Version 2025 Q4

XLMiner SDK

User Guide

Table of Contents

Introduction to XLMiner SDK

XLMiner SDK® (formerly known as SDK Data Mining) is a comprehensive data mining **Software Development Kit** included in **Solver SDK Comprehensive**. Data mining, or data science, is a discovery-driven data analysis technology used for identifying patterns and relationships in data sets. With overwhelming amounts of data now available from transaction systems and external data sources, organizations are presented with increasing opportunities to understand and gain insights into their data. Data science is still an emerging field, and is a convergence of Statistics, Machine Learning, and Artificial Intelligence.
Often, there may be more than one approach to a problem. XLMiner SDK provides a high-level API "tool belt" that offers a variety of methods to analyze data. It has extensive coverage of statistical and machine learning techniques for classification, regression, affinity analysis, data exploration, and reduction.

The worlds of commerce, research, and government are huge and varied. No single data analysis pattern can possibly be right for everyone. The XLMiner SDK provides a fast, solid, and well-tested foundation on which organizations can build and execute data analysis tasks to suit their needs precisely. XLMiner SDK focuses on data mining tasks and provides robust implementations of industry-standard data mining algorithms.

XLMiner SDK is most commonly used in one of the following scenarios.

The automation of recurring data analysis tasks
Developers and data scientists can integrate XLMiner SDK capabilities into scripts written in their preferred programming environment to process data and then analyze it automatically, perhaps even scheduling the data analysis tasks to run without human intervention.

Analyzing Large Data Volumes
In the age of Big Data, the demands of data analysis often exceed the resources of any graphical interactive environment. XLMiner SDK is designed to be scalable as well as to take advantage of modern distributed processing techniques.

Integration with other toolsets
Teams already using a toolset, such as R, will find that XLMiner SDK's R API supports native R DataFrames.

Interactive processing without giving up advanced library power
Interactive environments like R and Jupyter provide great ease and convenience, but often at the cost of processing power. The routines of the XLMiner SDK integrate seamlessly with these environments, giving data scientists the best of both worlds. .Net aficionados will be pleased to use XLMiner from the new C# interactive REPL command line environment XLMiner SDK supports the latest version of Visual Studio 2017. (Supports Java 8, Python (3.7, 3.8 & 3.9) and R (any 4.X) versions.)

XLMiner SDK Capabilities

XLMiner SDK supports all facets of the data mining process, including data exploration and transformation, visualization, feature selection, text mining, time series forecasting, affinity analysis, and unsupervised and supervised learning. It is recommended that you test different approaches to your problem and compare the resultant models, to select a well performing model.

- Data may be acquired from any source (file system, Web, data streams, databases, Big Data sources, ODATA, generic ODBC connection or specific DB connections for MS access, MS SQL, and Oracle) using any tool available for your desired programming language. Exception reporting helps you to quickly find and fix problems with your model and numerous built in output and report capabilities help you to evaluate its performance.

- Clean and transform your data with a comprehensive set of data handling utilities including categorizing data and handling missing values. Use Principal Components Analysis to reduce columns, and K-Means Clustering or Hierarchical Clustering to group data by rows. XLMiner SDK currently contains the following transformation utilities.
    - Factorizing
    - Binning
    - Missing values
    - One-hot-encoding
    - Reducing Categories
    - Rescaling
    - PCA
    - Sampling
    - Canonical Variate Analysis (CVA)
- Clusterize your data into a set of cohesive groups
    - K-Means Clustering
    - Hierarchical Clustering
- Automatically transform free-form text into structured data, identify the most frequently occurring terms and extract key concepts with latent semantic indexing. XLMiner's SDK Text Miner supports:
    - TF-IDF vectorization
    - Latent Semantic Analysis
- Apply the most popular exponential smoothing and Box-Jenkins ARIMA methods, with seasonality, to forecast time series, such as sales and inventory, from historical data. The following time series analysis tools are supported in XLMiner SDK:
    - Autocorrelations
    - Partial Autocorrelations
    - ARIMA
    - Exponential, Double Exponential, Moving Average and Holt Winters Smoothing
- Easily partition your data into training, validation, and test datasets, with no limits on dataset size.
- Use feature selection to automatically identify columns or variables with the greatest explanatory power for your desired classification or regression task.
- Use powerful Multiple Linear Regression with variable selection, and other supervised regression algorithms including Ensemble methods (using any regression engine as a weak learner).

    includes four regression algorithms and three different ensemble methods: Bagging, Boosting and Random Trees. Boosting and Bagging ensemble methods can use any prediction method as a base learner. XLMiner SDK provides extensive functionality for evaluating the performance of supervised models, including the goodness of fit metrics and charts (Lift Charts, Gain Charts, Decile Tables, ROC curves).

    - Multiple Linear Regression
    - k-Nearest Neighbors
    - Regression Trees
    - Neural Networks

- Use classical Discriminant Analysis and Logistic Regression, and other supervised classification algorithms, including Ensemble methods (using any classification engine as a weak learner).
- XLMiner SDK includes six classification algorithms. Boosting and Bagging ensemble methods can use any classification method as a base learner. XLMinerSDK provides extensive functionality for evaluating the performance of supervised models, including the goodness of fit metrics and charts (Lift Charts, Gain Charts, Decile Tables, RROC curves).
    - Discriminant Analysis
    - k-Nearest Neighbors
    - Logistic Regression
    - Classification Trees
    - Naïve Bayes
    - Neural Networks
- Use Affinity Analysis to discover association rules and perform market basket analysis.
- PMML model export/import for the following models:
    - Regression Models (Linear and Logistic)
    - Decision Trees (Classification and Regression)
    - Neural Networks (Classification and Regression)
    - K Nearest Neighbors (Classification and Regression)
    - Discriminant Analysis
    - Naïve Bayes
    - Random Trees (Classification and Regression)
    - Ensemble Bagging (Classification and Regression)
    - Ensemble Boosting (Classification and Regression)
    - Time Series
    - Association Rules
    - Transformations
- JSON model export/import for all supported models
- Feature-parity and model interoperability between XLMiner SDK and Analytic Solver Data Mining for Excel
- Import/Export (Load/Save) for all SDK objects (data structures: Vector, DataFrame, DataFrameVector; algorithmic objects: Estimators, Models) to/from JSON as well as custom SDK string format

Note: To solve a decision flow, use Frontline's RASON Data Mining at www.RASON.com.

This user guide currently only includes chapters on Time Series Analysis, Classification, Regression and Association Rules. However, XLMiner SDK installs over 100 examples in five different programming languages (C#, C++, Java, Python, and R) to give you practice with all its features and methods. (Scala users - see the Java SDK Distribution.) If you find yourself "stuck", help and support are only a call (888-831-0333) or email (support@solver.com) away.

Using XLMiner SDK

XLMiner SDK offers comprehensive facilities for deep data analysis for exploration/educational purposes, as well as for high-performance production applications. Usage can range from a single line of code with minimal user intervention (XLMiner SDK finds sensible defaults when possible), to a customized workflow where users can manually control all parts of exploration, fitting, and scoring processes. XLMiner SDK's easy-to-use API allows the combination of all available data mining techniques into a single, all-inclusive application, or pipeline. For example, your pipeline could first draw a sample from Big Data stored across an Apache Spark cluster, then use Feature Selection to determine the best inputs to a supervised algorithm, partition the data, fit a model, and score new data.

For exploration purposes, XLMiner SDK offers advanced algorithms that provide deep insights about your data/models. Using this information, you can fine-tune the parameters and search for optimal choices to prepare the application for deployment into a production application. At the deployment stage, “auxiliary” informative outputs might not be required, so the user can choose to remove this ancillary code for lightning fast performance.

Although XLMiner SDK's API is very high-level, advanced users still have the power to work within the full capabilities of the supported programming languages: C, C++, C#, Java, Python, R or Scala. All XLMiner SDK’s high-level objects, properties and methods are supported for each language. Example code for each of these languages can be found at C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples. (Scala users can refer to the Java SDK distribution.)

A successful data mining pipeline typically involves three distinct steps: Preparing the Data, Instantiating an Estimator, and Fitting a Model.

Preparing the Data

Regardless of the source of your data, it must be brought into an XLMiner **DataFrame** before it can be passed to any of the XLMiner mining engines. Since virtually all modern development environments can interface with the XLMiner SDK, your data may be prepped, cleansed and organized using the environment of your choice. Some commonly encountered data sources and formats include:

- Delimited text file formats such as CSV/TSV
- Excel
- ODATA
- JSON
- Generic ODBC connections
- Specific data base connections for MS Access, MS SQL, Oracle (with specific simplified syntax)

A **DataFrame**, in XLMiner SDK, is a collection of data organized into named columns of equal length and homogeneous type. XLMiner SDK uses DataFrames to deliver input data to an algorithm and to deliver the results of the algorithm back to the user. DataFrames hold heterogeneous data across columns (variables): numeric, categorical, or textual.

Many tools created for data analysis provide some variation of a DataFrame. At first glance a DataFrame might seem to resemble a database table or SQL query result, and indeed it does, but the resemblance is superficial. Unlike classic database tables and SQL query results, which are specifically row-oriented, DataFrames are column-oriented. In XLMiner, a DataFrame column is a vector, and it is these vectors that provide input data for XLMiner functions.

Data gathered in the real world is never ready for analysis “as is”. DataFrames typically require some type of preprocessing, such as partitioning into training, validation, and test subsets. XLMiner SDK developers can choose to manipulate their DataFrames in XLMiner SDK prior to involving data mining procedures, or you can use your host programming system to structure the data accordingly and load the finished product into XLMiner SDK for analysis.

Examples of basic DataFrame tasks are:

- Creating and filling DataFrames
- Selecting a subset of columns/rows
- Appending columns or rows
- Selecting subsets for training and verification models

Instantiate an Estimator

After the data is prepared, the next step is to instantiate an Estimator class, which will be specific to the particular data mining model of interest. An **Estimator** is the underlying algorithm that "fits" a Model using training data stored in DataFrame(s). An Estimator accepts DataFrame(s) as input through its fit method, and returns a Model object. It stores parameters, but not data, required for fitting the model or performing other data manipulations. This Model may be used to score/transform new data, to compute evaluation metrics or to extract other information using the fitted model and new data. Each Estimator has an attached set of Parameters or options.

For example, in the C# code below, the Classification Decision Tree estimator is instantiated, and two options are set: Minimum Number of Records in Leaves is set to 10 and Prior Probability Method is set to Empirical.

Classification Decision Tree is instantiated and two options are set

```
var estimator = new XLMiner.Classification.DecisionTree.Estimator();
estimator.MinNumRecordsInLeaves = 10;
estimator.PriorProbMethod = PriorProbMethodType.EMPIRICAL;
```

While in this example C++ code, a binning estimator is instantiated and 2 options are set: number of bins is set to 11 and Interval is set to IntervalType::Right_Closed (An example of a right closed interval is (1,3]).

Binning estimator is instantiated and 2 options are set

```
{
    using namespace Binning::EqualInterval;
    Estimator estimator;
    estimator.NumBins = 11;
    estimator.Interval = Binning::IntervalType::RIGHT_CLOSED
 }
```

Fit a Model

The XLMiner Model for each data mining class provides an interface through which the results of model-fitting can be explored. Each **Model** contains core methods to accept a DataFrame or multiple DataFrames as input data, perform a scoring or transformation, and return a new DataFrame. For example, a Model fitted by a classification or regression estimator transforms a DataFrame with input features into a DataFrame holding predicted response (i.e., to predict/classify new data). A Model may have optional or required parameters for scoring or transforming the data. Along with that, most Models offer additional functionality to obtain deeper insights about the model's performance on training and new data.

For example, going back to our first C# example code snippet.

Classification Decision Tree is instantiated and two options are set

```
var estimator = new XLMiner.Classification.DecisionTree.Estimator();
estimator.MinNumRecordsInLeaves = 10;
estimator.PriorProbMethod = PriorProbMethodType.EMPIRICAL;
```

To fit a Classification Decision Tree model, you would add the following line of code:

Fitting a Classification Decision Tree model

```
var model = estimator.fit(trainingInputColumns, trainingTargetColumn)
```

From here you could use the fitted model object to obtain the model-specific evaluations, performance evaluations such as confusion matrices and metrics, or score new data.

The primary purpose of any data analysis tool is to enable developers to create custom applications. Once the XLMiner model fitting and scoring tasks are complete, the results, in DataFrame format, can be employed in virtually any task, ranging from custom data visualization to support for automatic fraud detection and response.

XLMiner SDK Pipeline Example

The following example illustrates how to string together various algorithms to create a data mining workflow or pipeline. This example demonstrates how the results of one data mining function can flow seamlessly into the next.

This example code is in C#, but you can find this same example in the installation directory under Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples for each of the remaining supported example code.

Pipeline Example Code

```
public static int CompleteExample()
{
// 1. Read data from a file
      var data = XLMiner.Reader.textFile(PATH +
      "BostonHousingCat.txt");

// 2. [optional] Sample from data
      var sampler = new XLMiner.Sampling.SimpleRandomSampler()
      {
          SampleSize = 400
      };
      var sampledData = sampler.transform(data);

// 3. [optional, but recommended in order to compute model performance]
      Partition sampledData
      var partitioner = new XLMiner.Partitioner();
      partitioner.PartitionRatio["Training"] = 0.6;
      partitioner.PartitionRatio["Validation"] = 0.4;
      var partitions = partitioner.partition(sampledData);

//4.  Extract the input data and the output column (CAT.MEDV) for each
      partition
      string[] targetCol = new string[]{"CAT.MEDV"};
      var trainingInputColumns = partitions["Training"][Util.all,
      Util.except(targetCol)];
      var trainingTargetColumn = partitions["Training"][Util.all,
      targetCol];
      var validationInputColumns = partitions["Validation"][Util.all,
      Util.except(targetCol)];
      var validationTargetColumn = partitions["Validation"][Util.all,
      targetCol];
      int numTrainingRows = partitions["Training"].NumRows;


//5.  Instantiate an Estimator for Decision Tree Classification
      var estimator = new CTree.Estimator()
      {
            MinNumRecordsInLeaves = numTrainingRows / 10;
            PriorProbMethod = PriorProbMethodType.EMPIRICAL;
      };
      Console.WriteLine(estimator);


//6.  Fit a Model
      var model = estimator.fit(trainingInputColumns,
      trainingTargetColumn);
      Console.WriteLine(model);


//7. Score the validation set, then print to console the first 10 rows
//of the new DataFrame created by binding to columns.
      var predictedLabels = model.predict(validationInputColumns);
      Console.WriteLine(Util.head(util.colBind(new DataFrame[]
      {validationTargetColumn, predictedLabels})));


//8.  Construct and print the confusion matrix.
      using (var confusionMatrix =
      Evaluator.confusion(validationTargetColumn, predictedLabels))
      Console.WriteLine(confusionMatrix);


//9.  Compute the F1 Metric
      double f1metric = Metrics.f1(validationTargetColumn,
      predictedLabels, model.SuccessClass);
      Console.WriteLine("F1 Metric: " + f1metric);
```

```
//10. Clean up memory: All custom objects should be freed at this
//stage.
      Util.free(ref data);
      Util.free(ref sampler);
      Util.free(ref sampledData);
      Util.free(ref partitioner);
      Util.free(ref partitions);
      Util.free(ref trainingInputColumns);
      Util.free(ref trainingTargetColumn);
      Util.free(ref validationInputColumns);
      Util.free(ref validationTargetColumn);
      Util.free(ref estimator);
      Util.free(ref model);
      Util.free(ref predictedLabels);
return 0;
}
```

1. First, the data from the file BostonHousingCat.txt file is imported into the SDK. This file can be found in the installation directory under Frontline Systems/Analytic Solver SDK\XLMinerSDK\Examples/Datasets.
2. A sample is taken from the dataset using simple random sampling with size = 400. The first line calls the *SimpleRandomSampler* transformer constructor.

   While an estimator can fit models, which in turn can either transform data (such as Clustering or Principal Components Analysis) or classify or predict new values (such as Classification or Regression algorithms), simple transformers, such as Sampling or Partitioning, do not create a model. They simply take input data and transform it in one step – there is no information to extract from the training data to apply to new data. Simple transformers such as these do not have a fitting interface, but instead have a direct method or action such as “transform” or “sample” or “partition”.

   The last line samples the data and the result of this transformation is assigned to the variable *sampledData*.
3. The data is partitioned into training and validation partitions with 60% of the records assigned to the training partition and 40% assigned to the validation partition.

   The first line of code constructs the Partitioner transformer. The next two lines of code specify the Training and Validation partition percentages and the last line of code partitions *sampledData*. The result of this transformation (which is two partitions) is assigned to the variable *partitions*.
4. In step 4, the input data and output column (CAT. MEDV) are extracted for each partition, training and validation.
5. Now, you are ready to construct an Estimator and fit a model to the data. At this point, any classification algorithm could be called, but this example calls the Decision Tree Classification.

   The first line of code constructs the Classification Decision Tree Estimator, the next two lines of code set the options: the minimum number of records in leaves, which is set to the number of training rows divided by 10, and the prior probability method type.
6. The next line of code calls *estimator.fit* to fit the model with the input variables (*trainingInputColumns*) and the output variable (*trainingTargetColumn*).

7. The next line of code uses the model created by *estimator.fit* to score the validation partition and then prints to console the first 10 rows *(Util.head)* of a new DataFrame created by binding *(Util.colBind)* to columns *validationTargetColumn, predictedLabels*.
8. This next two lines of code construct and print the confusion matrix
9. Step 9 computes the F1 Metric.
10. Finally, memory is freed for all the variables used in the pipeline.

It's time to install XLMiner SDK and get started.

XLMiner SDK Limits

A license for Solver SDK Comprehensive combines the power of the Solver SDK with the power of XLMiner SDK. See this chart to discover the data handling limits of XLMiner SDK. (For information on Solver SDK, see the Solver SDK User Guide and Reference Guide.)

XLMinerSDK-limits.pdf (solver.com)

Note: Users who purchase a license for Solver SDK Platform or Pro will also be able to utilize XLMiner SDK at "basic" limits.

Installation

What you Need

Solver SDK 2025 Q4 software, which includes both Solver SDK and XLMiner SDK, you must be running one of the following operating systems: Windows 10, Windows 8, Windows 7, Windows Vista, Windows Server 2016, Windows Server 2012, Windows Server 2008. XLMiner SDK does not support a 32-bit version.

Getting Help

Frontline Systems technical support is available to offer assistance with installation of the XLMiner SDK by contacting Support by phone (888-831-0333) or email (support@solver.com).

Installing the Software

SDKSetup.exe installs both Solver SDK and XLMiner SDK. Your license determines your product selection. Your license may grant access to either Solver SDK or XLMiner SDK, or both. For more information on licensing, see *Logging in the First Time* below.

Note: Licenses for both Solver SDK Platform and Solver SDK Pro allow the use of XLMiner SDK at "basic" limits. See https://www.solver.com/xlminer-sdk-platform for more information.

To install Solver SDK and XLMiner SDK to work with **64-bit** version of Windows, run **SDKSetup64.exe**. Note: XLMiner SDK does not support a 32-bit version.

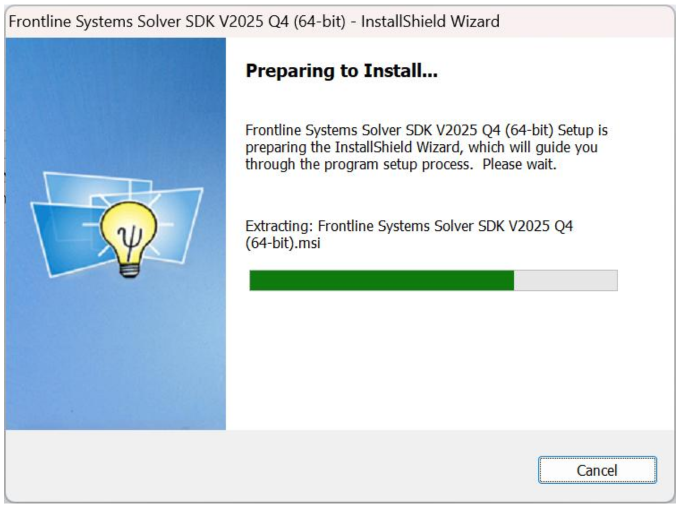
Depending upon your Windows security settings, one of the following prompts (displaying Frontline Systems, Inc. as publisher) may appear:

Do you want to run this software?

Do you want to allow this app to make changes to your device?

Click **Run** or **Yes** to proceed with installation of XLMiner SDK.

During file decompression, the following screen is displayed.



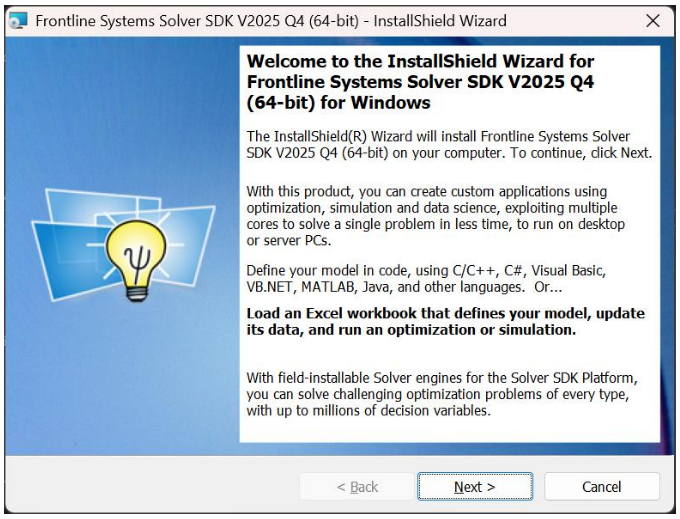
Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wizard

Preparing to Install...

Frontline Systems Solver SDK V2025 Q4 (64-bit) Setup is preparing the InstallShield Wizard, which will guide you through the program setup process. Please wait.

Extracting: Frontline Systems Solver SDK V2025 Q4 (64-bit).msi

After the files are decompressed, the Welcome Screen will appear. (File decompression can take up to 5 minutes.)



Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wizard

Welcome to the InstallShield Wizard for Frontline Systems Solver SDK V2025 Q4 (64-bit) for Windows

The InstallShield(R) Wizard will install Frontline Systems Solver SDK V2025 Q4 (64-bit) on your computer. To continue, click Next.

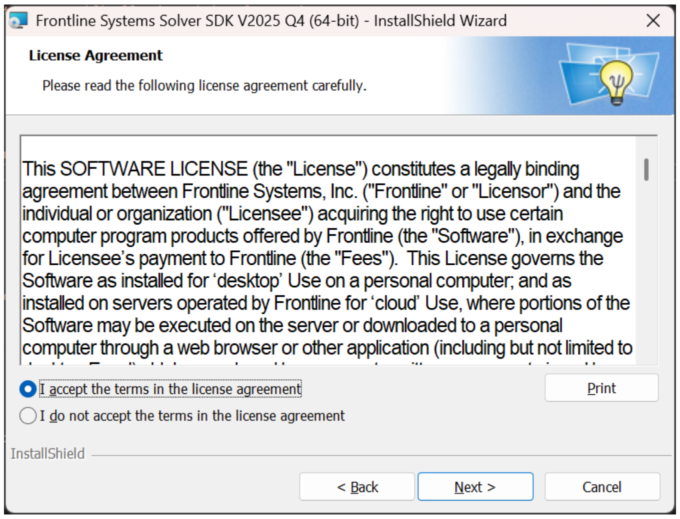
With this product, you can create custom applications using optimization, simulation and data science, exploiting multiple cores to solve a single problem in less time, to run on desktop or server PCs.

Define your model in code, using C/C++, C#, Visual Basic, VB.NET, MATLAB, Java, and other languages. Or...

Load an Excel workbook that defines your model, update its data, and run an optimization or simulation.

With field-installable Solver engines for the Solver SDK Platform, you can solve challenging optimization problems of every type, with up to millions of decision variables.

Click **Next** to proceed. In response to the License Agreement dialog, select *I accept the terms in the license agreement*, and click **Next** to proceed.



Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wizard

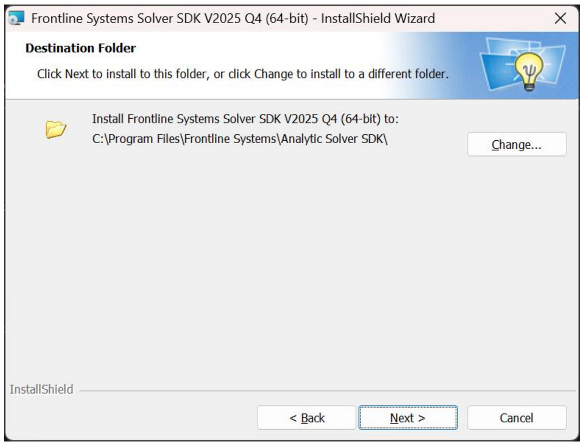
License Agreement

Please read the following license agreement carefully.

This SOFTWARE LICENSE (the "License") constitutes a legally binding agreement between Frontline Systems, Inc. ("Frontline" or "Licensor") and the individual or organization ("Licensee") acquiring the right to use certain computer program products offered by Frontline (the "Software"), in exchange for Licensee's payment to Frontline (the "Fees"). This License governs the Software as installed for 'desktop' Use on a personal computer; and as installed on servers operated by Frontline for 'cloud' Use, where portions of the Software may be executed on the server or downloaded to a personal computer through a web browser or other application (including but not limited to

[✓ I accept the terms in the license agreement]
I do not accept the terms in the license agreement

In the Destination Folder screen, click **Next** to install the software into the selected folder, or click **Change** to designate another destination folder. Click **Next** to proceed.



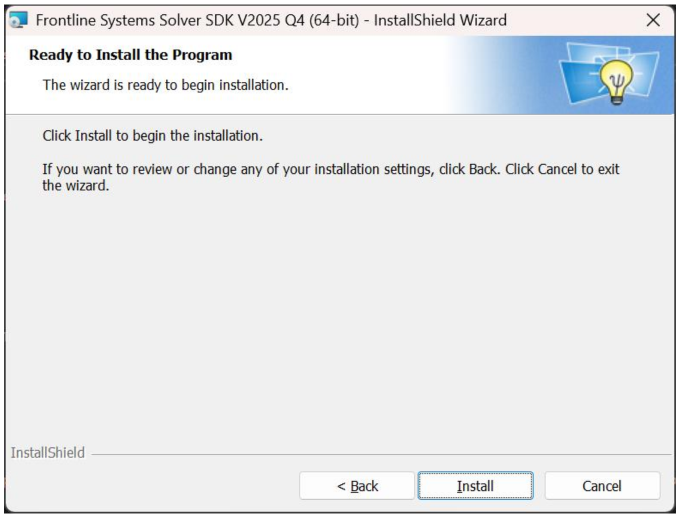
Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wizard

Destination Folder

Click Next to install to this folder, or click Change to install to a different folder.

Install Frontline Systems Solver SDK V2025 Q4 (64-bit) to:
C:\Program Files\Frontline Systems\Analytic Solver SDK\

Click **Next** to proceed. The next screen indicates that the preliminary steps are complete and installation is ready to begin. To change any installation settings, click **Back**; otherwise click **Install** to proceed.


Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wizard

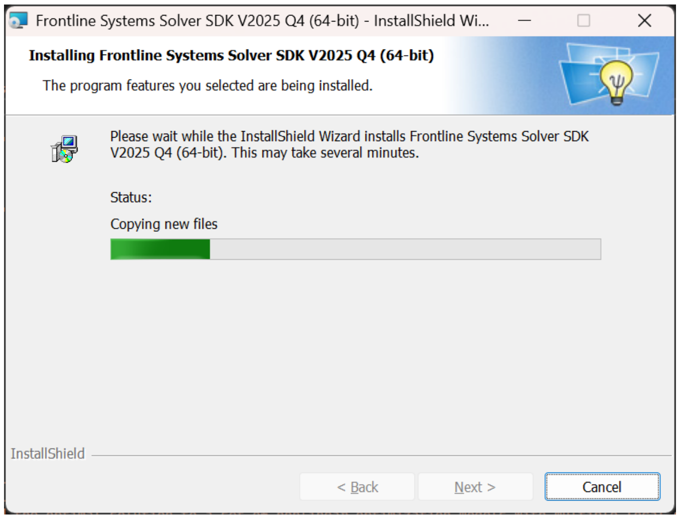
Ready to Install the Program

The wizard is ready to begin installation.

Click Install to begin the installation.

If you want to review or change any of your installation settings, click Back. Click Cancel to exit the wizard.

While the SDK files and examples are being installed, the progress is displayed on the following screen. This may take several minutes to complete.


Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wi...

Installing Frontline Systems Solver SDK V2025 Q4 (64-bit)

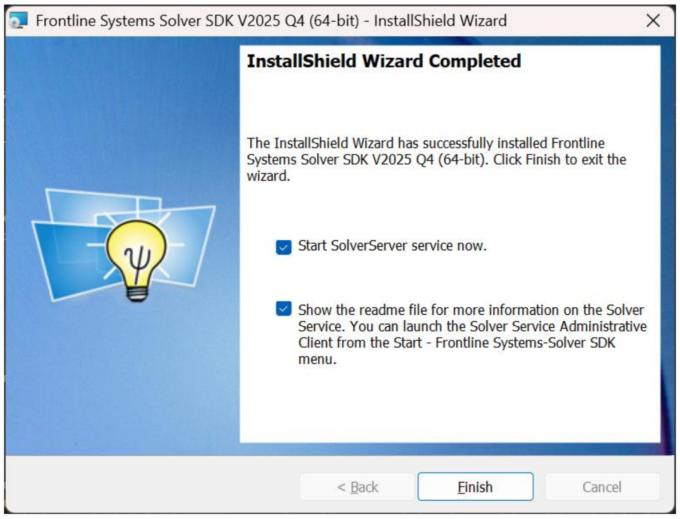
The program features you selected are being installed.

Please wait while the InstallShield Wizard installs Frontline Systems Solver SDK V2025 Q4 (64-bit). This may take several minutes.

Status:

Copying new files

When the installation is complete, the following screen is displayed. Click **Finish** to exit the InstallShield Wizard.


Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wizard

InstallShield Wizard Completed

The InstallShield Wizard has successfully installed Frontline Systems Solver SDK V2025 Q4 (64-bit). Click Finish to exit the wizard.

☑ Start SolverServer service now.

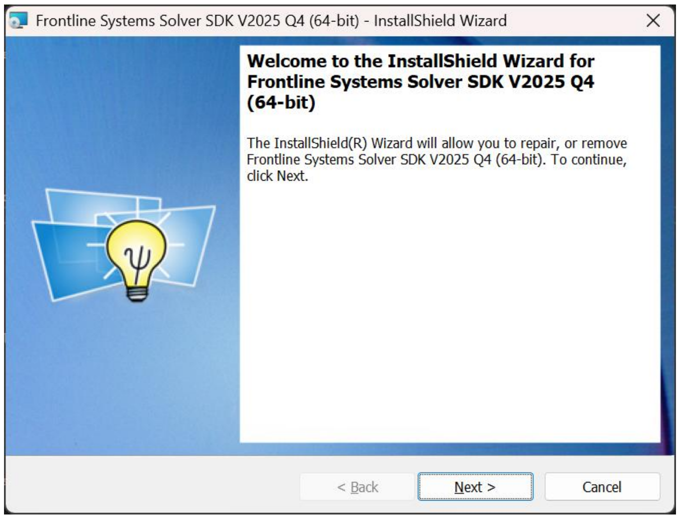
☑ Show the readme file for more information on the Solver Service. You can launch the Solver Service Administrative Client from the Start - Frontline Systems-Solver SDK menu.

If you would like to start the SDK Dedicated Server service, which allows a Client machine (PC, phone, tablet, etc.) to solve a model remotely with the Solver SDK Platform, then please check "Start the SolverServer Service now" checkbox. If you choose not to start the SDK Dedicated Server service at this time, it can be started manually at a later time. For more information on our SDK Dedicated Server service, please see the chapter, **Using SDK Dedicated Server**.

Once Solver SDK and XLMiner SDK are installed, you may find examples, help and library files at: C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples.

Uninstalling the Software

To uninstall Solver SDK, run the **SDKSetup** or **SDKSetup64** program as outlined above. A prompt will ask you to confirm whether you want to uninstall the software.


Frontline Systems Solver SDK V2025 Q4 (64-bit) - InstallShield Wizard
< Back
Next >
Cancel

Welcome to the InstallShield Wizard for Frontline Systems Solver SDK V2025 Q4 (64-bit)

The InstallShield(R) Wizard will allow you to repair, or remove Frontline Systems Solver SDK V2025 Q4 (64-bit). To continue, click Next.

Getting Started

Introduction

This chapter details the steps required to compile, link, and run the XLMiner SDK examples in C#, C++, JAVA, PYTHON, and R.

C++/C#: To simplify the input/output and focus on the XLMiner SDK routines and arguments, the C++ and C# example applications in this chapter use console calls for output, and were built as console applications in Microsoft Visual Studio

JAVA: See "Calling SDK from Java" for instructions on how to open and run the Java examples.

PYTHON: XLMiner SDK supports Python V3.7, 3.8 and 3.9.

R: See "Calling SDK from R" for instructions on how to open and run the R examples. XLMiner SDK supports any 4.X version of R.

SDKSetup64.exe installs a 64-bit XLMiner SDK into the path C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK. The path to the XLMiner SDK **dynamic libraries** is added to the system PATH during installation.

The following subfolders are installed into the Frontline Systems\Analytic Solver SDK\XLMiner directory: **Bin**, **Examples**, **Include**, and **Lib.** *Note: The Bin directory (located at …\Frontline Systems\Analytic Solver SDK\XLMinerSDK\) is added to the PATH during installation.*

- **Bin** – Contains the **XLMinerSDK.dll** along with other required dynamic libraries. This folder also includes the **_XLMiner.pyd** files for Python versions 3.7, 3.8 and 3.9 .
- **Examples** – Example code in five different programming languages (C#, C++, Java, Python and R) is installed into the Examples directory, illustrating how to call each method within XLMiner SDK. Example data sets referenced in the code are installed into the **Datasets** folder. See the following sections for instructions on how to run each example project.
- **Include** – Contains C++ header files to be used for development or included with **XLMinerAPI.h** (includes all header files).
- **Lib** – XLMiner SDK library files are installed into the Frontline Systems\Analytic Solver SDK\XLMinerSDK\Lib directory. If using C++, link to **XLMinerSDK.lib**, **Evaluator.lib**, and **XLData.lib** in your project or linkers response file (see Calling XLMiner SDK from C++). For JAVA, Python and R, **XLMinerJavaLibrary.jar**, **XLMiner.py** and **XLMiner.zip** are also installed into a **Lib** folder.

Many of the details of using the XLMiner SDK will vary depending upon the development environment you prefer to use. Since it is impractical to provide examples for all techniques in all programming languages within this Guide, this Guide will illustrate how to call XLMiner SDK using the C# programming language. Examples in the remaining supported languages can be found in the installation path at C:/Program Files/Frontline Systems/Analytic Solver SDK/XLMinerSDK\Examples.

Using Interactive Environments

The classic edit-compile-test cycle for program code remains the only practical method for producing professional applications and program libraries. However, thanks largely to multicore chips and copious RAM, there has be a great increase in recent years in the popularity of interactive environments. Indeed, interactive environments are often the best way to go for preliminary evaluations and exploratory data analysis.

R

R has always been an interactive environment, but the widely-used R Studio has extended the convenience of a graphical user interface, and more recently with support for R Notebooks. A Notebook supplements the interactivity of R by not only executing commands, but by recording the commands and output, including graphical output, into HTML format.

Python

Jupyter, originally dubbed “IPython”, provides a splendid way to learn XLMiner and perform interactive data analyses.

C# Interactive

C# Interactive, a relative newcomer, is available in two forms. Visual Studio 2017 includes an integrated C# interpreted command line, closely analogous to the F# command line introduced in earlier versions. The interactive C# environment within Visual Studio 2017 is a strictly 32-bit application and cannot be used with 64-bit XLMiner SDK. Perhaps more interesting and valuable is the standalone CSI command line REPL environment, which provides a fast and fun way for the C# developer to learn to manipulate XLMiner DataFrames and to perform interactive analyses.

64-Bit XLMiner SDK

The XLMiner setup program will install the SDK in the “Frontline Systems” folder, typically C:\Program Files\Frontline Systems\Analytic Solver SDK. In general, administrative permissions are required to modify files in subdirectories of Program Files, so it is generally easier to copy the XLMiner SDK examples into a different directory for compilation and testing. XLMiner SDK does not support a 32-bit version.

Calling XLMiner SDK from C++

To open the example code for C++:

Open Microsoft Visual Studio V2013 or later

Browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\C++.

Open the file **Test.vcxproj.** *Note:* Confirm that your target platform is x64 before compiling and running this project file.

When creating your own application:

To create an application that calls the XLMiner SDK in C++, you must include the import library files **XLData.lib**, **XLMinerSDK.lib**, and **Evaluator.lib** in your project or linkers response file. Alternatively, you can write code to use “run-time dynamic linking” by calling Windows API routines such as LoadLibrary(). Library files are installed into the **Lib** folder (see above for path).

To access the XLMiner SDK functions, you must add an **include** statement for the file **XLMinerAPI.h**. You can also add an optional **using namespace** directive for XLMiner.

Calling XLMiner SDK from C#

Calling XLMiner SDK from C# is especially easy. The programming environment, whether Visual Studio 2013 or later or C# Interactive, need only have a reference to XLMinerCSLibrary.dll. This library calls the unmanaged code in the native XLMiner libraries from managed .Net code generated by C#.

There is no better way to discover the many capabilities of the XLMiner SDK than to establish a reference in a Visual Studio project and then examine XLMiner using the Object Explorer.

In the C# Interactive environment, this is done from the command prompt using a reference directive. Alternatively, C# Interactive can be invoked specifying a startup script referencing the library. This latter alternative is much easier if you expect to be using C# interactive frequently.

In #r, use: "C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Bin\XLMinerCSLibrary.dll."

If you do not utilize "using" directives, you will have to preface class names with "XLMiner". This is sometimes advantageous for beginners as a reminder of where various methods are defined. It quickly becomes tedious, however, and professional XLMiner developers generally add "using XLMiner;" to their code.

To open the example code for C#:

Open Microsoft Visual Studio V2013 or later

Browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\C#.

Open the file **Test.csproj.**

Setting a reference—To access the objects in XLMiner SDK, you must set a reference. During installation of the SDK, a C# library XLMinerCSLibrary.dll is installed into the Bin directory in C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Bin.

Adding a directive—Adding the directive **using XLMiner** is optional in C#. If you choose not to add the directive, you must add **XLMiner** to the beginning of each SDK function call.

Calling XLMiner SDK from Python

The Python interpreter reads and parses the example script, compiles it into bytecode, and executes in run-time. To run the Python example, the required Python libraries must be downloaded and installed.

The XLMiner SDK for Python consists of native binaries compiled with certain configurations. As a result, it is important to install the appropriate version of Python. Before calling XLMiner SDK, confirm that Python DLLs are accessible using the system path.

There are two ways to view and run the Python examples: by command prompt executing sequential commands, or by writing a full script and executing the full code using a Python notebook or a full Python IDE.

Command Prompt—if using a command prompt, simply open a **python.exe** interpreter (or call it for each command) to work in "interactive shell" mode. Using this method, single commands are executed in succession.

Using a Python IDE or Notebook—open a Python IDE to execute a full script such as **PyCharm** by JetBrains, Visual Studio 2017 (with Python tools installed), or a Python Notebook such as **Jupyter**.

To call XLMiner SDK from Python, you will need the following:

_XLMiner.pyd and **XLMinerSDK.dll** (along with other binaries that **XLMinerSDK.dll** depends upon) are installed into the **Bin** folder (see above for complete path). Both files must be on the **PYTHONPATH** or in the same folder. The **Bin** folder (C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Bin) is added to the system path during installation. You also need **XLMiner.py**, which is installed into **Lib** folder.

To import the XLMiner extension module using `from XLMiner import *`, see **Examples.py**.

Calling XLMiner SDK from Java

To open the example code for Java:

Browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\Java.

Open Examples.java using an editor of your choice or a JAVA IDE

See the Lib folder within C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK.

Import all (or a portion) of the XLMiner API using:

import XLMiner.*;

import XLMiner.Binning.IntervalType;

If you choose not to import these packages, you must alter your function calls before use. For example, rather than *sampler.setReplaceOption(ReplaceType.REPLACE_NO)*, you would be required to use *sampler.setReplaceOption (XLMiner.Sampling.ReplaceType.REPLACE_NO)*.

The file **runJavaXLMinerSDK.bat** executes a simple .bat script that compiles and runs the JAVA examples. This is optional; only the JAVA Development Kit (JDK) is required to run these examples.

Calling XLMiner SDK from R

In the XLMiner examples for R, there are two scripts, Main.R and Examples.R. The actual use of XLMiner functions can be studied from the Examples.R. file. Main.R exists not only to invoke the examples, but also to install required packages. The XLMiner package has two prerequisites, Rcpp and R6, both can be downloaded from the standard R mirror servers. Rcpp is the most widely used package for providing an interface between R and compiled code written in C++, and R6 extends R with some additional capabilities for handling object references.

Once the prerequisites have been installed, the XLMiner package can be installed. This can be done by running Main.R, or it can be installed manually if you prefer. The zip file necessary for installation is XLMiner.zip and is in XLMiner SDK\Lib. (XLMiner SDK V2025 Q4 does not support 32-bit R.)

To open example code for R:

Browse to C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\R.

Examples.R contains definitions for R functions that illustrate sample usage of XLMiner SDK. **Main.R** contains the code to install the required R packages (Rcpp/R6), and defines some common data structures used for running the examples.

To run all examples, R command *source(file,...)* can be called with **Main.R**. This method works when using the R console or any R IDE. One of the most popular IDEs for R is **RStudio**, which is free. Along with scripting capabilities similar to R console, **RStudio** offers GUI and interactive features to work with R. For more details regarding **RStudio**, refer to the **RStudio** documentation.

Note: R has its own definition of **DataFrame**, which is very rich and convenient for the R environment. Therefore, XLMiner SDK for R provides the interface to work with native R **DataFrames**, performing automatic conversions.

Mining Data with XLMiner SDK

The XLMiner SDK Pattern

Data Preparation | Creating the Model | Examining and Applying the Results

The following chapters contain the information needed to call XLMiner SDK from the supported programming languages: C++, C#, Java (Scala), Python and R. While reading these chapters, keep in mind the three steps to any data mining pipeline: Data Preparation, Model Creation, and Result examination and application.

Data Preparation

Regardless of the source of the data, it must be brought into an XLMiner DataFrame before it can be passed to any XLMiner mining engine.

Model Creation

Next, an Estimator class is instantiated for a specific algorithm such as the Classification Decision Tree. After the Estimator is instantiated, the fit method will create an instance of the Model class – here is where the bulk of XLMiner's work is done.

Result Examination and Application

The properties and methods of a Model object provide users access to all information specific to the mining model of interest.

Usually, the results of a data mining task are themselves DataFrames. Not only does this ease the task of the developer by providing a common interface, it also makes it easy to use the output of one mining task as the input to another. For example, Neural Network algorithms are powerful, but they are also computationally intensive. Sometimes it is advantageous to use a task such as Feature Selection, PCA, and others to reduce the work required by the demanding Neural Network algorithm.

See the next chapter to examine the details of data preparation in XLMiner SDK.

Data Preparation

Introduction

Data gathered in the real world is never ready for analysis as is. Because virtually all modern development environments can interface with XLMiner SDK, your data can be prepped, cleansed and organized using whichever environment your team prefers. Regardless of whether your team works in R, C#, Python, or some other environment, the data itself may be available in a variety of formats.

DataFrames

Most of the XLMiner data mining engines require input data in the form of a DataFrame. Since different host environments deal with DataFrames each in their own way, we shall have to take a few moments to see how DataFrames work. We'll compare C#, R, and Python.

Despite the similar appearance of DataFrames across various development environments, the internal structure of the DataFrames included in these assorted environments is quite varied. These variations have enormous effects on data mining performance, both in terms of speed and the ability to handle large data volumes. XLMiner's DataFrames are crafted to take full advantage of the processing capabilities of modern CPU architectures.

DataFrames generally require some type of preprocessing, such as partitioning into training, validation and test subsets. XLMiner developers can either choose to manipulate the DataFrames in XLMiner SDK, prior to invoking data mining procedures, or in their host programming system. Then structure the data accordingly and load the finished product into XLMiner for analysis. Many XLMiner users choose to manipulate their DataFrames into the desired structure in R, which has over the years acquired a rich set DataFrame tools.

Basic DataFrame Tasks

- Creating and filling DataFrames
- Selecting a subset of columns or rows
- Appending columns or rows
- Selecting subsets for training, verification and testing of models.

The basic operations often appear in guises critical to data mining goals

The Dataset

Virtually all operations in XLMiner utilize DataFrames for input and output. It is therefore convenient to develop a class to encapsulate the most common XLMiner DataFrame operations and specifications. Such a class, dubbed XLMiner Dataset, can easily be implemented in C#. An example appears below. For an example of this class in the remaining supported programming languages see …/Frontline Systems/Analytic Solver SDK/XLMinerSDK/ Examples in the installation path.

Dataset class in C#

```
public class Dataset : IDisposable
{
public int targetColumnIndex = 0;
public DataFrameVector input, target;

public Dataset(DataFrame data, string targetColName, double[] ratios) :
```

```
            this(data, data.ColIndex[targetColName], ratios)
            { }

      public Dataset(DataFrame data, int targetColumnIndex, double[] ratios)
      {
            this.targetColumnIndex = targetColumnIndex;
            var partitions = Partition(data, ratios);
            int[] targetIndexSet = new int[] { targetColumnIndex };

            input = new DataFrameVector();
            target = new DataFrameVector();

            input.push_back(partitions[TRAINING][Util.all,
            Util.except(targetIndexSet)].setName(TRAINING));
            target.push_back(partitions[TRAINING][Util.all,
            targetIndexSet].setName(TRAINING));
            input.push_back(partitions[VALIDATION][Util.all,
            Util.except(targetIndexSet)].setName(VALIDATION));
            target.push_back(partitions[VALIDATION][Util.all,
            targetIndexSet].setName(VALIDATION));
      }

      public Dataset(DataFrame data, int targetColIndex) :
            this(data, targetColIndex, new double[] { 0.6, 0.4 })
            { }
      public Dataset(DataFrame data, string targetColName) :
            this(data, data.ColIndex[targetColName])
            { }
      public int NumTrainingRows
            {
               get { return trainingInputColumns.NumRows; }
            }
      public void Dispose()
            {
               Util.free(ref input);
               Util.free(ref target);
            }
      }
```

DataFrame Manipulation for R, Python, and C#

While all development environments provide means to achieve the same goals, there are substantial differences among the various programming languages and host environments.

Using DataFrames in C#

The Microsoft .Net Framework does not have a column-oriented structure like a DataFrame, so C# developers will likely wish to use XLMiner's built-in capabilities. DataFrames are easily created by reading data files directly into XLMiner using the XLMiner.Reader class.

Creating DataFrame by reading data files directly into SDK using XLMiner.Reader class

```
XLMiner.DataFrame df = XLMiner.Reader.textFile(PATH + "AdmissionsData-
001.csv");
```

Once we have created an XLMiner DataFrame, we can easily manipulate it.

The code below creates a new DataFrame, *df2*, using columns 0, 1, and 2 from the existing DataFrame, *df*.

Creating new DataFrame using columns 0, 1, and 2 from existing DataFrame

```
XLMiner.DataFrame df = XLMiner.Reader.textFile( "AdmissionsData-001.csv");
int[] test = { 0, 1, 2 };
var df2 = df[XLMiner.Util.All.value, test];
```

This code snippet creates a new DataFrame, *df3*, by including the first and third columns from the existing DataFrame *df*, while dispensing with a separate declaration for the array containing the column indexes.

Creating new DataFrame while dispensing with separate declaration for array containing column indexes

```
var df3 = df[XLMiner.Util.All.value, new int[] { 1,3}];
```

Note C# does not support syntax of the sort: "*MyStructure[:,3]*". As a result, XLMiner SDK provides an enumeration, *Util.All,* with only one value, the sole purpose of which is to indicate that all rows (or all columns) should be selected. A related method, *Util.except*, can be used to exclude unwanted columns when passing a DataFrame as an argument.

It is also possible to select rows and columns not only by indexes, but also by name.

Selecting rows and columns by name

```
print("Extracting a sub-DataFrame\n", df[2:, 0:2].Value)
print("Extracting a sub-DataFrame\n", df[["Record 1", "Record 2"],
["Feature 1", "Feature3"]].Value)
print("Extracting a sub-DataFrame\n", df[["Record 1", "Record 2"],
[0,2]].Value)
print("Extracting a sub-DataFrame\n", df[0:2, ["Feature 1", "Feature
3"]].Value)
```

Creating DataFrames from C# Arrays

In previous examples, we created XLMiner DataFrames by reading data from text files. We can also create DataFrames directly from C# data by appending single-dimensional arrays to a new DataFrame. Each of the appended arrays must have the same length and must be of datatype int, double, or string.

Create DataFrame directly from C# Data by appending single-dimensional arrays to new DataFrame

```
public static void RowsToDataFrame001()
    {
        int i = 0;
        int rowcount;
        int[] educ;
        int[] exper;
        double[] wage;
        string connString = "Data Source=localhost;initial catalog =
        StatisticsTest; Integrated security = SSPI";
        using (SqlConnection conn = new SqlConnection(connString))
        {
            using (SqlCommand cmd = new SqlCommand())
            {
                cmd.Connection = conn;
                conn.Open();
                cmd.CommandText = "SELECT COUNT(*) FROM [Wage-01]";
                rowcount = (int)cmd.ExecuteScalar();
                educ = new int[rowcount];
                exper = new int[rowcount];
                wage = new double[rowcount];
                cmd.CommandText = "SELECT wage, educ, exper FROM [Wage-
                01]";
                SqlDataReader reader = cmd.ExecuteReader();
                while (reader.Read())
                {
                    educ[i] = reader.GetInt32(1);
                    exper[i] = reader.GetInt32(2);
                    wage[i] = reader.GetDouble(0);
                    i++;
                }
            }
            XLMiner.DataFrame df = new XLMiner.DataFrame();
            df.append(wage);
            df.append(educ);
            df.append(exper);
            Console.WriteLine(df);
        }
    }
```

Using DataFrames in R

In contrast with C#, R possesses rich support for DataFrame structures. R developers using the XLMiner SDK can use R to manipulate DataFrames, and then supply the DataFrames to XLMiner methods as arguments. There is no support for directly manipulating XLMiner DataFrames within R; it would simply serve no purpose.

In addition to the core R support for DataFrames, there are also R packages that provide additional support for DataFrame work in R. Among the more popular is **dplyr**.

A note about Microsoft R…

In 2017, Microsoft made a major commitment to R with the acquisition of Revolution Analytics. There are now three editions of Microsoft R. Microsoft Open R provides the same functionality as other R implementations. Microsoft R Client is designed for the data scientist handling data on his or her workstation, and Microsoft R Server provides support for greater parallelism, data files spread across different disks, and integrated support for Hadoop, Teradata, and of course SQL Server. Both the Client and the Server editions include proprietary R packages not available for open-source R versions. Among the proprietary features in R Client and R Server is support for xdf files which are extended data frames that permit the R developer to work with data frames that, in the past, have been too large to fit in RAM. The xdf files can be read and processed in "chunks" so that the entire file is not required to be loaded into memory all at once, thereby greatly increasing the data volume that can be analyzed.

Using DataFrames in Python

In contrast with both C# and R, Python provides for the manipulation of XLMiner DataFrames in a manner which is both direct and completely intuitive for the Python developer.

In these examples, data, df02, and df03 are both XLMiner DataFrames

Manipulating XLMiner DataFrames in Python

```
df02=data[:,[1,3]]
df03=data[:,[0,1,2,3]]
```

You can obtain an XLMiner.DataFrameColumn using a column name.

Obtaining XLMiner DataFrameColumn using column name

```
Dfc01 = data[:, "CRIM"]
```

However, in Python you also can create a new DataFrame with multiple columns using names instead of indices.

Creating DataFrame using column names

```
f04=data[:,['CRIM', 'INDUS']] # using column names
f03=data[:,[0,1,2,3]] # using column indices
```

Partitioning DataFrames for Training and Validation

One of the risks of data mining in general is the potential for “overfitting” the data. It is often possible to find an exceedingly detailed model that fits the training data almost perfectly and then fails miserably the first time it is applied to a novel data point. To avoid the danger of overfitting, it is standard practice to use only a subset of the available data to build the model, holding back some data for subsequent validation and testing of the model. As mentioned in the data preparation section, XLMiner provides methods for partitioning the data into two or more sets for training the model and then validating it.

XLMiner provides a Partitioner class that makes it easy to create DataFrame subsets for training and validating mining models. In fact, you are not limited to two partitions, you may have several partitions as long as the sum of the probabilities for each partition is 1.0.

Partitioning dataset into training and validation sets

```
XLMiner.DataFrame df = XLMiner.Reader.textFile(PATH + "wine.txt");
Console.WriteLine("Original DataFrame NumRows: {0}", df.NumRows);
XLMiner.Partitioner partitioner = new XLMiner.Partitioner();
    partitioner.PartitionRatio["Training"] = 0.6;
    partitioner.PartitionRatio["Validation"] = 0.4;
DataFrameVector partitions = partitioner.partition(df);
```

Rows may be partitioned either randomly or based on their sequence in the DataFrame. The default is random, but repeated executions of the same code will always result in the same random numbers, and thus the same row subsets. This is generally preferable during development and testing. To achieve an unpredictable sequence in a production environment you must provide a random seed prior to partitioning the DataFrame. See below for an example of how to supply a random seed:

Setting a random seed prior to partitioning a DataFrame

```
XLMiner.Partitioner partitioner = new XLMiner.Partitioner();
partitioner.Seed = (new System.Random()).Next(System.Int32.MaxValue);
```

Synthetic Data Generation

Introduction

The newly released Synthetic Data Generation feature included in the latest version of XLMiner SDK allows users to generate synthetic data by automated Metalog probability distribution selection and parameter fitting, Rank Correlation or Copula fitting, and random sampling. This can be beneficial for several reasons such as when the actual training data is limited or when the data owner is unwilling to release the actual, full dataset but agrees to supply a limited copy or a synthetic version that statistically resembles the properties of the actual dataset.

This process consists of three main steps.

1. Fit and select a marginal probability distribution to each feature – by automated and semi-automated search within the family of bounded, semi-bounded or unbounded Metalog distributions.
2. Identify correlations among features, by using Rank Correlation or one of available Copulas – Clayton, Gumbel, Frank, Student or Gauss.
3. Generate the random sample consistent with the best-fit probability distributions and correlations.

Additional to the generated synthetic data, XLMiner SDK can optionally provide the details of the fitting process – fitted coefficients and goodness-of-fit metrics for all fitted candidate Metalog distributions, selected distribution for each feature and fitted correlation matrix.

To further explore the original/synthetic data and compare them, one may easily compute basic and advanced statistics for original and/or synthetic data, including but not limited to percentiles and Six Sigma metrics.

Examples

This guide provides three examples, each demonstrating the use of the SyntheticDataGenerator object in the C# Programming language. Other supported languages (C++, Java, Python, R) provide similar functionality with similar APIs. All examples, in all APIs, can be found in the installation directory C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples.

1. The first example, Summarizing Data Example, illustrates how to obtain statistics, percentiles and six sigma metrics on a dataset.
2. The second example, Synthetic Data Generator, is a standalone example illustrating the use of the SyntheticDataGenerator object.
3. The third example, LinearRegressionSimulation, shows the main Synthetic Data Generation real-world application for the Risk Analysis of Machine Learning (ML) models.

Summarizing Data Example

This example illustrates how a user could obtain statistics, percentiles and six sigma metrics on a dataset. For results, see the XLMiner SDK example code found at C:\Program Files\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples.

Summarizing Example Code

```
public static class Simulation
{
  public static int Summarizing()
  {
    try
    {
      using DataFrame data = Reader.textFile(PATH + "bh-scale-reg.txt");

      Console.WriteLine(Summarizer.summary(data));
      Console.WriteLine(Summarizer.advancedSummary(data));
      Console.WriteLine(Summarizer.sixSigma(data));
      Console.WriteLine(Summarizer.percentiles(data));
    }
    catch (XLMiner.Exception ex)
    {
      Console.WriteLine("Exception has occurred at
<Simulation.Summarizing>:\n" +
      ex.Message);
      return 1;
    }
    return 0;
 }
```

Synthetic Data Generator Example

This example demonstrates the API of SyntheticDataGenerator object in the C# programming language, consistent with the 3-step process defined above: fit and select distributions and correlations for all columns in the dataset and generate the synthetic data. The original/synthetic data can be further analyzed using "summarization" functionality as shown in the previous example.

<u>**Synthetic Data Generation Example Code**</u>

```
public static int SyntheticDataGeneration()
{
  try
  {
    using var data = Reader.textFile(PATH + "bh-scale-reg.txt");
    using var sdg = new SyntheticDataGenerator();
```

Create new Synthetic Data Generator() object, sdg.

```
// STEP 1: setup the marginal Metalog distribution fitting for all
// data columns
    {
      // auto/manual Metalog fit - True = Auto, False = Manual
      sdg.MetalogAuto = true;

      // OPTION 1: If Manual, specify fixed # Metalog terms for each column
      if (false == sdg.MetalogAuto)
      {
        // specify fixed # Metalog terms for each column (in this example
        // same for all columns)
        for (var i = 0; i < data.NumCols; i++)
        sdg.NumMetalogTerms[data.ColName[i]] = 2;
      }
      // OPTION 2: auto search for the best Metalog up to the specified #
      // terms
      else
      {
         for (var i = 0; i < data.NumCols; i++)
         {
          // specify the max # terms for the search for each column. max
          // range is [2,16]
          // (in this example same for all columns)
          sdg.NumMetalogTerms[data.ColName[i]] = 5;
          // choose the GOF metric to select the best Metalog for each
          // column (in this
          // example same for all columns)
          // CS, KS, AD, AIC, BIC, AICc, BICc, ML
          sdg.MetalogGoodnessOfFitType[data.ColName[i]] =
          SDG.GoodnessOfFitType.AIC;
        }
      };
      // optionally, set the lower/upper bound for some or all columns that
      // will be used
      // in Metalog fitting
      //sdg.MetalogLowerBound["CRIM"] = 0.00632; sdg.MetalogUpperBound["CRIM"] =
      //88.9762;
      //sdg.MetalogLowerBound["ZN"] = 0; sdg.MetalogUpperBound["ZN"] = 100;
      //sdg.MetalogLowerBound["INDUS"] = 0.46; sdg.MetalogUpperBound["INDUS"] =
      //27.74;
      //sdg.MetalogLowerBound["NOX"] = 0.385; sdg.MetalogUpperBound["NOX"] = 0.871;
      //sdg.MetalogLowerBound["RM"] = 3.561; sdg.MetalogUpperBound["RM"] = 8.78;
      //sdg.MetalogLowerBound["DIS"] = 1.1296; sdg.MetalogUpperBound["DIS"] =
      //12.1265;
      //sdg.MetalogLowerBound["AGE"] = 2.9; sdg.MetalogUpperBound["AGE"] = 100;
      //sdg.MetalogLowerBound["TAX"] = 187; sdg.MetalogUpperBound["TAX"] = 711;
      //sdg.MetalogLowerBound["PTRATIO"] = 12.6; sdg.MetalogUpperBound["PTRATIO"] =
      //22;
      //sdg.MetalogLowerBound["B"] = 0.32; sdg.MetalogUpperBound["B"] = 396.9;
      //sdg.MetalogLowerBound["LSTAT"] = 1.73; sdg.MetalogUpperBound["LSTAT"] =
      //37.97;
      //sdg.MetalogLowerBound["MEDV"] = 5; sdg.MetalogUpperBound["MEDV"] = 50;

      // or, use min/max of the columns data as bounds
      // note that if manual LowerBound/UpperBound was set before, it's not
      // overwritten
      sdg.UseMinMaxAsBounds = true;
      // at this point, can overwrite specific bounds as well
    };

// STEP 2: setup the correlation fitting for all data columns
    {
      // OPTION 1: no correlation, independent sample
      //sdg.CorrelationMethod = SDG.CorrelationType.NONE;

      // OPTION 2: Rank correlation (default)
      sdg.CorrelationMethod = SDG.CorrelationType.RANK;

      // OPTION 3: select copulas
      {
        //sdg.CorrelationMethod = SDG.CorrelationType.COPULA;

        // order sets the priority of copulas; if multiple selected, first
        // successfully fit copula will be selected and used for sample
        // generation
        //sdg.Copula[SDG.CopulaType.STUDENT] = true;
        //sdg.Copula[SDG.CopulaType.CLAYTON] = true;
        //sdg.Copula[SDG.CopulaType.FRANK] = false;
```

```
         //sdg.Copula[SDG.CopulaType.GUMBEL] = true;
         //sdg.Copula[SDG.CopulaType.GAUSS] = true;
       };
     };
// STEP 3: generate data
     {
       // optional, not setting at all or setting to 0 means no seed
       sdg.RandomSeed = 12345;
       // below also optional, advanced options for random generator,
       // sampling method and random streams
       sdg.RandomGenerator = SDG.RandomGeneratorType.MERSENNE_TWISTER;
       sdg.SamplingMethod = SDG.SamplingMethodType.LATIN_HYPERCUBE;
       sdg.RandomStream = SDG.RandomStreamType.INDEPENDENT;

       // # trials to simulate
       sdg.SampleSize = 100;
     };

     // display estimator
     Console.WriteLine(sdg);

     // fit distributions/correlations
     sdg.fit(data);
```

Fit the Metalog distributions and correlations

```
     // generate sample
     using var sample = sdg.transform(data);
```

Generate the synthetic data

```
     // print the messages log
     Console.WriteLine(sdg.LogDF);

     // print the sample
     Console.WriteLine(sample);

     // check if the Metalog was fitted (all columns or specific one)
     Console.WriteLine("Metalog fitted? " + (sdg.MetalogFittedAll ? "Yes" :
     "No"));
     //Console.WriteLine("Metalog fitted for RM? " + (sdg.MetalogFitted["RM"]
     //? "Yes" : "No"));

     // get the detailed report of fitted Metalog coefficients for specific
     // columns or all columns
     Console.WriteLine(sdg.MetalogCoefficientsAll);
     //Console.WriteLine(sdg.MetalogCoefficients["RM"]);

     // get the best fitted Metalog for each column
     Console.WriteLine(sdg.BestMetalogDF);

     // get the detailed report of Goodness of Fit for fitted Metalogs for
     // specific columns or all columns
     Console.WriteLine(sdg.MetalogGoodnessOfFitAll);
     //Console.WriteLine(sdg.MetalogGoodnessOfFit["RM"]);

     // check if the correlation was fitted
     Console.WriteLine("Correlation fitted? " + (sdg.CorrelationFitted ?
     "Yes" : "No"));

     // depending on whether the Rank Correlation or Copulas were chosen, get
```

```
      // the correlation fitting results
      if (sdg.CorrelationFitted == true)
      {
        if (sdg.CorrelationMethod == SDG.CorrelationType.RANK)
           Console.WriteLine(sdg.CorrelationSigma);
        else if (sdg.CorrelationMethod == SDG.CorrelationType.COPULA)
        {
           Console.WriteLine("Selected Copula: " + sdg.SelectedCopula);

          if (sdg.SelectedCopula == SDG.CopulaType.CLAYTON ||
          sdg.SelectedCopula == SDG.CopulaType.FRANK || sdg.SelectedCopula ==
          SDG.CopulaType.GUMBEL)
             Console.WriteLine("Copula Theta: " + sdg.CopulaTheta);

          if (sdg.SelectedCopula == SDG.CopulaType.GAUSS || sdg.SelectedCopula
          == SDG.CopulaType.STUDENT)
             Console.WriteLine(sdg.CorrelationSigma);

          if (sdg.SelectedCopula == SDG.CopulaType.STUDENT)
             Console.WriteLine("Student Copula Degrees of Freedom: " +
             sdg.CopulaDegreesOfFreedom);

          Console.WriteLine();
       }
     }
     // get basic/advanced statistics, percentiles, six sigma on
     // original/synthetic data using Summarizer object
     Console.WriteLine(Summarizer.summary(data));
     Console.WriteLine(Summarizer.summary(sample));

     Console.WriteLine(Summarizer.advancedSummary(data));
     Console.WriteLine(Summarizer.advancedSummary(sample));

     Console.WriteLine(Summarizer.sixSigma(data));
     Console.WriteLine(Summarizer.sixSigma(sample));

     Console.WriteLine(Summarizer.percentiles(data));
     Console.WriteLine(Summarizer.percentiles(sample));
   }

   // catch any exceptions
   catch (XLMiner.Exception ex)
   {
     Console.WriteLine("Exception has occurred at
     Simulation.SyntheticDataGeneration>:\n" + ex.Message);
     return 1;
   }
   return 0;
 }
```

Synthetic Data Generation Options

XLMiner SDK provides the following options for Synthetic Data Generation.

Metalog Distribution Fitting Options

- **Metalog Terms**

  `MetalogAuto = true or false`

  - If **False**, XLMiner SDK will attempt to fit the Metalog distribution with the specified number of terms.

    `Example:  sdg.MetalogAuto = false`

    - Number of Terms for Metalog Distributions

      `NumMetalogTerms[colname] = N; N = integer from [2,16]`

      Sets the number of terms for the Metalog Distribution for a given column.

  - If **True**, XLMiner SDK will attempt to fit *all possible* Metalog distributions, with the number of terms limited by the specified value, and select the best Metalog distribution according to the chosen Goodness-of-Fit test.

    `Example:  sdg.MetalogAuto = true`

    - Max Terms for Metalog Distributions

      `NumMetalogTerms[colname] = N; N = integer from [2, 16]`

      Sets the max number of terms for Metalog distribution for a given column.

- **Goodness of Fit Test**

  ```
  MetalogGoodnessOfFitType[colname] = GoodnessOfFitType.<type>
  ```

  ```
  Type  = CHI_SQUARE, KOLMOGOROV_SMIRNOV, ANDERSON_DARLING, AIC,
  BIC, AICc, BICc, MAX_LIKELIHOOD
  ```

  The Goodness of Fit test is used to select the best Metalog form for each column among the candidate distributions defined by a different number of terms, from 2 to the value passed for NumMetalogTerms. The default Goodness-of-Fit test is Anderson-Darling.

  XLMiner SDK offers the following Goodness of Fit Tests:

  - Chi Square – Uses the chi-square statistic to rank the distributions. Sample data is first divided into intervals using either equal probability, then the number of points that fall into each interval are compared with the expected number of points in each interval. The null hypothesis is rejected using a 90% significance level, if the chi-squared test statistic is greater than the critical value statistic.

    ```
    EXAMPLE:

    sdg.MetalogGoodnessofFitType[colName] =
    GoodnessOfFitType.CHI_SQUARE
    ```

  - Kolmogorov-Smirnoff –This test computes the difference (D) between the continuous distribution function (CDF) and the empirical cumulative distribution function (ECDF). The null hypothesis is rejected if, at the 90% significance level, D is larger than the critical value statistic.

    ```
    EXAMPLE:

    sdg.MetalogGoodnessofFitType[colName] =
    GoodnessOfFitType.KOLMOGOROV_SMIRNOV
    ```

  - [Default] Anderson -Darling –Ranks the fitted distributions using the Anderson Darling statistic, A2 . The null hypothesis is rejected using a 90% significance level, if A2 is larger than the critical value statistic. This test awards more weight to the distribution tails then the Kolmogorov-Smirnoff test.

EXAMPLE:

```
sdg.MetalogGoodnessofFitType[colName] =
GoodnessOfFitType.ANDERSON_DARLING
```

- AIC – The AIC test is a Chi Squared test corrected for the number of distribution parameters and sample size. AIC = Chi-Square Statistic + 2 * k + 2 * k * (k + 1) / (n – k – 1) where k is the number of distribution parameters and n is the sample size.

  EXAMPLE:

  ```
  sdg.MetalogGoodnessofFitType[colName] = GoodnessOfFitType.AIC
  ```

- AICc –When the sample size is small, there is a significant chance that the AIC test will select a model with many parameters. In other words, AIC will overfit the data. AICc was developed to reduce the possibility of overfitting by applying a penalty to the number of parameters. Assuming that the model is univariate, is linear in the parameters and has normally distributed residuals, the formula for AICc is: AICc = AIC + $2k$ 2+$2k$ $n-k-1$ where n = sample size, k = # of parameters. As the sample size approaches infinity, the penalty on the number of parameters converges to 0 resulting in AICc converging to AIC.

  EXAMPLE:

  ```
  sdg.MetalogGoodnessofFitType[colName] = GoodnessOfFitType.AICc
  ```

- BIC – The Bayesian information criterion (BIC) is defined as: BIC = k ln(n) = 2 ln ($\hat{L}$) where $\hat{L}$ = the maximized value of the likelikhood function of the model M. $\hat{L} = p(x|\theta, M)$ where θ are the parameter values that maximize the likelihood function and x is the observed data. n = Sample size k = Number of parameters

  BICc – The BICc is the alternative version of BIC, corrected for the sample size BICc = BIC + 2 * p * (p + 1) / (n – p - 1)

  EXAMPLE:

  ```
  sdg.MetalogGoodnessofFitType[colName] = GoodnessOfFitType.BIC
  sdg.MetalogGoodnessofFitType[colName] = GoodnessOfFitType.BICC
  ```

- Maximum Likelihood (ML) – The (negated) raw value of the estimated maximum log likelihood utilized in tests described above.

  EXAMPLE:

  ```
  sdg.MetalogGoodnessofFitType[colName] =
  GoodnessOfFitType.Max_LIKELIHOOD
  ```

- **Set Upper and Lower Bounds for Metalog distribution**

  As mentioned above, XLMiner SDK supports unbounded, semi-bounded and bounded Metalog distributions. The API for setting up semi-bounded or bounded distributions is explained below.

  `UseMinMaxAsBounds = True` sets the lower/upper bounds as minimum/maximum of each variable. However, if a lower or upper bound is manually set at any point before .fit(), XLMiner SDK will give priority to the manually set bounds while keeping the minimum/maximum for those variables where the bounds were not set manually.

  To check the Metalog bounds for all columns at once use: Bounds.

  ```
  Example:  Console.Writeline(sdg.Bounds);
  ```

  - Manually: Set each upper and lower bound separately using:

    ```
    MetalogLowerBound[colname] = <lowerBound>
    MetalogUpperBound[colname] = <upperBound>
    ```

Example: Following two lines of code sets the upper and lower bounds for the CRIM variable in the Synthetic Data Generator object.

```
sdg.MetalogLowerBound["CRIM"] = 0.00632
sdg.MetalogUpperBound["CRIM"] = 88.9762
```

- Automatically: Set UseMinMaxAsBounds to True to use the minimum and maximum values in each column as the lower and upper bound for each variable.

  ```
  UseMinMaxAsBounds = True or False
  Example:  sdg.UseMinMaxAsBounds = true
  ```

  If a bound has been set manually, UseMinMaxAsBounds will not overwrite the existing bound.

- **Generate Metalog Curves**

  Set computeMetalogCurves to true to compute Metalog PDF curves the selected Metalog distribution for all columns.

  ```
  Example: sdg.computeMetalogCurves = true
  ```

Correlation Fitting Options

- **Correlation Method**

  Use `CorrelationMethod` to fit a correlation between the variables. If this option is set to CorrelationType.None, then no correlation fitting will be performed.

  CorrelationMethod = CorrelationType.None

  Otherwise, there are two options for correlation fitting: rank (`CorrelationType.RANK`) and copula (`CorrelationType.COPULA`).

  - If *Rank* is selected, XLMiner SDK will use the Spearman rank order correlation to fit a correlation matrix for all columns. To select Rank use: `CorrelationType.RANK.`
  - If *Copula* is selected, XLMiner SDK will attempt to fit correlation using specified copulas. To select a Copula use: `CorrelationType.COPULA.`
    - If `CorrelationMethod = CorrelationType.COPULA`, the copulas may be specified using `Copula[<copulaType>]`. If multiple copulas are selected, the first successfully fit copula will be used in the sample generation.

      XLMiner SDK offers 5 types of copulas:

      - CopulaType.STUDENT
      - CopulaType.CLAYTON
      - CopulaType.FRANK
      - CopulaType.GUMBEL
      - CopulaType.GAUSS

      Example: The priority of the copulas given in the example code below is 1. student, 2. clayton, 3. gumbel and 4. gauss (order as listed).

      ```
      sdg.Copula[CopulaType.STUDENT] = true
      sdg.Copula[CopulaType.CLAYTON] = true
      sdg.Copula[CopulaType.FRANK] = false
      ```

```
sdg.Copula[CopulaType.GUMBEL] = true
sdg.Copula[CopulaType.GAUSS] = true
```

ⓘ

The Spearman rank order correlation coefficient is a *nonparametric* measure of correlation that is computed from a rank ordering of the trial values drawn for all variables. It can be used to induce correlations between *any* two uncertain variables, whether they have the same or different analytic distributions, or even custom distributions. This correlation coefficient ranges in value from -1 to

ⓘ

XLMiner SDK includes copulas to improve the method of defining the correlation or dependence between two or more variables. Copulas offer more flexibility over the rank order correlation method, and are able to capture complex correlations.

An n – dimensional copula C is a multi-variate probability distribution where the marginal probability distribution of each variable follows the Uniform(0,1) distribution. A major benefit of copulas is that they allow two or more uncertain variables to be correlated without changing the shape of the original uncertain variable distributions.

For some copula C, a multi-variate distribution F with distributions of F_1, F_2, … F_n can be written as:

$F(x_1, …, x_n) = C(F_1 (x_1), F_2 (x_2), …, F_n (x_n))$

XLMiner SDK supports five types of copulas: three Archimedean copulas (clayton, frank, and gumbel) and two elliptical copulas (Gauss and Student).

Generating Data Options

- Sample Size

  `SampleSize = N,` where N is any positive integer

  Use this opton to set the size of the generated sample. The default is 100.

  `Example:  SampleSize = 10`

- Random Seed

  `RandomSeed = N,` where N is any positive integer

  Setting the random number seed to a nonzero value (any number of your choice is OK) ensures that the *same* sequence of random numbers is used for each simulation. When the seed is zero or RandomSeed is not specified, the random number generator is initialized from the system clock, so the sequence of random numbers will be different in each simulation. Set the seed to ensure that the results from one simulation to another are strictly comparable.

  `Example:  RandomSeed = 12345`

- Random Generator

  `RandomGenerator = RandomGeneratorType.<Type>`

  `Type = HDR, LECUYER_CMRG, MERSENNE_TWISTER, PARK_MILLER, WELL`

  Use this option to select a random number generation algorithm. XLMiner SDK includes an advanced set of random number generation capabilities.

  Computer-generated numbers are never truly "random," since they are always computed by an algorithm – they are called *pseudorandom* numbers. A random number generator is designed to quickly generate sequences of numbers that are as close to being statistically independent as possible. Eventually, an algorithm will generate the same numbers seen sometime earlier in the sequence, and at this point the sequence will begin to repeat. The *period* of the random number generator is the number of values it can generate before repeating.

  A long period is desirable, but there is a tradeoff between the length of the period and the degree of statistical independence achieved within the period. Hence, XLMiner SDK offers a choice of five random number generators:

  - *Park-Miller* (`PARK_MILLER`)"Minimal" Generator with Bayes-Durham shuffle and safeguards. This generator has a period of 2^{31}-2. Its properties are good, but the following choices are usually better.

    `EXAMPLE:  sdg.RandomGenerator = RandomGeneratorType.MERSENNE_TWISTER`

  - Combined Multiple Recursive Generator of L'Ecuyer (`LECUYER_CMRG`). This generator has a period of 2^{191}, and excellent statistical independence of samples within the period.

    `EXAMPLE:  sdg.RandomGenerator = RandomGeneratorType.LECUYER_CMRG`

  - Well Equidistributed Long-period Linear (`WELL`) generator of Panneton, L'Ecuyer and Matsumoto. This generator combines a long period of 2^{1024} with very good statistical independence.

    `EXAMPLE:  sdg.RandomGenerator = RandomGeneratorType.WELL`

  - *Mersenne Twister* (default setting - `MERSENNE_TWISTER`) generator of Matsumoto and Nishimura. This generator has the longest period of 2^{19937}-1, but the samples are not as "equidistributed" as for the WELL and L-Ecuyer-CMRG generators.

    `EXAMPLE:  sdg.RandomGenerator = RandomGeneratorType.MERSENNE_TWISTER`

  - The *HDR* Random Number Generator (`HDR`), designed by Doug Hubbard. Permits data generation running on various computer platforms to generate identical or independent streams of random numbers.

    `EXAMPLE:  sdg.RandomGenerator = RandomGeneratorType.HDR`

- Sampling Method

  `SamplingMethod = SamplingMethodType.<type>`

  `Type = MONTE_CARLO, LATIN_HYPERCUBE, SOBOL_RQMC`

  Use this option to select *Monte Carlo, Latin Hypercube,* or *Sobol RQMC* sampling.

  - *Monte Carlo:* In standard Monte Carlo sampling, numbers generated by the chosen random number generator are used directly to obtain sample values. With this method, the variance or estimation error in computed samples is inversely proportional to the square root of the number of trials (controlled by the Sample Size); hence to cut the error in half, four times as many trials are required.

    `EXAMPLE:  SamplingMethod = SamplingMethod.MONTE_CARLO`

  XLMiner SDK provides two other sampling methods than can significantly improve the 'coverage' of the sample space, and thus reduce the variance in computed samples. This means that you can achieve a given level of accuracy (low variance or error) with fewer trials.

- *Latin Hypercube (default):* Latin Hypercube sampling begins with a stratified sample in each dimension (one for each selected variable), which constrains the random numbers drawn to lie in a set of subintervals from 0 to 1. Then these one-dimensional samples are combined and randomly permuted so that they 'cover' a unit hypercube in a stratified manner.

  ```
  EXAMPLE:  sdg.SamplingMethod = SamplingMethod.LATIN_HYPERCUBE
  ```

- *Sobol RQMC (Randomized QMC)***.** Sobol numbers are an example of so-called "Quasi Monte Carlo" or "low-discrepancy numbers," which are generated with a goal of coverage of the sample space rather than "randomness" and statistical independence. XLMiner SDK adds a "random shift" to Sobol numbers, which improves their statistical independence.

  ```
  EXAMPLE:  sdg.SamplingMethod = SamplingMethod.SOBOL_RQMC
  ```

- Random Stream

  ```
  RandomStream = RandomStreamType.<type>
  ```

  ```
  Type = INDEPENDENT or SINGLE
  ```

  Use this option to select a *Single Stream* or an *Independent Stream (the default)* for each variable.

  ```
  EXAMPLE:  sdg.RandomStream = sgd.RandomStreamType.SINGLE
  ```

  ```
  EXAMPLE:  sdg.RandomStream = sgd.RandomStreamType.INDEPENDENT
  ```

  If *Single Stream* is selected, a single sequence of random numbers is generated. Values are taken consecutively from this sequence to obtain samples for each selected variable. This introduces a subtle dependence between the samples for all distributions in one trial. In many applications, the effect is too small to make a difference – but in some cases, better results are obtained if *independent* random number sequences (streams) are used for each distribution in the model. XLMiner SDK offers this capability for Monte Carlo sampling and Latin Hypercube sampling; it does not apply to Sobol numbers.

Synthetic Data Generation Results

The following result options allow users to obtain the results of the Synthetic Data Generation.

Metalog Fitting Results

- MetalogFittedAll or MetalogFitted[colname] – Checks whether there was at least one feasible Metalog distribution fitted for all or specific columns.

  Example:

  ```
  Console.WriteLine("Metalog fitted ?") + (sdg.MetalogFittedAll? "Yes" : "No" ));
  ```

  Example:

  ```
  Console.WriteLine("Metalog fitted for RM ?") + (sdg.MetalogFitted["RM"]? "Yes" :
  "No" ));
  ```

- MetalogCoefficientsAll or MetalogCoefficients[colname] – Obtains the coefficients for fitted Metalog Distributions for all or specific columns.

  Example:

  ```
  Console.WriteLine(sdg.MegalogCoefficientsAll);
  ```

  Example:

  ```
  Console.WrtieLine(sdg.MegalogCoefficients["RM"]);
  ```

  Example Results:

  ```
  Metalog Coefficients: RM (2 x 3)
  ```

```
        double          double          double
        C1              C2              C3
2       0.11052         0.49587         NaN
3       0.020645        0.49587         0.18
```

- BestMetalog – Use BestMetalog to obtain the number of terms for the best fitted Metalog distribution for each column.

  Example:

  ```
  Console.WriteLine(sdg.BetMetalogDF)
  ```

  Example Results:

  ```
  Best Metalog Fit (12 x 1)
                  wstring
                  Terms
  CRIM            5
  ZN              2
  INDUS           2
  NOX             2
  RM              3
  AGE             4
  DIS             2
  TAX             3
  PTRATIO         3
  B               3
  LSTAT           2
  MEDV            2
  ```

- MetalogGoodnessOfFitAll or MetalogGoodnessOfFit[colname] – Gets the detailed report of Goodness of Fit tests for fitted Metalog Distributions for all or specific columns.

  Example for obtaining goodness of fit test results for all fitted Metalog distributions:

  ```
  Console.WriteLine(sdg.MetalogGoodnessOfFitAll);
  ```

  Example for obtaining goodness of fit test results for just the RM fitted Metalog distribution:

  ```
  Console.WriteLine(sdg.MetalogGoodnessOfFitAll[“RM”]);
  ```

  Example Results:

  ```
  Metalog Goodness of Fit: RM (2 x 8)
  Double  double  double  double  double  double  double  double
  CS      KS      AD      ML      AIC     AICc    BIC     BICc
  2     204.83  0.14535 25.937  589.4   1186.8  1186.9  1203.7  1203.8
  3     199.53  0.16422 25.869  583.56  1177.1  1177.2  1198.3  1198.4
  ```

Correlation Results

- CorrelationFitted – Checks if the correlations have been fitted.

```
Console.WriteLine("Correlation fitted ?") +
(sdg.CorrelationFitted? "Yes" : "No" ));
```

Obtains information on the correlation fitting.

- SelectedCopula – Determines what copula was selected if copua fitting was requested.
- CorrelationsSigma – Obtains the correlation matrix , when applicable – i.e. if rank correlation or Gauss/Student Copula was used.
- CorrelationTheta – Use this result to obtain the fitted correlation theta value when selected copula is Clayton, Frank or Grumbel.
- CopulaDegressOfFreedom – Use this result to obtain the degrees of freedom when copula is Student.
- Example code above first checks if the correlations have been fitted, and then, depending on the correlation type, (rank or copula), reports applicable results.

Advanced Statistics, Percentiles, Six Sigma Metrics

To obtain basic or advanced statistics, percentiles, six sigma metrics based on the generated synthetic data, use the Summarizer object as shown in the example lines of code below.

```
Console.WriteLine(Summarizer.summary(data));  //statistics
Console.WriteLine(Summarizer.summary(sample));

Console.WriteLine(Summarizer.advancedSummary(data));  //advanced statistics
Console.WriteLine(Summarizer.advancedSummary(sample));

Console.WriteLine(Summarizer.sixSigma(data)); //six sigma metrics
Console.WriteLine(Summarizer.sixSigma(sample));

Console.WriteLine(Summarizer.percentiles(data)); //percentiles
Console.WriteLine(Summarizer.percentiles(sample));
```

Description: Computed Statistics, Percentiles and Six Sigma Metrics

All statistics generated using **Summarizer.summary()** and **Summarizer.advancedSummary()** are briefly described below.

Statistics

- **Mean**, the average of all the values.
- **Standard Deviation**, the square root of variance.
- **Variance,** the spread of the distribution of values.
- **Skewness**, which describes the *asymmetry* of the distribution of values.
- **Kurtosis**, which describes the *peakedness* of the distribution of values.
- **Mode**, the most frequently occurring single value.
- **Minimum**, the minimum value attained.
- **Maximum**, the maximum value attained.
- **Range**, the difference between the maximum and minimum values.

Advanced Statistics

- **Mean Abs. Deviation**, returns the average of the absolute deviations.
- **SemiVariance**, measure of the dispersion of values.
- **SemiDeviation**, *one-sided* measure of dispersion of values.
- **Value at Risk 95%**, the maximum loss that can occur at a given confidence level.
- **Cond. Value at Risk**, is defined as the *expected value* of a loss *given that* a loss at the specified percentile occurs.
- **Mean Confidence**, returns the confidence "half-interval" for the estimated mean value (returned by the PsiMean() function.
- **Std. Dev. Confidence 95%**, returns the confidence 'half-interval' for the estimated standard deviation of the simulation trials (returned by the PsiStdDev() function).
- **Coefficient of Variation**, is defined as the ratio of the standard deviation to the mean.
- **Standard Error**, defined as the standard deviation of the sample mean.
- **Expected Loss**, returns the average of all negative data multiplied by the percentrank of 0 among all data.
- **Expected Loss Ratio**, returns the expected loss ratio.
- **Expected Gain** returns the average of all positive data multiplied by 1 - percentrank of 0 among all data.
- **Expected Gain Ratio**, returns the expected gain ratio**.**
- **Expected Value Margin**, returns the expected value margin.

Percentiles

- **Summarizer.percentiles()** generates numeric percentile values (from 1% to 99%) computed using all values for the variable. For example, the 75th Percentile value is a number such that three-quarters of the values occurring in the last simulation are less than or equal to this value.

Six Sigma Metrics

Summarizer.sixsigma() generates various computed Six Sigma measures, described below. These functions compute values related to the Six Sigma indices used in manufacturing and process control.

- **SigmaCP**: A Six Sigma index, SigmaCP predicts what the process is capable of producing if the process mean is centered between the lower and upper limits. This index assumes the process output is normally distributed.

  $$Cp = \frac{UpperSpecificationLimit - LowerSpecificationLimit}{6\hat{\sigma}}$$

  where $\hat{\sigma}$ is the estimated standard deviation of the process.
- **SigmaCPK:** A Six Sigma index, SigmaCPK predicts what the process is capable of producing if the process mean is not centered between the lower and upper limits. This index assumes the process output is normally distributed and will be negative if the process mean falls outside of the lower and upper specification limits.

  $$Cpk = \frac{MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{3\hat{\sigma}}$$

  where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
- **SigmaCPKLower:** A Six Sigma index, SigmaCPKLower calculates the one-sided Process Capability Index based on the lower specification limit. This index assumes the process output is normally distributed.

  $$Cp, lower = \frac{\hat{\mu} - LowerSpecificationLimit}{3\hat{\sigma}}$$

where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaCPKUpper:** A Six Sigma index, SigmaCPKUpper calculates the one-sided Process Capability Index based on the upper specification limit. This index assumes the process output is normally distributed.

  $$Cp, upper = \frac{UpperSpecificationLimit - \hat{\mu}}{3\hat{\sigma}}$$

  where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaCPM:** A Six Sigma index, SigmaCPM calculates the capability of the process around a target value. This index is referred to as the Taguchi Capability Index. This index assumes the process output is normally distributed and is always positive.

  $$Cpm = \frac{\hat{C}p}{\sqrt{1+(\frac{\hat{\mu}-T}{\hat{\sigma}})^2}}$$

  where $\hat{C}p$ is the process capability (SigmaCP), $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and T is the target process mean.

- **SigmaCPM:** A Six Sigma index, SigmaDefectPPM calculates the Defective Parts per Million.

  $$DPMO = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}}) + 1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}})) * 1000000$$

  where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaDefectShiftPPM:** A Six Sigma index, SigmaDefectShiftPPM calculates the Defective Parts per Million with an added shift.

  $$DPMOShift = (\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) +$$
  $$1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)) * 1000000$$

  where$\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaDefectShiftPPMLower:** A Six Sigma index, SigmaDefectShiftPPMLower calculates the Defective Parts per Million, with a shift, below the lower specification limit.

  $$DPMOshift, lower = (\delta^{-1}(\frac{LowerSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$$

  where$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaDefectShiftPPMUpper:** A Six Sigma index,SigmaDefectShiftPPMUpper calculates the Defective Parts per Million, with a shift, above the lower specification limit.

  $$DPMOshift, upper = (\delta^{-1}(\frac{UpperSpecificationLimit}{\hat{\sigma}} - Shift) * 1000000$$

  where$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaK:** A Six Sigma index, SigmaK calculates the Measure of Process Center and is defined as:

  $$1 - \frac{2*MIN(UpperSpecificationLimit - \hat{\mu}, \hat{\mu} - LowerSpecificationLimit)}{UpperSpecificationLimit - LowerSpecificationLimit}$$

  where$\hat{\mu}$ is the process mean.

- **SigmaLowerBound:** A Six Sigma index, SigmaLowerBound calculates the Lower Bound as a specific number of standard deviations below the mean and is defined as:

$\hat{\mu} - \hat{\sigma} * \#StandardDeviations$

where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaProbDefectShift:** A Six Sigma index, SigmaProbDefectShift calculates the Probability of Defect, with a shift, outside of the upper and lower limits. This statistic is defined as:

  $$\delta^{-1}\left(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift\right) +$$
  $$1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$$

  where $\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaProbDefectShiftLower:** A Six Sigma index, SigmaProbDefectShiftLower calculates the Probability of Defect, with a shift, outside of the lower limit. This statistic is defined as:

  $$\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$$

  where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaProbDefectShiftUpper:** A Six Sigma index, SigmaProbDefectShiftUpper calculates the Probability of Defect, with a shift, outside of the upper limit. This statistic is defined as:

  $$1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$$

  where$\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaSigmaLevel:** A Six Sigma index, SigmaSigmaLevel calculates the Process Sigma Level with a shift. This statistic is defined as:

  $$-\delta(\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) +$$
  $$1 - \delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift))$$

  where $\hat{\mu}$ is the process mean ,$\hat{\sigma}$ is the standard deviation of the process $\delta$is the standard normal cumulative distribution function, and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaUpperBound:** A Six Sigma index, SigmaUpperBound calculates the Upper Bound as a specific number of standard deviations above the mean and is defined as:

  $\hat{\mu} - \hat{\sigma} * \#StandardDeviations$

  where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaYield:** A Six Sigma index, SigmaYield calculates the Six Sigma Yield with a shift, or the fraction of the process that is free of defects. This statistic is defined as:

  $$\delta^{-1}(\frac{UpperSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift) -$$
  $$\delta^{-1}(\frac{LowerSpecificationLimit - \hat{\mu}}{\hat{\sigma}} - Shift)$$

  where $\hat{\mu}$ is the process mean, $\hat{\sigma}$ is the standard deviation of the process and δ^{-1} is the standard normal inverse cumulative distribution function.

- **SigmaZLower:** A Six Sigma index, SigmaZLower calculates the number of standard deviations of the process that the lower limit is below the mean of the process. This statistic is defined as:

  $$\frac{\hat{\mu} - LowerSpecificationLimit}{\hat{\sigma}}$$

  where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

- **SigmaZMin:** A Six Sigma index, SigmaZMin calculates the minimum of SigmaZLower and SigmaZUpper. This statistic is defined as:

  $$\frac{MIN(\hat{\mu}-LowerSpecificationLimit, UpperSpecificationLimit-\hat{\mu})}{\hat{\sigma}}$$

  where$\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.
- **SigmaZUpper:** A Six Sigma index, SigmaZUpper calculates the number of standard deviations of the process that the upper limit is above the mean of the process. This statistic is defined as:

  $$\frac{UpperSpecificationLimit-\hat{\mu}}{\hat{\sigma}}$$

  Where $\hat{\mu}$ is the process mean and $\hat{\sigma}$ is the standard deviation of the process.

Multiple Linear Regression with Simulation Example

The example below illustrates how to perform the Risk Analysis of fitted Multiple Regression model using synthetic data generation techniques. The methods demonstrated below are applicable to any supervised classification or regression method.

See the Synthetic Data Generation example above for explanations on Step 1, Step 2 and Step 3.

Important Note

Steps 1, 2 and 3 may be condensed to a single line of code:

```
auto const sample = SyntheticDataGenerator().fitTransform(data);
```

However, this step should be taken with extreme caution as most users should/want to set at least a few critical options such as upper and lower bounds, sample size, number of terms in the Metalog functions and/or selecting whether rank correlation or copulas as utilize.

If a user opts to forgo setting options for the Synthetic Data Generator then the data will be fit to unbounded Metalog functions with up to 5 terms, a sample size of 100 and rank correlation.

```
public static int LinearRegressionSimulation()
{
// example of simulating response prediction
// Linear Regression estimator/model params/evaluations are minimal
// for complete Linear Regression example, see Regression::LinearRegression()

try
{
  using Dataset data = new Dataset(Reader.textFile(PATH + "bh-scale-
  reg.txt"), "MEDV", new double[] { 1.0, 0.0 });

  using var model = new
  LinearRegression.Estimator().fit(data.input[TRAINING],
  data.target[TRAINING]);
  // setup synthetic data generation
  using var sdg = new SyntheticDataGenerator();

// STEP 1: setup the marginal Metalog distribution fitting for all data
columns
```

```
  {
    // auto/manual Metalog fit
    sdg.MetalogAuto = true;

    // OPTION 1: manual Metalog with the specified # terms
    if (false == sdg.MetalogAuto)
    {
      // specify fixed # Metalog terms for each column (in this example same
      //for all columns)
      for (var i = 0; i < data.input[TRAINING].NumCols; i++)

      sdg.NumMetalogTerms[data.input[TRAINING].ColName[i]] = 2;
    }
    // OPTION 2: auto search for the best Metalog up to the specified # terms
    else
    {
      for (var i = 0; i < data.input[TRAINING].NumCols; i++)
      {
        // specify the max # terms for the search for each column. max range
        //is [2,16] (in this example same for all columns)
        sdg.NumMetalogTerms[data.input[TRAINING].ColName[i]] = 5;

        // choose the GOF metric to select the best Metalog for each column
        //(in this example same for all columns)
        // CS, KS, AD, AIC, BIC, AICc, BICc, ML
        sdg.MetalogGoodnessOfFitType[data.input[TRAINING].ColName[i]] =
        SDG.GoodnessOfFitType.AIC;
      }
    };

    // optionally, set the lower/upper bound for some or all columns that
    // will be used in Metalog fitting
    //sdg.MetalogLowerBound["CRIM"] = 0.00632; sdg.MetalogUpperBound["CRIM"] =
      88.9762;
   //sdg.MetalogLowerBound["ZN"] = 0; sdg.MetalogUpperBound["ZN"] = 100;
   //sdg.MetalogLowerBound["INDUS"] = 0.46; sdg.MetalogUpperBound["INDUS"] = 27.74;
   //sdg.MetalogLowerBound["NOX"] = 0.385; sdg.MetalogUpperBound["NOX"] = 0.871;
   //sdg.MetalogLowerBound["RM"] = 3.561; sdg.MetalogUpperBound["RM"] = 8.78;
   //sdg.MetalogLowerBound["DIS"] = 1.1296; sdg.MetalogUpperBound["DIS"] = 12.1265;
   //sdg.MetalogLowerBound["AGE"] = 2.9; sdg.MetalogUpperBound["AGE"] = 100;
   //sdg.MetalogLowerBound["TAX"] = 187; sdg.MetalogUpperBound["TAX"] = 711;
   //sdg.MetalogLowerBound["PTRATIO"] = 12.6; sdg.MetalogUpperBound["PTRATIO"] = 22;
   //sdg.MetalogLowerBound["B"] = 0.32; sdg.MetalogUpperBound["B"] = 396.9;
   //sdg.MetalogLowerBound["LSTAT"] = 1.73; sdg.MetalogUpperBound["LSTAT"] = 37.97;
   //sdg.MetalogLowerBound["MEDV"] = 5; sdg.MetalogUpperBound["MEDV"] = 50;

    // or, use min/max of the columns data as bounds
    // note that if manual LowerBound/UpperBound was set before, it's not
    //overwritten
    sdg.UseMinMaxAsBounds = true;

    // at this point, can overwrite specific bounds as well
  };

// STEP 2: setup the correlation fitting for all data columns
  {
    // OPTION 1: no correlation, independent sample
    //sdg.CorrelationMethod = SDG.CorrelationType.NONE;
    // OPTION 2: Rank correlation (default)
    sdg.CorrelationMethod = SDG.CorrelationType.RANK;

    // OPTION 3: select copulas
    {
      //sdg.CorrelationMethod = SDG.CorrelationType.COPULA;

        // order sets the priority of copulas
        // if multiple selected, first successfully fit copula will be
        //selected and used for sample generation
        //sdg.Copula[SDG.CopulaType.STUDENT] = true;
        //sdg.Copula[SDG.CopulaType.CLAYTON] = true;
        //sdg.Copula[SDG.CopulaType.FRANK] = false;
        //sdg.Copula[SDG.CopulaType.GUMBEL] = true;
        //sdg.Copula[SDG.CopulaType.GAUSS] = true;
    };
  };
// STEP 3: generate data
  {
    // optional, not setting at all or setting to 0 means no seed
    sdg.RandomSeed = 12345;

    // below also optional, advanced
    sdg.RandomGenerator = SDG.RandomGeneratorType.MERSENNE_TWISTER;
    sdg.SamplingMethod = SDG.SamplingMethodType.LATIN_HYPERCUBE;
    sdg.RandomStream = SDG.RandomStreamType.INDEPENDENT;

    // # trials to simulate
    sdg.SampleSize = 100;
  };
```

Fits the Metalog distributions and correlations for each variable in the training partition.

```
  // fit distributions/correlations
  sdg.fit(data.input[TRAINING]);
```

Generates the synthetic data using the training partition as input

```
  // generate sample
  using var sample = sdg.transform(data.input[TRAINING]);

  // print the messages log
  Console.WriteLine(sdg.LogDF);
```

Scores the training data using the MLR fitted model.

```
  // predict actual data using fitted MLR model
  using var actualPrediction = model.predict(data.input[TRAINING]);

  // predict synthetic data using fitted MLR model
  using var syntheticPrediction = model.predict(sample).setColName(0,
  "MEDV");
```

Scores the synthetic data using the MLR fitted model.

```
  // combine synthetic sample and prediction into a single "table" report
  var syntheticTable = Util.colBind(new DataFrame[] { syntheticPrediction,
  sample });
```

Optional: XLMiner SDK can filter results by calculating any valid Excel expression that references the variables or the output variable.

```
  // calculate expression
  // can use any valid Excel expression that references variables and/or
  // response
  using var syntheticExpression =
  SyntheticDataGenerator.calculateExpression(syntheticTable, "IF(CRIM<10,
  MEDV, 2*MEDV)");
```

Add expression results to synthetic table results.

```
  syntheticTable = Util.colBind(new DataFrame[] { syntheticExpression,
  syntheticTable });
  Console.WriteLine(Util.head(syntheticTable));

  // get basic/advanced statistics, percentiles, six sigma on
  //original/synthetic data prediction using Summarizer object
  Console.WriteLine(Summarizer.summary(data.target[TRAINING]));
  Console.WriteLine(Summarizer.summary(actualPrediction));
  Console.WriteLine(Summarizer.summary(syntheticPrediction));
  Console.WriteLine(Summarizer.summary(syntheticExpression));
```

Summary statistics for both training partition and synthetic data.

Advanced stats for both training partition and synthetic data.

```
  Console.WriteLine(Summarizer.advancedSummary(data.target[TRAINING]));
  Console.WriteLine(Summarizer.advancedSummary(actualPrediction));
  Console.WriteLine(Summarizer.advancedSummary(syntheticPrediction));
  Console.WriteLine(Summarizer.advancedSummary(syntheticExpression));

  Console.WriteLine(Summarizer.sixSigma(data.target[TRAINING]));
  Console.WriteLine(Summarizer.sixSigma(actualPrediction));
  Console.WriteLine(Summarizer.sixSigma(syntheticPrediction));
  Console.WriteLine(Summarizer.sixSigma(syntheticExpression));
```

Six Sigma metricsfor both training partition and synthetic data.

```
  Console.WriteLine(Summarizer.percentiles(data.target[TRAINING]));
  Console.WriteLine(Summarizer.percentiles(actualPrediction));
  Console.WriteLine(Summarizer.percentiles(syntheticPrediction));
  Console.WriteLine(Summarizer.percentiles(syntheticExpression));
}
catch (XLMiner.Exception ex)
{
  Console.WriteLine("Exception has occurred at
  <Simulation.LinearRegressionSimulation>:\n" + ex.Message);
  return 1;
}
return 0;
}}
```

Percentiles for both training partition and synthetic data.

Catch and report an errors.

Data Transformation

Introduction

This chapter focuses on XLMiner SDK's tools for data transformation. Data Transformation (in this context) is a fundamental step when building a data mining model that involves determining if the dataset is in a reasonable condition. It involves cleaning, preprocessing and reducing data. With these tools you'll be able to normalize your data, manage missing data, reduce the number of categories in a categorical variable, gain the knowledge to know when a variable is redundant, or not, etc. XLMiner SDK includes the following data transformation utilities.

- Scaling

  Use this utility to rescale one or more features in your data.

- Factorizing

  Use this utility to encode the values of categorical features into factors, i.e. ordinal integers.

- Binning

  Binning a dataset is a process of grouping continuous data into categorical groups based on the counts or the intervals.

- One-Hot-Encoding

  Transforms categorical data into a numeric format appropriate for use with some Data Mining algorithms that require numeric features.

- Reducing categories

  This utility allows users to create a new categorical variable that reduces the total number of categories. You can reduce the number of categories "by frequency" or "manually".

- PCA

  Principal Component Analysis (PCA) is a mathematical procedure that transforms a number of (possibly) correlated variables into a smaller number of uncorrelated variables called principal components. The objective of PCA is to reduce the dimensionality (number of variables) of the dataset but retain as much of the original variability in the data as possible.

- K-Means and Hierarchical Clustering

  Cluster Analysis, also called data segmentation, has a variety of goals which all relate to grouping or segmenting a collection of objects (also called observations, individuals, cases, or data rows) into subsets or "clusters". These "clusters" are grouped in such a way that the observations included in each cluster are more closely related to one another than objects assigned to different clusters.

- Sampling

  Sampling techniques choose a representative subset of data in order to identify patterns or trends in a complete dataset.

- Partitioning

  One very important issue when fitting a model is how well the newly created model will behave when applied to new data. To address this issue, the dataset can be divided into multiple partitions: a training partition used to create the model, a validation partition to test the performance of the model and, if desired, a third test partition. Partitioning is performed randomly, to protect against a biased partition, according to proportions specified by the user or according to rules concerning the dataset type.

- Canonical Variate Analysis (CVA)

  XLMiner SDK produces the canonical variates for a dataset based on an orthogonal representation of the original variates. This has the effect of choosing a representation which maximizes the distance between classes.

Scaling: Simple but Critical

How does an ACT score of 30 compare with an SAT score of 670? Without further information, it's impossible to say since the scores have different scales. In data mining it is common to rescale raw data so that vectors representing different data all share the same scale. Two commonly used methods are *standardization* and *normalization*. Standardization transforms data into *z-scores*, or data with a mean of zero and a standard deviation of one. Normalization, on the other hand, rescales the data to fit into the finite range from 0.0 to 1.0.

One point of possible confusion is that, in this context, the term "normalization" is not related to a "normal" distribution but rather to what mathematicians call the "norm" of a vector. Vector norms can be defined in several ways; two of the most common are the sum of the absolute values of the vector elements and the square root of sum of the squares of the vector elements. These two norm definitions are referred to as the L1 and L2 norms, respectively. The default L2 norm represents the familiar Euclidean distance.

In the following example code, we standardize the column vectors in the Hald dataset. A more detailed example may be found in the Examples folder in your Frontline installation folder.

Using the Rescaling feature to standardize column vectors

```
public static int Rescaling()
 {
   try
     {
       DataFrame data = Reader.textFile((PATH + "hald-small.txt"));
       Console.WriteLine(data);
       var estimator = new Scaler.Estimator()
       {
       Technique = Scaler.Technique.STANDARDIZATION;
       };
       Console.WriteLine(estimator);
       var model = estimator.fit(data);
       Console.WriteLine(model);

       var rescaledData = model.transform(data);
       Console.WriteLine(rescaledData);
       var pmmlFile = PATH + "PMML/transformation-rescaling.xml";
       model.toPMML(pmmlFile);

       Console.WriteLine(estimator.
            setTechnique(Scaler.Technique.NORMALIZATION).
            setCorrection(0.01).
```

```
              fit(data).
              transform(data));
          Console.WriteLine(estimator.
              setTechnique(Scaler.Technique.ADJUSTED_NORMALIZATION).
                 setCorrection(0.01).
                 fit(data).
                 transform(data));
          Console.WriteLine(estimator.
                 setTechnique(Scaler.Technique.UNIT_NORMALIZATION).
                 setNormType(Scaler.NormType.L2).
                 fit(data).
                 transform(data));
         }


      catch (XLMiner.Exception ex)
       {
        Console.WriteLine("Exception has occurred at
        <Transformation.Rescaling>:\n" + ex.Message);
         return 1;
        }
     return 0;
   }
```

Scaling Options

XLMiner SDK provides the following methods for feature scaling: Standardization (*Technique = Scaler.Technique.STANDARDIZATION*), Normalization(*Scaler.Technique.NORMALIZATION*), Adjusted Normalization(*Scaler.Technique.ADJUSTED_NORMALIZATION*) and Unit Norm(*Scaler.Technique.UNIT_NORMALIZATION*).

- *Scaler.Technique.STANDARDIZATION*

  Standardization adjusts the feature values to zero mean and unit variance. (x−mean)/std.dev.

- *Scaler.Technique.NORMALIZATION*

  Normalization scales the data values to the [0,1] range. (x−min)/(max−min)

  The Correction option specifies a small positive number ε that is applied as a correction to the formula. The corrected formula is widely used in Neural Networks when Logistic Sigmoid function is used to activate the neurons in hidden layers – it ensures that the data values never reach the asymptotic limits of the activation function. The corrected formula is [x−(min−ε)]/[(max+ε)−(min−ε)].

- *Scaler.Technique.ADJUSTED_NORMALIZATION*

  Adjusted Normalization scales the data values to the [-1,1] range. [2(x−min)/(max−min)]−1

  The Correction option specifies a small positive number ε that is applied as a correction to the formula. The corrected formula is widely used in Neural Networks when Hyperbolic Tangent function is used to activate the neurons in hidden layers – it ensures that the data values never reach the asymptotic limits of the activation function. The corrected formula is {2[(x−(min−ε))/((max+ε)−(min−ε))]}−1.

- *Scaler.Technique.UNIT_NORMALIZATION*

  Unit Normalization is another frequently used method to scale the data such that the feature vector has a unit length. This usually means dividing each value by the Euclidean length (L2-norm) of the vector. In some applications, it can be more practical to use the Manhattan Distance (L1-norm).

  The default norm is L2.

Factorizing

Factorizing is most often applied to data values that "sit on the fence" as it were, between string and numeric data. Though it is not a requirement, factors are very often ordered. For example, the strings "poor", "fair", "good", and "excellent" have an implied order not apparent in the character data itself.

Categorical variables can be nominal or ordinal. Nominal variable values have no order, for example, True or False or Male or Female. Values for an ordinal variable have a clear order but no fixed unit of measurement, i.e. Kinder, First, Second, Third, Fourth, and Fifth or a Size Chart using S, M, and L. The strings, while not numerical, do have an implied order. Factorizing converts a string variable such as S, M or L, into a new numeric, categorical variable.

The example code below first illustrates how to create a Factor estimator, then fit a model using that estimator. Afterwards, the example illustrates how to create a model manually, i.e. without using *estimator.fit*.

When using *estimator.fit* to fit a factoring model, XLMiner SDK will find unique values for the selected variable, sorted in lexicographical order. Alternative, XLMiner SDK allows users to manually provide levels in any order desired, based on prior knowledge of the data.

Factorizing example code

```
public static int Factorizing()
 {
   try
     {//Fits a model using the Factor Estimator
      DataFrame data = Reader.textFile(PATH + "BostonHousingCat.txt");

      //Create Factor Estimator
      using (var estimator = new Factor.Estimator())
      {
        //sets the base indicies to 1 for CHAS and 5 for RAD
        estimator.BaseIndex["CHAS"] = 1;
        estimator.BaseIndex["RAD"] = 5;

        Console.WriteLine(estimator);

        //Fits the model
        using (var model = estimator.fit(data))
        {
           Console.WriteLine(model);
           //Transforms data into factors using the fitted model
           var factors = model.transform(data);
           Console.WriteLine(Util.head(factors));

           //Converts transformed factors back to original values (here
           for  illustration purposes only)
           var defactorized = model.defactor(factors);
           Console.WriteLine(Util.head(defactorized));

           //Counts the # of records assigned to each level (factor)
           var counts = model.count(data);
           Console.WriteLine(counts);

           Util.free(ref factors);
           Util.free(ref defactorized);
           Util.free(ref counts);
         }
      }

      //Example code below fits a model manually
      Console.WriteLine("Based on previous knowledge, we set levels as
      the order we prefer.");

      //Creates a model manually, without using estimator.fit()
      using (var model = new Factor.Model())
       {
         //sets the base indicies to 1 for CHAS and 5 for RAD
         model.BaseIndex["CHAS"] = 1;
          model.BaseIndex["RAD"] = 5;

          //Supplying levels manually
          model.Levels["CHAS"] = new string[] { "0", "1" };
          model.Levels["RAD"] = new string[] { "1", "2", "3", "4", "5",
          "6", "7", "8", "24" };
          Console.WriteLine(model);

          var factors = model.transform(data);
```

```
                Console.WriteLine(Util.head(factors));

                var defactorized = model.defactor(factors);
                Console.WriteLine(Util.head(defactorized));

                var counts = model.count(data);
                Console.WriteLine(counts);

                Util.free(ref factors);
                Util.free(ref defactorized);
                Util.free(ref counts);
                }
                   Util.free(ref data);
                }

           catch (XLMiner.Exception ex)
            {
                   Console.WriteLine("Exception has occurred at
                   <Transformation.Factorizing>:\n" + ex.Message);
                      return 1;
            }
           return 0;
    }
```

Binning

Binning, sometimes known as bucketing, is often used to reduce large numeric ranges into a manageable number of “bins” or “buckets”, similar to what is done in visual form in a histogram.

XLMiner supports the two most commonly techniques for binning: “Equal Interval” and “Equal Count”. As their names suggest, “Equal Interval” method creates bins of equal width, and “Equal Count” selects bin ranges so that each bin contains approximately the same number of data points.

The example code below uses the Binning Estimator to fit a model using the default parameters. Afterwards, the data is transformed again using modified values for parameters *BinValue* and *RankParam*.

Note: In general there can be two different sets of parameters: parameters for the estimator (i.e. for fitting the model) and parameters for the model (i.e. for scoring/transforming new data). In the latter case, the fitted model remains the same but the options may be altered for further actions performed on the new datasets.

Binning example code

```
      public static int Binning()
        {
          try
```

```
{
  DataFrame data = Reader.textFile(PATH + "Binning.txt");
  {
     //Creates an array, eParams, to hold binning parameters
     //for two variables in the dataset, x2 and x4
    EstimatorParams[] eParams = new EstimatorParams[]
    {
      new EstimatorParams("x2", 11,
            Binning::MethodType.EQUAL_INTERVAL,
            Binning::IntervalType.RIGHT_CLOSED),
      new EstimatorParams("x4", 4,
            Binning::MethodType.EQUAL_COUNT,
            Binning::IntervalType.CLOSED)
    };

    //Constructs new Binning Estimator
    var estimator = new Binning::Estimator();

    //for each variable apply the options specified for eParams
    //above
    foreach (var eParam in eParams)
    {
      estimator.NumBins[eParam.name] = eParam.size;
      estimator.Interval[eParam.name] = eParam.interval;
      estimator.Method[eParam.name] = eParam.method;
    }
    Console.WriteLine(estimator);

    //Fit the model
    var model = estimator.fit(data);
    Console.WriteLine(model);

    //Save the model using PMML format
    var pmmlFile = PATH + "PMML/transformation-binning.xml";
    model.toPMML(pmmlFile);

    //Group data into bins using the fitted model
    var transformed = model.transform(data);
```

```
                Console.WriteLine(transformed);

                //Specify a different set of parameters on the MODEL
                model.BinValueOption["x2"] = Binning::BinValueType.MID_VALUE;
                model.BinValueOption["x4"] = Binning::BinValueType.RANK;

                //Specify rank parameters for BinValueType.RANK
                var rank = new Binning::RankParam()
                {
                  start = 1.0,
                  step = 5.0
                };

                //Apply rank parameters to “x4”
                model.Rank["x4"] = rank;
                //Group data into bins using the fitted model
                transformed = model.transform(data);
                Console.WriteLine(transformed);

                //Create the frequency table
                var frequency = model.getFrequencyTable(data);
                Console.WriteLine(frequency);
               }
               catch (XLMiner.Exception ex)
               {
                  Console.WriteLine(“Exception has occurred at
                      <Transformation.Binning>:\n” + ex.Message);
                  return 1;
               }
               return 0;
            }
```

A variable can be binned in the following ways:

Equal Count (Binning::MethodType.EQUAL_COUNT)

When using this method, the data is binned in such a way that each bin contains the same number of records. When this option is selected, the options *RANK*, *MID_VALUE*, *MEAN* and *MEDIAN* are available.

Equal Interval (Binning::MethodType.EQUAL_INTERVAL)

Each interval is based on bin width. When this method is selected, the entire data range is divided into a specified number of bins such that all bins have equal width. The options *RANK* and *MID_VALUE* are available with this method.

There are four options for the *BinValueType* parameter: *RANK*, *MID_VALUE*, *MEAN* and *MEDIAN*.

Rank of the bin (Binning::BinValueType.RANK)

In this option each value in the variable is assigned a rank according to the start and step values as specified by the user. This option may be used with both *EQUAL_COUNT* and *EQUAL_INTERVAL*.

Using step option with *BinValueType.RANK*

```
var rank = new Binning::RankParam()

{

        Start = 1.0, Step = 5.0

};
```

"Middle" of the bin (Binning::BinValueType.MID_VALUE)

The "middle" value of the bin is calculated as the average of the lower and upper break points of the bin interval. This option may be used with both *EQUAL_COUNT* and *EQUAL_INTERVAL*.

Mean of the bin (Binning::BinValueType.MEAN)

Mean of the bin is calculated as an average value of all data points that belong to this bin.

Median of the bin (Binning::BinValueType.MEDIAN)

Median of the bin is calculated as the median of all data points that belong to this bin.

XLMiner SDK supports three Interval Types:

IntervalType.RIGHT_CLOSED - (1,3]

IntervalType.LEFT_CLOSED - [1,3)

IntervalType.CLOSED - [1,3]

Principal Component Analysis

Principal component analysis (PCA) is a mathematical procedure that transforms a number of (possibly) correlated variables into a smaller number of uncorrelated variables called principal components. The objective of principal component analysis is to reduce the dimensionality (number of variables) of the dataset but retain as much of the original variability in the data as possible. The first principal component accounts for most of the variability in the data, the second principal component accounts for most of the remaining variability, and so on.

In a mathematical sense, PCA determines as set of eigenvectors that provide an alternative coordinate system with which the data can be described. This new set of coordinates is more "natural" in terms of the statistics, though it may not seem more natural to us humans. Viewing data in this new coordinate system can provide new insights.

More important in terms of data transformation is the fact that some of these new coordinates will account for a much greater fraction of the observed variance than others. This permits us to gain a computational advantage. Rather than consider, say, the 42 dimensions of the original data set, we can work with the three dimensions defined by the three most important eigenvectors.

Principal components can be calculated directly using the covariance matrix, or the data can first be scaled and the calculation performed using the correlation matrix. The correct choice depends on the characteristics of the data likely to be important to the analysis, so neither method is intrinsically "better" than the other. However, users should be aware that the covariance matrix method is likely to exaggerate the importance of data values that are numerically larger. For example, in data about athletic performance, 500-meter race times would end up being weighted more heavily than 100-meter race times, for no reason other than the times are larger.

Principal Components Analysis example code

```
public static int PrincipalComponentsAnalysis()
  {
    try
        {
          DataFrame data = Reader.textFile(PATH + "Irisfacto.txt");

          //create PCA estimator using the correlation method type
          var estimator = new PCA::Estimator()
          {
            MatrixMethod = PCA.MatrixMethodType.CORRELATION
          };

          //prints the estimator
          Console.WriteLine(estimator);

          //Score new data using the fitted model
          //var scores = estimator.fitTransform(data);

          //Fit the model
          var model = estimator.fit(data);

          // option 1:
          model.NumPrincipalComponents = 2;

          // option 2:
          // setting VarianceCutoff will compute and set
          //NumPrincipalComponents accordingly.
          //model.VarianceCutoff = 0.98;

          //sends the model to the output including eigenvalues and
          //Principal Components
          Console.WriteLine(model);
          //output explained variance
          Console.WriteLine(model.PrincipalVariance);

          //Transform the data using the model
          var scores = model.transform(data);
          Console.WriteLine(Util.head(scores));
          //Generate Q statistics
          var qStatistic = model.qStatistic(data, scores);
          Console.WriteLine(Util.head(qStatistic));

          //Generate Hotteling’s T-Squared statistics
          var tSquaredStatistic = model.tSquaredStatistic(scores);
          Console.WriteLine(Util.head(tSquaredStatistic));

          //Free memory
          Util.free(ref data);
          Util.free(ref estimator);
          Util.free(ref model);
          Util.free(ref scores);
          Util.free(ref qStatistic);
          Util.free(ref tSquaredStatistic);
          }
          catch (XLMiner.Exception ex)
          {
             Console.WriteLine("Exception has occurred at
             <Transformation.PrincipalComponentsAnalysis>:\n" +
             ex.Message);
             return 1;
          }
             return 0;
        }
```

Pros: Principal Component Analysis can reduce the computational complexity of problems by projecting the problem into a data space with fewer dimensions. It achieves this by considering only those dimensions in a new virtual data space that contribute most to explaining the observed variance.

Cons: The virtual data space of the PCA can be very abstract and is likely to be difficult to relate the dimensions in the virtual space back to dimensions in the original data.

Classification Methods

Introduction

XLMiner SDK supports all facets of the Data Mining process, including data exploration, visualization, transformation, feature selection, text mining, time series forecasting, affinity analysis, and unsupervised and supervised learning. This chapter focuses on the supervised learning technique, classification. Classification methods are pervasive in the world of data mining. Indeed, many parts our awareness is devoted to observing and classifying the form and function of objects around us in the real world. Therefore, it should not be surprising that classification methods are wide and varied. For example, scientists may wish to classify video images of fish into species for automatic sorting, or the postal service might need to classify handwritten postal codes for automatic sorting, and yet these two problems may require quite different approaches.

A great many goals of data mining can be framed as classification questions. Can this loan application be classified as "high" or "low" risk? Will this student become a member of the "graduating" class? Such applications of classification illustrate that the data mining techniques themselves cannot be sorted into rigid and mutually exclusive categories. For example, we can obtain classification discriminants for fish of known species based on some algorithm; that's a classification task. The minute we apply our discriminants to a new fish, it's a classification method.

In this section we will take an overview of the classification algorithms provided by XLMiner and get a feel for the characteristics of our data which should be considered when selecting a particular data mining model.

See the Synthetic Data Generation chapter, that appears earlier in this guide, to learn how to add simulated data to your classification model. XLMiner SDK can be used to perform, in a fully automated manner, a risk analysis of the model's performance on *new synthetic data* which is individually different, but whose overall behavior is consistent with the known data.

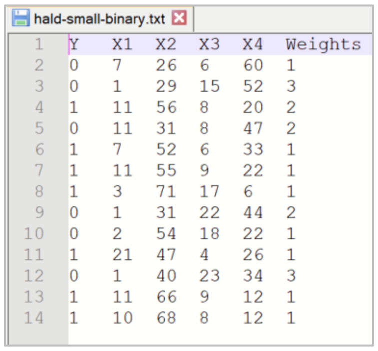
A Find Best Model Classification Example

XLMiner SDK includes comprehensive, powerful support for data mining and machine learning. Using these tools, you can "train" or fit your data to a wide range of statistical and machine learning models: Classification and regression trees, neural networks, linear and logistic regression, discriminant analysis, naïve Bayes, k-nearest neighbors and more. But the task of choosing and comparing these models, and selecting parameters for each one was up to you.

With the new Find Best Model options, you can automate this work as well! Find Best Model uses methods similar to those in (expensive high-end) tools like DataRobot and RapidMiner, to automatically choose types of ML models and their parameters, validate and compare them according to criteria that you choose, and deliver the model that best fits your data.

The following C# example, illustrates how to fit a classification model and score new data using the Find Best Model feature in XLMiner SDK. To run a similar example, compile and run the project, Test.csproj located in …\Frontline Systems\Analytic Solver SDK\XLMiner\Examples\C#. See the function FindBestModel defined in the class, Classification. See the example below which illustrates how to use a single classification method.

This data set uses the hald-small-binary.txt data set to illustrate how to call the FindBestModel feature.


hald-small-binary.txt

Y	X1	X2	X3	X4	Weights
0	7	26	6	60	1
0	1	29	15	52	3
1	11	56	8	20	2
0	11	31	8	47	2
1	7	52	6	33	1
1	11	55	9	22	1
1	3	71	17	6	1
0	1	31	22	44	2
0	2	54	18	22	1
1	21	47	4	26	1
0	1	40	23	34	3
1	11	66	9	12	1
1	10	68	8	12	1

Data Preparation

The **textFile** utility is a simple delimited file reading facility. Users could obtain data from other sources using XLMiner SDK data connectors or any desired tool available for the target programming language.

This example uses the Dataset constructor to partition the dataset, *data*, into two partitions: a training partition containing 60% of the records and a validation partition containing 40% of the records.

Using Dataset constructor to partition the data

```
string targetColumnName = "Y";
Dataset data = new Dataset(Reader.textFile(PATH + "hald-small-
binary.txt"), targetColumnName);
```

Defining the Estimator

Learners may be added to the estimator in two ways: full or partial control.

1. Full control allows users to select the learners and edit the learner parameters.
2. Partial control which allows users to select the learners with their default parameters.

Two learners are added to the estimator, the logistic regression learner, using full control, and the classification tree learner, using partial control.

- The logistic regression learner sets the Fit Intercept option to True. (This is the only option available to this learner.)
- The classification tree learner uses all option settings at their default values.

The learners available to the Find Best Model classification feature are: "Bagging" (Ensemble Method), "Boosting" (Ensemble Method), "DecisionTree", "KNearestNeighbors", "LinearDiscriminant", "LogisticModel", "NaiveBayes", "NeuralNetwork", "RandomTrees" (Ensemble Method).

Defining Find Best Model Estimator

```
var estimator = new FBMC.Estimator()

// add learners
estimator.addLearner(
  new LogisticRegression.Estimator().setFitIntercept(true)
);
estimator.addLearner(
  new CTree.Estimator()
);
```

Fitting a Model

A model is fit to the training dataset using both the logistic regression and classification tree learners.

Fitting a Model

```
var model = estimator.fit(data.input[TRAINING], data.target[TRAINING]);

//print any errors that occur during the fitting process
var fitMessages = model.Messages;
if (fitMessages.Length != 0)
  Console.WriteLine(fitMessages);
```

Examining the Results and Applying the Model

The fitted model is applied to both the training and validation partitions. The model performance table is returned. The modelPerformance output is a table containing evaluations for every available output metric for all available learners on all partitions. The metrics returned are: Accuracy, Specificity, Sensitivity, Precision, and the F1 metric.

```
Console.WriteLine(model.modelPerformance(data.input[TRAINING],
data.target[TRAINING]));

Console.WriteLine(model.modelPerformance(data.input[VALIDATION],
data.target[VALIDATION]));
```

Determining the Best Learner

The accuracy scoring metric is selected as the threshold for determining which learner has performed the best on the data. The setScoringMetric may be changed to Accuracy, Specificity, Sensitivity, Precision or F1. See the table below for a brief description of each statistic.

Statistic	Description
Accuracy ClassificationMetricType.Accuracy	Accuracy refers to the ability of the classifier to predict a class label correctly.
Specificity ClassificationMetricType.Specificity	Specificity is defined as the proportion of negative classifications that were actually negative.
Sensitivity ClassificationMetricType.Sensitivity	Sensitivity is defined as the proportion of positive cases there were classified correctly as positive.
Precision ClassificationMetricType.Precision	Precision is defined as the proportion of positive results that are truely positive.
F1 ClassificationMetricType.F1	Calculated as 0.743 –2 x (Precision * Sensitivity)/(Precision + Sensitivity) The F-1 Score provides a statistic to balance between Precision and Sensitivity, especially if an uneven class distribution exists. The closer the F-1 score is to 1 (the upper bound) the better the precision and recall.

```
var selectedPartition = VALIDATION;
model.setScoringMetric(ClassificationMetricType.ACCURACY);

Console.WriteLine("Best Learner based on partition [" + selectedPartition +
"] " + " and metric [" +
FBMC.Model.getScoringMetricString(model.getScoringMetric()) + "]: " +
model.findBestLearner(data.input[selectedPartition],
data.target[selectedPartition], true));
```

Scoring

Scoring on the validation partition is performed using the learner with the best (largest) value for the accuracy Classification metric.

```
Console.WriteLine(model);
// prediction based on best learner
var predictedLabels = model.predict(data.input[VALIDATION]);
Console.WriteLine(Util.head(Util.colBind(new DataFrame[] {
data.target[VALIDATION], predictedLabels })));
```

A Neural Network Classification Example

The following C# example, illustrates how to fit a classification model and score new data using the Neural Network algorithm. To run a similar example, compile and run the project, Test.csproj located in …\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\C#. See the function NeuralNetwork defined in the class, Classification.

This data set uses the wine.txt data set, which contains the results of a chemical analysis of wines grown in the same region of Italy. Each wine type, A, B, or C is derived using a different cultivar. The analysis found the following thirteen elements in each sample: Alcohol, Malic_Acid, Ash, Ash_Alcalinity, Magnesium, Total_Phenols, Flavonoids, Nonflavanoid_Phenols, Proanthocyanins, Color_Intensity, Hue, OD280_0D315, and Proline.

Data Preparation

1. The **textFile** utility is a simple delimited file reading facility. Users could obtain data from other sources using XLMiner SDK data connectors or any desired tool available for the target programming language.

   This example uses the Dataset constructor to partition the dataset, *data*, into two partitions: a training partition containing 60% of the records and a validation partition containing 40% of the records. (See line 120 in Test.cs for an example of the constructor.)

   Using Dataset constructor to partition the data

   ```
   Dataset data = new Dataset(Reader.textFile(PATH + "wine.txt"),
   targetColumnName);
   ```

Building the Model

1. Due to the somewhat abstract nature of a neural network algorithm, there are more property values to work with compared to more conventional mining models. *WeightDecay*, *LearningRate* all relate to the way a learning network approaches its new "more knowledgeable" state during training. At first, it might seem that a fast learning rate would be a good thing, but it's generally not a good idea to chase the input data too wildly. Fast learning also means fast forgetting if some input data does not support a developing pattern. Good neural networks need time to be exposed to enough training data so that a pattern can emerge.

   Neural networks require the definition of an activation function that determines how artificial neurons are influenced by their input. In the early days of neural network research, scientists were discouraged by the fact that neural networks could not classify all but the most trivial input. They realized that severe limitations are imposed if the activation function is linear. LOGISTIC_SIGMOID, TANH, and SOFTMAX are among the more commonly used nonlinear activation functions; all three are supported by XLMiner.

   In the following example, one "hidden" layer is defined, consisting of 4 nodes. There is no universally agreed upon rule of thumb for choosing the number of hidden layers and the number of nodes within each layer – this is discussed in further detail later in this chapter.

Defining Neural Network Estimator

```
var estimator = new NeuralNetworkClassification.Estimator()
{
  NumNeurons = new int[] { 4 },
  NumEpochs = 100,
  ErrorTolerance = 0.01,
  WeightDecay = 0.0,
  LearningRate = 0.4,
  WeightMomentum = 0.7,
  HiddenLayerActivation = ActivationType.LOGISTIC_SIGMOID,
  OutputLayerActivation = ActivationType.SOFTMAX,
  WeightInitSeed = 12345,

  LearningOrder = LearningOrder.RANDOM,
  LearningOrderSeed = 12345
};
```

Note: During online (stochastic) learning, the Learning Order parameter (*XLMiner.LearningOrder*) allows to present the training records to a neural network in a random order. This might have a dramatic effect on a network's training, especially when the training data is arranged according to some order.

2. It's important to know when to quit. You can tell the network to quit when there doesn't seem to be any changes, when the model has become close to the data, or simply when a certain amount of time has elapsed. It is common to place a limit on the number of times the neural network algorithm will pass through the data. A single pass through the data is usually referred to as an "epoch". In this example, the *NumEpochs* property has been set to 30.

   Setting Stopping Rules

```
{
   DataForErrorComputation =
   StoppingRule.DataForErrorComputation.TRAIN_AND_VALID,
     MaxNumEpochsWithNoImprovement = 30,
     MinRelativeErrorChange = 0.00001,
     MinRelativeErrorChangeComparedToNullModel = 0.0001,
     MaxTrainingTimeSeconds = 5.0,
}
```

3. XLMiner SDK supports categorical variables as input variables when using a classification technique. Since none of the output variables in this example are categorical, these two lines of code are not used and are shown for informational purposes only. Note: Any non-numeric variables are automatically considered as categorical (nominal) variables.

   Example code for specifying categorical variables

```
//CategoricalFeaturesOffsets = new int[] { 0, 1 };
```

```
//or
//CategoricalFeaturesNames = new string[] { "X1", "X2" };
```

4. A new estimator, *Scaler*, is constructed using the *Scaler.Estimator* object. Data is rescaled to values of [0,1] (so that values for each variable are in [0,1] range) using the NORMALIZATION scaling technique. A new DataFrame, *rescaledData* is created to hold the rescaled data for both partitions.

   Using Scaler estimator to rescale values using NORMALIZATION technique

   ```
   Scaler.Estimator scaler = new Scaler.Estimator()
   {
    Technique = estimator.HiddenLayerActivation ==
    XLMiner.NeuralNetwork.ActivationType.LOGISTIC_SIGMOID ?
      Scaler.Technique.NORMALIZATION:
      Scaler.Technique.ADJUSTED_NORMALIZATION
   };
   //fit a model to the training set using the scaler estimator
   var scalerModel = scaler.fit(data.input[TRAINING]);

   //create new Dataset, rescaledData, from data Dataset
   Dataset rescaledData = data;

   //Apply rescaling model to both the Training and Validation partitions
   (transforming the data)
   rescaledData.input[TRAINING] =
   scalerModel.transform(data.input[TRAINING]).setName(TRAINING);
   rescaledData.input[VALIDATION] =
   scalerModel.transform(data.input[VALIDATION]).setName(VALIDATION);
   ```

5. The model is fit using the training partition for neural network learning and the validation partition for evaluating the errors and examining stopping conditions.

   Fitting a Model

   ```
   var model = estimator.fit(rescaledData.input[TRAINING],
     rescaledData.input[VALIDATION],
     rescaledData.target[TRAINING],
       rescaledData.target[VALIDATION]);

   //Setting the successClass and successProbability parameters
   model.SuccessClass = rescaledData.successClass;
   model.SuccessProbability = rescaledData.successProbability;
   ```

   a) The fitted model is exported into PMML format.

      Saving model to PMML file format

```
    var pmmlFile = PATH + "PMML/classification-neural-network.xml";
    model.toPMML(pmmlFile,scalerModel);
```

Examining the Results and Applying the Model

6. The fitted Classification model is used to score the rescaled training partition.
   - Two columns, *trainingTargetColumn* and *predictedLabels*, are merged into a new DataFrame, then the two columns are printed side-by-side.
   - The fitted model is used to compute posterior probabilities for each record (wine instance) to belong to one of the classes denoting the wine type.
   - The function printClassificationMetrics prints the confusion matrix using the trainingTargetColumn and predictedLabels.

   See Test.csproj at //Frontline Systems/Analytic Solver SDK/XLMinerSDK/Examples/C# for the full example.

   Printing the confusion matrix

```
using (var predictedLabels =
model.predict(rescaledTrainingData.input[TRAINING]))
{
  Console.WriteLine(Util.tail(Util.colBind(new DataFrame[]
  {data.target[TRAINING], predictedLabels })));
  Console.WriteLine(Util.tail(model.posteriorProbability
  (rescaledData.input[TRAINING])));
  PrintClassificationMetrics(data.target[TRAINING], predictedLabels);
}
```

7. The fitted Classification model is used to score the rescaled validation partition.
   - Two columns, *validationTargetColumn* and *predictedLabels*, are merged into a new DataFrame, then the two columns are printed side-by-side.
   - Again, the fitted model is used to compute posterior probabilities for each record (wine instance) to belong to one of the classes denoting the wine type.

   See Test.csproj at //Frontline Systems/Analytic Solver SDK/XLMinerSDK/Examples/C# for the full example.

   Fitted Classification model used to score rescaled validation partition

```
using (var predictedLabels =
model.predict(rescaledData.input[VALIDATION]))
{
 Console.WriteLine(Util.tail(Util.colBind(new DataFrame[]
 {
   data.target[VALIDATION], predictedLabels })));
   Console.WriteLine(Util.tail(model.posteriorProbability
   (rescaledData.input[VALIDATION])));
   PrintClassificationMetrics(data.target[VALIDATION], predictedLabels);
 }
```

Classification Algorithms

As mentioned in the Introduction, the selection of the classification method is based on several factors such as the number of variables (or features) in the data set, the number of records in the dataset, the speed of the algorithm, and the desired level of interpretability. XLMiner SDK includes six core classification algorithms along with three ensemble methods:

- Classification Trees
- Discriminant Analysis
- k-Nearest Neighbors
- Logistic Regression
- Naïve Bayes
- Neural Networks
- Ensembles (Bagging, Boosting, Random Trees)
- Find Best Model

A complete discussion of each method is beyond the scope of this guide; however, a brief discussion of each is contained in the following sections.

Find Best Model Method

Use Find Best Model to automatically run all or a portion of the classification methods on a dataset, all at once. The learner performing the "best" on the dataset is determined by a user – selected classification metric. Based on this metric, Find Best Model scores the training, validation, or test partitions using this "best" learner.

Discriminant Analysis

Discriminant Analysis (DA) uses either a linear or (new in the latest version!) quadratic combination of input (predictor) variables to classify a categorical output variable into two or more classes.

LDA, the default, employs these combinations (discriminant functions) to maximize the uniformity within each class while at the same time differentiating the classes as much as possible. For each record, the linear combinations are used to compute scores that measure the distance of an observation to each class. The observation is assigned to the class with the highest classification score. LDA depends on two leading assumptions when calculating classification scores: that input variables are roughly from a multivariate normal distribution and variability between class members is consistent across the class. LDA can be susceptible to outliers, and the size of the smallest class must be larger than the number of input variables. In addition, the predictive power of this method is reduced when the input variables are correlated.

The latest version of XLMiner SDK now contains Quadratic Discriminant Analysis (QDA). QDA produces a quadratic decision boundary, rather than a linear decision boundary. While QDA also assumes that the data is normally distributed, QDA does *not* assume that all classes share the same covariance matrix.

QDA is a more flexible technique when compared to LDA. QDA's performance improves over LDA when the class covariance matrices are disparate. Since each class has a different covariance matrix, the number of parameters that must be estimated increases significantly as the number of dimensions (predictors) increase. As a result, LDA might be a better choice over QDA on datasets with small numbers of observations and large numbers of classes. It's advisable to try both techniques to determine which one performs best on your model. You can easily switch between LDA and QDA simply by setting this option to true: estimator.Quadratic = true.

Note: Users of XLMiner's Discriminant Analysis classification method can specify prior probabilities. This might be useful if the percentage of observations belonging to a specific class is known. If the prior probabilities are known, this method can use these probabilities when calculating the posterior probabilities.

Pros: Discriminant Analysis is very fast, even for large data. This technique is especially useful and well-interpretable when the number of features is not large. If group distributions are indeed normal, Discriminant Analysis will produce a perfect fit. Discriminant Analysis produces a stable model when groups are well-separated and allows multiclass learning which can explain data in lower dimensions.

Cons: Discriminant Analysis does not apply when the number of features exceeds the number of records. For high-dimensional data, a Discriminant Analysis model becomes overcomplicated and less stable. For highly non-Normal distributions, Discriminant Analysis may fail to capture the structure of the data.

Logistic Regression

Logistic Regression measures the relationship between the categorical dependent variable and one or more independent variables by estimating probabilities using a logistic function. This method is viewed as a special case of a generalized linear model, and is thus analogous to linear regression. Examples of binary responses are "success/fail," "survive/die," "yes/no," "buy/don't buy," or default/don't default."

XLMiner's Logistic Regression allows users to specify several parameters including the presence of an intercept term, confidence level for the calculated odds, and the maximum number of iterations. Collinearity diagnostics and other evaluations can be obtained from the Logistic Regression model.

Pros: Logistic Regression is a very popular classification method (especially for problems with 2 classes) which works extremely well in real-world applications. This method makes no assumptions about the distribution of independent variables or concerning the linear relationship between independent and response variables. Unlike Linear Regression, error terms are not assumed to be normally distributed. Logistic Regression performs well with data containing categorical predictors (1-of-C encoded using XLMiner) and is able to handle large high-dimensional datasets. Efficient convex optimization algorithms exist.

Cons: No closed-form solution exists for logistic regression. For low dimensional data where classes are well-separated, this method may yield to Discriminant Analysis in stability. Logistic regression suffers from low rank matrices (such as in linear models), for example when the number of records is less than the number of features or when collinearity is present. *XLMiner SDK provides **embedded variable selection** and **best subset** techniques to overcome this issue.*

k-Nearest Neighbors

The k-Nearest Neighbors classification method is a very simple but powerful algorithm which makes classification decisions for a given record based on information provided by neighboring records. Using the Euclidean distance measure, this method identifies the k observations in the training data that are most similar to a given observation and performs majority voting – choosing the most frequent group among the k nearest neighbors. The learning stage is absent for this algorithm, the training data is the model. This classification technique requires independent variables to be scaled appropriately. The best model can be chosen by assessing the classification error for various values of k. Using the validation error is more appropriate in order to decrease the chance of overfitting.

Pros: The k-Nearest Neighbors algorithm very often performs well in practice, producing stable and easily interpretable results. It is a "Lazy Learning" method -- the "model" is immediately available.

Cons: The k-Nearest Neighbors algorithm is computationally and memory-wise expensive. It focuses on local structure of data and can fail to capture the global picture. This method can suffer from the "Curse of dimensionality" which, in this case, refers to the phenomena that occurs in high dimensional datasets where the "nearest neighbors" become more and more blurry. The k-Nearest Neighbor method is extremely sensitive to outliers and noise and may demonstrate poor performance on data with undersampled or oversampled groups.

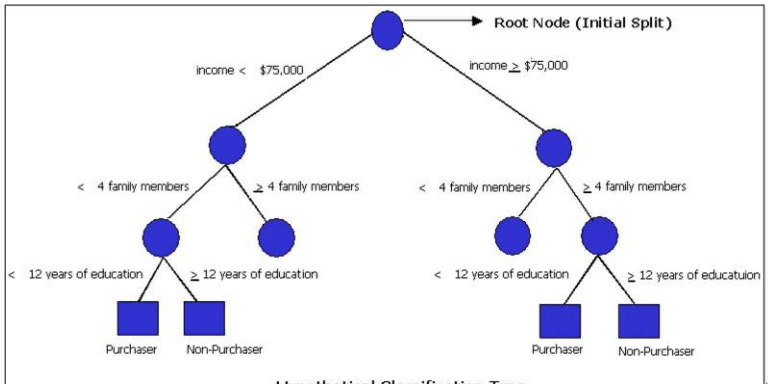
Classification Tree

Classification Trees, also known as Decision Trees, is a classification method that generates easily understood "rules" (i.e., "If INCOME > 75,000 AND NUM FAMILY MEMBERS > 4 AND YRS OF EDU < 12 THEN class = PURCHASER).

Initially, a Training Set is created where the classification ("purchaser" or "non-purchaser") is known for each record. Next, the algorithm systematically assigns each record to one of two subsets based on an input variable condition (i.e., INCOME >= $75,000 or INCOME < $75,000). This process is repeated for each subset using a new condition until no more useful splits are found in each subset.

In the example classification tree below, the Training Set is initially split using the rule, INCOME >= 75,000. This opening split is referred to as the root node. Following the tree down the right "branch," the next splitting rule is "FAMILY MEMBERS >= 4." Any records with greater than four family members are again split on the rule "YEARS of EDUCATION >= 12." The "leaves" hold the final classification of "purchaser" or "non-purchaser."

The final classification rule for the non-purchaser using the right branch would be: If Income >= $75,000 AND Family Members >= 4 AND Years of Education >=12 THEN Class = 0 (Non-Purchaser).



Hypothetical Classification Tree

XLMiner uses the Gini index as the splitting criterion. A Gini index of 1 indicates that each record in the node belongs to a different category, while a Gini index of 0 indicates that each record belongs to the same category. For a complete discussion of this index, see *Classification and Regression Trees* (3) by Leo Breiman and Richard Friedman.

Pruning is the process of removing leaves and branches to improve the performance of the decision tree. The tree-building algorithm splits at the root node, where the largest number of records and the most information are known. Each subsequent split has a smaller and less representative population of records from which to gain information. As a result, towards the bottom of the tree, a particular node might display information specific only to the records assigned to that node. Sometimes these rules can become meaningless and the tree must be "pruned". Decision trees are typically pruned using the Validation Set.

XLMiner SDK allows users to limit tree growth by levels, splits, or nodes or by specifying the number of records in a terminal node.

Classification Trees produce easy-to-understand rules as their output. This can be a big advantage as users typically understand these models after only a brief study and can simply "follow the rules" to understand the class assignments. Since this technique does not require the data to be normalized or the removal of blank values, the amount of data preparation is normally minimal. Classification trees can be used with large data sets using standard computing resources to produce a model in a reasonable amount of time.

Pros: Classification trees produce well-interpretable models with transparent results comprised of explicit if-then rules by non-expert users. This method works well with raw data having different scales, missing values and outliers. Classification trees are computationally efficient for moderate size datasets and possess implicit *feature selection* where top nodes correspond to the most informative, important features. The classification tree method does not impose explicit assumptions about underlying relationships in data.

Cons: Classification Trees provide a greedy heuristic approach for generally NP-Hard problems. The solution corresponds to a local optimum. Often the predictive accuracy of classification trees is weaker than with other classification techniques.

Naïve Bayes

This classification method uses the Bayes Theorem to classify observations into two or more classes. Bayes Theorem estimates the probability of event A occurring given the probability that event B has already occurred or:

$$Prob\ (A\ given\ B) = \frac{Prob\ (B\ given\ A) * Prob(A)}{Prob(B)}$$

For example, if we were to calculate the probability of a 1st grade student enrolled in public school allergic to peanuts, and there are 1,000 students in the Training Set and 50 are allergic to peanuts, then Prob (A and B) is 5%. If there are 500 1st graders in the Training Set, then Prob (A) is 50%. According to Bayes Theorem, the possibility of a 1st grader being allergic to peanuts is 10% (50/500). The naïve Bayes classifier assumes that each input variable is conditionally independent of all other input variables.

A disadvantage of this method is that it requires a large number of records to obtain accuracy. In addition, naïve Bayes is unable to determine or "learn" interactions between features (i.e., it is unable to "understand" that you love avocados and tomatoes, but not both in the same dish).

In some circumstances, the distribution of a specific input variable is not characteristic of the larger population (i.e., if there were only a few 1st graders in the training partition). For cases such as these, XLMiner SDK allows users to specify Prior Probabilities to compensate for the difference between the distribution of records in the training data set and the true population.

Common uses of this method include document classification (i.e., classifying an email as spam, classifying newspaper articles as technology, politics, or sports, or classifying a review as negative or positive).

Pros: The Naïve Bayes classification method is applicable to very high-dimensional data, where other more complex models may fail. A small training sample is usually sufficient for parameter estimation. This method can be applied to discrete (categorical) and continuous data and is computationally and memory-wise efficient. Naïve Bayes is robust in terms of handling irrelevant features and is a perfect classifier when the independence assumption holds true.

Cons: In the Naïve Bayes algorithm, the independence assumption is rather strong. Naïve Bayes will not adjust for the case when features are strongly related to each other. For multinomial models, features in new patterns must contain values which the model has already observed during the training phase. Otherwise, the probability is undefined.

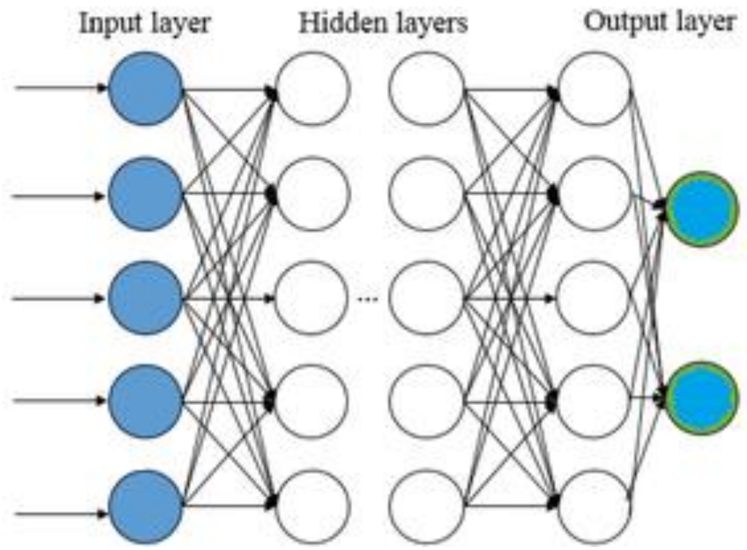
Neural Networks

Artificial neural networks (ANNs) have been said by some to work "like the brain works". It is more accurate to say that the design of artificial neural networks is biologically inspired but the implementation is purely digital.

Artificial neural networks are still a subject of active research. An ANN architecture commonly used in data mining has an input layer which receives user data, an output layer that indicates the classification, and a variable number of "hidden layers" which get their name from the fact that the values for each of the hidden nodes is used only by the neural network itself. As the network is trained, connections between nodes change weight. Connections that are often reinforced gain weight (or become stronger), and others weaken.

Loosely speaking, we can say that the first hidden layer discerns patterns in the input data. A second hidden layer discerns patterns in the first hidden layer, or patterns within the patterns of the input data. A third layer, if it exists, would detect patterns within the second hidden layer, or patterns, within the patterns, within the patterns, of the input data, and so forth. Complex and subtle data could be analyzed with many hidden layers, but at the cost of large calculation time.

As mentioned in the discussion of the example code at the beginning of this chapter, there are no simple rules for



selecting the optimal number of hidden layers and the optimal number of nodes within each layer. Very, very generally, we can say that the number of nodes in the hidden layers need not exceed the number of inputs. One hidden layer is probably sufficient for most classification problems encountered in the world of business and commerce. Detecting subtle effects may require adding additional layers, and will also require a greater volume of data for training and testing to avoid the ever-present hazard of overfitting.

Many parameters can be varied to tune a neural network mode. The three most important are the number of nodes, or "neurons" in a hidden layer, the number of hidden layers, and the definition of the activation function that determines how the numerical values for nodes and connections will be calculated.

There are many additional parameters that can be changed to control the behavior of the artificial neural network. While this control can be valuable, it also increases the risk of overfitting the data. In other words, a neural network could be developed that fits the input data almost perfectly but has no generality and cannot effectively categorize new data. While all data mining models must be carefully validated, validation is especially critical for neural networks.

Pros: Neural Networks are "Universal Approximators" and can be used when the nature of the data is barely interpretable. These methods can detect highly nonlinear relationships between independent and dependent variables and also can recognize and take into account relationships between predictors. Learning is automated to some extent in neural networks so less formal modeling is required. Neural networks provide robust models for large high-dimensional datasets, overcoming many problems of conventional learning techniques. Lastly, no strong explicit assumptions are involved in neural networks.

Cons: Neural networks are considered a "Black-box" learning method, meaning that the models are almost not interpretable and they can be computationally expensive. These methods are prone to overfitting, unless necessary steps are taken to prevent this drawback. Neural networks greatly depend on chosen architecture, optimization parameters, choice of activation and error functions. However, general rules do exist to simplify selection of these options or automatic selection can be used.

Ensemble Methods

Some data mining models are sensitive to the effects of outliers, a small number of points far away from the mean can have a disproportionate effect on the resulting model. The untoward effects of outliers can be reduced using ensemble methods. For example, we can create a set of decision trees, rather than a single tree, by randomly sampling again and again from our input data. The “average” of these output trees is likely to be a better representation of the data than any of the individually calculated trees.

XLMiner offers three powerful ensemble methods: Boosting, Bagging, and Random Trees (also known as Random Forests). Both Bagging and Boosting accept any of the following classifiers as their base learner: Discriminant Analysis, Logistic Regression, k-Nearest Neighbors, Classification Trees, Naïve Bayes, or Neural Networks. The Random Trees ensemble method uses the Classification Tree as a weak learner internally.

The six classification methods can be used to find one model resulting in good classifications of the new data. Ensemble methods, however, allow us to combine multiple “weak” classification models that, when taken together, form a new, more accurate “strong” classification model. These methods work by creating multiple diverse classification models, by taking different samples of the original data set, and then combining their outputs. (Outputs may be combined by several techniques including majority vote for classification and averaging for regression.) This combination of models effectively reduces the variance in the “strong” model. The three types of ensemble methods offered in XLMiner SDK (Bagging, Boosting, and Random Trees) differ on the following three items:

1) the selection of training data for each classifier or a weak learner;
2) how the “weak” models are generated;
3) how the outputs are combined. In all three methods, each weak learner is trained on the entire training dataset to become proficient in some portion of the data set.

Bagging, or bootstrap aggregating, was one of the first ensemble algorithms developed. It is a simple, effective algorithm. Bagging generates several training sets by using random sampling with replacement (bootstrap sampling), applies the classification algorithm to each data set, then takes the majority vote among the models to determine the classification of the new data. The biggest advantage of bagging is the relative ease in which the algorithm can be parallelized, making this method a better selection for very large data sets.

In comparison, Boosting successively trains models to concentrate on the misclassified records in previous models. Once completed, all classifiers are combined by a weighted majority vote. XLMiner SDK offers three different variations of Boosting as implemented by the AdaBoost algorithm—one of the most popular ensemble algorithms in use today: M1 (Freund), M1 (Breiman), and SAMME (Stagewise Additive Modeling using a Multi-class Exponential).

For Classification Trees, a third ensemble method is available, Random Trees. This method is a variation of Bagging that works by training multiple “weak” classification trees using a fixed number of randomly selected features (square root of the number of features for classification and one third of the number of features for regression), then takes the mode of each class to create a “strong” classifier. Using this method, the number of “weak” trees generated could range from several hundred to several thousand depending upon the size and difficulty of the training Set. Random Trees are parallelizable because they are a variant of Bagging. However, since Random Trees selects a limited number of features in each iteration, the performance of Random Trees is typically faster than that of Bagging.

Regression Methods

Introduction

Regression is the goal of much of data mining. We not only wish to model existing data, we want to predict whether individuals are likely to become customers, whether loan applicants are likely to default, and whether economic conditions favor continued growth. XLMiner provides a rich set of methods to apply data mining models to regression.

As mentioned, the categorization of data mining tasks is not rigid or exclusive, and the regression methods are a clear example of this. An economist might, for example, develop a model for local consumer prices using Multiple Linear Regression. This same model could very well be used in the prediction of future cost of living indexes for varied geographical areas.

After the model is fit, or after the regression algorithm has determined the association between the predictor variables and output variable, the model may be used to score new data or predict future values for the output variable based on new input values. XLMiner SDK offers several performance metrics for evaluating the accuracy and predictive power of a model.

The regression algorithm selection depends upon several factors, including the size of the data set, the nature of the data (continuous or discrete), and algorithm speed. As a result, along with examining fundamental properties and rules of thumb, data scientists will often try several different regression algorithms to see which one performs best on their data set. XLMiner SDK features the following regression methodologies:

- Multiple Linear Regression,
- k-Nearest Neighbors,
- Regression Trees,
- Neural Networks
- Ensembles (Bagging, Boosting, Random Trees)
- Find Best Model

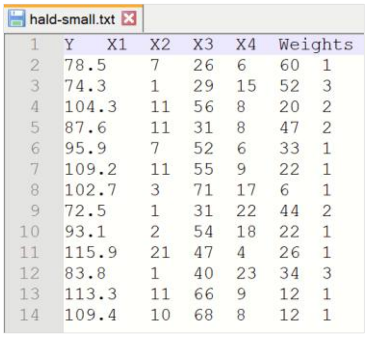
See the Synthetic Data Generation chapter, that appears earlier in this guide, to learn how to add simulated data to your regression model. XLMiner SDK can be used to perform, in a fully automated manner, a risk analysis of the model's performance on *new synthetic data* which is individually different, but whose overall behavior is consistent with the known data.

A Find Best Model Regression Example

As discussed previously, XLMiner SDK includes comprehensive, powerful support for data mining and machine learning. Using these tools, you can "train" or fit your data to a wide range of statistical and machine learning models: Classification and regression trees, neural networks, linear and logistic regression, discriminant analysis, naïve Bayes, k-nearest neighbors and more. But the task of choosing and comparing these models, and selecting parameters for each one was up to you. With the new Find Best Model options, you can automate this work as well!

The following C# example, illustrates how to fit a regression model and score new data using the Find Best Model feature in XLMiner SDK. To run a similar example, compile and run the project, Test.csproj located in …\Frontline Systems\Analytic Solver SDK\XlminerSDK\Examples\C#. See the function FindBestModel defined in the class, Regression. See the example below which illustrates how to use a single regression method.

This data set uses the hald-small.txt data set to illustrate how to call the FindBestModel feature.



```
Y     X1  X2  X3  X4  Weights
78.5  7   26  6   60  1
74.3  1   29  15  52  3
104.3 11  56  8   20  2
87.6  11  31  8   47  2
95.9  7   52  6   33  1
109.2 11  55  9   22  1
102.7 3   71  17  6   1
72.5  1   31  22  44  2
93.1  2   54  18  22  1
115.9 21  47  4   26  1
83.8  1   40  23  34  3
113.3 11  66  9   12  1
109.4 10  68  8   12  1
```

Data Preparation

The **textFile** utility is a simple delimited file reading facility. Users could obtain data from other sources using XLMiner SDK data connectors or any desired tool available for the target programming language.

This example uses the Dataset constructor to partition the dataset, *data*, into two partitions: a training partition containing 60% of the records and a validation partition containing 40% of the records.

Using Dataset constructor to partition the data

```
string targetColumnName = "Y";
Dataset data = new Dataset(Reader.textFile(PATH + "hald-small.txt"),
targetColumnName);
```

Defining the Estimator

Learners may be added to the estimator in two ways: full or partial control.

3. Full control allows users to select the learners and edit the learner parameters.
4. Partial control which allows users to select the learners with their default parameters.

Two learners are added to the estimator, the linear regression learner, using full control, and the regression tree learner, using partial control.

- The linear regression learner sets the Fit Intercept option to True. (This is the only option available to this learner.)
- The regression tree learner uses all option settings at their default values.

The learners available to the Find Best Model classification feature are: "Bagging" (Ensemble Method), "Boosting" (Ensemble Method), "DecisionTree", "KNearestNeighbors", "LinearModel", "NeuralNetwork", "RandomTrees" (Ensemble Method).

Defining Find Best Model Estimator

```
var estimator = new FBMR.Estimator()

// add learners
estimator.addLearner(
  new LinearModel.Estimator().setFitIntercept(true)
);
estimator.addLearner(
  new DecisionTree.Estimator()
);
```

Fitting a Model

A model is fit to the training dataset using both the linear regression and regression tree learners.

Fitting a Model

```
var model = estimator.fit(data.input[TRAINING], data.target[TRAINING]);

//print any errors that occur during the fitting process
var fitMessages = model.Messages;
if (fitMessages.Length != 0)
  Console.WriteLine(fitMessages);
```

Examining the Results and Applying the Model

The fitted model is applied to both the training and validation partitions. The model performance table is returned. The modelPerformance output is a table containing evaluations for every available output metric for all available learners on all partitions. The regression metrics returned are:
the F1 metric.

```
Console.WriteLine(model.modelPerformance(data.input[TRAINING],
data.target[TRAINING]));

Console.WriteLine(model.modelPerformance(data.input[VALIDATION],
data.target[VALIDATION]));
```

Determining the Best Learner

The accuracy scoring metric is selected as the threshold for determining which learner has performed the best on the data. The setScoringMetric may be changed to R2, SSE, MSE, RMSE or MAD. See the table below for a brief description of each statistic.

Statistic	Description
R2 RegressionMetricType.R2	Coefficient of Determination - Examines how differences in one variable can be explained by the difference in a second variable, when predicting the outcome of a given event. RMSE = $\frac{SSR}{SST} = \frac{\sum_i(\hat{y}_i - \bar{y})^2}{\sum_i(y_i - \bar{y})^2}$ where $\hat{y}_i$ is the predicted value for obs i y_i is the actual value for obs i $\bar{y}$ is mean of the y values
SSE RegressionMetricType.SSE	Sum of Squared Error – The sum of the squares of the differences between the actual and predicted values. SSE = $\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$ $\hat{y}_i$ is the predicted value for obs i y_i is the actual value for obs i
MSE RegressionMetricType.MSE	Mean Squared Error – The average of the squared differences between the actual and predicted values. MSE = $\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$

	$\hat{y}_i$ is the predicted value for obs i y_i is the actual value for obs i
RMSE RegressionMetricType.RMSE	Root Mean Squared Error – The standard deviation of the residuals. RSE = $\sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$ $\hat{y}_i$ is the predicted value for obs i y_i is the actual value for obs i
MAD RegressionMetricType.MAD	Mean Absolute Deviation - Average distance between each data value and the sample mean; describes variation in a data set. MAD = $\frac{1}{n}\sum_{i=1}^{n}\lvert x_i - \bar{x}\rvert$ where x_i is the ith obs in the sample where $\bar{x}$ is the sample mean

```
var selectedPartition = VALIDATION;
model.setScoringMetric(RegressionMetricType.R2);

Console.WriteLine("Best Learner based on partition [" + selectedPartition +
"] " + " and metric [" +
FBMR.Model.getScoringMetricString(model.getScoringMetric()) + "]: " +
model.findBestLearner(data.input[selectedPartition],
data.target[selectedPartition], true));
```

Scoring

Scoring on the validation partition is performed using the learner with the best (largest) value for the accuracy metric.

```
Console.WriteLine(model);
// prediction based on best learner
var predictedLabels = model.predict(data.input[VALIDATION]);
Console.WriteLine(Util.head(Util.colBind(new DataFrame[] {
data.target[VALIDATION], predictedLabels })));
```

Regression Algorithms

The following C# example illustrates how to fit a regression model and score new data using the Bagging ensemble method with the Regression Tree algorithm as the base learner. To run a similar example as the one shown below, compile and run the project, Test.csproj, located in …\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\C#. See the function Example.Regression.Ensemble.Bagging().

This example uses the **BostonHousingReg.txt** example dataset, which contains 14 variables and concerns the housing values of owner-occupied homes in census tracts within the Boston area. A description of each variable is provided in the table below.

This example:

- Reads data from the text file BostonHousingReg.txt
- Partitions the data set into Training and Validation Sets.
- Constructs a new Bagging ensemble estimator object for regression using a regression tree estimator as the base leaner. Various options are set for both estimators.

- Fits a Bagging ensemble model using the estimator's *fit()* method
- Scores new data (the Validation dataset in this example) using the *predict()* method of the model

Data Preparation

1. The **textFile** utility is a simple delimited file reading facility. Users could obtain data from other sources using XLMiner SDK data connectors or any desired tool available for the target programming language.

   This example uses the Dataset constructor to partition the dataset, *data*, into two partitions: a training partition containing 60% of the records and a validation partition containing 40% of the records. (See line 120 in Test.cs for an example of the constructor.)

   Using Dataset constructor to partition the data

```
string targetColumnName = "MEDV";

Dataset data = new Dataset(Reader.textFile(PATH + "BostonHousingReg.txt"),
targetColumnName);
```

Building the Model

2. A new Bagging regression estimator is constructed. Afterwards, the regression tree algorithm is specified as the base learner using *estimator.setBaseEstimator()*.

   Defining Bagging estimator with basic parameters

```
//construct regression bagging estimator

var estimator = new RBagging.Estimator()

   {

    NumWeakLearners = 2,

    BootstrapSeed = 10

   };

//insert any estimator as a weak learner to be bagged.  In this example

//the regression tree algorithm is used

estimator.setBaseEstimator(new

RTree.Estimator()).setMinNumRecordsInLeaves(data.NumTrainingRows/10));
```

3. The model is fit using the Training Set input and target columns.

   Fitting a Model

```
//Fit the model

var model = estimator.fit(data.input[TRAINING], data.target[TRAINING]);

//save model to PMML file format

var pmmlFile = PATH + "PMML/regression-bagging.xml";

model.toPMML(pmmlFile);
```

Examining and Applying the Results

4. First the predicted labels for the validation partition are computed. Next, two columns, *validationTargetColumn* and *predictedLabels*, are merged into a new DataFrame, then the two columns are printed side-by-side.

   Scoring Validation Partition

   ```
   var predictedLabels = model.predict(data.input[VALIDATION]);
   Console.WriteLine(Util.head(Util.colBind(new DataFrame[]
   {data.target[VALIDATION], predictedLabels})));
   ```

Regression Algorithms

As mentioned in the Introduction, the selection of the regression method is based on several factors such as the number of variables (or features) in the data set, the number of records in the data set, the speed of the algorithm, and desired level of interpretability.

A complete discussion of each method is beyond the scope of this guide. However, a brief discussion of each, along with common applications, may be found below.

Find Best Model Method

Use Find Best Model to automatically run all or a portion of the regression methods on a dataset, all at once. The learner performing the "best" on the dataset is determined by a user – selected regression metric. Based on this metric, Find Best Model scores the training, validation, or test partitions using this "best" learner.

Multiple Linear Regression Method

Multiple linear regression has a long and venerable history in data mining and in statistics in general. It starts by assuming a linear relationship between *n* independent variables and a dependent variable indicated by “Y” in the following equation:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots \beta_n x_n + \epsilon$$

The values in the equation above are often taken directly from the data of interest, but this need not be the case. For example, x_1 might be the square or the logarithm of an actual data point. Thus “linear” algorithms can describe relationships that are decidedly non-linear.

A great benefit of linear regression is the tremendous transparency of the result. One ends with coefficients of an equation. Not only is it possible to plug in values to make predictions, but one can easily see the contribution of each input by examining the equation. This characteristic has made multiple linear regression a favorite among many scientists, including economists who need to develop mathematical models. The XLMiner SDK linear regression algorithms accept both numerical and categorical input.

Pros: Multiple Linear Regression offers two key advantages. The first is the algorithm’s ability to uncover relationships between one or more of the predictor variables to the output variable and the second is the algorithm’s ability to identify outliers or anomalies in the data.

Cons: Care must be taken to ensure the data does not violate the underlying statistical assumptions of linearity and normal distribution of the error terms. Considerable experimentation is often required to transform the input data to approximate linearity.

k-Nearest Neighbors Method

In the previous chapter, the k-Nearest Neighbors classification method is described. This method can also be extended to predict the value of a continuous output variable as well as a categorical output variable. When used for regression, neighboring observations are again selected based on the Euclidean similarity measure. However, in this case, the measure calculates the distance between the observations and the k existing observations. A weighted average of the output variables for the k-nearest neighbors is calculated. This calculated value becomes the observation's predicted output value. See the k-Nearest Neighbors Method in the Classification Methods chapter for this algorithm's advantages and disadvantages.

Regression Tree Method

Regression Trees behave similarly to classification trees in that they produce easy-to-understand and interpret "rules." However, where classification trees predict the class for a given observation (i.e., purchaser/non-purchaser), Regression Trees predict a continuous value for the output variable. For more information on Regression Trees, see Classification Trees Method in the Classification Methods chapter. See the Classification Tree Method in the Classification Methods chapter for this algorithm's advantages and disadvantages.

Neural Network Method

We encountered artificial neural networks in the classification section. There is nothing fundamentally different about the neural networks we encounter in the regression chapter. But while the inner workings are essentially the same, classification neural networks output a categorical value that suggests the class membership of the input, regression neural networks output a continuous variable representing a prediction of some numerical value such as securities prices, inflation rate, or housing prices. In this way, the applications of neural networks are analogous to those of linear regression, with the obvious exception that neural networks can be used in situations where simple mathematical assumptions about the input data simply do not apply. See the Neural Network Method in the Classification Methods chapter for this algorithm's advantages and disadvantages.

Ensemble Methods

XLMiner offers three powerful ensemble methods: Boosting, Bagging, and Random Trees, also known as Random Forests. Both Bagging and Boosting accept any predictor as its base learner: Multiple Linear Regression, k-Nearest Neighbors, Regression Trees, or Neural Networks. The Random Trees ensemble method uses the Regression Tree method internally. For more information on Ensemble Methods, please see the Classification Methods chapter.

Clustering Methods

Introduction

Clustering methods share many features in common with classification techniques. In general, classification techniques assign data elements to categories that are known in advance, such as handwritten numerals between 0 and 9 or successful graduation from a university. In contrast, clustering methods attempt to discern similarities and differences among data points which may not be previously known. Some of the greatest successes, both in terms of statistical success and impact on public health, have been cluster analyses of disease outbreaks.

k-Means Method

One disadvantage of this algorithm occurs when the majority of observations belong to one class. In this case, the records of the dominant class could skew the prediction of the new record.

Pros: Fast and efficient even for large data. Easy to analyze and interpret the results. Works well in practice even if the assumptions are not met.

Cons: K-Means algorithms require the number of clusters to be specified in advance. The result is locally-optimal and depends on the choice of initial centers. Sensitive to outliers and non-uniform clusters with different densities.

The example below uses k-Means to perform a cluster analysis on the Cluster dataset contained within cluster.txt. Three clusters are created (*NumClusters = 3*).

K-Means clustering example code

```
public static int KMeans()
  {
      try
      {
          var data = Reader.textFile(PATH + "cluster.txt");

          //construct new k-Means clustering estimator
          var estimator = new KMeans.Estimator()
          {
              NumClusters = 3,
              MaxIterations = 10,
              NumStarts = 1,
              RandomSeed = 12345
          };
          Console.WriteLine(estimator);
```

```
            //Fit a model to Dataframe, data
            var model = estimator.fit(data);
            Console.WriteLine(model);

            //Send stored model in PMML format to output file
            var pmmlFile = PATH + “PMML/clustering-kmeans.xml”;
            model.toPMML(pmmlFile);

            // model info output, based on training data
            Console.WriteLine(model.ClusterCenters + "\n");
            Console.WriteLine(model.InterClusterDistances + "\n");
            Console.WriteLine(model.RandomCentersSummary + "\n");

            //training or new data
            //predicted cluster labels
            Console.WriteLine(model.predict(data) + "\n");
            //average distances within clusters
            Console.WriteLine(model.ClustersSummary(data) + "\n");
            //record-to-cluster distances
            Console.WriteLine(model.getRecordToClusterDistances(data) +
            "\n");
            //metrics
            Console.WriteLine(model.FittingMetrics + “\n”);

            Util.free(ref estimator);
            Util.free(ref model);
        }
        catch (XLMiner.Exception ex)
        {
            Console.WriteLine("Exception has occurred at
              <Clustering.KMeans>:\n" + ex.Message);
            return 1;
        }
        return 0;
    }
```

Hierarchical Clustering

As its name implies, hierarchical clustering builds a tree graph which clusters more closely related data points into nodes of the graph. Hierarchical clustering algorithms can start by considering the data points as one large cluster which is divided again and again, or they can start by considering each data point as an individual node and then iteratively collecting together the most closely related nodes. The former method is referred to as divisive and the latter as agglomerative.

Pros: The tree graph into which hierarchical clustering organizes data makes relationships among the data easy to visualize. In contrast with the K-means methods, hierarchical clustering does not demand the specification of the number of clusters and can yield as few or as many clusters as approximate the desired result.

Cons: Hierarchical clustering does not scale well and is sometimes not practical for larg data sets.

The example below uses Hierarchical Clustering applied to the Hald dataset. This code creates a new Hierarchical Clustering estimator, sets several parameters, fits a model to the data, then creates a dendrogram using the results.

Hierarchical clustering example code

```
public static int Hierarchical()
{
    try
    {
        var data = Reader.textFile(PATH + "cluster.txt");

        //create new estimator
        var estimator = new Hierarchical.Estimator()
        { //set hierarchical clustering options
            Linkage = Xlminer.Clustering.Hierarchical.
            LinkageType.SINGLE_LINKAGE,
            Dissimilarity = Xlminer.DissimilarityType.EUCLIDEAN,
            InputDataType = Xlminer.Clustering.Hierarchical
            InputType.RAW_DATA
        };
        Console.WriteLine(estimator);

        //fit the model to the data DataFrame
        var model = estimator.fit(data);

        //set two model options
        model.NumClusters = 3;
        model.NumDendrogramLeaves = 5;

        Console.WriteLine(model);

        // score new data
        Console.WriteLine(model.predict());

        //create a dendrogram using the results from fitted model
        using (var dendrogram = model.dendrogram())
        {
            Console.WriteLine(dendrogram);
            Console.WriteLine(dendrogram.getDendrogramLeafMembers(0));
        }

        //Free memory
        Util.free(ref estimator);
        Util.free(ref model);
    }
    catch (XLMiner.Exception ex)
    {
        Console.WriteLine("Exception has occurred at
            <Clustering.Hierarchical>:\n" + ex.Message);
        return 1;
    }

    return 0;
}
```

Time Series Analysis

Introduction

XLMiner SDK supports the analysis and forecasting of data sets that contain observations generated sequentially through partitioning, autocorrelations, ARIMA models, and smoothing techniques. It is impossible to imagine a serious business analysis that does not include some examination of a time series dataset. Retail stores must anticipate how many winter coats should be on the shelves long before the weather turns cold and airlines must be able to allocate resources to handle changing demand during peak vacation months.

Time series analysis faces some special problems not found in other areas of data mining such as the possible effects of seasonality (think bathing suit sales every May). Individual values in a time series are not independent; there is a strong chance that today's values are correlated with past values.

XLMiner SDK includes the most popular techniques for performing a time series analysis including autocorrelation techniques, smoothing methods (exponential, double exponential, moving average and Holt Winters) and Box-Jenkins (ARIMA) methods, with and without seasonality.

Smoothing Methods

Probably the simplest of the techniques brought to bear on time series datasets are the techniques of smoothing. Random, or apparently random, fluctuations are present and may obscure trends or seasonal changes important to the analyst. XLMiner supports industry-standard methods of smoothing, which are implemented in SDK's Smoothing classes.

These include:

- Moving averages
- Exponential smoothing
- Double exponential smoothing
- Holt-winters: Additive, Multiplicative or No Trend

Autocorrelation Techniques

As its name suggests, autocorrelation techniques look for correlation between values representing different points in time. Most modern time series analyses include both moving average smoothing techniques and autocorrelation. One of the most commonly applied time series analysis techniques is ARIMA, which not only uses both moving averages and autocorrelation, but also attempts to take into account the fact that the underlying probability distribution of the data may itself change over time.

- **Autocovariance (ACV)** is the covariance of a variable with previous values of itself measured at some specific time lag.
- **Autocorrelation (ACF)** is a normalized version of autocovariance, that is, with the values adjusted to a mean of zero and a variance of one. When determining if an autocorrelation exists, the original time series is compared to the "lagged" series. This lagged series is simply the original series moved one time-period forward ahead (x_n vs x_{n+1}). Suppose there are five time-based observations: 10, 20, 30, 40, and 50. When lag = 1, the original series is moved forward one time-period. When lag = 2, the original series is moved forward two time-periods.

Day	Observed Value	Lag-1	Lag-2
1	10		

2	20	10	
3	30	20	10
4	40	30	20
5	50	40	30

Autocorrelation is computed according to the formula

$$r_k = \frac{\sum_{i=k+1}^{n}(Y_t - \bar{Y})(Y_{t-k} - \bar{Y})}{\sum_{i=1}^{n}(Y_t - \bar{Y})^2}$$

where, k = 0, 1, 2, …., n

where, Y_t is the Observed Value at time t, $\bar{Y}$ is the mean of the Observed Values, and Y_{t-k} is the value for Lag-k.

For example, using the values above, the autocorrelation for Lag-1 and Lag-2 is calculated as follows

$\bar{Y} = (10 + 20 + 30 + 40 + 50) / 5 = 30$

$r_1 = ((20 – 30) * (10 - 30) + (30 - 30) * (20 - 30) + (40 - 30) * (30 - 30) + (50 – 30) * (40 – 30)) / ((10 – 30)^2 + (20 - 30)^2 + (30 – 30)^2 + (40 – 30)^2 + (50 – 30)^2) = 0.4$

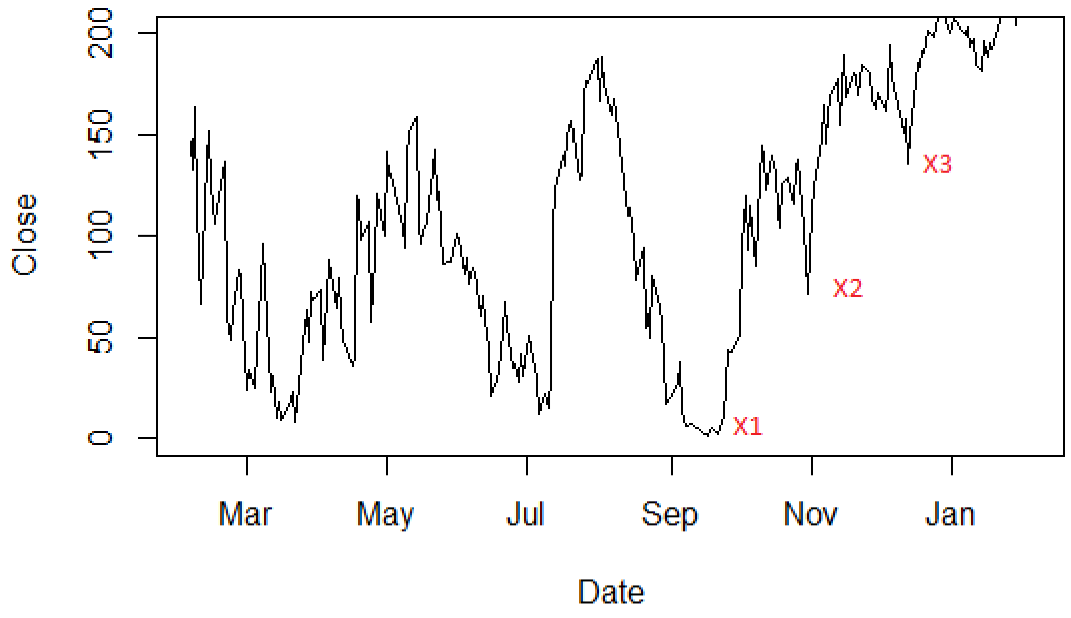
$r_2 = ((30 – 30) * (10 – 30) + (40 – 30) * (20 – 30) + (50 – 30) * (30 – 30)) / (((10 – 30)^2 + (20 - 30)^2 + (30 – 30)^2 + (40 – 30)^2 + (50 – 30)^2) = -0.1$

- **Partial Autocorrelation Function (PACF)** - This technique computes and plots the partial autocorrelations between the original series and the lags. PACF eliminates all linear dependence in the time series beyond the specified lag.
- **ARIMA** (Autoregressive Integrated Moving Average) model is a regression-type model that includes autocorrelation. The basic assumption in estimating the ARIMA coefficients is that the data is stationary (i.e., the trend or seasonality cannot affect the variance). This is generally not true. To achieve the stationary data, XLMiner applies ordinary or seasonal “differencing” (or both).

After XLMiner fits the model, various results are available. The quality of the model is evaluated by comparing the time plot of the actual values with the forecasted values. If both curves are close, the model can be assumed a good fit. The model should expose any trends and seasonality, if any exist. If the residuals are random, the model is a good fit; however, if the residuals exhibit a trend, the model should be refined. Fitting an ARIMA model with parameters (0, 1, 1) will give the same results as exponential smoothing. Fitting an ARIMA model with parameters (0, 2, 2) provides the same results as double-exponential smoothing.

Time Series Example

This example uses stock closing prices to illustrate a time series analysis. A plot of the data illustrates many common time series features. Of course, quantitative analysis is always required when making a decision, but a clear plot can help guide your thinking. When considering the plot above, we would not be at all surprised if there were a statistical correlation between, say, points X1, X2, and X3. The distance along the horizontal time axis between these points is the *lag*. Calculating correlations between data points and other data points differing by some lag is how our graphical impressions begin to become quantitatively confirmed.



Plot of the data in close.txt, the sample data for the XLMiner time series example.

Notice also that the three labelled points seem to be part of an upward trend. A critical part of time series analysis is the isolation and separation of periodic lagged values from trends. Not shown in the above plot is seasonality, in which periodic changes are related to the seasons or time of year.

Time series analysis usually makes use of a very general technique known as *smoothing*. Virtually all real-world data is noisy; smoothing, as its name implies, smooths out short-term noise that might be obscuring important trends.

The distinction between autocorrelation and partial autocorrelation is not always clear. We might, for example, expect a strong correlation between today's stock market data and that from yesterday. We would expect that there would be a correlation between today's data and data from two days ago. But is there some sort of two-day cycle? If correlation is seen for a two-day lag, how much of that correlation comes from the simple fact that today is correlated with yesterday, and yesterday was correlated with the day before yesterday. Autocorrelation merely calculates the statistical correlation between points separated by some lag. The partial correlation method, would, in this example, eliminate the trivial correlation. Whatever correlation is left may be attributable to some actual two-day cycle.

Time Series Example

The following C# example illustrates how to thread together various XLMiner SDK objects to fit a model to time series data using Autocovariance, Autocorrelation, Partial Autocorrelation, and ARIMA. To run a similar example, compile and run the project, **Test.csproj** located in …\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\C#. See the function *TimeSeriesAnalysis* defined in the class, *TimeSeries*.

This time series example:

- Reads data from close.txt, and populates a DataFrame with the data
- Calculates the autocovariance
- Calculates the autocorrelation
- Calculates the partial autocorrelation
- Uses ARIMA to fit a model using the training partition
- Reports the number of iterations and -2Log-Likelihood

- Reports residuals and their standard deviation
- Generates a forecast using the fitted model

Data Preparation

1) The example starts by creating an XLMiner DataFrame from the data source of interest. In the case of time series analysis, however, we want to use a very simple data structure consisting of a single column of values with row names containing the dates or timestamps. The *XLMiner.Convert* class has a method specifically for this purpose.

   As in previous chapters, this example uses the Dataset constructor to partition the dataset, *data*, into two partitions: a training partition containing 60% of the records and a validation partition containing 40% of the records. (See line 120 in Test.cs for an example of the constructor.)

   `XLMiner.Convert.toTimeSeries` creates a DataFrame with a single column with dates as rows.

   Creating DataFrame

```
var data = XLMiner.Convert.toTimeSeries(Reader.textFile(PATH +
"close.txt"));
```

Building the Model

2) *XLMiner.TimeSeries.Estimator.difference()* takes time series data and applies the lag operator to it. For more information on lag operator, see *Introduction to Time Series and Forecasting* 2nd Edition by Peter Brockwell and Richard Davis.[1]

   Applying Lag Operator to Time Series Data

```
int diff = 1, seasonalDiff = 1, period = 12;

var differencedData = XLMiner.TimeSeries.Estimator.difference(data, diff,
seasonalDiff, period);
```

3) Calculates the autocovariance of the time series using the minimum and maximum lags.

   Calculating Autocovariance

```
int minLag = 4;

int maxLag = 10;

var acvf = XLMiner.TimeSeries.Estimator.autocovariance(data, minLag,
maxLag);
```

4) Calculates the autocorrelation of the DataFrame, data, using the minimum and maximum lags.

   Calculating Autocorrelation

```
var acf = XLMiner.TimeSeries.Estimator.autocorrelation(data, minLag,
maxLag);
```

5) Calculates the partial auto-correlation of the DataFrame, data, using the minimum and maximum lags.

[1] Brockwell, Peter J, and Richard A. Davis. Introduction to Time Series and Forecasting, Second Edition. New York: Springer-Verlag, 2002.

Calculating Partial Auto-correlation

```
var pacf = XLMiner.TimeSeries.Estimator.partialAutocorrelation(data, minLag,
maxLag);
```

Construct a new TimeSeries estimator passing the required parameters for the ARIMA model: p = 2, d = 1, q = 2, P = 1, D = 1, Q = 1 and period = 12.

Construct a New Time Series Estimator

```
XLMiner.TimeSeries.Estimator estimator = new
XLMiner.TimeSeries.Estimator()
   {
      AutoRegressiveOrder = 2,
      MovingAverageOrder = 2,
      Difference = 1,
      Period = 12,
      SeasonalAutoRegressiveOrder = 1,
      SeasonalMovingAverageOrder = 1,
      SeasonalDifference = 1
   };
```

6) Fits the model using the TimeSeries estimator with the number of forecasts equal to 5 and the required confidence level equal to 95%.

   Fit the Model

```
var model = estimator.fit(data);
model.NumForecasts = 5;
model.ConfidenceLevel = 0.95;
```

7) Saves the model to PMML File Format.

   Save the Model to PMML Format

```
model.toPMML(PATH + “PMML/arima.xml”);
```

Examining and Applying the Results

8) Reports the number of iterations performed by the internal optimization methods, the -2Log-Likelihood and the statistics from the Ljung-Box Test.

   Display Number of Iterations

```
Console.WriteLine("Number of iterations: " + model.NumIterations);
Console.WriteLine("-2 * Log-likelihood: " + -2.0 * model.LogLikelihood);
Console.WriteLine(model.LjungBoxInfo);
```

9) Uses the fitted ARIMA model to score the time series data represented as a DataFrame in XLMiner SDK.

Score Time Series Data

```
var fitted = model.transform(data);
```

10) Constructs a new DataFrame, residuals, which holds the residuals calculated by subtracting the fitted values from the actual values.

    Compute the Residuals

```
var residuals = Util.subtract(data, fitted).setColName(0, "Residuals");
```

11) Computes the standard deviation of the vector of residuals.

    Compute the Standard Deviation of the Residuals

```
var stdDevResiduals =
Util.standardDeviation(XLMiner.Convert.toDoubleVector(residuals));
```

12) Creates a new Dataframe, *standardizedResiduals*, to hold the standardized residuals calculated as *residual/stdDevResiduals*.

    Create new DataFrame to Hold Standardized Residuals

```
//Creates new DF and initially fills the one column with "regular"
//residuals

DataFrame standardizedResiduals = new DataFrame(residuals);

//Column is transformed by dividing "regular" residuals by the standard
//deviation, resulting in a Dataframe with one column, which is the
//standardized residual

standardizedResiduals.setColName(0, "Std. Residuals");

for (var i = 0; i < residuals.getNumRows(); i++)

   standardizedResiduals[i,0].AsDouble =

   residuals[i,0].AsDouble/stdDevResiduals;
```

13) Displays the standard deviation of the vector or residuals.

    Display Standard Deviation of Residuals

```
Console.WriteLine("Residuals Standard Deviation: " + stdDevResiduals);
```

Prints the actual value, the fitted value, the residual and the standardized residuals side-by-side.

Print Results

```
Console.WriteLine(Util.tail(Util.colBind(new DataFrame[] { data, fitted,
residuals, standardizedResiduals }).setName("Fitted Values")));
```

14) Prints the actual data and fitted values side by side.

    Print Actual and Predicted Values

```
Console.WriteLine(model.errorMeasures(data, fitted));
```

15) Generates a forecast using the fitted ARIMA model.

Generate Forecast

```
var forecast = model.forecast(data).setName("Forecast Table”);
Console.WriteLine(forecast);
```

16) Generate the variance covariance matrix.

Generate Variance Covariance Matrix

```
var varCovar = model.getVarCovarMatrix();
Console.WriteLine(varCovar);
```

Association Rules

Introduction

Unlike many of the techniques used in XLMiner, association rules do not involve any sort of inference. Association rules are simply calculations of probability based on observations. For example, given that a customer has purchased a bag of potato chips, what is the probability they will also purchase a bottle of cola?

Retail stores can use association rules to plan shelf layout, for example making sure that one cannot pick up a bag of chips without seeing the tempting bottle of cola. Association analysis is ubiquitous on websites, where so-called "recommender" systems will suggest books of potential interest to a customer, based on probabilistic associations with a book the customer already has in their shopping cart.

The fact that there is no statistical inference involved in association rules does not mean that there is no interpretation involved or that care in interpretation need not be taken. For example, is the calculated probability based on a small or a large number of observations? This is referred to as the *support*. Is the probability interesting? (Some people actually refer to a characteristic they call *interestingness*!) For example, many people might stop at a small corner grocery store for a quart of milk on their way home. If many people buy milk, a conditional probability that includes milk purchase may not be of particular value. In other words, while people who buy a bag of candy may have a high probability of buying milk, there is also a high probability of buying milk among folks who do not buy candy. This concept relates to a calculated value called *lift*, discussed in the advanced topics section.

A Little Twist with the Input Data

Most of the data to which XLMiner methods will be applied are appropriately structured as DataFrames. Not so with the input data for association rules. For an association rules analysis, XLMiner expects input in a form representing individual transactions or events, but each transaction may involve a variable number of values. This is not surprising since some people buy ten books and others only two. It does, however, constrain the ways we can present the data to XLMiner.

The XLMiner binary format, similar to a DataFrame in many ways, is a rectangular matrix in which each row represents a transaction and each column represents, say, a product which may have been purchased. The format gets its name from the fact that the columns contain a one if the product was purchased and contain a zero if it was not. This format does not lend itself to an association analysis when there are large numbers of products which may be purchased, how many columns would Walmart require, and how many zeros would there be in such a matrix?

More useful in many real-world applications is the itemset format. In this format, each row represents a tab-separated list containing the individual items purchased by a single customer in one transaction.

An Association Rules Example

The following C# example illustrates how to use the Association Rules algorithm to recognize associations and correlations in a dataset. To run a similar example as the one shown below, compile and run the project, Test.csproj, located in …\Frontline Systems\Analytic Solver SDK\XLMinerSDK\Examples\C#. See the function Example.AffinityAnalysis.AssociationRules().

This example:

- Begins by reading data from a text file, AssociationsItemList.txt and populates a DataFrame with this data
- Constructs a model
- Extracts rules from the data.

1) Read data into DataFrame

   Read Data into DataFrame

   ```
   var data = Reader.itemsetToBinary(PATH + "AssociationsItemList.txt");
   ```

   Note: If the data contained in binary format, the data could be read in using:

   ```
   var data = Reader.textFile(PATH + "Associations.txt");
   ```

2) A new Association Rules estimator, is constructed and two parameters, *MinSupport* and *Method*, are set. Use *MinSupport* to specify the minimum number of transactions in which a particular itemset must appear for it to qualify for inclusion in an association rule here. The default value is 10% of the total number of rows. Method can be of type APRIORI or T_TREE.

   Construct Association Rules Estimator

   ```
   var estimator = new Association.Estimator()
   {
      MinSupport = 100.0/data.NumRows,
      Method = Association.MethodType.APRIORI
   };
   ```

3) Fit model to the data.

   Fit the Model

   ```
   var model = estimator.fit(data);
   ```

4) Set the option, MinConfidence, on the fitted model.

   Set an Option

   ```
   model.MinConfidence = 0.5;
   ```

5) Saved model to PMML file format.

   Save Model to PMML Format

   ```
   model.toPMML(PATH + "PMML/association-rules.xml");
   ```

6) Extract rules from the data.

   Extract Rules

   ```
   var rules = model.transform(data);
   ```

7) Use utility sorting function to sort by passing column names in descending order.

   Use Utility Sorting Function

   ```
   rules = Util.sort(rules, new string[] { "Lift-Ratio", "Confidence" }, new
   int[] {SortType.DESCENDING, SortType.DESCENDING });
   ```

8) Display the results.

Display Results

```
rules.Name = "Rules Sorted by Lift-Ratio then by Confidence";
Console.WriteLine("Number of transactions: " + data.NumRows);
Console.WriteLine("Number of items: " + data.NumCols);
Console.WriteLine("Number of rules: " + rules.NumRows + "\n");
Console.WriteLine(Util.head(rules, 100));
```

9) Free memory.

Free Memory

```
Util.free(ref data);
Util.free(ref model);
Util.free(ref rules);
```

Advanced Topics: Association Rules

Support

Support is the fraction of sales in which two items were purchased together

Support = 0.05 implies N(A,B)/N = 0.05

Confidence

Confidence provides a measure of the strength of the probabilistic relation between two events. For example, if 95% of the people who purchased A also purchased B, we would have a confidence of 0.95.

Confidence = 0.95% implies N(A,B)/N(A) =.95

Notice that support is not involved in this calculation. In this sense, the term "confidence" must not be confused with "confidence" in the statistical inference sense. In association rules, a high confidence might arise from a small number of observations.

Expected Confidence

Expected confidence is the number of times we would expect two items to be purchased together simply based on random chance. For example, if half of all customers purchased A and half of all customers purchased B, we would expect that 25% of customers would purchase both A and B solely by chance.

Lift

Lift is an indicator of how useful the confidence might be. It is the ratio between the actual number of pairings of A and B and the number of pairings to be expected from chance alone. That is, lift is the observed confidence divided by the expected confidence.

lift = (sup(A U B)) / (sup(A)*sup(B))

XLMiner Text Mining

Introduction

Text mining is the practice of automated analysis of one document, or a collection of documents (corpus), and extracting non-trivial information from those documents. In addition, Text Mining usually involves the process of transforming unstructured textual data into structured representation by analyzing the patterns derived from the text. The results can be analyzed to discover interesting knowledge, some of which would only be found by a human carefully reading and analyzing the text. Typical widely-used tasks of Text Mining include but are not limited to Automatic Text Classification/Categorization, Topic Extraction, Concept Extraction, Documents/Terms Clustering, Sentiment Analysis, Frequency-based Analysis and many more. Some of these tasks could not be completed by a human, which makes Text Mining a particularly useful and applicable tool in modern Data Science.

XLMiner SDK takes an integrated approach to text mining as it does not totally separate analysis of unstructured data from traditional data mining techniques applicable for structured information. While XLMiner SDK is a very powerful tool for analyzing text only, it also offers automated treatment of mixed data, i.e. combination of multiple unstructured and structured fields. This is a particularly useful feature that has many real-world applications, such as analyzing maintenance reports, evaluation forms, insurance claims, etc. XLMiner SDK uses the "bag of words" model – the simplified representation of text, where the precise grammatical structure of text and exact word order is disregarded. Instead, syntactic, frequency-based information is preserved and is used for text representation. Although such assumptions might be harmful for some specific applications of Natural Language Processing (NLP), it has been proven to work very well for applications such as Text Categorization, Concept Extraction and others, which are the particular areas addressed by XLMiner SDK's capabilities. It has been shown in many theoretical/empirical studies that syntactic similarity often implies semantic similarity. One way to access syntactic relationships is to represent text in terms of Generalized Vector Space Model (GVSP). Advantage of such representation is a meaningful mapping of text to the numeric space, the disadvantage is that some semantic elements, e.g. order of words, are lost (recall the bag-of-words assumption).

Input to Text Miner (the Text Mining tool within XLMiner SDK) could be of two main types – few relatively large documents (e.g. several books) or relatively large number of smaller documents (e.g. collection of emails, news articles, product reviews, comments, tweets, Facebook posts, etc.). While XLMiner SDK is capable of analyzing large text documents, it is particularly effective for large corpuses of relatively small documents. Obviously, this functionality has limitless number of applications – for instance, email spam detection, topic extraction in articles, automatic rerouting of correspondence, sentiment analysis of product reviews and many more. The input for text miner is a dataframe, with at least one column that contains free-form text (or file paths to documents in a file system containing free-form text), and, optionally, other columns that contain traditional structured data.

The output for the text mining is a set of reports that contain general explorative information about the collection of documents and structured representations of text (free-form text columns are expanded to a set of new columns with numeric representation. The new columns will each correspond to either (i) a single term (word) found in the "corpus" of documents, or, if requested, (ii) a concept extracted from the corpus through Latent Semantic Indexing (LSI, also called LSA or Latent Semantic Analysis). For more on Latent Semantic Analysis, see the example below.

The statistics produced and displayed in the Term-Document Matrix contain basic information on the frequency of terms appearing in the document collection. With this information we can "rank" the significance or importance of these terms relative to the collection and particular document.

Text Mining Example Code

In the example below, XLMiner SDK Text Miner will be applied to the DataFrame constructed in *TMData()* function in the subsequent examples below.

Construction of TMData Function

```
      public static DataFrame TMData()
        {
          try
           {
             DataFrame df = new DataFrame("Romeo");

             String[] vec = new String[]
             {
              "Romeo and Juliet.",
              "Juliet: O happy dagger!",
              "'Romeo died by dagger.'",
              "'Live free or die', that's the New-Hampshire's motto.",
              "'Did you know, New - Hampshire is in New - England.'"
              };

              df.append(vec);

             return df;
            }
             catch (XLMiner.Exception ex)
             {
              Console.WriteLine("Exception has occurred at
              <TextMining.tmData>:\n"
              + ex.Message);
              return null;
             }
 }
```

Term Document Text Mining Example

In the example below, you will learn how to use Text Miner in XLMiner SDK to process/analyze approximately 1000 text files and use the results for automatic topic categorization.

Processing/Analyzing text files for automatic topic categorization

```
      public static int Vectorizing()
       {
         try
           {
             //Create new dataframe, romeoCorpus, from TMData dataframe (created
//in example above
 DataFrame romeoCorpus = TMData();

 //Create TFIDF estimator
 var estimator = new TfIdf.Estimator()
 {
   //Set estimator options
   Preprocessing =
     TM.Preprocessing.REMOVE_STOPWORDS |
     TM.Preprocessing.NORMALIZE_CASE |
     TM.Preprocessing.STEM |
     TM.Preprocessing.NORMALIZE_URL |
     TM.Preprocessing.NORMALIZE_EMAIL |
     TM.Preprocessing.NORMALIZE_NUMBER |
     TM.Preprocessing.NORMALIZE_MONEY |
     TM.Preprocessing.REMOVE_HTML_TAGS,
     MaxVocabulary = 5,
     MaxTermLength = 10,
     MinStemmedTermLength = 2,
     MinDocumentFrequency = 5,
     MaxDocumentFrequency = 95,

     // If these options are used, text appearing before the first
     //occurrence of the Start Phrase will be disregarded and
     //similarly, text appearing after End Phrase (if used) will be
     //disregarded
     //StartPhrase = "startPhrase",
     //EndPhrase = "endPhrase",

     // These next 4 lines of code allow the replacement or removal
     //of nonsensical terms such as HTML tags, URLs, Email addresses,
     //etc. from the document collection.  For example, any URL
     //tokens will be replaced with “urltoken”
     UrlToken = "urltoken",
     EmailToken = "emailtoken",
     NumberToken = "numbertoken",
     MoneyToken = "moneyamountterm",

     //Specify terms to be excluded
```

```
      //ExclusionTerms = new string[] { "exclusionTerm1",
      //"exclusionTerm2",
      //"exclusionTerm3" },

      // When used, all other terms will be disregarded except those
      //added here.  i.e. to mine each document for a specific word
      //such as “dagger”
      //ExclusiveInclusionTerms = new string[] {
      //"exclusiveInclusionTerm1",
      //"exclusiveInclusionTerm2" },
      //Specify additional stopwords
      //StopwordsExtraTerms = new string[] { "stopword1" }
   };

//Combine synonyms i.e. replace “coupe”, “sedan”, “convertible” with
//“auto”
//estimator.setSynonyms("rootTerm1", new string[] { "synonym1",
//"synonym2" });
//estimator.setSynonyms("rootTerm2", new string[] { "synonym1",
//"synonym2" });

//Combine phrases i.e. replace “station wagon” with “wagon”
//estimator.setPhraseReplacement("phrase1", "phraseReplacement");
//estimator.setPhraseReplacement("phrase2", "phraseReplacement");

Console.WriteLine(estimator);
Console.WriteLine("TF-IDF Estimator: chosen preprocessing -- " +
estimator.Preprocessing);
Console.WriteLine();

//Fit the model
var model = estimator.fit(romeoCorpus);

//set term document matrix scheme options on “model”
model.WeightingSchemeTerm = TM.WeightingScheme.Term.LOGARITHMIC;
model.WeightingSchemeDocument = TM.WeightingScheme.Document.INVERSE;
model.WeightingSchemeNormalization =
TM.WeightingScheme.Normalization.NONE;
```

```
            Console.WriteLine(model);

            //Save model to PMML file format
            var pmmlFile = PATH + "PMML/text-mining-tfidf.xml";
            model.toPMML(pmmlFile);

            //display output to the console
            Console.WriteLine(model.DetailedVocabulary);
            Console.WriteLine(model.TermCountInfo);
            Console.WriteLine(model.DocInfo);
            Console.WriteLine(model.getTopTermsInfo(5, 3));

            //Perform transformation on remeoCorpus dataframe
            using (var tfRomeo = model.transform(romeoCorpus))
            Console.WriteLine(tfRomeo);

            //Free memory
            Util.free(ref estimator);
            Util.free(ref model);
          }
            catch (XLMiner.Exception ex)
          {
            Console.WriteLine("Exception has occurred at
            <TextMining.Vectorizing>:\n" +
            ex.Message);
            return 1;
          }
      return 0;
     }
```

Preprocessing Parameter Descriptions

Analyze Specified Terms Only

`ExclusiveInclusionTerms = new string[] { "exclInclTerm1","exclInclTerm2" }`

Using the keyword, *ExclusiveInclusionTerms*, terms to be considered for text mining can be added and removed. All other terms will be disregarded. For example, if wanting to mine each document for a specific part name such as "dagger", use: `ExclusiveInclusionTerms = new string[] { "dagger" }`.

Start Phrase/End Phrase

```
StartPhrase = "startPhrase"

EndPhrase = "endPhrase"
```

If used, text appearing before the first occurrence of the Start Phrase will be disregarded and similarly, text appearing after End Phrase (if used) will be disregarded. For example, if text mining the transcripts from a Live Chat service, you would not be particularly interested in any text appearing before the heading “Chat Transcript” or after the heading “End of Chat Transcript”. Thus, you would enter “Chat Transcript” into the Start Phrase field and “End of Chat Transcript” into the End Phrase field.

Stopword removal

TM.Preprocessing.REMOVE_STOPWORDS

Set this preprocessing flag to remove over 300 commonly used words/terms (such as a, to, the, and, etc.) from the document collection during preprocessing.

Exclusion list

```
ExclusionTerms = new string[] { "exclusionTerm1", "exclusionTerm2"}
```

Terms entered into the Exclusion list will be removed from the document collection. This is beneficial if all or a large number of documents in the collection contain the same terms, for example, “from”, “to”, “subject” in a collection of emails. If all documents contain the same terms, including these words in the analysis will not provide any benefit and could bias the analysis.

Synonym Reduction

```
estimator.setSynonyms("rootTerm1", new string[] { "synonym1", "synonym2" })

estimator.setSynonyms("rootTerm2", new string[] { "synonym1", "synonym2" })
```

Use *setSynonyms* to replace synonyms such as “car”, “automobile”, “convertible”, “vehicle”, “sedan”, “coupe”, “subcompact”, and “jeep” with “auto”. During pre-processing, XLMiner SDK will replace the terms “car”, “automobile”, “convertible”, “vehicle”, “sedan”, “coupe”, “subcompact” and “jeep” with the term “auto”.

Phrase Reduction

estimator.setPhraseReplacement("phrase1", "phraseReplacement")

estimator.setPhraseReplacement("phrase2", "phraseReplacement")

XLMiner SDK also allows the combining of words into phrases that indicate a singular meaning such as “station wagon” which refers to a specific type of car rather than two distinct tokens – station and wagon.

Maximum vocabulary size

```
MaxVocabulary = 5
```

Use this keyword to reduce the number of terms in the final vocabulary to the most frequently occurring in the collection. The default is “1000”.

Perform stemming

TM.Preprocessing.STEM

Stemming is the practice of stripping words down to their “stems” or “roots”, for example, stemming terms such as “argue”, “argued”, “argues”, “arguing”, and “argus” would result in the stem “argu”. However, “argument” and “arguments” would stem to “argument”. The stemming algorithm utilized in XLMiner SDK is “smart” in the sense that while “running” would be stemmed to “run”, “runner” would not. XLMiner SDK uses the Porter Stemmer 2 algorithm for the English Language. For more information on this algorithm, please see the Webpage: http://tartarus.org/martin/PorterStemmer/

Normalize case

TM.Preprocessing.NORMALIZE_CASE

When this flag is turned on, XLMiner SDK converts all text to a consistent (lower) case, so that Term, term, TERM, etc. are all normalized to a single token "term" before any processing, rather than creating three independent tokens with different case. This simple method can dramatically affect the frequency distributions of the corpus, leading to biased results.

Term Normalization

Term normalization flags allow users to replace or remove nonsensical terms such as HTML tags, URLs, Email addresses, etc. from the document collection.

Note: It's possible to remove normalized terms completely by including the normalized term (for example, "emailtoken") in the Exclusion list.

Minimum Stemmed Term Length

```
MinStemmedTermLength = 2
```

If stemming reduced a term's length to 2 or less characters, Text Miner will disregard the term. This option is selected by default.

Remove HTML tags

TM.Preprocessing.REMOVE_HTML_TAGS

If this option is turned on, HTML tags will be removed from the document collection. HTML tags and text contained inside these tags contain technical, computer-generated information that is not typically relevant to the goal of the text mining application. This option is not turned on by default.

Normalize URL's

TM.Preprocessing.NORMALIZE_URL

If this option is turned on, URLs appearing in the document collection will be replaced with the term, "urltoken". URLs do not normally add any meaning, but it is sometimes interesting to know how many URLs are included in a document. This option is not selected by default.

Normalize email addresses

TM.Preprocessing.NORMALIZE_EMAIL

If this option is turned on, email addresses appearing in the document collection will be replaced with the term, "emailtoken". This option is not selected by default.

Normalize numbers

TM.Preprocessing.NORMALIZE_NUMBER

If this option is turned on, numbers appearing in the document collection will be replaced with the term, "numbertoken". This option is not selected by default.

Normalize monetary amounts

TM.Preprocessing.NORMALIZE_MONEY

If this option is turned on, monetary amounts will be substituted with the term, "moneytoken". This option is not selected by default.

Minimum Document Frequency

```
MinDocumentFrequency = 5
```

If specified, Text Miner will remove terms that appear in less than the percentage of documents specified. For most text mining applications, rarely occurring terms do not typically offer any added information or meaning to the document in relation to the collection. The default percentage is 2%.

Maximum Document Frequency

```
MaxDocumentFrequency = 95
```

If specified, Text Miner will remove terms that appear in more than the percentage of documents specified. For many text mining applications, the goal is identifying terms that have discriminative power or terms that will differentiate between a number of documents. The default percentage is 98%.

Maximum term length

```
MaxTermLength = 10
```

If selected, Text Miner will remove terms that contain a set number of characters. This option can be extremely useful for removing some parts of text which are not actual English words, for example, URLs or computer-generated tokens, or to exclude very rare terms such as Latin species or disease names, i.e. Pneumonoultramicroscopicsilicovolcanoconiosis.

Latent Semantic Analysis

As discussed above, the output for Text Miner is a set of reports that contain general explorative information about the collection of documents and structured representations of text (free-form text columns are expanded to a set of new columns with numeric representation. The new columns will each correspond to either (i) a single term (word) found in the “corpus” of documents, or, if requested, (ii) a concept extracted from the corpus through Latent Semantic Indexing (LSI, also called LSA or Latent Semantic Analysis).

Latent Semantic Indexing uses singular value decomposition (SVD) to map the terms and documents into a common space to find patterns and relationships. For example: if we inspected our document collection, we might find that each time the term “dagger” appeared in a document, the document also included the terms “sharp” and “shiny”. Or each time the term “Juliet” appeared in a document, the terms “beautiful” and “lovely” also appeared. However, there is no detectable pattern regarding the use of the terms “dagger” and “Juliet”. Documents including “dagger” might not include “Juliet” and documents including “Juliet” might not include “dagger”. Our four terms, lovely, beautiful, sharp, and shiny describe two different issues: a knife and an attractive lady. Latent Semantic Indexing will attempt to 1. Distinguish between these two different topics, 2. Identify the documents that deal with daggers, beautiful women or both and 3. Map the terms into a common semantic space using singular value decomposition (SVD). SVD is a tool used by Text Miner to extract concepts that explain the main dimensions of meaning of the documents in the collection.

Processing/Analyzing text files using Latent Semantic Analysis

```
      public static int LatentSemanticAnalysis()
        {
          try
             {
               DataFrame tdmRomeo;

                //Using romeoCorpus, create a new TFIDF estimator, fit the
               //model, transform the romeoCorpus dataset (apply model to),
               //then assigns resultant dataset to tdmRomeo
                using (DataFrame romeoCorpus = TMData())
               {
```

```
 Console.WriteLine(romeoCorpus);
 tdmRomeo = (new TfIdf.Estimator()).
                  fit(romeoCorpus).
            transform(romeoCorpus);
}

Console.WriteLine(tdmRomeo);

//Create a new LSA estimator
var estimator = new LSA.Estimator()
  {
   //Set various options – see below for explanations
   MaxNumConcepts = 4,
   //MinPercentExplained = 50,
   ComputeTermImportance = true,
   ComputeConceptImportance = true
  };
  Console.WriteLine(estimator);

  //Fit the model on tdmRomeo
  var model = estimator.fit(tdmRomeo);
  Console.WriteLine(model);

  //Save the model to PMML File format
  var pmmlFile = PATH + "PMML/text-mining-lsa.xml";
  model.toPMML(pmmlFile);

  //Print Concept Importance, Term Importance, and Term
  //Concept Matrices to the console
  Console.WriteLine(model.ConceptImportance);
  Console.WriteLine(model.TermImportance);
  Console.WriteLine(model.getTermConceptMatrix(tdmRomeo));

  //score tdmRomeo dataframe using fitted model
  using (var tfRomeo = model.transform(tdmRomeo))
  Console.WriteLine(tfRomeo);
```

```
                //Free memory
                Util.free(ref estimator);
                Util.free(ref model);
            }
                catch (XLMiner.Exception ex)
                {
                    Console.WriteLine("Exception has occurred at
                    <TextMining.LatentSemanticAnalysis>:\n" + ex.Message);
                    return 1;
                }

                return 0;
            }
}
```

Latent Semantic Analysis Parameter Descriptions

This section describes the options available for the latent semantic analysis estimator.

Maximum number of concepts or Minimum percentage explained

`MinPercentExplained = 50`

If the option *MinPercentExplained* is specified, XLMiner SDK will identify the concepts with singular values that, when taken together, sum to the minimum percentage explained, 75% is the default.

`MaxNumConcepts = 4`

If *MaxNumConcepts* is specified, XLMiner SDK will identify the top number of concepts according to the value entered here. The default is 1 concept.

Term Importance

ComputeTermImportance = true

Set this option to True to produce the Term Importance matrix. This table displays each term along with its Importance as calculated by singular value decomposition. This option is not selected by default.

Concept Importance

ComputeConceptImportance = true

Set this option to True to produce the Concept Importance matrix. This table displays the total number of concepts extracted, the Singular Value for each, the Cumulative Singular Value and the % of Singular Value explained which is used when Minimum percentage explained is selected for Concept Extraction – Latent Semantic Indexing on the Representation tab. This option is not selected by default.

XLMiner and Big Data

Big Data

“Big data” has sometimes been characterized by what are known as the “Three V’s”, Volume, Velocity, and Variety. Volume, of course, is what gave big data its name; the sheer quantity of data prevents the more conventional methods of data storage from being effective. The Hadoop distributed file system is far and away the most commonly used platform for dealing with big data volumes. “Velocity” refers to the speed at which the data arrives, and the speed at which it may need to be evaluated. Few conventional transactional databases must deal with new data as fast as, say, a big data platform examining Twitter feeds. “Variety”, of course, refers to the fact that unlike traditional relational data, big data presents itself in a wide variety of formats and structures, not all of them designed to make life easier for the analysts.

Some technology commentators have felt the need to add yet two more V’s, Veracity and Value. Data within big data stores does not always share the same degree of reliability as traditional relational data. Value, of course, is the small flakes of gold we can obtain by panning the large volumes of sand and silt.

Apache Spark

XLMiner’s big data abilities build upon the computational strength of Apache Spark. Spark is a distributed computational environment that is regarded by many as superior to the minimal map-reduce capabilities built into the basic Hadoop architecture. If you wish to use XLMiner with your in-house Hadoop system, you must be running Spark as a part of your big data system, and it must be possible for XLMiner to submit tasks to Spark using the Spark Job Server.

Connecting to an Apache Spark Cluster

The XLMiner SDK software communicates over the network with a Frontline Systems supplied, server-side software package that runs on one of the computers in the Spark cluster. The first step in connecting XLMiner SDK to your organization's own Apache Spark cluster is to contact Frontline Systems Sales and Technical Support at 775-831-0300.

After the server-side software package is installed, the proper entries for the cluster options can be used. For university instructors teaching courses in business analytics to MBA and undergraduate business students, using methods such as data mining, optimization and simulation, who would like to give their students hands-on experience with the use of Big Data in decision-making, without a need for programming expertise or other "data science" preparation, Frontline Systems operates an Apache Spark cluster "in the cloud" on Amazon Web Services, preloaded with a set of interesting, publicly available Big Data datasets (such as the Airline dataset illustrated in this chapter), and sample exercises and case studies using the datasets, that we can make available at a nominal cost for student use. For further information about this option, please contact Frontline Systems Academic Sales and Support at 775-831-0300 or academic@solver.com.

Storage Sources and Data Formats

XLMiner SDK can process data from Hadoop Distributed File System (HDFS), local file systems that are visible to Spark cluster, and Amazon S3. Performance is best with HDFS, and it is recommended that you load data from a local file system or Amazon S3 into HDFS. If the local file system is used, the data must be accessible at the same path on all Spark workers, either via a network path, or because it was copied to the same location on all workers.

At present, XLMiner SDK can process data in Apache Parquet and CSV (delimited text) formats. Performance is far better with Parquet, which stores data in a compressed, columnar representation; it is highly recommended that you convert CSV data to Parquet before you seek to sample or summarize the data

The Big Data Sampler

This example illustrates how to use the latest XLMiner SDK using Data stored across an Apache Spark compute cluster where the Frontline Systems access server is installed. By drawing a representative sample of Big Data from all the nodes in the cluster, Excel users can easily train data mining and text mining models directly on their desktops.

Sampling from Big Data

```
public static int Sampling()
{
  try
    {
     //create new Big Data Sampler, sampler
     XLMiner.BigData.Sampler sampler = new XLMiner.BigData.Sampler();

     // server settings for custom Apache Spark clusters
     // port for the Spark REST server must be 8090
     //sampler.SparkServer = <endpoint for Spark cluster>

     // -- if data source is HDFS or network-mounted file system
     //sampler.FileLocation = <file location URL - hdfs://...>

     // -- if data source is AWS S3
     //sampler.AWSS3 = true;
     //sampler.FileLocation = <file location URL - s3n://...>
     //sampler.AWSS3AccessKey = <AWS S3 access key>
     //sampler.AWSS3SecretKey = <AWS S3 access key>

     // -- if you have access to Frontline's cluster:
     // see available datasets preloaded on the cluster
     //var solverDatasets = sampler.solverDatasets();
     //Console.WriteLine(solverDatasets);

     // -- If Apache Parquet – which this example uses
     sampler.DataFormat = XLMiner.BigData.Format.PARQUET;

     // -- if delimited text
     //sampler.DataFormat = XLMiner.BigData.Format.CSV;
     //sampler.HeaderExist = true;
     //sampler.Delimiter = "\t";
```

```
// Set general Sampling options
sampler.TrackRowID = true;
sampler.WithReplacement = false;
sampler.RandomSeed = 12345;

// If using approximate sampling, set these options
sampler.SamplingType = XLMiner.BigData.Sampling.Type.APPROXIMATE;
sampler.SampleFraction = 0.01;

// -- if using exact sampling, set these options
//sampler.SamplingType = XLMiner.BigData.Sampling.Type.EXACT;
//sampler.SampleSize = 15;

// Infer schema from data source
var schema = sampler.inferSchema();
Console.WriteLine("Schema: [" + String.Join(",", schema) + "]");

// -- optionally set variables to be included in a sample
//sampler.SelectedVariables = schema;
// or
sampler.SelectedVariables = new string[] { "Var1", "Var2", "Var3"
};
// or if not set - all variables would be included in a sample

Console.WriteLine(sampler);

// -- submit Big Data job synchronously: get results immediately
//var sample = sampler.run();
//Console.WriteLine(sample);

// Submit Big Data job asynchronously: get Job ID for later
//retrieval
var jobID = sampler.submit();
Console.WriteLine("Job ID: " + jobID);

//... and later retrieve results
```

```
  var status = XLMiner.BigData.JobStatus.RUNNING;
  while (status != XLMiner.BigData.JobStatus.FAILED || status !=
  XLMiner.BigData.JobStatus.FINISHED)
  {
    status = sampler.jobStatus();
    if (status == XLMiner.BigData.JobStatus.FAILED || status ==
    XLMiner.BigData.JobStatus.JOB_ERROR)
    {
      Console.WriteLine("Job failed to complete");
      break;
    }
      else if (status == XLMiner.BigData.JobStatus.RUNNING)
    {
      Console.WriteLine("Job is still running. Waiting for 1
      millisecond...");
      System.Threading.Thread.Sleep(10);
    }
      else if (status == XLMiner.BigData.JobStatus.FINISHED)
    {
      Console.WriteLine("Job is completed...");
      var sample = sampler.results();
      Console.WriteLine(sample);
      break;
    }
   }

  // Get cluster info
  Console.WriteLine(sampler.clusterInfo());

  // Get duration info
  Console.WriteLine(sampler.durationInfo());

  // Get # rows in original BD source
  Console.WriteLine(sampler.numRows());
  }
catch (XLMiner.Exception ex)
         {
             Console.WriteLine("Exception has occurred at
                    <BigData.Sampling>:\n" + ex.Message);
                    return 1;
                }

          return 0;
}
```

Parameter Explanations

See below for explanation of parameters for Big Data Sampling.

Track record IDs

`sampler.TrackRowID = true`

If this option is set to True, data records in the resulting sample will carry the correct ordinal IDs that correspond to the original data records, so that records can be matched. Note: Selecting this option may significantly increase running time so it should be applied only when necessary.

Sample with Replacement

`sampler.WithReplacement = true`

When set to True, records in the dataset may be chosen for inclusion in the sample multiple times.

Random Seed

`sampler.RandomSeed = 12345`

If an integer value is passed for Random seed, XLMiner SDK will use this value to set the feature selection random number seed. Setting the random number seed to a nonzero value ensures that the same sequence of random numbers is used each time the dataset is chosen for sampling. The default value is “12345”. If left blank, the random number generator is initialized from the system clock, so the random sample would be represented by different records from run to run. If you need the results from successive samples to be strictly comparable, you should set the seed. To do this, type the desired number you want into the box. This option accepts positive integers with up to 10 digits.

Exact Sampling

`sampler.SamplingType = XLMiner.BigData.Sampling.Type.EXACT`

When this option is selected, XLMiner SDK will return a fixed – size sampled subset of data according to the setting for Desired Sample Size.

Desired Sample Size

`sampler.SampleSize = 15`

When Exact Sampling is used, enter the number of records to be included in the sample using this parameter.

Approximate Sampling

sampler.SamplingType = XLMiner.BigData.Sampling.Type.APPROXIMATE

When this option is selected, the size of the resultant sample will be determined by the value entered for Desired Sample Fraction. Approximate sampling is much faster than Exact Sampling. Usually, the resultant fraction is very close to the Desired Sample Fraction so this option should be preferred over exact sampling as often as possible. Even if the resultant sample slightly deviates from the desired size, this would be easy to correct in Excel.

Desired Sample Fraction

`sampler.SampleFraction = 0.01`

When Approximate Sampling is used, enter the expected size of the sample as a fraction of the dataset's size using this parameter.

If Sampling with Replacement is selected, the value for Desired Sample Fraction must be greater than 0. If Sampling without replacement (i.e. Sampling with Replacement is not selected), the Desired Sample Fraction becomes the probability that each element is chosen and, as a result, Desired Sample Fraction must be between 0 and 1.

The Big Data Summarizer

The Big Data Summarization feature in XLMiner SDK is useful for rapid extraction of key metrics contained in data, which can be immediately used by data analysts and decision makers. This feature provides similar functionality as standard SQL engines, but for the data, volume and complexity which extends far beyond your desktop or laptop computer. This tool is a great assistant for composing reports, constructing informative visualizations, building prescriptive and predictive models that can drive the directions of consequent analysis.

Using XLMiner SDK's Big Data Summarizer

```
public static int Summarizing()
{
  try
      {
       //Create new summarizer
       XLMiner.BigData.Summarizer summarizer = new
       XLMiner.BigData.Summarizer();

       // server settings for custom Apache Spark clusters
       // port for the Spark REST server must be 8090
       //summarizer.SparkServer = <endpoint for Spark cluster>

       // -- if data source is HDFS or network-mounted file system
       //summarizer.FileLocation = <file location URL - hdfs://...>

       // -- if data source is AWS S3
       //summarizer.AWSS3 = true;
       //summarizer.FileLocation = <file location URL - s3n://...>
       //summarizer.AWSS3AccessKey = <AWS S3 access key>
       //summarizer.AWSS3SecretKey = <AWS S3 access key>

       // -- if you have access to Frontline's cluster:
       // see available datasets preloaded on the cluster
       //var solverDatasets = summarizer.solverDatasets();
       //Console.WriteLine(solverDatasets);
```

```
// -- if Apache Parquet as in this example
summarizer.DataFormat = XLMiner.BigData.Format.PARQUET;

// -- if delimited text
//summarizer.DataFormat = XLMiner.BigData.Format.CSV;
//summarizer.HeaderExist = true;
//summarizer.Delimiter = "\t";

//General summarization options [Sum, Average, Standard
Deviation,
//Min, or Max]
summarizer.AggregationType =
XLMiner.BigData.Summarization.AggregationType.AVG;
summarizer.ComputeGroupCounts = true;

// infer schema from data source
var schema = summarizer.inferSchema();
Console.WriteLine("Schema: [" + String.Join(",", schema) + "]");

// -- optionally set variables to be included in a summary
//summarizer.SelectedVariables = schema;
// or
summarizer.SelectedVariables = new string[] { "Var1", "Var2",
"Var3" };
// or if not set - all variables would be included in a summary

// -- optionally set grouping variables
//summarizer.GroupingVariables = schema;
// or
summarizer.GroupingVariables = new string[] { "Var4", "Var5" };
// or if not set - there would be no grouping

Console.WriteLine(summarizer);

// -- submit Big Data job synchronously: get results immediately
//var summary = summarizer.run();
// or
```

```
//var summary = summarizer.summarizeAndCount();
//Console.WriteLine(summary);

// Submit Big Data job asynchronously: get Job ID for later
//retrieval
var jobID = summarizer.submit();
Console.WriteLine("Job ID: " + jobID);

//... and later retrieve results
var status = XLMiner.BigData.JobStatus.RUNNING;
while (status != XLMiner.BigData.JobStatus.FAILED || status !=
XLMiner.BigData.JobStatus.FINISHED)
{
  status = summarizer.jobStatus();
  if (status == XLMiner.BigData.JobStatus.FAILED || status ==
     XLMiner.BigData.JobStatus.JOB_ERROR)
     {
       Console.WriteLine("Job failed to complete");
       break;
     }
       else if (status == XLMiner.BigData.JobStatus.RUNNING)
     {
       Console.WriteLine("Job is still running. Waiting for 1
           millisecond...");
       System.Threading.Thread.Sleep(10);
      }
        else if (status == XLMiner.BigData.JobStatus.FINISHED)
      {
        Console.WriteLine("Job is completed...");
        var summary = summarizer.results();
        Console.WriteLine(summary);
        break;
      }
 }

 // Get cluster info
 Console.WriteLine(summarizer.clusterInfo());
```

```
                // Get duration info
                Console.WriteLine(summarizer.durationInfo());

                // Get # rows in original BD source
                Console.WriteLine(summarizer.numRows());
    }
            catch (XLMiner.Exception ex)
                    {
                        Console.WriteLine("Exception has occurred at
                        <BigData.Summarizing>:\n" + ex.Message);
                        return 1;
                    }

              return 0;
}
```

Glossary

Activation Function – Virtually all practical artificial neural networks require a nonlinear activation function if they are to work on any but the most trivial classification problems. Among the most commonly used activation functions are logistic sigmoid, the hyperbolic tangent (tanh), and the SOFTMAX function.

ANOVA – Analysis of variance. This ubiquitous technique for statistical hypothesis testing is built on the same mathematical foundation as multiple regression. While ANOVA itself is not a data mining technique, it is used in XLMiner to report the statistical summary and significance levels for XLMiner's linear regression mining models.

ARIMA – Autoregressive integrated moving average; one of the more common algorithms for the analysis of time series

Bagging – "Bagging", short for "bootstrap aggregating", is a technique most commonly used for obtaining more stable models from weak learners, which can be unduly affected by outliers. A component of many ensemble techniques, bagging consists of creating models on multiple randomly selected subsets of the original training dataset.

Binning – Binning takes an essentially continuous variable and breaks it into categorical bins. For example, rows containing a column for age could be "binned" in groups 20-29, 30-39, etc. Binning can also improve analysis by reducing the effect of small observational errors and can help identify outliers. Binning is also called *discretization*.

Boosting – Boosting is a set of ensemble methods in which the results of individual weak learners are combined to yield a stronger learner. XLMiner uses the ls boost algorithm, based on least-squares optimization.

Ensemble methods – Ensemble methods integrate multiple data mining models to produce a more stable and reliable model. Often, ensemble methods use bagging to resample the training data and produce multiple closely related models from a single data set.

Epoch – In artificial neural network models, a single pass of the data through the model is referred to as an epoch.

Exponential smoothing – In a moving average, all values within the averaged interval are given equal weight in the calculation. In exponential smoothing data values close to a data point are weighted more strongly than those further away. For example, in a five-day exponentially smoothed average of stock prices, the price two day previous would carry less weight in the average than the previous day's value, and the previous day's value would carry less weight than that day's price. Several different methods of calculation can be used for exponential smoothing.

MAD - Mean Absolute Deviation

Moving average – A moving average, also called a running average, calculates for each data point the average of that point and other points within some interval. For example, a moving average of stock prices over time could calculate the average price of a stock including the previous and subsequent 30 minutes. This has the net effect of smoothing fluctuation that occur on smaller time scales.

Multicollinearity - When two predictors in a regression analysis are strongly correlated it may be difficult or impossible to obtain a unique reliable regression solution. The Variance Inflation Factor is an indicator of potential problems. A VIF greater than 10 is regarded as cause for concern; a VIF greater than 1 indicates the potential for biased results.

One-hot encoding – A protocol for turning categorical variables with N possible values into an N-dimensional binary vector where only one bit, the one corresponding to the variable value, is set to one. One-hot encoding gets its name from its format, since only one bit can be set to one, the remaining bits are zero. This is used to avoid potential problems using categorical values for classification and regression. For example, an encoding scheme that set “red” to 4 and “cyan” to 8 would be a problem for regression since it implies that cyan is somehow twice red.

PMML - The XML markup language Predictive Model Markup Language

TF-IDF – Term frequency – inverse document frequency. A common technique for assigning weights to the importance of specific words in documents relative to the set to which a document belongs. For example, “heart” might be a significant word in a document part of a body of documents about food, but would be less significant in a document belonging to a set of documents about heart disease.