioValues: []
```

Set the `VaRLevel` to 99%.

```
cmc.VaRLevel = 0.99;
```

The `Portfolio` property contains information about migration values, ratings, LGDs and weights.

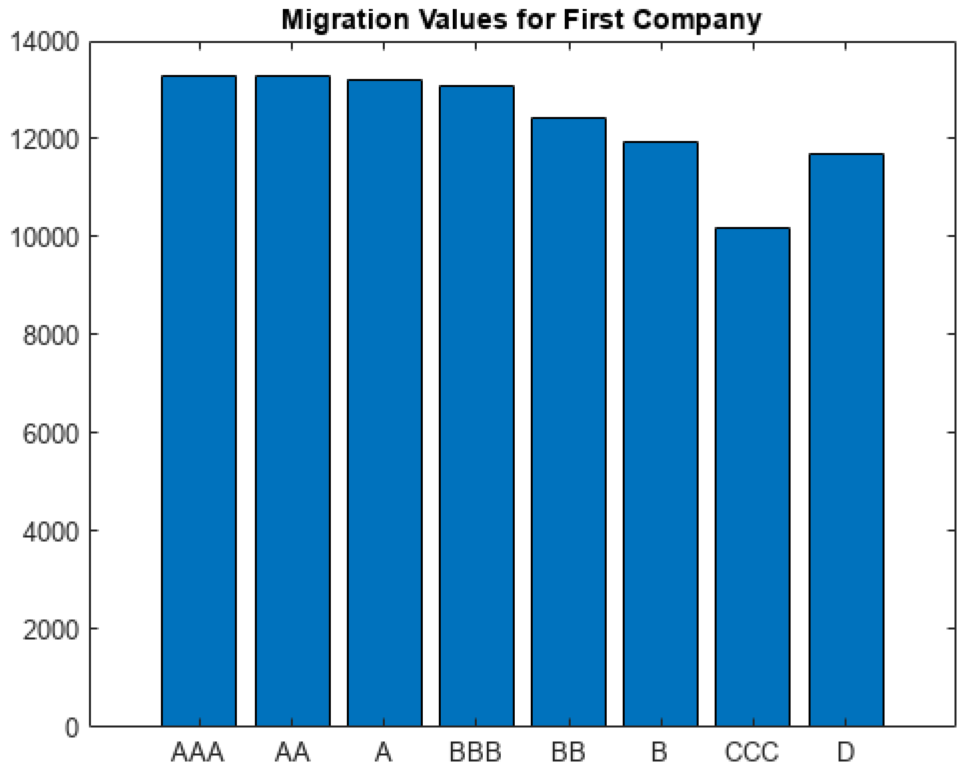
```
head(cmc.Portfolio)
```

```
    ID    MigrationValues    Rating     LGD                   Weights
    __    _______________    ______    ______    ___________________________________

    1       1x8 double       "A"       0.6509      0       0       0     0.5     0.5
    2       1x8 double       "BBB"     0.8283      0    0.55       0       0    0.45
    3       1x8 double       "AA"      0.6041      0     0.7       0       0     0.3
    4       1x8 double       "BB"      0.6509      0    0.55       0       0    0.45
    5       1x8 double       "BBB"     0.4966      0       0    0.75       0    0.25
    6       1x8 double       "BB"      0.8283      0       0       0    0.65    0.35
    7       1x8 double       "BB"      0.6041      0       0       0    0.65    0.35
    8       1x8 double       "BB"      0.4873    0.5       0       0       0     0.5
```

The columns in the migration values are in the same order of the ratings, with the default rating in the last column.

For example, these are the migration values for the first counterparty. Note that the value for default is higher than some of the non-default ratings. This is because the migration value for the default rating is a reference value (for example, face value, forward value at current rating, or other) that is multiplied by the recovery rate during the simulation to get the value of the asset in the event of default. The recovery rate is 1-`LGD` when the `LGD` input to `creditMigrationCopula` is a constant `LGD` value (the `LGD` input has one column). The recovery rate is a random quantity when the `LGD` input to `creditMigrationCopula` is specified as a mean and standard deviation for a beta distribution (the `LGD` input has two columns).

```
bar(cmc.Portfolio.MigrationValues(1,:))
xticklabels(cmc.RatingLabels)
title('Migration Values for First Company')
```



Use the `simulate` function to simulate 100,000 scenarios, and then view portfolio risk measures using the `portfolioRisk` function.

```
cmc = simulate(cmc,1e5)
```

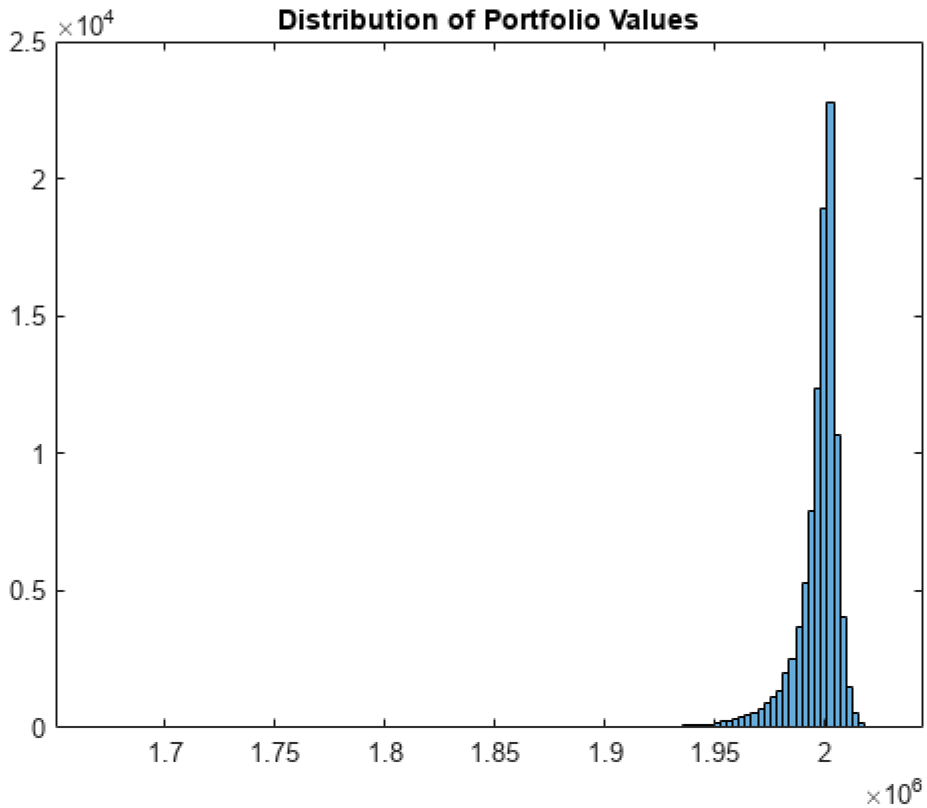
```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioValues: [2.0082e+06 1.9950e+06 1.9933e+06 2.0009e+06 1.9819e+06 1.9955e+06 1.9962e
```

```
portRisk = portfolioRisk(cmc)
```

```
portRisk=1×4 table
      EL        Std      VaR      CVaR 
    ______    _____    _____    _____

    4515.9    12963    57176    83975
```

View a histogram of the portfolio values.

```
h = histogram(cmc.PortfolioValues,125);
title('Distribution of Portfolio Values');
```



Create a `creditMigrationCopula` Object and Analyze Results

Load the saved portfolio data.

```
load CreditMigrationData.mat;
```

Scale the bond prices for portfolio positions for each bond.

```
migrationValues = migrationPrices .* numBonds;
```

Create a `creditMigrationCopula` object with a four-factor model using `creditMigrationCopula`.

```
cmc = creditMigrationCopula(migrationValues,ratings,transMat,...
lgd,weights,'FactorCorrelation',factorCorr)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
```

```
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9500
          UseParallel: 0
      PortfolioValues: []
```

Set the `VaRLevel` to 99%.

```
cmc.VaRLevel = 0.99;
```

Use the `simulate` function to simulate 100,000 scenarios, and then view portfolio risk measures by using the `portfolioRisk` function.

```
cmc = simulate(cmc,1e5)
```

```
cmc = 
  creditMigrationCopula with properties:

            Portfolio: [250x5 table]
    FactorCorrelation: [4x4 double]
         RatingLabels: [8x1 string]
     TransitionMatrix: [8x8 double]
             VaRLevel: 0.9900
          UseParallel: 0
      PortfolioValues: [2.0082e+06 1.9950e+06 1.9933e+06 2.0009e+06 1.9819e+06 1.9955e+06 1.9962e
```

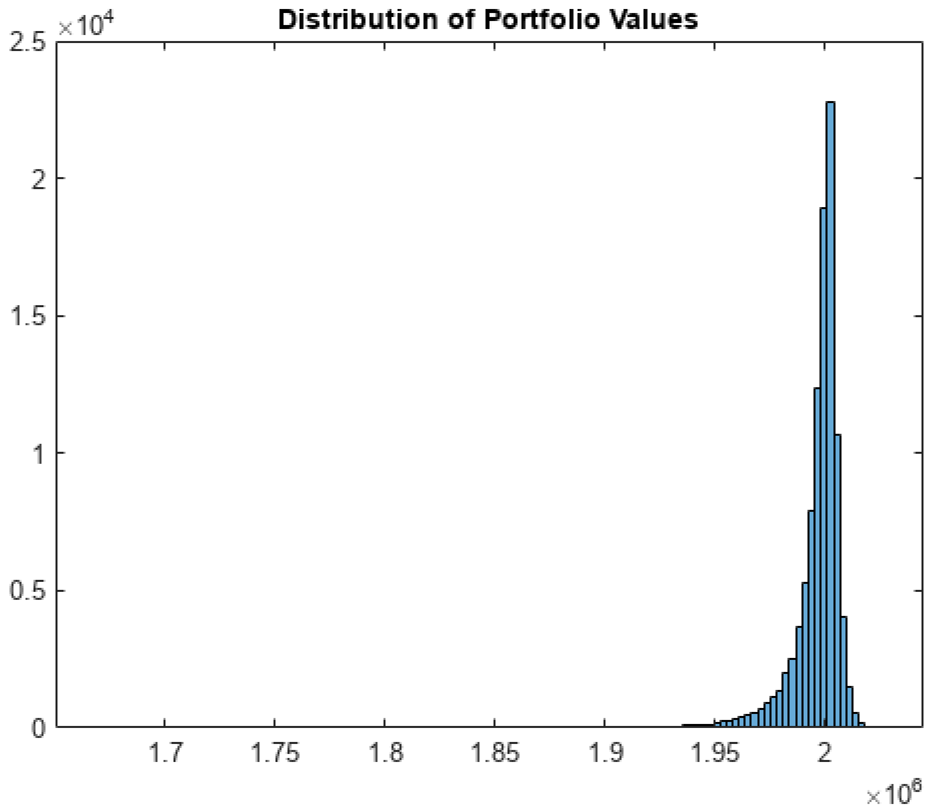
```
portRisk = portfolioRisk(cmc)
```

portRisk=*1×4 table*

EL	Std	VaR	CVaR
4515.9	12963	57176	83975

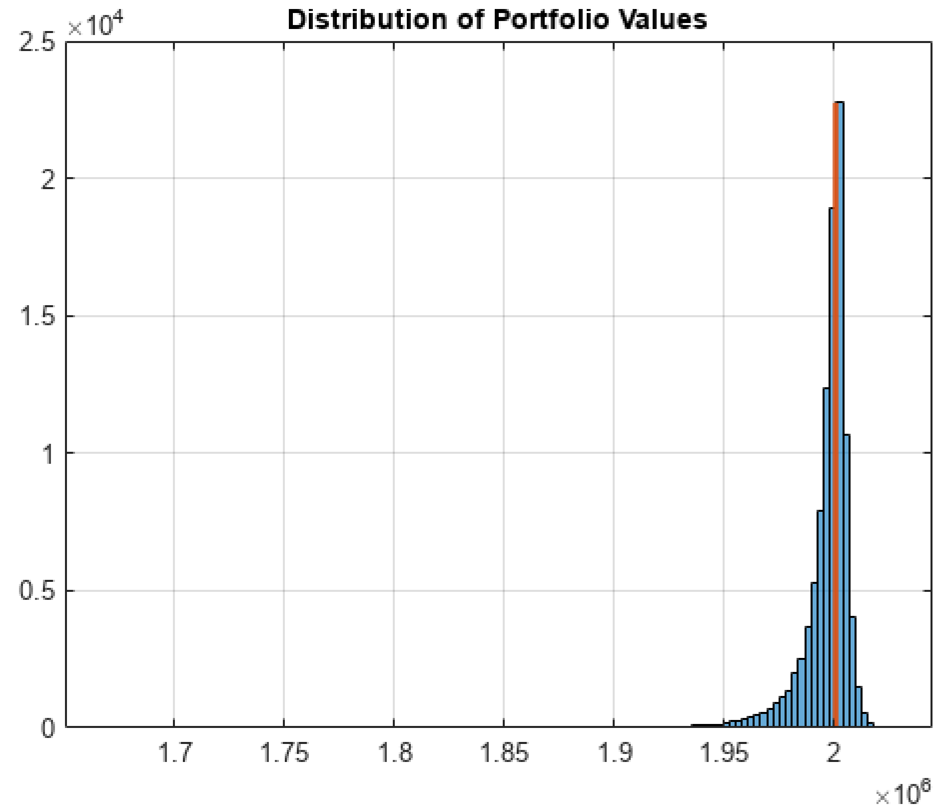
View a histogram of the portfolio values.

```
h = histogram(cmc.PortfolioValues,125);
title('Distribution of Portfolio Values');
```



Overlay the value that the portfolio takes if all counterparties maintained their current credit ratings.

```
CurrentRatingValue = portRisk.EL + mean(cmc.PortfolioValues);
     hold on
     plot([CurrentRatingValue CurrentRatingValue],[0 max(h.Values)],...
         'LineWidth',2);
     grid on
```



Version History

Introduced in R2017a

References

[1] Crouhy, M., Galai, D., and Mark, R. "A Comparative Analysis of Current Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 59 – 117.

[2] Gordy, M. "A Comparative Anatomy of Credit Risk Models." *Journal of Banking and Finance.* Vol. 24, 2000, pp. 119 – 149.

[3] Gupton, G., Finger, C., and Bhatia, M. *"CreditMetrics – Technical Document."* J. P. Morgan, New York, 1997.

[4] Jorion, P. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[5] Löffler, G., and Posch, P. *Credit Risk Modeling Using Excel and VBA.* Wiley Finance, 2007.

[6] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management: Concepts, Techniques, and Tools.* Princeton University Press, 2005.

See Also

`table` | `simulate` | `portfolioRisk` | `riskContribution` | `confidenceBands` | `getScenarios` | `creditDefaultCopula` | `nearcorr`

Topics

"creditMigrationCopula Simulation Workflow" on page 4-10
"One-Factor Model Calibration" on page 4-64
"Credit Rating Migration Risk" on page 1-10

External Websites

Parallel Computing with MATLAB (53 min 27 sec)

developmentTriangle

Create `developmentTriangle` object

Description

Use this workflow to generate projected ultimate claims for a `developmentTriangle`:

1. Load or generate the claims data for the development triangle.
2. Create a `developmentTriangle` object.
3. Use `view` to display the `developmentTriangle` data and use `claimsPlot` to plot the reported claims.
4. Use `linkRatios` to compute the link ratio factors (development factors or age-to-age factors) and use `linkRatioAverages` to calculate averages from those factors. Also, you can plot link ratios using `linkRatiosPlot`.
5. Use `cdfSummary` to calculate the cumulative development factors (CDFs) and the percentage of total claims.
6. Use `ultimateClaims` to calculate the projected ultimate claims.
7. Use `fullTriangle` to display the development triangle that includes ultimate claims.

Creation

Syntax

```
dT = developmentTriangle(data)
dT = developmentTriangle( ___ ,Name,Value)
```

Description

`dT = developmentTriangle(data)` creates a `developmentTriangle` object using `data`. You can plot `dT` using `claimsPlot`.

`dT = developmentTriangle( ___ ,Name,Value)` sets properties on page 6-574 using name-value pair arguments. Specify one or more name-value pair arguments after the input argument in the previous syntax. For example, `dT_reported = developmentTriangle(data,'Origin','AccidentYear','Development','DevelopmentYear','Claims','ReportedClaims')`.

Input Arguments

`data` — Claims data
table

Claims data, specified as a table with at least three columns. If you specify `data` as a three-column table and do not specify name-value pair arguments for `'Origin'`, `'Development'` and `'Claims'`, the software obtains origin years from the first column, development years from the second column, and claims from the third column by default.

Data Types: `table`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `dT_reported = developmentTriangle(data,'Origin','AccidentYear','Development','DevelopmentYear','Claims','ReportedClaims')`

`Origin` — Name of the column containing the origin years
first column of `data` table (default) | character vector | string

Name of the column containing the origin years, specified as the comma-separated pair consisting of `'Origin'` and a character vector or string.

Data Types: `char` | `string`

`Development` — Name of column containing development years
second column of `data` table (default) | character vector | string

Name of column containing development years, specified as the comma-separated pair consisting of `'Development'` and a character vector or string.

Data Types: `char` | `string`

`Claims` — Name of column containing claims periods
third column of `data` table (default) | character vector | string

Name of column containing claims periods, specified as the comma-separated pair consisting of `'Claims'` and a character vector or string.

Data Types: `double`

`Cumulative` — Flag to indicate if data is cumulative or incremental
`true` (cumulative) (default) | logical with a value of `true` or `false`

Flag to indicate if data is cumulative or incremental, specified as the comma-separated pair consisting of `'Cumulative'` and a scalar logical value.

Data Types: `logical`

Properties

`Origin` — Name of column containing origin years
first column of `data` table (default) | cell array

Name of column containing origin years, returned as a cell array.

Data Types: `cell`

`Development` — Name of column containing development years
second column of `data` table (default) | cell array

Name of column containing development years, returned as a cell array.

Data Types: `cell`

`Claims` — Name of column containing claims period
third column of `data` table (default) | vector

Name of column containing claims period, returned as a vector.

Data Types: `double`

`LatestDiagonal` — Latest claim values for each `Origin` period
vector

Latest claim values for each `Origin` period, returned as a vector.

Data Types: `double`

`Description` — User-defined description
`" "` (default) | string

User-defined description, returned as a string.

Data Types: `string`

`SelectedLinkRatio` — Selected link ratios for CDF calculations
simple average (default) | vector

Selected link ratios for the CDF calculations, returned as a vector.

Data Types: `double`

`TailFactor` — Tail factor constant
`1` (default) | numeric

Tail factor constant, returned as a numeric.

Data Types: `double`

Object Functions

view	Display developmentTriangle object
linkRatios	Compute link ratios for developmentTriangle object
linkRatioAverages	Compute link ratio averages for developmentTriangle object
cdfSummary	Compute CDFs to ultimate claims for developmentTriangle object
ultimateClaims	Compute ultimate claims for developmentTriangle object
fullTriangle	Display full development triangle including ultimate claims
linkRatiosPlot	Plot link ratios for development triangle
claimsPlot	Plot claims for development triangle

Examples

Create `developmentTriangle` Object

Create a `developmentTriangle` object using simulated claims data.

```
load InsuranceClaimsData.mat;
disp(data)

    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9
       2010             24                 4635          3371.2
       2010             36               4866.8          4079.1
       2010             48               4964.1            4487
       2010             60               5013.7          4711.4
       2010             72               5038.8          4805.6
       2010             84                 5059          4853.7
       2010             96               5074.1          4877.9
       2010            108               5084.3          4887.7
       2010            120               5089.4          4892.6
       2011             12                 3968          2055.5
       2011             24               4682.3          3638.3
       2011             36               4963.2          4365.9
       2011             48               5062.5          4758.9
       2011             60               5113.1          4949.2
       2011             72               5138.7          5048.2
       2011             84               5154.1          5098.7
       2011             96               5169.6          5124.2
       2011            108               5179.9          5134.4
       2012             12                 4217          2242.4
       2012             24               5060.4          3946.7
       2012             36                 5364          4696.6
       2012             48               5508.9          5119.3
       2012             60               5558.4          5324.1
       2012             72               5586.2          5430.5
       2012             84               5608.6          5484.8
       2012             96               5625.4          5512.3
       2013             12               4374.2          2373.8
       2013             24               5205.3          4130.4
       2013             36               5517.7          4915.2
       2013             48               5661.1          5357.6
       2013             60               5740.4          5571.9
       2013             72               5780.6          5677.8
       2013             84               5803.7          5728.9
       2014             12               4499.7          2421.8
       2014             24               5309.6          4189.6
       2014             36               5628.2          4985.6
       2014             48               5785.8          5434.3
       2014             60               5849.4          5651.7
       2014             72               5878.7          5759.1
       2015             12               4530.2          2484.1
       2015             24               5300.4          4272.6
       2015             36               5565.4          5084.4
       2015             48               5715.7          5541.9
       2015             60               5772.8          5763.6
       2016             12               4572.6          2481.7
       2016             24               5304.2          4218.9
       2016             36               5569.5          5020.5
       2016             48               5714.3          5472.4
       2017             12               4680.6          2577.9
       2017             24               5523.1          4382.4
       2017             36               5854.4          5171.2
```

```
        2018              12              4696.7              2580
        2018              24              5495.1            4386.1
        2019              12              4945.9            2764.8
```

Use `developmentTriangle` to convert the data to a development triangle, which is the standard form for representing claims data.

```
dT = developmentTriangle(data)
```

```
dT = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.0010
               SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.0010
```

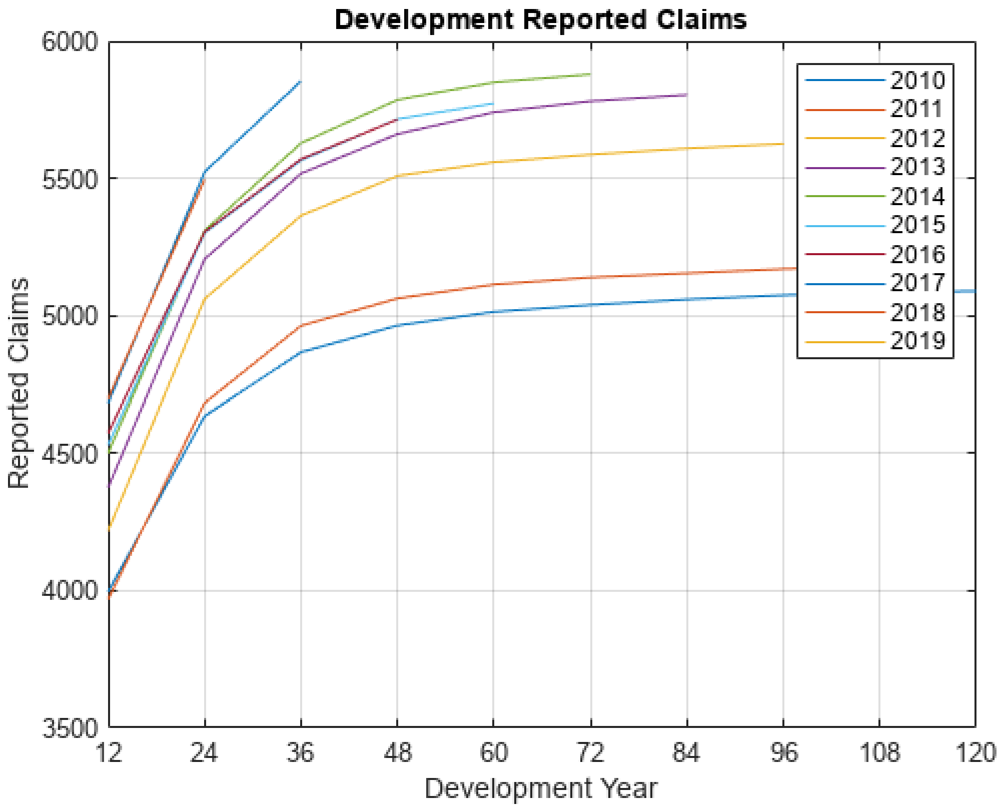
Use the `view` function to display the `developmentTriangle` contents in table form. Each row represents an origin period and each column represents a development period.

```
developmentTriangleTable = view(dT)
```

```
developmentTriangleTable=10×10 table
               12        24        36        48        60        72        84        96        108
             ______    ______    ______    ______    ______    ______    ______    ______    _____

    2010     3995.7      4635    4866.8    4964.1    5013.7    5038.8      5059    5074.1    5084
    2011       3968    4682.3    4963.2    5062.5    5113.1    5138.7    5154.1    5169.6    5179
    2012       4217    5060.4      5364    5508.9    5558.4    5586.2    5608.6    5625.4      Na
    2013     4374.2    5205.3    5517.7    5661.1    5740.4    5780.6    5803.7       NaN      Na
    2014     4499.7    5309.6    5628.2    5785.8    5849.4    5878.7       NaN       NaN      Na
    2015     4530.2    5300.4    5565.4    5715.7    5772.8       NaN       NaN       NaN      Na
    2016     4572.6    5304.2    5569.5    5714.3       NaN       NaN       NaN       NaN      Na
    2017     4680.6    5523.1    5854.4       NaN       NaN       NaN       NaN       NaN      Na
    2018     4696.7    5495.1       NaN       NaN       NaN       NaN       NaN       NaN      Na
    2019     4945.9       NaN       NaN       NaN       NaN       NaN       NaN       NaN      Na
```

To visualize the development triangles, use `plot`.

```
plot(table2array(developmentTriangleTable)');
xticklabels(developmentTriangleTable.Properties.VariableNames)
xlabel('Development Year')
ylabel('Reported Claims')
title('Development Reported Claims')
legend(developmentTriangleTable.Properties.RowNames)
grid on
```



Version History

Introduced in R2020b

See Also

`chainLadder` | `expectedClaims` | `bornhuetterFerguson`

Topics

“Mean Square Error of Prediction for Estimated Ultimate Claims” on page 4-161
“Bootstrap Using Chain Ladder Method” on page 4-168
“Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

esbacktest

Create `esbacktest` object to run suite of table-based expected shortfall (ES) backtests by Acerbi and Szekely

Description

The general workflow is:

1. Load or generate the data for the ES backtesting analysis.
2. Create an `esbacktest` object. For more information, see “Create esbacktest” on page 6-579 and “Properties” on page 6-582.
3. Use the `summary` function to generate a summary report for the number of observations, expected, and observed average severity ratio.
4. Use the `runtests` function to run all tests at once.
5. For additional test details, run the following individual tests:

   - `unconditionalNormal` — Unconditional ES backtest assuming returns distribution is normal
   - `unconditionalT` — Unconditional ES backtest assuming returns distribution is `t`

   For more information, see “Overview of Expected Shortfall Backtesting” on page 2-20.

Creation

Syntax

```
ebt = esbacktest(PortfolioData,VaRData,ESData)
ebt = esbacktest( ___ ,Name,Value)
```

Description

`ebt = esbacktest(PortfolioData,VaRData,ESData)` creates an `esbacktest` (`ebt`) object using portfolio outcomes data and corresponding value-at-risk (VaR) and ES data. The `ebt` object has the following properties:

- PortfolioData on page 6-0 — `NumRows`-by-`1` numeric array containing a copy of the `PortfolioData`
- VaRData on page 6-0 — `NumRows`-by-`NumVaRs` numeric array containing a copy of the `VaRData`
- ESData on page 6-0 — `NumRows`-by-`NumVaRs` numeric array containing a copy of the `ESData`
- PortfolioID on page 6-0 — String containing the `PortfolioID`
- VaRID on page 6-0 — `1`-by-`NumVaRs` string vector containing the `VaRID`s for the corresponding columns in `VaRData`
- VaRLevel on page 6-0 — `1`-by-`NumVaRs` numeric array containing the `VaRLevel`s for the corresponding columns in `VaRData`

Note

- Test results from esbacktest are only approximate since no distribution information is passed as input. When distribution information is available, use esbacktestbysim; in particular, the minimally biased test is recommended (see `minBiasAbsolute` and `minBiasRelative`).
- The simulation of critical values assumes a mean of 0 for the underlying distribution. The critical values are sensitive to the mean of the underlying distribution. If the ES prediction is based on distributions with means significantly away from 0, the critical values in `esbacktest` will be unreliable.
- The required input arguments for `PortfolioData`, `VaRData`, and `ESData` must all be in the same units. These arguments can be expressed as returns or as profits and losses. There are no validations in the `esbacktest` object regarding the units of these arguments.
- If there are missing values (`NaN`s) in `PortfolioData`, `VaRData`, and `ESData`, the row of data is discarded before applying the tests. Therefore, a different number of observations are reported for models with a different number of missing values. The reported number of observations equals the original number of rows minus the number of missing values. To determine if there are discarded rows, use the `'Missing'` column of the `summary` report.
- Because the critical values are precomputed, only certain numbers of observations, VaR levels, and test levels are supported.
  - The number of observations (number of rows in the data minus the number of missing values) must be from 200 through 5000.
  - The `VaRLevel` input argument must be between `0.90` and `0.999`; the default is `0.95`.
  - The `TestLevel` (test confidence level) input argument for the `runtests`, `unconditionalNormal`, and `unconditionalT` functions must be between `0.5` and `0.9999`; the default is `0.95`.

`ebt = esbacktest( ___ ,Name,Value)` sets Properties on page 6-582 using name-value pairs and any of the arguments in the previous syntax. For example, `ebt = esbacktest(PortfolioData,VaRData,ESData,'VaRID','TotalVaR','VaRLevel',.999)`. You can specify multiple name-value pairs as optional name-value pair arguments.

Input Arguments

`PortfolioData` — Portfolio outcomes data

`NumRows`-by-`1` numeric array | `NumRows`-by-`1` numeric columns table | `NumRows`-by-`1` numeric columns timetable

Portfolio outcomes data, specified as a `NumRows`-by-`1` numeric array, `NumRows`-by-`1` numeric columns table, or a `NumRows`-by-`1` timetable with a numeric column containing portfolio outcomes data. The `PortfolioData` input argument sets the PortfolioData on page 6-0 property.

Note `PortfolioData` must be in the same units as `VaRData` and `ESData`. `PortfolioData`, `VaRData`, and `ESData` can be expressed as returns or as profits and losses. There are no validations in the `esbacktest` object regarding the units of portfolio, VaR, and ES data.

Data Types: `double` | `table` | `timetable`

VaRData — Value-at-risk (VaR) data
NumRows-by-NumVaRs numeric array | NumRows-by-NumVaRs table with numeric columns | NumRows-by-NumVaRs timetable with numeric columns

Value-at-risk (VaR) data, specified as a NumRows-by-NumVaRs numeric array, NumRows-by-NumVaRs numeric columns table, or NumRows-by-NumVaRs timetable with numeric columns. The VaRData input argument sets the VaRData on page 6-0 property.

Negative VaRData values are allowed. However, negative VaR values indicate a highly profitable portfolio that cannot lose money at the given VaR confidence level. The worst-case scenario at the given confidence level is still a profit.

Note VaRData must be in the same units as PortfolioData and ESData. VaRData, PortfolioData, and ESData can be expressed as returns or as profits and losses. There are no validations in the esbacktest object regarding the units of portfolio, VaR, and ES data.

Data Types: double | table | timetable

ESData — Expected shortfall data
NumRows-by-NumVaRs positive numeric array | NumRows-by-NumVaRs table with positive numeric columns | NumRows-by-NumVaRs timetable with positive numeric columns

Expected shortfall data, specified as a NumRows-by-NumVaRs positive numeric array, NumRows-by-NumVaRs table with positive numeric columns, or NumRows-by-NumVaRs timetable with positive numeric columns containing ES data. The ESData input argument sets the ESData on page 6-0 property.

Note ESData must be in the same units as PortfolioData and VaRData. ESData, PortfolioData, and VaRData can be expressed as returns or as profits and losses. There are no validations in the esbacktest object regarding the units of portfolio, VaR, and ES data.

Data Types: double | table | timetable

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: ebt = esbacktest(PortfolioData,VaRData,ESData,'VaRID','TotalVaR','VaRLevel',.999)

PortfolioID — User-defined ID
character vector | string

User-defined ID for PortfolioData input, specified as the comma-separated pair consisting of 'PortfolioID' and a character vector or string. The PortfolioID name-value pair argument sets the PortfolioID on page 6-0 property.

If PortfolioData is a numeric array, the default value for PortfolioID is 'Portfolio'. If PortfolioData is a table, PortfolioID is set to the corresponding variable name in the table, by default.

Data Types: char | string

VaRID — VaR identifier
character vector | cell array of character vectors | string | string array

VaR identifier for VaRData columns, specified as the comma-separated pair consisting of 'VaRID' and a character vector, cell array of character vectors, string, or string array.

Multiple VaRID values are specified using a 1-by-NumVaRs (or NumVaRs-by-1) cell array of character vectors or a string vector with user-defined IDs for the VaRData columns. A single VaRID identifies a VaRData column and the corresponding ESData column. The VaRID name-value pair argument sets the VaRID on page 6-0 property.

If NumVaRs = 1, the default value for VaRID is 'VaR'. If NumVaRs > 1, the default value is 'VaR1', 'VaR2', and so on. If VaRData is a table, 'VaRID' is set by default to the corresponding variable names in the table.

Data Types: char | cell | string

VaRLevel — VaR confidence level
0.95 (default) | numeric between 0.90 and 0.999

VaR confidence level, specified as the comma-separated pair consisting of 'VaRLevel' and a numeric value between 0.90 and 0.999 or a 1-by-NumVaRs (or NumVaRs-by-1) numeric array. The VaRLevel name-value pair argument sets the VaRLevel on page 6-0 property.

Note When specifying a VarLevel > 99%, ensure that the number of observations is sufficient to generate an appropriate critical value. In addition, when running a test, use a TestLevel > 95%. For very high VaR levels (for example, VarLevel > 99%) and a relatively small number of observations, the probability of VaR failures is very small and the distribution of the test statistic has a discrete nature, leading to unexpected non-monotonicity around some critical values. Larger number of observations and higher test confidence levels preserve the expected behavior of critical values when the VarLevel is very high.

Data Types: double

Properties

PortfolioData — Portfolio data for ES backtesting analysis
numeric array

Portfolio data for ES backtesting analysis, specified as a NumRows-by-1 numeric array containing a copy of the portfolio data.

Data Types: double

VaRData — VaR data for ES backtesting analysis
numeric array

VaR data for ES backtesting analysis, specified as a `NumRows`-by-`NumVaRs` numeric array containing a copy of the VaR data.

Data Types: `double`

`ESData` — Expected shortfall data for ES backtesting analysis
numeric array

Expected shortfall data for ES backtesting analysis, specified as a `NumRows`-by-`NumVaRs` numeric array containing a copy of the `ESData`.

Data Types: `double`

`PortfolioID` — Portfolio identifier
string

Portfolio identifier, specified as a string.

Data Types: `string`

`VaRID` — VaR identifier
string | string array

VaR identifier, specified as a `1`-by-`NumVaRs` string array containing the VaR IDs for the corresponding columns in `VaRData`.

Data Types: `string`

`VaRLevel` — VaR level
numeric array with values between `0.90` and `0.999`

VaR level, specified as a `1`-by-`NumVaRs` numeric array with values from `0.90` through `0.999`, containing the VaR levels for the corresponding columns in `VaRData`.

Data Types: `double`

`esbacktest` Property	Set or Modify Property from Command Line Using `esbacktest`	Modify Property Using Dot Notation
`PortfolioData`	Yes	No
`VaRData`	Yes	No
`ESData`	Yes	No
`PortfolioID`	Yes	Yes
`VaRID`	Yes	Yes
`VaRLevel`	Yes	Yes

Object Functions

summary	Basic expected shortfall (ES) report on failures and severity
runtests	Run all expected shortfall (ES) backtests for esbacktest object
unconditionalNormal	Unconditional expected shortfall (ES) backtest by Acerbi-Szekely with critical values for normal distributions
unconditionalT	Unconditional expected shortfall (ES) backtest by Acerbi-Szekely with critical values for t distributions

Examples

Create esbacktest Object and Run ES Backtests for Single VaRLevel at 99.9%

`esbacktest` takes in portfolio outcomes data, the corresponding value-at-risk (VaR) data, and the expected shortfall (ES) data and returns an `esbacktest` object.

Create an `esbacktest` object.

```
load ESBacktestData
ebt = esbacktest(Returns,VaRModel1,ESModel1,'VaRLevel',VaRLevel)
```

```
ebt = 
  esbacktest with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x1 double]
           ESData: [1966x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9750
```

`ebt`, the `esbacktest` object, contains a copy of the given portfolio data (`PortfolioData` property), the given VaR data (`VaRData` property), and the given ES data (`ESData`) property. The object also contains all combinations of portfolio ID, VaR ID, and VaR level to be tested (`PortfolioID`, `VaRID`, and `VaRLevel` properties).

Run the tests using the `ebt` object.

```
runtests(ebt)
```

```
ans=1×5 table
    PortfolioID    VaRID    VaRLevel    UnconditionalNormal    UnconditionalT
    ___________    _____    ________    ___________________    ______________

    "Portfolio"    "VaR"     0.975            reject               reject
```

Change the `PortfolioID` and `VaRID` properties using dot notation. For more information on creating an `esbacktest` object, see `esbacktest`.

```
ebt.PortfolioID = 'S&P';
ebt.VaRID = 'Normal at 97.5%';
disp(ebt)
```

```
  esbacktest with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x1 double]
           ESData: [1966x1 double]
      PortfolioID: "S&P"
            VaRID: "Normal at 97.5%"
         VaRLevel: 0.9750
```

Run all tests using the updated `esbacktest` object.

```
runtests(ebt)
```

ans=*1×5 table*

PortfolioID	VaRID	VaRLevel	UnconditionalNormal	UnconditionalT
"S&P"	"Normal at 97.5%"	0.975	reject	reject

Version History

Introduced in R2017b

References

[1] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall*. MSCI Inc. December, 2014.

[2] Basel Committee on Banking Supervision. "*Minimum Capital Requirements for Market Risk*". January, 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

`summary` | `runtests` | `unconditionalNormal` | `unconditionalT` | `esbacktestbysim` | `table` | `timetable` | `varbacktest`

Topics

"Expected Shortfall (ES) Backtesting Workflow with No Model Distribution Information" on page 2-30
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

esbacktestbyde

Create `esbacktestbyde` object to run suite of Du and Escanciano expected shortfall (ES) backtests

Description

The general workflow is:

1. Load or generate the data for the ES backtesting analysis.
2. Create an `esbacktestbyde` object. For more information, see Create esbacktestbyde on page 6-586 and Properties on page 6-589.
3. Use the `summary` function to generate a summary report on the failures and severities.
4. Use the `runtests` function to run all tests at once.
5. For additional test details, run the following individual tests:
   - `unconditionalDE` — Unconditional ES backtest by Du-Escanciano
   - `conditionalDE` — Conditional ES backtest by Du-Escanciano
6. `simulate` — Simulate critical values for test statistics

For more information, see “Overview of Expected Shortfall Backtesting” on page 2-20 and “Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63.

Creation

Syntax

```
ebtde = esbacktestbyde(PortfolioData,DistributionName)
ebtde = esbacktestbyde( ___ ,Name,Value)
```

Description

`ebtde = esbacktestbyde(PortfolioData,DistributionName)` creates an `esbacktestbyde` (`ebtde`) object using portfolio outcomes data and model distribution information. The `esbacktestbyde` object has the following properties:

- PortfolioData on page 6-0 — `NumRows`-by-`1` numeric array or `NumRows`-by-`1` table or timetable with a numeric column containing portfolio outcomes data.
- VaRData on page 6-0 — Computed VaR data using distribution information from `PortfolioData`, returned as a `NumRows`-by-`NumVaRs` numeric array.
- ESData on page 6-0 — Computed ES data using distribution information from `PortfolioData`, returned as a `NumRows`-by-`NumVaRs` numeric array.
- Distribution on page 6-0 — Model distribution information, returned as a structure.
- PortfolioID on page 6-0 — User-defined portfolio ID.
- VaRID on page 6-0 — VaRIDs for the corresponding column in `PortfolioData`.

- VaRLevel on page 6-0 — VaRLevel for the corresponding columns in `PortfolioData`.

`ebtde = esbacktestbyde( ___ ,Name,Value)` sets Properties on page 6-582 using name-value pairs and any of the arguments in the previous syntax. For example, `ebtde = esbacktestbyde(PortfolioData,DistributionName,'VaRID','TotalVaR','VaRLevel',.99)`. You can specify multiple name-value pairs as optional name-value pair arguments.

Input Arguments

`PortfolioData` — Portfolio outcome data

`NumRows`-by-1 numeric array | `NumRows`-by-1 table of numeric columns | `NumRows`-by-1 timetable with one numeric column

Portfolio outcome data, specified as a `NumRows`-by-1 numeric array, `NumRows`-by-1 table of numeric columns, or a `NumRows`-by-1 timetable with a numeric column containing portfolio outcomes data. The `PortfolioData` input argument sets the PortfolioData on page 6-0 property.

Unlike other ES backtesting classes, the `esbacktestbyde` does not require VaR data or ES data inputs. The distribution information from `PortfolioData` is sufficient to run the tests. `esbacktestbyde` uses the distribution information to apply the cumulative distribution function to the portfolio data and map it into the (0,1) interval. The ES backtests are applied to the mapped data.

Note Before applying the tests, the function discards rows with missing values (`NaN`) in the `PortfolioData` or `Distribution` parameters. Therefore, the reported number of observations equals the original number of rows minus the number of missing values.

Data Types: `double` | `table` | `timetable`

`DistributionName` — Model distribution name

character vector with a value of `'normal'` or `'t'` | string with a value of `"normal"` of `"t"`

Model distribution name for ES backtesting analysis, specified as a character vector with a value of `'normal'` or `'t'` or string with a value of `"normal"` or `"t"`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ebtde = esbacktestbyde(PortfolioData,"t",'DegreesOfFreedom',10,'Location',Mu,'Scale',Sigma,'PortfolioID',"S&P",'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],'VaRLevel',VaRLevel)`

Name-Value Pairs for 'normal' or 't' Distributions

`PortfolioID` — User-defined ID

character vector | string

User-defined ID for `PortfolioData` input, specified as the comma-separated pair consisting of `'PortfolioID'` and a character vector or string. The `'PortfolioID'` name-value pair argument sets the PortfolioID on page 6-0 property.

If `PortfolioData` is a numeric array, the default value for `PortfolioID` is `'Portfolio'`. If `PortfolioData` is a table or timetable, `PortfolioID` is set to the corresponding variable name in the table, by default.

Data Types: `char` | `string`

`VaRID` — VaR identifier
character vector | cell array of character vectors | string | string array

VaR identifier for the VaR model, specified as the comma-separated pair consisting of `'VaRID'` and a character vector, cell array of character vectors, string, or string array.

You can specify multiple `VaRID` values by using a `1`-by-`NumVaRs` (or `NumVaRs`-by-`1`) cell array of character vectors or a string vector with user-defined IDs for the different VaR levels The `'VaRID'` name-value pair argument sets the VaRID on page 6-0 property.

If `NumVaRs = 1`, the default value for `VaRID` is `'VaR'`. If `NumVaRs > 1`, the default value is `'VaR1'`, `'VaR2'`, and so on.

Data Types: `char` | `cell` | `string`

`VaRLevel` — VaR confidence level
`0.95` (default) | numeric between `0` and `1`

VaR confidence level, specified as the comma-separated pair consisting of `'VaRLevel'` and a scalar numeric value between `0` and `1` or a `1`-by-`NumVaRs` (or `NumVaRs`-by-`1`) numeric array. The `'VaRLevel'` name-value pair argument sets the VaRLevel on page 6-0 property.

Data Types: `double`

`Simulate` — Indicates if simulation for statistical significance of tests runs
`true` (default) | scalar logical with a value of `true` or `false`

Indicates if simulation for statistical significance of tests runs when an `esbacktestbyde` object is created, specified as the comma-separated pair consisting of `'Simulate'` and a scalar logical value.

Data Types: `logical`

Name-Value Pairs for 'normal' Distributions

`Mean` — Means for the normal distribution
`0` (default) | vector

Means for the normal distribution, specified as the comma-separated pair consisting of `'Mean'` and a `NumRows`-by-`1` vector. This parameter is used only when `DistributionName` is `'normal'`.

Data Types: `double`

`StandardDeviation` — Standard deviations
`1` (default) | positive vector

Standard deviations, specified as the comma-separated pair consisting of `'StandardDeviation'` and a `NumRows`-by-`1` positive vector. This parameter is only used when `DistributionName` is `"normal"`.

Data Types: double

Name-Value Pairs for 't' Distributions

DegreesOfFreedom — Degrees of freedom for 't' distribution

scalar integer ≥ 3

Degrees of freedom for 't' distribution, specified as the comma-separated pair consisting of 'DegreesOfFreedom' and a scalar integer ≥ 3.

Note You must set this name-value parameter when DistributionName is 't'.

Data Types: double

Location — Location parameters for 't' distribution

0 (default) | vector

Location parameters for 't' distribution, specified as the comma-separated pair consisting of 'Location' and a NumRows-by-1 vector. This parameter is used only when DistributionName is 't'.

Data Types: double

Scale — Scale parameters for 't' distribution

1 (default) | positive vector

Scale parameters for 't' distribution, specified as the comma-separated pair consisting of 'Scale' and a NumRows-by-1 positive vector. This parameter is used only when DistributionName is 't'.

Data Types: double

Properties

PortfolioData — Portfolio data for ES backtesting analysis

numeric array

Portfolio data for ES backtesting analysis, returned as a NumRows-by-1 numeric array containing a copy of the portfolio data.

Data Types: double

VaRData — VaR data computed using distribution information

numeric array

VaR data computed using distribution information, returned as a NumRows-by-NumVaRs numeric array.

Data Types: double

ESData — ES data computed using distribution information

numeric array

ES data computed using distribution information, returned as a NumRows-by-NumVaRs numeric array.

Data Types: double

`Distribution` — Model distribution information
struct

Model distribution information, returned as a struct.

For a `normal` distribution, the `Distribution` structure has the fields `'Name'` (set to `normal`), `'Mean'`, and `'StandardDeviation'`, with values set to the corresponding inputs.

For a `t` distribution, the `Distribution` structure has the fields `'Name'` (set to `t`), `'DegreesOfFreedom'`, `'Location'`, and `'Scale'`, with values set to the corresponding inputs.

Data Types: `struct`

`PortfolioID` — Portfolio identifier
string

Portfolio identifier, returned as a string.

Data Types: `string`

`VaRID` — VaR identifier
string | string array

VaR identifier, returned as a `1`-by-`NumVaRs` string array containing the VaR ES model, where `NumVaRs` is the number of VaR levels.

Data Types: `string`

`VaRLevel` — VaR level
numeric array with values between `0.90` and `0.99`

VaR level, returned as a `1`-by-`NumVaRs` numeric array.

Data Types: `double`

`esbacktestbyde` Property	Set or Modify Property from Command Line Using `esbacktestbyde`	Modify Property Using Dot Notation
`PortfolioData`	Yes	No
`VaRData`	No	No
`ESData`	No	No
`Distribution`	Yes	No
`PortfolioID`	Yes	Yes
`VaRID`	Yes	Yes
`VaRLevel`	Yes	Yes

Object Functions

summary	Basic expected shortfall (ES) report on failures and severity
runtests	Run all expected shortfall (ES) backtests for esbacktestbyde object
unconditionalDE	Unconditional Du-Escanciano (DE) expected shortfall (ES) backtest
conditionalDE	Conditional Du-Escanciano (DE) expected shortfall (ES) backtest
simulate	Simulate Du-Escanciano (DE) expected shortfall (ES) test statistics

Examples

Create an esbacktestbyde Object and Run ES Backtests

Create an `esbacktestbyde` object for a *t* model with 10 degrees of freedom at three different VaR levels, and then run Du and Escanciano ES backtests.

```
load ESBacktestDistributionData.mat
    rng('default'); % For reproducibility
    ebtde = esbacktestbyde(Returns,"t",...
       'DegreesOfFreedom',T10DoF,...
       'Location',T10Location,...
       'Scale',T10Scale,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel);
  runtests(ebtde)
```

```
ans=3×5 table
    PortfolioID        VaRID         VaRLevel    ConditionalDE    UnconditionalDE
    ___________    _____________    ________    _____________    _______________

       "S&P"       "t(10) 95%"        0.95         reject            accept
       "S&P"       "t(10) 97.5%"     0.975         reject            accept
       "S&P"       "t(10) 99%"        0.99         reject            reject
```

Create Two esbacktestbyde Objects and Run ES Backtests

Create two `esbacktestbyde` objects, one with a normal distribution and another with a *t* distribution with 5 degrees of freedom, at three different VaR levels. Then run Du and Escanciano ES backtests using `runtests`.

```
load ESBacktestDistributionData.mat
    rng('default'); % For reproducibility
    ebtde1 = esbacktestbyde(Returns,"normal",...
       'Mean',NormalMean,...
       'StandardDeviation',NormalStd,...
       'PortfolioID',"S&P",...
       'VaRID',["Normal 95%","Normal 97.5%","Normal 99%"],...
       'VaRLevel',VaRLevel);
    ebtde2 = esbacktestbyde(Returns,"t",...
       'DegreesOfFreedom',T5DoF,...
       'Location',T5Location,...
       'Scale',T5Scale,...
       'PortfolioID',"S&P",...
       'VaRID',["t(5) 95%","t(5) 97.5%","t(5) 99%"],...
       'VaRLevel',VaRLevel);
```

Concatenate results in a single table.

```
t = [runtests(ebtde1);runtests(ebtde2)];
disp(t)
```

```
    PortfolioID        VaRID         VaRLevel    ConditionalDE    UnconditionalDE
    ___________    ______________    ________    _____________    _______________

       "S&P"       "Normal 95%"        0.95         reject            accept     
       "S&P"       "Normal 97.5%"     0.975         reject            reject     
       "S&P"       "Normal 99%"        0.99         reject            reject     
       "S&P"       "t(5) 95%"          0.95         reject            accept     
       "S&P"       "t(5) 97.5%"       0.975         reject            accept     
       "S&P"       "t(5) 99%"          0.99         accept            accept     
```

Version History

Introduced in R2019b

References

[1] Du, Z., and J. C. Escanciano. "Backtesting Expected Shortfall: Accounting for Tail Risk." *Management Science*. Vol. 63, Issue 4, April 2017.

[2] Basel Committee on Banking Supervision. "*Minimum Capital Requirements for Market Risk*". January 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

`summary` | `runtests` | `unconditionalDE` | `conditionalDE` | `simulate` | `esbacktestbysim`

Topics

“Workflow for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-63
“Rolling Windows and Multiple Models for Expected Shortfall (ES) Backtesting by Du and Escanciano” on page 2-72
“Expected Shortfall Estimation and Backtesting” on page 2-44
“Overview of Expected Shortfall Backtesting” on page 2-20
“ES Backtest Using Du-Escanciano Method” on page 2-24
“Comparison of ES Backtesting Methods” on page 2-26

esbacktestbysim

Create `esbacktestbysim` object to run simulation-based suite of expected shortfall (ES) backtests by Acerbi and Szekely

Description

The general workflow is:

1. Load or generate the data for the ES backtesting analysis.
2. Create an `esbacktestbysim` object. For more information, see "Create esbacktestbysim" on page 6-593.
3. Use the `summary` function to generate a summary report for the given data on the number of observations and the number of failures.
4. Use the `runtests` function to run all tests at once.
5. For additional test details, run the following individual tests:

   - `conditional` — Conditional test of Acerbi-Szekely (2014)
   - `unconditional` — Unconditional test of Acerbi-Szekely (2014)
   - `quantile` — Quantile test of Acerbi-Szekely (2014)
   - `minBiasAbsolute` — Minimally biased absolute test of Acerbi-Szekely (2017)
   - `minBiasRelative` — Minimally biased relative test of Acerbi-Szekely (2017)

   For more information, see "Overview of Expected Shortfall Backtesting" on page 2-20.

Creation

Syntax

```
ebts = esbacktestbysim(PortfolioData,VaRData,ESData,DistributionName)
ebts = esbacktestbysim( ___ ,Name,Value)
```

Description

`ebts = esbacktestbysim(PortfolioData,VaRData,ESData,DistributionName)` creates an `esbacktestbysim` (`ebts`) object and simulates portfolio outcome scenarios to compute critical values for these tests:

- `conditional`
- `unconditional`
- `quantile`
- `minBiasAbsolute`
- `minBiasRelative`

The `ebts` object has the following properties:

- PortfolioData on page 6-0 — `NumRows`-by-`1` numeric array containing a copy of the `PortfolioData`
- VaRData on page 6-0 — `NumRows`-by-`NumVaRs` numeric array containing a copy of the `VaRData`
- ESData on page 6-0 — `NumRows`-by-`NumVaRs` numeric array containing a copy of the `ESData`
- Distribution on page 6-0 — Structure containing the model information, including model distribution name and distribution parameters. For example, for a normal distribution, `Distribution` has fields `'Name'`, `'Mean'`, and `'StandardDeviation'`, with values set to the corresponding inputs.
- PortfolioID on page 6-0 — String containing the `PortfolioID`
- VaRID on page 6-0 — `1`-by-`NumVaRs` string vector containing the `VaRID`s for the corresponding columns in `VaRData`
- VaRLevel on page 6-0 — `1`-by-`NumVaRs` numeric array containing the `VaRLevel`s for the corresponding columns in `VaRData`.

Note

- The required input arguments for `PortfolioData`, `VaRData`, and `ESData` must all be in the same units. These arguments can be expressed as returns or as profits and losses. There are no validations in the `esbacktestbysim` object regarding the units of these arguments.
- If there are missing values (`NaN`s) in `PortfolioData`, `VaRData`, `ESData`, or `Distribution` parameters data, the row of data is discarded before applying the tests. Therefore, a different number of observations are reported for models with a different number of missing values. The reported number of observations equals the original number of rows minus the number of missing values. To determine if there are discarded rows, use the `'Missing'` column of the `summary` report.

`ebts = esbacktestbysim( ___ ,Name,Value)` sets Properties on page 6-598 using name-value pairs and any of the arguments in the previous syntax. For example, `ebts = esbacktestbysim(PortfolioData,VaRData,ESData,DistributionName,'VaRID','TotalVaR','VaRLevel',.99)`. You can specify multiple name-value pairs.

Input Arguments

`PortfolioData` — Portfolio outcomes data

`NumRows`-by-`1` numeric array | `NumRows`-by-`1` numeric columns table | `NumRows`-by-`1` numeric columns timetable

Portfolio outcomes data, specified as a `NumRows`-by-`1` numeric array, `NumRows`-by-`1` table, or a `NumRows`-by-`1` timetable with a numeric column containing portfolio outcomes data. The `PortfolioData` input argument sets the PortfolioData on page 6-0 property.

Note `PortfolioData` data must be in the same units as `VaRData` and `ESData`. There are no validations in the `esbacktestbysim` object regarding the units of portfolio, VaR, and ES data. `PortfolioData`, `VaRData`, and `ESData` can be expressed as returns or as profits and losses.

Data Types: `double` | `table` | `timetable`

VaRData — Value-at-risk (VaR) data
NumRows-by-NumVaRs numeric array | NumRows-by-NumVaRs table with numeric columns | NumRows-by-NumVaRs timetable with numeric columns

Value-at-risk (VaR) data, specified as a NumRows-by-NumVaRs numeric array, NumRows-by-NumVaRs table, or a NumRows-by-NumVaRs timetable with numeric columns. The VaRData input argument sets the VaRData on page 6-0 property.

Negative VaRData values are allowed. However negative VaR values indicate a highly profitable portfolio that cannot lose money at the given VaR confidence level. The worst-case scenario at the given confidence level is still a profit.

Note VaRData must be in the same units as PortfolioData and ESData. There are no validations in the esbacktestbysim object regarding the units of portfolio, VaR, and ES data. VaRData, PortfolioData, and ESData can be expressed as returns or as profits and losses.

Data Types: double | table | timetable

ESData — Expected shortfall data
NumRows-by-NumVaRs positive numeric array | NumRows-by-NumVaRs table with positive numeric columns | NumRows-by-NumVaRs timetable with positive numeric columns

Expected shortfall data, specified as a NumRows-by-NumVaRs positive numeric array, NumRows-by-NumVaRs table, or NumRows-by-NumVaRs timetable with positive numeric columns containing ES data. The ESData input argument sets the ESData on page 6-0 property.

Note ESData data must be in the same units as PortfolioData and VaRData. There are no validations in the esbacktestbysim object regarding the units of portfolio, VaR, and ES data. ESData, PortfolioData, and VaRData can be expressed as returns or as profits and losses.

Data Types: double | table | timetable

DistributionName — Distribution name
string with values normal and t

Distribution name, specified as a string with a value of normal or t. The DistributionName input argument sets the 'Name' field of the Distribution on page 6-0 property.

Data Types: string

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: ebts = esbacktestbysim(PortfolioData,VaRData,ESData,DistributionName,'VaRID','TotalVaR','VaRLevel',.99)

PortfolioID — User-defined ID
character vector | string

User-defined ID for PortfolioData input, specified as the comma-separated pair consisting of 'PortfolioID' and a character vector or string. The PortfolioID name-value pair argument sets the PortfolioID on page 6-0 property.

If PortfolioData is a numeric array, the default value for PortfolioID is 'Portfolio'. If PortfolioData is a table, PortfolioID is set to the corresponding variable name in the table, by default.

Data Types: char | string

VaRID — VaR identifier
character vector | cell array of character vectors | string | string array

VaR identifier for VaRData columns, specified as the comma-separated pair consisting of 'VaRID' and a character vector, cell array of character vectors, string, or string array. Multiple VaRIDs are specified using a 1-by-NumVaRs (or NumVaRs-by-1) cell array of character vectors, or a string array with user-defined IDs for the VaRData columns. A single VaRID identifies a VaRData column and the corresponding ESData column. The VaRID name-value pair argument sets the VaRID on page 6-0 property.

If NumVaRs = 1, the default value for VaRID is 'VaR'. If NumVaRs > 1, the default value is 'VaR1', 'VaR2', and so on. If VaRData is a table, 'VaRID' is set by default to the corresponding variable names in the table.

Data Types: char | cell | string

VaRLevel — VaR confidence level
0.95 (default) | numeric or numeric array with values between 0 and 1

VaR confidence level, specified as a scalar with the comma-separated pair consisting of 'VaRLevel' and a numeric value between 0 and 1 or a 1-by-NumVaRs (or NumVaRs-by-1) numeric array with a numeric value between 0 and 1. The VaRLevel name-value pair argument sets the VaRLevel on page 6-0 property.

Data Types: double

Mean — Means for normal distribution
0 (default) | numeric | numeric array

Means for the normal distribution, specified as a comma-separated pair consisting of 'Mean' and a numeric value or a NumRows-by-1 numeric array. The Mean name-value pair argument sets the 'Mean' field of the Distribution on page 6-0 property.

Note You set the Mean name-value pair argument only when the DistributionName input argument is specified as normal.

Data Types: double

StandardDeviation — Standard deviation for normal distribution
1 (default) | positive numeric | positive numeric array

Standard deviation for the normal distribution, specified as a comma-separated pair consisting of 'StandardDeviation' and a positive numeric value or a NumRows-by-1 array. The StandardDeviation name-value pair argument sets the 'StandardDeviation' field of the Distribution on page 6-0 property.

Note You set the StandardDeviation name-value pair argument only when the DistributionName input argument is specified as normal.

Data Types: double

DegreesOfFreedom — Degrees of freedom for t distribution
integer ≥ 3

Degrees of freedom for the t distribution, specified as a comma-separated pair consisting of 'DegreesOfFreedom' and an integer value ≥ 3. The DegreesOfFreedom name-value pair argument sets the 'DegreesOfFreedom' field of the Distribution on page 6-0 property.

Note The DegreesOfFreedom name-value pair argument is only set when the DistributionName input argument is specified as t. A value for DegreesOfFreedom is *required* when the value of DistributionName is t.

Data Types: double

Location — Location parameters for t distribution
0 (default) | numeric | numeric array

Location parameters for the t distribution, specified as a comma-separated pair consisting of 'Location' and a numeric value or a NumRows-by-1 array. The Location name-value pair argument sets the'Location' field of the Distribution on page 6-0 property.

Note The Location name-value pair argument is only set when the DistributionName input argument is specified as t.

Data Types: double

Scale — Scale parameters for t distribution
1 (default) | positive numeric

Scale parameters for the t distribution, specified as a comma-separated pair consisting of 'Scale' and a positive numeric value or a NumRows-by-1 array. The Scale name-value pair argument sets the 'Scale' field of the Distribution on page 6-0 property.

Note The Scale name-value pair argument is only set when the DistributionName input argument is specified as t.

Data Types: double

`Simulate` — Indicates if simulation for statistical significance is run
`true` (default) | values are `true` or `false`

Indicates if a simulation for statistical significance is run when you create an `esbacktestbysim` object, specified as a logical scalar with the comma-separated pair consisting of `'Simulate'` and a value of `true` or `false`.

Data Types: `logical`

Properties

`PortfolioData` — Portfolio data for ES backtesting analysis
numeric array

Portfolio data for the ES backtesting analysis, specified as a `NumRows`-by-`1` numeric array containing a copy of the portfolio data.

Data Types: `double`

`VaRData` — VaR data for ES backtesting analysis
numeric array

VaR data for the ES backtesting analysis, specified as a `NumRows`-by-`NumVaRs` numeric array containing a copy of the VaR data.

Data Types: `double`

`ESData` — Expected shortfall data
numeric array

Expected shortfall data for ES backtesting analysis, specified as a `NumRows`-by-`NumVaRs` numeric array containing a copy of the `ESData`.

Data Types: `double`

`Distribution` — Distribution information
structure

Distribution information, including distribution name and the associated distribution parameters, specified as a structure.

For a `normal` distribution, the `Distribution` structure has fields `'Name'` (set to `normal`), `'Mean'`, and `'StandardDeviation'`, with values set to the corresponding inputs.

For a `t` distribution, the `Distribution` structure has fields `'Name'` (set to `t`), `'DegreesOfFreedom'`, `'Location'`, and `'Scale'`, with values set to the corresponding inputs.

Data Types: `struct`

`PortfolioID` — Portfolio identifier
string

Portfolio identifier, specified as a string.

Data Types: `string`

VaRID — VaR identifier
string | string array

VaR identifier, specified as a 1-by-NumVaRs string array containing the VaR IDs for the corresponding columns in VaRData.

Data Types: string

VaRLevel — VaR level
numeric array with values between 0 and 1

VaR level, specified as a 1-by-NumVaRs numeric array with values between 0 and 1 containing the VaR levels for the corresponding columns in VaRData.

Data Types: double

esbacktestbysim Property	Set or Modify Property from Command Line Using esbacktestbysim	Modify Property Using Dot Notation
PortfolioData	Yes	No
VaRData	Yes	No
ESData	Yes	No
Distribution	Yes	No
PortfolioID	Yes	Yes
VaRID	Yes	Yes
VaRLevel	Yes	Yes

Object Functions

summary	Basic expected shortfall (ES) report on failures and severity
runtests	Run all expected shortfall backtests (ES) for esbacktestbysim object
conditional	Conditional expected shortfall (ES) backtest by Acerbi and Szekely
unconditional	Unconditional expected shortfall backtest by Acerbi and Szekely
quantile	Quantile expected shortfall (ES) backtest by Acerbi and Szekely
minBiasRelative	Minimally biased relative test for Expected Shortfall (ES) backtest by Acerbi-Szekely
minBiasAbsolute	Minimally biased absolute test for Expected Shortfall (ES) backtest by Acerbi-Szekely
simulate	Simulate expected shortfall (ES) test statistics

Examples

Create esbacktestbysim Object and Run ES Backtests

esbacktestbysim takes in portfolio outcomes data, the corresponding value-at-risk (VaR) data, the expected shortfall (ES) data, and the Distribution information and returns an esbacktestbysim object.

Create an esbacktestbysim object and display the Distribution property.

```
load ESBacktestBySimData
rng('default'); % for reproducibility
```

```
ebts = esbacktestbysim(Returns,VaR,ES,"t",...
       'DegreesOfFreedom',10,...
       'Location',Mu,...
       'Scale',Sigma,...
       'PortfolioID',"S&P",...
       'VaRID',["t(10) 95%","t(10) 97.5%","t(10) 99%"],...
       'VaRLevel',VaRLevel)
```

```
ebts = 
  esbacktestbysim with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x3 double]
           ESData: [1966x3 double]
     Distribution: [1x1 struct]
      PortfolioID: "S&P"
            VaRID: ["t(10) 95%"    "t(10) 97.5%"    "t(10) 99%"]
         VaRLevel: [0.9500 0.9750 0.9900]
```

```
ebts.Distribution
```

```
ans = struct with fields:
                Name: "t"
    DegreesOfFreedom: 10
            Location: 0
               Scale: [1966x1 double]
```

ebts, the esbacktestbysim object, contains a copy of the given portfolio data (PortfolioData property), the given VaR data (VaRData property), the given ES data (ESData) property, and the given Distribution information. The object also contains all combinations of portfolio ID, VaR ID, and VaR level to be tested (PortfolioID, VaRID, and VaRLevel properties).

Run the tests using the ebts object.

```
TestResults = runtests(ebts)
```

TestResults=3×8 *table*

PortfolioID	VaRID	VaRLevel	Conditional	Unconditional	Quantile	MinB
"S&P"	"t(10) 95%"	0.95	reject	accept	reject	a
"S&P"	"t(10) 97.5%"	0.975	reject	reject	reject	r
"S&P"	"t(10) 99%"	0.99	reject	reject	reject	r

Change the PortfolioID property using dot notation. For more information on creating an esbacktestbysim object, see esbacktestbysim.

```
ebts.PortfolioID = 'S&P, 1996-2003'
```

```
ebts = 
  esbacktestbysim with properties:

    PortfolioData: [1966x1 double]
          VaRData: [1966x3 double]
           ESData: [1966x3 double]
     Distribution: [1x1 struct]
```

```
      PortfolioID: "S&P, 1996-2003"
            VaRID: ["t(10) 95%"    "t(10) 97.5%"    "t(10) 99%"]
         VaRLevel: [0.9500 0.9750 0.9900]
```

Run all tests using the updated `esbacktestbysim` object.

```
runtests(ebts)
```

```
ans=3×8 table
      PortfolioID           VaRID        VaRLevel    Conditional    Unconditional    Quantile
    ________________    _____________    ________    ___________    _____________    ________

    "S&P, 1996-2003"    "t(10) 95%"        0.95        reject          accept         reject
    "S&P, 1996-2003"    "t(10) 97.5%"     0.975        reject          reject         reject
    "S&P, 1996-2003"    "t(10) 99%"        0.99        reject          reject         reject
```

Version History

Introduced in R2017b

References

[1] Acerbi, C., and B. Szekely. *Backtesting Expected Shortfall*. MSCI Inc. December, 2014.

[2] Basel Committee on Banking Supervision. *Minimum Capital Requirements for Market Risk*. January, 2016 (https://www.bis.org/bcbs/publ/d352.pdf).

See Also

`summary` | `runtests` | `conditional` | `unconditional` | `quantile` | `simulate` | `minBiasRelative` | `minBiasAbsolute` | `esbacktest` | `table` | `timetable` | `varbacktest` | `esbacktestbyde`

Topics

"Expected Shortfall (ES) Backtesting Workflow Using Simulation" on page 2-34
"Expected Shortfall Estimation and Backtesting" on page 2-44
"Overview of Expected Shortfall Backtesting" on page 2-20
"Comparison of ES Backtesting Methods" on page 2-26

expectedClaims

Create `expectedClaims` object

Description

Use this workflow to generate unpaid claims for an `expectedClaims`:

1. Load or generate the data for the development triangle.
2. Create a `developmentTriangle` object.
3. Create an `expectedClaims` object.
4. Use the `ultimateClaims` function to calculate the projected ultimate claims.
5. Use the `ibnr` function to calculate the incurred-but-not-reported (IBNR) claims.
6. Use the `unpaidClaims` function to calculate the unpaid claims.
7. Use the `summary` function to generate a summary report for the expected claims analysis.

Creation

Syntax

```
ec = expectedClaims(dT_reported,dT_paid,earnedPremium)
ec = expectedClaims( ___ ,Name,Value)
```

Description

`ec = expectedClaims(dT_reported,dT_paid,earnedPremium)` creates an `expectedClaims` object using the `developmentTriangle` objects for reported claims (`dT_reported`) and paid claims (`dT_paid`), as well as the `earnedPremium`.

`ec = expectedClaims( ___ ,Name,Value)` sets properties on page 6-603 using name-value pairs and any of the arguments in the previous syntax. For example, `ec = expectedClaims(dT_reported,dT_paid,earnedPremium,'InitialClaims',initialSelectedUltimateClaims)`. You can specify multiple name-value arguments.

Input Arguments

`dT_reported` — Development triangle for reported claims
`developmentTriangle` object

Development triangle for reported claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

`dT_paid` — Development triangle for paid claims
`developmentTriangle` object

Development triangle for paid claims, specified as a previously created `developmentTriangle` object.

Data Types: `object`

`earnedPremium` — Earned premium for each `Origin` period
array

Earned premium for each `Origin` period, specified as an array.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `ec = expectedClaims(dT_reported,dT_paid, earnedPremium,'InitialClaims',initialSelectedUltimateClaims)`

`InitialClaims` — Initial selected ultimate claims
average of the projected reported ultimate claims and the projected paid Ultimate Claims (default) | array

Initial selected ultimate claims, specified as the comma-separated pair consisting of `'InitialClaims'` and an array.

Data Types: `double`

Properties

`ReportedTriangle` — Development triangle for reported claims
`developmentTriangle` object

Development triangle for reported claims, returned as a `developmentTriangle` object containing the origin years, development years, and claims.

Data Types: `object`

`PaidTriangle` — Development triangle for paid claims
`developmentTriangle` object

Development triangle for paid claims, returned as a `developmentTriangle` object containing the origin years, development years, and claims.

Data Types: `object`

`earnedPremium` — Earned premium for each `Origin` period
array

Earned premium for each `Origin` period, returned as an array.

Data Types: `double`

InitialClaims — Initial selected ultimate claims
average of the projected reported ultimate claims and the projected paid Ultimate Claims (default) | array

Initial selected ultimate claims, returned as an array.

Data Types: double

Object Functions

ultimateClaims	Compute projected ultimate claims for expectedClaims object
ibnr	Compute IBNR claims for expectedClaims object
unpaidClaims	Compute unpaid claims estimates for expectedClaims object
summary	Display summary report for different claims estimates

Examples

Create expectedClaims Object

Create an expectedClaims object for simulated insurance claims data.

```
load InsuranceClaimsData.mat;
head(data)
```

```
    OriginYear    DevelopmentYear    ReportedClaims    PaidClaims
    __________    _______________    ______________    __________

       2010             12               3995.7          1893.9  
       2010             24                 4635          3371.2  
       2010             36               4866.8          4079.1  
       2010             48               4964.1            4487  
       2010             60               5013.7          4711.4  
       2010             72               5038.8          4805.6  
       2010             84                 5059          4853.7  
       2010             96               5074.1          4877.9  
```

Use developmentTriangle to convert the data to a development triangle, which is the standard form for representing claims data. Create two developmentTriangle objects, one for reported claims and one for paid claims.

```
dT_reported = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Cla
```

```
dT_reported = 
  developmentTriangle with properties:

                          Origin: {10x1 cell}
                     Development: {10x1 cell}
                          Claims: [10x10 double]
                  LatestDiagonal: [10x1 double]
                     Description: ""
                      TailFactor: 1
    CumulativeDevelopmentFactors: [1.3069 1.1107 1.0516 1.0261 1.0152 1.0098 1.0060 1.0030 1.001
              SelectedLinkRatio: [1.1767 1.0563 1.0249 1.0107 1.0054 1.0038 1.0030 1.0020 1.001
```

```
dT_paid = developmentTriangle(data,'Origin','OriginYear','Development','DevelopmentYear','Claims
```

```
dT_paid = 
  developmentTriangle with properties:

                         Origin: {10x1 cell}
                    Development: {10x1 cell}
                         Claims: [10x10 double]
                 LatestDiagonal: [10x1 double]
                    Description: ""
                     TailFactor: 1
    CumulativeDevelopmentFactors: [2.4388 1.4070 1.1799 1.0810 1.0378 1.0178 1.0080 1.0030 1.0010
               SelectedLinkRatio: [1.7333 1.1925 1.0914 1.0417 1.0196 1.0097 1.0050 1.0020 1.0010
```

Create an `expectedClaims` object where the first input argument is the reported development triangle and the second input argument is the paid development triangle.

```
earnedPremium = [17000; 18000; 10000; 19000; 16000; 10000; 11000; 10000; 14000; 10000];
ec = expectedClaims(dT_reported, dT_paid,earnedPremium)
```

```
ec = 
  expectedClaims with properties:

          ReportedTriangle: [1x1 developmentTriangle]
              PaidTriangle: [1x1 developmentTriangle]
             EarnedPremium: [10x1 double]
             InitialClaims: [10x1 double]
           CaseOutstanding: [10x1 double]
    EstimatedClaimsRatios: [10x1 double]
     SelectedClaimsRatios: [10x1 double]
```

Version History

Introduced in R2020b

See Also

`developmentTriangle` | `chainLadder` | `bornhuetterFerguson`

Topics

“Overview of Claims Estimation Methods for Non-Life Insurance” on page 1-16

customLifetimePDModel

Create `customLifetimePDModel` object for lifetime probability of default

Description

Create and analyze a `customLifetimePDModel` object to calculate the lifetime probability of default (PD) using this workflow:

1. Fit a PD model that can predict PD for a loan or a portfolio of loans.
2. Define a function handle for a function that predicts the PD in your designated PD model.
3. Use `customLifetimePDModel` and pass the specified function handle to create a `customLifetimePDModel` object. The designated model is now wrapped as a lifetime PD model.
4. Use `predict` to predict the conditional PD and `predictLifetime` to predict the lifetime PD.
5. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
6. Use `modelCalibration` to return the RMSE of the observed and predicted PD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
CustomLifetimePDModel = customLifetimePDModel(pdFcnHandle,IDVar=idvar_value,
ResponseVar=responsevar_value)
CustomLifetimePDModel = customLifetimePDModel( ___ ,Name=Value)
```

Description

`CustomLifetimePDModel = customLifetimePDModel(pdFcnHandle,IDVar=idvar_value, ResponseVar=responsevar_value)` creates a `customLifetimePDModel` object for a PD model using required name-value arguments and sets model object properties on page 6-609.

`CustomLifetimePDModel = customLifetimePDModel( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax. The optional name-value arguments set model object properties on page 6-609. For example, `CustomLifetimePDModel = customLifetimePDModel(pdFcnHandle,IDVar='ID',AgeVar='YOB',Description='Scorecard as lifetime PD model',LoanVars='ScoreGroup',MacroVars={'GDP''Market'},ModelID='ScorecardLifetime',ResponseVar='Default')` creates a `CustomLifetimePDModel` object.

Input Arguments

`pdFcnHandle` — Function handle for custom model probability of default prediction function
function_handle

Function handle for custom model probability of default prediction function, specified as a function handle.

The function takes in a data table which includes variables that you specify in `AgeVar`, `LoanVars`, and `MacroVars`, and returns a predicted conditional PD value for each row of the table.

Note Because the `pdFcnHandle` function passes the data input in its entirety to the prediction and validation methods, it allows extra columns in the data table for other variables, such as `IDVar`, `ResponseVar`, and grouping variables.

Data Types: `function_handle`

`customLifetimePDModel` Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `CustompdModel = customLifetimePDModel(pdFcnHandle,IDVar='ID',AgeVar='YOB',Description='Scorecard as lifetime PD model',LoanVars='ScoreGroup',MacroVars={'GDP''Market'},ModelID='ScorecardLifetime',ResponseVar='Default')`

Required customLifetimePDModel Name-Value Arguments

`IDVar` — ID variable indicating which column in data input contains loan or borrower ID

string | character vector

ID variable indicating which column in the data accepted as input by `pdFcnHandle` contains the loan or borrower ID, specified as `IDVar` and a string or character vector.

Note `IDVar` is required for lifetime PD prediction using `predictLifetime`.

Data Types: `string` | `char`

`ResponseVar` — Variable indicating which column in the data input contains response variable

string | character vector

Variable indicating which column in the data accepted as input by `pdFcnHandle` contains the response variable, specified as `ResponseVar` and a string or character vector.

Note `ResponseVar` is required for model validation when you use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Data Types: `string` | `char`

Optional customLifetimePDModel Name-Value Arguments

`ModelID` — User-defined model ID

`customLifetimePDModel` (default) | string | character vector

User-defined model ID, specified as `ModelID` and a string or character vector. The software uses the `ModelID` to format outputs and is expected to be short.

Data Types: `string` | `char`

`Description` — User-defined description for model

`""` (default) | string | character vector

User-defined description for the model, specified as `Description` and a string or character vector.

Data Types: `string` | `char`

`AgeVar` — Age variable indicating which column in data input contains loan age information

`""` (default) | string | character vector

Age variable indicating which column in the data accepted as input by `pdFcnHandle` contains the loan age information, specified as `AgeVar` and a string or character vector.

Note `AgeVar`, `LoanVars`, and `MacroVars` work together as data input for the predictor variables of the model. You must specify at least one of these inputs. The `predict` function validates that the data input contains all the predictor variables.

If the distinction between `AgeVar`, `LoanVars`, and `MacroVars` is not important for the custom model's PD prediction, use `LoanVars` to store all the predictor variables in the model.

An age variable is common for lifetime PD modeling. When you specify `AgeVar` the `predictLifetime` function uses it to validate the periodicity of the rows in the data.

Data Types: `string` | `char`

`LoanVars` — Loan variables indicating which column in data input contains loan-specific information

`""` (default) | string array | cell array of character vectors

Loan variables indicating which column in the data accepted as input by `pdFcnHandle` contains the loan-specific information, such as origination score or loan-to-value ratio, specified as `LoanVars` and a string array or cell array of character vectors.

Note `AgeVar`, `LoanVars`, and `MacroVars` work together as data input for the predictor variables of the model. You must specify at least one of these inputs. The `predict` function validates that the data input contains all the predictor variables.

If the distinction between `AgeVar`, `LoanVars`, and `MacroVars` is not important for the custom model's PD prediction, use `LoanVars` to store all the predictor variables in the model.

Data Types: `string` | `cell`

`MacroVars` — Macro variables indicating which column in data input contains macroeconomic information
`""` (default) | string array | cell array of character vectors

Macro variables indicating which column in the data accepted as input by `pdFcnHandle` contains the macroeconomic information, such as gross domestic product (GDP) growth or unemployment rate, specified as `MacroVars` and a string array or cell array of character vectors.

Note `AgeVar`, `LoanVars`, and `MacroVars` work together as data input for the predictor variables of the model. You must specify at least one of these inputs. The `predict` function validates that the data input contains all the predictor variables.

If the distinction between `AgeVar`, `LoanVars`, and `MacroVars` is not important for the custom model's PD prediction, use `LoanVars` to store all the predictor variables in the model.

Data Types: `string` | `cell`

Properties

`ModelID` — User-defined model ID
`customLifetimePDModel` (default) | string

User-defined model ID, returned as a string.

Data Types: `string`

`Description` — User-defined description
`""` (default) | string

User-defined description, returned as a string.

Data Types: `string`

`Model` — Custom model defined using function handle
custom model

Custom model defined using the function handle (`pdFcnHandle`), returned as the PD prediction function handle (`pdFcnHandle`).

Data Types: `function_handle`

`IDVar` — ID variable indicating which column in data input contains loan or borrower ID
string

ID variable indicating which column in the data input defined by `pdFcnHandle` contains loan or borrower ID, returned as a string.

Data Types: `string`

`AgeVar` — Age variable indicating which column in data input contains loan age information
`""` (default) | string

Age variable indicating which column in the data input defined by `pdFcnHandle`contains loan age information, returned as a string.

Data Types: `string`

`LoanVars` — Loan variables indicating which column in data input contains loan-specific information
`""` (default) | string array

Loan variables indicating which column in the data input defined by `pdFcnHandle` contains loan-specific information, returned as a string array.

Data Types: `string`

`MacroVars` — Macro variables indicating which column in data input contains macroeconomic information
`""` (default) | string array

Macro variables indicating which column in the data input defined by `pdFcnHandle` contains macroeconomic information, returned as a string array.

Data Types: `string`

`ResponseVar` — Variable indicating which column in data input contains response variable
string

Variable indicating which column in the data input defined by `pdFcnHandle` contains the response variable, returned as a string.

Data Types: `string`

Object Functions

predict	Compute conditional PD
predictLifetime	Compute cumulative lifetime PD, marginal PD, and survival probability
modelDiscrimination	Compute AUROC and ROC data
modelCalibration	Compute RMSE of predicted and observed PDs on grouped data
modelDiscriminationPlot	Plot ROC curve
modelCalibrationPlot	Plot observed default rates compared to predicted PDs on grouped data

Examples

Create Custom Lifetime PD Model

This example shows how to use the `customLifetimePDModel` object with a function handle to wrap a credit scorecard model as a `customLifetimePDModel` model.

Load Data

Load the credit portfolio data. The data set is in panel data format, with multiple rows per loan.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
```

```
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP      Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup     YOB    Default    Year     GDP     Market
    __    __________     ___    _______    ____    _____    ______

    1      Low Risk       1        0       1997     2.72      7.61
    1      Low Risk       2        0       1998     3.57     26.24
    1      Low Risk       3        0       1999     2.86      18.1
    1      Low Risk       4        0       2000     2.43      3.19
    1      Low Risk       5        0       2001     1.26    -10.51
    1      Low Risk       6        0       2002    -0.59    -22.95
    1      Low Risk       7        0       2003     0.63      2.78
    1      Low Risk       8        0       2004     1.85      9.48
```

Fit Credit Scorecard Model

Use `creditscorecard` to create a `creditscorecard` object, use `autobinning` to perform automatic binning of specified predictors, and then use `fitmodel` to fit a logistic regression model to weight of evidence (WOE) data. In this example, the entire data set is used to train the model.

```
sc = creditscorecard(data,'IDVar','ID','PredictorVars',{'ScoreGroup' 'YOB' 'GDP' 'Market'},'Respo
sc = autobinning(sc);
sc = autobinning(sc,'YOB','Algorithm','Split');
sc = fitmodel(sc,'Display','off');
displaypoints(sc)
```

```
ans=16×3 table
      Predictors            Bin           Points
    ______________    ________________    _______

    {'ScoreGroup'}    {'High Risk'  }     0.61102
    {'ScoreGroup'}    {'Medium Risk'}      1.3043
    {'ScoreGroup'}    {'Low Risk'   }      1.9113
    {'ScoreGroup'}    {'<missing>'  }         NaN
```

```
    {'YOB'       }    {'[-Inf,2)'   }    0.56226
    {'YOB'       }    {'[2,5)'      }     1.0024
    {'YOB'       }    {'[5,7)'      }     1.4549
    {'YOB'       }    {'[7,Inf]'    }      2.509
    {'YOB'       }    {'<missing>'  }        NaN
    {'GDP'       }    {'[-Inf,0.63)'}      1.042
    {'GDP'       }    {'[0.63,Inf]' }     1.1657
    {'GDP'       }    {'<missing>'  }        NaN
    {'Market'    }    {'[-Inf,2.78)'}     1.0731
    {'Market'    }    {'[2.78,9.48)'}     1.1219
    {'Market'    }    {'[9.48,Inf]' }     1.2294
    {'Market'    }    {'<missing>'  }        NaN
```

Create `customLifetimePDModel` Object Using Function Handle

Use `customLifetimePDModel` with a function handle for the `probdefault` function.

```
pdFcnHandle = @(data) probdefault(sc,data);
pdModel = customLifetimePDModel(pdFcnHandle,IDVar='ID',AgeVar='YOB', ...
          Description='Scorecard as lifetime PD model',LoanVars='ScoreGroup', ...
          MacroVars={'GDP' 'Market'},ModelID='ScorecardLifetime',ResponseVar='Default');
disp(pdModel)
```

```
  CustomLifetimePD with properties:

            ModelID: "ScorecardLifetime"
        Description: "Scorecard as lifetime PD model"
    UnderlyingModel: @(data)probdefault(sc,data)
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

```
pdModel.UnderlyingModel
```

```
ans = function_handle with value:
    @(data)probdefault(sc,data)
```

Predict Lifetime PD

Use the `predictLifetime` function to predict lifetime cumulative PD values for the first ID associated with the first eight rows of the `data`. The data input to `predictLifetime` must be in panel data form, with multiple rows per loan, and the function computes the cumulative probability of default for each period. For more information, see “Time Interval and Data Input for Lifetime Prediction” on page 6-348.

```
predictLifetime(pdModel,data(1:8,:))
```

```
ans = 8×1

    0.0085
    0.0134
    0.0182
    0.0236
    0.0272
    0.0312
```

```
    0.0324
    0.0335
```

Validate Model

By wrapping the scorecard model as a lifetime PD model, all the validation functionality of the lifetime PD models is available. For example, use `modelCalibrationPlot` to visualize the observed default rates compared to the predicted probabilities of default.

```
modelCalibrationPlot(pdModel,data,'YOB')
```

Version History

Introduced in R2022b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

Functions
`fitLifetimePDModel` | `Probit` | `Cox` | `Logistic`

Topics
“Create Custom Lifetime PD Model for Credit Scorecard Model with Function Handle” on page 3-131
“Create Custom Lifetime PD Model for Decision Tree Model with Function Handle” on page 4-224
“Expected Credit Loss Computation” on page 4-124
“Overview of Lifetime Probability of Default Models” on page 1-25

Logistic

Create `Logistic` model object for lifetime probability of default

Description

Create and analyze a `Logistic` model object to calculate the lifetime probability (PD) of default using this workflow:

1. Use `fitLifetimePDModel` to create a `Logistic` model object.
2. Use `predict` to predict the conditional PD and `predictLifetime` to predict the lifetime PD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the RMSE of the observed and predicted PD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
LogisticPDModel = fitLifetimePDModel(data,ModelType)
LogisticPDModel = fitLifetimePDModel( ___ ,Name,Value)
```

Description

`LogisticPDModel = fitLifetimePDModel(data,ModelType)` creates a `Logistic` PD model object.

If you do not specify variable information for `IDVar`, `AgeVar`, `LoanVars`, `MacroVars`, and `ResponseVar`, then:

- `IDVar` is set to the first column in the `data` input.
- `LoanVars` is set to include all columns from the second to the second-to-last columns of the `data` input.
- `ResponseVar` is set to the last column in the `data` input.

`LogisticPDModel = fitLifetimePDModel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The optional name-value pair arguments set model object properties on page 6-617. For example, `LogisticPDModel = fitLifetimePDModel(data(TrainDataInd,:),"Logistic",'ModelID',"Logistic_A",'Description',"Logisitic_model",'AgeVar',"YOB",'IDVar',"ID",'LoanVars',"ScoreGroup",'MacroVars',{'GDP','Market'}'ResponseVar',"Default")` creates a `LogisticPDModel` object using a `Logistic` model type.

Input Arguments

`data` — Data
table

Data, specified as a table, in panel data form. The `data` must contain an ID column. The response variable must be a binary variable with the value `0` or `1`, with `1` indicating default.

Data Types: `table`

`ModelType` — Model type
string with value `"Logistic"` | character vector with value `'Logistic'`

Model type, specified as a string with the value of `"Logistic"` or a character vector with the value of `'Logistic'`.

Data Types: `char` | `string`

`Logistic` Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `LogisticPDModel = fitLifetimePDModel(data(TrainDataInd,:),"Logistic",'ModelID',"Logistic_A",'Description',"Logisitic_model",'AgeVar',"YOB",'IDVar',"ID",'LoanVars',"ScoreGroup",'MacroVars',{'GDP','Market'}'ResponseVar',"Default")`

`ModelID` — User-defined model ID
`Logistic` (default) | string | character vector

User-defined model ID, specified as the comma-separated pair consisting of `'ModelID'` and a string or character vector. The software uses the `ModelID` to format outputs and is expected to be short.

Data Types: `string` | `char`

`Description` — User-defined description for model
`""` (default) | string | character vector

User-defined description for model, specified as the comma-separated pair consisting of `'Description'` and a string or character vector.

Data Types: `string` | `char`

`IDVar` — ID variable indicating which column in `data` contains loan or borrower ID
1st column of `data` (default) | string | character vector

ID variable indicating which column in `data` contains the loan or borrower ID, specified as the comma-separated pair consisting of `'IDVar'` and a string or character vector.

Data Types: `string` | `char`

`AgeVar` — Age variable indicating which column in `data` contains loan age information
if not specified, then `LoanVars` (default) | string | character vector

Age variable indicating which column in `data` contains the loan age information, specified as the comma-separated pair consisting of `'AgeVar'` and a string or character vector.

Data Types: `string` | `char`

`LoanVars` — Loan variables indicating which column in `data` contains loan-specific information
all columns of `data` that are not the first or last column (default) | string array | cell array of character vectors

Loan variables indicating which column in `data` contains the loan-specific information, such as origination score or loan-to-value ratio, specified as the comma-separated pair consisting of `'LoanVars'` and a string array or cell array of character vectors.

Data Types: `string` | `cell`

`MacroVars` — Macro variables indicating which column in `data` contains macroeconomic information
if not specified, then `LoanVars` (default) | string array | cell array of character vectors

Macro variables indicating which column in `data` contains the macroeconomic information, such as gross domestic product (GDP) growth or unemployment rate, specified as the comma-separated pair consisting of `'MacroVars'` and a string array or cell array of character vectors.

Data Types: `string` | `cell`

`ResponseVar` — Variable indicating which column in `data` contains response variable
string | character vector

Variable indicating which column in `data` contains the response variable, specified as the comma-separated pair consisting of `'ResponseVar'` and a string or character vector.

Note The response variable *values* in the `data` must be a binary variable with `0` or `1` values, with `1` indicating default.

Data Types: `string` | `char`

Properties

`ModelID` — User-defined model ID
`Logistic` (default) | string

User-defined model ID, returned as a string.

Data Types: `string`

`Description` — User-defined description
`""` (default) | string

User-defined description, returned as a string.

Data Types: `string`

`UnderlyingModel` — Underlying statistical model
compact linear model

Underlying statistical model, returned as a compact generalized linear model object. For more information, see `fitglm` and `CompactGeneralizedLinearModel`.

Data Types: `CompactGneralizedLinearModel`

`IDVar` — ID variable indicating which column in `data` contains loan or borrower ID
1st column of `data` (default) | string

ID variable indicating which column in `data` contains loan or borrower ID, returned as a string.

Data Types: `string`

`AgeVar` — Age variable indicating which column in `data` contains loan age information
if not specified, then `LoanVars` (default) | string

Age variable indicating which column in `data` contains loan age information, returned as a string.

Data Types: `string`

`LoanVars` — Loan variables indicating which column in `data` contains loan-specific information
all columns of `data` that are not the first or last column (default) | string array

Loan variables indicating which column in `data` contains loan-specific information, returned as a string array.

Data Types: `string`

`MacroVars` — Macro variables indicating which column in `data` contains macroeconomic information
if not specified, then `LoanVars` (default) | string array

Macro variables indicating which column in `data` contains macroeconomic information, returned as a string array.

Data Types: `string`

`ResponseVar` — Variable indicating which column in `data` contains response variable
string

Variable indicating which column in `data` contains the response variable, returned as a string.

Data Types: `string`

Object Functions

predict	Compute conditional PD
predictLifetime	Compute cumulative lifetime PD, marginal PD, and survival probability
modelDiscrimination	Compute AUROC and ROC data
modelCalibration	Compute RMSE of predicted and observed PDs on grouped data
modelDiscriminationPlot	Plot ROC curve
modelCalibrationPlot	Plot observed default rates compared to predicted PDs on grouped data

Examples

Create Logistic Lifetime PD Model

This example shows how to use `fitLifetimePDModel` to create a `Logistic` model using credit and macroeconomic data.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);

TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create Logistic Lifetime PD Model

Use `fitLifetimePDModel` to create a `Logistic` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Logistic",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
disp(pdModel)
```

```
  Logistic with properties:

            ModelID: "Logistic"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Display the underlying model.

```
pdModel.UnderlyingModel
```

```
ans = 
Compact generalized linear regression model:
    logit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue   
                              __________    _________    _______    ___________

    (Intercept)                  -2.7422      0.10136    -27.054      3.408e-161
    ScoreGroup_Medium Risk      -0.68968     0.037286    -18.497      2.1894e-76
    ScoreGroup_Low Risk          -1.2587     0.045451    -27.693     8.4736e-169
    YOB                         -0.30894     0.013587    -22.738     1.8738e-114
    GDP                         -0.11111     0.039673    -2.8006       0.0051008
    Market                    -0.0083659    0.0028358    -2.9502       0.0031761


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Predict Conditional and Lifetime PD

Use the `predict` function to predict conditional PD values. The prediction is a row-by-row prediction.

```
dataCustomer1 = data(1:8,:);
CondPD = predict(pdModel,dataCustomer1)
```

```
CondPD = 8×1

    0.0092
    0.0053
    0.0045
    0.0039
    0.0037
    0.0037
    0.0019
    0.0012
```

Use `predictLifetime` to predict the lifetime cumulative PD values (computing marginal and survival PD values is also supported). The `predictLifetime` function uses the ID variable (see the `'IDVar'` property for the `Logistic` object) to transform conditional PDs to cumulative PDs for each ID.

```
LifetimePD = predictLifetime(pdModel,dataCustomer1)
```

```
LifetimePD = 8×1

    0.0092
    0.0145
    0.0189
    0.0228
    0.0264
    0.0300
    0.0319
    0.0330
```

Validate Model

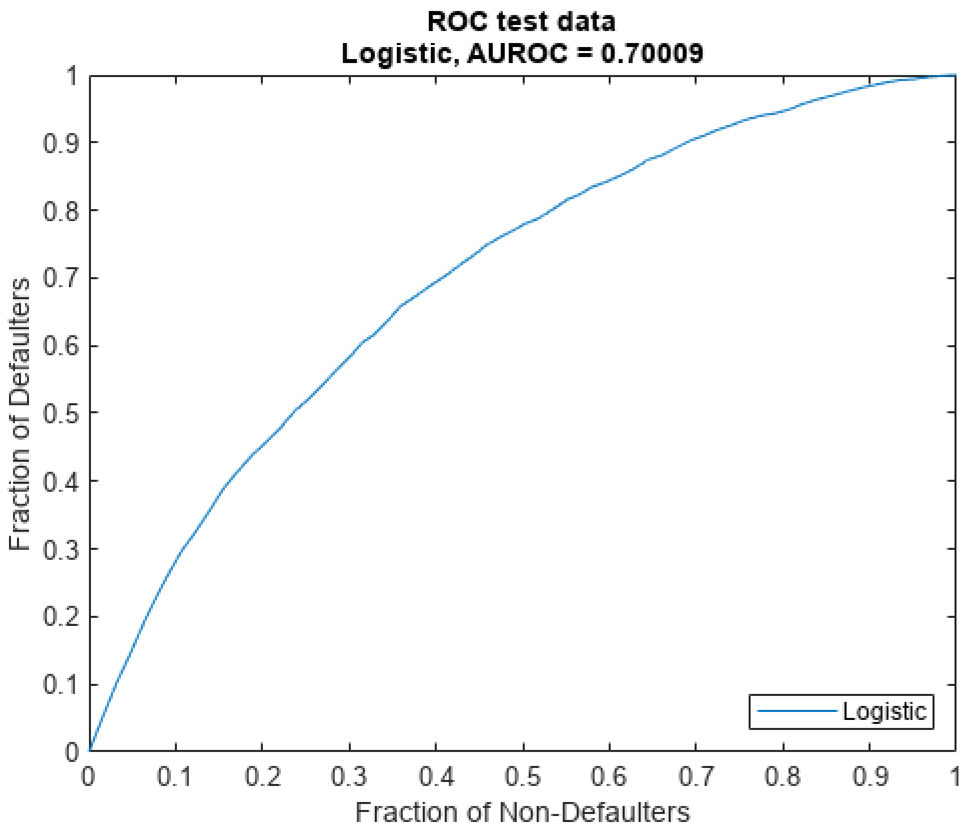
Use `modelDiscrimination` to measure the ranking of customers by PD.

```
DiscMeasure = modelDiscrimination(pdModel,data(TestDataInd,:),DataID='test data');
disp(DiscMeasure)
```

```
                             AUROC 
                            _______

    Logistic, test data    0.70009
```

Use `modelDiscriminationPlot` to visualize the ROC curve.

```
modelDiscriminationPlot(pdModel,data(TestDataInd,:),DataID='test data');
```



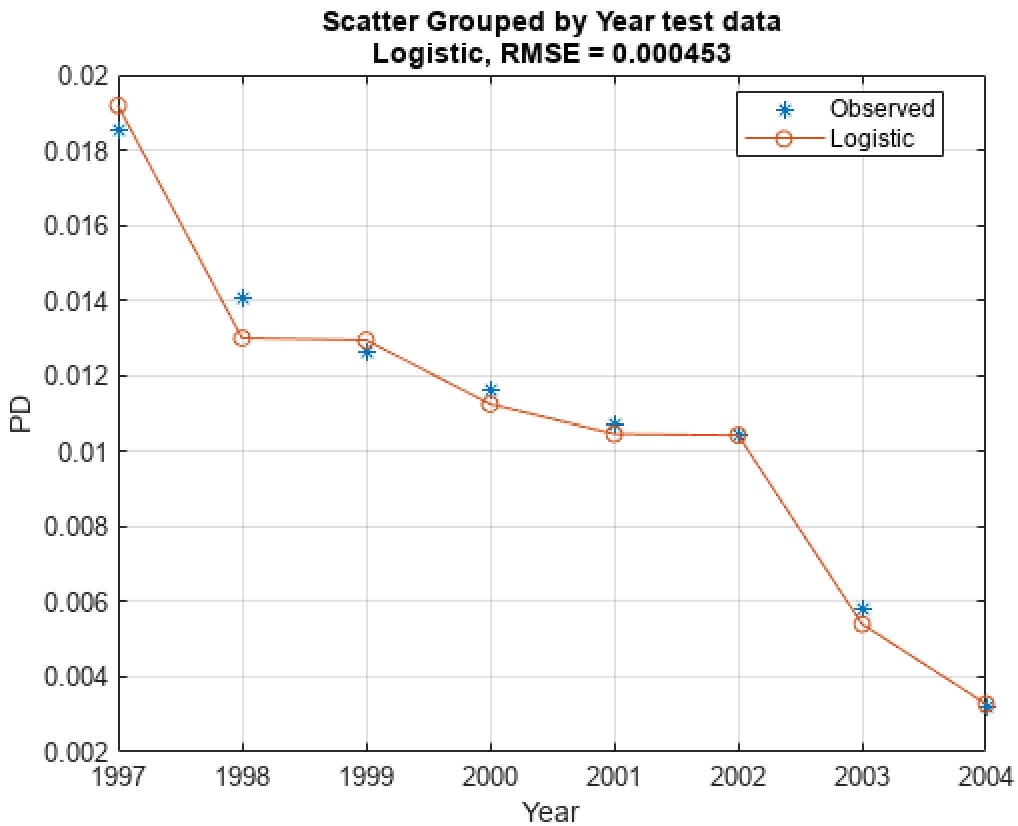
Use `modelCalibration` to measure the calibration of the predicted PD values. The `modelCalibration` function requires a grouping variable and compares the accuracy of the observed default rate in the group with the average predicted PD for the group. For example, you can group by calendar year using the `'Year'` variable.

```
CalMeasure = modelCalibration(pdModel,data(TestDataInd,:),'Year',DataID='test data');
disp(CalMeasure)
```

	RMSE
Logistic, grouped by Year, test data	0.000453

Use `modelCalibrationPlot` to visualize the observed default rates compared to the predicted probabilities of default (PD).

```
modelCalibrationPlot(pdModel,data(TestDataInd,:),'Year',DataID='test data');
```



More About

Time Interval for `Logistic` Models

For `Logistic` and `Probit` models, there is a time interval implicit in the data, specifically, in the definition of the default variable.

For example, if the default indicator is defined so that it takes the value `1` if there is a default over a 3-month period, the time interval is 3-months. In this case, the predicted PD values are 3-month PD predictions. Then the PD for month 18 would be the conditional probability that there is a default between months 15 and 18, given that there has been no default in the first 15 months.

Because the `data` input for `fitLifetimePDModel` is in panel data form, there is an implicit or explicit time stamp for each row, and the time interval used in the default definition should be the same as the time increments between consecutive rows. If there is an optional age variable (`AgeVar`) in the training `data`, the time interval should be the same as the age increments (for the same ID) from one row to the next.

`Logistic` and `Probit` models do not explicitly infer or store the time interval information. However, the predicted PD values returned by `predict` are consistent with the time interval implicit in the panel training `data`, which in turn should be the same as the time interval used to define the default variable.

Unlike `Logistic` and `Probit` models, `Cox` models require an `AgeVar` variable, and the time interval is inferred from the increments in the age values in the training `data`. `Cox` models store the time interval value as the `TimeInterval` property. For more information, see “Lifetime Prediction and Time Interval” on page 6-342.

Version History

Introduced in R2020b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

R2023a: `Model` property renamed to `UnderlyingModel`

Behavior changed in R2023a

The `Model` property is renamed to `UnderlyingModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

Functions

`fitLifetimePDModel` | `Probit` | `Cox` | `customLifetimePDModel`

Topics

“Basic Lifetime PD Model Validation” on page 4-129
“Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
“Compare Lifetime PD Models Using Cross-Validation” on page 4-121
“Expected Credit Loss Computation” on page 4-124
“Compare Model Discrimination and Model Calibration to Validate of Probability of Default” on page 4-144
“Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models” on page 4-75

"Overview of Lifetime Probability of Default Models" on page 1-25

Probit

Create `Probit` model object for lifetime probability of default

Description

Create and analyze a `Probit` model object to calculate lifetime probability of default (PD) using this workflow:

1. Use `fitLifetimePDModel` to create a `Probit` model object.
2. Use `predict` to predict the conditional PD and `predictLifetime` to predict the lifetime PD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the RMSE of observed and predicted PD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
ProbitPDModel = fitLifetimePDModel(data,ModelType)
ProbitPDModel = fitLifetimePDModel( ___ ,Name,Value)
```

Description

`ProbitPDModel = fitLifetimePDModel(data,ModelType)` creates a `Probit` PD model object.

If you do not specify variable information for `IDVar`, `AgeVar`, `LoanVars`, `MacroVars`, and `ResponseVar`, then:

- `IDVar` is set to the first column in the `data` input.
- `LoanVars` is set to include all columns from the second to the second-to-last columns of the `data` input.
- `ResponseVar` is set to the last column in the `data` input.

`ProbitPDModel = fitLifetimePDModel( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The optional name-value pair arguments set the model object properties on page 6-628. For example, `ProbitPDModel = fitLifetimePDModel(data(TrainDataInd,:),"Probit",'ModelID',"Probit_A",'Description',"Probit_model",'AgeVar',"YOB",'IDVar',"ID",'LoanVars',"ScoreGroup",'MacroVars',{'GDP','Market'},'ResponseVar',"Default")` creates a `ProbitPDModel` object using a `Probit` model type.

Input Arguments

`data` — Data
table

Data, specified as a table, in panel data form. The data must contain an ID column. The response variable must be a binary variable with the value 0 or 1, with 1 indicating default.

Data, specified as a table where the first column is IDVar, the last column is the ResponseVar, and all other columns are LoanVars.

Data Types: table

ModelType — Model type
string with value "Probit" | character vector with value 'Probit'

Model type, specified as a string with the value "Probit" or a character vector with the value 'Probit'.

Data Types: char | string

Probit Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: ProbitPDModel =
fitLifetimePDModel(data(TrainDataInd,:),"Probit",'ModelID',"Probit_A",'Descri
pion',"Probit_model",'AgeVar',"YOB",'IDVar',"ID",'LoanVars',"ScoreGroup",'Mac
roVars',{'GDP','Market'},'ResponseVar',"Default")

ModelID — User-defined model ID
Probit (default) | string | character vector

User-defined model ID, specified as the comma-separated pair consisting of 'ModelID' and a string or character vector. The software uses the ModelID to format outputs and is expected to be short.

Data Types: string | char

Description — User-defined description for model
"" (default) | string | character vector

User-defined description for model, specified as the comma-separated pair consisting of 'Description' and a string or character vector.

Data Types: string | char

IDVar — ID variable indicating which column in data contains loan or borrower ID
1st column of data (default) | string | character vector

ID variable indicating which column in data contains the loan or borrower ID, specified as the comma-separated pair consisting of 'IDVar' and a string or character vector.

Data Types: string | char

AgeVar — Age variable indicating which column in data contains loan age information
if not specified, then LoanVars (default) | string | character vector

Age variable indicating which column in data contains the loan age information, specified as the comma-separated pair consisting of 'AgeVar' and a string or character vector.

Data Types: `string` | `char`

`LoanVars` — Loan variables indicating which column in `data` contains loan-specific information
all columns of `data` that are not the first or last column (default) | string array | cell array of character vectors

Loan variables indicating which column in `data` contains the loan-specific information, such as origination score or loan-to-value ratio, specified as the comma-separated pair consisting of `'LoanVars'` and a string array or cell array of character vectors.

Data Types: `string` | `cell`

`MacroVars` — Macro variables indicating which column in `data` contains macroeconomic information
if not specified, then `LoanVars` (default) | string array | cell array of character vectors

Macro variables indicating which column in `data` contains the macroeconomic information, such as gross domestic product (GDP) growth or unemployment rate, specified as the comma-separated pair consisting of `'MacroVars'` and a string array or cell array of character vectors.

Data Types: `string` | `cell`

`ResponseVar` — Variable indicating which column in `data` contains response variable
string | character vector

Variable indicating which column in `data` contains the response variable, specified as the comma-separated pair consisting of `'ResponseVar'` and a string or character vector.

Note The response variable *values* in the `data` must be a binary variable with `0` or `1` values, with `1` indicating default.

Data Types: `string` | `char`

Properties

`ModelID` — User-defined Model ID
`Probit` (default) | string

User-defined model ID, returned as a string.

Data Types: `string`

`Description` — User-defined description
`""` (default) | string

User-defined description, returned as a string.

Data Types: `string`

`UnderlyingModel` — Underlying statistical model
compact linear model

Underlying statistical model, returned as a compact generalized linear model object. For more information, see `fitglm` and `CompactGeneralizedLinearModel`.

Data Types: `CompactGneralizedLinearModel`

`IDVar` — ID variable indicating which column in `data` contains loan or borrower ID
1st column of `data` (default) | string

ID variable indicating which column in `data` contains the loan or borrower ID, returned as a string.

Data Types: `string`

`AgeVar` — Age variable indicating which column in `data` contains loan age information
if not specified, then `LoanVars` (default) | string

Age variable indicating which column in `data` contains the loan age information, returned as a string.

Data Types: `string`

`LoanVars` — Loan variables indicating which column in `data` contains loan-specific information
all columns of `data` that are not the first or last column (default) | string array

Loan variables indicating which column in `data` contains the loan-specific information, returned as a string array.

Data Types: `string`

`MacroVars` — Macro variables indicating which column in `data` contains macroeconomic information
if not specified, then `LoanVars` (default) | string array

Macro variables indicating which column in `data` contains the macroeconomic information, returned as a string array.

Data Types: `string`

`ResponseVar` — Variable indicating which column in `data` contains response variable
string

Variable indicating which column in `data` contains the response variable, returned as a string.

Data Types: `logical`

Object Functions

predict	Compute conditional PD
predictLifetime	Compute cumulative lifetime PD, marginal PD, and survival probability
modelDiscrimination	Compute AUROC and ROC data
modelCalibration	Compute RMSE of predicted and observed PDs on grouped data
modelDiscriminationPlot	Plot ROC curve
modelCalibrationPlot	Plot observed default rates compared to predicted PDs on grouped data

Examples

Create Probit Lifetime PD Model

This example shows how to use `fitLifetimePDModel` to create a `Probit` model using credit and macroeconomic data.

Load Data

Load the credit portfolio data.

```
load RetailCreditPanelData.mat
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year
    __    __________    ___    _______    ____

    1      Low Risk      1        0       1997
    1      Low Risk      2        0       1998
    1      Low Risk      3        0       1999
    1      Low Risk      4        0       2000
    1      Low Risk      5        0       2001
    1      Low Risk      6        0       2002
    1      Low Risk      7        0       2003
    1      Low Risk      8        0       2004
```

```
disp(head(dataMacro))
```

```
    Year     GDP     Market
    ____    _____    ______

    1997     2.72      7.61
    1998     3.57     26.24
    1999     2.86      18.1
    2000     2.43      3.19
    2001     1.26    -10.51
    2002    -0.59    -22.95
    2003     0.63      2.78
    2004     1.85      9.48
```

Join the two data components into a single data set.

```
data = join(data,dataMacro);
disp(head(data))
```

```
    ID    ScoreGroup    YOB    Default    Year     GDP     Market
    __    __________    ___    _______    ____    _____    ______

    1      Low Risk      1        0       1997     2.72      7.61
    1      Low Risk      2        0       1998     3.57     26.24
    1      Low Risk      3        0       1999     2.86      18.1
    1      Low Risk      4        0       2000     2.43      3.19
    1      Low Risk      5        0       2001     1.26    -10.51
    1      Low Risk      6        0       2002    -0.59    -22.95
    1      Low Risk      7        0       2003     0.63      2.78
    1      Low Risk      8        0       2004     1.85      9.48
```

Partition Data

Separate the data into training and test partitions.

```
nIDs = max(data.ID);
uniqueIDs = unique(data.ID);

rng('default'); % for reproducibility
c = cvpartition(nIDs,'HoldOut',0.4);
```

```
TrainIDInd = training(c);
TestIDInd = test(c);

TrainDataInd = ismember(data.ID,uniqueIDs(TrainIDInd));
TestDataInd = ismember(data.ID,uniqueIDs(TestIDInd));
```

Create a Probit Lifetime PD Model

Use `fitLifetimePDModel` to create a `Probit` model using the training data.

```
pdModel = fitLifetimePDModel(data(TrainDataInd,:),"Probit",...
    'AgeVar','YOB',...
    'IDVar','ID',...
    'LoanVars','ScoreGroup',...
    'MacroVars',{'GDP','Market'},...
    'ResponseVar','Default');
disp(pdModel)
```

```
  Probit with properties:

            ModelID: "Probit"
        Description: ""
    UnderlyingModel: [1x1 classreg.regr.CompactGeneralizedLinearModel]
              IDVar: "ID"
             AgeVar: "YOB"
           LoanVars: "ScoreGroup"
          MacroVars: ["GDP"    "Market"]
        ResponseVar: "Default"
```

Display the underlying model.

```
disp(pdModel.UnderlyingModel)
```

```
Compact generalized linear regression model:
    probit(Default) ~ 1 + ScoreGroup + YOB + GDP + Market
    Distribution = Binomial

Estimated Coefficients:
                               Estimate        SE         tStat       pValue   
                              __________    _________    _______    ___________

    (Intercept)                  -1.6267      0.03811    -42.685              0
    ScoreGroup_Medium Risk      -0.26542      0.01419    -18.704     4.5503e-78
    ScoreGroup_Low Risk         -0.46794     0.016364    -28.595     7.775e-180
    YOB                         -0.11421    0.0049724    -22.969    9.6208e-117
    GDP                        -0.041537     0.014807    -2.8052      0.0050291
    Market                    -0.0029609    0.0010618    -2.7885      0.0052954


388097 observations, 388091 error degrees of freedom
Dispersion: 1
Chi^2-statistic vs. constant model: 1.85e+03, p-value = 0
```

Predict Conditional and Lifetime PD

Use the `predict` function to predict conditional PD values. The prediction is a row-by-row prediction.

```
dataCustomer1 = data(1:8,:);
CondPD = predict(pdModel,dataCustomer1)
```

```
CondPD = 8×1

    0.0095
    0.0054
    0.0045
    0.0039
    0.0036
    0.0036
    0.0017
    0.0009
```

Use `predictLifetime` to predict the lifetime cumulative PD values (computing marginal and survival PD values is also supported). The `predictLifetime` function uses the ID variable (see the `'IDVar'` property for the `Logistic` object) to transform conditional PDs to cumulative PDs for each ID.

```
LifetimePD = predictLifetime(pdModel,dataCustomer1)
```

```
LifetimePD = 8×1

    0.0095
    0.0149
    0.0193
    0.0232
    0.0267
    0.0302
    0.0318
    0.0327
```

Validate Model

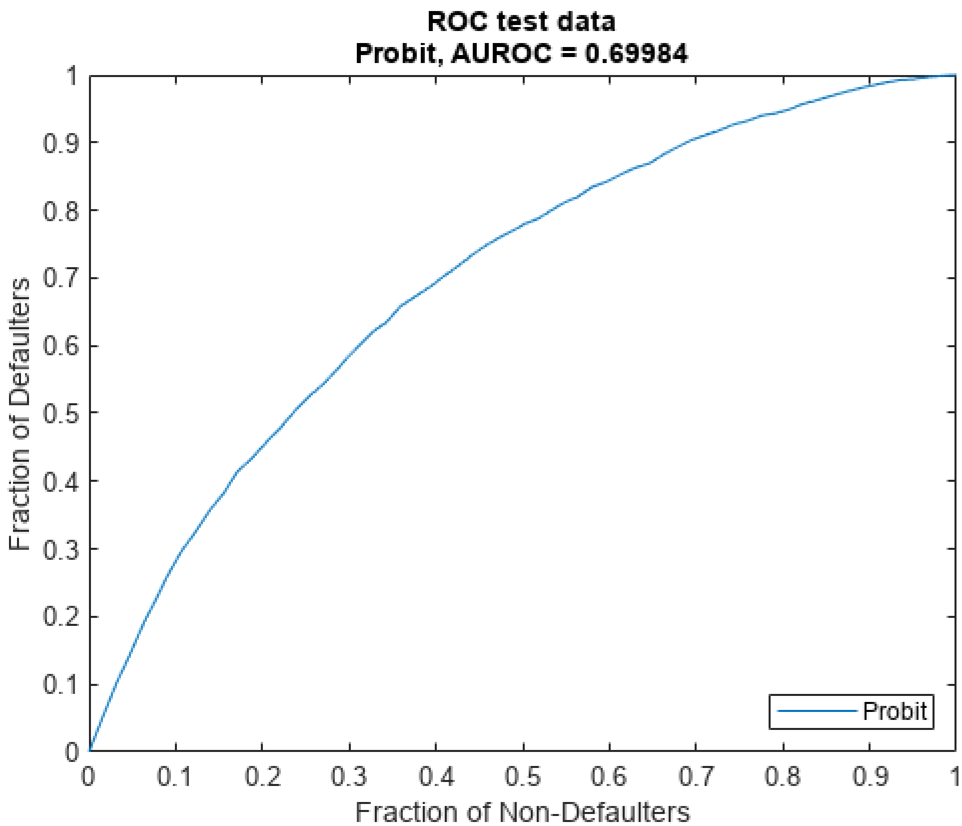
Use `modelDiscrimination` to measure the ranking of customers by PD.

```
DiscMeasure = modelDiscrimination(pdModel,data(TestDataInd,:),DataID='test data');
disp(DiscMeasure)
```

```
                          AUROC 
                         _______

    Probit, test data    0.69984
```

Use `modelDiscriminationPlot` to visualize the ROC curve.

```
modelDiscriminationPlot(pdModel,data(TestDataInd,:),DataID='test data');
```



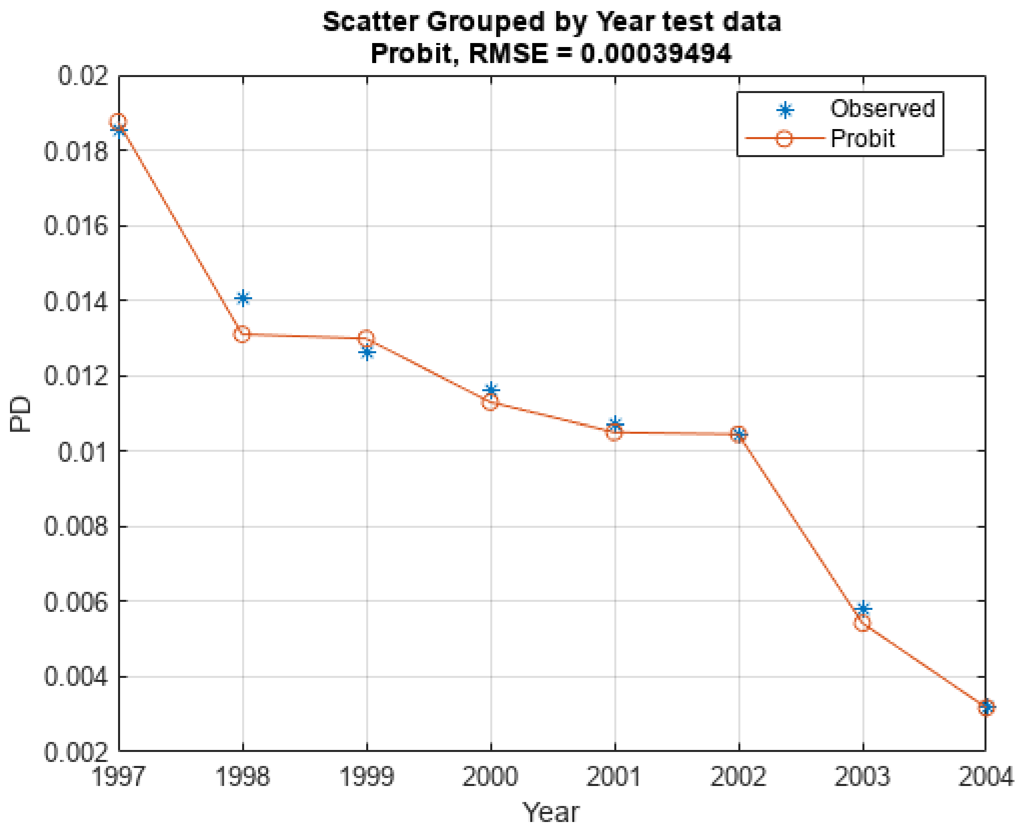
Use `modelCalibration` to measure the calibration of the predicted PD values. The `modelCalibration` function requires a grouping variable and compares the accuracy of the observed default rate in the group with the average predicted PD for the group. For example, you can group by calendar year using the `'Year'` variable.

```
CalMeasure = modelCalibration(pdModel,data(TestDataInd,:),'Year',DataID='test data');
disp(CalMeasure)
```

```
                                        RMSE
                                     __________

    Probit, grouped by Year, test data    0.00039494
```

Use `modelCalibrationPlot` to visualize the observed default rates compared to the predicted probabilities of default (PD).

```
modelCalibrationPlot(pdModel,data(TestDataInd,:),'Year',DataID='test data');
```



More About

Time Interval for `Probit` Models

For `Logistic` and `Probit` models, there is a time interval implicit in the data, specifically, in the definition of the default variable.

For example, if the default indicator is defined so that it takes the value `1` if there is a default over a 3-month period, the time interval is 3-months. In this case, the predicted PD values are 3-month PD predictions. Then the PD for month 18 would be the conditional probability that there is a default between months 15 and 18, given that there has been no default in the first 15 months.

Because the `data` input for `fitLifetimePDModel` is in panel data form, there is an implicit or explicit time stamp for each row, and the time interval for the default definition should be the same as the time increments between consecutive rows. If there is an optional age variable (`AgeVar`) in the training `data`, the time interval is the same as the age increments (for the same ID) from one row to the next.

`Logistic` and `Probit` models do not explicitly infer or store the time interval information. However, the predicted PD values returned by `predict` are consistent with the time interval implicit in the panel training `data`, which in turn should be the same as the time interval used to define the default variable.

Unlike `Logistic` and `Probit` models, `Cox` models require an `AgeVar` variable, and the time interval is inferred from the increments in the age values in the training `data`. `Cox` models store the time interval value as the `TimeInterval` property. For more information, see “Lifetime Prediction and Time Interval” on page 6-342.

Version History

Introduced in R2020b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function
Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function
Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

R2023a: `Model` property renamed to `UnderlyingModel`
Behavior changed in R2023a

The `Model` property is renamed to `UnderlyingModel`.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Breeden, Joseph. *Living with CECL: The Modeling Dictionary.* Santa Fe, NM: Prescient Models LLC, 2018.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk: Machine Learning with Python.* Independently published, 2020.

See Also

Functions
`fitLifetimePDModel` | `Logistic` | `Cox` | `customLifetimePDModel`

Topics
“Basic Lifetime PD Model Validation” on page 4-129
“Compare Logistic Model for Lifetime PD to Champion Model” on page 4-113
“Compare Lifetime PD Models Using Cross-Validation” on page 4-121
“Expected Credit Loss Computation” on page 4-124
“Compare Model Discrimination and Model Calibration to Validate of Probability of Default” on page 4-144
“Compare Probability of Default Using Through-the-Cycle and Point-in-Time Models” on page 4-75

“Overview of Lifetime Probability of Default Models” on page 1-25

Regression

Create `Regression` model object for exposure at default

Description

Create and analyze a `Regression` model object to calculate the exposure at default (EAD) using this workflow:

1. Use `fitEADModel` to create a `Regression` model object.
2. Use `predict` to predict the EAD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the R-square, RMSE, correlation, and sample mean error of the predicted and observed EAD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
RegressionEADModel = fitEADModel(data,ModelType)
RegressionEADModel = fitEADModel(___,Name=Value)
```

Description

`RegressionEADModel = fitEADModel(data,ModelType)` creates a `Regression` EAD model object.

`RegressionEADModel = fitEADModel(___,Name=Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The optional name-value pair arguments set model object properties on page 6-639. For example, `eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'},ConversionMeasure="ccf",DrawnVar='Drawn',LimitVar='Limit',ResponseVar='EAD')` creates an `eadModel` object using a `Regression` model type.

Input Arguments

`data` — Data for loss given default
table

Data for loss given default, specified as a table.

Data Types: `table`

`ModelType` — Model type
string with value `"Regression"` | character vector with value `'Regression'`

Model type, specified as a string with the value of `"Regression"` or a character vector with the value of `'Regression'`.

Data Types: char | string

Regression Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'},ConversionMeasure="ccf",DrawnVar='Drawn',LimitVar='Limit',ResponseVar='EAD')

ModelID — User-defined model ID
"Regression" (default) | string | character vector

User-defined model ID, specified as ModelID and a string or character vector. The software uses the ModelID text to format outputs and is expected to be short.

Data Types: string | char

Description — User-defined description for model
"" (default) | string | character vector

User-defined description for model, specified as Description and a string or character vector.

Data Types: string | char

PredictorVars — Predictor variables
all columns of data except for the ResponseVar (default) | string array | cell array of character vectors

Predictor variables, specified as PredictorVars and a string array or cell array of character vectors. PredictorVars indicates which columns in the data input contain the predictor information. By default, PredictorVars is set to all the columns in the data input except for the ResponseVar.

Data Types: string | cell

ResponseVar — Response variable
last column of data (default) | string | character vector

Response variable, specified as ResponseVar and a string or character vector. The response variable contains the EAD data and must be a numeric variable. By default, the ResponseVar is set to the last column of data.

Data Types: string | char

BoundaryTolerance — Boundary tolerance
1e-7 (default) | positive numeric

Boundary tolerance, specified as BoundaryTolerance and a positive scalar numeric. The BoundaryTolerance value perturbs the EAD response values away from 0 and 1, before applying a response transformation.

Data Types: double

`LimitVar` — Limit variable
string | character vector

Limit variable, specified as `LimitVar` and a string or character vector. `LimitVar` indicates which column in `data` contains the limit amount. The limit amount value in the `data` must be a positive numeric value. The limit depends on the loan. If its a credit card, the limit is the credit limit, and if this is a mortgage limit it is the initial loan amount. In general, `LimitVar` is the maximum amount that can be borrowed.

Note `LimitVar` is required when `ConversionMeasure` is `'ccf'` or `'lcf'`. For more information on CCF and LCF, see "Exposure at Default Regression Models" on page 6-645.

Data Types: `string` | `char`

`DrawnVar` — Drawn variable
string | character vector

Drawn variable, specified as `DrawnVar` and a string or character vector. `DrawnVar` is the balance on the account at the time of observation, prior to default and EAD is the balance at the time of default. `DrawnVar` indicates which column in the `data` contains the drawn amount. The drawn variable value in the `data` can be a positive or negative numeric value.

Note `DrawnVar` is required when `ConversionMeasure` is `'ccf'`.

If the `ConversionMeasure` is `'lcf'`, `DrawnVar` is not required. In this case, `DrawnVar` is set to `""`.

For more information on CCF, see "Exposure at Default Regression Models" on page 6-645.

Data Types: `string` | `char`

`ConversionMeasure` — Conversion measure for EAD response values
`"ccf"` (default) | character vector with value of `'ccf'` or `'lcf'` | string with value of `"ccf"` or `"lcf"`

Response transform, specified as `ConversionMeasure` and a character vector or string.

- `"ccf"` — Credit conversion factor (CCF) is the portion of the undrawn amount that will be converted into credit. The undrawn amount is the limit minus the drawn amount. The EAD thus becomes the drawn amount plus the CCF times the limit minus the drawn amount (`EAD = Drawn + CCF*(Limit - Drawn)`).
- `"lcf"` — Limit conversion factor (LCF) is a fraction of the limit representing the total exposure. The EAD is then defined as the LCF times the limit (`EAD = LCF*Limit`).

For more information on CCF and LCF, see "Exposure at Default Regression Models" on page 6-645.

Data Types: `string` | `char`

Properties

`ModelID` — User-defined model ID
`"Regression"` (default) | string

User-defined model ID, returned as a string.

Data Types: `string`

`Description` — User-defined description
`""` (default) | string

User-defined description, returned as a string.

Data Types: `string`

`UnderlyingModel` — Underlying statistical model
compact linear model

Underlying statistical model, returned as a compact linear model object. The compact version of the underlying regression model is an instance of the `classreg.regr.CompactLinearModel` class. For more information, see `fitlm` and `CompactLinearModel`.

Data Types: `CompactLinearModel`

`PredictorVars` — Predictor variables
all columns of `data` except for `ResponseVar` (default) | string array

Predictor variables, returned as a string array.

Data Types: `string`

`ResponseVar` — Response variable
last column of `data` (default) | string

Response variable, returned as a scalar string.

Data Types: `string`

`LimitVar` — Limit variable
string

Limit variable, returned as a string.

Data Types: `string`

`DrawnVar` — Drawn variable
string

Drawn variable, returned as a string.

Data Types: `string`

`BoundaryTolerance` — Boundary tolerance
`1e-7` (default) | positive numeric

This property is read-only.

Boundary tolerance, returned as a scalar positive numeric.

Data Types: `double`

`ConversionMeasure` — Conversion measure for EAD response values
`"ccf"` (default) | string with value of `"ccf"` or `"lcf"`

Conversion measure, returned as a string.

Data Types: `string`

`ConversionTransform` — Conversion transform
`"complog"` (default) | string with value `"complog"` or `"logit"`

This property is read-only.

Conversion transform, returned as a string that is `"complog"` if `ConversionMeasure` is `"ccf"` and `"logit"` when `ConversionMeasure` is `"lcf"`.

Data Types: `string`

Object Functions

predict	Predict exposure at default
modelDiscrimination	Compute AUROC and ROC data
modelDiscriminationPlot	Plot ROC curve
modelCalibration	Compute R-square, RMSE, correlation, and sample mean error of predicted and observed EADs
modelCalibrationPlot	Scatter plot of predicted and observed EADs

Examples

Create Regression EAD Model

This example shows how to use `fitEADModel` to create a `Regression` model for exposure at default (EAD).

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Regression` or `Tobit`.

```
ModelType = Regression ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Regression EAD Model

Use `fitEADModel` to create a `Regression` model using `EADData`.

```
eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'}, ...
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Regression with properties:

    ConversionTransform: "logit"
      BoundaryTolerance: 1.0000e-07
                ModelID: "Regression"
            Description: ""
        UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
          PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
            ResponseVar: "EAD"
               LimitVar: "Limit"
               DrawnVar: "Drawn"
      ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'BoundaryTolerance'`, `'LimitVar'`, and `'DrawnVar'` name-value arguments to modify the transformation.

```
eadModel.UnderlyingModel
```

```
ans =
Compact linear regression model:
    EAD_lcf_logit ~ 1 + UtilizationRate + Age + Marriage

Estimated Coefficients:
                            Estimate       SE         tStat       pValue   
                           _________    _________    _______    __________

    (Intercept)              -2.4745      0.29892    -8.2781    1.6448e-16
    UtilizationRate           6.0045      0.19901     30.172    7.703e-182
    Age                    -0.020095    0.0073019     -2.752     0.0059471
    Marriage_not married    -0.03509      0.13935    -0.2518        0.8012


Number of observations: 4378, Error degrees of freedom: 4374
Root Mean Squared Error: 4.48
R-squared: 0.173,  Adjusted R-Squared: 0.173
F-statistic vs. constant model: 305, p-value = 5.7e-180
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-vale argument.

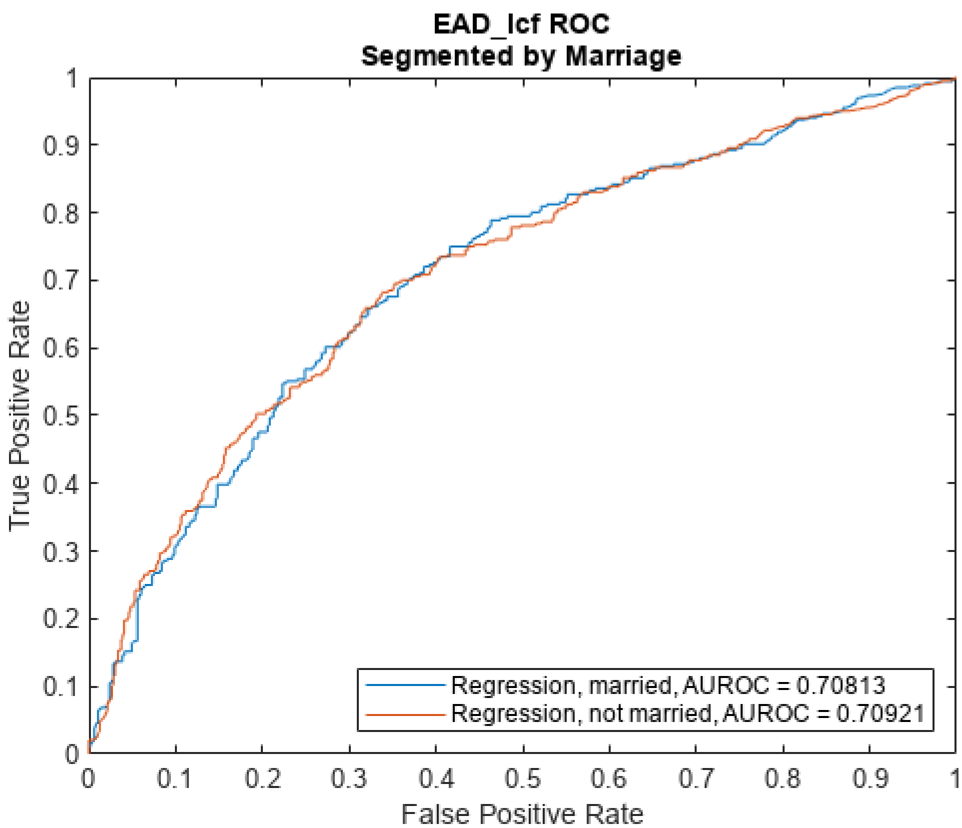
```
predictedEAD = predict(eadModel, EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel, EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

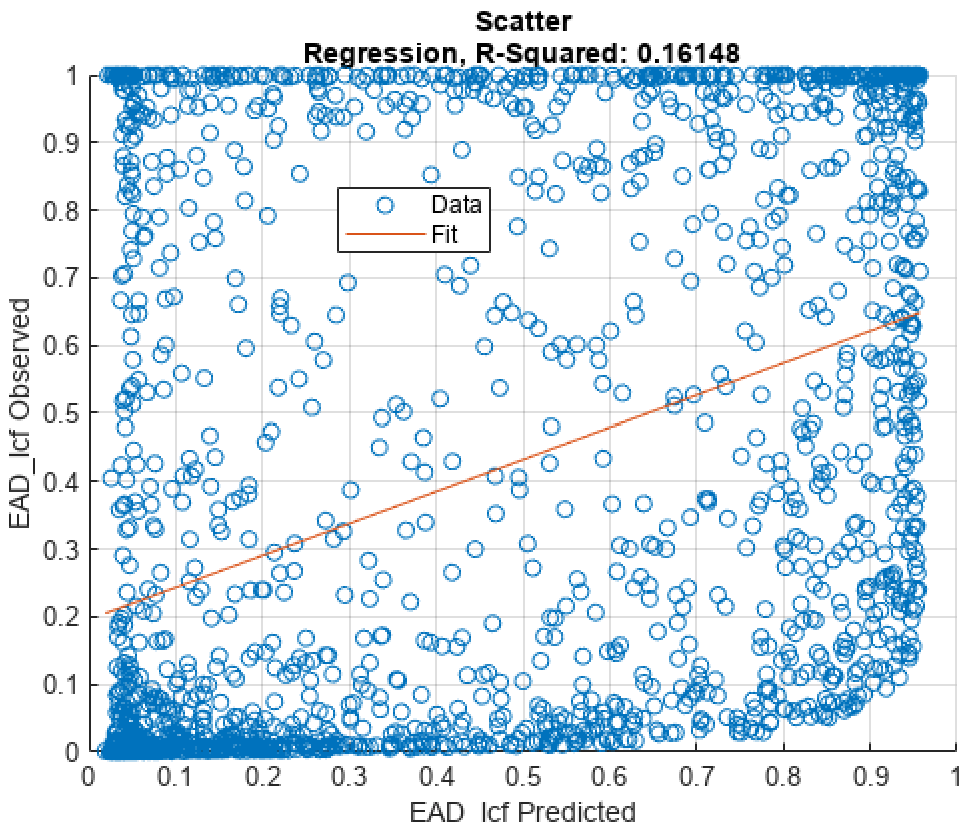
```
ModelLevel = ConversionMeasure ;
[DiscMeasure1, DiscData1] = modelDiscrimination(eadModel, EADData(TestInd,:),ModelLevel=ModelLeve
modelDiscriminationPlot(eadModel, EADData(TestInd, :), ModelLevel=ModelLevel,SegmentBy="Marriage'
```



Use `modelCalibration` and then `modelCalibrationPlot` to show a scatter plot of the predictions.

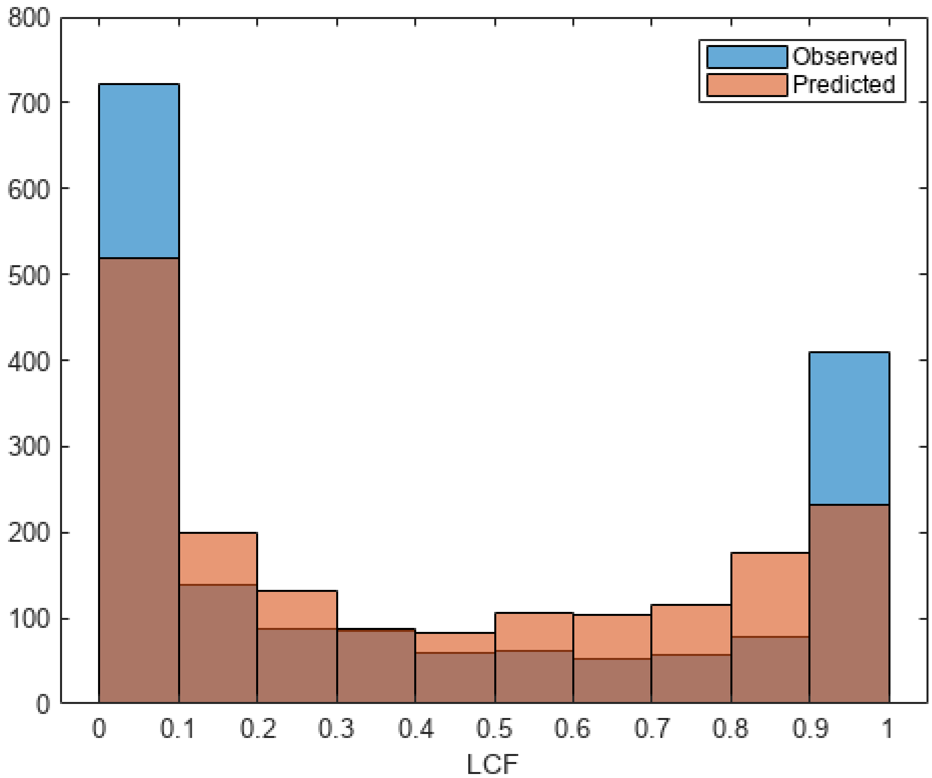
```
YData = Observed ;
```

```
[CalMeasure1, CalData1] = modelCalibration(eadModel, EADData(TestInd,:), ModelLevel=ModelLevel);
modelCalibrationPlot(eadModel, EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Plot a histogram of observed with respect to the predicted EAD.

```
figure;
histogram(CalData1.Observed);
hold on;
histogram(CalData1.(('Predicted_' + ModelType)));
xlabel(ConversionMeasure);
legend('Observed', 'Predicted');
```



More About

Exposure at Default Regression Models

You can transform EAD data using linear regression models.

You can relate the EAD to a scaling variable and derive conversion measures like credit conversion factor (CCF) and limit conversion factor (LCF) using the `'ccf'` or `'lcf'` options for the `ConversionMeasure` name-value argument. In general, `Regression` models that use a `ConversionMeasure` for conversion factors are more robust, as all observations scale to a common denomination.

The following table summarizes the supported transformations using the `'ccf'` or `'lcf'` options for the `ConversionMeasure` name-value argument:

Measure	EAD Formula	Lower Bound	Upper Bound	Inverse Transformation
CCF	`EAD = Drawn + CCF Ã— (Limit - Drawn)`	-Inf	1	`CCF = 1 - e`$^{(-CCF_t)}$
LCF	`EAD = LCF  Limit`	0	1	`LCF = e`^{LCF_t} `â^•` `(1 + e`^{LCF_t}`)`

Version History

Introduced in R2021b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

Functions

`fitEADModel` | `Tobit` | `Beta`

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Exposure at Default Models" on page 1-34

Tobit

Create `Tobit` model object for exposure at default

Description

Create and analyze a `Tobit` model object to calculate the exposure at default (EAD) using this workflow:

1. Use `fitEADModel` to create a `Tobit` model object.
2. Use `predict` to predict the EAD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the R-squared, RMSE, correlation, and sample mean error of predicted and observed EAD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
TobitEADModel = fitEADModel(data,ModelType)
TobitEADModel = fitEADModel(___,Name=Value)
```

Description

`TobitEADModel = fitEADModel(data,ModelType)` creates a `Tobit` EAD model object.

`TobitEADModel = fitEADModel(___,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax. The optional name-value arguments set the model object properties on page 6-650. For example, `eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'},ConversionMeasure="ccf",DrawnVar='Drawn',LimitVar='Limit',ResponseVar='EAD')` creates an `eadModel` object using a `Tobit` model type.

Input Arguments

`data` — Data for exposure at default
table

Data for exposure at default, specified as a table.

Data Types: `table`

`ModelType` — Model type
string with value `"Tobit"` | character vector with value `'Tobit'`

Model type, specified as a string with the value of `"Tobit"` or a character vector with the value of `'Tobit'`.

Data Types: `char` | `string`

Tobit Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'},ConversionMeasure="ccf",DrawnVar='Drawn',LimitVar='Limit',ResponseVar='EAD')`

`ModelID` — User-defined model ID
`"Tobit"` (default) | string | character vector

User-defined model ID, specified as `ModelID` and a string or character vector. The software uses the `ModelID` text to format outputs and is expected to be short.

Data Types: `string` | `char`

`Description` — User-defined description for model
`""` (default) | string | character vector

User-defined description for model, specified as `Description` and a string or character vector.

Data Types: `string` | `char`

`PredictorVars` — Predictor variables
all columns of `data` except for `ResponseVar` (default) | string array | cell array of character vectors

Predictor variables, specified as `PredictorVars` and a string array or cell array of character vectors. `PredictorVars` indicates which columns in the `data` input contain the predictor information. By default, `PredictorVars` is set to all the columns in the `data` input except for `ResponseVar`.

Data Types: `string` | `cell`

`ResponseVar` — Response variable
last column of `data` (default) | string | character vector

Response variable, specified as `ResponseVar` and a string or character vector. The response variable contains the EAD data and must be a numeric variable. By default, `ResponseVar` is set to the last column.

Data Types: `string` | `char`

`LimitVar` — Limit variable
string | character vector

Limit variable, specified as `LimitVar` and a string or character vector. `LimitVar` indicates which column in `data` contains the limit amount. The limit amount value in the `data` must be a positive numeric value. The limit depends on the loan. If its a credit card, the limit is the credit limit, and if this is a mortgage limit it is the initial loan amount. In general, `LimitVar` is the maximum amount that can be borrowed.

Note `LimitVar` is required when `ConversionMeasure` is `'ccf'` or `'lcf'`. For more information on CCF and LCF, see “Conversion Measure Options” on page 6-658.

Data Types: `string` | `char`

`DrawnVar` — Drawn variable
string | character vector

Drawn variable, specified as `DrawnVar` and a string or character vector. `DrawnVar` is the balance on the account at the time of observation, prior to default and EAD is the balance at the time of default. `DrawnVar` indicates which column in `data` contains the drawn amount. The drawn variable value in the `data` can be a positive or negative numeric value.

Note `DrawnVar` is required when `ConversionMeasure` is `'ccf'`.

If the `ConversionMeasure` is `'lcf'`, `DrawnVar` is not required. In this case, `DrawnVar` is set to `""`.

For more information on CCF, see “Conversion Measure Options” on page 6-658.

Data Types: `string` | `char`

`ConversionMeasure` — Conversion measure for EAD response values
`"ccf"` (default) | character vector with value of `'ccf'` or `'lcf'` | string with value of `"ccf"` or `"lcf"`

Response transform, specified as `ConversionMeasure` and a character vector or string.

- `"ccf"` — Credit conversion factor (CCF) is the portion of the undrawn amount that will be converted into credit. The undrawn amount is the limit minus the drawn amount. The EAD thus becomes the drawn amount plus the CCF times the limit minus the drawn amount (`EAD = Drawn + CCF*(Limit - Drawn)`).

  Note A `Tobit` model with `"ccf"` can be unstable.
- `"lcf"` — Limit conversion factor (LCF) is a fraction of the limit representing the total exposure. The EAD is then defined as the LCF times the limit (`EAD = LCF*Limit`).

For more information on CCF and LCF, see “Conversion Measure Options” on page 6-658.

Data Types: `string` | `char`

`CensoringSide` — Censoring side
`"both"` (default) | character vector with value of `'left'`, `'right'`, or `'both'` | string with value of `"left"`, `"right"`, or `"both"`

Censoring side, specified as `CensoringSide` and a character vector or string. `CensoringSide` indicates whether the desired Tobit model is left-censored, right-censored, or censored on both sides.

Data Types: `string` | `char`

`LeftLimit` — Left-censoring limit
`0` (default) | numeric between `0` and `1`

Left-censoring limit, specified as `LeftLimit` and a scalar numeric between `0` and `1`.

Data Types: `double`

`RightLimit` — Right-censoring limit
`1` (default) | numeric between `0` and `1`

Right-censoring limit, specified as `RightLimit` and a scalar numeric between `0` and `1`.

Data Types: `double`

`SolverOptions` — `optimoptions` object
object

Options for fitting, specified as `SolverOptions` and an `optimoptions` object that is created using `optimoptions` from Optimization Toolbox™. The defaults for the `optimoptions` object are:

- `"Display"` — `"none"`
- `"Algorithm"` — `"sqp"`
- `"MaxFunctionEvaluations"` — `500` □ Number of model coefficients
- `"MaxIterations"` — The number of Tobit model coefficients is determined at run time; it depends on the number of predictors and the number of categories in the categorical predictors.

Note When using `optimoptions` with a Tobit model, specify the `SolverName` as `fmincon`.

Data Types: `object`

Properties

`ModelID` — User-defined model ID
`Tobit` (default) | string

User-defined model ID, returned as a string.

Data Types: `string`

`Description` — User-defined description
`""` (default) | string

User-defined description, returned as a string.

Data Types: `string`

`UnderlyingModel` — Underlying statistical model
compact linear model

This property is read-only.

Underlying statistical model, returned as a compact linear model object. The compact version of the underlying regression model is an instance of the `classreg.regr.CompactLinearModel` class. For more information, see `fitlm` and `CompactLinearModel`.

Data Types: `CompactLinearModel`

`PredictorVars` — Predictor variables
all columns of `data` except for the `ResponseVar` (default) | string array

Predictor variables, returned as a string array.

Data Types: `string`

`ResponseVar` — Response variable
last column of `data` (default) | string

Response variable, returned as a string.

Data Types: `string`

`LimitVar` — Limit variable
string

Limit variable, returned as a string.

Data Types: `string`

`DrawnVar` — Drawn variable
string

Drawn variable, returned as a string.

Data Types: `string`

`ConversionMeasure` — Conversion measure for EAD response values
`"ccf"` (default) | string with value of `"ccf"` or `"lcf"`

Response transform, returned as a string.

Data Types: `string`

`CensoringSide` — Censoring side
`"both"` (default) | string with value of `"left"`, `"right"`, or `"both"`

This property is read-only.

Censoring side, returned as a string.

Data Types: `string`

`LeftLimit` — Left-censoring limit
`0` (default) | numeric between `0` and `1`

This property is read-only.

Left-censoring limit, returned as a scalar numeric between `0` and `1`.

Data Types: `double`

`RightLimit` — Right-censoring limit
`1` (default) | numeric between `0` and `1`

This property is read-only.

Right-censoring limit, returned as a scalar numeric between `0` and `1`.

Data Types: `double`

Object Functions

predict	Predict exposure at default
modelDiscrimination	Compute AUROC and ROC data
modelDiscriminationPlot	Plot ROC curve
modelCalibration	Compute R-square, RMSE, correlation, and sample mean error of predicted and observed EADs
modelCalibrationPlot	Scatter plot of predicted and observed EADs

Examples

Create Tobit EAD Model

This example shows how to use `fitEADModel` to create a `Tobit` model for exposure at default (EAD).

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age      Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Tobit` or `Regression`.

```
ModelType = Tobit ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Tobit EAD Model

Use `fitEADModel` to create a `Tobit` model using the `EADData`.

```
eadModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'}, ...
    ConversionMeasure=ConversionMeasure,DrawnVar="Drawn",LimitVar="Limit",ResponseVar="EAD");
disp(eadModel);
```

```
  Tobit with properties:

        CensoringSide: "both"
            LeftLimit: 0
           RightLimit: 1
              ModelID: "Tobit"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: "Drawn"
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(eadModel.UnderlyingModel);
```

```
Tobit regression model:
     EAD_lcf = max(0,min(Y*,1))
     Y* ~ 1 + UtilizationRate + Age + Marriage

Estimated coefficients:
                             Estimate        SE         tStat       pValue  
                            __________    __________    ________    ________

    (Intercept)                0.22735      0.025254      9.0025           0
    UtilizationRate            0.47364      0.016435      28.818           0
    Age                     -0.0013929    0.00061973     -2.2477    0.024646
    Marriage_not married     -0.006888       0.01213    -0.56784     0.57017
    (Sigma)                    0.36419     0.0038798      93.868           0

Number of observations: 4378
Number of left-censored observations: 0
Number of uncensored observations: 4377
Number of right-censored observations: 1
Log-likelihood: -1791.06
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-vale argument.

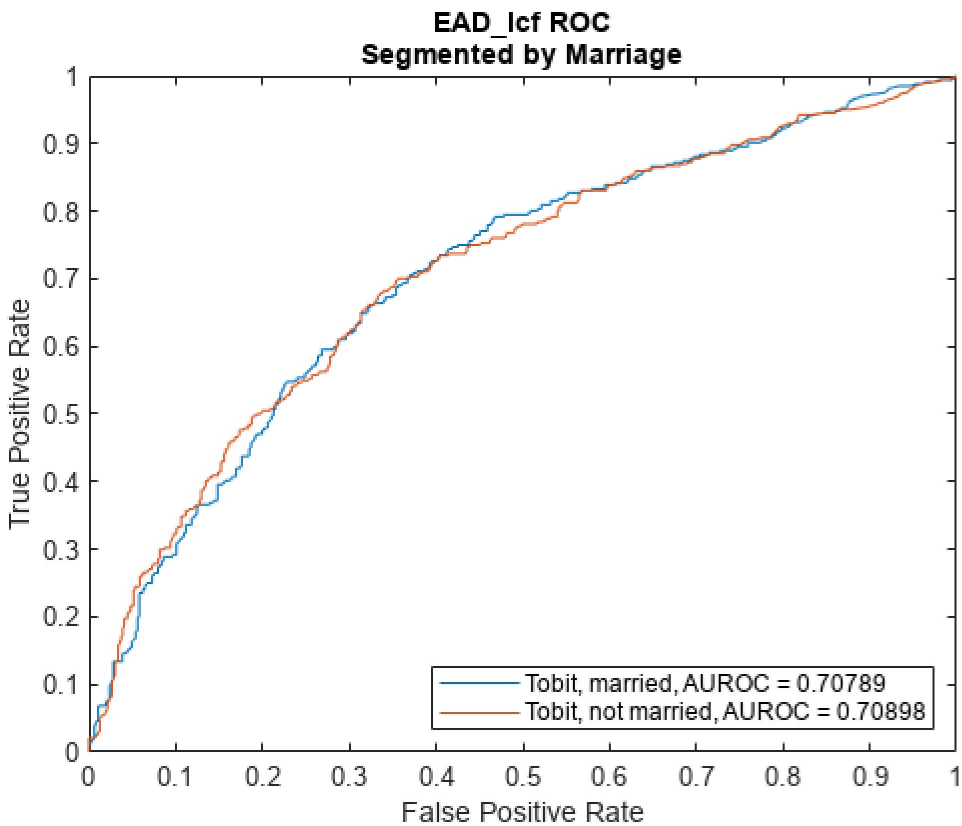
```
predictedEAD = predict(eadModel,EADData(TestInd,:),ModelLevel="ead");
predictedConversion = predict(eadModel,EADData(TestInd,:),ModelLevel="ConversionMeasure");
```

Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

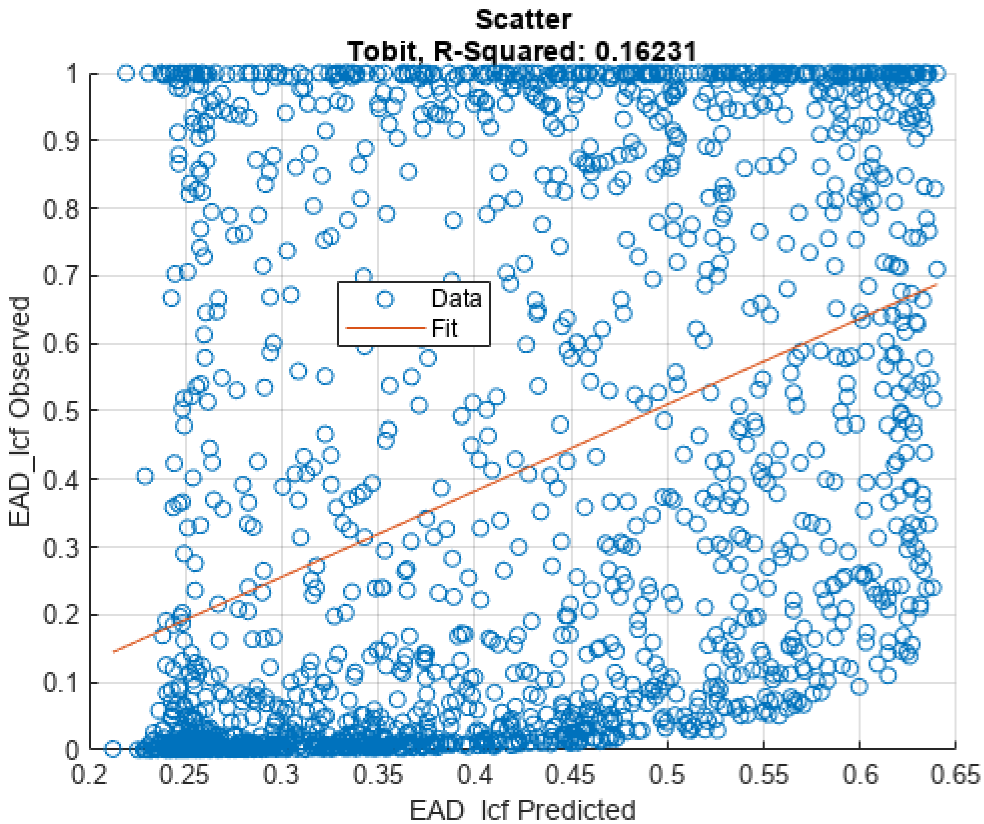
Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

```
ModelLevel = ConversionMeasure ;

[DiscMeasure1,DiscData1] = modelDiscrimination(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel
modelDiscriminationPlot(eadModel,EADData(TestInd, :),ModelLevel=ModelLevel,SegmentBy="Marriage")
```



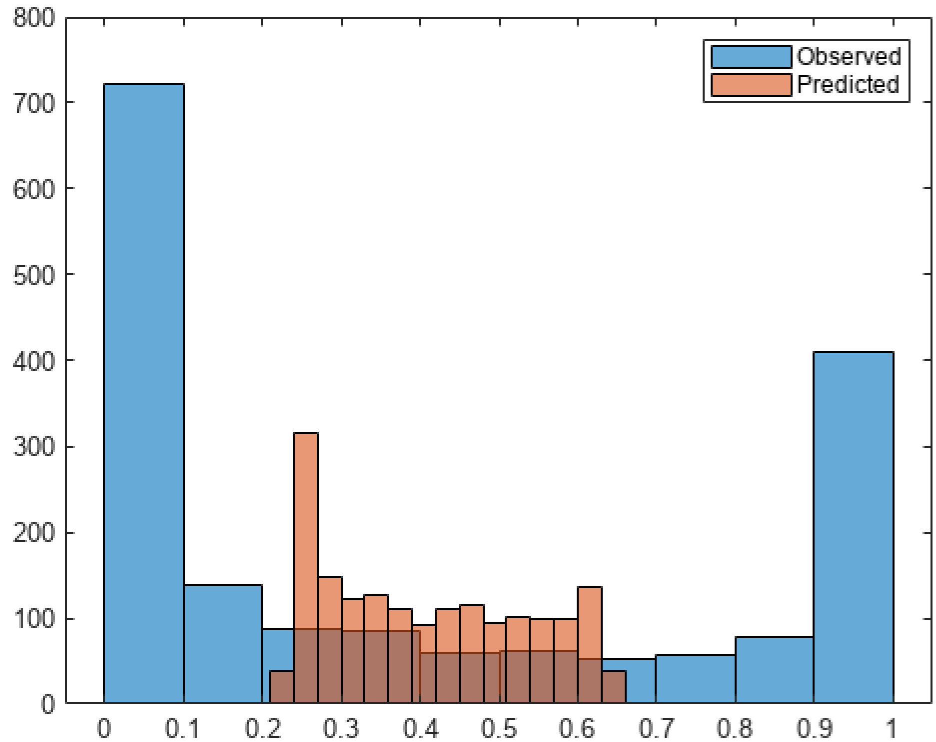
Use `modelCalibration` and then `modelCalibrationPlot` to show a scatter plot of the predictions.

```
YData = Observed ;

[CalMeasure1,CalData1] = modelCalibration(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel);
modelCalibrationPlot(eadModel,EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Plot a histogram of observed with respect to the predicted EAD.

```
figure;
histogram(CalData1.Observed);
hold on;
histogram(CalData1.(('Predicted_' + ModelType)));
legend('Observed','Predicted');
```



More About

Exposure at Default Tobit Models

The exposure at default (EAD) Tobit models fit a Tobit model to EAD data.

Tobit models are "censored" regression models. Tobit models assume that the response variable can be observed only within certain limits, and no value outside the limits can be observed. Using `ModelLevel`, you can set the Tobit model level to `EAD`, `CCF`, or `LCF` conversion measures. The `EAD` model level does not have any range, the `CCF` conversion measure has a range of `-Inf` to `1`, and the `LCF` conversion measure is `0` to `1`. A distribution of response values where there is a high frequency of observations at the limits is consistent with the model assumptions.

The Tobit model combines the following two formulas:

$$Y = \min\{\max\{L, Y*\}, R\}$$
$$Y* = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \sigma\varepsilon = X\beta + \sigma\varepsilon$$

where

- Y is the observed response variable, the observed EAD data for an EAD model.
- L is the left limit, the lower bound for the response values, typically `0` for EAD models.
- R is the right limit, the upper bound for the response values, typically `1` for EAD models.

- Y^* is a latent, unobserved variable.
- β_j is the coefficient of the jth predictor (or the intercept for j = 0).
- σ is the standard deviation of the error term.
- ϵ is the error term, assumed to follow a standard normal distribution.

The first formula above is written using min and max operators and is equivalent to

$$Y = \begin{Bmatrix} L & \text{if } Y* \leq L \\ Y* & \text{if } L < Y* < R \\ R & \text{if } Y* \geq R \end{Bmatrix}$$

The standard deviation of the error is explicitly indicated in the formulas. Unlike traditional regression least-squares estimation, where the standard deviation of the error can be inferred from the residuals, for Tobit models the estimation is via maximum likelihood and the standard deviation needs to be handled explicitly during the estimation. If there are p predictor variables, the Tobit model estimates p+2 coefficients, namely, one coefficient for each predictor, plus an intercept, plus a standard deviation.

Three censoring side options are supported in the Tobit EAD models with the CensoringSide name-value argument:

- 'both' — This option is the default option, with censoring on both sides. The estimation uses left and right limits.
- 'left' — The left-censored version of the model has no right limit (or $R = \infty$). The relationship between Y and Y^* is Y = maxâ¡$\{L,Y^*\}$.
- 'right' — The right-censored version of the model has no left limit (or $L = -\infty$). The relationship between Y and Y^* is Y = min$\{Y^*,R\}$.

The parameters of the Tobit model are estimated using maximum likelihood. For observation i = 1,...,n, the likelihood function is

$$LF(\beta, \sigma \middle| X_i, Y_i) = \begin{Bmatrix} \Phi(L; X_i\beta,\sigma) & \text{if } Y_i \leq L \\ \phi(Y_i;X_i\beta,\sigma) & \text{if } L < Y_i < R \\ 1 - \Phi(R; X_i\beta, \sigma) & \text{if } Y_i \geq R \end{Bmatrix}$$

where

- $\Phi(x;m,s)$ is the cumulative normal distribution with mean m and standard deviation s.
- $\varphi(x;m,s)$ is the normal density function with mean m and standard deviation s.

This likelihood function is for models censored on both sides. For left-censored models, the right limit has no effect, and the likelihood function has two cases only ($R = \infty$); likewise for right-censored models ($L = -\infty$).

The log-likelihood function is the sum of the logarithm of the likelihood functions for individual observations

$$LLF(\beta, \sigma \middle| X, Y) = \sum_{i=1}^{n} \log(LF(\beta, \sigma \middle| X_i, Y_i))$$

The parameters are estimated by maximizing the log-likelihood function. The only constraint is that the Ïƒ parameter must be positive.

To predict an EAD value, Tobit EAD models return the unconditional expected value of the response, given the predictor values

$$EAD_i^{pred} = E[Y_i | X_i]$$

The expression for the expected value can be separated into the cases

$$E[Y] = E[Y | Y = L]P(Y = L)$$
$$+E[Y | L < Y < R]P(L < Y < R)$$
$$+E[Y | Y = R]P(Y = R)$$

Using the previous expression and the properties of the (truncated) normal distribution, it follows that

$$E[Y_i | X_i] = \Phi(a_i)L + (\Phi(b_i) - \Phi(a_i))(X_i\beta + \sigma\lambda_i) + (1 - \Phi(b_i))R$$

where

$$a_i = \frac{L - X_i\beta}{\sigma}, b_i = \frac{R - X_i\beta}{\sigma}, \text{ and } \lambda_i = \frac{\phi(a_i) - \phi(b_i)}{\Phi(b_i) - \Phi(a_i)}$$

This expression applies to the models censored on both sides. For models censored on one side only, the corresponding expressions can be derived from here. For example, for left-censored models, let the R limit in the expression above go to infinity, and the resulting expression is

$$E[Y_i | X_i] = \Phi(a_i)L + (1 - \Phi(a_i))\left(X_i\beta + \sigma \frac{\phi(a_i)}{1 - \Phi(a_i)}\right)$$

Similarly, for right-censored models, the L limit is decreased to minus infinity to get

$$E[Y_i | X_i] = \Phi(b_i)\left(X_i\beta - \sigma \frac{\phi(b_i)}{\Phi(b_i)}\right) + (1 - \Phi(b_i))R$$

Conversion Measure Options

You can relate the EAD to a scaling variable and derive conversion measures like credit conversion factor (CCF) and limit conversion factor (LCF) using the `'ccf'` or `'lcf'` options for the `ConversionMeasure` name-value argument.

The following table summarizes the supported transformations using the `'ccf'` or `'lcf'` options for the `ConversionMeasure` name-value argument:

Measure	EAD Formula	Lower Bound	Upper Bound	Inverse Transformation
CCF	`EAD = Drawn + CCF Ã— (Limit - Drawn)`	-Inf	1	`CCF = 1 - e`$^{(-CCF_t)}$
LCF	`EAD = LCF  Limit`	0	1	`LCF = e`^{LCF_t} `â^• (1 + e`^{LCF_t}`)`

Version History

Introduced in R2021b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function
Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function
Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

Functions
`fitEADModel` | `Regression` | `Beta`

Topics
“Compare Results for Regression and Tobit EAD Models” on page 4-151
“Overview of Loss Given Default Models” on page 1-31

Beta

Create `Beta` model object for exposure at default

Description

Create and analyze a `Beta` model object to calculate the exposure at default (EAD) using this workflow:

1. Use `fitEADModel` to create a `Beta` model object.
2. Use `predict` to predict the EAD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the R-squared, RMSE, correlation, and sample mean error of predicted and observed EAD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
BetaEADModel = fitEADModel(data,ModelType)
BetaEADModel = fitEADModel( ___ ,Name=Value)
```

Description

`BetaEADModel = fitEADModel(data,ModelType)` creates a `Beta` EAD model object.

`BetaEADModel = fitEADModel( ___ ,Name=Value)` specifies options using one or more name-value arguments in addition to the input arguments in the previous syntax. The optional name-value arguments set the model object properties on page 6-663. For example, `BetaEADModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'},ConversionMeasure="lcf",DrawnVar='Drawn',LimitVar='Limit',ResponseVar='EAD')` creates an `BetaEADModel` object using a `Beta` model type.

Input Arguments

`data` — Data for exposure at default
table

Data for exposure at default, specified as a table.

Data Types: `table`

`ModelType` — Model type
string with value `"Beta"` | character vector with value `'Beta'`

Model type, specified as a string with the value of `"Beta"` or a character vector with the value of `'Beta'`.

Data Types: char | string

Beta Name-Value Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: BetaEADModel = fitEADModel(EADData,ModelType,PredictorVars={'UtilizationRate','Age','Marriage'},ConversionMeasure="lcf",LimitVar='Limit',ResponseVar='EAD',BoundaryTolerance=1e5)

ModelID — User-defined model ID
"Beta" (default) | string | character vector

User-defined model ID, specified as ModelID and a string or character vector. The software uses the ModelID text to format outputs and is expected to be short.

Data Types: string | char

Description — User-defined description for model
"" (default) | string | character vector

User-defined description for model, specified as Description and a string or character vector.

Data Types: string | char

PredictorVars — Predictor variables
all columns of data except for ResponseVar (default) | string array | cell array of character vectors

Predictor variables, specified as PredictorVars and a string array or cell array of character vectors. PredictorVars indicates which columns in the data input contain the predictor information. By default, PredictorVars is set to all the columns in the data input except for ResponseVar.

Data Types: string | cell

ResponseVar — Response variable
last column of data (default) | string | character vector

Response variable, specified as ResponseVar and a string or character vector. The response variable contains the EAD data and must be a numeric variable. By default, ResponseVar is set to the last column.

Data Types: string | char

BoundaryTolerance — Value to perturb EAD responses
1e-7 (default) | positive numeric

Value to perturb EAD response values away from 0 to 1, specified as BoundaryTolerance and a positive scalar numeric.

Data Types: double

LimitVar — Limit variable
string | character vector

Limit variable, specified as `LimitVar` and a string or character vector. `LimitVar` indicates which column in `data` contains the limit amount. The limit amount value in the `data` must be a positive numeric value. The limit depends on the loan. If the loan is a credit card, the limit is the credit limit. If the loan is a mortgage, the limit is the initial loan amount. In general, `LimitVar` is the maximum amount that can be borrowed.

Note `LimitVar` is required when `ConversionMeasure` is `'lcf'`. For more information on LCF, see “Conversion Measure Options” on page 6-658.

Data Types: `string` | `char`

`DrawnVar` — Drawn variable
string | character vector

Drawn variable, specified as `DrawnVar` and a string or character vector. `DrawnVar` is the balance on the account at the time of observation, before default, and EAD is the balance at the time of default. `DrawnVar` indicates which column in `data` contains the drawn amount. The drawn variable value in the `data` can be a positive or negative numeric value.

Note When the `ConversionMeasure` is `'lcf'`, `DrawnVar` is not required. In this case, `DrawnVar` is set to `""`.

Data Types: `string` | `char`

`ConversionMeasure` — Conversion measure for EAD response values
`"lcf"` (default) | character vector with value `'lcf'` | string with value `"lcf"`

Response transform, specified as `ConversionMeasure` and a character vector or string. Limit conversion factor (LCF) is a fraction of the limit representing the total exposure. The EAD is then defined as the LCF times the limit (`EAD = LCF*Limit`).

For more information on LCF, see “Conversion Measure Options” on page 6-658.

Data Types: `string` | `char`

`SolverOptions` — `optimoptions` object
object

Options for fitting, specified as `SolverOptions` and an `optimoptions` object that is created using `optimoptions` from Optimization Toolbox. The defaults for the `optimoptions` object are:

- `"Display"` — `"none"`
- `"Algorithm"` — `"quasi-newton"`
- `"MaxFunctionEvaluations"` — `500` × Number of model coefficients
- `"MaxIterations"` — 1000

Note When using `optimoptions` with a `Beta` model, specify the `SolverName` as `fminunc`.

The number of `Beta` model coefficients is determined at run time, depending on the number of predictors and the number of categories in the categorical predictors.

Data Types: `object`

Properties

`ModelID` — User-defined model ID
`Beta` (default) | string

User-defined model ID, returned as a string.

Data Types: `string`

`Description` — User-defined description
`""` (default) | string

User-defined description, returned as a string.

Data Types: `string`

`UnderlyingModel` — Underlying statistical model
compact linear model

This property is read-only.

Underlying statistical model, returned as a compact linear model object. The compact version of the underlying regression model is an instance of the `risk.internal.credit.BetaModel` class.

Data Types: `object`

`PredictorVars` — Predictor variables
all columns of `data` except for the `ResponseVar` (default) | string array

Predictor variables, returned as a string array.

Data Types: `string`

`ResponseVar` — Response variable
last column of `data` (default) | string

Response variable, returned as a string.

Data Types: `string`

`LimitVar` — Limit variable
string

Limit variable, returned as a string.

Data Types: `string`

`DrawnVar` — Drawn variable
string

Drawn variable, returned as a string.

Data Types: `string`

ConversionMeasure — Conversion measure for EAD response values
"lcf" (default) | string with value "lcf"

Response transform, returned as a string.

Data Types: string

BoundaryTolerance — Value to perturb LGD responses
1e-7 (default) | positive numeric

Value to perturb LGD response values away from 0 to 1, returned as a positive scalar numeric.

Data Types: double

Object Functions

predict	Predict exposure at default
modelDiscrimination	Compute AUROC and ROC data
modelDiscriminationPlot	Plot ROC curve
modelCalibration	Compute R-square, RMSE, correlation, and sample mean error of predicted and observed EADs
modelCalibrationPlot	Scatter plot of predicted and observed EADs

Examples

Create Beta EAD Model

This example shows how to use `fitEADModel` to create a `Beta` model object for exposure at default (EAD).

Load EAD Data

Load the EAD data.

```
load EADData.mat
head(EADData)
```

```
    UtilizationRate    Age     Marriage        Limit         Drawn          EAD    
    _______________    ___    ___________    __________    __________    __________

        0.24359        25     not married         44776         10907         44740
        0.96946        44     not married    2.1405e+05    2.0751e+05         40678
              0        40     married        1.6581e+05             0    1.6567e+05
        0.53242        38     not married    1.7375e+05         92506        1593.5
         0.2583        30     not married         26258        6782.5        54.175
        0.17039        54     married        1.7357e+05         29575        576.69
        0.18586        27     not married         19590          3641        998.49
        0.85372        42     not married    2.0712e+05    1.7682e+05    1.6454e+05
```

```
rng('default');
NumObs = height(EADData);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Select Model Type

Select a model type for `Beta`.

```
ModelType = Beta ;
```

Select Conversion Measure

Select a conversion measure for the EAD response values.

```
ConversionMeasure = LCF ;
```

Create Beta EAD Model

Use `fitEADModel` to create a `Beta` model object using the `TrainingInd` data.

```
BetaEADModel = fitEADModel(EADData(TrainingInd,:),ModelType,PredictorVars={'UtilizationRate','Age
    ConversionMeasure=ConversionMeasure,LimitVar="Limit",ResponseVar="EAD",BoundaryTolerance=2e-0
disp(BetaEADModel);
```

```
  Beta with properties:

    BoundaryTolerance: 2.0000e-05
              ModelID: "Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["UtilizationRate"    "Age"    "Marriage"]
          ResponseVar: "EAD"
             LimitVar: "Limit"
             DrawnVar: ""
    ConversionMeasure: "lcf"
```

Display the underlying model. The underlying model's response variable is the transformation of the EAD response data. Use the `'LimitVar'` and `'DrawnVar'` name-value arguments to modify the transformation.

```
disp(BetaEADModel.UnderlyingModel);
```

```
Beta regression model:
     logit(EAD_lcf) ~ 1_mu + UtilizationRate_mu + Age_mu + Marriage_mu
     log(EAD_lcf) ~ 1_phi + UtilizationRate_phi + Age_phi + Marriage_phi

Estimated coefficients:
                                 Estimate        SE         tStat        pValue  
                                __________    _________    _________    __________

    (Intercept)_mu                -0.68477       0.1145      -5.9807    2.5234e-09
    UtilizationRate_mu              1.7029     0.077717       21.912             0
    Age_mu                      -0.0056329    0.0027489      -2.0492      0.040543
    Marriage_not married_mu      -0.025614     0.051927     -0.49328       0.62186
    (Intercept)_phi               -0.46429     0.095342      -4.8697    1.1838e-06
    UtilizationRate_phi            0.41621      0.06701       6.2112    6.0942e-10
    Age_phi                      -0.001282    0.0023261     -0.55114       0.58159
    Marriage_not married_phi    0.00014873     0.042884    0.0034682       0.99723

Number of observations: 2627
Log-likelihood: -2931.19
```

Predict EAD

EAD prediction operates on the underlying compact statistical model and then transforms the predicted values back to the EAD scale. You can specify the `predict` function with different options for the `'ModelLevel'` name-vale argument.

```
predictedEAD = predict(BetaEADModel,EADData(TestInd,:))
```

```
predictedEAD = 1751×1
10^5 ×

    0.1758
    0.1029
    0.1528
    0.0832
    0.3261
    0.5148
    0.0648
    0.0531
    0.0712
    0.3215
      ⋮
```

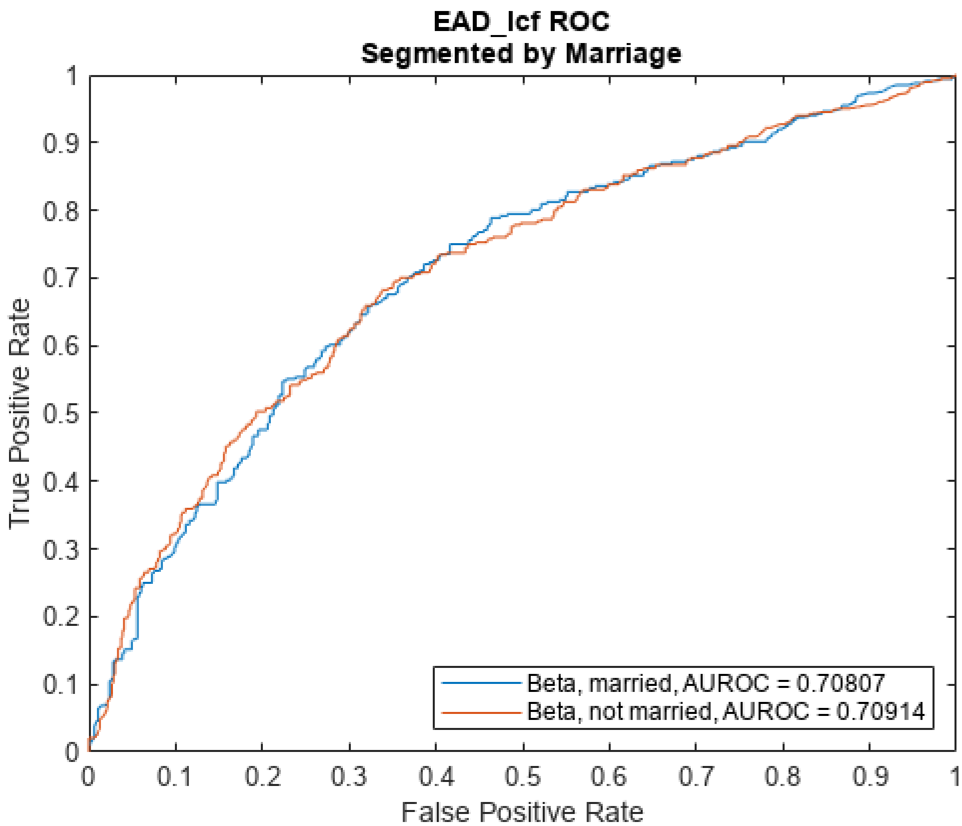
Validate EAD Model

For model validation, use `modelDiscrimination`, `modelDiscriminationPlot`, `modelCalibration`, and `modelCalibrationPlot`.

Use `modelDiscrimination` and then `modelDiscriminationPlot` to plot the ROC curve.

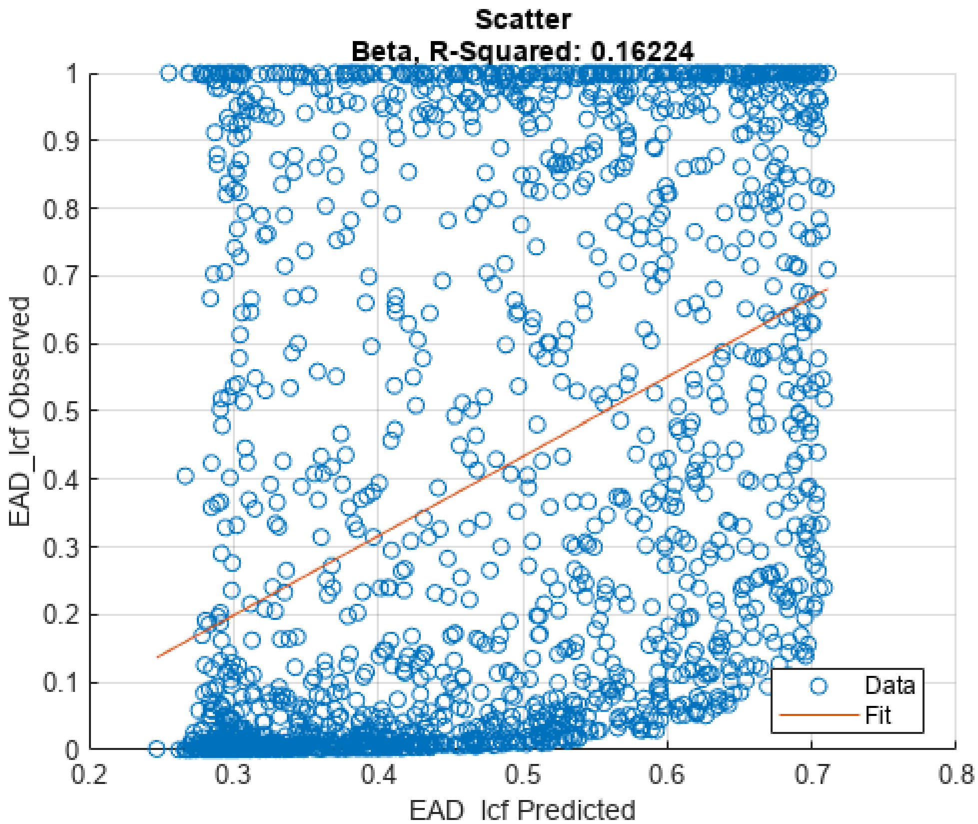
```
ModelLevel = ConversionMeasure ;
```

```
[DiscMeasure1,DiscData1] = modelDiscrimination(BetaEADModel,EADData(TestInd,:),ModelLevel=ModelLe
modelDiscriminationPlot(BetaEADModel,EADData(TestInd, :),ModelLevel=ModelLevel,SegmentBy="Marriag
```



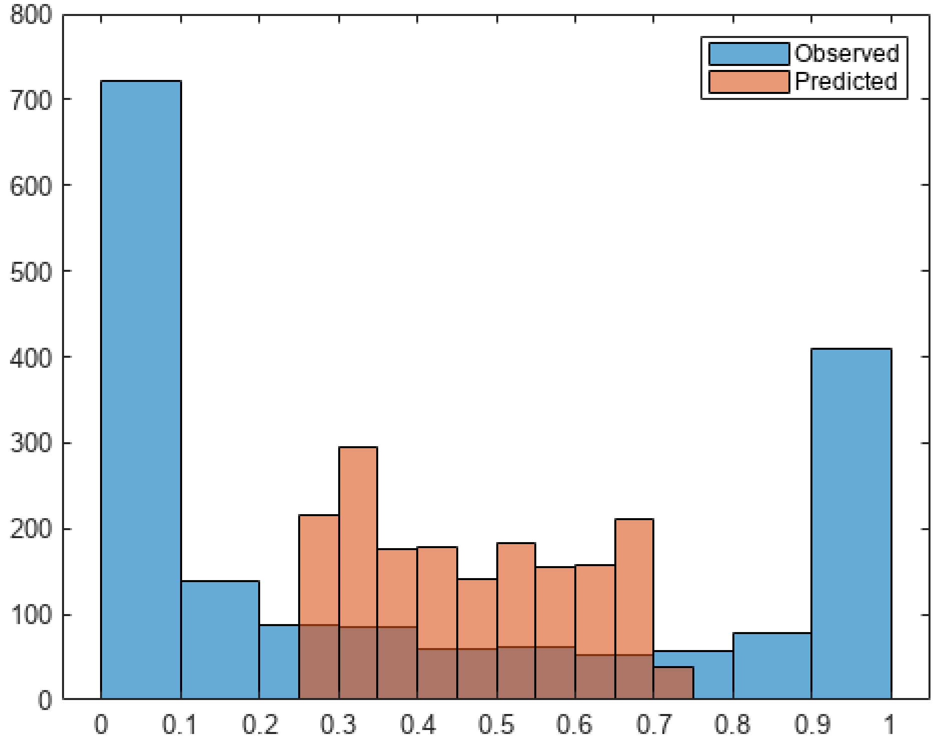
Use `modelCalibration` and then `modelCalibrationPlot` to show a scatter plot of the predictions.

```
YData = Observed;

[CalMeasure1,CalData1] = modelCalibration(BetaEADModel,EADData(TestInd,:),ModelLevel=ModelLevel)
modelCalibrationPlot(BetaEADModel,EADData(TestInd,:),ModelLevel=ModelLevel,YData=YData);
```



Plot a histogram of observed EAD with respect to the predicted EAD.

```
figure;
histogram(CalData1.Observed);
hold on;
histogram(CalData1.(('Predicted_' + ModelType)));
legend('Observed','Predicted');
```



More About

Beta Regression Models

Beta regression models model continuous variables that assume values in the open standard unit interval (0, 1).

The beta regression model is based on the assumption that the dependent variable is beta-distributed and that its mean is related to a set of regressors through a linear predictor with unknown coefficients and a link function. The beta regression model also includes a precision parameter that may be constant or depend on a potentially different set of regressors through a link function as well.

If the variable takes on values in (a, b) with $a < b$, you can model (y - a) / (b - a). Also, if y also assumes the extremes 0 and 1, a useful transformation in practice is y (n - 1) + 0.5) / n, where n is the sample size.

The beta regression model is based on an alternative parameterization of the beta density in terms of the mean and a precision parameter: $\mu = a / (a + b)$ and $\varphi = a + b$, $0 < \mu < 1$, $\varphi > 0$.

$$f(y; \mu, \phi) = \frac{\Gamma(\phi)}{\Gamma(\mu\phi)\Gamma((1-\mu)\phi)} y^{\mu\phi - 1}(^{1}, \quad 0<y<1,$$

$$y \sim \mathrm{B}(\mu,\varphi) \text{ with } \mathrm{E}(y)=\mu,\ \mathrm{VAR}(\varphi)=\mu(1\text{-}\mu)/(1+\varphi)$$

The beta regression model is defined as

$$g_1(\mu_i) = \eta_{1i} = x_i^T \beta,$$

$$g_2(\phi_i) = \eta_{2i} = z_i^T \gamma,$$

where $\beta = (\beta_0,\beta_1,\ldots,\beta_k)^T$, $\gamma = (\gamma_0,\gamma_1,\ldots,\gamma_k)^T$ are the regression coefficients, η_{1i}, η_{2i} are the linear predictors, and x_i, z_i are the independent variables. g_1 and g_2 are the link functions where g_1 is logit and g_2 is log.

Parameter estimation is performed by maximum likelihood (ML).

Version History

Introduced in R2022b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

[3] Brown, Iain. *Developing Credit Risk Models Using SAS Enterprise Miner and SAS/STAT: Theory and Applications.* SAS Institute, 2014.

[4] Roesch, Daniel and Harald Scheule. *Deep Credit Risk.* Independently published, 2020.

See Also

Functions

`fitEADModel` | `Regression` | `Tobit`

Topics

"Compare Results for Regression and Tobit EAD Models" on page 4-151
"Overview of Loss Given Default Models" on page 1-31

Regression

Create `Regression` model object for loss given default

Description

Create and analyze a `Regression` model object to calculate the loss given default (LGD) using this workflow:

1. Use `fitLGDModel` to create a `Regression` model object.
2. Use `predict` to predict the LGD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the R-square, RMSE, correlation, and sample mean error of the predicted and observed LGD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
RegressionLGDModel = fitLGDModel(data,ModelType)
RegressionLGDModel = fitLGDModel(___,Name,Value)
```

Description

`RegressionLGDModel = fitLGDModel(data,ModelType)` creates a `Regression` LGD model object.

`RegressionLGDModel = fitLGDModel(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The optional name-value pair arguments set model object properties on page 6-673. For example, `lgdModel = fitLGDModel(data,'regression','PredictorVars',{'LTV' 'Age' 'Type'},'ResponseVar','LGD','ResponseTransform','probit','BoundaryTolerance',1e-6)` creates a `lgdModel` object using a `Regression` model type.

Input Arguments

`data` — Data for loss given default
table

Data for loss given default, specified as a table where the first column and all other columns except the last column are `PredictorVars`, the last column is `ResponseVar`.

Data Types: `table`

`ModelType` — Model type
string with value `"Regression"` | character vector with value `'Regression'`

Model type, specified as a string with the value of `"Regression"` or a character vector with the value of `'Regression'`.

Data Types: `char` | `string`

`Regression` Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `lgdModel = fitLGDModel(data,'regression','PredictorVars',{'LTV' 'Age' 'Type'},'ResponseVar','LGD','ResponseTransform','probit','BoundaryTolerance',1e-6)`

`ModelID` — User-defined model ID
`"Regression"` (default) | string | character vector

User-defined model ID, specified as the comma-separated pair consisting of `'ModelID'` and a string or character vector. The software uses the `ModelID` text to format outputs and is expected to be short.

Data Types: `string` | `char`

`Description` — User-defined description for model
`""` (default) | string | character vector

User-defined description for model, specified as the comma-separated pair consisting of `'Description'` and a string or character vector.

Data Types: `string` | `char`

`PredictorVars` — Predictor variables
all columns of `data` except for the `ResponseVar` (default) | string array | cell array of character vectors

Predictor variables, specified as the comma-separated pair consisting of `'PredictorVars'` and a string array or cell array of character vectors. `PredictorVars` indicates which columns in the `data` input contain the predictor information. By default, `PredictorVars` is set to all the columns in the `data` input except for the `ResponseVar`.

Data Types: `string` | `cell`

`ResponseVar` — Response variable
last column of `data` (default) | string | character vector

Response variable, specified as the comma-separated pair consisting of `'ResponseVar'` and a string or character vector. The response variable contains the LGD data and must be a numeric variable. An LGD value of `0` indicates no loss (full recovery), `1` indicates total loss (no recovery), and values between `0` and `1` indicate a partial loss. By default, the `ResponseVar` is set to the last column of `data`.

Data Types: `string` | `char`

`BoundaryTolerance` — Boundary tolerance
`1e-5` (default) | positive numeric

Boundary tolerance, specified as the comma-separated pair consisting of 'BoundaryTolerance' and a positive scalar numeric. The BoundaryTolerance value perturbs the LGD response values away from 0 and 1, before applying a ResponseTransform.

Data Types: double

ResponseTransform — Response transform
"logit" (default) | character vector with value of'logit', 'probit', or 'log' | string with value of "logit", "probit", or "log"

Response transform, specified as the comma-separated pair consisting of 'ResponseTransform' and a character vector or string.

Data Types: string | char

Properties

ModelID — User-defined model ID
"Regression" (default) | string

User-defined model ID, returned as a string.

Data Types: string

Description — User-defined description
"" (default) | string

User-defined description, returned as a string.

Data Types: string

UnderlyingModel — Underlying statistical model
compact linear model

Underlying statistical model, returned as a compact linear model object. The compact version of the underlying regression model is an instance of the classreg.regr.CompactLinearModel class. For more information, see fitlm and CompactLinearModel.

Data Types: CompactLinearModel

PredictorVars — Predictor variables
all columns of data except for ResponseVar (default) | string array

Predictor variables, returned as a string array.

Data Types: string

ResponseVar — Response variable
last column of data (default) | string

Response variable, returned as a scalar string.

Data Types: string

BoundaryTolerance — Boundary tolerance
1e-5 (default) | numeric

This property is read-only.

Boundary tolerance, returned as a scalar numeric.

Data Types: `double`

`ResponseTransform` — Response transform
`"logit"` (default) | string

This property is read-only.

Response transform, returned as a string.

Data Types: `string`

Object Functions

predict	Predict loss given default
modelDiscrimination	Compute AUROC and ROC data
modelDiscriminationPlot	Plot ROC curve
modelCalibration	Compute R-square, RMSE, correlation, and sample mean error of predicted and observed LGDs
modelCalibrationPlot	Scatter plot of predicted and observed LGDs

Examples

Create Regression LGD Model

This example shows how to use `fitLGDModel` to create a `Regression` model for loss given default (LGD).

Load LGD Data

Load the LGD data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age         Type           LGD    
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

```
rng('default');
NumObs = height(data);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Regression LGD Model

Use `fitLGDModel` to create a `Regression` model using the `TrainingInd` data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'regression',...
        'ModelID','Example Probit',...
        'Description','Example LGD probit regression model.',...
        'PredictorVars',{'LTV' 'Age' 'Type'},...
        'ResponseVar','LGD','ResponseTransform','probit','BoundaryTolerance',1e-6);
disp(lgdModel)
```

```
  Regression with properties:

    ResponseTransform: "probit"
    BoundaryTolerance: 1.0000e-06
              ModelID: "Example Probit"
          Description: "Example LGD probit regression model."
      UnderlyingModel: [1x1 classreg.regr.CompactLinearModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model. The underlying model's response variable is the probit transformation of the LGD response data. Use the `'ResponseTransform'` and `'BoundaryTolerance'` arguments to modify the transformation.

```
lgdModel.UnderlyingModel
```

```
ans = 
Compact linear regression model:
    LGD_probit ~ 1 + LTV + Age + Type

Estimated Coefficients:
                       Estimate       SE        tStat       pValue  
                       ________    ________    _______    __________

    (Intercept)         -2.2212     0.14824    -14.984    2.7409e-48
    LTV                   1.192     0.17183     6.9372    5.3152e-12
    Age                -0.60356    0.035256    -17.119    1.3828e-61
    Type_investment     0.61814      0.1018     6.0723    1.4933e-09


Number of observations: 2093, Error degrees of freedom: 2089
Root Mean Squared Error: 1.74
R-squared: 0.197,  Adjusted R-Squared: 0.196
F-statistic vs. constant model: 171, p-value = 4.39e-99
```

Predict LGD

For LGD prediction, use `predict`. The LGD model applies the inverse transformation so the predictions are in the LGD scale, not in the transformed scale used to fit the underlying model.

```
predictedLGD = predict(lgdModel,data(TestInd,:))
```

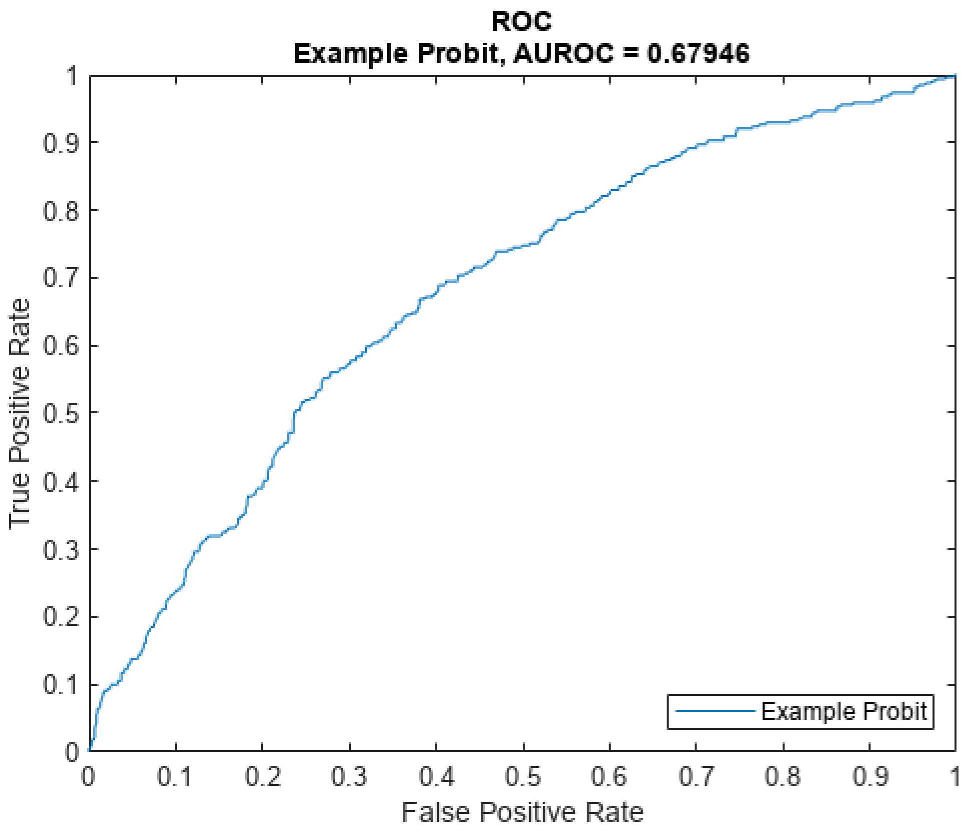
```
predictedLGD = 1394×1

    0.0011
    0.0058
    0.2091
    0.0015
    0.0192
    0.0620
    0.0757
```

```
0.0000
0.0156
0.0388
  ⋮
```

Validate LGD Model

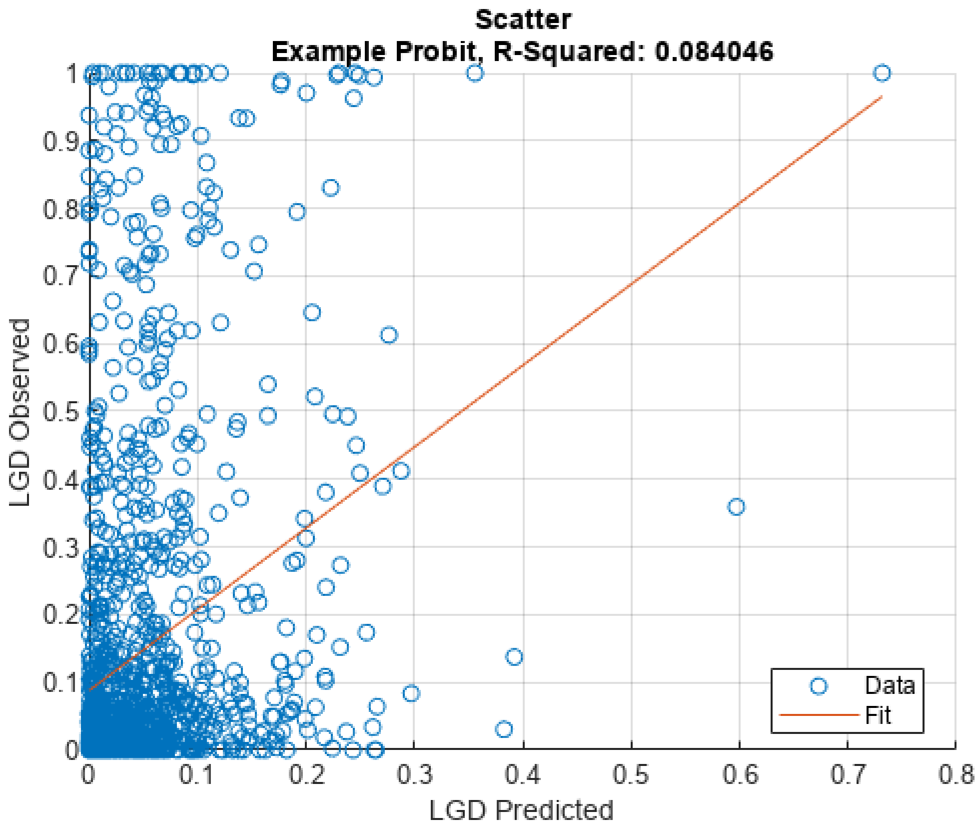
Use `modelDiscriminationPlot` to plot the ROC curve.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:))
```



Use `modelCalibrationPlot` to show a scatter plot of the predictions.

```
modelCalibrationPlot(lgdModel,data(TestInd,:))
```



More About

Loss Given Default Regression Models

You can transform LGD data using linear regression models.

The loss given default (LGD) regression models transform the LGD response variable

$$TransformedLGD = f(LGD)$$

and fit a linear regression model

$$\begin{aligned} TransformedLGD &= \beta_0 + \beta_1 X_1 + \ldots\beta_p X_p + \varepsilon \\ &= \mathrm{X}\beta + \varepsilon \end{aligned}$$

For prediction, the models first predict on the transformed space using the underlying linear regression model and the estimated coefficients $\hat{\beta}$

$$TransformedLGD_{pred} = X\hat{\beta}$$

You can then apply the inverse transformation to return predictions on the LGD scale

$$LGD_{pred} = f^{-1}(TransformedLGD_{pred})$$

The following table summarizes the supported transformations using the `ResponseTransform` name-value argument and their corresponding inverses, where $\Phi(x)$ is the cumulative normal distribution:

'ResponseTransform'	$\boldsymbol{f(LGD)}$	$\boldsymbol{f^{-1}(TransformedLGD)}$
'logit'	$TransformedLGD = \log\left(\frac{LGD}{1 - LGD}\right)$	$LGD = \frac{1}{1 + \exp\left(-TransformedLGD\right)}$
'probit'	$TransformedLGD = \Phi^{-1}(LGD)$	$LGD = \Phi(TransformedLGD)$
'log'	$TransformedLGD = \log\left(LGD\right)$	$LGD = \exp\left(TransformedLGD\right)$

Version History

Introduced in R2021a

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

Functions

`fitLGDModel` | `Tobit` | `Beta`

Topics

"Model Loss Given Default" on page 4-90
"Basic Loss Given Default Model Validation" on page 4-131
"Compare Tobit LGD Model to Benchmark Model" on page 4-133
"Compare Loss Given Default Models Using Cross-Validation" on page 4-140
"Overview of Loss Given Default Models" on page 1-31

Beta

Create `Beta` model object for loss given default

Description

Create and analyze a `Beta` model object to calculate loss given default (LGD) using this workflow:

1. Use `fitLGDModel` to create a `Beta` model object.
2. Use `predict` to predict the LGD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the R-squared, RMSE, correlation, and sample mean error of predicted and observed LGD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
BetaLGDModel = fitLGDModel(data,ModelType)
BetaLGDModel = fitLGDModel(___,Name=Value)
```

Description

`BetaLGDModel = fitLGDModel(data,ModelType)` creates a `Beta` LGD model object.

`BetaLGDModel = fitLGDModel(___,Name=Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The optional name-value pair arguments set the model object properties on page 6-681. For example, `BetaLGDModel = fitLGDModel(data,'Beta',PredictorVars={'LTV' 'Age' 'Type'},ResponseVar='LGD',BoundaryTolerance=1e-4)` creates a `BetaLGDModel` object using a `Beta` model type.

Input Arguments

`data` — Data for loss given default
table

Data for loss given default, specified as a table.

Data Types: `table`

`ModelType` — Model type
string with value `"Beta"` | character vector with value `'Beta'`

Model type, specified as a string with the value of `"Beta"` or a character vector with the value of `'Beta'`.

Data Types: `char` | `string`

Beta Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `BetaLGDModel = fitLGDModel(data,'Beta',PredictorVars={'LTV' 'Age' 'Type'},ResponseVar='LGD',BoundaryTolerance=1e-4)`

`ModelID` — User-defined model ID
`"Beta"` (default) | string | character vector

User-defined model ID, specified as `ModelID` and a string or character vector. The software uses the `ModelID` text to format outputs and is expected to be short.

Data Types: `string` | `char`

`Description` — User-defined description for model
`""` (default) | string | character vector

User-defined description for model, specified as `Description` and a string or character vector.

Data Types: `string` | `char`

`PredictorVars` — Predictor variables
all columns of `data` except for `ResponseVar` (default) | string array | cell array of character vectors

Predictor variables, specified as `PredictorVars` and a string array or cell array of character vectors. `PredictorVars` indicates which columns in the `data` input contain the predictor information. By default, `PredictorVars` is set to all the columns in the `data` input except for `ResponseVar`.

Data Types: `string` | `cell`

`ResponseVar` — Response variable
last column of `data` (default) | string | character vector

Response variable, specified as `ResponseVar` and a string or character vector. The response variable contains the LGD data and must be a numeric variable. An LGD value of `0` indicates no loss (full recovery), `1` indicates total loss (no recovery), and values between `0` and `1` indicate a partial loss. By default, `ResponseVar` is set to the last column.

Data Types: `string` | `char`

`BoundaryTolerance` — Value to perturb LGD responses
`1e-5` (default) | positive numeric

Value to perturb LGD response values away from 0 to 1, specified as `BoundaryTolerance` and a positive scalar numeric.

Data Types: `double`

`SolverOptions` — `optimoptions` object
object

Options for fitting, specified as `SolverOptions` and an `optimoptions` object that is created using `optimoptions` from Optimization Toolbox. The defaults for the `optimoptions` object are:

- "Display" — "none"
- "Algorithm" — "quasi-newton"
- "MaxFunctionEvaluations" — 500 × Number of model coefficients
- "MaxIterations" — 1000

Note When using optimoptions with a Beta model, specify the SolverName as fminunc.

The number of Beta model coefficients is determined at run time, depending on the number of predictors and the number of categories in the categorical predictors.

Data Types: object

Properties

ModelID — User-defined model ID
Beta (default) | string

User-defined model ID, returned as a string.

Data Types: string

Description — User-defined description
"" (default) | string

User-defined description, returned as a string.

Data Types: string

UnderlyingModel — Underlying statistical model
compact linear model

This property is read-only.

Underlying statistical model, returned as a compact linear model object. The compact version of the underlying regression model is an instance of the risk.internal.credit.BetaModel class.

Data Types: object

PredictorVars — Predictor variables
all columns of data except for the ResponseVar (default) | string array

Predictor variables, returned as a string array.

Data Types: string

ResponseVar — Response variable
last column of data (default) | string

Response variable, returned as a string.

Data Types: string

BoundaryTolerance — Value to perturb LGD responses
1e-5 (default) | positive numeric

Value to perturb LGD response values away from 0 to 1, returned as a positive scalar numeric.

Data Types: double

Object Functions

predict	Predict loss given default
modelDiscrimination	Compute AUROC and ROC data
modelDiscriminationPlot	Plot ROC curve
modelCalibration	Compute R-square, RMSE, correlation, and sample mean error of predicted and observed LGDs
modelCalibrationPlot	Scatter plot of predicted and observed LGDs

Examples

Create Beta LGD Model

This example shows how to use `fitLGDModel` to create a `Beta` model object for loss given default (LGD).

Load LGD Data

Load the LGD data.

```
load LGDData.mat
head(data)
```

```
      LTV         Age          Type           LGD    
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

```
rng('default');
NumObs = height(data);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Beta LGD Model

Use `fitLGDModel` to create a `Beta` model object using the `TrainingInd` data.

```
BetaLGDModel = fitLGDModel(data(TrainingInd,:),'Beta',...
   ModelID='Example LGD Beta',...
   PredictorVars={'LTV' 'Age' 'Type'},...
   ResponseVar='LGD',...
   BoundaryTolerance=2e-05);
disp(BetaLGDModel)
```

```
  Beta with properties:

    BoundaryTolerance: 2.0000e-05
              ModelID: "Example LGD Beta"
          Description: ""
      UnderlyingModel: [1x1 risk.internal.credit.BetaModel]
        PredictorVars: ["LTV"    "Age"    "Type"]
          ResponseVar: "LGD"
```

Display the underlying model.

```
disp(BetaLGDModel.UnderlyingModel)
```

```
Beta regression model:
     logit(LGD) ~ 1_mu + LTV_mu + Age_mu + Type_mu
     log(LGD) ~ 1_phi + LTV_phi + Age_phi + Type_phi

Estimated coefficients:
                           Estimate       SE        tStat        pValue  
                           ________    ________    ________    __________

    (Intercept)_mu          -1.3884     0.13116     -10.585             0
    LTV_mu                  0.61404     0.14988      4.0967    4.3501e-05
    Age_mu                 -0.47446    0.039808     -11.919             0
    Type_investment_mu      0.45159    0.084719      5.3305    1.0847e-07
    (Intercept)_phi          -0.113     0.12578    -0.89843       0.36906
    LTV_phi                0.030906     0.14692     0.21036       0.83341
    Age_phi                 0.23972    0.040152      5.9703    2.7749e-09
    Type_investment_phi    -0.14724    0.078187     -1.8831      0.059823

Number of observations: 2093
Log-likelihood: -4976.51
```

Predict LGD

For `Beta` models, use `predict` to calculate the predicted LGD value, which is the unconditional expected value of the response, given the predictor values.

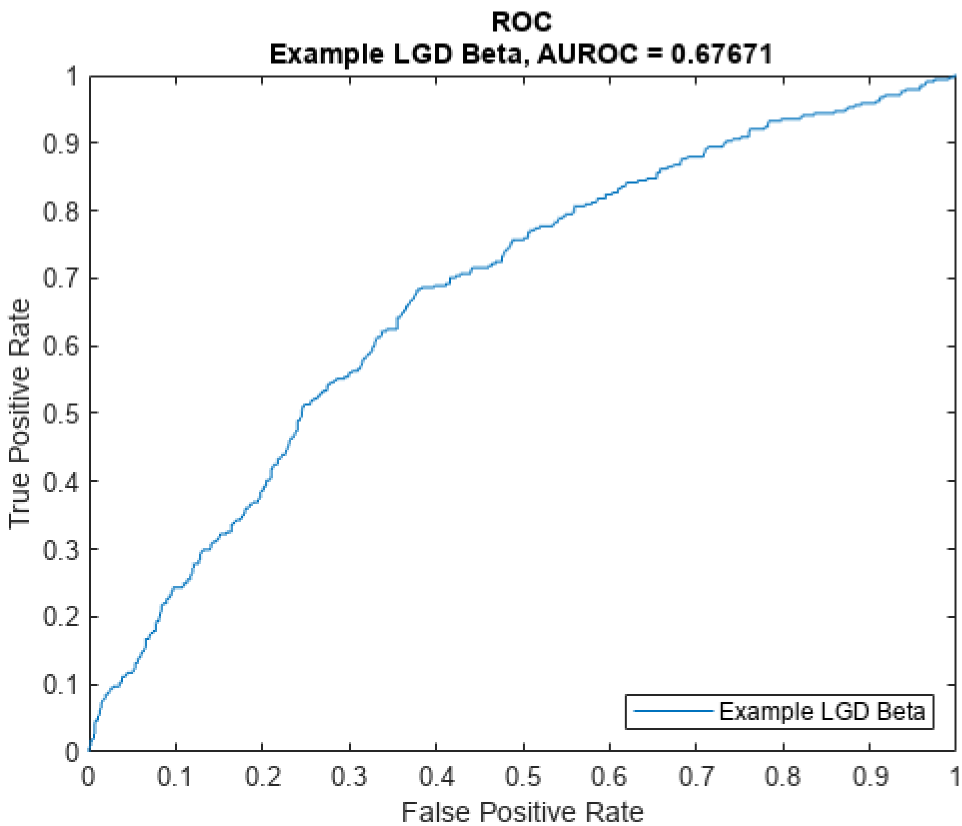
```
predictedLGD = predict(BetaLGDModel,data(TestInd,:))
```

```
predictedLGD = 1394×1

    0.0935
    0.1483
    0.3520
    0.0963
    0.1879
    0.2587
    0.2671
    0.0212
    0.1768
    0.2249
      ⋮
```

Validate LGD Model

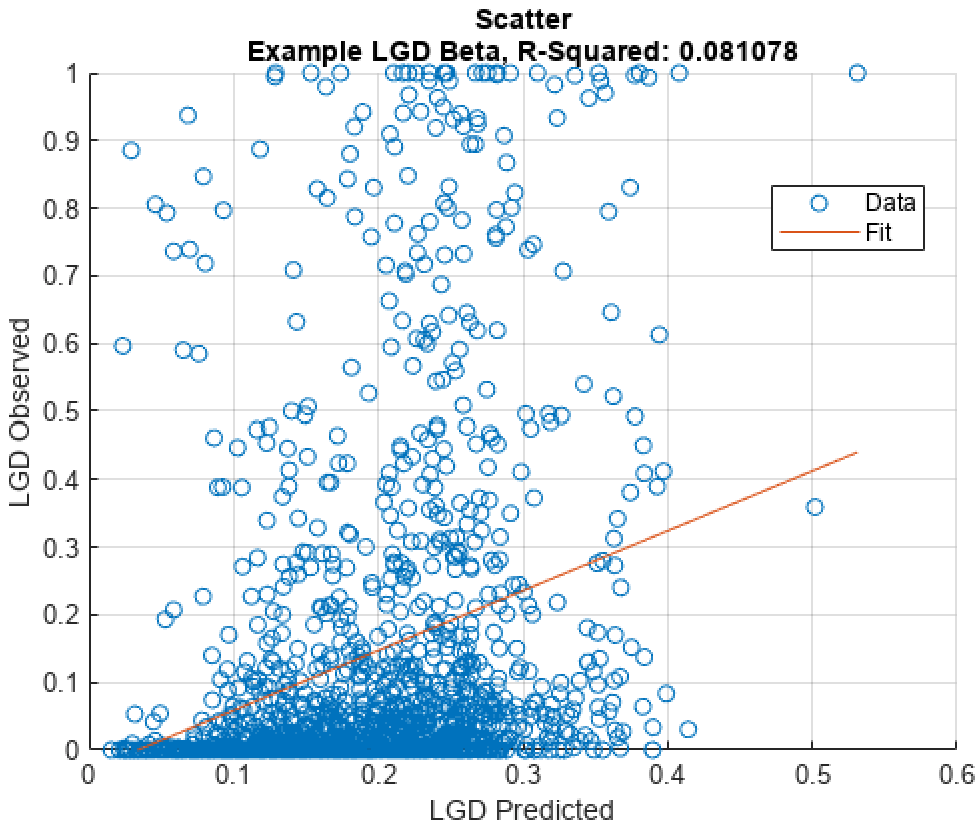
Use `modelDiscriminationPlot` to plot the ROC curve.

```
modelDiscriminationPlot(BetaLGDModel,data(TestInd,:))
```



Use `modelCalibrationPlot` to show a scatter plot of the predictions.

```
modelCalibrationPlot(BetaLGDModel,data(TestInd,:))
```



More About

Beta Regression Models

Beta regression models model continuous variables that assume values in the open standard unit interval (0, 1).

The beta regression model is based on the assumption that the dependent variable is beta-distributed and that its mean is related to a set of regressors through a linear predictor with unknown coefficients and a link function. The beta regression model also includes a precision parameter that may be constant or depend on a potentially different set of regressors through a link function as well.

The beta regression model is based on an alternative parameterization of the beta density in terms of the mean and a precision parameter: $\mu = a / (a + b)$ and $\varphi = a + b$, $0 < \mu < 1$, $\varphi > 0$.

$$f(y;\mu,\phi) = \frac{\Gamma(\phi)}{\Gamma(\mu\phi)\Gamma((1-\mu)\phi)} y^{\mu\phi - 1}(^{1}, \quad 0<y<1,$$

$$y \sim \mathrm{B}(\mu,\varphi) \text{ with } \mathrm{E}(y)=\mu, \ \mathrm{VAR}(\varphi)=\mu(1\text{-}\mu)/(1+\varphi)$$

The beta regression model is defined as

$$g_1(\mu_i) = \eta_{1i} = x_i^T\beta,$$

$$g_2(\phi_i) = \eta_{2i} = z_i^T\gamma,$$

where $\beta = (\beta_0,\beta_1,\ldots,\beta_k)^T$, $\gamma = (\gamma_0,\gamma_1,\ldots,\gamma_k)^T$ are the regression coefficients, η_{1i}, η_{2i} are the linear predictors, and x_i, z_i are the independent variables. g_1 and g_2 are the link functions where g_1 is logit and g_2 is log.

Parameter estimation is performed by maximum likelihood (ML).

Version History

Introduced in R2022b

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

Functions

`fitLGDModel` | `Regression` | `Tobit`

Topics

"Model Loss Given Default" on page 4-90
"Basic Loss Given Default Model Validation" on page 4-131
"Compare Tobit LGD Model to Benchmark Model" on page 4-133
"Compare Loss Given Default Models Using Cross-Validation" on page 4-140
"Overview of Loss Given Default Models" on page 1-31

Tobit

Create `Tobit` model object for loss given default

Description

Create and analyze a `Tobit` model object to calculate loss given default (LGD) using this workflow:

1. Use `fitLGDModel` to create a `Tobit` model object.
2. Use `predict` to predict the LGD.
3. Use `modelDiscrimination` to return AUROC and ROC data. You can plot the results using `modelDiscriminationPlot`.
4. Use `modelCalibration` to return the R-squared, RMSE, correlation, and sample mean error of predicted and observed LGD data. You can plot the results using `modelCalibrationPlot`.

Creation

Syntax

```
TobitLGDModel = fitLGDModel(data,ModelType)
TobitLGDModel = fitLGDModel(___,Name,Value)
```

Description

`TobitLGDModel = fitLGDModel(data,ModelType)` creates a `Tobit` LGD model object.

`TobitLGDModel = fitLGDModel(___,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the previous syntax. The optional name-value pair arguments set the model object properties on page 6-689. For example, `lgdModel = fitLGDModel(data,'tobit','PredictorVars',{'LTV' 'Age' 'Type'},'ResponseVar','LGD','CensoringSide','left','LeftLimit',1e-4)` creates a `lgdModel` object using a `Tobit` model type.

Input Arguments

`data` — Data for loss given default
table

Data for loss given default, specified as a table.

Data Types: `table`

`ModelType` — Model type
string with value `"Tobit"` | character vector with value `'Tobit'`

Model type, specified as a string with the value of `"Tobit"` or a character vector with the value of `'Tobit'`.

Data Types: `char` | `string`

Tobit Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: lgdModel = fitLGDModel(data,'tobit','PredictorVars',{'LTV' 'Age' 'Type'},'ResponseVar','LGD','CensoringSide','left','LeftLimit',1e-4)

ModelID — User-defined model ID
"Tobit" (default) | string | character vector

User-defined model ID, specified as the comma-separated pair consisting of 'ModelID' and a string or character vector. The software uses the ModelID text to format outputs and is expected to be short.

Data Types: string | char

Description — User-defined description for model
"" (default) | string | character vector

User-defined description for model, specified as the comma-separated pair consisting of 'Description' and a string or character vector.

Data Types: string | char

PredictorVars — Predictor variables
all columns of data except for ResponseVar (default) | string array | cell array of character vectors

Predictor variables, specified as the comma-separated pair consisting of 'PredictorVars' and a string array or cell array of character vectors. PredictorVars indicates which columns in the data input contain the predictor information. By default, PredictorVars is set to all the columns in the data input except for ResponseVar.

Data Types: string | cell

ResponseVar — Response variable
last column of data (default) | string | character vector

Response variable, specified as the comma-separated pair consisting of 'ResponseVar' and a string or character vector. The response variable contains the LGD data and must be a numeric variable. An LGD value of 0 indicates no loss (full recovery), 1 indicates total loss (no recovery), and values between 0 and 1 indicate a partial loss. By default, ResponseVar is set to the last column.

Data Types: string | char

CensoringSide — Censoring side
"both" (default) | character vector with value of 'left', 'right', or 'both' | string with value of "left", "right", or "both"

Censoring side, specified as the comma-separated pair consisting of 'CensoringSide' and a character vector or string. CensoringSide indicates whether the desired Tobit model is left-censored, right-censored, or censored on both sides.

Data Types: string | char

LeftLimit — Left-censoring limit
0 (default) | numeric between 0 and 1

Left-censoring limit, specified as the comma-separated pair consisting of 'LeftLimit' and a scalar numeric between 0 and 1.

Data Types: double

RightLimit — Right-censoring limit
1 (default) | numeric between 0 and 1

Right-censoring limit, specified as the comma-separated pair consisting of 'RightLimit' and a scalar numeric between 0 and 1.

Data Types: double

SolverOptions — optimoptions object
object

Options for fitting, specified as the comma-separated pair consisting of 'SolverOptions' and an optimoptions object that is created using optimoptions from Optimization Toolbox. The defaults for the optimoptions object are:

- "Display" — "none"
- "Algorithm" — "sqp"
- "MaxFunctionEvaluations" — 500 × Number of model coefficients
- "MaxIterations" — The number of Tobit model coefficients is determined at run time, it depends on the number of predictors and the number of categories in the categorical predictors.

Note When using optimoptions with a Tobit model, specify the SolverName as fmincon.

Data Types: object

Properties

ModelID — User-defined model ID
Tobit (default) | string

User-defined model ID, returned as a string.

Data Types: string

Description — User-defined description
"" (default) | string

User-defined description, returned as a string.

Data Types: string

UnderlyingModel — Underlying statistical model
compact linear model

This property is read-only.

Underlying statistical model, returned as a compact linear model object. The compact version of the underlying regression model is an instance of the `classreg.regr.CompactLinearModel` class. For more information, see `fitlm` and `CompactLinearModel`.

Data Types: `CompactLinearModel`

`PredictorVars` — Predictor variables
all columns of `data` except for the `ResponseVar` (default) | string array

Predictor variables, returned as a string array.

Data Types: `string`

`ResponseVar` — Response variable
last column of `data` (default) | string

Response variable, returned as a string.

Data Types: `string`

`CensoringSide` — Censoring side
`"both"` (default) | string with value of `"left"`, `"right"`, or `"both"`

This property is read-only.

Censoring side, returned as a string.

Data Types: `string`

`LeftLimit` — Left-censoring limit
`0` (default) | numeric between `0` and `1`

This property is read-only.

Left-censoring limit, returned as a scalar numeric between `0` and `1`.

Data Types: `double`

`RightLimit` — Right-censoring limit
`1` (default) | numeric between `0` and `1`

This property is read-only.

Right-censoring limit, returned as a scalar numeric between `0` and `1`.

Data Types: `double`

Object Functions

predict	Predict loss given default
modelDiscrimination	Compute AUROC and ROC data
modelDiscriminationPlot	Plot ROC curve
modelCalibration	Compute R-square, RMSE, correlation, and sample mean error of predicted and observed LGDs
modelCalibrationPlot	Scatter plot of predicted and observed LGDs

Examples

Create Tobit LGD Model

This example shows how to use `fitLGDModel` to create a `Tobit` model for loss given default (LGD).

Load LGD Data

Load the LGD data.

```
load LGDData.mat
head(data)
```

```
      LTV        Age          Type           LGD
    _______    _______    ___________    _________

    0.89101    0.39716    residential     0.032659
    0.70176     2.0939    residential      0.43564
    0.72078     2.7948    residential    0.0064766
    0.37013      1.237    residential     0.007947
    0.36492     2.5818    residential            0
      0.796     1.5957    residential      0.14572
    0.60203     1.1599    residential     0.025688
    0.92005    0.50253    investment      0.063182
```

```
rng('default');
NumObs = height(data);
c = cvpartition(NumObs,'HoldOut',0.4);
TrainingInd = training(c);
TestInd = test(c);
```

Create Tobit LGD Model

Use `fitLGDModel` to create a `Tobit` model using the `TrainingInd` data.

```
lgdModel = fitLGDModel(data(TrainingInd,:),'Tobit',...
   'ModelID','Example Tobit',...
   'PredictorVars',{'LTV' 'Age' 'Type'},...
   'ResponseVar','LGD',...
   'CensoringSide','left',...
   'LeftLimit',1e-4);
disp(lgdModel)
```

```
  Tobit with properties:

      CensoringSide: "left"
          LeftLimit: 1.0000e-04
         RightLimit: 1
            ModelID: "Example Tobit"
        Description: ""
    UnderlyingModel: [1x1 risk.internal.credit.TobitModel]
      PredictorVars: ["LTV"    "Age"    "Type"]
        ResponseVar: "LGD"
```

Display the underlying model. The underlying model is a left-censored Tobit model. Use the `'CensoringSide'` argument and the `'LeftLimit'` and `'RightLimit'` arguments to modify the underlying Tobit model.

```
disp(lgdModel.UnderlyingModel)
```

```
Tobit regression model, left-censored:
     LGD = max(0.0001,Y*)
```

```
     Y* ~ 1 + LTV + Age + Type

Estimated coefficients:
                       Estimate       SE         tStat       pValue  
                       ________    _________    _______    __________

    (Intercept)        0.057356     0.026619     2.1547      0.031297
    LTV                  0.2003     0.030641     6.5369    7.8659e-11
    Age                -0.09405    0.0072991    -12.885             0
    Type_investment     0.10071     0.017913     5.6223    2.1363e-08
    (Sigma)             0.28833    0.0055231     52.204             0

Number of observations: 2093
Number of left-censored observations: 547
Number of uncensored observations: 1546
Number of right-censored observations: 0
Log-likelihood: -638.353
```

Predict LGD

For Tobit models, use `predict` to calculate the predicted LGD value, which is the unconditional expected value of the response, given the predictor values.

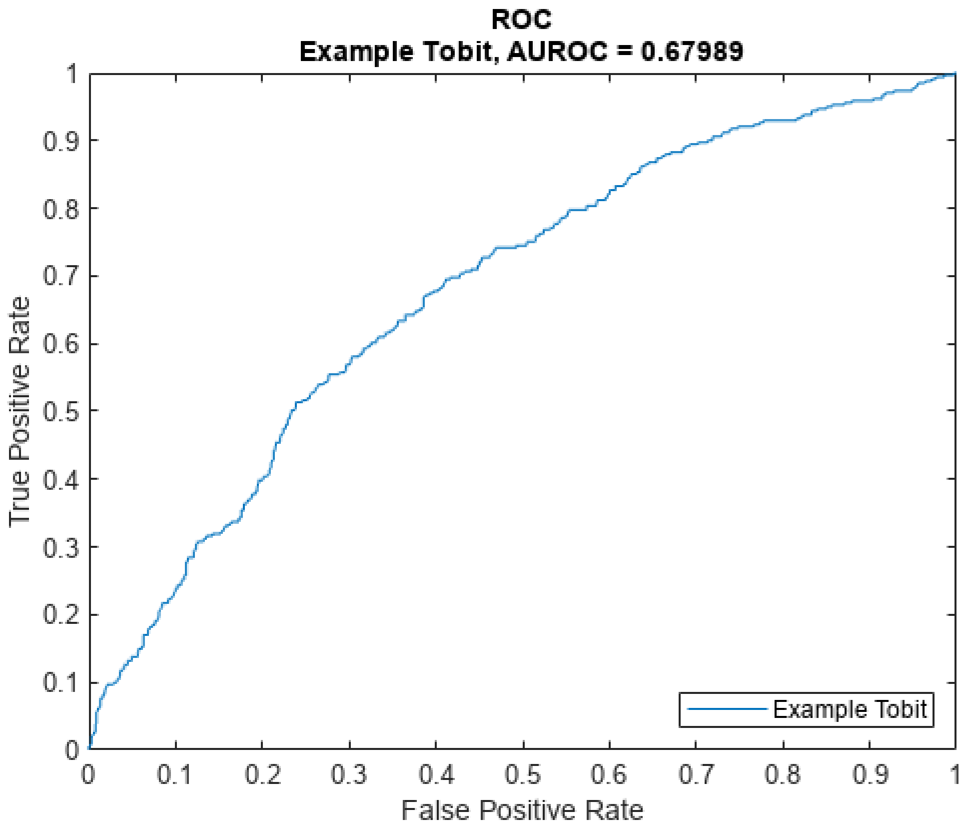
```
predictedLGD = predict(lgdModel,data(TestInd,:))
```

```
predictedLGD = 1394×1

    0.0871
    0.1228
    0.3181
    0.0926
    0.1654
    0.2215
    0.2347
    0.0102
    0.1576
    0.1969
      ⋮
```

Validate LGD Model

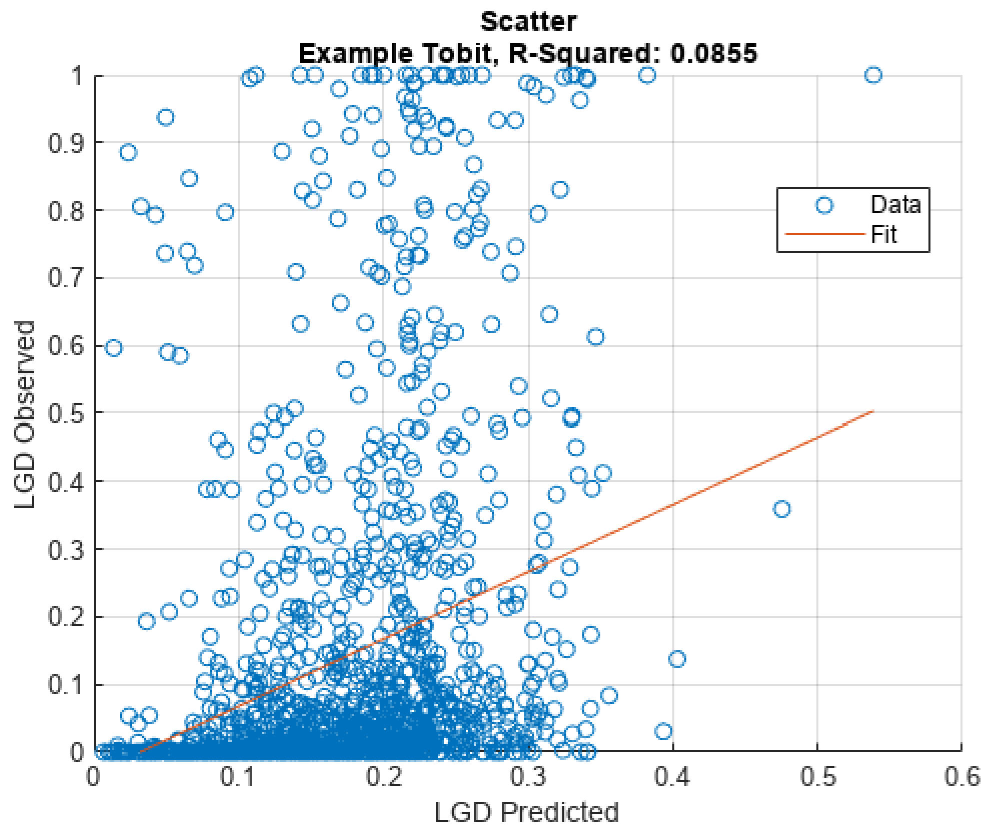
Use `modelDiscriminationPlot` to plot the ROC curve.

```
modelDiscriminationPlot(lgdModel,data(TestInd,:))
```



Use `modelCalibrationPlot` to show a scatter plot of the predictions.

```
modelCalibrationPlot(lgdModel,data(TestInd,:))
```



More About

Loss Given Default Tobit Models

The loss given default (LGD) Tobit models fit a Tobit model to LGD data.

Tobit models are "censored" regression models. Tobit models assume that the response variable can be observed only within certain limits, and no value outside the limits can be observed. In the case of LGD models, the limits are typically 0 (total recovery or cure) and 1 (total loss). A distribution of response values where there is a high frequency of observations at the limits is consistent with the model assumptions. For LGD models, it is common to have distributions with a high proportion of cures, or high proportion of total losses, or both.

The Tobit model combines the following two formulas:

$$Y = \min\{\max\{L, Y^*\}, R\}$$
$$Y^* = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \sigma\varepsilon = X\beta + \sigma\varepsilon$$

where

- Y is the observed response variable, the observed LGD data for an LGD model.
- L is the left limit, the lower bound for the response values, typically `0` for LGD models.
- R is the right limit, the upper bound for the response values, typically `1` for LGD models.

- Y^* is a latent, unobserved variable.
- β_j is the coefficient of the *j*th predictor (or the intercept for j = `0`).
- σ is the standard deviation of the error term.
- ε is the error term, assumed to follow a standard normal distribution.

The first formula above is written using `min` and `max` operators and is equivalent to

$$Y = \begin{cases} L & \text{if } Y^* \leq L \\ Y^* & \text{if } L < Y^* < R \\ R & \text{if } Y^* \geq R \end{cases}$$

The standard deviation of the error is explicitly indicated in the formulas. Unlike traditional regression least-squares estimation, where the standard deviation of the error can be inferred from the residuals, for Tobit models the estimation is via maximum likelihood and the standard deviation needs to be handled explicitly during the estimation. If there are *p* predictor variables, the Tobit model estimates *p*+2 coefficients, namely, one coefficient for each predictor, plus an intercept, plus a standard deviation.

Three censoring side options are supported in the Tobit LGD models with the `CensoringSide` name-value argument:

- `'both'` — This is the default option, with censoring on both sides. The estimation uses left and right limits.
- `'left'` — The left-censored version of the model has no right limit (or $R = \infty$). The relationship between Y and Y^* is $Y = \mathtt{max}\{L,Y^*\}$.
- `'right'` — The right-censored version of the model has no left limit (or $L = -\infty$). The relationship between Y and Y^* is $Y = \mathtt{min}\{Y^*,R\}$.

The parameters of the Tobit model are estimated using maximum likelihood. For observation $i = 1, \ldots, n$, the likelihood function is

$$LF(\beta, \sigma \mid X_i, Y_i) = \begin{cases} \Phi(L; X_i\beta, \sigma) & \text{if } Y_i \leq L \\ \phi(Y_i; X_i\beta, \sigma) & \text{if } L < Y_i < R \\ 1 - \Phi(R; X_i\beta, \sigma) & \text{if } Y_i \geq R \end{cases}$$

where

- $\Phi(x;m,s)$ is the cumulative normal distribution with mean m and standard deviation s.
- $\phi(x;m,s)$ is the normal density function with mean m and standard deviation s.

This likelihood function is for models censored on both sides. For left-censored models, the right limit has no effect, and the likelihood function has two cases only ($R = \infty$); likewise for right-censored models ($L = -\infty$).

The log-likelihood function is the sum of the logarithm of the likelihood functions for individual observations

$$LLF(\beta, \sigma \mid X, Y) = \sum_{i=1}^{n} \log(LF(\beta, \sigma \mid X_i, Y_i))$$

The parameters are estimated by maximizing the log-likelihood function. The only constraint is that the σ parameter must be positive.

To predict an LGD value, Tobit LGD models return the unconditional expected value of the response, given the predictor values

$$LGD_i^{pred} = E[Y_i | X_i]$$

The expression for the expected value can be separated into the cases

$$\begin{aligned} &E[Y] = E[Y|Y = L]P(Y = L) \\ &+E[Y|L < Y < R]P(L < Y < R) \\ &+E[Y|Y = R]P(Y = R) \end{aligned}$$

Using the previous expression and the properties of the (truncated) normal distribution, it follows that

$$E[Y_i | X_i] = \Phi(a_i)L + (\Phi(b_i) - \Phi(a_i))(X_i\beta + \sigma\lambda_i) + (1 - \Phi(b_i))R$$

where

$$a_i = \frac{L - X_i\beta}{\sigma}, b_i = \frac{R - X_i\beta}{\sigma}, \text{ and } \lambda_i = \frac{\phi(a_i) - \phi(b_i)}{\Phi(b_i) - \Phi(a_i)}$$

This expression applies to the models censored on both sides. For models censored on one side only, the corresponding expressions can be derived from here. For example, for left-censored models, let the R limit in the expression above go to infinity, and the resulting expression is

$$E[Y_i | X_i] = \Phi(a_i)L + (1 - \Phi(a_i))\left(X_i\beta + \sigma \frac{\phi(a_i)}{1 - \Phi(a_i)}\right)$$

Similarly, for right-censored models, the L limit is decreased to minus infinity to get

$$E[Y_i | X_i] = \Phi(b_i)\left(X_i\beta - \sigma \frac{\phi(b_i)}{\Phi(b_i)}\right) + (1 - \Phi(b_i))R$$

Version History

Introduced in R2021a

R2023a: `modelAccuracy` object function is renamed to `modelCalibration` function

Not recommended starting in R2023a

The `modelAccuracy` object function is renamed to `modelCalibration` function. The use of `modelAccuracy` is discouraged, use `modelCalibration` instead.

R2023a: `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function

Not recommended starting in R2023a

The `modelAccuracyPlot` object function is renamed to `modelCalibrationPlot` function. The use of `modelAccuracyPlot` is discouraged, use `modelCalibrationPlot` instead.

References

[1] Baesens, Bart, Daniel Roesch, and Harald Scheule. *Credit Risk Analytics: Measurement Techniques, Applications, and Examples in SAS.* Wiley, 2016.

[2] Bellini, Tiziano. *IFRS 9 and CECL Credit Risk Modelling and Validation: A Practical Guide with Examples Worked in R and SAS.* San Diego, CA: Elsevier, 2019.

See Also

Functions

`fitLGDModel` | `Regression` | `Beta`

Topics

"Model Loss Given Default" on page 4-90
"Basic Loss Given Default Model Validation" on page 4-131
"Compare Tobit LGD Model to Benchmark Model" on page 4-133
"Compare Loss Given Default Models Using Cross-Validation" on page 4-140
"Overview of Loss Given Default Models" on page 1-31

varbacktest

Create `varbacktest` object to run suite of value-at-risk (VaR) backtests

Description

The general workflow is:

1. Load or generate the data for the VaR backtesting analysis.
2. Create a `varbacktest` object. For more information, see “Create varbacktest” on page 6-698.
3. Use the `summary` function to generate a summary report for the given data on the number of observations and the number of failures.
4. Use the `runtests` function to run all tests at once.
5. For additional test details, run the following individual tests:

   - `tl` — Traffic light test
   - `bin` — Binomial test
   - `pof` — Proportion of failures
   - `tuff` — Time until first failure
   - `cc` — Conditional coverage mixed
   - `cci` — Conditional coverage independence
   - `tbf` — Time between failures mixed
   - `tbfi` — Time between failures independence

   For more information, see “VaR Backtesting Workflow” on page 2-6.

Creation

Syntax

```
vbt = varbacktest(PortfolioData,VaRData)
vbt = varbacktest( ___ ,Name,Value)
```

Description

`vbt = varbacktest(PortfolioData,VaRData)` creates a `varbacktest` (`vbt`) object using portfolio outcomes data and corresponding value-at-risk (VaR) data. The `vbt` object has the following properties:

- PortfolioData on page 6-0 — `NumRows`-by-`1` numeric array containing a copy of the `PortfolioData`
- VaRData on page 6-0 — `NumRows`-by-`NumVaRs` numeric array containing a copy of the `VaRData`
- PortfolioID on page 6-0 — String containing the `PortfolioID`

- VaRID on page 6-0 — 1-by-NumVaRs string vector containing the VaRIDs for the corresponding columns in VaRData
- VaRLevel on page 6-0 — 1-by-NumVaRs numeric array containing the VaRLevels for the corresponding columns in VaRData.

Note

- The required input arguments for PortfolioData and VaRData must all be in the same units. These arguments can be expressed as returns or as profits and losses. There are no validations in the varbacktest object regarding the units of these arguments.
- If there are missing values (NaNs) in the data for PortfolioData or VaRData, the row of data is discarded before applying the tests. Therefore, a different number of observations are reported for models with different number of missing values. The reported number of observations equals the original number of rows minus the number of missing values. To determine if there are discarded rows, use the 'Missing' column of the summary report.

---

vbt = varbacktest(___ ,Name,Value) sets Properties on page 6-701 using name-value pairs and any of the arguments in the previous syntax. For example, vbt = varbacktest(PortfolioData,VaRData,'PortfolioID','Equity100','VaRID','TotalVaR','VaRLevel',.99). You can specify multiple name-value pairs as optional name-value pair arguments.

Input Arguments

PortfolioData — Portfolio outcomes data

NumRows-by-1 numeric array | NumRows-by-1 numeric columns table | NumRows-by-1 numeric columns timetable

Portfolio outcomes data, specified as a NumRows-by-1 numeric array, NumRows-by-1 table, or a NumRows-by-1 timetable with a numeric column containing portfolio outcomes data. PortfolioData input sets the PortfolioData on page 6-0 property.

Note The required input arguments for PortfolioData and VaRData must all be in the same units. These arguments can be expressed as returns or as profits and losses. There are no validations in the varbacktest object regarding the units of these arguments.

Data Types: double | table | timetable

VaRData — Value-at-risk (VaR) data

NumRows-by-NumVaRs numeric array | NumRows-by-NumVaRs table with numeric columns | NumRows-by-NumVaRs timetable with numeric columns

Value-at-risk (VaR) data, specified using a NumRows-by-NumVaRs numeric array, NumRows-by-NumVaRs table, or a NumRows-by-NumVaRs timetable with numeric columns. VaRData input sets the VaRData on page 6-0 property.

If VaRData has more than one column (NumVaRs> 1), the PortfolioData is tested against each column in VaRData. By default, a 0.95 VaR confidence level is used for all columns in VaRData. (Use VaRLevel to specify different VaR confidence levels.)

The convention is that VaR is a positive amount. Therefore, a failure is recorded when the loss (the negative of the portfolio data) exceeds the VaR, that is, when

```
-PortfolioData > VaRData
```

For example, a VaR of 1,000,000 (positive) is violated whenever there is an outcome worse than a 1,000,000 loss (the negative of the portfolio outcome, or loss, is larger than the VaR).

Negative `VaRData` values are allowed, however negative VaR values indicate a highly profitable portfolio that cannot lose money at the given VaR confidence level. That is, the worst-case scenario at the given confidence level is still a profit.

Note The required input arguments for `PortfolioData` and `VaRData` must all be in the same units. These arguments can be expressed as returns or as profits and losses. There are no validations in the `varbacktest` object regarding the units of these arguments.

Data Types: `double` | `table` | `timetable`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `vbt = varbacktest(PortfolioData,VaRData,'PortfolioID','Equity100','VaRID','TotalVaR','VaRLevel',.99)`

`PortfolioID` — User-defined ID for `PortfolioData` input
character vector | string

User-defined ID for `PortfolioData` input, specified as the comma-separated pair consisting of `'PortfolioID'` and a character vector or string. The `PortfolioID` name-value pair argument sets the PortfolioID on page 6-0 property.

If `PortfolioData` is a numeric array, the default value for `PortfolioID` is `'Portfolio'`. If `PortfolioData` is a table, `PortfolioID` is set by default to the corresponding variable name in the table.

Data Types: `char` | `string`

`VaRID` — VaR identifier for `VaRData` columns
character vector | cell array of character vectors | string | string array

VaR identifier for `VaRData` columns, specified as the comma-separated pair consisting of `'VaRID'` and a character vector or string. Multiple `VaRID`s are specified using a `1`-by-`NumVaRs` (or `NumVaRs`-by-`1`) cell array of character vectors or string vector with user-defined IDs for the `VaRData` columns. The `VaRID` name-value pair argument sets the VaRID on page 6-0 property.

If `NumVaRs = 1`, the default value for `VaRID` is `'VaR'`. If `NumVaRs > 1`, the default value is `'VaR1'`, `'VaR2'`, and so on. If `VaRData` is a table, `'VaRID'` is set by default to the corresponding variable names in the table.

Data Types: char | cell | string

VaRLevel — VaR confidence level
0.95 (default) | numeric with values between 0 and 1 | numeric array with values between 0 and 1

VaR confidence level, specified as the comma-separated pair consisting of 'VaRLevel' and a numeric between 0 and 1 or a 1-by-NumVaRs numeric array with values between 0 and 1 for the corresponding columns in VaRData. The VaRLevel name-value pair argument sets the VaRLevel on page 6-0 property.

Data Types: double

Properties

PortfolioData — Portfolio data for VaR backtesting analysis
numeric array

Portfolio data for the VaR backtesting analysis, specified as a NumRows-by-1 numeric array containing a copy of the portfolio data.

Data Types: double

VaRData — VaR data for VaR backtesting analysis
numeric array

VaR data for the VaR backtesting analysis, specified as a NumRows-by-NumVaRs numeric array containing a copy of the VaR data.

Data Types: double

PortfolioID — Portfolio identifier
string

Portfolio identifier, specified as a string.

Data Types: string

VaRID — VaR identifier
string array

VaR identifier, specified as a 1-by-NumVaRs string array containing the VaR IDs for the corresponding columns in VaRData.

Data Types: string

VaRLevel — VaR level
numeric array with values between 0 and 1

VaR level, specified as a 1-by-NumVaRs numeric array containing the VaR levels for the corresponding columns in VaRData.

Data Types: double

varbacktest Property	Set or Modify Property from Command Line Using varbacktest	Modify Property Using Dot Notation
PortfolioData	Yes	No
VaRData	Yes	No
PortfolioID	Yes	Yes
VaRID	Yes	Yes
VaRLevel	Yes	Yes

Object Functions

tl	Traffic light test for value-at-risk (VaR) backtesting
bin	Binomial test for value-at-risk (VaR) backtesting
pof	Proportion of failures test for value-at-risk (VaR) backtesting
tuff	Time until first failure test for value-at-risk (VaR) backtesting
cc	Conditional coverage mixed test for value-at-risk (VaR) backtesting
cci	Conditional coverage independence test for value-at-risk (VaR) backtesting
tbf	Time between failures mixed test for value-at-risk (VaR) backtesting
tbfi	Time between failures independence test for value-at-risk (VaR) backtesting
summary	Report on varbacktest data
runtests	Run all tests in varbacktest

Examples

Create varbacktest Object and Run VaR Backtests for Single VaR at 95%

varbacktest takes in portfolio outcomes data and corresponding value-at-risk (VaR) data and returns a varbacktest object.

Create a varbacktest object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

vbt, the varbacktest object, contains a copy of the given portfolio data (PortfolioData property), the given VaR data (VaRData property) and all combinations of portfolio ID, VaR ID, and VaR level to be tested (PortfolioID, VaRID, and VaRLevel properties).

Run the tests using the vbt object.

```
runtests(vbt)
```

```
ans=1×11 table
    PortfolioID    VaRID    VaRLevel     TL       Bin       POF       TUFF       CC       CCI
    ___________    _____    ________    _____    ______    ______    ______    ______    ______

    "Portfolio"    "VaR"      0.95      green    accept    accept    accept    accept    accept
```

Change the `PortfolfioID` and `VaRID` properties using dot notation.

```
vbt.PortfolioID = 'S&P'
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "S&P"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

```
vbt.VaRID = 'Normal at 95%'
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "S&P"
            VaRID: "Normal at 95%"
         VaRLevel: 0.9500
```

Run all tests using the updated `varbacktest` object.

```
runtests(vbt)
```

```
ans=1×11 table
    PortfolioID         VaRID         VaRLevel     TL       Bin       POF       TUFF       CC
    ___________    _______________    ________    _____    ______    ______    ______    ______

       "S&P"       "Normal at 95%"      0.95      green    accept    accept    accept    accept
```

Run VaR Backtests for a Single VaR at 95%

Create a `varbacktest` object.

```
load VaRBacktestData
vbt = varbacktest(EquityIndex,Normal95)
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
```

```
      PortfolioID: "Portfolio"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

vbt, the varbacktest object, contains a copy of the given portfolio data (PortfolioData property), the given VaR data (VaRData property) and all combinations of portfolio ID, VaR ID, and VaR level to be tested (PortfolioID, VaRID, and VaRLevel properties).

Run the tests using the varbacktest object.

```
runtests(vbt)
```

```
ans=1×11 table
    PortfolioID    VaRID    VaRLevel     TL       Bin        POF       TUFF        CC        CCI
    ___________    _____    ________    _____    ______    ______    ______    ______    ______

    "Portfolio"    "VaR"      0.95      green    accept    accept    accept    accept    accept
```

Change the PortfolfioID and VaRID properties using dot notation.

```
vbt.PortfolioID = 'S&P'
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "S&P"
            VaRID: "VaR"
         VaRLevel: 0.9500
```

```
vbt.VaRID = 'Normal at 95%'
```

```
vbt = 
  varbacktest with properties:

    PortfolioData: [1043x1 double]
          VaRData: [1043x1 double]
      PortfolioID: "S&P"
            VaRID: "Normal at 95%"
         VaRLevel: 0.9500
```

Run all tests using the updated varbacktest object.

```
runtests(vbt)
```

```
ans=1×11 table
    PortfolioID         VaRID          VaRLevel     TL       Bin        POF       TUFF        CC
    ___________    _______________    ________    _____    ______    ______    ______    ______

       "S&P"       "Normal at 95%"      0.95      green    accept    accept    accept    accept
```

Run VaR Backtests for Multiple VaRs at Different Confidence Levels

Create a `varbacktest` object that has multiple VaR identifiers with different confidence levels.

```
load VaRBacktestData
    vbt = varbacktest(EquityIndex,...
       [Normal95 Normal99 Historical95 Historical99 EWMA95 EWMA99],...
       'PortfolioID','Equity',...
       'VaRID',{'Normal95' 'Normal99' 'Historical95' 'Historical99' 'EWMA95' 'EWMA99'},...
       'VaRLevel',[0.95 0.99 0.95 0.99 0.95 0.99]);
```

Run the summary report for the `varbacktest` object.

```
summary(vbt)
```

ans=*6×10 table*

PortfolioID	VaRID	VaRLevel	ObservedLevel	Observations	Failures	Exp
"Equity"	"Normal95"	0.95	0.94535	1043	57	52
"Equity"	"Normal99"	0.99	0.9837	1043	17	10
"Equity"	"Historical95"	0.95	0.94343	1043	59	52
"Equity"	"Historical99"	0.99	0.98849	1043	12	10
"Equity"	"EWMA95"	0.95	0.94343	1043	59	52
"Equity"	"EWMA99"	0.99	0.97891	1043	22	10

Run all tests using the `varbacktest` object.

```
runtests(vbt)
```

ans=*6×11 table*

PortfolioID	VaRID	VaRLevel	TL	Bin	POF	TUFF	CC
"Equity"	"Normal95"	0.95	green	accept	accept	accept	accept
"Equity"	"Normal99"	0.99	yellow	reject	accept	accept	accept
"Equity"	"Historical95"	0.95	green	accept	accept	accept	accept
"Equity"	"Historical99"	0.99	green	accept	accept	accept	accept
"Equity"	"EWMA95"	0.95	green	accept	accept	accept	accept
"Equity"	"EWMA99"	0.99	yellow	reject	reject	accept	reject

Run the traffic light test (`tl`) using the `varbacktest` object.

```
tl(vbt)
```

ans=*6×9 table*

PortfolioID	VaRID	VaRLevel	TL	Probability	TypeI	Increase
"Equity"	"Normal95"	0.95	green	0.77913	0.26396	0
"Equity"	"Normal99"	0.99	yellow	0.97991	0.03686	0.26582
"Equity"	"Historical95"	0.95	green	0.85155	0.18232	0
"Equity"	"Historical99"	0.99	green	0.74996	0.35269	0
"Equity"	"EWMA95"	0.95	green	0.85155	0.18232	0
"Equity"	"EWMA99"	0.99	yellow	0.99952	0.0011122	0.43511

Run VaR Backtests for Multiple Portfolios and Concatenate Results

Use `varbacktest` with table inputs and name-value pair arguments to create two `varbacktest` objects and run the concatenated summary report. `varbacktest` uses the variable names in the table inputs as `PortfolioID` and `VaRID`.

```
load VaRBacktestData
vbtE = varbacktest(DataTable(:,2),DataTable(:,3:4),'VaRLevel',[0.95 0.99]);
vbtD = varbacktest(DataTable(:,5),DataTable(:,6:7),'VaRLevel',[0.95 0.99]);
[summary(vbtE); summary(vbtD)]
```

```
ans=4×10 table
     PortfolioID            VaRID            VaRLevel    ObservedLevel    Observations    Failures
    _____________    ___________________    ________    _____________    ____________    ________

    "Equity"         "VaREquity95"            0.95          0.94343           1043            59
    "Equity"         "VaREquity99"            0.99          0.97891           1043            22
    "Derivatives"    "VaRDerivatives95"       0.95          0.95014           1043            52
    "Derivatives"    "VaRDerivatives99"       0.99          0.97028           1043            31
```

Run all the tests and concatenate the results.

```
[runtests(vbtE); runtests(vbtD)]
```

```
ans=4×11 table
     PortfolioID            VaRID            VaRLevel      TL        Bin        POF        TUFF
    _____________    ___________________    ________    ______    ______    ______    ______    __

    "Equity"         "VaREquity95"            0.95      green     accept    accept    accept    ac
    "Equity"         "VaREquity99"            0.99      yellow    reject    reject    accept    re
    "Derivatives"    "VaRDerivatives95"       0.95      green     accept    accept    accept    ac
    "Derivatives"    "VaRDerivatives99"       0.99      red       reject    reject    accept    re
```

Run the `pof` test and concatenate the results.

```
 [pof(vbtE); pof(vbtD)]
```

```
ans=4×9 table
     PortfolioID            VaRID            VaRLevel     POF       LRatioPOF     PValuePOF     Obse
    _____________    ___________________    ________    ______    __________    __________    ____

    "Equity"         "VaREquity95"            0.95      accept       0.91023       0.34005
    "Equity"         "VaREquity99"            0.99      reject        9.8298     0.0017171
    "Derivatives"    "VaRDerivatives95"       0.95      accept    0.00045457       0.98299
    "Derivatives"    "VaRDerivatives99"       0.99      reject        26.809    2.2457e-07
```

Version History

Introduced in R2016b

References

[1] Basel Committee on Banking Supervision, *Supervisory Framework for the Use of 'Backtesting' in Conjunction with the Internal Models Approach to Market Risk Capital Requirements.* January, 1996, https://www.bis.org/publ/bcbs22.htm.

[2] Christoffersen, P. "Evaluating Interval Forecasts." *International Economic Review.* Vol. 39, 1998, pp. 841 – 862.

[3] Cogneau, Ph. *“Backtesting Value-at-Risk: How Good is the Model?*" Intelligent Risk, PRMIA, July, 2015.

[4] Haas, M. "*New Methods in Backtesting.*" Financial Engineering, Research Center Caesar, Bonn, 2001.

[5] Jorion, Ph. *Financial Risk Manager Handbook.* 6th Edition. Wiley Finance, 2011.

[6] Kupiec, P. "Techniques for Verifying the Accuracy of Risk Management Models." *Journal of Derivatives.* Vol. 3, 1995, pp. 73 – 84.

[7] McNeil, A., Frey, R., and Embrechts, P. *Quantitative Risk Management.* Princeton University Press, 2005.

[8] Nieppola, O. “Backtesting Value-at-Risk Models.” Helsinki School of Economics, 2009.

See Also

`tl` | `tuff` | `bin` | `pof` | `cc` | `cci` | `tbf` | `tbfi` | `summary` | `runtests` | `table` | `esbacktest` | `esbacktestbysim`

Topics

“VaR Backtesting Workflow” on page 2-6
“Value-at-Risk Estimation and Backtesting” on page 2-10
“Overview of VaR Backtesting” on page 2-2
“Comparison of ES Backtesting Methods” on page 2-26