Skytree Server User Guide

Release 15.4.0

SKYTREE®

Notices

Copyright

Disclaimer

Trademarks

Company Information:
Skytree, Inc.
1731 Technology Drive, Suite 700
San Jose, CA 95110
408.392.9300
www.skytree.net

Customers with a support contact can contact support using any of the following methods:
- Log in to support.skytree.net.
- Send questions to support@skytree.net.
- Call 408.392.9300 and select option 3.
- You can also view additional Skytree Server documentation by logging on to support.skytree.net.

Freemium customers can send inquiries to the LinkedIn® Community Group for Skytree:
https://www.linkedin.com/groups/4468542

Revision Date: March 03, 2016

SKYTREE®

Contents

Chapter 1 Overview

Skytree is a machine learning platform that gives organizations the power to discover deep analytic insights, predict future trends, make recommendations, and reveal untapped markets and customers. As the only general purpose, scalable machine learning system on the market today, Skytree is built for the highest accuracy at unprecedented speed and scale.

Skytree is comprised of two components: Skytree Server™ and Skytree Platform™. This guide describes the features available in Skytree Server along with examples on how best to use this for your own specific needs.

Note: Code samples are provided throughout this document that users can copy and paste. This method works best when the document is viewed in Adobe® Reader®.

Frequently Asked Questions

- Running the Software (page 1)
- Datasets (page 1)
- Algorithms/Executables (page 2)
- File Formats (page 3)
- Company (page 3)

Running the Software

- **What operating systems do you support?**

  We support Linux.

Datasets

- **Where can I find the datasets used in the documentation?**

  The examples datasets are distributed along with the installation and are in the `/path/to/installation/SkytreeServer[<-version>]/datasets` folder. By default this location is `/opt/skytree/SkytreeServer/datasets`.

Algorithms/Executables

- **Which algorithms/methods does Skytree Server include?**

  We currently support

  - Neighbor search: All neighbor search, with several options. See *Nearest Neighbors* (page 31) for more information.
  - Kernel Density Estimation (KDE): Computation of density estimates using one of the most popular nonparametric methods, kernel density estimator. See *Kernel Density Estimation* (page 36) for more information.
  - K-Means: The well known clustering algorithm. See *k-means* (page 41) for more information.
  - Linear Regression. See *Linear Regression* (page 125) for more information.
  - Logistic Regression. See *Logistic Regression* (page 124) for more information.
  - Singular Value Decomposition. See *Fast Singular Value Decomposition* (page 39) for more information.
  - Support Vector Machine. See *Support Vector Machines* (page 107) for more information.
  - Two-Point Correlation. See *The Two-Point Correlation* (page 48) for more information.
  - Item-Based Collaborative Filtering for recommendations. See *Item-Based Collaborative Filtering* (page 139) for more information.
  - Gradient Boosted Trees. See *Gradient Boosted Trees* (page 86) and *Gradient Boosted Trees Regression* (page 96) for more information.
  - Random Forests: Bagged decision trees. See *Random Decision Forests* (page 83) and *Random Decision Forest Regression* (page 94) for more information.
  - Generalized Linear Model Regression. See *Generalized Linear Models* (page 130) for more information.

- **What about the other machine learning algorithms?**

  Skytree Server is a subscription product that keeps expanding in its breadth and depth of machine learning methods, as well as increasing in speed. Please contact Skytree support (support@skytree.net) for requests or suggestions.

- **What is the advantage of Skytree Server?**

  The goal of Skytree Server is to provide scalable machine learning solutions for business and scientific applications. Skytree Server modules are significantly faster than popular alternatives, support scaling the data and computation across many nodes of a cluster, and provide automation support for tuning to find the best machine learning model faster.

- **An executable is crashing or has a bug, what should I do?**

  Please contact Skytree support (support@skytree.net).

- **If I have questions, requests, what do I do? Does Skytree Server provide support?**

  You can contact support using any of the following methods:

  - Log in to support.skytree.net.
  - Send questions to support@skytree.net.
  - Call 408.392.9300 and select option 3.

You can also view additional Skytree documentation by logging on to https://docs.skytree.net.

File Formats

- **What kind of file formats does the Skytree Server support?**

  We use our own format which is in a text form. More information about the spec can be found here *Skytree Server File Format* (page 9). We provide tools for exporting the following to Skytree format:

  - Matlab matrices
  - SVMLight files
  - Delimited text files (e.g. CSV, TSV)
  - JDBC Databases
  - Avro files
  - libpcap capture files
  - json files

- **Why do I have to convert my data to the Skytree Server format?**

  The existing popular formats are lacking type and sparsity information. Representing the data in the right format is critical as it can save a lot of memory and maximize performance in large scale computations. If you have data in another format please contact Skytree support (support@skytree.net). We may be able to provide a script for converting it.

Company

- **Are there other products from Skytree, Inc.?**

  The company also provides custom machine learning/optimization solutions for clients.

Additional Information

Refer to the following Skytree Server documents for additional information:

- *Skytree Server Release Notes*
- *Skytree Server Installation Guide*

SKYTREE®

Chapter 2 Introduction to Machine Learning

The main goal of ML is to analyze large volumes of data and extract patterns, rules and models that can be useful either for characterizing existing data or for predicting the properties of future data. Specifically, ML allows you to perform the following tasks:

- **Density Estimation**: Estimate how common or uncommon each point is. An example of a density estimation method is:
  - *Kernel Density Estimation* (page 36)
- **Classification**: Given labeled data (e.g., class labels), predict the class membership of a new data point. Examples of classification methods are:
  - *Nearest Neighbors Classification* (page 52)
  - *Support Vector Machines* (page 107)
  - *Logistic Regression* (page 124)
  - *Gradient Boosted Trees* (page 86)
  - *Random Decision Forests* (page 83)
  -
- **Regression**: Given the values associated with points, predict the associated value of a new point. Examples of regression algorithms are:
  - *Linear Regression* (page 125)
  - *Gradient Boosted Trees Regression* (page 96)
  - *Random Decision Forest Regression* (page 94)
  - *Generalized Linear Models* (page 130)
- **Dimensionality Reduction**: Given a set of points, determine how many dimensions are required to represent it. Potentially find the dimensions that are most important or generate new ones. One example of dimensionality reduction algorithms is:
  - *Fast Singular Value Decomposition* (page 39)
- **Clustering**: Given a set of data points, group the points in clusters according to a criterion. An example of a clustering algorithm is:

    - *k-means* (page 41)
- **Recommendations**: Compute a similarity matrix consisting of similarity measures between pairs of candidate items. Then, using the computed similarities and the set of items a target user has viewed, bought, or rated, a make a recommendation for each candidate item.
    - *Item-Based Collaborative Filtering* (page 139)
- **Multidimensional Querying**: Find points according to their overall characteristics across all dimensions, not just a few specified dimensions as in a typical SQL query.
    - *Nearest Neighbors* (page 31)
- **Multivariate Statistics**: Characterize the dataset according to its overall characteristics across all dimensions, not just a few specified dimensions.
    - *The Two-Point Correlation* (page 48)

Most ML settings involve two stages: training and testing/evaluation. In the training stage, labeled data (and possibly unlabeled) are used to generate a model. In the testing/evaluation stage, the model is evaluated on data that have not been presented in the training stage. If the label of the new data is known then we can generate a score that measures the generalization power of the model.

Data Representation/Dimensionality

There are many different ways in which you can represent data. In ML, the most common way is through tables. Data are represented as points in a d-dimensional space. So, the collection of data is just a long matrix of N x d size, where N is the number of points.

The dimensions are also called features or attributes. The number of dimensions can range from a few to thousands, or even millions. In reality, when the number of dimensions is very high, the features are often correlated. In such high-dimensional scenarios, often the sparsity is very high, meaning that every data point has few non-zero entries. This dimensionality of the data is called “extrinsic” dimensionality.

The extrinsic dimensionality is not a good measure for identifying the complexity of a dataset. It is often claimed that datasets consisting of millions of points that are hundreds of thousands of dimensions can be analyzed. Due to correlations, however, the true dimensionality (also known as “intrinsic” dimensionality) is much lower. In some cases, some of the dimensions can actually be expressed as a function of other dimensions.

The fact that the intrinsic dimensionality might be much lower than the extrinsic can have significant impact on the performance of an algorithm. Furthermore, the number of points of the dataset is not always indicative of the expected performance of an algorithm. It is always possible to have two datasets that have the same number of points and dimensions and get completely different scalability performance.

Machine learning often produces better results when the intrinsic dimensionality of the data is low or data points tend to form clusters; otherwise, the algorithms may suffer from the “Curse of dimensionality” [donoho2000high]. Different algorithms may be best, depending on the intrinsic and extrinsic dimensionality of the data. In the next section, we will discuss some of these methods.

Classification

Given two sets of points, one for class A and one for class B, we wish to find a model that can separate efficiently the points of classes A and B. While we can always find a model that separates the points for both classes in the training set, such a model is likely to be very complex and will fail to generalize well on new points that were not used in the training phases.

The goal of machine learning is to produce models that are as simple as possible and generalize well on unknown data.

Regression

The problem of regression is well known in the linear algebra/statistics community. Given a number of independent variables, we would like to find a linear model for predicting the dependent variable. In general, the regressors are not independent variables and there are dependencies (linear or not) between them. A very important challenge in regression is to remove the redundant regressors that do not contribute in the prediction of the dependent variable. In other words, we follow the principle of Occam's razor (http://en.wikipedia.org/wiki/Occam%27s_razor) and look for the simplest model that explains the data. In linear regression, a simple model corresponds to a model containing as many zero coefficients as possible. Skytree Server has a fully featured linear regression module with many options for feature selection and model simplification. See *Linear Regression* (page 125) and [thompson1995stepwise].

Parametric vs. Non-Parametric Methods in ML

Machine learning models can be separated into “parametric” [Wasserman2004all] and “non-parametric” [Wasserman2006all]. In general, parametric models are appropriate if the correct model class and parameters are chosen. Conversely, non-parametric models are able to fit any underlying data distribution and depend only on the available examples (data points). If the true model class is known, non-parametric models will converge to this only in the limit of infinite data. Another drawback is that non-parametric models are often relatively slow.

In reality, it is almost impossible to determine the true model for the data problem at hand. Moreover, if the model is very complex (which is commonly the case), tuning the parameters of the model can lead to a very difficult, non-convex optimization problem. Thus, non-parametric methods tend to be more accurate because they do not rely on knowing the correct data model. Skytree Server addresses the speed constraints of non-parametric models with a focus on high-performance computing, distributing data and computation, and, where appropriate, using heuristics and approximations.

K-Nearest Neighbors Classifier: The Simplest Classifier

The simplest classifier is the k-nearest neighbors (knn) classifier for two-class classification. A detailed description and use instructions can be found in *Nearest Neighbors* (page 31). The following points provide a more basic description of the nearest neighbors algorithm:

1. A set of points G is given that contains points that have either label A or B.
2. For a new point p, for which you want to predict the label, compute the Euclidean distances d with every point in G.
3. Sort all the distances in d and keep the k points with the smallest distances.
4. From the k nearest points chosen in the previous step, count the points (n_1) that have label A and the points (n_2) that have label B.
5. If $n_1 > n_2$, then assign label A to point p, otherwise assign label B.

Although the algorithm is simple the complexity for computing the k-nearest neighbors for one point is linear in the size of the dataset, which might be very expensive when the number of points in G becomes large. We will see later how this can be computed efficiently. Another issue with this method is how we choose k.

The most effective method for parameter tuning in ML is “leave-one-out cross validation.”

The following steps demonstrate how this method works for the knn case:

1. Choose an initial value of k.
2. For every point p in the training set G, compute its knn label by querying G without considering p.
3. Compute the number of points for which knn computes the right label.
4. Increase k until you reach a maximum k.
5. Keep the k that gives the highest score computed in 3.

Clustering

As previously mentioned, ML produces useful results only if the data has an interesting structure. One way of finding an interesting pattern is to define a similarity measure and use a clustering algorithm. Clustering is a useful preprocessing step in most machine learning algorithms, as it unveils multi-modality, which is very common in real data. A notion of intrinsic dimensionality of the given dataset can be defined in terms of the discovered multi-modality. However, identifying clusters is not an easy problem and in many cases it is ill-posed. There are two types of clustering: hard and soft. In hard clustering, each point is assigned to one and only one cluster, while in soft clustering each point has memberships (which have probabilistic interpretation in some cases) to multiple clusters. Skytree Server supports *k-means* (page 41), which is a hard clustering method.

Chapter 3 Data Preparation

The Skytree Server data preparation utilities provide a set of functions for getting the data into machine-learning-usable form. The user starts with one or more CSV file(s) containing all of the data deemed useful for the machine learning task. These CSV files should contain data in tabular form, including labels for classification tasks or target values for regression tasks. Additional input formats are described in *Other Data Sources* (page 26).

The main steps in the data preparation process are described below.

1. *Generate Header* (page 11): This step reads the data files and recommends data types.

2. *Convert Data* (page 18): This step converts the data from raw format to a machine-learning-usable format.

Skytree Server File Format

Skytree Server natively supports files listing comma- or space-separated real values.

Note: Use either spaces or commas to separate values in lists. Do not include spaces when specifying comma-separated lists.

Skytree Server also offers its own file format with special columns allocated for identifiers, classification labels, and other values.

Note: Skytree Server allows the use of quotation marks to indicate that a space, comma, or other mark should be interpreted as part of the data rather than as a column delimiter. For example, if a single column expresses a city and state code separated by a comma, you can surround that content with quotation marks so that the entire string (including the spaces and commas, but not the quotation marks) serves as the value for that column (for example, `"Atlanta, GA"` or `"San Jose, CA"` On a similar note, if you want to include quotation marks in a column's value, then you can surround the content with quotations marks and replace the intended quotation marks with double quotation marks. For example, `"my quoted ""data""` would evaluate to `my quoted "data"`.

Please refer to this documentation if you want to prepare the data using your own tools. Skytree Server also offers data preparation tools that can be used to prepare the data into a form acceptable for machine learning with Skytree Server.

Skytree Server File Specification

Data points are stored in ASCII as comma- or space-separated values. In addition, two optional lines may be given at the top of the file:

- A line starting with keyword "header" followed by a specification of the data types present in the file.
- A line starting with keyword "attribute_names" followed by a list of column titles.

The following is a example data file:

```
header,meta:3,real:2
1,0,1,3.4,6.37
-1,0,2,2.4,9.37
-1,0,3,3.4,66.37
-1,0,4,11.7,6.7
```

The "header" line is used to provide 2 pieces of information:

- Meta Information: Number of columns used for class labels, target values, and unique identifiers, etc.
- Data Specification: How many columns the data has for each different type of data.

These specifications are given as comma-separated *data tags* of the form "type:n", where "type" gives the type or purpose of the columns and "n" gives the number of columns. If no header is given, Skytree Server assumes that all observed columns contain dense, real-valued data.

Meta Information

In the above example, "header,meta:3,real:2" indicates that each row has values for a total of 5 columns—3 "meta" columns followed by 2 real-valued columns. Only the real-valued columns will feature in measures such as the Euclidean distance, though the meta columns will be involved in the training of classifiers, regressors, and the like.

Specifically, the *meta-data* can contain 0 to 3 columns. In order, these are:

1. Class label: For example, +1 and -1 for classification problems, where +1 represents the positive class and -1 represents the negative class.
2. Target value: A real value associated with a point for regression problems.
3. Id: A unique point identifier. This can be either a string or a number. Note that string IDs cannot contain commas or newlines.

Note, however, that many algorithms accept classification labels or regression targets in a separate file, in which case the meta columns are unused. It is OK to include meta columns even if they will not be used; however, if it is desired to omit them (for instance, to reduce file size), then the meta tag in the header should also be omitted or replaced with "meta:0".

If present, meta columns and the meta tag must always come to the left of any other data expressed in the file.

Categorical Data

Consider instead a data file with the header:

```
header,real:4,int:10
```

Rows in this file would contain values for four real-valued columns followed by 10 categorical columns. In a given column, each discrete categorical value is represented by a different integer. For instance, if a particular column named “`color`” could have three different values — say `blue`, `green`, and `red` — then these might be represented as the integers 5, 10, and 15. Note that categorical integers do not need to start from 0 or 1, nor do they need to be contiguous.

Skytree Server handles categorical values differently from real values. In particular, measures of difference or similarity involving categorical values only consider whether those values are equal or unequal. Thus, unlike real-valued columns, the values 1 and 2 are just as different as the values 1 and 200. This property makes data tag “`int:n`” appropriate for attributes that have no clear notion of distance (“exactly how different is this from that?”) or sequence (“what comes first, this or that?”). On the other hand, “`int:n`” is *not* appropriate for attributes that just happen to be integral, but can be subtracted from one another to form meaningful distances, or can be meaningfully compared with one another using the less-than or greater-than operators; for this kind of data, it is better to use “`real:n`” instead.

Datasets can contain only real columns, only categorical columns, or both real and categorical columns. If both types of columns are present, then the real columns must precede the categorical columns, i.e., the header must always read “`real:n,int:m`”, never “`int:m,real:n`”.

Sparse Data

The `gbt`, `gbtr`, `rdf`, `rdfr`, `svm`, `nn`, `nnplus`, `nnc`, and `wnnc` modules support sparse data inputs, where absent values are understood as 0. Sparse data is indicated in the header with a special data tag “`sparse:real:n`” and is expressed throughout the file as colon-separated attribute-ID:value pairs. For instance, a data file might contain:

```
header real:3 sparse:real:50
1.3 2.4 5.6 3:5.0 52:3.5
4.7 2.1 6.8 4:6.9 15:1.0 20:0.8 21:1.0
2.0 2.3 4.1
```

This file contains a mixture of dense and sparse real-valued columns. The dense columns must occur to the left of the sparse columns and are represented as before, without attribute IDs. Then, a number of attribute-ID:value pairs up to the specified number of sparse columns follows to the right. Note that no attribute-ID:value need be given if a row’s sparse columns are all 0. Also note that attributes are numbered from left to right, starting from zero for the *first dense column* (excluding meta columns); accordingly, the attribute ID of the first sparse column equals the number of dense columns and the ID of the last sparse column equals the sum of the dense and sparse columns minus one. If a dataset contains no dense columns, “`real:n`” may be omitted or replaced with “`real:0`”.

Sparse data cannot be used in conjunction with categorical data, nor can sparse data itself be categorical. However, Boolean attributes represented as real (sparse or otherwise) will act exactly as they would if represented as categorical. By extension, multi-valued categorical data can be represented in sparse by assigning a separate sparse attribute to each possible value.

Generate Header

This first step takes in the raw file and analyzes the following:

1. The number of columns in the file
2. The type of each column:
   - For numeric types, it calculates various statistics on the data.

- For text, it creates dictionaries (containing all the possible values for the columns).

The output is a `header` file containing all this information.

Header Specification

An example header contain lines like this:

```
# column number: 1
integer:categorical=false:min=-1.0:max=981.0:mean=5.5596:sigma=49.387
# column number: 2
integer:categorical=false:min=-2.0:max=700.0:mean=17.966:sigma=39.034
# column number: 3
floating:categorical=false:min=-22:max=700.4:mean=87.987:sigma=6667.0
# column number: 4
text:categorical=true;dictionary="no","yes"
```

Briefly, this header describes the data file to have 4 columns:

- Column 1 contains integer values with a minimum value of -1 and a maximum value of 981.
- Column 2 contains integer values with a minimum value of -2 and a maximum value of 700.
- Column 3 contains floating-point values with a minimum value of -22 and maximum value of 700.4.
- Column 4 contains text data with two possible values: "no" and "yes".

Each line in the header file, except for comment lines beginning with "#", represents one data column. It is advisable to open the header file and check the contents to see that the column types generated are consistent with the user's view of the data.

The header is generated only once and can be used subsequently to transform different subsets of the same data format. It thus specifies a particular dataset.

A Simple Example

In the following example, a file named `income.data` contains the data to be analyzed.

```
# generate-header.sh                            \
  -file                        income.data   \
  -header_out                  income.header \
  -ignore_lines                1             \
  -use_column_names                          \
  -ignore_inconsistent_rows
```

The `-ignore_lines` argument tells the utility to ignore the first line when generating the header, as it may contain header information such as column names, etc. Because the utility tries to determine if a column contains numeric data, headers have to be ignored because they can contain text. If no lines are to be ignored, then this argument can be omitted.

The flag `-use_column_names` indicates that the first line of the first file contains column names, which can be used as arguments to other flags.

By default, the utility checks that all lines have the same number of columns, issuing an error if there's any inconsistency. The flag `-ignore_inconsistent_rows` can be used to ignore rows with too few or too many columns.

A `-time_stamp` option can be used to specify the name or number of a column that should be labeled with a time stamp. Refer to *Time Series* (page 25) for more information.

Specifying a Delimiter

Skytree Server allows you to specify a `-delimiter` option to set the delimiter that is present in the file. Allowed values include `TAB`, `COMMA`, `SEMICOLON`, `VBAR`, and `SPACE`. This value is auto-detected by default, allowing Skytree Server to read CSV and TSV files as well as other delimited files. This option should be set manually in the case where a delimiter is ambiguous or cannot be auto-detected.

Note: Skytree Server allows you to add quotations around text so that it will not be confused as a delimiter (for example, when the delimiter is specified as `COMMA`, and an entry includes a deliberate comma). In addition, because multiple spaces can be used to separate columns, it is necessary to explicitly denote empty column values with "" when using `-delimiter=SPACE`.

```
# generate-header.sh                    \
  -file             income.data   \
  -header_out       income.header \
  -delimiter        COMMA         \
  -ignore_lines     1             \
  -use_column_names               \
  -ignore_inconsistent_rows
```

Identifying Label Columns

If one of the columns in the input CSV is to be used as the label in classification tasks, then it must be identified in the header generation process. Run the following:

```
# generate-header.sh                    \
  -file             income.data   \
  -header_out       income.header \
  -ignore_lines     1             \
  -use_column_names               \
  -label_index      15
```

This identifies column 15 as the label index. Column indices start at 1.

If `-use_column_names` is given, and the first line of the data file contains comma-separated text indicating column names, then those column names can be used wherever an argument to `generate-header.sh` specifies a column. In this example, a column name could be used in place of 15.

Identifying ID Columns

The `-id_index` option can be used to prepare a column to be interpreted as a string ID:

```
# generate-header.sh                \
  -file              income.data    \
  -header_out        income.header  \
  -ignore_lines      1              \
  -use_column_names                 \
  -id_index          2
```

This column is built as non-categorical text in the output header file, and no dictionary is built. If this is specified during `generate-header.sh`, then either the same `-id_index` must be specified during `convert-data.sh` (thereby copying the strings into the output file) or the column must be ignored using `-ignore_columns`.

Specifying Categorical Columns

Sometimes all the data within a column will be numeric, but you want to consider the data categorical because the data does not express magnitude. In the following, columns 3 and 5 are both treated as categorical:

```
# generate-header.sh                  \
  -file                income.data    \
  -header_out          income.header  \
  -ignore_lines        1              \
  -label_index         15             \
  -categorical_number 3,5
```

When the `-categorical_number` option is used, values that are numerically equivalent are considered to be the same. So if `100.0`, 100, and 0100 all appear in a column, those are all treated as the same value. If you want to consider different representations of a number as distinct, you can use the `-categorical_text` option instead:

```
# generate-header.sh                 \
  -file              income.data     \
  -header_out        income.header   \
  -ignore_lines      1               \
  -label_index       15              \
  -categorical_text 3
```

Consider numbers that are used as product codes. In that situation, you might regard 0100 and 100 as distinct. You can use both `-categorical_number` and `-categorical_text` when invoking `generate-header.sh`, if needed. Both flags accept ranges of column numbers or names, such as `3-5`, or a range with a step size, such as `3:2:9`.

Note: It is possible for column names to include dashes or colons. Skytree Server can accept these as column names if they cannot be interpreted as ranges. If a column name is ambiguous (for example, part of it matches another column name), then Skytree Server will fail with an error message. When included in ranges, hyphenated column names can only be notated with a colon. Similarly, column names that include colons can only be specified in ranges using a hyphen.

Finally, use the `-categorical_warn_pct` option to request a warning when the number of categories exceeds the specified percentage of items.

Missing Values

Real-world datasets often contains missing values. Consider a column containing numbers and some missing values. If missing values are indicated with a string such as ?, then the data preparation utility assumes that the column contains text data even though most of the entries in this column are numbers. Use the -missing_value option to avoid such a mis-characterization:

```
# generate-header.sh           \
  -file          income.data    \
  -header_out    income.header  \
  -ignore_lines  1              \
  -label_index   15             \
  -missing_value ?
```

Note: Skytree Server expects the same string to represent a missing value in any column.

Ignoring Columns

There may be instances when your data file includes information that you want to be ignored during the convert-data.sh process. The -ignore_columns option allows you to specify a single, list, or range of column names or numbers to be ignored during header generation. These columns will be treated as non-categorical text, and no dictionaries will be formed. In addition, these columns will be flagged with an "ignored" type in the header.

When specified here, the same columns must also be included in the -ignore_columns option during convert-data.sh.

```
# generate-header.sh                      \
  -file                income.data        \
  -header_out          income.header      \
  -label_index         15                 \
  -id_index            1                  \
  -min_percentiles     income.mins        \
  -max_percentiles     income.max         \
  -ignore_columns      6-9,13             \
  -categorical_text    3                  \
  -categorical_number  2                  \
  -time_stamp          Date               \
  -words_columns       10
```

Note: It is possible for column names to include dashes or colons. Skytree Server can accept these as column names if they cannot be interpreted as ranges. If a column name is ambiguous (for example, part of it matches another column name), then Skytree Server will fail with an error message. When included in ranges, hyphenated column names can only be notated with a colon. Similarly, column names that include colons can only be specified in ranges using a hyphen.

Multiple Input Files

If the data is split across many files, it is advisable to use all these files to generate the header. This will give a complete picture of the data. Notable reasons include:

- One file may not contain all the different values categorical and/or text columns can take, and it is important to have complete dictionaries.
- Simple statistics such as means, variances, minimum values etc. can be different across files, and these should be accounted for.

The example below shows how to use two files:

```
# generate-header.sh              \
  -file           income.data     \
  -file           income.test     \
  -header_out     income.header   \
  -ignore_lines   1               \
  -ignore_lines   1               \
  -label_index    15              \
  -missing_value ?
```

Note that two arguments have been provided for `-file` and two arguments for `-ignore_lines`. This is because the two files may have different numbers of leading lines to be ignored. Therefore, `-ignore_lines` has to be repeated for each file, even if it is the same for all files. You can omit all `-ignore_lines` arguments, which is equivalent to specifying 0 for all files.

In order to use column names when the `-use_column_names` flag is given, the `-ignore_lines` value for the first file must be at least 1, while the `-ignore_lines` argument for other files can be any value. The column names are taken from the first file.

Suggesting Sparse Columns

The `convert-data.sh` script can produce files with columns in a space-saving sparse format, which some Skytree Server methods can use.

Whether it's useful to make a column sparse depends on the ratio of zero to non-zero entries in particular columns, which may not be apparent on inspection of a data file. The `generate-header.sh` script can suggest the columns that should be made sparse.

```
# generate-header.sh                   \
  -file                 income.data    \
  -header_out           income.header  \
  -ignore_lines         1              \
  -sparse_suggest_pct   50
```

In the console output, the script will indicate any columns for which the ratio of zero to non-zero values exceeds 50 percent.

Percentiles and Java Options

The `generate-header.sh` script can pass the flags `-min_percentiles` or `-max_percentiles` with accompanying file arguments. Such files contain comma-separated vectors of percentiles (numbers between 0 and 1) for clamping or ignoring numeric data values during the subsequent *convert data* step. The vector length must be the same as the number of columns in the input data files. For example:

```
# generate-header.sh                            \
  -file             income.data                 \
  -header_out       income.header               \
  -ignore_lines     1                           \
  -label_index      15                          \
  -missing_value    ?                           \
  -min_percentiles  income.mins                 \
  -- -Xmx20g
```

Because `income.data` contains 15 columns, the file `income.mins` might look like:

```
0.0,0.0,0.0,0.25,0.5,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
```

A percentile of 0.0 indicates that the actual minimum value for a column is used as the `min` in the header file. A percentile of 0.5 indicates that the median value becomes the `min` for that column, and so on.

In this example, the argument following the double-dash (`--`) is passed to the Java run-time environment. The `-min_percentiles` and `-max_percentiles` options are particularly memory intensive, so you may need to modify the default behavior of the script. Here, the additional argument sets the maximum Java heap to 20 GB. Any needed Java run-time flags can be passed in this way.

Identifying Date Columns

The `-time_stamp` option specifies a column name or number that includes a measure of time to be converted to a UNIX timestamp. If the timestamp is not already in UNIX format, then the `-date_format` option must also be specified. Optionally, the user can specify the `-time_zone` option to define which time zone the date column is in.

```
# generate-header.sh                                          \
  -file                 SimulatedDataWithCPC.csv              \
  -header_out           simulated.header                      \
  -use_column_names                                           \
  -ignore_lines         1                                     \
  -categorical_number   1,2,3                                 \
  -time_stamp           Date                                  \
  -categorical_text     Age,Gender,Location,Interest          \
  -date_format          "MM/dd/yy"
```

Note: The `-date_format` option is based on the Java SimpleDateFormat class definition: http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html

Text Handling

The `generate-header.sh` script can pass the `-words_columns` flag with an accompanying column specification argument. For each possible word, this parameter creates a notional column in the output .st file. The entry will be 1 if the word occurs in a row, and 0 if it does not. Because most rows don't contain any particular word, the columns are always made sparse. (That is, there is one entry for each word that occurs; otherwise the column is not represented in the output.) For example:

```
# generate-header.sh                            \
  -file             income.data                 \
  -header_out      income.header        \
  -ignore_lines    1                    \
  -words_columns   6
```

Using libsvm Data

Skytree Server accepts libsvm-formatted data to be specified in `generate-header`. Input lines can contain both dense columns as normal as well as sparse entries of the form:

`<column-number>:<value>`

Column numbers must range from 1 to a value given on the command line using `-num_sparse_input_columns`. If `-num_sparse_input_columns` is not provided, then sparse entries cannot be used. In addition, any values found outside the permitted range result in a run-time exception. Dense column numbers are shifted such that they follow the sparse columns, even if they are listed before the sparse data in the input file. For example, if a file given with `-num_sparse_input_columns=100` contains the lines:

`apple, 1:0.5, 20:100, 100:foo`

`banana, 25:1.0, 100:"bar,baz"`

then values "`apple`" and "banana" will be stored in column 101.

Accordingly, we recommend that you specify column names, especially for the dense columns, which are formatted the same way as data, e.g.:

`label, 1:"pixel_1", 2:"pixel_2"`

With Skytree Server, it possible to provide incomplete sets of column names if some of the columns (e.g. the sparse columns) do not need to be named. If column names are omitted for a set of columns, then those columns can only be referenced by number, while all others can be referenced by name or number.

As implied above, sparse columns can contain numerical or categorical data, can be separated by any delimiter, and can contain quoted data. Absent sparse values are treated the same as the string "0", even for categorical columns. The `-skip_bad_lines` and `-max_bad_lines_per_mapper` options handle sparse column numbers given outside the permitted range, which currently results in the lines being dropped entirely, rather than ignoring the individual entries.

Convert Data

Once the header is generated, the task at hand is to convert the data into the Skytree Server format. The Skytree Server format is similar to a CSV format with certain reserved columns. There are two basic inputs to the conversion utility: the input header and the input data file. The resulting output is a file containing the data to be used for machine learning with Skytree Server.

The sections that follow include some examples.

A Simple Example

The input header `income.header` is used to transform the input data from the CSV file `income.data` into the file `income.data_prep.st` in the Skytree Server format.

```
# convert-data.sh               \
  -file          income.data    \
  -header_in     income.header  \
  -ignore_lines  1              \
  -data_out      income.data_prep.st
```

The output of the conversion utility indicates that the data contains 15 columns available for machine learning and appears as follows:

```
Reading the input data specification
Neither -ignore_out_of_range nor -clamp_out_of_range has been
  provided. Out of range values may not be handled properly.
Missing values are not being imputed to mean or are not being ignored.
  They will be set to ?.
Commencing transformation process.
Dense Dimensions: 15. Total Dimensions: 15
Finished transformation process
```

Identifying Label Columns

If the `income.data` file also contains classification labels, these must be identified so that they do not become part of the data matrix. The labels are output to a separate file that can later be used with the various Skytree Server classification methods. Keep in mind that the column to be used as labels must be a categorical column and have an associated dictionary. If available, a column name can be used. Please refer to *Generate Header* (page 11). Here is an example of how to create a labels file:

```
# convert-data.sh                   \
  -file         income.data         \
  -header_in    income.header       \
  -ignore_lines 1                   \
  -label_index  15                  \
  -data_out     income.data_prep.st \
  -labels_out   income.data_prep.labels
```

The data labels will be output to `income.data_prep.labels`. The output will be similar to the following:

```
Reading the input data specification
Neither -ignore_out_of_range nor -clamp_out_of_range have been
  provided. Out of range values may not be handled properly.
Missing values are not being imputed to mean or are not being ignored.
  They will be set to ?.
Commencing transformation process.
Dense Dimensions: 14. Total Dimensions: 14
Finished transformation process.
```

Class Labels

When running `convert-data.sh` and generating a labels file, the labels generated are always "`-1`" and "`1`" for a binary classification problem and "1" to "*N*" for a multi-class classification problem. If the original dataset has a more descriptive label, such as "<=50k" and ">50k" (as is the case with the `income.data` file), then `convert-data.sh` creates a mapping from the original labels to the integers. It does this via lexicographically ordering the label column. The first label gets mapped to "`-1`" and the second label gets mapped to "`1`" in the case of binary classification. For multi-class classification, the original labels get mapped to integers, starting from 1.

Identifying Target Columns

If instead of classification labels, the `income.data` file contains regression targets, a similar process may be followed. The targets are output to a separate file that can later be used with the various Skytree Server regression methods. Here is an example of how to create a targets file:

```
# convert-data.sh                    \
  -file          income.data         \
  -header_in     income.header       \
  -ignore_lines 1                    \
  -target_index 1                    \
  -data_out      income.data_prep.st  \
  -targets_out   income.data_prep.targets
```

The data labels will be output to `income.data_prep.targets`. If available, a column name can be used in place of a column number for `target_index`.

Identifying ID Columns

When ranking data during information retrieval, Skytree Server expects that group IDs are already provided in the third column of the .st file. Use the `-id_index` option with `convert-data.sh`, and point that to the column containing the group IDs.

```
# convert-data.sh                    \
  -file          income.data         \
  -header_in     income.header       \
  -ignore_lines 1                    \
  -id_index      1                   \
  -data_out      income.data_prep.st  \
  -targets_out   income.data_prep.targets
```

In the above example, the group IDs are output to `income.data_prep.st` in the third meta column.

Note: If `-id_index` was specified during `generate-header.sh`, then either the same `-id_index` must be specified here (thereby copying the strings into the output file) or the column must be ignored using `-ignore_columns`.

Mean Imputation

Mean imputation is built into the data preparation utilities and will apply only to columns containing numeric data. For categorical values, mean imputation will replace missing values with 0 for categorical data transformation strategy 1 (horizontalization, see below) or -1 for strategy 2 (unique identifying value). The following command imputes means for missing values:

```
# convert-data.sh                       \
  -file          income.data            \
  -header_in     income.header          \
  -ignore_lines 1                       \
  -label_index   15                     \
  -data_out     income.data_prep.st     \
  -labels_out   income.data_prep.labels \
  -mean_impute
```

Missing Value Identification

If missing values are present in the data as identified during the `generate-header` process, and no imputation is performed, then the `convert-data` process indicates missing values with the character '?' in the file generated through `-data_out`. Examples of these files can be found in the datasets folder (e.g. `income.missing.data.st` and `income.missing.test.st`). Skytree Server modules that support missing values within the algorithm can use input data with missing values in this format.

Out-of-Range Values

Values that are out of the range specified in the header can be dealt with using one of two options: `-clamp_out_of_range` or `-ignore_out_of_range`. If `-clamp_out_of_range` is specified, any values in a column outside of the minimum and maximum values specified in the header are clamped to be the minimum and maximum. If `-ignore_out_of_range` is specified, then any rows that have values outside of the range specified in the header are ignored.

Normalization

Normalization is built into the data preparation utilities and is only applicable to columns containing numeric data. Two forms of normalization are available:

- **Unit**: This form makes the data range between 0 and 1.
- **Standard**: This gives the data a mean value of 0 and unit variance.

To normalize the data, the following command can be used for `unit` normalization:

```
# convert-data.sh                       \
  -file         income.data             \
  -header_in    income.header           \
  -ignore_lines 1                       \
  -label_index  15                      \
  -data_out     income.data_prep.st     \
  -labels_out   income.data_prep.labels \
  -normalize    unit
```

To use `standard` normalization, simply replace the `unit` keyword with `standard`.

Ignoring Columns

Sometimes data can contain columns that are to be ignored for whatever reason (not relevant to the analysis, exploratory modeling, etc.). The following example illustrates how to ignore specific columns from `income.data`:

```
# convert-data.sh                         \
  -file           income.data             \
  -header_in      income.header           \
  -ignore_lines   1                       \
  -label_index    15                      \
  -data_out       income.data_prep.st     \
  -labels_out     income.data_prep.labels \
  -normalize      unit                    \
  -ignore_columns 2,6-9,13
```

This example will ignore columns 2 and 13 and all columns in the range 6 to 9 including both 6 and 9. If available, column names can be used in place of column numbers for the `-ignore_columns` argument. A range with a step size, such as `6:2:10` can also be used.

Note: It is possible for column names to include dashes or colons. Skytree Server can accept these as column names if they cannot be interpreted as ranges. If a column name is ambiguous (for example, part of it matches another column name), then Skytree Server will fail with an error message. When included in ranges, hyphenated column names can only be notated with a colon. Similarly, column names that include colons can only be specified in ranges using a hyphen.

Weighted Columns

If there is a need to scale numeric columns by a constant, then the following can be used:

```
# convert-data.sh                       \
  -file         income.data             \
  -header_in    income.header           \
  -ignore_lines 1                       \
  -label_index  10                      \
  -data_out     income.data_prep.st     \
  -labels_out   income.data_prep.labels \
  -normalize    unit                    \
  -weights      weights.csv
```

The above example assumes `weights.csv` is a file that contains one line representing the weights that need to be applied to each column. The weights must be in CSV format and contain exactly the same number of entries as there are relevant lines (i.e., not commented) in the header `income.header`. For example:

```
0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.10
```

If the original data file `income.data` contains 4 columns, then the above is a valid weights specification. It will multiply column 1 by 0.1, column 2 by 0.2 and so on. If the column contains text, is in the ignore list, or is the label column, then the weight value in that column has no effect, though it must still be specified to maintain order.

Categorical Data Transformation

There are two simple ways (strategies) to transform categorical data into a form acceptable for machine learning:

1. Horizontalize: Create a new column for each different value a categorical column can take. A value of `0.707` is assigned to the right column, and the columns for all other values get 0.

2. Replace the categorical value with a unique number identifying the value.

The difference between the two methods is illustrated below. Consider example data as follows:

```
blue,1,3
red,4,5
blue,2,7
...
```

With strategy 1 (horizontalize), the following transformation will be applied (assuming that column 1 can only take values of `blue` and `red`):

```
0.7071067811865475,0,1,3
0,1,4,5
0.7071067811865475,0,2,7
```

With strategy 2 (unique identifier), the following transformation would be applied:

```
1,1,3
2,4,5
1,2,7
```

The `convert-data.sh` utility defaults to the unique identifier strategy. This can be changed if desired to strategy 1 through the argument `-horizontalize`:

```
# convert-data.sh                      \
  -file         income.data            \
  -header_in    income.header          \
  -ignore_lines 1                      \
  -label_index  15                     \
  -data_out     income.data_prep.st    \
  -labels_out   income.data_prep.labels \
  -normalize    unit                   \
  -horizontalize
```

The proper choice between strategy 1 and strategy 2 depends on the machine learning method. For example, if you are performing nearest neighbor (nn) actions, you can normalize the non-horizontalized variables, and then the rough distance for any continuous dimension would be 1 on average.

Note: Datasets should not include both categorical values and sparse data. The categorical values should be horizontalized first.

Sparse Data

Some Skytree Server methods can accept data in a sparse format, which can significantly reduce the size of the data file produced by `convert-data.sh`. In the sparse format, only non-zero values appear in the output file. In this example, specifying two sparse columns reduces the size of the data file over 10%.

```
# convert-data.sh                      \
  -file           income.data          \
  -header_in      income.header        \
  -ignore_lines   1                    \
  -data_out       income.data_prep.st  \
  -sparse_columns 11,12
```

The generate-header.sh script can suggest the columns that should be made sparse. Refer to *Generate Header* (page 11).

Categorical columns can also be made sparse when -horizontalize is specified. As stated in the section, *Categorical Data Transformation* (page 22), the -horizontalize option creates a new column for each unique value that a categorical column can take.

```
# convert-data.sh                             \
  -file         income.data                   \
  -header_in    income.header                 \
  -ignore_lines 1                             \
  -label_index  15                            \
  -data_out     income.data_prep.st           \
  -labels_out   income.data_prep.labels       \
  -normalize    unit                          \
  -horizontalize
```

Extracting Date Columns

If a date format was specified in generate-header.sh, then you can also us the -extract_from_timestamp option to pull specific date columns from the file (for example, day_of_week, week_of_month, etc.). This extracted information can then be used, for example, when attempting to predict when users will be most likely to buy items.

The following code samples show this process beginning with generate-header.sh. Note again that this process extracts date columns that already exist in the -header_in file. This is different than specifying -time_series in the convert-data.sh process, which takes existing columns and outputs those to individual files in a single folder based on the ID.

```
# generate-header.sh                                \
  -file               SimulatedDataWithCPC.csv      \
  -header_out         simulated.header              \
  -use_column_names                                 \
  -ignore_lines       1                             \
  -categorical_number 1,2,3                         \
  -time_stamp         Date                          \
  -categorical_text   Age,Gender,Location,Interest  \
  -date_format        "MM/dd/yy"
```

```
# convert-data.sh                                       \
  -file                   SimulatedDataWithCPC.csv      \
  -target_index           13                            \
  -targets_out            simulated.targets             \
  -header_in              simulated.header              \
  -data_out               data.st                       \
  -ignore_lines           1                             \
  -ignore_columns         Ad_id,Impressions,Clicks      \
  -extract_from_timestamp second,month,day_of_week,year
```

Refer to the *Convert Data Options* (page 223) in the Command Reference Appendix for a list of keywords that can be used with -extract_from_timestamp.

Time Series

Often data is desired in the form of a time series, where each row represents a point in time. Columns of `type=time` in data files are automatically formatted into a UNIX standard. This works well when you want to divide by the ID of the different time series. In some cases, you might want time-series information to be outputted to individual files in a single folder. This can be performed by enabling the `-time_series` option.

When time series is enabled, use the `-id_index` option to specify the column name or number for the data to be outputted as separate files. The `-data_out` option specifies the folder where each file will be written.

Note: When specifying time-series data, the `-data_out` option points to a folder rather than a file.

The `-id_map_file` option is required and provides the filename for each ID. Use the `-max_open_files` option to specify the maximum number of files that can be open at one time. Finally, the `-time_zone` option defines the way the data is formatted.

```
# convert-data.sh                                \
  -file           SimulatedDataWithCPC.csv   \
  -header_in      simulated.header           \
  -data_out       ~/data.simulated           \
  -id_map_file    simulated.map.out          \
  -time_series                               \
  -id_index       "Interest"                 \
  -max_open_files  6                         \
  -ignore_lines    1
```

Text Handling

The `convert-data.sh` script can also pass the `-words_columns` flag with an accompanying column specification argument. For each possible word, this parameter creates a notional column in the output `.st` file. The entry will be 1 if the word occurs in a row, and 0 if it does not. Because most rows don't contain any particular word, the columns are always made sparse. (That is, there is one entry for each word that occurs; otherwise the column is not represented in the output.) For example:

```
# convert-data.sh                \
  -file           income.data   \
  -header_in      income.header \
  -words_columns   6            \
  -rare_words_pct  10           \
  -stop_words_pct  10
```

The `-rare_words_pct` and `-stop_words_pct` filter out rare and common words from the `.st` file. This filtering does not change the number of notional columns, so multiple runs of `convert-data.sh` with different filtering percentages will yield output files where the column numbers are identical. Also, note that words that can be parsed as numbers become the word "_NUMBER_". In addition, the `-ignore_new_words` option can be used to prevent words not seen in the header generation process from being counted.

Other Data Sources

You can create CSV files from JDBC-enabled databases, JSON, Avro, and libpcap capture file data sources. The resulting CSV file can then be processed with other data preparation tools to create input files for Skytree Server.

JDBC Databases

Relational databases can be used as the source of data for machine learning with Skytree Server if they have a JDBC (Java database connectivity) driver available. Most commonly-used relational database systems have such a driver. *Common JDBC Connection Formats* (page 26) provides a sample of some common URL formats and drivers.

The `db-connect.sh` script allows you to run SQL queries against a JDBC database and store the results to a CSV file. This CSV file can then be used with the other data preparation tools.

The `db-connect.sh` script takes the following required arguments:

- `-db_url` - a database URL
- `-jdbc_driver` - the name of the class for the driver
- `-sql_query` - the query string.

In addition, you must supply either `-csv_out`, the name of an output file, or `-header_out`, the name of a header file, or both.

While output files are in CSV format, header files are in the same format as those created by `generate-header.sh`. The driver class must be available in your Java CLASSPATH.

If the database requires authentication, you can use the `-user` and `-password` flags. If the `-user` flag is used without the `-password` flag, which is recommended, the script will still prompt for a password.

Use the `-help` flag to get all available options or review the *Database Connection Options* (page 227) in the Command Reference Appendix.

```
# db-connect.sh                                              \
  -jdbc_driver org.sqlite.JDBC                               \
  -db_url      jdbc:sqlite:ints.db                           \
  -csv_out     data.actual                                   \
  -header_out  header.actual                                 \
  -null_string NULL                                          \
  -sql_query   "SELECT some_int,some_tiny_int,some_small_int,  \
                some_big_int FROM int_table"
```

Common JDBC Connection Formats

The required format for database URLs varies among database systems but usually starts with `jdbc:`. The following table describes some example connection URL formats and JDBC drivers. This table is meant to be for reference only. Please consult your database vendor's documentation to determine how to obtain a JDBC driver and how to use it, as these URL formats can change between releases.

Table 1: *JDBC URLs and Driver Class Names*

Database	URL Format	Driver Class Name
DB2® v8 (type 2)	jdbc:db2:<database>	com.ibm.db2.jcc.DB2XADataSource com.ibm.db2.jcc.DB2ConnectionPoolDataSource
DB2® v8 (type 4)	jdbc:db2://<host>:<port>/<database>	com.ibm.db2.jcc.DB2Driver
Microsoft SQL Server	jdbc:microsoft:sqlserver://<host>:<port>;databaseName=<database>;SelectMethod=<select_mode>	com.microsoft.jdbc.sqlserver.SQLServerDriver
MySQL	jdbc.mysql://<host>:<port>/<database>	com.mysql.jdbc.Driver
Netezza®	jdbc:netezza://<host>:<port>/<database>	org.netezza.Driver
Oracle (thin) 9i and 10g	jdbc:oracle:thin@<host>:<port>:<database>	oracle.jdbc.driver.OracleDriver
Oracle type 2 (OCI) 9i and 10g	jdbc:oracle:oci:@<database>	oracle.jdbc.OracleDriver

JSON Files

JSON is a lightweight data representation associated with the Javascript programming language. JSON objects have a tree-like structure, so to create a tabular CSV file requires that the tree be flattened.

Suppose you have a file containing the following JSON:

```
{ "name":"Fred Smith","address":{"number":1425,"street":"Park Avenue"},
"age":42,"height_inches":75 }
{ "name":"Betty Smith","address":{"street":"Poplar Way"},"age":51 }
```

Running the script `json2csv.sh` on this file produces:

```
address_number,address_street,age,height_inches,name
1425,Park Avenue,42,75,Fred Smith
?,Poplar Way,51,?,Betty Smith
```

The first line contains labels for the data columns. The labels represent paths to get to the data. For example, the second label, `address_number`, represents the `number` fields within the its containing object, which itself is given by the field `address`. Note that the JSON object for Betty Smith does not have her street number or `height_inches`, so in the CSV file, those fields have a ? indicating missing values.

The script `json2csv.sh` takes one or more filenames, and produces new files with the same name, but a `.csv` extension added. Each JSON object must start on a newline, though each object may span several lines.

The CSV files created by `json2csv.sh` can be processed by the `generate-header.sh` and `convert-data.sh` scripts, in the same way as any other CSV files.

Avro Files

Avro is a data serialization library which can store data in a binary format. Like JSON, the objects in an Avro file have a tree-like structure, which can be flattened to a CSV format.

The script `avro2csv.sh` takes one or more filenames as arguments. Those files must be in Avro binary format. The resulting CSV files have the same names as the input filenames, with a `.csv` suffix added.

libpcap Capture Files

libpcap is a popular library for programming network applications that work at the individual packet level. The Wireshark application uses that library, and can save series of network packets to capture files for later use.

The `pcap2csv.sh` script produces CSV files from libpcap capture files, which may be in the original or "new generation" format. The script extracts somewhat different information for IPv4 and IPv6 packets.

For both IPv4 and IPv6 packets, the CSV contains the fields:

```
arrival time, frame length
```

where `arrival time` is the time elapsed in seconds since the Unix epoch (UTC January 1, 1970), and `frame length` is the number of bytes of data in the packet.

For IPv4 traffic, the CSV output contains these fields:

```
DSCP, ECN, total length, identification, flags, don't fragment,
more fragments, time to live, protocol number, protocol name,
source address, destination address
```

For IPv6 packets, the output fields are:

```
traffic class, differentiated service, ECN, ECN-capable transport,
ECN-CE, flow label, payload length, next header,
protocol name/extension header, hop limit, source address, destination address
```

Consult a network reference for more information on the significance of these fields.

By default, the `pcap2csv.sh` script will output both IPv4 and IPv6 packets to the same output file. In that case, for an IPv4 packet, the columns associated with IPv6 packets will contain ?; likewise, for IPv6 packets, the IPv4 columns will contain ?.

If desired, the flags `-ipv4_only` and `-ipv6_only` can be used to filter the output so that the CSV file contains only IPv4 or IPv6 packets. Note that capture files may contain non-IP packets, which are ignored by the script.

Sliding Window for Time Series

Often data comes in the form of a time series, where each row represents a point in time. For machine learning purposes, it can be useful to examine slices of such data at a time. The `time-slicer.sh` script allows you to create statistics from the sliced data files, which you can then use with the other data preparation tools.

`time-slicer.sh` makes use of a sliding window that contains rows of input files by splitting the data using the ID, computing statistics from the data, and writing each split into a file containing a single series. Each file is created with a map between the ID and the respective file. For example, assume an input file:

```
                    ID,sensor1,sensor2,time,date
  a,-0.882383307710675,-0.953257943291345,1409264744,2014-08-28 15:25:44
  a,0.999857493489427,-0.318425138553527,1409264745,2014-08-28 15:25:45
   a,0.175486589042668,-0.17393956332564,1409264745,2014-08-28 15:25:45
```

The following command can be used to slice this data. The flags `-input_file` and `-output_file` are required. The flag `-window_width` is also required and specifies the size of a slice. The `-ID_column` option indicates that a column is used as a key to identify distinct blocks of rows. The column can be specified with an index or a column name. The utility assumes that such blocks are contiguous in the input file. If this option is given, output rows always contain values from a single block.

```
# time-slicer.sh                                \
  -input_file                   input_data      \
  -output_file                  output.actual   \
  -window_width                 2               \
  -write_column_descriptors                     \
  -write_window_descriptors                     \
  -use_column_names                             \
  -ID_column                    ID              \
  -time_stamp                   Date            \
  -ignore_columns               sensor1,time    \
  -window_increment             1
```

Using `time-slicer.sh`, an output file will be similar to the following:

```
WIN_DESCR,ID (KEY COLUMN),sensor2_min,sensor2_max,sensor2_median,
sensor2_mean,sensor2_sd,sensor2_count,sensor2_missing,
sensor2_zero_cross_rate,sensor2_extremes
```

```
WIN_1390922744000_TO_1390922745000,a,-0.953257943291345,
-0.318425138553527,-0.953257943291345,-0.6358415409224359,
0.4488945811497865,2.0,0.0,1.0,0.0]
```

Note: In the output file, the missing columns (`*_missing`) and extremes (`*_extremes`) are ratios of missing values and extreme value counts over the total number of points within the window or within the number of rows if `-time_gap` is not specified.

Use the `-help` flag to see all available options, or refer to *Time Series Options* (page 335) in the Command Reference Appendix.

SKYTREE®

Chapter 4 Discovery

The Skytree Server Discovery methods provide the solid enterprise-level base for your organization's data discovery needs with the highest in speed as well as statistical rigor. Skytree Server Discovery provides strong fundamentals for essential tasks of unsupervised machine learning: clustering, density estimation, dimension reduction, and multidimensional querying. Skytree Server Discovery performs these operations as needed by most organizations, with best-in-class scalability and response time. Methods include state-of-the-art fast/scalable modules for:

- Nearest Neighbors (page 31)
- Nearest Neighbors Plus (page 33)
- Kernel Density Estimation (page 36)
- Fast Singular Value Decomposition (page 39)
- k-means (page 41)
- The Two-Point Correlation (page 48)

Nearest Neighbors

Nearest neighbor search is a fundamental algorithm in statistics and machine learning. For a given point, the goal is to find the nearest neighbor from a reference dataset. The distance between points is often defined in terms of the Euclidean distance. In general, any similarity can be used, but only if the similarity satisfies the triangular inequality (http://en.wikipedia.org/wiki/Triangle_inequality).

The Skytree Server k-nearest-neighbors method uses state-of-the-art algorithms, data structures and computational techniques to reduce computational complexity achieving many orders of magnitude speedup over a conventional implementation.

Overview

Nearest neighbor search is one of the most fundamental operations occurring ubiquitously in many machine learning problems. The simplest form of the problem is defined as: for the given set R of N points in a metric space and a query point q within this metric space, find the closest point in R to q based on the given metric. For example, nearest neighbors search can be used for a three-dimensional astronomy dataset containing spatial coordinates of stars with a Euclidean metric. For each star, we can find its nearest neighboring stars which hence exert the largest gravitational attraction on it. The nearest neighbors search problem can also be posed as an optimization problem in which the goal is to minimize the distance between the query point q and the returned neighbor point.

Without loss of generality, we assume that the metric is the Euclidean metric defined as:

$$d(q, r) = \sqrt{\sum_{i=1}^{N} (q_i - r_{\mathrm{i}})^2}$$

A natural generalization of this problem is to find k closest points to the given query. This is the k-nearest-neighbor problem where the number of closest neighbors required is k.

A batch form of the problem called “all-nearest-neighbors” is another generalization. In this variant, we find k nearest neighbors for each query point in the given set. These query points are independent of each other and one can run the single query point k-nearest neighbors algorithm multiple times to obtain the solution for each point separately.

K-Nearest Neighbors Classifier

If each of the points in the reference set is labeled, we can build a k-nearest-neighbor classifier. In this case the reference set will contain all the labeled data (the training set) and the query set will contain all the points for which the labels have to be predicted. If the label of a given query point is known (when it is part of a validation set), then we can use it to compute the empirical training error. For example, consider the binary classification problem. For a given value of *k*, we can predict the label of a given query point in the following ways:

- **Voting**: Choose the label by a majority vote of the k neighbors of the query point with each neighbor getting one vote (k should be odd).
- **Weighted scoring**: Choose the label by weighting the vote of each of the k neighbors by its distance to the query point.

k-nearest neighbors classifiers have been successfully applied in many business problems, such as: [henley1996k] and [baesens2003benchmarking]. Refer to *Nearest Neighbors Classification* (page 52) for more information

Computational Complexity

For a given query point and a reference set R of size N, a naive algorithm for computing its 1-nearest neighbor is to calculate the distance between the query point and each reference point. This leads to an algorithm with complexity $O(N)$. This algorithm extended to the all-nearest-neighbors problem with a query set of size N is of complexity $O(N^2)$. This naive algorithm cannot handle even datasets of moderate size.

Usage Examples

Run the following command to see the options available for the nearest neighbors modules:

```
# skytree-server nn --help
```

A Simple Example

Run the `skytree-server` neighbors module with the following arguments:

```
# skytree-server  nn              \
  --references_in random_1kx6.st \
  --k_neighbors   5
```

The above call will compute the five nearest-neighbors for each of the points in the set (excluding itself as the nearest neighbor candidate) and their corresponding distances but it will not write them to the disk. If you want to do so, you must do:

```
# skytree-server  nn              \
  --references_in random_1kx6.st \
  --k_neighbors   5              \
  --distances_out distances      \
  --indices_out   neighbors
```

Note that the `distances` file will contain the distances, not the squared distances.

Bichromatic Neighbors

When the reference and the query set are the same, this is referred to as the monochromatic neighbors problem.

When you define only the reference set using the flag `--references_in`, the program assumes that the same set is both the reference and the query. If a point belongs to the query and the reference set, then its nearest neighbor is itself. When the `--queries_in` flag is not set, the program will compute the nearest neighbors but will exclude this trivial neighbor which is the point itself. In the following setting, though, the program will compute the five nearest neighbors for each query point. For each query point, the first nearest neighbor in this case is the query point itself.

```
# skytree-server  nn              \
  --references_in random_1kx6.st \
  --k_neighbors   5              \
  --queries_in    random_1kx6.st \
  --distances_out distances      \
  --indices_out   neighbors
```

Nearest Neighbors Plus

Nearest Neighbors Plus offers the same basic functionality that is available in the nn module but contains additional options. These include several means of approximation as well as control over the memory footprint of the computation.

Run the following to see the available options:

```
# skytree-server nnplus --help
```

You can also refer to *Nearest Neighbors Plus Options* (page 299) in the Command Reference Appendix for information on the available options.

A Simple Example

The `nnplus` module computes a similar result to nn if none of the additional options offered by `nnplus` are employed. For instance, compare the following:

```
# skytree-server  nn              \
  --k_neighbors   5              \
  --references_in random_1kx6.st \
  --indices_out   neighbors.1    \
  --distances_out distances.1
```

```
# skytree-server  nnplus         \
  --k_neighbors   5              \
  --references_in random_1kx6.st \
  --indices_out   neighbors.2    \
  --distances_out distances.2
```

These both perform *all-k-nearest-neighbors* because no `--queries_in` are provided. All-nearest-neighbors finds for each reference point the nearest k other reference points. Note that there are no differences in the found neighbor indices; however, compressed computation (see next section) is enabled by default for `nnplus`, which may result in small differences in the output neighbor distances. Compressed computation is not available in nn.

Compressed Computation

By default, Nearest Neighbors Plus uses compressed distance computations. This both reduces the memory footprint of computation and accelerates computation, but reduces the accuracy of computed distances by an amount that is usually negligible.

In order to perform *exactly* the same computation as in the nn module, you must use:

```
# skytree-server  nnplus         \
  --k_neighbors   5              \
  --references_in random_1kx6.st \
  --indices_out   neighbors      \
  --distances_out distances      \
  --compression=off
```

Note: Unlike other command line options, the = is mandatory for `--compression`. Using a space will not give the desired result. Keywords `off`, no, `false`, and 0 all have the same effect.

Controlling Memory Usage

You can also limit the (relative) amount of total system memory used by `nnplus` with the `--memory_usage` option. By default, Skytree Server makes an effort not to exceed 75% of its host system's memory, breaking computation into smaller chunks if necessary. Setting `--memory_usage=1.00` grants Skytree Server all memory resources, but computation will still be partitioned to avoid paging on large datasets, if possible. All such attempts to conserve memory are disabled by setting `--memory_usage=0`.

Rank Approximate Nearest Neighbors

Rank approximation is a powerful means of finding approximate nearest neighbors that aims to preserve the meaning of results better than similar methods that approximate by distance. It guarantees that, with high probability, found neighbors have a rank error that is less than a specified percentage of the dataset. For instance, one might use:

```
# skytree-server    nnplus           \
  --k_neighbors     10               \
  --rank_error_tol  0.05             \
  --rank_error_prob 0.90             \
  --references_in   random_100kx6.st \
  --indices_out     neighbors        \
  --distances_out   distances
```

This will find 10 neighbors, each with a 90% chance of having an actual rank that differs from 10 by less than 5% of the data. In this case, 5% of the data is 5000 points; for faster operation, this number should be a good deal greater than the number of neighbors sought.

This form of approximation is preferable to distance-based approximations because nearest neighbors and neighbors with very high rank may nonetheless have very similar distances, especially in high-dimensional datasets.

Alpha-Beta Approximation

Alpha-beta approximation is another form of nearest neighbor approximation. It is designed to assist in cases where dimensionality is high but the (unknown) intrinsic dimensionality of the data is less extreme. The `--alpha` argument, or fidelity, must be set to a small percentage while the `--beta` argument, or restarts, should be set to a small integer:

```
# skytree-server   nnplus            \
  --k_neighbors    10                \
  --alpha          0.2               \
  --beta           5                 \
  --references_in  random_100kx6.st  \
  --indices_out    neighbors         \
  --distances_out  distances
```

For moderate dimensionalities (around 30), values near `--alpha=0.20` are reasonable, while values like `--alpha=0.05` or less are better for higher-dimensional data (100-1000 or more). Values such as `--beta=5` are appropriate for all dimensionalities. Larger `--alpha` and `--beta` will improve accuracy somewhat, though smaller values are key for speed.

Alpha-beta approximation can be used in conjunction with rank approximation.

Bichromatic Nearest Neighbors

The all-*k*-nearest-neighbors search performed when `--queries_in` are omitted, where reference points seek neighbors other than themselves, is sometimes also called *monochromatic* nearest neighbors. On the other hand, if one does provide `--queries_in`, `nnplus` will find nearest neighbors from the reference set for each point in the given query set. This is called a *bichromatic* nearest neighbors search:

```
# skytree-server  nnplus           \
  --k_neighbors   10               \
  --references_in random_100kx6.st \
  --queries_in    random_1kx6.st   \
  --indices_out   neighbors        \
  --distances_out distances
```

Bichromatic nearest neighbors may be used in conjunction with either or both of rank approximation and alpha-beta approximation. Another option of potential interest is `--algorithm=fast`. While by default this option is set to

`--algorithm=fastest`, our "fastest" method is best calibrated to handle situations when both the query set and the reference set are large (e.g. when they are the same). The `--algorithm=fast` setting may sometimes prove faster in situations when the dataset provided in `--queries_in` is much smaller than that given by `--references_in`:

```
# skytree-server  nnplus          \
  --k_neighbors   10              \
  --references_in random_100kx6.st \
  --queries_in    random_1kx6.st   \
  --indices_out   neighbors       \
  --distances_out distances       \
  --algorithm     fast
```

Kernel Density Estimation

Estimating a probability density function is a fundamental task in statistical inference. A probability density function of a continuous random variable describes the relative likelihood of the variable at each point in the observation space. In general, a point with a low probability density represents a rare event, which in some cases corresponds to an outlier. The probability density of a point can also be used to decide if it belongs to a certain class of points.

The Skytree Server Kernel Density Estimation (KDE) method uses state-of-the-art algorithms, data structures and computational techniques to reduce computational complexity achieving many orders of magnitude speedup. It also offers several important features:

- Fast computation of KDE for big query and reference datasets
- Support for Gaussian and Epanechnikov kernels
- Approximate computation of KDE with deterministic error bounds

Overview

Kernel Density Estimation (KDE) is the most widely used and studied nonparametric density estimation method. The "model" is the reference dataset $R = \{r_1, \cdots, r_m\}$, containing the reference points $r_j \in \mathbf{R}^d$. In addition, define a local kernel function $K_h(\cdot)$ to be centered upon each reference point, and its scale parameter h (the bandwidth). The common choices for kernels include the Gaussian:

$$K_h(x) = e^{-x^2/(2h^2)}$$

and the Epanechnikov kernel:

$$K_h(x) = \begin{cases} 1 - x^2/h^2, & x \le h; \\ 0, & x > h. \end{cases}$$

We are given the query dataset $Q = \{q_1, \cdots, q_n\}$, which contains points $q_i \in \mathbf{R}^d$ whose densities we want to predict. The density estimate at the i-th query point is:

$$\hat{f}_h(q_i) = \frac{1}{|R|} \sum_{r_j \in R} \frac{1}{V_{d,h}} K_h(\|q_i - r_j\|)$$

where $\|q_i - r_j\|$ denotes the Euclidean distance between the i-th query point q_i and the j-th reference point r_j, d the dimensionality of the data, $|R|$ the size of the reference dataset, and $V_{d,h} = \int_{z \in R^d} K_h(z)\,dz$ a normalizing constant depending on d and h.

To summarize:

1. **Input**: The set of reference points (observation points), the set of query points for which the probability densities need to be predicted, and the bandwidth value h

2. **Output**: The density estimates for each query point, $\hat{f}(q_i)$

A histogram, for example, can be seen as a kernel density estimator for which the kernel is a histogram bin and the bandwidth is the bin width.

Note that the kernel density estimator produces density estimates from a "smoothed" version of the underlying probability density function; this is due to the effects of the local kernel function centered at each reference point. Nevertheless with no assumptions on the true underlying distribution, given only that $h \rightarrow 0$, $|R|h \rightarrow \infty$, and some mild conditions on $K(\cdot)$, it can be shown that $\int | \hat{f}_h(x) - f(x) | \, dx \rightarrow 0$ as $|R| \rightarrow \infty$ with probability 1. In other words, as more data are observed and the bandwidth shrinks, the estimate converges to the true probability density.

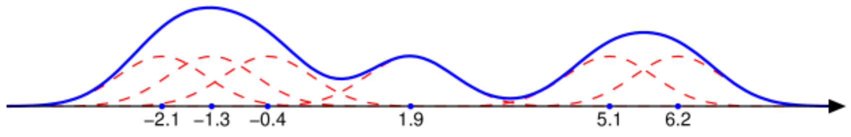

Figure 1: *One-dimensional kernel density estimation example*

In the above image, there are six Gaussians of bandwidth 0.5 centered at x = −2.1,−1.3,−0.5, 1.9, 5.1, 6.2. The KDE estimate at the given x is obtained by dividing the contributions of each of the Gaussians by the number of them that are present, which is six. (Image courtesy wikipedia.)

Choosing the Right Bandwidth

In order to obtain accurate predictions for each query point $q_i \in Q$, it is important to choose the correct bandwidth h. Too large a bandwidth, and predicted densities will be over-smoothed. Too small a bandwidth will result in over-fitting. Instead, we need to find the asymptotically optimal bandwidth h^* for the given reference dataset R.

There are two main types of cross-validation methods for selecting the optimal bandwidth. Cross-validation methods use the reference dataset as queries, effectively setting $Q = R$.

1. Likelihood cross-validation minimizes the Kullback-Liebler divergence $\int p(x) \log(p(x) / \hat{f}_h(x)) dx$, yielding the score $CV_{LK}(h) = \frac{1}{|R|} \sum_{r_j \in R} \log \hat{f}_{h,-j}(r_j)$, where the $-j$ subscript denotes an estimate using all $|R|$ points except the j-th. The bandwidth $h^*_{CV_{LK}}$ that maximizes $CV_{LK}(h)$ is an asymptotically optimal bandwidth in the likelihood cross-validation sense.

2. Least-squares cross-validation instead minimizes the integrated squared error $\int (\hat{f}_h(x) - p(x))^2 dx$, yielding the score $CV_{LS}(h) = \frac{1}{|R|} \sum_{r_j \in R} (\hat{f}_{\sqrt{2h},-j}(r_j) - 2\hat{f}_{h,-j}(r_j))$. Note that the first term in the summation, $\hat{f}_{\sqrt{2h}-j}(\cdot)$, is evaluated using the convolution of the kernel with itself $K_h(\cdot) * K_h(\cdot)$. For the Gaussian kernel with bandwidth of h, the convolution kernel $K_h(\cdot) * K_h(\cdot)$ is the Gaussian kernel with bandwidth of $\sqrt{2h}$

A simple optimization for choosing the optimal bandwidth is to perform a grid search of log-scaled values. Typically, we start from a very large bandwidth value h_{over} as the upper bound and start decreasing the value until the minimum threshold is reached. For example, the upper bound can be chosen to be half of the diagonal length of the bounding box of the reference set. The minimum threshold is chosen to be five orders of magnitude below this over-smoothing bandwidth (that is, $10^{-5}h_{over}$). For the likelihood cross-validation, we choose the bandwidth $h^*_{CV_{LK}} \in [10^{-5}h_{over}, h_{over}]$ that maximizes the likelihood score. For the least-squares cross-validation, we choose the bandwidth $h^*_{CV_{LS}} \in [10^{-5}h_{over}, h_{over}]$ that minimizes the least-squares score.

Note that both cross-validation scores require $|R| = N$ density estimates, each of which is based on $N-1$ points. This results in a brute-force computational cost that scales quadratically (that is $O(N^2)$). Furthermore, nonparametric methods require a large number of reference points for convergence to the true underlying distribution. This has prevented many practitioners from applying nonparametric methods for function estimation.

Usage Examples

Run the following command to see the options available for KDE:

```
# skytree-server kde --help
```

You can also refer to *Kernel Density Estimation Options* (page 275) in the Command Reference Appendix for information on the available options.

A Simple Example

Run the `skytree-server` KDE module with the following arguments to find kernel density estimates on a set of points:

```
# skytree-server   kde             \
  --references_in  random_1kx6.st \
  --kernel         gaussian       \
  --bandwidth      0.1            \
  --relative_error 0.1
```

This command will not output a file with the densities unless you also specify the `--densities_out` option. Note that we only provided `--references_in` and not the `--queries_in` option. When you don't specify `--queries_in`, then the program works in leave-one-out mode, where every point of the reference set is treated as a query point and its density is computed by excluding the contribution of the point itself.

Bichromatic KDE

As a note, when the reference and the query set are the same, KDE is said to be *monochromatic*. If they are different, then it is called *bichromatic*.

In this case we provide the same file for `--references_in` and `--queries_in`. Therefore, the algorithm will treat the query points as if they were distinct and not perform leave-one-out estimation. Density estimates will thus include the self contributions of the points, but otherwise be the same.

```
# skytree-server   kde             \
  --references_in  random_1kx6.st \
  --queries_in     random_1kx6.st \
  --kernel         gaussian       \
  --bandwidth      0.1            \
  --relative_error 0.1
```

Gaussian and Epanechnikov Kernels

Because the Gaussian kernel has infinitely long (though very low-valued) tails, setting `--relative_error=0` will be too slow as computation-accelerating prunes are made impossible. On the other hand, the Epanechnikov kernel has finite support and as such can be used for fast computation of exact KDE:

```
# skytree-server   kde              \
  --references_in  random_1kx6.st   \
  --kernel         epan             \
  --bandwidth      0.22361          \
  --relative_error 0
```

Bandwidths have different scales for Gaussian and Epanechnikov kernels. As a rule of thumb, one may multiply a Gaussian bandwidth by $\sqrt{5}$ (or, more simply, by 2) to obtain a similarly behaving Epanechnikov bandwidth. Different kernels will tend to yield different statistical performance, though this difference quickly becomes insignificant as the dataset size grows.

Fast Singular Value Decomposition

The Singular Value Decomposition (SVD) is a fundamental linear algebraic operation whose abundant useful properties have placed it at the computational center of many methods in machine learning and related fields. Principal component analysis (PCA) and its kernel nonlinear variants are prominent examples, and countless other instances are found in manifold and metric learning, clustering, natural language processing/search, collaborative filtering, bioinformatics and more.

Notwithstanding the utility of the SVD, it is critically bottle-necked by a computational complexity that can render it impractical on massive datasets. Yet massive datasets are increasingly common in applications, many of which require real-time responsiveness. Such applications could use SVD-based methods more liberally if the SVD were not so slow to compute. The SVD module has accelerated methods with automatic relative error control.

Overview

Assume we are given the input matrix $A \in R^{m \times n}$. Then there exists a factorization of the form:

$$A = U \Sigma V^T,$$

where $U \in R^{m \times k}$ is a matrix whose columns form an orthonormal basis spanning the columns of A, $V \in R^{n \times k}$ is a matrix with columns forming an orthonormal basis spanning the rows of A, and $\Sigma = diag(\sigma_1, \ldots, \sigma_k) \in R^{k \times k}$ are the singular values. To preserve equality in the above, k must equal the full rank of A; however useful reduced-dimensionality projections of the data are obtained by specifying smaller values of k. Values $\sigma_1, \ldots, \sigma_k$ monotonically decrease in magnitude and represent the amount of the data's overall covariance captured by their corresponding columns of U and V.

To summarize, the inputs and the outputs to the method are:

1. **Input**: A matrix $A \in R^{m \times n}$ and the desired rank k
2. **Output**: Reduced-rank factors of A, $U \in R^{m \times k}$, $V \in R^{n \times k}$, and $\Sigma = diag(\sigma_1, \ldots, \sigma_k) \in R^{k \times k}$, such that $A \approx U \Sigma V^T$

Choosing the Parameters in SVD

Choosing the rank k is often done *ad hoc* based on a prior domain knowledge. PCA is usually used for visualization purposes, and only two or three components are retained in most cases. Another way is to specify the amount of covariance to capture (for example, 90%) and to choose the minimum number k that achieves such a level.

Usage Examples

Run the following command to see the options available for SVD.

```
# skytree-server svd --help
```

You can also refer to *Singular Value Decomposition Options* (page 325) in the Command Reference Appendix for information on the available options.

A Simple Example

Run the `skytree-server` SVD module with the following arguments:

```
# skytree-server   svd            \
  --references_in  pm_100x40.st  \
  --lsv_out        lsv_out       \
  --rsv_out        rsv_out       \
  --sv_out         sv_out
```

This example will run SVD on the dataset contained in the file "`pm_100x40.st`". The left and transposed right singular vectors will be output to files called "`lsv_out`" and "`rsv_out`" respectively. The singular values will be output to the file "`sv_out`".

The above example demonstrates that the fast SVD algorithm does not always output enough principal components to form a full-rank reconstruction of the input matrix. Instead, it only outputs the amount necessary to achieve the target relative error, which defaults to 10%. We can obtain more principal components by reducing the amount of relative error:

```
# skytree-server   svd           \
  --references_in  pm_100x40.st \
  --lsv_out        lsv_out      \
  --rsv_out        rsv_out      \
  --sv_out         sv_out       \
  --relative_error 0.001
```

This now produces 5 principal components. We can restrict the emitted result to only the first 3 by additionally providing `--svd_rank`:

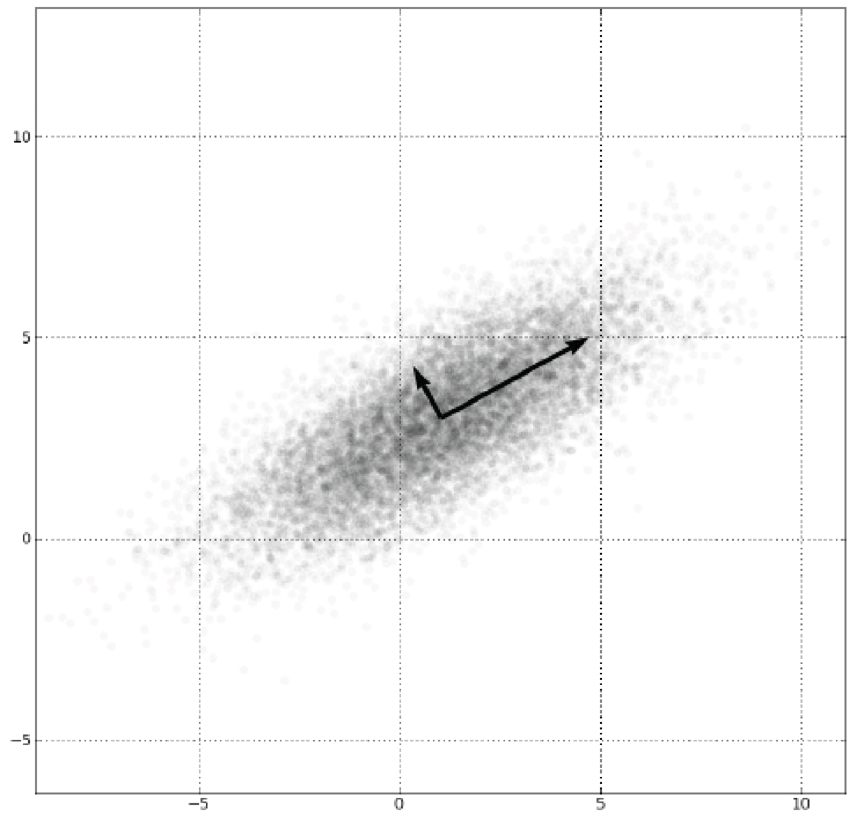

***Figure 2:** Orthogonal directions called by principal directions*

In this example, principal component analysis is used to discover two orthogonal directions called principal directions, the first of which captures the bulk of the data's covariance. (Image courtesy of Wikipedia.)

```
# skytree-server   svd           \
  --references_in  pm_100x40.st \
  --relative_error 0.001         \
  --lsv_out        lsv_out       \
  --rsv_out        rsv_out       \
  --sv_out         sv_out        \
  --svd_rank       3
```

k-means

Clustering is a form of unsupervised learning that tries to find structures in the data without using any labels or target values. Clustering partitions a set of observations into separate groupings such that an observation in a given group is more similar to another observation in the same group than to another observation in a different group. k-means is very popular in retail and customer segmentation [zheng-tobacco], [reynolds1999relationship] and for outlier detection (e.g., in fraud detection/prevention).

Skytree Server has an accelerated version of Lloyd's algorithm and supports the following features:

- Efficient, fast algorithm scales to large numbers of points and clusters
- Automatic determination of optimal number of clusters

- Smart initialization of cluster centroids for faster convergence
- The progressive computation of k-means. The k-means algorithm is an inherently iterative method, which can be made to terminate after a fixed number of iterations
- Multiple restarts to avoid local minima in the produced clusters
- Cross-validation and Bayesian Information Criterion to find the optimal number of clusters

Note: k-means only accepts real-valued data. Because of this, categorical data must be horizontalized.

Overview

Clustering methods require a notion of similarity. The canonical form of k-means defines the similarity measure using the Euclidean metric. For $x, y, z \in R^d$, x is more similar to y than to z if and only if $|| x - y || \leq || x - z ||$. Given a set of observations, $X = (x_1, x_2, \ldots x_N)$, where each observation $x_i \in R^d$, then k-means clustering aims to partition the N observations into k sets ($k < N$) $S = \{S_1, S_2, \ldots S_k\}$ with the objective being to minimize the squared Euclidean distance between points in any subset and their centers of mass:

$$\underset{S}{\arg\min} \sum_{i=1}^{n} \sum_{x_j \in S_i} || x_j - \mu_i ||^2$$

where μ_i is the mean of S_i.

1. **Input**: A set of N reference points
2. **Output**: The coordinates of the centers of each of the k clusters indexed from 0 to $k - 1$ and the membership vector of each point where the i-th component denotes the cluster index to which the point belongs

Note: In this section, "mean", "centroid", and "center" are used interchangeably to mean the geometric center of a set of points. For a set of points $(x_1, x_2, \ldots x_N)$, $x_i \in R^d$, the "mean", "centroid", or "center" of the set is given by:

$$\mu = \frac{1}{N} \sum_{i=1}^{n} X_i$$

A Simple Algorithm for a Fixed Number of Clusters

The naive version of the k-means algorithm (sometimes referred to as "Lloyd's algorithm") is the following:

1. **Initialization Step**: Choose k points in the geometrical space to represent the starting k cluster means. These are called the k means.
2. **Assignment Step**: Assign each point in the data to the closest mean. This step involves calculating the distance from each input point in the data to each of the k means and thus has $O(k \cdot N)$ complexity where N is the size of the input dataset.
3. **Update Step**: Calculate the new set of k means as that of the center of mass of observations for their respective cluster.
4. **Termination Step**: If the new means are the same as the old means, i.e., nothing changed, terminate. Otherwise go to step 2.

Note: Numerical round-off errors can cause some of the k distances between a given point and the means of the k^2 clusters to have the same value. In this case, we employ a simple tie-breaking rule to assign the point to the cluster with the lowest cluster index to facilitate convergence every time.

Determine the Number of Clusters

Choosing the right number of clusters is a challenge in k-means, which leads to the number of clusters often being chosen in an *ad hoc* manner based on prior knowledge.

Skytree Server uses a method based on the Bayesian Information Criterion (BIC) to automatically choose the number of clusters k.

The BIC is defined as $BIC(C|X) = L(X|C) - \frac{p}{2}\log N$, where $L(X|C)$ is the log-likelihood of the dataset according to the cluster model C, $p = k(d+1)$ is the number of parameters with d the dimensionality of the dataset and k the number of clusters. N is the number of points in the dataset.

The cluster model that maximizes the BIC score is chosen.

Usage Examples

Run the following command to see the options available for kmeans:

```
# skytree-server kmeans --help
```

You can also refer to *k-means Options* (page 277) in the Command Reference Appendix for information on the available options.

A Simple Example

Our first example creates clusters for 100,000 randomly generated 6-dimensional points from file `random_100kx6.st`. An output file, `centroids.csv`, is created that contains the centroid positions for the number of clusters specified (4 in this case). Another optional output file, `assignments.csv`, contains the cluster memberships for each point in the input datasets. Typical results are shown in the image that follows.

```
# skytree-server    kmeans            \
  --references_in   random_100kx6.st  \
  --centroids_out   centroids.csv     \
  --k_clusters      4                 \
  --memberships_out assignments.csv
```

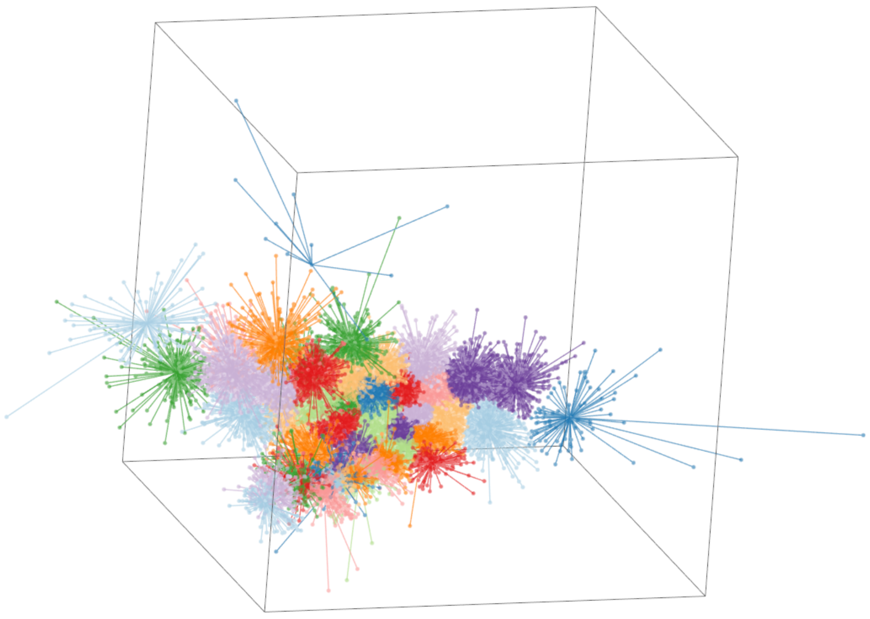

Figure 3: *k-means clustering*

The above images shows a visualization of k-means clustering for $k=50$ on a 4-dimensional astronomical dataset (Sloan Digital Sky Survey) consisting of 100k points. The first 3 dimensions are shown. Colors indicate cluster membership, and the points are connected to their associated cluster centroid.

The distortions (the squared Euclidean distance to the closest centroid) for each point in the input set can be obtained via the `--distortions_out` option as follows:

```
# skytree-server    kmeans                \
  --references_in   random_100kx6.st      \
  --centroids_out   centroids.csv         \
  --k_clusters      4                     \
  --memberships_out assignments.csv       \
  --distortions_out distortions.csv
```

The algorithm to compute clusters can be specified by the `--algorithm` option. By default, we use a fast exact heuristic method on top of the traditional Lloyd's algorithm (specified by `--algorithm=fast`). However, the original Lloyd's algorithm can be used as follows:

```
# skytree-server    kmeans                \
  --references_in   random_100kx6.st      \
  --centroids_out   centroids.csv         \
  --k_clusters      4                     \
  --algorithm       lloyds                \
  --memberships_out assignments.csv
```

The Lloyd's algorithm and the fast exact heuristic method can be run in distributed mode. (See *k-means* (page 176) for more details.)

k-means with Sparse Data

The kmeans module can be seamlessly applied to sparse data as demonstrated with the following 12419-dimensional sparse dataset:

```
# skytree-server    kmeans             \
  --references_in   docword_nips.st    \
  --centroids_out   centroids.csv      \
  --k_clusters      4                  \
  --memberships_out assignments.csv
```

k-means with Online Pre-Processing

It has been proven that online k-means can quickly converge closely to the solution. Here we show an example of online k-means:

```
# skytree-server  kmeans         \
  --references_in sdss.train.st \
  --k_clusters    10             \
  --centroids_out centroids.csv \
  --algorithm     online_fast
```

The algorithm runs for one epoch (one pass over the dataset) in online mode and then it continues in batch mode. You can specify more epochs with, e.g. `--epochs=2` and restrict the number of iterations in batch mode with, e.g. `--iterations=1`. Notice that if you set it to `--iterations=0`, it will still run in batch mode once. This behavior is also obtained by specifying `--algorithm=online`.

Note: All online algorithms assume that data points are independent and identically distributed (i.i.d.), or presented in random order. In the dataset we provided, two classes of points are stored sequentially in chunks. To sidestep this problem, the order of the input data is automatically randomized by default.

If you know that your data points are i.i.d., then you can turn off the randomization as below.

```
# skytree-server  kmeans         \
  --references_in sdss.train.st \
  --k_clusters    10             \
  --centroids_out centroids.csv \
  --algorithm     online_fast   \
  --randomize     false
```

Often, with randomization enabled, online k-means tends to produce better initial distortions, and the following batch k-means requires fewer iterations to converge. Note that this observation may not always be true since, randomized input order or not, the initial cluster centers are always assigned randomly, and some starting positions may lead to faster convergence.

In short, only if you know that your input data is i.i.d. (or already in random order) should you use `--randomize=false`.

Initialization of the Centroids with k-means++

k-means results are sensitive to the selection of the initial centroids. The default option is to pick random points from the reference dataset. The kmeans++ centroid initialization option uses an advanced algorithm to pick the centroids. This will take some extra time, but it leads to faster convergence. The initialization scheme can be specified by the `--initialization` option (`--initialization=random` by default) as follows:

```
# skytree-server    kmeans            \
  --references_in   sdss.train.st     \
  --centroids_out   centroids.csv     \
  --k_clusters      10                \
  --memberships_out assignments.csv   \
  --initialization  kmeans++
```

k-means with Automatic Multiple Restarts

When choosing random initial centroids from the reference points, it is best to restart `kmeans` several times and pick the best set of resulting centroids. This can be done automatically using the `--n_restarts` option as follows:

```
# skytree-server    kmeans            \
  --references_in   sdss.train.st     \
  --centroids_out   centroids1.csv    \
  --k_clusters      10                \
  --memberships_out assignments.csv   \
  --n_restarts      10
```

The set of centroids returned is the one with the lowest distortion.

Finding the Optimal Number of Clusters

We seldom know the number of clusters in a given dataset. Skytree Server provides a method to automatically determine the best value for k. For this you must provide a minimum and maximum number of clusters acceptable. The algorithms will search for the optimal k value between these two limits with a default step size of 1. For faster scanning, you can specify a larger step size for k and/or limit the number of iterations for each value of k. You can quickly scan for the optimal number of clusters k with the following:

```
# skytree-server   kmeans           \
  --references_in  3gaussians.st    \
  --centroids_out  centroids.csv    \
  --iterations     1000             \
  --k_min          2                \
  --k_max          10               \
  --k_step         1
```

For these parameters, the optimal number of centroids is found to be 3. This is expected, since the input points are sampled from three Gaussian blobs. For more realistic cases, you could do several scans with increasing accuracy. (e.g., Start first with large values for `--k_max` and `--k_step` and a small number of iterations with the `--iterations` option, and when a first candidate for k is found, you can update the range for k in combination with a smaller value of `--k_step`, no limit on the number of iterations and a larger number of restarts specified by `--n_restarts`.)

A fraction of the dataset is held out as the validation set during the search for the optimal number of clusters. This fraction defaults to `0.2` but can be specified with the `--percentage_hold_out` options as follows:

```
# skytree-server         kmeans          \
  --references_in        3gaussians.st   \
  --centroids_out        centroids.csv   \
  --iterations           1000            \
  --k_min                2               \
  --k_max                10              \
  --k_step               1               \
  --percentage_hold_out 0.25
```

Finding the Cluster Memberships for Unseen Points

Once you have used k-means to find the cluster centroids for a given set of reference points, you might want to find the cluster memberships for a given set of query points (typically, a different set of points). The kmeans module can be used for that purpose with the `--run_mode` option. By default, `--run_mode=train`, resulting in k-means clustering. Membership assignment (and distortion computation) on a different set of points can be achieved by running k-means with `--run_mode=eval` and the `--queries_in` option as follows:

```
# skytree-server     kmeans           \
  --centroids_in     centroids1.csv   \
  --queries_in       sdss.test.st     \
  --memberships_out assignments.csv  \
  --distortions_out distortions.csv  \
  --run_mode         eval
```

Coreset Construction for k-means

A coreset refers to a summary of the data that can be used to approximate the entire dataset for clustering purposes. Using distributed coreset construction, you can create a summary of a dataset to use with future kmeans clustering. The cluster structure in the dataset is preserved with a much smaller weighted dataset.

Use `--run_mode=coreset` tells kmeans to run in coreset mode. The `--coreset_out` option creates a summary of a dataset to use for future kmeans clustering. This output file includes 5 meta columns: (0) label, (1) target, (2) id, (3) point weight, and (4) score weight. Note that in this output, the points from the table retrain their IDs. Generated points, however, will have an ID of `-1`, indicating that they are not actual points in the dataset.

You can also use the `--coreset_size` option to define the size of the output coreset. This size is specified in terms of a fraction of the actual file size. When kmeans runs in coreset mode, this value defaults to 0.01 (corresponding to 1% of the size of the original dataset).

```
# skytree-server    kmeans          \
  --references_in   sdss.train.st   \
  --coreset_out     cores           \
  --coreset_size    0.05            \
  --run_mode        coreset
```

The Two-Point Correlation

The two-point correlation function is a special case of the *n*-point correlation function, which serves to measure the fractal dimension of a dataset. These functions are commonly used to test whether two different datasets are derived from related distributions. For instance, in astrophysics, researchers can test whether simulated galaxy evolution produces distributions of galaxies similar to those of observed by testing whether their computed *n*-point correlation values are similar.

Skytree Server Two-Point Correlation uses state-of-the-art algorithms, data structures and computational techniques to reduce computational complexity achieving many orders of magnitude speedup. It offers several important features:

- Fast computation of the 2-point correlation, including for large datasets.
- Monochromatic (dataset vs. itself) and bichromatic (two datasets) computation.

It also offers options for matchers with lower-bound distance 0 and variable upper-bound distance, or *radius*. This may be evaluated *monochromatically*—between a dataset and itself—or *bichromatically*, where pairs represent points from either of two input sets. Bichromatic computation is useful, for instance, when breaking up large computations into smaller, component problems.

Overview

Each member of the family of *n*-point correlation functions takes as input a dataset and produces as output an integer. In the general case, member functions consider all size N subsets of the input dataset and count those subsets that satisfy a *matcher* condition, M. For an input set R, this is expressed:

$$c = |\{ R_n \subset R : |R_n| = N, M(R_n) = \text{true} \}|$$

Matcher conditions often consider measures such as the distances between points and the angles formed between them, and may reject subsets of points when some of them are too near each other, are too distant, or form very sharp or very wide angles.

The Two-Point Correlation

In the special case that n=2, matchers simply test whether each unique pair of points is between a given minimum and maximum distance of each other. This may be written as:

$$c = \sum_{r_i \in R} \sum_{r_j \in R} I(h_{\text{lo}} \leq d(r_i, r_j) \leq h_{\text{hi}})$$

where *I* is the indicator function, which returns 1 whenever its argument condition is true.

Correlation Profiles

As previously mentioned, the n-point correlation functions are commonly used to test whether two datasets are derived from related distributions. However, any single n-point correlation function—i.e., any particular matcher—may prove inadequate for this purpose. Accordingly, it is common practice to measure a dataset's *correlation profile*. This is a graph with correlation values along the y-axis and a spread of matchers along the x-axis. For instance, in the 2-point correlation, the x-axis may represent matcher conditions with lower-bound distance h_{lo} = 0 and upper-bound distance $h_{hi} = x$. Similar *correlation profiles* are a much stronger indication that the two datasets are derived from related distributions than similar correlation values in isolation.

Usage Examples

Run the following command to see the options available for two-point correlation:

```
# skytree-server two_pt --help
```

You can also refer to *Two-Point Correlation Options* (page 339) in the Command Reference Appendix for information on the available options.

A Simple Example

You can compute the 2-point correlation with radius 1 using the following syntax:

```
# skytree-server  two_pt                                  \
  --references_in adult_train_transformed_stratified_10k.st \
  --radius        1
```

We can then compare this to the correlation value computed for a different sample from the same dataset:

```
# skytree-server  two_pt                                \
  --references_in adult_test_transformed_stratified_6k.st \
  --radius        1
```

(Note that we must normalize the returned values by the dataset size in order to properly compare them.)

Finally, if we sum the above two results along with the result of the following, we arrive at the 2-point correlation of the union of the two sets:

```
# skytree-server  two_pt                                  \
  --references_in adult_train_transformed_stratified_10k.st \
  --queries_in    adult_test_transformed_stratified_6k.st   \
  --radius        1
```

Chapter 5 Prediction

Skytree Server Prediction is a collection of supervised learning modules that solve regression and classification problems. Specifically, Prediction is comprised of the following machine learning methods:

- Nearest Neighbors Classification
- Ensemble Learning methods (Gradient Boosting Trees and Random Decision Forests)
- AutoModel
- Support Vector Machines
- Metric Learning
- Logistic Regression
- Linear Regression
- Generalized Linear Models - Classification and Regression

Each is represented by their own unique module name. Note that the modules that end with the letter "r" signify regression modules, while the rest of the modules are classification modules.

Model Training and Model Validation

All Skytree Server modules require a training dataset to train a model. This is done through the `--training_data_in` and `--training_labels_in` options for classification, and the `--training_targets_in` option for regression. A further set of module specific parameters can be used to train the model. All Skytree Server modules allow for model validation and parameter selection through (i) cross-validation (specified through `--num_folds`), (ii) a holdout set (specified through `--holdout_ratio`, a number between 0 and 1), and (iii) a separate tuning file (specified through `--tuning_data_in` and `--tuning_labels_in`/ `--tuning_targets_in`).

Model/Result Output

All Skytree Server modules can output the model in binary form by specifying the output filename through the `--model_out` option. When a file is specified through the option `--testing_in`, Skytree Server also allows several different options to write out the results obtained on the testing dataset in different files through the options `--predicted_labels_out` and `--predicted_probabilities_out` (for classification) and `--predicted_targets_out` (for regression). These output files will have predictions obtained for the testing dataset. The model used for testing is the model that has the parameters that produce the best results during model validation. This model is then trained on the entire training dataset and then used for testing purposes.

Model Evaluation

Evaluating the predictions produced as outputs of these supervised learning modules can be done using the `score` module. (See *Scoring* (page 151) for more information.)

AutoModel

The AutoModel module tunes over a large set of algorithms and parameter to obtain the best performing model. Optionally, it allows the user to restrict the space to search in. (See *AutoModel Module* (page 103) for more information.)

Advanced Nearest Neighbor Methods

Advanced Nearest Neighbor Methods is a collection term for Nearest Neighbor Classification (nnc) and Weighted Nearest Neighbor Classification (wnnc). Both nnc and wnnc allow you to perform binary classification with various model validating options.

Refer to the following sections:

- Nearest Neighbors Classification (page 52)
- Weighted Nearest Neighbors Classification (page 54)

Nearest Neighbors Classification

Nearest neighbors classification first performs a *k* nearest neighbors search on points from a labeled reference set, and then classifies query points according the most common label amongst their found neighbors. As nearest neighbors classification is clearly influenced by nearest neighbors search, all options available for `nnplus` are also available for nnc. In addition, there are options to perform classification for multiple values of k simultaneously as well as to control the output of various scores.

The examples will use the datasets distributed along with the documentation.

Again, run the following to see all available options:

```
# skytree-server nnc --help
```

You can also refer to *Nearest Neighbors Classification Options* (page 291) in the Command Reference Appendix for information on the available options.

A Simple Example

Run the following to perform nearest neighbors classification:

```
# skytree-server       nnc               \
  --k_neighbors        20                \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st      \
  --probabilities_out  probs.nnc         \
  --scores_out          scores.nnc        \
  --labels_out          labels.nnc
```

This produces two scores files, `scores.nnc.class1` and `scores.nnc.class-1` (for classes 1 and -1), two class probabilities files, `probs.nnc.class1` and `probs.nnc.class-1`, but only one file for labels, `labels.nnc`. The file names' class suffixes will match the class labels provided by `--training_labels_in`.

The scores files contain the counts of neighbors found from either class. In the case of nnc (without class weights), the class probabilities are equal to scores divided by k.

Modified Class Weights

To accommodate situations where training sets are formed by stratified sampling of either class, the `--classweight` argument can be used to artificially inflate the impact of one class or the other. This is used as follows:

```
# skytree-server        nnc               \
  --k_neighbors         20                \
  --training_in         sdss.train.st     \
  --training_labels_in  sdss.train.labels \
  --testing_in          sdss.test.st      \
  --probabilities_out   probabilities.nnc \
  --labels_out          labels.nnc        \
  --classweight         10                \
  --classweight         1
```

The `--classweight` parameter must be repeated for each class in `--training_labels_in`. Its values are assigned according to the numerical order of the class labels. In this case, the `--classweight=10` is for class -1 while class 1 still has `--classweight=1`. Here, we use `--probabilities_out` instead of `--scores_out` because, while the scores for class 1 won't change, its probability results will. More details regarding the usage of class weights in Skytree Server is presented in *Class Weights* (page 164).

Leave-One-Out Cross Validation

It is possible to call nnc without `--testing_in` to perform leave-one-out cross-validation. Leave-one-out cross-validation finds classifications for each training point considering all training data points other than itself:

```
# skytree-server        nnc             \
  --k_neighbors         20              \
  --training_in         sdss.train.st   \
  --training_labels_in  sdss.train.labels
```

After classification, nnc computes classification accuracy, probability-based Gini index, and a number of other measures such as Precision/Recall, F-Score and Capture Deviation.

Tuning Over Multiple Values of K

nnc permits simultaneous computation to assist with and accelerate the comparison of scores and/or classifications for different values of k. For example, try:

```
# skytree-server       nnc            \
  --k_neighbors        5:5:25         \
  --training_in        sdss.train.st  \
  --training_labels_in sdss.train.labels
```

or

```
# skytree-server       nnc            \
  --k_neighbors        5,10,15,20,25  \
  --training_in        sdss.train.st  \
  --training_labels_in sdss.train.labels
```

Both (equivalent) commands will perform leave-one-out cross-validation tuning for each value of k in the set {5, 10, 15, 20, 25}. Finding different classification results simultaneously is much faster than performing separate runs of k nearest neighbors classification.

It is possible to specify a separate tuning set with the `--tuning_in` and `--tuning_labels_in` options.

Acceleration

All accelerating options available to `nnplus` can also be used with any form of nearest neighbors classification. See *Compressed Computation* (page 34), *Rank Approximate Nearest Neighbors* (page 34), and *Alpha-Beta Approximation* (page 35) for additional details. Also, keep in mind that `--algorigthm=fast` can sometimes be faster for smaller query sets.

Weighted Nearest Neighbors Classification

Weighted Nearest Neighbors considers neighbor distances in addition to their labels, thereby weighting nearer neighbors more than those that are farther away. In addition to the nnc options, wnnc includes additional options that allow the weighing of points based on distance. See examples for further details.

The following command describes all available wnnc options:

```
# skytree-server wnnc --help
```

You can also refer to *Weighted Nearest Neighbors Classification Options* (page 340) in the Command Reference Appendix for information on the available options.

A Simple Example

The following will perform weighted nearest neighbors classification:

```
# skytree-server       wnnc               \
  --k_neighbors        20                 \
  --dist_weight        1/r                \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels  \
  --testing_in         sdss.test.st       \
  --scores_out         scores             \
  --probabilities_out  probabilities      \
  --labels_out         labels
```

As with nnc, this produces two scores files named for either class and one labels file. The scores files now contain sums of the values returned by the distance weight function as applied to each found neighbor from the corresponding class. There are two probabilities file produced, one for each class. This formula shows how the probabilities are calculated for class 1. The probability of a point belonging to class -1 is computed by subtracting the probability of the point belonging class 1 from 1.

$$P(q) = \frac{\text{score}_1(q)}{\text{score}_1(q) + \text{score}_{-1}(q)},$$

which no longer has a denominator necessarily equal to k.

Distance Weight Functions

The previous section explicitly uses `--dist_weight=1/r`, the default distance weight function. Also available is `1/r^2` as well as both the Gaussian (`gaussian` and `fixed_gaussian`) and Epanechnikov (epan and `fixed_epan`) kernels, e.g.:

```
# skytree-server       wnnc                \
  --k_neighbors        20                  \
  --dist_weight        1/r^2               \
  --training_in        sdss.train.st       \
  --training_labels_in sdss.train.labels   \
  --testing_in         sdss.test.st        \
  --scores_out         scores              \
  --probabilities_out  probabilities       \
  --labels_out         labels
```

All six distance weight functions can yield different classification behavior. The `1/r^2` function tends to favor near points even more so than `1/r`. On the other hand, the Gaussian and Epanechnikov kernels, defined

$$K_h(d) = e^{-\frac{1}{2}\left(\frac{d}{h}\right)^2} \qquad \text{and} \qquad K_h(d) = \max\left\{0,\ 1-\left(\frac{d}{h}\right)^2\right\}$$

respectively (un-normalized), distribute values more evenly over points of low to intermediate distance. The `fixed_gaussian` and `fixed_epan` weight functions have user-specifiable fixed bandwidths, while the `gaussian` and epan weight functions use bandwidths that are relative to each query point's k-th neighbor distance.

Kernel Bandwidths

The Gaussian and Epanechnikov kernels are parametrized by a bandwidth h. This parameter controls kernel smoothness, with smaller values resulting in kernels that drop to 0 more quickly. For wnnc, kernel bandwidths can be set as ratios of the k-th nearest neighbor distance or are fixed (in the case of the `fixed_ kernels`):

```
# skytree-server       wnnc              \
  --k_neighbors        20                \
  --dist_weight        epan              \
  --bandwidth          0.75              \
  --bandwidth          1.10              \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st      \
  --scores_out         scores            \
  --probabilities_out  probabilities     \
  --labels_out         labels
```

Like `--classweight`, the `--bandwidth` parameter must be repeated for each class and its values are assigned in the numerical order of the class labels. Further, Gaussian kernel bandwidths are always shrunk an additional factor of $\sqrt{5}$, which results in behavior more similar to the Epanechnikov kernel.

If omitted, the default `--bandwidth` scaling factor is `1`. In practice, however, bandwidth optimization is critical for maximizing accuracy metrics when using the Gaussian and Epanechnikov kernels.

Mitigating Class Imbalance

Extreme imbalances in class sizes can limit the effectiveness of conventional nearest neighbors classifiers. Problems arise when (understandably) weak signals from the rare class are truncated to no signal at all, leading to uninformative probabilities of exactly zero. To help mitigate this, Skytree Server implements the `--imbalance` option, which forces consideration of the rare class even for low values of k.

The `--imbalance` option in wnnc finds the k nearest neighbors from each class, and then weights their contributions to the final probability according to the selected function of distance. Unweighted k-NN classification would obviously always return equal class probabilities across the board in this context, but weighting by distance can result in a comparatively small or large contribution from the equal numbers of found neighbors.

The overall effect is that `--imbalance` always returns a nonzero probability even for very rare classes. These probabilities may always be very small, and as such, prediction may never label any points as belonging to the rare class. Nonetheless, it becomes possible to compare points that have, e.g., a 2% chance and a 0.05% chance of being members of the rare class, whereas both may have been predicted to have a 0% chance without the `--imbalance` option. This allows for the formation of sets of rare class candidates when used in conjunction with the `--probability_threshold` parameter, which defaults to 0.5 but can instead be configured down to, e.g., 0.01 in order select all points that have a mildly heightened chance of being members of the rare class. In certain contexts, `--probability_threshold` may be tuned automatically, such as with `--classification_objective`.

```
# skytree-server       wnnc              \
  --k_neighbors        20                \
  --dist_weight        1/r               \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st      \
  --scores_out         scores            \
  --probabilities_out  probabilities     \
  --labels_out         labels            \
  --imbalance
```

Note that, though the SDSS dataset is not particularly unbalanced, the `--imbalance` option can still impact results.

Assymetric Values of K

When using `--imbalance`, it is possible to set different values of `--k_neighbors` for either class. This is again accomplished by repeating the `--k_neighbors` parameter for each class in the numerical order of their labels:

```
# skytree-server       wnnc               \
  --k_neighbors        200                \
  --k_neighbors        10                 \
  --dist_weight        1/r                \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st       \
  --scores_out         scores             \
  --probabilities_out  probabilities      \
  --labels_out         labels             \
  --imbalance
```

This can be interpreted as finding at least 200 neighbors from class -1 and 10 neighbors from class 1.

Sampling

Upsampling or downsampling of the classes can help restore balance, and can be specified with the `--sampling_ratio` option, given for each class:

```
# skytree-server       wnnc               \
  --k_neighbors        20                 \
  --dist_weight        1/r                \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st       \
  --scores_out         scores             \
  --probabilities_out  probabilities      \
  --labels_out         labels             \
  --imbalance                             \
  --sampling_ratio     0.1                \
  --sampling_ratio     2.4
```

The above example will downsample the negative class (label -1) by a factor of 10, and upsample the positive class (label 1) by a factor of 2.4. There's also an option to let the algorithm set the sampling ratios automatically to obtain balance, with `--sample_imbalance`. Further, if `--sample_imbalance` is specified, then `--imbalance_scale` can be used to scale both classes by an additional factor:

```
# skytree-server       wnnc               \
  --k_neighbors        20                 \
  --k_neighbors        10                 \
  --dist_weight        fixed_epan         \
  --bandwidth          0.2                \
  --bandwidth          0.3                \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st       \
  --scores_out         scores             \
  --probabilities_out  probabilities      \
  --labels_out         labels             \
  --imbalance                             \
  --sample_imbalance                      \
  --imbalance_scale    2
```

The above example would equalize the class sizes at a size of 2 times the original size of the smaller class.

For all sampling, the option `--sample_with_replacement` can be set to on or `off` for sampling with or without replacement.

Also note that sampling uses random number generators that can be seeded to obtain reproducible results, for example:

```
[INFO] To reproduce, use --table_sampling_seed=1359937539.
```

Asymmetric values of k may be used together with *Tuning Over Multiple Values of K* (page 53) by simply repeating `--k_neighbors` options in like fashion. This emits results for all combinations of k for either class; some care is warranted so as not to consume more storage than intended, as the number of such combinations can grow quickly.

Tuning with or without Leave-One-Out Cross Validation

It is possible to tune over multiple k-values and kernel bandwidths, in combination with all other parameters as specified in the previous section, by providing `--tuning_in` and `--tuning_labels_in` if a tuning set is available.

The Prediction methods allow you to omit `--tuning_in` in order to perform leave-one-out cross validation on the training data:

```
# skytree-server       wnnc              \
  --k_neighbors        10:5:20           \
  --k_neighbors        10,20,50          \
  --dist_weight        fixed_epan        \
  --bandwidth          0.1:0.1:0.2       \
  --bandwidth          0.3:0.1:0.5       \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --imbalance                            \
  --sample_imbalance                     \
  --imbalance_scale    2
```

The above command will tune over 54 parameter configurations and report parameter combinations that produce the highest Gini, highest Gini to Capture Deviation ratio, highest F-Score, and highest Accuracy respectively. Refer to *Scoring* (page 151) for more information about scoring.

Note that tuning and testing are not currently supported at the same time. Instead, the (tuning-related) input required to reproduce a winning model is reported, for example for best Gini:

```
To reproduce the model, use '--imbalance --k_neighbors=15 \
--k_neighbors=20 --bandwidth=0.2 --bandwidth=0.3 \
--probability_threshold=0.8208208919380429'
```

The optimal model can then be reproduced for testing as follows:

```
# skytree-server          wnnc              \
  --k_neighbors           15                \
  --k_neighbors           20                \
  --dist_weight           fixed_epan        \
  --bandwidth             0.2               \
  --bandwidth             0.3               \
  --training_in           sdss.train.st     \
  --training_labels_in    sdss.train.labels \
  --imbalance                               \
  --sample_imbalance                              \
  --imbalance_scale       2                       \
  --probability_threshold 0.8208208919380429 \
  --testing_in            sdss.test.st            \
  --scores_out            scores
```

Acceleration and Memory Usage Control

All of the performance-related options available to `nnplus` are available to `wncc`. See *Compressed Computation* (page 34), *Rank Approximate Nearest Neighbors* (page 34), and *Alpha-Beta Approximation* (page 35) for additional details.

Ensemble Learning Methods

The Skytree Server Ensemble Learning methods consist of implementations of the following methods for classification and regression:

- Random Decision Forests (page 83)
- Gradient Boosted Trees (page 86)
- Random Decision Forest Regression (page 94)
- Gradient Boosted Trees Regression (page 96)

Introduction

Executing classification and regression with Ensemble methods typically involves the following steps:

1. **Prepare**: Identify the training and the test datasets and the ensemble learning method to apply.
2. **Tune**: Find the right parameters (number of trees, sampling ratios etc.) for the model using *only* the training data.
3. **Test**: Once finalized, apply the model to test data and obtain results (labels, probabilities, targets, etc.).
4. **Score**: Evaluate model performance on the test set using Gini, accuracy, F-score, yield, etc.

Using the Ensemble methods, steps 2 and 3 (tuning and testing) can be easily combined, if preferred.

Ensemble Module Interface

The Random Decision Forest (`rdf|rdfr`) and Gradient Boosted Trees (`gbt|gbtr`) modules share similar input interfaces. For example, both modules have the same input data options, `--training_in` and `--testing_in`, and the same model validation options, `--num_folds` and `--holdout_ratio`. Output and other options are also consistent across both modules, but they differ in certain algorithm specific parameters. For example, `--learning_rate` can be specified in `gbt` but not in `rdf`.

The examples that follow will use the notation [`gbt|rdf`] to signify that they can use either the `rdf` or `gbt` module, with concepts remaining the same. Any example using features specific to one module will use only that module's name. Those examples will be in module specific sections.

Input Interface File

Options to the ensemble modules can not only be passed via the command-line interface, but also via an input file (or any combination).

For example, the following options:

```
# skytree-server        [gbt|rdf]           \
  --training_in         income.data.st      \
  --training_labels_in  income.data.labels  \
  --num_trees           100                 \
  --model_out           model.simple
```

can also be specified from an input file, `input`, containing the following lines:

```
 echo "training_in  = income.data.st
 training_labels_in = income.data.labels
 num_trees          = 100
 model_out          = model.simple" > input
```

The location of the input file is specified with the `--input_file` option:

```
# skytree-server [gbt|rdf] --input_file input
```

Input options can be read from the command-line and from an input file. The order of precedence is as follows:

1. Command-line
2. Input file
3. Input file [`tunable`] section (if applicable, see later section)

More examples are shown in later sections *Tuning Parameter Specification via Input File* (page 66) for GBT/RDF and *Tuning Parameter Specification via Input File* (page 77) for GBTR/RDFR.

Classification with gbt and rdf

This first example illustrates how to build a simple model and save it to a file. In the following command we provide the training data, training labels and the number of trees to build in the ensemble method. We then train the model and save it.

```
# skytree-server        [gbt|rdf]             \
  --training_in         income.data.st        \
  --training_labels_in  income.data.labels    \
  --num_trees           100                   \
  --model_out           model.simple
```

The resulting model is stored in `model.simple` which is in binary format. Also, the only parameter needed to run the ensemble method is the number of trees (all other parameters have default settings).

To load the model and apply it to a test set the following command may be run:

```
# skytree-server      [gbt|rdf]             \
  --model_in          model.simple          \
  --testing_in        income.test.st        \
  --probabilities_out probabilities.simple  \
  --labels_out        labels.simple
```

The results for the test data are now in `probabilities.simple` and `labels.simple`. Now we can run the scoring module to see what the accuracy of the generated model is in terms of Gini, classification accuracy, etc.

Now let's combine the above 2 steps into a single step where we train the model, save it and apply it to the test set to obtain probabilities and labels at the same time. This is done as follows:

```
# skytree-server       [gbt|rdf]            \
  --training_in        income.data.st       \
  --training_labels_in income.data.labels   \
  --testing_in         income.test.st       \
  --num_trees          100                  \
  --model_out          model.simple         \
  --probabilities_out  probabilities.simple \
  --labels_out         labels.simple
```

Handling Skewed Data

Often, the class distribution in the data is skewed, with far fewer points in one of the classes. To effectively classify such data, the Ensemble modules incorporate automatic class sampling via two approaches described in the next section.

Note: The dataset used in the next section, `income.data.st`, is not skewed and is used only for the purpose of illustration.

Specifying the Sampling Ratio

The `--sampling_ratio` argument can be used to down-sample or up-sample each class separately, in order to build a model which better accounts for the smaller class' distribution. This has the effect of sampling for each tree in the ensemble. Random Decision Forests by default sample the data up to a `--sampling_ratio` of `1`. In the Gradient Boosting Trees module sampling gives rise to the Stochastic Gradient Boosted Tree (http://en.wikipedia.org/wiki/Gradient_boosting#Stochastic_gradient_boosting) method.

The effect of setting `--sampling_ratio` to different values is illustrated with a simple example. Let's assume our dataset has 90 data points with the following distribution:

80 points are -1

10 points are +1

Also, let's assume that the training has to occur with the minority class `+1` up-sampled 2 times and the majority class `-1` down-sampled 4 times, thus effectively making the 2 classes have equal number of points in the sample. This is easily achieved by using the `--sampling_ratio` argument as follows:

```
# skytree-server       [gbt|rdf]              \
  --training_in        income.data.st         \
  --training_labels_in income.data.labels     \
  --testing_in         income.test.st          \
  --num_trees          100                     \
  --model_out          model.sampled           \
  --sampling_ratio     0.25                    \
  --sampling_ratio     2                       \
  --probabilities_out  probabilities.sampled   \
  --labels_out         labels.sampled
```

Notes:

- Notice there are two `--sampling_ratio` arguments. The first argument value (0.25) is applied to the class with -1 label and the second argument value is applied to the class with value +1.
- The default sampling method is sampling with replacement. To sample without replacement you may specify `--sample_with_replacement off`. Keep in mind that up-sampling is then *not* possible. In this mode, all sampling ratios must be > 0 and < 1.
- If only one `--sampling_ratio` argument is provided, both classes will be sampled in that ratio, i.e., it is the same as providing the *same* argument value for `--sampling_ratio` for each class separately.

Imbalance Handling

Instead of specifying the `--sampling_ratio`, the `--imbalance` option can be used to automatically handle imbalanced data and correct for its skewness. When the `--imbalance` option is used, the sampling ratio for both the classes are set internally such that the sampling ratio assigned to the minority class is 1, and the sampling ratio assigned to the majority class is calculated such that the final sample thus created has approximately the same number of minority and majority classes.

```
# skytree-server       [gbt|rdf]                 \
  --training_in        income.data.st            \
  --training_labels_in income.data.labels        \
  --testing_in         income.test.st            \
  --num_trees          100                       \
  --model_out          model.imbalance           \
  --imbalance                                    \
  --probabilities_out  probabilities.imbalance   \
  --labels_out         labels.imbalance
```

Note: The `--imbalance` option cannot be used together with the `--sampling_ratio` option.

In addition to the `--imbalance` option, a tunable `--imbalance_scale` option can also be provided. This option causes more data to be used when set to a value > 1 or less data when set to a value < 1.

Modified Class Weights

To accommodate situations where input reference sets are formed by stratified sampling of either class, the `--classweight` argument may be used to artificially inflate the impact of one class or the other. Suppose that the training set `income.data.st` is created by down-sampling the non-target class to a fifth of its original size. Then the ensemble training is performed as follows:

```
# skytree-server       [gbt|rdf]                 \
  --training_in        income.data.st            \
  --training_labels_in income.data.labels       \
  --testing_in         income.test.st           \
  --num_trees          100                      \
  --model_out          model.with.classweights  \
  --probabilities_out  probabilities.imbalance  \
  --labels_out         labels.imbalance         \
  --classweight        5                        \
  --classweight        1
```

The `--classweight` parameter must be repeated for each class in `--training_labels_in` and its values are assigned according to the numerical order of the class labels. In this case, the `--classweight=5` is for class -1 while class 1 still has `--classweight=1`. More details regarding the usage of class weights in Skytree Server is presented in *Class Weights* (page 164).

Tuning the Model

The process of tuning for the right parameter settings usually involves the following steps:

1. Split the original training data into a new tuning and new training dataset.
2. Train the model with the new training dataset and a setting of parameters.
3. Assess accuracy of model on new tuning data.
4. Possibly repeat the above 3 steps many times and average the accuracy.

The above 4 steps are then repeated for multiple parameter settings, and the parameter settings with the best accuracy (based on user defined measures like Gini, classification accuracy, F-score, etc.) are chosen to build the final production model. This is called Cross-Validation (http://en.wikipedia.org/wiki/Cross-validation_(statistics)).

There are a few general ways to do cross-validation:

- Holdout set: Splitting the training data randomly into a training dataset and a tuning dataset. Typically, between 20-30% is reserved for tuning. The rest is used for training.
- Multiple restarts of holdout sets: The above process can be repeated many times, each time randomly generating a new training and tuning split. The accuracy results over all runs are then averaged.
- K-Fold cross validation.
- User supplied tuning dataset and labels.

Skytree Server Ensemble modules are able to do all of the above methods for tuning parameters automatically. Simple examples are given in the next sections.

Tuning the Number of Trees with a Holdout Set

To automatically holdout a random portion of the training data the `--holdout_ratio` argument can be used.

```
# skytree-server       [gbt|rdf]          \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --num_trees          100
```

The command above will hold out 20% of the training data for use in evaluating the model. Though no tuning parameters were explicitly specified, the presence of a tuning set automatically puts Skytree Server into tuning mode. Any *tunable* parameters will be tuned given the user provided values, ranges and/or step sizes.

In the above example the only argument that can be tuned is the number of trees. The default behavior is to score the model using the tuning set after every 10% of iterations (in the example above, every 10 trees). Thus the model will build 10 trees and then score on the holdout data and report. Then it will go on to build 20 trees and score on the holdout data and so on reporting accuracy scores each step of the way.

E.g., to score every 5 iterations instead of every 10 in the example above, run:

```
# skytree-server       [gbt|rdf]          \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --num_trees          5:5:100            \
  --loglevel           verbose
```

This will search for the best value of `--num_trees` in the set 5, 10, 15, ..., 95, 100. Notice that by specifying `--loglevel=verbose` scoring results are reported as the model is generated.

Note: Tuning the number of trees does not significantly affect runtime.

Upon completion, the model parameters giving the ten best Ginis, F-Scores, accuracies and capture deviations, respectively, are reported. For each of the best models (one per metric) all parameters are reported as part of an option string that can be used to reproduce the models. All tuning results can be saved to a file using the `--tuning_results_out` option. Users can specify whether the format of this output file is JSON or CSV using the `--tuning_results_format` option. By default, the tuning results output format is CSV.

When scoring, the F-Scores and classification accuracies reported are those obtained by optimizing the probability threshold used in classification. This threshold is reported for the best models as the `--probability_threshold` in the option string described above. By default, when considering the models with the best Gini and capture deviation (which depend only probabilities, not classification), the probability threshold giving the highest F-Score is used. If the accuracy of the model is of greater interest, then specify `--classification_objective=accuracy`. For further information, please refer to *The Probability Threshold* (page 73).

Tuning the Number of Trees using K-Fold Cross Validation

Tuning can also be done using K-fold cross validation. This is better than using a single hold out set which is randomly generated as above. The downside is that it can be approximately `K` times slower. One strategy is to hone in on approximate parameters using a holdout set and then fine tune the parameters using K-fold cross validation. In the above we replace `--holdout_ratio` with `--num_folds`. The parameters with the best average results over all folds are returned:

```
# skytree-server       [gbt|rdf]          \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --num_folds          5                  \
  --num_trees          5:5:100            \
  --loglevel           verbose
```

If both --num_folds and --holdout_ratio are provided then the algorithm will not do K-fold cross validation but instead will repeat --num_folds times with a new holdout tuning set and return the parameters with the best average results over these --num_folds runs.

Tuning the Number of Trees using User-Supplied Tuning Data

You can generate a tuning dataset separately that should be used for tuning as well. This is done as follows:

```
# skytree-server       [gbt|rdf]          \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --tuning_in          income.tune.st     \
  --tuning_labels_in   income.tune.labels \
  --num_trees          100
```

In this case the model will be built and scored on the input tuning dataset provided.

You can also specify a --tuning_models_out option, which allows all models trained during the tuning phase to be written to files using a user-specified prefix. When tuning over the number of trees, Skytree Server generates a single model file storing all of the trees tuned over rather than generating separate lists. To limit the number of trees when using this model, specify the --num_trees option when loading this model with the --model_in. An index file is also written, giving the parameters used for each model. If a test dataset was specified via the --testing_in option, the appropriate model will be read during the testing phase instead of being re-trained.

Tuning for Skewed Data

The --sampling_ratio option can also be tuned by specifying appropriate ranges for the option(s). The following command will test four different parameter combinations for the sampling ratio (0.75, 0.25), (0.75, 0.5), (1.0, 0.25), (1.0, 0.5):

```
# skytree-server       [gbt|rdf]          \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --num_trees          100                \
  --sampling_ratio     0.75,1.0           \
  --sampling_ratio     0.25,0.5
```

If only a single ratio is specified, the cross-combinations of the parameters will not be used. Thus, this command produces only two parameter combinations:

```
# skytree-server       [gbt|rdf]          \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --num_trees          100                \
  --sampling_ratio     0.25,0.5
```

When automatically handling skewed data via the --imbalance option, the --imbalance_scale parameter can also be specified and tuned:

```
# skytree-server       [gbt|rdf]          \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --num_trees          100                \
  --imbalance                             \
  --imbalance_scale    0.5:0.5:3.0
```

Tuning Parameter Specification via Input File

As mentioned in the previous section, the previous tuning example can also be specified from an input file:

```
# skytree-server [gbt|rdf]  \
  --input_file   input
```

with the following content of the file `input`:

```
training_in        = income.data.st
training_labels_in = income.data.labels
num_trees          = 100
model_out          = model.simple
holdout_ratio      = 0.2
imbalance          = on
imbalance_scale    = 0.5:0.5:3.0
```

Moreover, multiple such tuning parameter scans be specified at once by specifying multiple `[tunable]` sections containing at least one tunable parameter per section:

```
training_in        = income.data.st
training_labels_in = income.data.labels
model_out          = model.simple
holdout_ratio      = 0.2
imbalance          = on
```

```
[tunable]
num_trees          = 100:10:200
imbalance_scale    = 0.5:0.5:3.0
```

```
[tunable]
num_trees          = 100:20:500
imbalance_scale    = 0.5:0.05:1.0
```

Option Order of Precedence

One complete set of options is assembled for each `[tunable]` section. Command-line options have highest priority, followed by options from the non-`[tunable]` (top) part of the input file (if any), then followed by the tuning options from each corresponding `[tunable]` section. The input file may contain only `[tunable]` sections, or it may contain only non-tunable options, or any combination thereof.

The order of precedence for options is:

1. Command-line
2. Input file
3. Input file `[tunable]` section(s)

In the following example, the value of `num_trees` will be ignored for all `[tunable]` sections, because the first specification of `num_trees`=100 will override any subsequent settings. As a (possibly unintended) consequence, the input file below will run three tuning runs with the same `num_trees`=100 option, ignoring `num_trees`=500, `num_trees`=1000 and `num_trees`=5000:

```
# skytree-server       rdf                \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --input_file         input              \
  --num_dimensions     2,5
```

with input file `input`

```
num_trees          = 100 <== overrides values below
holdout_ratio      = 0.2
imbalance          = on
smoothing          = 0,0.001,0.01
```

```
[tunable]
num_trees          = 500
imbalance_scale    = 0.5,0.6
```

```
[tunable]
num_trees          = 1000
imbalance_scale    = 0.5:0.1:1.0
```

```
[tunable]
num_trees          = 5000
imbalance_scale    = 0.5:0.5:3.0
```

Tuning Classifiers with Score Weights

With Skytree Server, the predictions of the learned classifiers can be scored using weights to get the weighted classification score (see *Weighted Scoring* (page 169) for details on weighted scoring). The Ensemble learning methods allow the user to tune for model parameters with respect to the weighted classification score. This is accomplished by scoring the predictions for the tuning set (in each tuning fold) with respect to the weights associated with the individual tuning points. These score weights do not affect the model in any way. Score weights only factor in when comparing the performances of different models that are evaluated using the score weights on the tuning data.

If the user supplies the tuning data using the `--tuning_in` option, the weights associated with the individual points in the tuning data can be specified with the `--tuning_score_weights_in` option. For example, tuning for the number of trees (that is, the forest size) with the user supplied tuning data (specified with `--tuning_in`) and corresponding weights (specified with `--tuning_score_weights_in`) can be accomplished as follows:

```
# skytree-server              [gbt|rdf]           \
  --training_in               income.data.st      \
  --training_labels_in        income.data.labels  \
  --tuning_in                 income.tune.st      \
  --tuning_labels_in          income.tune.labels  \
  --tuning_score_weights_in   income.tune.weights \
  --num_trees                 100
```

This will choose the best forest size with respect to the weighted tuning score. The score weights can also be used if we are using holdout sets from the training data for tuning. For example, tuning for the number of trees with a holdout set from the training data (specified with `--holdout_ratio`) and its corresponding weights (specified with `--training_score_weights_in`) can done as follows:

```
# skytree-server                [gbt|rdf]           \
  --training_in                 income.data.st      \
  --training_labels_in          income.data.labels  \
  --training_score_weights_in   income.data.weights \
  --holdout_ratio               0.2                 \
  --num_trees                   100
```

This command will holdout 20% of the training data for evaluating the model learned on the rest 80% of the training data. The specification of the `--training_score_weights_in` implies that the weights associated with the points in the holdout set (20% of the training data) will be used to score the predictions of the learned model on the holdout set and the best parameter setting will chosen with respect to the weighted tuning score.

In a similar fashion, the weights can be specified in K-fold cross validation. In this case, the weighted scoring in each fold will be done with respect to the weights of the individual points in the holdout set used for tuning in that fold. For example, the 5-fold cross validation with weights is done as follows:

```
# skytree-server                [gbt|rdf]           \
  --training_in                 income.data.st      \
  --training_labels_in          income.data.labels  \
  --training_score_weights_in   income.data.weights \
  --num_folds                   5                   \
  --num_trees                   100                 \
  --loglevel                    verbose
```

Training Classifiers with Point Weights

Many times it is viable to put weights on certain points when building a model. A simple example of this may be that sometimes the training data may have multiple repeats of the same point. For example, if a point is repeated 20 times and represented with 20 rows in the training data, this could be substituted with a point weight of 20 and 1 row in the training data. More advanced versions of this concept involve continuous point weights where for example a point may be up-weighted in the training data by 10% and thus have a point weight of 1.1. This may be done for many reasons - time based weighting of points in sliding window techniques, cost weighting as part of the trained model etc. In other words given a set of training data and a set of training point weights the model is built with each point represented point weight times. This is different from score weights since score weights are never seen in the training phase and are only used when a model has finished building and the score modules are called. Point weights are the opposite, they are only seen when the model is being built and have nothing to do with scoring.

The only way to specify point weights is through the `--training_point_weights_in` option. There must be one row in this file per row in the `--training_in` file. A simple example:

```
# skytree-server                [gbt|rdf]           \
  --training_in                 income.data.st      \
  --training_labels_in          income.data.labels  \
  --training_point_weights_in   income.data.weights \
  --testing_in                  income.test.st      \
  --num_trees                   100                 \
  --labels_out                 labels.pointweights
```

An example involving tuning is presented next:

```
# skytree-server               [gbt|rdf]           \
  --training_in                income.data.st      \
  --training_labels_in         income.data.labels  \
  --training_point_weights_in  income.data.weights \
  --holdout_ratio              0.2                 \
  --testing_in                 income.test.st      \
  --num_trees                  100                 \
  --labels_out                 labels.pointweights.holdout
```

This command will holdout 20% of the training data for tuning the model learned on the rest 80% of the training data. The specification of the `--training_point_weights_in` implies that the weights associated with the points in the training set (the randomly held out 80% of the training data) will be used to train the model. The point weights for 20% of the data held out for tuning will be ignored. Similar reasoning will follow when `--num_folds` is specified for each fold produced for cross validation.

Tuning and Testing

Skytree Server provides the following methods for tuning and testing:

- Tuning for Gini (default) (page 69)
- Tuning for Best Classification Accuracy (page 69)
- Tuning for Yield (page 70)

Tuning for Gini (default)

A testing dataset can be specified via the `--testing_in` option when Skytree Server runs in tuning mode. Skytree Server will first tune for the best parameters as above and use the model giving the *best Gini* (by default) for prediction using the test set. If `--model_out` is specified then the best model will be stored to a file as well.

If a holdout dataset was generated, a new model will first be generated using the full training dataset.

The following command runs the entire process:

```
# skytree-server         [gbt|rdf]             \
  --training_in          income.data.st        \
  --training_labels_in   income.data.labels    \
  --holdout_ratio        0.2                   \
  --testing_in           income.test.st        \
  --num_trees            5:5:100               \
  --num_dimensions       1:1:4                 \
  --probabilities_out    probabilities.tuning  \
  --labels_out           labels.tuning
```

Tuning for Best Classification Accuracy

If it is desired to pick the model with the best accuracy rather than the best Gini the `--testing_objective` argument can be used. This is illustrated in the following example:

```
# skytree-server        [gbt|rdf]            \
  --training_in         income.data.st       \
  --training_labels_in  income.data.labels   \
  --holdout_ratio       0.2                  \
  --testing_in          income.test.st       \
  --num_trees           5:5:100              \
  --num_dimensions      1:1:4                \
  --probabilities_out   probabilities.tuning \
  --labels_out          labels.tuning        \
  --testing_objective   accuracy
```

Note: If a holdout dataset was generated, a new model will first be generated using the full training dataset.

Tuning for Yield

A `--yield_values_in` option can be used to specify a file for calculating yield values during tuning. This file must be the same length as either the training or tuning vector, depending on whether a tuning table or holdouts are used.

```
# skytree-server         [gbt|rdf]          \
  --training_in          yield.data.st      \
  --training_labels_in   yield.data.labels \
  --num_folds            2                  \
  --model_out            model              \
  --num_trees            10:10:100          \
  --max_splits           2:2:12             \
  --tree_depth           3,8                \
  --learning_rate        .1,.25             \
  --yield_values_in      yield.data.values \
  --log                  run.log
```

You can also specify `--testing_objective=yield`. In this case, the yield value will be used to determine the best model. A `--yield_values_in` file is required if `--testing_objective=yield` is specified; otherwise it is optional.

The logs show the best tuning results for Gini and Yield as well as the parameters used to reproduce these models.

```
========================================================================
Best tuning results for Gini
========================================================================
Metrics:
-------------------------------------------------------------------------------------------
|   #   |      Gini  | Capt. dev. |     Prec@k |    F-Score |   Accuracy |      Yield |
-------------------------------------------------------------------------------------------
|     1 |   0.995849 |   0.023764 |   0.974207 |   0.981042 |   0.994205 |   0.989117 |
|     2 |   0.995816 |   0.023237 |   0.975103 |   0.981042 |   0.994205 |   0.986329 |
|     3 |   0.995747 |   0.020803 |   0.975836 |   0.981042 |   0.994205 |   0.990965 |
|     4 |   0.995722 |   0.024440 |   0.975836 |   0.980707 |   0.994103 |   0.974812 |
|     5 |   0.995680 |   0.021182 |   0.973433 |   0.980069 |   0.993897 |   0.990597 |
|     6 |   0.995657 |   0.014581 |   0.976541 |   0.979697 |   0.993798 |   0.964130 |
|     7 |   0.995505 |   0.029381 |   0.971339 |   0.981361 |   0.994303 |   0.980777 |
|     8 |   0.995500 |   0.028040 |   0.974681 |   0.981025 |   0.994202 |   0.990709 |
|     9 |   0.995497 |   0.035927 |   0.971315 |   0.981361 |   0.994303 |   0.980381 |
|    10 |   0.995490 |   0.023865 |   0.975201 |   0.978421 |   0.993392 |   0.973755 |
========================================================================
Best tuning results for Yield
========================================================================
Metrics:
-------------------------------------------------------------------------------------------
|   #   |       Gini  | Capt. dev. |     Prec@k |    F-Score |   Accuracy |      Yield |
-------------------------------------------------------------------------------------------
|     1 |   0.994347 |   0.073209 |   0.964396 |   0.977451 |   0.993106 |   0.994363 |
|     2 |   0.995375 |   0.023597 |   0.974446 |   0.979748 |   0.993805 |   0.994200 |
|     3 |   0.993837 |   0.077078 |   0.962682 |   0.977451 |   0.993106 |   0.994029 |
|     4 |   0.995328 |   0.025753 |   0.972265 |   0.979748 |   0.993805 |   0.993818 |
|     5 |   0.990495 |   0.465575 |   0.961564 |   0.943534 |   0.983529 |   0.993460 |
|     6 |   0.990495 |   0.465575 |   0.961564 |   0.943534 |   0.983529 |   0.993460 |
|     7 |   0.990495 |   0.465575 |   0.961564 |   0.943534 |   0.983529 |   0.993460 |
|     8 |   0.990485 |   0.438614 |   0.961505 |   0.943534 |   0.983529 |   0.993460 |
|     9 |   0.990485 |   0.438614 |   0.961505 |   0.943534 |   0.983529 |   0.993460 |
|    10 |   0.990485 |   0.438614 |   0.961505 |   0.943534 |   0.983529 |   0.993460 |

Parameters to reproduce the model with the best Gini:
--num_trees=80 --tree_depth=8 --max_splits=10 --min_node_weight=0.000000 --learning_rate=0.250000
--regularization --regularization_bins=200 --trim=off --categorical_sampling_seed=23085656
--probability_threshold=0.0705677047256971 --compression=on

Parameters to reproduce the model with the best Yield:
--num_trees=100 --tree_depth=3 --max_splits=4 --min_node_weight=0.000000 --learning_rate=0.250000
--regularization --regularization_bins=200 --trim=off --categorical_sampling_seed=23085656
--probability_threshold=0.2845532632717532 --compression=on
```

Using the `--yield_values_in` option, the results can be scored as follows:

```
# skytree-server          score              \
  --yield_values_in       yield.data.values  \
  --predicted_labels_in   labels.tuning
```

Multi-Class Classification

Until now, we have only referred to binary classification problems. Skytree Server also supports multi-class classification problems in the Random Decision Forest and Gradient Boosted Trees modules. The interface for multi-class classification is essentially the same as that for binary classification except that certain features are disabled when the labels presented have more than 2 classes.

Here is a simple way to run multi-class classification:

```
# skytree-server        [gbt|rdf]                           \
  --training_in         letter-recognition.data.train.st     \
  --training_labels_in  letter-recognition.data.train.labels \
  --num_trees           100                                  \
  --testing_in          letter-recognition.data.test.st      \
  --labels_out          labels                               \
  --probabilities_out   probabilities                        \
  --output_with_ids
```

The results in the labels file above can be scored as follows:

```
# skytree-server        score                               \
  --true_labels_in      letter-recognition.data.test.labels  \
  --predicted_labels_in labels
```

Scoring outputs a per-class confusion matrix, an aggregated confusion matrix, and the classification accuracy of the model.

```
Confusion Matrix:
+-------------------------------------------------------------------+
|   Labels        |    Pred 0 |    Pred 1 |    ...    |    Pred 9 |
|===================================================================|
|          True 0 |       13  |       11  |    ...    |        9  |
|-----------------+-----------+-----------+-----------+-----------+
|          True 1 |        9  |       10  |    ...    |       13  |
|-----------------+-----------+-----------+-----------+-----------+
|          True 2 |        6  |        9  |    ...    |       11  |
|-----------------+-----------+-----------+-----------+-----------+
|          ...    |   ...     |   ...     |    ...    |   ...     |
|-----------------+-----------+-----------+-----------+-----------+
|          True 9 |       12  |       13  |    ...    |        9  |
+-------------------------------------------------------------------+
Aggregate:
+-----------------------------------------------------------------+
|       Class         |    Total    |    Right    |    Wrong    |
|=================================================================|
|                   0 |         110 |          13 |          97 |
|---------------------+-------------+-------------+-------------|
|                   1 |         119 |          10 |         109 |
|---------------------+-------------+-------------+-------------|
|                   2 |         105 |           7 |          98 |
|---------------------+-------------+-------------+-------------|
|              ...    |      ...    |      ...    |      ...    |
|---------------------+-------------+-------------+-------------|
|                   9 |          91 |           9 |          82 |
|---------------------+-------------+-------------+-------------|
|             Overall |        1000 |          99 |         901 |
+-----------------------------------------------------------------+
Classification Accuracy: 0.099
```

The above `skytree-server score` configuration doesn't have any specific toggle for changing to multi-class mode. Skytree Server will **automatically detect if the problem is multi-class** based only on the `--training_in` dataset. The `--testing_in` and `--tuning_in` datasets are not used to shift to multi-class mode, so if the training dataset does not contain more than two labels, then the problem is assumed to be binary. In any case, if the training data does not have more than two labels, there is no way for the model to predict outside of these two labels.

The following is a more advanced example and illustrates that the tuning interface and results interface for the `rdf` module do not change for multi-class problems.

```
# skytree-server       rdf                                   \
  --training_in        letter-recognition.data.train.st      \
  --training_labels_in letter-recognition.data.train.labels  \
  --holdout_ratio      0.3                                   \
  --num_trees          10:10:100                             \
  --num_dimensions     2,6,8                                 \
  --testing_in         letter-recognition.data.train.st      \
  --labels_out         labels                                \
  --probabilities_out  probabilities
```

A similarly advanced example for GBT would be as follows:

```
# skytree-server       gbt                                \
  --training_in        letter-recognition.data.train.st     \
  --training_labels_in letter-recognition.data.train.labels \
  --holdout_ratio      0.3                                  \
  --num_trees          10:10:100                            \
  --learning_rate      0.06:0.02:0.14                       \
  --testing_in         letter-recognition.data.train.st     \
  --labels_out         labels                               \
  --probabilities_out  probabilities                        \
  --fast_read
```

Limitations

Presently there are some restrictions on the options allowed in Skytree Server multi-class RDF and GBT. These are enumerated below.

- `--sampling_ratio`, `--imbalance_scale`, `--sample_with_replacement`, and `--imbalance` are not supported. For RDF, the training data is sampled with replacement up to the size of the training dataset. For GBT, no sampling is performed.
- `--classification_objective` and `--probability_threshold` are specific to binary classification problems as well.
- `--smoothing` is not supported in RDF because it is only applicable to binary problems.
- `--ensemble_size`, `--trim`, and `--test_point_variable_importances_out` are not supported with GBT multi-class.
- `--classweight` is not supported.
- `--testing_objective` is set to `accuracy`. This is the only allowed setting because other metrics such as GINI are not well defined for multi-class.

The Probability Threshold

In ensemble methods the predicted class label is derived from the probabilities generated for being of a particular class type. For example the probabilities that are output by the Ensemble modules are the probability of being class `+1`.

Typically, if for any given point, the probability of being +1 is > 0.5, then we assign that point the label of +1. This is the default behavior of Skytree Server as well when it is not doing automatic tuning. However, a probability threshold of 0.5 or 50% is not necessarily the best value for highest classification accuracy, or for that matter any scoring metric that utilizes the labels output. Therefore, when Skytree Server is in tuning mode it automatically reports the best probability threshold for any given metric.

Two cases arise depending on the `--testing_objective`.

1. `--testing_objective` is `accuracy` or `fscore` i.e. label based metric is being used to pick the best model. In this case the probability threshold is automatically tuned and applied in tuning as well as testing and stored with the model. It is also reported in the log. Therefore if `--testing_objective` is accuracy, then all the scores and models will use the best possible probability threshold for `accuracy`. Similarly for `fscore`.

2. `--testing_objective` is `gini` or `capture_dev` or `precision_at_k`, i.e. probability based metric is being used to pick the best model.
   a. In this case, if the `--testing_objective` is `gini` or `capture_dev`, then the `--classification_objective` argument comes into play. If `--classification_objective` is `accuracy`, then the model will still be picked based on best `gini` or `capture_dev` but the probability threshold will be tuned for `accuracy` and the labels outputted for the test data will be based on this probability threshold. Similarly, for `fscore`.
   b. In this case, if the `--testing_objective` is `precision_at_k` (where $k \in (0,1)$ is specified via `--k_for_precision` and defaults to 0.1), the `--classification_objective` is not used at all. The probability threshold is chosen to correspond to the top 100*k*_th percentile probability. This implies that any test point with the probability of being +1 higher than this threshold is probably in the top 100*k*-th percentile.

Regression with gbtr and rdfr

The examples that follow will use [`gbtr|rdfr`] to signify that they can use either the `rdfr` or `gbtr` module, with concepts remaining the same. Any example using features specific to one module will use only that module's name. Those examples will be in module specific sections.

The first example illustrates how to build a simple model and save it to a file. In the following command we provide the training and testing data, training and testing targets, and the number of trees to build in the ensemble method. We then train the model and save it.

```
# skytree-server        [gbtr|rdfr]          \
  --training_in         income.data.st       \
  --training_targets_in income.data.targets  \
  --testing_in          income.test.st       \
  --num_trees           100                  \
  --model_out           model.simple         \
  --targets_out         targets.[gbtr|rdfr]
```

Given actual regression targets `income.data.targets` for the training points in `income.train.st`, this will produce a file, `targets.rdfr` or `targets.gbtr`, containing predicted labels for the test points in `income.test.st`.

Run the following command to produce the regression error metrics from the two target files:

```
# skytree-server         score                \
  --true_targets_in      income.test.targets  \
  --predicted_targets_in targets.[gbtr|rdfr]
```

The output will contain the following metrics:

- Mean absolute error
- Mean squared error
- Root mean squared error
- L1 error
- L2 error
- Relative L1 error
- Relative L2 error

- Coefficient of determination
- Normalized Gini

Tuning the Model

The process of tuning for the right parameter settings usually involves the following steps:

1. Split the original training data into a new tuning and new training dataset.
2. Train the model with the new training dataset and a setting of parameters.
3. Assess accuracy of model on new tuning data.
4. Possibly repeat the above 3 steps many times and average the accuracy.

The above 4 steps are then repeated for multiple parameter settings and the parameter settings with the best accuracy (based on user defined measures like mean absolute error, mean squared error, etc.) are chosen to build the final production model. This is called Cross-Validation (http://en.wikipedia.org/wiki/Cross-validation_(statistics)).

As with classification, there are a few general ways to do cross-validation:

- Holdout set: Splitting the training data randomly into a training dataset and a tuning dataset. Typically, between 20-30% is reserved for tuning. The rest is used for training.
- Multiple restarts of holdout sets: The above process can be repeated many times, each time randomly generating a new training and tuning split. The accuracy results over all runs are then averaged.
- K-Fold cross validation.
- User supplied tuning dataset and labels.

The Ensemble modules are able to do all of the above methods for tuning parameters automatically. Simple examples are given in the next sections.

Tuning the Number of Trees with a Holdout Set

To automatically holdout a random portion of the training data the `--holdout_ratio` argument can be used.

```
# skytree-server        [gbtr|rdfr]           \
  --training_in         income.data.st        \
  --training_targets_in income.data.targets   \
  --holdout_ratio       0.2                   \
  --num_trees           100
```

The command above will hold out 20% of the training data for use in evaluating the model. Though no tuning parameters were explicitly specified, the presence of a tuning set automatically puts Skytree Server into *tuning mode*. Any *tunable* parameters will be tuned given the user provided values, ranges and/or step sizes.

In the above example, the only argument that can be tuned is the number of trees. The default behavior is to score the model using the tuning set after every 10% of iterations (in the example above, every 10 trees). Thus the model will build 10 trees and then score on the holdout data and report. Then it will go on to build 20 trees and score on the holdout data and so on, reporting scores each step of the way.

E.g., to score every 5 iterations instead of every 10 in the example above, run:

```
# skytree-server         [gbtr|rdfr]             \
  --training_in          income.data.st          \
  --training_targets_in  income.data.targets     \
  --holdout_ratio        0.2                     \
  --num_trees            5:5:100                 \
  --loglevel             verbose
```

This will search for the best value of `--num_trees` in the set 5, 10, 15, ..., 95, 100. Notice that by specifying `--loglevel=verbose` scoring results are reported as the model is generated.

Note: Tuning the number of trees does not significantly affect runtime.

Upon completion, the model parameters giving the ten best mean absolute errors, mean squared errors, normalized Gini, and coefficient of determination are reported. For each of the best models (one per metric) all parameters are reported as part of an option string that can be used to reproduce the models. All tuning results can be saved to a file using the `--tuning_results_out` option. Users can specify whether the format of this output file is JSON or CSV using the `--tuning_results_format` option. By default, the tuning results output format is CSV.

Tuning the Number of Trees using K-Fold Cross Validation

Tuning can also be done using K-fold cross validation. This is better than using a single hold out set which is randomly generated as above. The downside is that it can be approximately K times slower. One strategy is to hone in on approximate parameters using a holdout set and then fine tune the parameters using K-fold cross validation. In the above we replace `--holdout_ratio` with `--num_folds`. The parameters with the best average results over all folds are returned:

```
# skytree-server         [gbtr|rdfr]            \
  --training_in          income.data.st         \
  --training_targets_in  income.data.targets    \
  --num_folds            5                      \
  --num_trees            5:5:100                \
  --loglevel             verbose
```

If both `--num_folds` and `--holdout_ratio` are provided then the algorithm will not do K-fold cross validation but instead repeat `--num_folds` times with a new holdout tuning set and return the parameters with the best average results over these `--num_folds` runs.

Tuning the Number of Trees using User-Supplied Tuning Data

The user can generate a tuning dataset separately that should be used for tuning as well. This is done as follows:

```
# skytree-server         [gbtr|rdfr]             \
  --training_in          income.train.st         \
  --training_targets_in  income.train.targets    \
  --tuning_in            income.tune.st          \
  --tuning_targets_in    income.tune.targets     \
  --num_trees            100
```

In this case the model will be built and scored on the input tuning dataset provided.

The `--tuning_models_out` option allows all models trained during the tuning phase to be written to files using a user-specified prefix. An index file is also written giving the parameters used for each model. If a test dataset was also specified via the `--testing_in` option, the appropriate model will be read during the testing phase instead of being re-trained.

Selecting the Testing/Output Model

If Skytree Server runs in tuning mode along with either the `--testing_in` and/or `--model_out` option, the tuned model with the best mean absolute error will be used for testing and/or file output. You can control this behavior by specifying the `--testing_objective`, which can take values of `mean_absolute_error` (default), `mean_squared_error`, `normalized_gini`, or `coeff_determination`. (Refer to *Normalized Gini* (page 166) and *Coefficient of Determination* (page 165) for additional information.)

Tuning Parameter Specification via Input File

As mentioned above, the previous tuning example can also be specified from an input file:

```
# skytree-server [gbtr|rdfr] \
  --input_file   input
```

with the following content of the file `input`:

```
training_in          = income.data.st
training_targets_in  = income.data.targets
num_trees            = 100
holdout_ratio        = 0.2
```

Moreover, multiple such tuning parameter scans be specified at once by specifying multiple `[tunable]` sections containing at least one tunable parameter per section:

```
training_in = income.data.st
training_targets_in = income.data.targets
holdout_ratio = 0.2
```

```
[tunable]
num_trees = 20:20:500
num_dimensions = 4,5,9
```

```
[tunable]
num_trees = 200:50:1000
tree_depth = 0,10,20
```

```
[tunable]
num_trees = 10:5:200
tree_depth = 0:2:10
```

Order of Precedence

One complete set of options is assembled for each `[tunable]` section. Command-line options have highest priority, followed by options from the non-`[tunable]` (top) part of the input file (if any), then followed by the tuning options from each corresponding `[tunable]` section. The input file can contain only `[tunable]` sections, or it may contain only non-tunable options, or any combination thereof.

The order of precedence for options is as follows:

1. Command-line
2. Input file

3. Input file `[tunable]`section

In the following example, the value of `num_trees` will be ignored for all `[tunable]` sections, because the first specification of `num_trees`=100 will override any subsequent settings. As a (possibly unintended) consequence, the input file below will run three tuning runs with the same `num_trees`=100 option, ignoring `num_trees`=500, `num_trees`=1000, and `num_trees`=5000:

```
# skytree-server         rdfr               \
  --training_in          income.data.st     \
  --training_targets_in  income.data.targets \
  --input_file           input              \
  --num_dimensions       2,5
```

with input file `input`

```
num_trees          = 100 <== overrides values below
holdout_ratio      = 0.2
```

```
[tunable]
num_trees          = 500
tree_depth         = 2,3
```

```
[tunable]
num_trees          = 1000
tree_depth         = 2:1:5
```

```
[tunable]
num_trees          = 5000
tree_depth         = 2:2:10
```

Tuning Regressors with Score Weights

Similar to classifier tuning with score weights (*Tuning Classifiers with Score Weights* (page 67)), the Ensemble Learning methods allow the user to tune for model parameters with respect to the weighted regression score (see *Weighted Scoring* (page 169) for details on weighted regression scoring). This is accomplished by scoring the predictions for the tuning set (in each tuning fold) with respect to the weights associated with the individual tuning points.

If the user supplies the tuning data using the `--tuning_in` option, the weights associated with the individual points in the tuning data can be specified with the `--tuning_score_weights_in` option. For example, tuning for the number of trees (that is, the forest size) with the user supplied tuning data (specified with `--tuning_in`) and corresponding weights (specified with `--tuning_score_weights_in`) can be accomplished as follows:

```
# skytree-server             [gbtr|rdfr]          \
  --training_in              income.train.st      \
  --training_targets_in      income.train.targets \
  --tuning_in                income.tune.st       \
  --tuning_targets_in        income.tune.targets  \
  --tuning_score_weights_in  income.tune.weights  \
  --num_trees                100
```

This will choose the best forest size with respect to the weighted tuning regression score. The score weights can also be used if we are using holdout sets from the training data for tuning. For example, tuning for the number of trees with a holdout set from the training data (specified with `--holdout_ratio`) and its corresponding weights (specified with `--training_score_weights_in`) can done as follows:

```
# skytree-server               [gbtr|rdfr]         \
  --training_in                income.data.st      \
  --training_targets_in        income.data.targets \
  --training_score_weights_in income.data.weights \
  --holdout_ratio              0.2                 \
  --num_trees                  100                 \
  --loglevel                   verbose
```

This command will holdout 20% of the training data for evaluating the model learned on the rest 80% of the training data. The specification of the `--training_score_weights_in` implies that the weights associated with the points in the holdout set (20% of the training data) will be used to score the predictions of the learned model on the holdout set and the best parameter setting will chosen with respect to the weighted tuning score.

In a similar fashion, the weights can be specified in K-fold cross validation. In this case, the weighted scoring in each fold will be done with respect to the weights of the individual points in the holdout set used for tuning in that fold. For example, the 5-fold cross validation with weights is done as follows:

```
# skytree-server               [gbtr|rdfr]         \
  --training_in                income.data.st      \
  --training_targets_in        income.data.targets \
  --training_score_weights_in income.data.weights \
  --num_folds                  5                   \
  --num_trees                  100                 \
  --loglevel                   verbose
```

Missing Values

The sections that follow describe how Skytree Server handles missing values in testing and training data.

Training

A common technique of dealing with missing data is imputation i.e. when some value is used to replace the missing value before training begins. While this technique has its merits, it may be preferable to handle missing values that occur in the training data within the model itself. Tree-based methods have several techniques that elegantly handle missing values. In Skytree Server, the tree-based methods, `gbt`, `rdf`, `rdfr`, and `gbtr` handle missing values by creating ternary splits when evaluating dimensions that may have missing values for possible split points. In other words, when missing values are present in the data for a particular tree node and attribute, the split point is chosen based on a 3-way split into the missing node, left child, and right child. This has the advantage of naturally modeling missing value behavior in different parts of the space as dictated by the method being used.

Testing

Missing values can occur even in testing data. The problem is exacerbated here because you may have patterns of missing data in the test data that were not present in the training dataset. Complex patterns can emerge in different parts of the data space. In the training data, for example, it is possible that an attribute *age* (for example) had no missing *income* values for all *age* values < 18. But the test data may have many points in which age < 18, and the *income* attribute has missing values.

While predicting for a test point, if we encounter a node for which the attribute used to split on has a missing value in the test point, we check to see if there is a missing value branch in the node. If so, we follow that branch. If there is no such branch, which can happen as explained above, then we use the score within that node and stop recursion for that tree and return.

Usage

The following should be noted when using missing values within Skytree Server.

- No special flags need to be provided for data that contains missing values. If the data contains missing values as appropriately signaled, the modules will handle them appropriately.
- The `--fast_read` option will work with data that contains missing values.
- The following symbols can be used to represent data with missing values
  - '?' : This is the way Skytree Server Data Preparation encodes missing values. This is the recommended method.
  - '`[+|-]inf`' : This is NOT case sensitive, so '`-INF`', '`+iNf`' will be decoded to missing values.
  - '`inf`' : This is NOT case sensitive, so '`INF`', '`iNf`' will be decoded to missing values.
  - 'nan' : This is NOT case sensitive, so 'nan', 'nAn' will be decoded to missing values.
  - '`[+|-]`nan' : This is NOT case sensitive, so '-NAN', '+nAn' will all be decoded to missing values.

A Simple Example

The following example illustrates how to run datasets with missing values.

```
# skytree-server       gbt                        \
  --training_in        income.missing.data.st     \
  --training_labels_in income.missing.data.labels \
  --testing_in         income.missing.test.st     \
  --num_trees          10                         \
  --labels_out         labels
```

Notice that there are no missing-values-specific arguments; i.e. no additional arguments are needed, and missing values are automatically detected in the input files.

Note: We recommend that you review the `income.missing.data.st` and `income.missing.test.st` files.

Variable Importances

Skytree Server provides methods for helping determine the importance of intrinsic variables and for continuous dimensions. Refer to the following sections:

- Intrinsic Variable Importances (page 81)
- Intrinsic Split Importances (page 82)
- Test Point Variable Importances (page 83)

Intrinsic Variable Importances

In tree-based methods the importance of each predictor variable in the model is not the same (i.e., certain predictor variables will play a more important role than others in the tree model in determining how accurate the model is). One way to determine relative importance amongst predictor variables is to use information used to split nodes [breiman1984cart]. In essence, each node partitions the data using a particular predictor variable and that results in some improvement in accuracy. For this predictor variable, summing up the weighted accuracy improvement across all non-leaf nodes in the tree and comparing this summed value with that for other dimensions is a way to compare importance of different predictor variables. This idea can be extended to ensembles of trees where the improvements are summed up over all trees as well [hastie2009elem], [friedman2001gbt]. The importance values are scaled so that the most important variable gets an importance of 100. This is achieved by dividing the raw importance values for all the dimensions by the maximum value and then multiplying by 100. Only dimensions which are completely unused in the model will have an importance of 0.

All of Skytree Server tree-based ensemble methods have the `--variable_importances_out` option that can be used to output the relative importance of predictor variables in the ensemble model. This output file includes a single column. The first row is the importances of the first column variable in the .st file, the second row is the importance of the second column, and so on. The following is a simple example:

```
# skytree-server              [gbt|rdf]              \
  --training_in               income.data.st         \
  --training_labels_in        income.data.labels     \
  --holdout_ratio             0.2                    \
  --testing_in                income.test.st         \
  --num_trees                 5:5:100                \
  --num_dimensions            1:1:4                  \
  --probabilities_out         probabilities.tuning   \
  --labels_out                labels.tuning          \
  --variable_importances_out  variable_importances.out
```

In the above example, first tuning of parameters occurs where a holdout training sample is used to pick the best set of parameters and then the best parameters found are applied to the whole training data and then the model is applied to the test data to obtain results. The variable importances output are for the last model built (i.e., the one that is applied to the test data).

Variable importance may be similarly computed and output for regression models.

Variable Importances Output as JSON

When outputting output the relative importance of predictor variables in the ensemble model, a `--variable_importances_out_as_json` option can be specified. This enables writing variable importances as a JSON file with feature names, attribute names, and column IDs along with the importance. All available information about the column is printed in the following format:

```
{
 "featureName": <string>,
 "attributeName": <string>,
 "originalColumnId": <integer>,
 "trainingColumnId": <integer>,
 "variableImportance": <float>
}
```

The "`featureName`" and "`originalColumnId`" values are taken from the dataset information (from the comment section of the training `.st` file). The "`attributeName`" value comes from the header line that might be present in the training `.st` file. Finally, the "`trainingColumnId`" value is the column ID in the training `.st` file.

Intrinsic Split Importances

Variable importances provide insight into relative utility of different dimensions versus each other in a given model. Taking that to another level is a feature called split importances for continuous dimensions. In this analysis we bucket each continuous dimension into **B** bins and then measure the per point improvement that splits within these buckets result in. In other words the concept is very similar to variable importances except for the following key differences:

- We attribute importances to buckets within each dimension. Buckets are simply equidistant ranges between the min and max values within a dimension. The number of buckets is configurable.
- We divide the final improvement, which is the sum of improvements from splits that occurred within that bucket, by the number of points that were in the split nodes. The reason for this is that some buckets just happen to have more points in them. For example, let's say the dataset has a dimension called age and the minimum age is 0 and the maximum age is 90 yet 80% of the dataset has age between 30 and 45. That means that the absolute value of split importances in the corresponding buckets will be very high but this does not provide any insight into whether that range provides splits that are better than any other bucket. On the other hand a bucket which has very high split importance but has only 2 points in it does not provide any useful information either. What is interesting are buckets with large support that differ in the split importance. What that might indicate is that when splits occurred in one of those buckets the average improvement per point was higher than in another bucket (for example).
- Split importances are only available for continuous dimensions.

All of Skytree Server tree based ensemble methods have the `--split_importances_out` option that can be used to output the relative split importance of predictor variable split bins in the ensemble model. The following is a simple example:

```
# skytree-server           [gbt|rdf]               \
  --training_in            income.data.st          \
  --training_labels_in     income.data.labels      \
  --holdout_ratio          0.2                     \
  --testing_in             income.test.st          \
  --num_trees              5:5:100                 \
  --num_dimensions         1:1:4                   \
  --probabilities_out      probabilities.tuning    \
  --labels_out             labels.tuning           \
  --variable_importances_out variable_importances.out \
  --split_importances_out    split_importances.out
```

In the above example, first tuning of parameters occurs where a holdout training sample is used to pick the best set of parameters and then the best parameters found are applied to the whole training data and then the model is applied to the test data to obtain results. The variable importances and split importances output are for the last model built (i.e., the one that is applied to the test data).

Test Point Variable Importances

Whereas intrinsic variable importances indicate relative importance of attribute variables for the whole model and essentially model the training data, it may be desirable to compute the variables that contributed most toward the prediction that a model makes for a particular point. This can be achieved through the `--test_point_variable_importances_out` option.

A simple example is as follows:

```
# skytree-server                        gbt               \
  --training_in                         income.data.st    \
  --training_labels_in                  income.data.labels \
  --testing_in                          income.test.st    \
  --num_trees                           100               \
  --test_point_variable_importances_out income.test.vimps \
  --model_out                           model.simple
```

This produces a file called `income.test.vimps`, which contains the test point variable importances for the test dataset, `income.test.st`. The resulting table in `income.test.vimps` is the same size as the table in the test dataset. This is because the results include a vector for each point in the dataset, and the length of each vector is the number of attributes in the dataset. Also, it is important to note that this can be computationally more expensive for the prediction phase and, hence, may take longer run. (Training time is not affected by this output, but testing time is.)

Finally, models built and stored using `--model_out` can be used to get this output for different testing inputs as follows:

```
# skytree-server                       gbt               \
  --model_in                           model.simple      \
  --test_point_variable_importances_out income.data.vimps \
  --testing_in                         income.data.st
```

Random Decision Forests

The Random Decision Forests algorithm is an ensemble classifier composed of a collection of decision trees. The ensemble returns classification based on the aggregate results of the individual trees. Each tree in the forest is a weak classifier, trained by selecting a subset of the training data, usually with replacement, and a random selection of the available features.

Each tree is grown using a sampling of the original training data. Nodes of the tree are split by considering a subset of all features of the data and are grown without pruning.

Each ensemble classification requires classification by each tree in the forest. The class with a plurality of votes determines the final classification.

Several factors affect the error of the model. Strong individual trees in the ensemble decrease the error rate and occur with a larger number of trees. However, a large number of trees also introduces correlated trees, which increases the error rate. For these reasons, among others, it’s important that the parameters of the Random Decision Forest be tuned to provide the best possible error rate.

The `rdf` module can be applied to binary classification problems.

To see all available options for the `rdf` module, run:

```
# skytree-server rdf --help
```

or refer to the *Random Decision Forests Options* (page 302) in the Command Reference appendix.

The remainder of this section discusses features specific to Random Decision Forests and explores how to maximize classification performance for your machine learning problem using these unique features.

Handling Categorical Data

The Ensemble methods can handle categorical data seamlessly. There are a few options available in RDF for the user to fine tune the way categorical variables are handled. This allows more control in the way data is split in the decision tree based on a categorical attribute.

We illustrate briefly categorical decision splits with the help of an example. Assume one of the attributes in the dataset is 'color' and it can take 4 values *blue*, *green*, *red*, *black*. Also, assume only binary splits for the moment. To split the data into 2 parts we would have to create a combination of the above attribute values for example *blue*, *green* or *blue*, *red*, *black* or simply *blue*, for which all data with those values would be partitioned left, and the rest of the data partitioned right. In all there are 2^{4-1}=8 such combinations for this example and in general 2^{q-1} such combinations where q is the number of possible unique categorical attribute values. A naive algorithm searches over all these different combinations and picks the one with the least impurity. Skytree Server has an optimized algorithm for finding the best partition in O(q) complexity vs. O(2^{q-1}) complexity for binary classification problems using the Random Decision Forest module.

Sometimes, it is preferable to try a few random combinations of attributes to split on. Skytree Server provides options for various ways to employ good categorical splitting mechanisms. The following explains these in more detail:

- `--categorical_selection_method`: The way in which categorical attribute value partitions are derived. The available options are:
  - `random`: Create randomly distributed, equally sized splits on the attribute values.
  - `exact`: Search for the best partition possible. As explained above, this can be very expensive for categorical attributes with large number of unique attribute values (except for the case of Random Decision Forest binary classification). If specified, for data with 10 or less unique attribute values the exhaustive search is performed. For larger number of values a heuristic search is performed.
  - `random_exact`: For unique attribute values less than or equal to `--categorical_random_split_threshold`, this performs exact search and for higher number of values random splitting is attempted.
  - `one_vs_all`: Splits are created with single attribute value on one side and all other values on the other side. For example: blue or not blue.
  - `one_vs_all_random`: Tries both `random` and `one_vs_all`.
  - `one_vs_all_exact`: For unique attribute values less than or equal to `--categorical_random_split_threshold`, this performs an exact search. For a higher number of values, `one_vs_all splitting` is attempted.

  Broadly speaking, in Random Decision Forests for binary classification both `random_exact` and `exact` can be deployed effectively.
- `--categorical_random_split_tries`: This is the number of times random splitting of the attribute space is attempted.
- `--categorical_random_split_threshold`: Used as explained above.

Tuning the Number of Dimensions

In addition to tuning the number of trees for example, the number of randomly selected dimensions considered in splitting each tree node may also be tuned. By default, the `rdf` module uses $\log_2(D + 1)$ dimensions, where D is the total number of attributes in the dataset. To change this behavior, specify the number of dimensions via the `--num_dimensions` option. To consider the parameter in tuning, specify a range (or a comma-separated list):

```
# skytree-server       rdf                \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --num_trees          100                \
  --num_dimensions     1:1:4
```

Similarly, you can tune the `--tree_depth` and `--smoothing` options.

Dimension Sampling

When using `--num_dimensions` with the RDFR, RDF, or Ensemble GBT modules, the way in which dimensions are sampled can be controlled in two ways.

1. **Uniform sampling**: For all dimensions, select with a uniform random distribution whether the dimension are continuous or categorical. In other words, given an example dataset with say 20 "real" and continuous dimensions and 10 categorical dimensions, the chance of any dimension being selected if `--num_dimensions` is set to 10 is 1/3. This feature is disabled by default and can be enabled by using `--cardinality_based_dimension_sampling=false`.

2. **Cardinality based dimension sampling**: This feature biases the sampling by making it more likely for categorical dimensions with higher cardinality (i.e. that is those with a larger number of unique values in the node we are in) to be picked during dimension selection for that node. Assume the same as the Uniform Sampling example, where we have 30 dimensions in our dataset, of which 10 are continuous and 20 are categorical. Assume also that 10 of the categorical dimensions had 2 values each in the given node, and the other 10 had 4. In this case, the categorical dimensions with 4 unique values would be twice as likely to be picked in dimension selection than the rest of the categorical dimensions and would be 4 times as likely as any of the continuous dimensions. This feature is enabled by default.

Sparse Data

The `rdf` module is capable of accelerated and more memory efficient execution for sparse datasets. As with multi-class classification (described earlier), no special options need to be set to enable sparse operation. Rather, if the input datasets (`--training_in`, `--tuning_in`, and `--testing_in`) are represented in the sparse file format given in *Skytree Server File Format* (page 9), then `rdf` will employ a more compact internal representation of the sparse data. Note, however, that if any of `--training_in`, `--tuning_in`, or `--testing_in` is represented as sparse, then all of them must be represented as sparse. Additionally, saved models trained from data stored in the sparse format may only be used to predict on testing sets also stored in the sparse format, and likewise for dense.

Using Less Memory

You can use the `--num_cached_trees` option to limit the amount of memory used by the `rdf` module. Lowering this option's value results in lower memory usage at the expense of increased runtime.

Gradient Boosted Trees

Gradient Boosted Trees (GBT) is a machine learning algorithm combining decision trees with boosting. The latter is a numerical gradient optimization method that minimizes a loss function, in this case, the deviance.

The optimization is achieved by successively adding trees that best reduce the loss function. The first tree reduces the loss function to the greatest extent possible, while subsequent trees focus on a residual that represents the poorest responses of the model.

As the boosting iteration proceeds, the algorithm adjusts the relative weighting of each tree without modifying the previous trees themselves. The resulting model is a linear combination of all trees.

The `gbt` module can be applied to binary classification problems. The data labels must be `-1` and `1`. To see all available options for the `gbt` module, run:

```
# skytree-server gbt --help
```

or refer to *Gradient Boosted Trees Options* (page 247) in the Command Reference appendix.

The remainder of this section discusses some of the features unique to Gradient Boosted Trees and explores how to maximize classification performance for your machine learning problem using these unique features.

Smart Search

In the case where users might not know the best configuration options to specify in order to train and tune a model, or in the case where users want to repeat an experiment, GBT allows for hyper-parameter optimization (smart search).

With a series of `--smart_search` options, users only specify the training set and model validation options (tuning dataset, holdout ratio, number of folds, etc.), and GBT will automatically search for the best parameter settings without any additional required input from the user. When scoring, smart search will tune over the `--testing_objective` while reporting additional scores.

The example below specifies to try `1000` different parameter settings.

```
# skytree-server              gbt                    \
  --training_in               ranking.train.st       \
  --training_labels_in        ranking.train.targets \
  --holdout_ratio             0.3                    \
  --num_folds                 10                     \
  --smart_search                                     \
  --smart_search_iterations 1000
```

Note: The above example performs smart search with a default testing objective of `gini`. The `--testing_objective` option guides the sequence of parameters used during smart search and should be explicitly stated if the desired objective is not the default (`gini`for binary; `accuracy` for multi-class).

By default, smart search restricts the parameter space (for example, in `--num_trees`, `--max_splits`, `--tree_depth`, etc.) in order to reduce the training time. If you do not want to restrict the upper bound for these parameters, specify `--limit_parameters=off`.

In some cases, users may want to input values for certain parameters and let GBT smart search tune over the rest. Users can provide open or closed intervals for any tuning parameter using <min>:<max> or <min>: or :<max>. Specific values at regular intervals can also be specified with <min>:<step>:<max>. For example:

- `--num_trees=2:10` can be used to specify an interval.
- `--num_trees=100:` specifies a lower bound.
- `--num_trees=10:10:100` & `--num_trees=54` can be used to specify exact values to use during smart search.

Tunable intervals or exact values that are explicitly set will be passed to `skytree-server`. All other parameters will be sampled during smart search.

Note: `--smart_search` cannot be used with `--model_in`. Similarly, `--probability_threshold` cannot be specified with `--smart_search` because this option cannot be used during tuning.

Restarting Smart Search

There may be instances where you want to restart a smart search run. For example, if you initially specify to run 100 iterations, but after the program completes you want to run an additional 100 iterations. The `--smart_search_restart_out` option allows you to specify a file name that saves the original smart search data. You can specify this file using the `--smart_search_restart_in` option to continue iterations from the original run.

The example below creates a smart search output file named "ranking.50.restart".

```
# skytree-server              gbt                    \
  --training_in               ranking.train.st       \
  --training_labels_in        ranking.train.targets  \
  --holdout_ratio             0.3                    \
  --num_folds                 10                     \
  --smart_search                                     \
  --smart_search_iterations   50                     \
  --smart_search_restart_out  ranking.50.restart
```

The example above runs smart search with 50 iterations. The following example similarly specifies to run 100 smart search iterations. In addition, the "ranking.50.restart" file is specified as an input file. Because this file already runs 50 iterations, this example will run a total of 150 smart search iterations.

Note: When using `--smart_search_restart_in`, the specified input data files must be the same as the data specified in the original `--smart_search_restart_out` run.

```
# skytree-server              gbt                    \
  --training_in               ranking.train.st       \
  --training_labels_in        ranking.train.targets  \
  --smart_search_restart_in   ranking.50.restart     \
  --smart_search                                     \
  --smart_search_iterations   100
```

Notice that only the smart search options and the input data files from the original run are required. Users can specify additional options only if those options were specified in the original smart search run (for example, `--num_folds`). In this case, the values for the options must match the original run.

Fast Tree Construction

By default, trees of a GBT model are built using a fast heuristic method instead of exact split criterion. This feature is enabled or disabled using the `--regularization` option. The regularization feature bins each dimension and then splits tree nodes based on the resulting data. The number of bins used is controlled via the `--regularization_bins` option; its default value is 200.

The fast method consumes less memory and allows each GBT to be built by multiple distributed processes. See *Running Distributed Jobs* (page 173) for further information regarding running the fast method in distributed jobs.

Currently, the `--trim` option is not supported with the `--regularization` option.

Note: The `--regularization` option defaults to OFF for ensemble GBT/GBTR.

Ranking

Gradient Boosted Trees have been shown to be very successful at solving ranking problems in the information retrieval domain, such as Web search results and recommendation engines. The GBT module can be used to create models that can make use of labeled data in this domain to return accurate ranking of information given by a particular query.

The prerequisites for using this form of GBT is that the training data must be divided into queries or groups. A query or group is a set of documents or other information that was retrieved in the past, and some of these (represented as rows in the dataset) were relevant while others were not. This can be considered in a binary setting, where documents are marked 1 for relevant and 0 for not relevant or in a ranking setting where irrelevant documents are marked 0 and relevant documents can have varying values of relevance depending on how relevant they were. For example in Web searches, "clicks" can be represented as having a relevance of 1, "purchases" can have a relevance of 5, and no activity must be set to have a relevance of 0.

The GBT module then can take training data that contains groups like these with each row labeled as explained above to build a predictive model, then apply the results to a list of documents or information items, and rank them according to the query. Finally, the results can be scored. Typically when scoring various information retrieval, metrics can be used. The popular metrics (loss functions) for binary problems are Mean Reciprocal Rank (MRR) and Mean Absolute Precision (MAP). For more general problems, Normalized Discounted Cumulative Gain (NDCG) is commonly used.

Input format

The input format for the data is the same as that for GBT multi-class or binary classification with one difference: the group IDs must be included in the third meta column of the ".st" file created by `convert-data.sh`. (Refer to *Convert Data* (page 18).) This is easily achieved through the Data Preparation utilities by specifying the `-id_index` argument and pointing that at the column that contains the group IDs in your raw data. The `convert-data.sh` process will then automatically place the group IDs in the third column of the generated .st file. Refer to *Identifying ID Columns* (page 20) for more information.

A Simple Example

A simple example to build a model, save it, and simultaneously apply the model to a test dataset is as follows.

```
# skytree-server       gbt                    \
  --training_in        ranking.train.st      \
  --training_labels_in ranking.train.labels  \
  --num_trees          100                   \
  --loss_function      ndcg                  \
  --model_out          model                 \
  --testing_in         ranking.test.st       \
  --probabilities_out  probabilities
```

The file "probabilities" will now contain the scores for all the input test points in "ranking.train.st". Note that group information is not needed at this time. Group information is useful only when scoring is performed. An example of scoring is as follows:

```
# skytree-server    score-recommendation  \
  --true_labels_in  ranking.test.targets  \
  --group_ids_in    ranking.test.groupids \
  --scores_in       probabilities
```

Tuning Parameters

All tuning options available in the GBT module, such as cross-validation and holdout set tuning, are available when ranking. The difference is that, while GBT for classification splits the training data by row, in ranking problems whole groups must be split into training and tuning datasets. For example, if the training dataset contains 10,000 groups and 1 million training points overall, then 5-fold cross validation will create training datasets with 8,000 groups and tuning datasets with 2,000 groups. The number of rows in each dataset will depend on the group assignments and should obey similar ratios for large datasets.

An example of tuning is as follows:

```
# skytree-server       gbt                   \
  --training_in        ranking.train.st      \
  --training_labels_in ranking.train.targets \
  --num_trees          100                   \
  --holdout_ratio      0.3                   \
  --learning_rate      0.1:0.05:0.3          \
  --tree_depth         1,2,3                 \
  --loss_function      ndcg
```

Tunable Parameters

Gradient Boosted Trees have the following tunable parameters (in addition to the options shown in the previous section):

- `--tree_depth`: Depth to which each tree in the ensemble is built
- `--max_splits`: Number of leaf splits in each tree
- `--learning_rate`: Regularization parameter
- `--regularization_bins`: Number of bins used with `--regularization`

The following tunable parameters: `--ensemble_size`, `--num_dimensions`, `--sampling_ratio`, and `--imbalance_scale` are part of the *Ensemble Gradient Boosted Trees* (page 92) method.

Tuning the Tree Depth and Learning Rate

To tune the tree depth and the learning rate, specify either a comma-separated list of values, or a list of the form `<minimum value>:<step size>:<maximum value>` for the `--tree_depth` and/or `--learning_rate` options.

If `--tree_depth` is not set, and values for `--max_splits` and `--min_node_weight` are likewise not set, then this value defaults to 3.

If `--tree_depth` is explicitly set to 0, trees will be built to the fullest extent. In this case, you must specify a positive value for either `--max_splits` or `--min_node_weight`.

The following command tunes for three parameters:

```
# skytree-server       gbt                 \
  --training_in        income.data.st      \
  --training_labels_in income.data.labels  \
  --holdout_ratio      0.2                 \
  --num_trees          5:5:25              \
  --tree_depth         2,3                 \
  --learning_rate      0.05:0.05:0.2
```

Tuning the Maximum Splits

The `--max_splits` option allows you to specify the number of splits in the tree. This option uses split priority to build the trees rather than building depth first. A value > 0 must be specified for this option if `--tree_depth` is set to 0 and `--min_node_weight` is not specified.

```
# skytree-server       gbt                 \
  --training_in        income.data.st      \
  --training_labels_in income.data.labels  \
  --num_folds          5                   \
  --num_trees          25,50               \
  --max_splits         4
```

Offsets

In GBT and GBTR, the first tree typically acts as an offset. Skytree Server allows you change this behavior by specifying an offset file to use when training, tuning, and testing.

An offset file is any single-column file, similar to targets, probabilities, and labels. The number of rows in the offset file must match the number of points in the training/tuning/testing file. An offset file can be created using `--scores_out`.

Note: Offsets can only be used with binary classification and logistic scoring.

```
# skytree-server       gbt                  \
  --training_in        income.tune.st       \
  --training_labels_in income.tune.labels   \
  --testing_in         income.test.st       \
  --scores_out         income.test.offsets  \
  --num_trees          10
```

The offset file can the be referenced when tuning, training, and testing. Note that the example below assumes that an `income.train.offsets` file already exists.

```
# skytree-server         gbt                    \
  --training_in          income.train.st        \
  --training_labels_in   income.train.labels    \
  --training_offsets_in  income.train.offsets   \
  --num_trees            10                     \
  --loss_function        logistic               \
  --testing_in           income.test.st         \
  --testing_offsets_in   income.test.offsets    \
  --probabilities_out    probabilities          \
  --labels_out           labels                 \
  --scores_out           scores
```

Important Notes About Using Offsets

- In order to use offsets while tuning with `--tuning_in`, both `--training_offsets_in` and `--tuning_offsets_in` must be specified.
- In order to use offsets during training/tuning and testing when training was done with offsets, then:
  - `--labels_out`, `--probabilites_out`, and `--targets_out` will only work if the `--testing_offsets_in` option is included;
  - otherwise, if `--testing_offsets_in` is not included, then `--scores_out` can be specified to get an output.
- When testing with `--model_in`, the `--testing_offsets_in` option can be provided, but you will receive a warning if:
  - the input model was trained with offsets, but the `--testing_offsets_in` option was not provided,
  - or the input model was trained without offsets, but the `--testing_offsets_in` option was provided.

  In both of these above cases, Skytree Server will continue to run.

Model Visualization

The decision trees of a GBT model can be stored as files containing their JSON representations by specifying the `--visualization_out` option along with a suitable directory.

The trees can be viewed in a Web browser after running the supplied `mktreeviz.sh` script. That script should be run with the same directory argument that was given for the `--visualization_out` option.

The `index.html` file created in the directory provides links to individual tree HTML files.

Note: Column labels from your original CSV data files, if they exist, will appear in the trees if you have used the data preparation tools from the current Skytree Server release. You may want to rerun those tools if you have data files created by earlier versions of the data preparation tools.

Browser Security

Skytree Server visualizations are stored and viewed from your local file system. However, modern browsers limit or restrict access to local files as a security precaution. If you are comfortable with circumventing those policies while viewing the visualizations, please follow the appropriate instructions for your browser, or if you have Python installed, a local server can be used as described in the next section.

Running a Local Python Server

If you have Python installed, the simplest and probably most secure option, is to run a local Python server. To do so, open a terminal and navigate to the directory with the visualization files. Then, for Python 2.x enter:

```
# python -m SimpleHTTPServer 9001
```

For Python 3.x enter:

```
# python -m http.server 9001
```

You will then be able to access the site through your browser, using:

```
http://localhost:9001
```

Note that you can substitute any port number in place of 9001.

Google Chrome

To allow Chrome access to your local files you will need to launch it from the command line as follows:

```
# chrome --allow-file-access-from-files
```

If you launch Chrome through a GUI shortcut, you can also edit that shortcut to include the additional flag: right-click on **Shortcut > Properties > Target**.

Mozilla Firefox

To enable access to local files in Firefox:

Enter `about:config` in the address bar. Search for `security.fileuri.strict_origin_policy` and set its value to `false`.

Apple Safari

To access files locally in Safari, you must turn on the **Develop** menu under **Preferences > Advanced > Show develop menu in menu bar**.

You will now have an additional Develop menu in your toolbar. From this menu, toggle **Disable local file restrictions**. It is also advisable to select **Disable caches** in order to prevent old data from persisting.

Ensemble Gradient Boosted Trees

The ensemble Gradient Boosted Tree (GBT) algorithm applies the concept of bootstrap aggregation ("bagging") to GBTs. The training table is sampled a number of times, and a new GBT is built for each sample.

The method is invoked using the same `gbt` module presented in the previous section but requires that `--ensemble_size` (the number of GBTs to build) be specified:

```
# skytree-server       gbt                \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --ensemble_size      1:2:9              \
  --num_dimensions     2:1:4              \
  --min_node_weight    4                  \
  --num_trees          25
```

Note that the `--num_trees` option must also be given, and each GBT member of the ensemble will build that specified number of trees. The `--min_node_weight` option specifies the minimum bound on the total weight in each leaf during tree building.

Of course, the time required for building such a model may increase significantly in comparison to a single GBT model, depending on the number of ensemble members specified.

Tunable Parameters

All GBT parameters presented in the previous section can also be tuned for ensemble GBTs. In addition, you can also tune the `--ensemble_size` and the number of randomly selected dimensions (`--num_dimensions`) used for each tree.

```
# skytree-server       gbt                \
  --training_in        income.data.st     \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                \
  --ensemble_size      1:2:9              \
  --num_dimensions     2:1:4              \
  --num_trees          25
```

Note: Tuning differs slightly between ensemble and regular GBTs. For a single GBT, tuning the number of trees is relatively inexpensive as scoring is done at intermediate stages while the algorithm builds towards the maximum number of trees. This obviates the need to rebuild a full model for each `--num_trees` setting.

For GBT ensembles, the intermediate scoring is for ensemble members instead, i.e., for each full GBT. Tuning for `--ensemble_size` is therefore inexpensive (and done automatically) while tuning for the number of trees requires additional models to be fully built.

The tunable `--sampling_ratio` and `--imbalance_scale` options allow numerous smaller GBTs to be built on down-sampled data. This may provide an effective model for skewed datasets. Typically we find that using the `--imbalance` option along with `--ensemble_size` can give a boost in accuracy for highly skewed problems if the right parameters are used for base GBTs.

Dimension Sampling

When using `--num_dimensions` with the RDFR, RDF, or Ensemble GBT modules, the way in which dimensions are sampled can be controlled in two ways.

1. **Uniform sampling**: For all dimensions, select with a uniform random distribution whether the dimension are continuous or categorical. In other words, given an example dataset with say 20 "real" and continuous dimensions and 10 categorical dimensions, the chance of any dimension being selected if `--num_dimensions` is set to 10 is 1/3. This feature is disabled by default and can be enabled by using `--cardinality_based_dimension_sampling=false`.

2. **Cardinality based dimension sampling**: This feature biases the sampling by making it more likely for categorical dimensions with higher cardinality (i.e. that is those with a larger number of unique values in the node we are in) to be picked during dimension selection for that node. Assume the same as the Uniform Sampling example, where we have 30 dimensions in our dataset, of which 10 are continuous and 20 are categorical. Assume also that 10 of the categorical dimensions had 2 values each in the given node, and the other 10 had 4. In this case, the categorical dimensions with 4 unique values would be twice as likely to be picked in dimension selection than the rest of the categorical dimensions and would be 4 times as likely as any of the continuous dimensions. This feature is enabled by default.

Random Decision Forest Regression

The Random Decision Forest Regression algorithm is an ensemble regression model comprising a collection of regression trees. The ensemble returns predictions based on the average of the results of the individual trees. Each tree in the forest is a weak regression model, trained by selecting a subset of the training data, usually with replacement, and a random selection of the available features.

Each tree is grown using a sampling of the original training data. Nodes of the tree are split by considering a subset of all features of the data and are grown without pruning.

Each ensemble regression prediction requires a regression prediction by each tree in the forest. The average of the predictions of the individual trees determines the final prediction.

Several factors affect the error of the model. Strong individual trees in the ensemble decrease the error rate and occur with a larger number of trees. However, a large number of trees also introduces correlated trees which increase the error rate. For these reasons, among others, it's important that the parameters of the Random Decision Forest Regression model be tuned to provide the best possible error rate.

The `rdfr` module can be applied to regression problems.

To see all available options for the `rdfr` module, run:

```
# skytree-server rdfr --help
```

or refer to the *Random Decision Forests Regression Options* (page 312) in the Command Reference appendix.

The remainder of this section discusses features specific to Random Decision Forest Regression and explores how to maximize prediction performance for your machine learning problem using these unique features.

Tunable Parameters

Random Decision Forest Regression has three tunable parameters in addition to the options shown in the previous section:

- `--tree_depth`: This is the depth to which each tree in the ensemble is built. This is not a parameter that is typically tuned in Random Decision Forest Regression but can be. If left to 0, the trees are built to full extent.
- `--num_dimensions`: The number of dimensions that should be sampled at any given tree node for finding a node partition.
- `--sampling_ratio`: This is the ratio to which the data is down-sampled or up-sampled for each decision tree construction. Because Random Decision Forests require sampling, if `--sample_with_replacement=off`, the `--sampling_ratio` option is mandatory and should be set to a real number between 0 and 1. By default, the

sampling ratio is set to 1 and the sampling is done with replacement (that is, `--sample_with_replacement=on`).

- `--min_node_weight`: The minimum bound on the total weight in each leaf during tree building

Tuning the Number of Dimensions

In addition to tuning the number of trees for example, the number of randomly selected dimensions considered in splitting each tree node may also be tuned. By default, the `rdfr` module uses (*D/3*) dimensions, where *D* is the total number of attributes in the dataset. To change this behavior, use `--num_dimensions` to specify the number of dimensions. To consider the parameter in tuning, specify a range:

```
# skytree-server        rdfr                 \
  --training_in         income.data.st       \
  --training_targets_in income.data.targets  \
  --holdout_ratio       0.2                  \
  --num_trees           100                  \
  --num_dimensions      1:1:4
```

Dimension Sampling

When using `--num_dimensions` with the RDFR, RDF, or Ensemble GBT modules, the way in which dimensions are sampled can be controlled in two ways.

1. **Uniform sampling**: For all dimensions, select with a uniform random distribution whether the dimension are continuous or categorical. In other words, given an example dataset with say 20 "real" and continuous dimensions and 10 categorical dimensions, the chance of any dimension being selected if `--num_dimensions` is set to 10 is 1/3. This feature is disabled by default and can be enabled by using `--cardinality_based_dimension_sampling=false`.

2. **Cardinality based dimension sampling**: This feature biases the sampling by making it more likely for categorical dimensions with higher cardinality (i.e. that is those with a larger number of unique values in the node we are in) to be picked during dimension selection for that node. Assume the same as the Uniform Sampling example, where we have 30 dimensions in our dataset, of which 10 are continuous and 20 are categorical. Assume also that 10 of the categorical dimensions had 2 values each in the given node, and the other 10 had 4. In this case, the categorical dimensions with 4 unique values would be twice as likely to be picked in dimension selection than the rest of the categorical dimensions and would be 4 times as likely as any of the continuous dimensions. This feature is enabled by default.

Tuning the Sampling Ratio

As with all other tuning parameters, the sampling ratios can be specified with a comma-separated list of values or a list of the form `<minimum value>:<step size>:<maximum value>`. For example, the sampling ratio can be tuned over the values 0.2, 0.3, 0.4, 0.5 with

```
# skytree-server        rdfr                 \
  --training_in         income.data.st       \
  --training_targets_in income.data.targets  \
  --holdout_ratio       0.2                  \
  --num_trees           100                  \
  --sampling_ratio      0.2,0.3,0.4,0.5
```

or with

```
# skytree-server        rdfr                 \
  --training_in         income.data.st       \
  --training_targets_in income.data.targets  \
  --holdout_ratio       0.2                  \
  --num_trees           100                  \
  --sampling_ratio      0.2:0.1:0.5
```

Sparse Data

The `rdfr` module is capable of accelerated and more memory efficient execution for sparse datasets. No special options need to be set to enable sparse operation. Rather, if the input datasets (`--training_in`, `--tuning_in`, and `--testing_in`) are represented in the sparse file format given in *Skytree Server File Format* (page 9), then `rdfr` will employ a more compact internal representation of the sparse data. Note, however, that if any of `--training_in`, `--tuning_in`, or `--testing_in` is represented as sparse, then all of them must be represented as sparse. Additionally, saved models trained from data stored in the sparse format may only be used to predict on testing sets also stored in the sparse format, and likewise for dense.

Using Less Memory

You can use the `--num_cached_trees` option to limit the amount of memory used by the `rdfr` module. Lowering this option's value results in lower memory usage at the expense of increased runtime.

Gradient Boosted Trees Regression

Gradient Boosted Trees Regression is a machine learning algorithm combining regression trees with boosting. The latter is a numerical gradient optimization method that minimizes a loss function, in this case, the deviance.

The optimization is achieved by successively adding trees that best reduce the loss function. The first tree reduces the loss function to the greatest extent possible, while subsequent trees focus on a residual that represents the poorest responses of the model.

As the boosting iteration proceeds, the algorithm adjusts the relative weighting of each tree without modifying the previous trees themselves. The resulting model is a linear combination of all trees.

The `gbtr` module can be applied to regression problems.

To see all available options for the `gbtr` module, run:

```
# skytree-server gbtr --help
```

or refer to *Gradient Boosted Trees Regression Options* (page 259) in the Command Reference appendix.

The remainder of this section discusses some of the features unique to Gradient Boosted Trees Regression and explores how to maximize regression performance for your machine learning problem using these unique features.

Smart Search

In the case where users might not know the best configuration options to specify in order to train and tune a model, or in the case where users want to repeat an experiment, GBTR allows for hyper-parameter optimization (smart search).

With a series of `--smart_search` options, users need only specify the training set and model validation options (tuning dataset, holdout ratio, number of folds, etc.), and GBTR will automatically search for the best parameter settings without any additional required input from the user. When scoring, smart search will tune over the `--testing_objective` while reporting additional scores.

The example below specifies to try 1000 different parameter settings.

```
# skytree-server             gbtr                  \
  --training_in              income.train.st       \
  --training_targets_in      income.train.targets  \
  --holdout_ratio            0.3                   \
  --num_folds                10                    \
  --smart_search                                   \
  --smart_search_iterations 1000
```

Note: The above example performs smart search with a default testing objective of `mean_absolute_error`. The `--testing_objective` option guides the sequence of parameters used during smart search and should be explicitly stated if the desired objective is not the default.

By default, smart search restricts the parameter space (for example, in `--num_trees`, `--max_splits`, `--tree_depth`, etc.) in order to reduce the training time. If you do not want to restrict the upper bound for these parameters, specify `--limit_parameters=off`.

In some cases, users may want to input values for certain parameters and let GBTR smart search tune over the rest. Users can provide open or closed intervals for any tuning parameter using <min>:<max> or <min>: or :<max>. Specific values at regular intervals can also be specified with <min>:<step>:<max>. For example:

- `--num_trees=2:10` can be used to specify an interval.
- `--num_trees=100:` specifies a lower bound.
- `--num_trees=10:10:100` & `--num_trees=54` can be used to specify exact values to use during smart search.

Tunable intervals or exact values that are explicitly set will be passed to `skytree-server`. All other parameters will be sampled during smart search.

Note: When performing a GBTR smart search run without specifying a `--loss_function`, Skytree Server will automatically tune over all loss function types except `gminv`. You can only tune over `gminv` by explicitly specifying `--loss_function=gminv` in the command. Refer to *Loss Function* (page 101) for more information about the available `--loss_function` options.

Restarting Smart Search

There may be instances where you want to restart a smart search run. For example, if you initially specify to run 100 iterations, but after the program completes you want to run an additional 100 iterations. The `--smart_search_restart_out` option allows you to specify a file name that saves the original smart search data. You can specify this file using the `--smart_search_restart_in` option to continue iterations from the original run.

The example below creates a smart search output file named "`income.50.restart`".

```
# skytree-server              gbtr                  \
  --training_in               income.data.st        \
  --training_targets_in      income.train.targets  \
  --holdout_ratio            0.3                   \
  --num_folds                10                    \
  --smart_search                                   \
  --smart_search_iterations  50                    \
  --smart_search_restart_out income.50.restart
```

The example above runs smart search with 50 iterations. The following example similarly specifies to run 100 smart search iterations. In addition, the "`income.50.restart`" file is specified as an input file. Because this file already runs 50 iterations, this example will run a total of 150 smart search iterations.

Note: When using `--smart_search_restart_in`, the specified input data files must be the same as the data specified in the original `--smart_search_restart_out` run.

```
# skytree-server             gbtr                 \
  --training_in              income.train.st      \
  --training_targets_in      income.train.targets \
  --smart_search_restart_in  income.50.restart    \
  --smart_search                                  \
  --smart_search_iterations  100
```

Notice that only the smart search options and the input data files from the original run are required. Users can specify additional options only if those options were specified in the original smart search run (for example, `--num_folds`). In this case, the values for the options must match the original run.

Fast Tree Construction

By default, trees of a GBTR model are built using a fast heuristic method instead of exact split criterion. This feature is enabled or disabled using the `--regularization` option. The regularization feature bins each dimension and then splits tree nodes based on the resulting data. The number of bins used is controlled via the `--regularization_bins` option; its default value is 200.

The fast method consumes less memory and allows each GBTR to be built by multiple distributed processes. See *Running Distributed Jobs* (page 173) for further information regarding running the fast method in distributed jobs.

Tunable Parameters

Gradient Boosted Trees Regression has four more tunable parameters (in addition to the options described previously):

- `--tree_depth`: This is the depth to which each tree in the ensemble is built. The default tree depth is set to 3.
- `--max_splits`: Number of leaf splits in each tree
- `--learning_rate`: A regularization parameter
- `--sampling_ratio`: This is the ratio to which the data is down-sampled or up-sampled for the construction of each decision tree in the ensemble. Gradient Boosted Regression does not involve any sampling by default. Setting `--sampling_ratio` invokes Stochastic Gradient Boosted Regression. If `--sampling_ratio` is set along with `--sample_with_replacement=off`, the `--sampling_ratio` should be set to a real number between 0 and 1. By default, the sampling is done with replacement (that is, `--sample_with_replacement=on`).

- `--min_node_weight`: The minimum bound on the total weight in each leaf during tree building
- `--regularization_bins`: Number of bins used with `--regularization`

Similar to the case of classification, the Ensemble Learning methods allow the user to "bag" (bootstrap aggregate) multiple Gradient Boosted Regressors. The following are the tunable parameters specific to Ensemble Gradient Boosted Regressors. (See *Ensemble Gradient Boosted Trees* (page 92) for further details on "bagging" GBTs.)

- `--ensemble_size`: The number of gradient boosted regressors to aggregate.
- `--num_dimensions`: The number of dimensions that should be sampled at any given node for finding a node partition. By default, all the dimensions are chosen to find an appropriate partition. This option can only be used in conjunction with the `--ensemble_size` option.
- `--sampling_ratio`: The behavior of this option changes if used with `--ensemble_size`. In this case, the sampling ratio corresponds to the size of the random sample of the data on which a complete Gradient Boosted Regressor is learned. If `--ensemble_size` is set, the default `--sampling_ratio` is set to 1 and the sampling is done with replacement.

Tuning the Tree Depth and Learning Rate

To tune the tree depth and learning rate, specify either a comma-separated list of values, or a list of the form `<minimum value>:<step size>:<maximum value>` for the `--tree_depth` and/or `--learning_rate` options.

If `--tree_depth` value is not set, and values for `--max_splits` and `--min_node_weight` are likewise not set, then this value defaults to 3.

If `--tree_depth` is explicitly set to 0, trees will be built to the fullest extent. In this case, you must specify a positive value for either `--max_splits` or `--min_node_weight`.

```
# skytree-server        gbtr                 \
  --training_in         income.data.st       \
  --training_targets_in income.data.targets  \
  --holdout_ratio       0.2                  \
  --num_trees           5:5:25               \
  --tree_depth          2,3                  \
  --learning_rate       0.05:0.05:0.2
```

Tuning the Maximum Splits

The `--max_splits` option allows you to specify the number of splits in the tree. This option uses split priority to build the trees rather than building depth first. A value > 0 must be specified for this option if `--tree_depth` is set to 0 and `--min_node_weight` is not specified.

```
# skytree-server       gbtr                \
  --training_in        income.data.st      \
  --training_labels_in income.data.labels  \
  --num_folds          5                   \
  --num_trees          25,50               \
  --max_splits         4                   \
```

Tuning the Sampling Ratio

As with all other tuning parameters, the sampling ratios can be specified with a comma-separated list of values or a list of the form `<minimum value>:<step size>:<maximum value>`. For example, the sampling ratio can be tuned over the values 0.2, 0.3, 0.4, 0.5 with

```
# skytree-server         gbtr                    \
  --training_in          income.data.st          \
  --training_targets_in  income.data.targets     \
  --num_folds            5                       \
  --num_trees            10                      \
  --sampling_ratio       0.2,0.3,0.4,0.5
```

or with

```
# skytree-server         gbtr                    \
  --training_in          income.data.st          \
  --training_targets_in  income.data.targets     \
  --num_folds            5                       \
  --num_trees            10                      \
  --sampling_ratio       0.2:0.1:0.5
```

Offsets

In GBT and GBTR, the first tree typically acts as an offset. Skytree Server allows you change this behavior by specifying an offset file to use when training, tuning, and testing.

An offset file is any single-column file, similar to targets, probabilities, and labels. The number of rows in the offset file must match the number of points in the training/tuning/testing file. An offset file can be created using `--scores_out`.

Note: Offsets can only be used with binary classification and logistic scoring.

```
# skytree-server        gbtr                 \
  --training_in         income.tune.st       \
  --training_labels_in  income.tune.labels   \
  --testing_in          income.test.st       \
  --scores_out          income.test.offsets \
  --num_trees           10
```

The offset file can the be referenced when tuning and training. Note that the example below assumes that an `income.train.offsets` file already exists.

```
# skytree-server         gbtr                   \
  --training_in          income.train.st        \
  --training_labels_in   income.train.labels    \
  --training_offsets_in  income.train.offsets \
  --num_trees            10                     \
  --loss_function        logistic               \
  --testing_in           income.test.st         \
  --testing_offsets_in   income.test.offsets    \
  --probabilities_out    probabilities          \
  --labels_out           labels                 \
  --scores_out           scores
```

Important Notes About Using Offsets

- In order to use offsets while tuning with `--tuning_in`, both `--training_offsets_in` and `--tuning_offsets_in` must be specified.
- In order to use offsets during training/tuning and testing when training was done with offsets, then:
  - `--labels_out`, `--probabilites_out`, and `--targets_out` will only work if the `--testing_offsets_in` option is included;
  - otherwise, if `--testing_offsets_in` is not included, then `--scores_out` can be specified to get an output.
- When testing with `--model_in`, the `--testing_offsets_in` option can be provided, but you will receive a warning if:
  - the input model was trained with offsets, but the `--testing_offsets_in` option was not provided,
  - or the input model was trained without offsets, but the `--testing_offsets_in` option was provided.

  In both of these above cases, Skytree Server will continue to run.

Loss Function

In GBTR, the default loss function method performs least absolute deviation (`lad`). Certain cases can exist, however, in which the median starting value for this loss function can lead to poor results (for example, if the median is the lowest or highest value in the tree node). The `--loss_fuction` option allows you to specify a different method to use when performing regression. Available methods include least-squares (`ls`), Huber (`huber`), Poisson-log (`pslog`), Gamma-log (`gmlog`), and Gamma-inverse (`gminv`), or Tweedie distribution (`tdlog`).

When `tdlog` is specified, users must also specify a `--tweedie_exponent` value. Users can tune over this option with values > 1.0 and < 2.0.

When `huber` is specified, you can optionally set the top percentile of error that should be considered as outliers. This value must be between 0 and 1 and defaults to `0.9`.

When `lad` or `huber` is specified, then the `--approximate_quantiles` flag becomes enabled. This allows for an error tolerance of .001 for quantiles on loss functions. For example, if you specify a `--huber_loss_quantile` of .33, then quantiles ranging from .329 to .331 will also be included.

When `pslog`, `gmlog`, or `tdlog` is specified, users can optionally define a `--log_clamp` value that is > 0. The `--log_clamp` value defaults to 20. When used for tuning, the output file will have extra columns for these options.

The example below shows how to specify the least-squares `--loss_function`.

```
# skytree-server        gbtr                  \
  --training_in         income.data.st        \
  --training_targets_in income.data.targets   \
  --holdout_ratio       0.2                   \
  --num_trees           5:5:25                \
  --loss_function       ls                    \
  --model_out           model.simple          \
  --targets_out         targets.gbtr
```

Model Visualization

Similar to GBT, GBTR models can also be visualized. Refer to *Model Visualization* (page 91) for more information.

PMML Model Export

Trained ensemble models can be exported into PMML (Predictive Model Markup Language) format using the `--pmml_out` option. This allows the use of an external code for scoring.

Note: The PMML file that is produced requires that missing values are encoded as '?'. This is a reserved symbol for missing values, and as such, should not be used anywhere else in your data.

For more information on PMML format, please refer to the PMML 4.2 section on the *Data Mining Group Web site* (http://www.dmg.org/v4-2/GeneralStructure.html).

Refer to the sections that follow for additional information and usage restrictions.

Random Decision Forests - Classification and Regression

To export in PMML format, use the following option with RDF or RDFR:

```
--pmml_out=<file>
```

The resulting model uses confidence rather than probability to express each leaf's `ScoreDistribution` elements. Confidence is different from probability in that the confidences of all classes do not necessarily sum to one, and in fact nodes can have confidence values that are greater than one.

Given a query point, prediction is performed by, for each class, summing the confidences at the leaf nodes encountered by the query point in all the trees. Then, the posterior probability of a given class is found by dividing that class's total confidence by the sum of all classes' confidences. Alternately, the class with the largest overall confidence may be taken as the query point's predicted class.

All options available for RDF and RDFR are available for PMML output.

For RDF models, the labels column is included in the PMML output. This is configured using the `--probability_threshold` parameter. If this is not specified when tuning a model, then it is set to the optimal probability threshold chosen by the model. If this is not specified, and tuning is not performed, then this value is set to 0.5.

Note: The size of the resulting PMML file is dependent on the number and depth of the trees. It may be very large.

Gradient Boosted Trees - Classification and Regression

As with RDF, to export in PMML format, use the following option with GBT and GBTR:

```
--pmml_out=<file>
```

In order to compute scores for the model, scores of all the trees in the model have to be summed. A function is then applied to the resulting sum. The function is provided under the node

```
<PostProcessor>
    <![CDATA[(1/(1 + e^(-2*s)))]]>
</PostProcessor>
```

in the PMML file itself. The symbol `s` in the function refers to the sum of the scores of all the trees.

All options available for GBT are available for PMML output except for ranking functions (i.e., if `--loss_function` is specified as other than `logistic` (default)). In addition, the labels column is included in the PMML output. This is configured using the `--probability_threshold` parameter. If this is not specified when tuning a model, then it is set to the optimal probability threshold chosen by the model. If this is not specified, and tuning is not performed, then this value is set to 0.5.

All options available for GBTR are available for PMML output.

Ensemble Gradient Boosted Trees

The PMML option `--pmml_out=<file>` can also be used with Ensemble GBT, i.e., in combination with the `--ensemble_size=`*n* option.

Because Ensemble GBT applies the concept of "bagging", the resulting models have sets of GBTs. Unfortunately, the standard PMML format does not currently support the concept of multiple models with a depth of hierarchy greater than one. In order to overcome this limitation we introduce custom PMML to correctly represent that model.

In the PMML export of Ensemble GBT, we embed a complete GBT within each `<Segment>` by introducing another level of `<Segmentation>`. Each GBT within a `<Segment>` block has its own `<PostProcessor>`, which is used to compute result of that ensemble. The results of all GBTs can then be averaged to get the final answer.

AutoModel Module

Some users want control over the parameters that are included when training and testing models for predictions. Other users may be uncertain about which parameters to include. For these users, the `automodel` module provides the convenience of performing an all-method smart search with minimal required parameter input. Users need only specify the training set and model validation options (tuning dataset, holdout ratio, number of folds, etc.), and `automodel` will automatically search for the best parameter setting without any additional required input from the user. In addition, `automodel` can also determine whether to solve a classification or regression problem based on whether the user specifies labels or targets.

Users still have the ability to specify additional parameters when running `automodel`. By default, `automodel` restricts the parameter space (for example, in `--num_trees`, `--max_splits`, `--tree_depth`, etc.) in order to reduce the training time. If you do not want to restrict the upper bound for these parameters, specify `--limit_parameters=off`.

To see all available options for the `automodel` module, run:

```
# skytree-server automodel --help
```

or refer to *AutoModel Options* (page 208) in the Command Reference appendix.

Note: Because `automodel` always performs an all-method smart search, the `--smart_search` option is always ON. Attempting to turn this OFF will have no effect.

Classification with AutoModel

The automodel module can be used for generating models (via --model_out) or for testing (via --testing_in). The following classification example shows how to use automodel to train a model over 10 folds. When training models, the --model_out option is required.

```
# skytree-server       automodel            \
  --training_in        ranking.train.st     \
  --training_labels_in ranking.train.targets \
  --num_folds          10                   \
  --model_out          ranking.model
```

Regression with AutoModel

The following example shows how to build a simple model and save it to a file. The automodel module automatically recognizes this as a regression problem when target files are specified.

```
# skytree-server       automodel           \
  --training_in          income.data.st      \
  --training_targets_in  income.data.targets \
  --testing_in           income.test.st      \
  --num_trees            100                 \
  --model_out            model.simple        \
  --targets_out          targets.predict
```

Training and Testing a Model

The following example shows how to train and test a model using automodel. In this case, --model_out is optional. The tuning option used for this example is --holdout_ratio.

```
# skytree-server       automodel            \
  --training_in        ranking.train.st     \
  --training_labels_in ranking.train.targets \
  --testing_in         ranking.test.st      \
  --holdout_ratio      0.3                  \
  --num_trees          100                  \
  --model_out          ranking.model
```

Testing Objectives

Similar to other ensemble modules, the automodel module allows you to explicitly set a testing objective to use during tuning. automodel will tune over the --testing_objective while reporting additional scores.

The following example shows how to explicitly set the --testing_objective to accuracy during tuning.

```
# skytree-server       automodel            \
  --training_in        ranking.train.st     \
  --training_labels_in ranking.train.targets \
  --num_folds          10                   \
  --model_out           ranking.model          \
  --testing_objective   accuracy
```

The available `--testing_objective` options vary depending on whether the problem is a classification or regression problem. Refer to *AutoModel Options* (page 208) in the Command Reference appendix for usage information. An error will occur if you attempt to include a classification testing objective in a regression problem and vice versa.

Saving Models

All of the models trained for each of the parameter settings can be saved with the `--tuning_models_out` option. This can only be used when a validation set is provided for tuning (i.e, when the `--tuning_in` and the `--tuning_labels_in` options are specified). The resulting tuning index file is reported in JSON format.

```
# skytree-server         automodel          \
  --training_in          income.data.st     \
  --training_labels_in   income.data.labels \
  --tuning_in            income.tune.st     \
  --tuning_labels_in     income.tune.labels \
  --num_trees            100                \
  --model_out            model.simple       \
  --tuning_models_out    income.tune.models
```

Saving Tuning Results

Tuning results can be saved to a file using the `--tuning_results_out` option. The results include the module to which the model belongs as well as a list of the parameters that were used. The output file is reported in JSON format.

```
# skytree-server        automodel              \
  --training_in         ranking.train.st       \
  --training_labels_in  ranking.train.targets  \
  --num_trees           10                     \
  --model_out           ranking.model          \
  --tuning_results_out  ranking.results
```

Computing Labels

The following example shows how to use `automodel` for testing a model and outputting the computed labels. In this case, `--model_in` is required.

```
# skytree-server        automodel       \
  --model_in            model.simple    \
  --testing_in          income.test.st  \
  --labels_out          labels.simple
```

Restarting Smart Search

Similar to `--smart_search` in `gbt` and `gbtr`, the `automodel` module allows you to specify the number of smart search iterations to run. In this case, there may be instances where you want to restart an `automodel` module project. The `--smart_search_restart_out` option allows you to specify a file name that saves the original smart search data. Once created, you can specify this file using the `--smart_search_restart_in` option to continue iterations from the original run.

The example below creates a smart search output file named "`ranking.50.restart`".

```
# skytree-server             automodel             \
  --training_in              ranking.train.st      \
  --training_labels_in       ranking.train.targets \
  --holdout_ratio            0.3                   \
  --num_folds                10                    \
  --smart_search_iterations  50                    \
  --smart_search_restart_out ranking.50.restart
```

The example above runs smart search with 50 iterations. The following example similarly specifies to run 100 smart search iterations. In addition, the "`ranking.50.restart`" file is specified as an input file. Because this file already runs 50 iterations, this example will run a total of 150 smart search iterations.

Note: When using `--smart_search_restart_in`, the specified input data files must be the same as the data specified in the original `--smart_search_restart_out` run.

```
# skytree-server             automodel             \
  --training_in              ranking.train.st      \
  --training_labels_in       ranking.train.targets \
  --smart_search_restart_in  ranking.50.restart    \
  --smart_search_iterations  100
```

Notice that only the smart search options and the input data files from the original run are required. Users can specify additional options only if those options were specified in the original smart search run (for example, `--num_folds`). In this case, the values for the options must match the original run.

PMML Model Export

Trained AutoModel models can be exported into PMML (Predictive Model Markup Language) format using the `--pmml_out` option. This allows the use of an external code for scoring.

Note: The PMML file that is produced requires that missing values are encoded as '?'. This is a reserved symbol for missing values, and as such, should not be used anywhere else in your data.

For classification models, the labels column is included in the PMML output. This is configured using the `--probability_threshold` parameter. If this is not specified when tuning a model, then it is set to the optimal probability threshold chosen by the model. If this is not specified, and tuning is not performed, then this value is set to 0.5.

For more information on PMML format, please refer to the PMML 4.2 section on the *Data Mining Group Web site* (http://www.dmg.org/v4-2/GeneralStructure.html).

Support Vector Machines

The Support Vector Machine (http://en.wikipedia.org/wiki/Support_vector_machine) (SVM) is one of the most successful classification algorithms. It is based on the revolutionary theory of kernel learning. The main concept is that SVM tries to fit a hyperplane that optimally separates different classes of data. The hyperplane can be linear or nonlinear.

- Find the best boundary to separate points into two classes
- Maps points onto a high-dimensional space so that it can learn complex nonlinear decision boundaries
- If no perfect separation, maximizes the margin
- An advantage is no local-minimum situations
- Naively scales as N^2-N^3; Skytree Server scaling is data dependent

Overview

The SVM is a form of linear classifier, though it can work efficiently in infinite-dimensional kernel spaces to learn detailed, nonlinear decision boundaries. We are given a set of points $R = \{(x_1;y_1) \ldots (x_n;y_n)\}$, where each $y_i \in \{-1,1\}$ is the class label for the i-th point $x_i \in R^D$. SVM learns a linear discriminant function $\hat{y}(x) = sgn(w^T \phi(x) + b)$ where $\phi : R^D \rightarrow H$ is a mapping from the input space to a feature space H. Here the learning implies computing w and b, the linear hyperplane that best separates the training examples with the label -1 and the ones with the label 1. The mapping ϕ could be chosen explicitly or be implicitly induced by a kernel function.

The training set is said to be linearly separable when there exists a linear discriminant function whose sign matches the class of all training examples:

$$\hat{y}(x_i) = y_i \; \forall \; 1 \leq i \leq N$$

A linearly separable data usually allows an infinite number of separating hyperplanes. Therefore, the hyperplane that maximizes the margin is chosen to predict better on unseen examples. The figure below shows the optimal hyperplane separating positive and negative examples with the maximal margin. The position of the hyperplane is solely determined by the few examples that are closest to it (the support vectors).

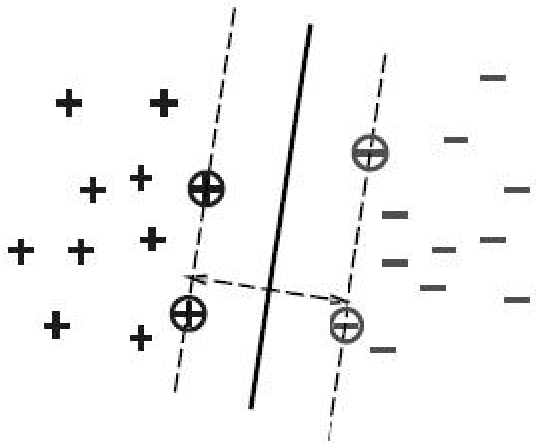

Figure 4: *Optimal hyperplane*

The following optimization problem expresses this choice.

$$min\ p(w,\ b)\ =\ \frac{1}{2}w^2$$

$$\text{subject to } \forall\, i\ \ y_i\,(w^T\phi\,(x_i)\ +\ b)\ \geq\ 1$$

Directly solving this problem is difficult because the constraints are quite complex. The mathematical tool of choice for simplifying this problem is the Lagrangian duality theory. This approach leads to solving the following dual problem:

$$max\, p(\alpha)\ = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N} y_1\alpha_i y_j\alpha_j\phi(x_i)^T\ \phi(x_j)$$

$$\text{subject to}\begin{cases}\forall_i & \alpha_i \geq\ 0,\\ \Sigma_i & y_i\alpha_i =\ 0\end{cases}$$

The linear discriminant function can then be written as:

$$\hat{y}\,(x)\ =\ w_*^T\ x + b^* = \sum_{i=1}^{N} y_i\alpha_i\,\phi\ (x_i)^T\ \phi(x)\ +\ b^*$$

The dual optimization problem and the linear discriminant function only involve the patterns x through the computation of dot products in feature space. There is no need to compute the features $\phi(x)$ when one knows how to compute the dot products directly and the resulting optimization problem is convex.

Instead of hand-choosing a feature function $\phi(x)$, it has been proposed to directly choose a kernel function $K\,(x;\, x')$ that represents a dot product $\phi(x)^T\,\phi(x')$. $K\,(x;\, x')$ induces a mapping from the original input space to a possibly high-dimensional feature space. Although the corresponding feature space could be infinite-dimensional, all computations can be performed without ever computing a feature vector $\phi(x)$. Complex nonlinear classifiers are computed using the linear mathematics of the optimal hyperplanes and this is referred to as the *kernel trick*.

Soft margins can be used in case the problem is noisy. If there exists no hyperplane that can split the "yes" and "no" examples, the Soft Margin method will choose a hyperplane that splits the examples as cleanly as possible, while still maximizing the distance to the nearest cleanly split examples. Further discussion of this topic is beyond the scope of this document. For more information on soft margins (http://en.wikipedia.org/wiki/Support_vector_machine#Soft_margin) and SVMs you may refer to [steinwart2008support].

Support Vector Machines are becoming more and more successful in business applications, [min2005bankruptcy], [van-bayesian].

Computational Complexity

The number of support vectors finally produced by the optimization is the critical component of the computational cost of solving the dual problem. Since the asymptotic number of support vectors grows linearly with the number of examples, the computational cost of solving the SVM problem has both a quadratic and a cubic component. Thus computational complexity is at least $O(N^2)$ but tends towards $O(N^3)$ for noisy problems. Empirical evidence shows that modern SVM solvers come close to these scaling laws.

- Kernels are expensive to compute. For example, the computation of the Gaussian kernel $e^{-\gamma(d^2)}$, where d is the distance between two observations, is an expensive operation taking many processor cycles.

- The size of the kernel matrix is N^2. For even medium sized datasets storing this in memory becomes prohibitive.

The most successful methods available today fall into the category of decomposition methods. They address the full-scale dual problem by solving a sequence of smaller quadratic programming sub-problems. The core of these methods is that they use small sized working sets, as few as two elements, and solve the corresponding optimization problem analytically. This choice dramatically simplifies the decomposition method.

SVM Usage Examples

Information on how to obtain the sample datasets used in the examples can be found in *datasets*. You can run the following command to see the options available for SVM:

```
# skytree-server svm --help
```

Note: The file format expected is the Skytree Server file format. Any CSV file can easily be converted to this format using a single command and the conversion tools that are provided. Refer to *Skytree Server File Format* (page 9) for more information.

Training and Testing a Linear SVM

The following command trains an SVM model using a linear kernel with a regularization parameter `0.1` and then predicts the labels for a test set using the trained model:

```
# skytree-server       svm                      \
  --training_in        sdss.train.st            \
  --training_labels_in sdss.train.labels        \
  --lambda             0.1                      \
  --testing_in         sdss.test.st             \
  --labels_out         sdss.test.labels.lsvm.0.1
```

This trains an SVM model with a bias term and the linear kernel for a regularization of `0.1` on `sdss.train.st`. Once the model is trained, the labels for the test set `sdss.test.st` are evaluated using this learned model and output in `sdss.test.labels.lsvm.0.1`. The kernel type is `linear` by default. The SVM model also uses a bias term by default. Hence the following command explicitly sets these values but is equivalent to the above in behavior:

```
# skytree-server       svm                      \
  --training_in        sdss.train.st            \
  --training_labels_in sdss.train.labels        \
  --lambda             0.1                      \
  --kernel             linear                   \
  --exclude_bias_term  off                      \
  --testing_in         sdss.test.st             \
  --labels_out         sdss.test.labels.lsvm.0.1
```

The bias term can be removed from the linear SVM formulation with the following command:

```
# skytree-server       svm                      \
  --training_in        sdss.train.st            \
  --training_labels_in sdss.train.labels        \
  --lambda             0.1                         \
  --exclude_bias_term                              \
  --testing_in         sdss.test.st                \
  --labels_out         sdss.test.labels.lsvm.0.1
```

A trained linear SVM model can be saved for later use in the following way:

```
# skytree-server       svm                         \
  --training_in        sdss.train.st               \
  --training_labels_in sdss.train.labels           \
  --lambda             0.1                         \
  --model_out          sdss.lsvm.model.0.1
```

This will save the linear SVM model in a file named `sdss.lsvm.model.0.1`. Once the model is saved, it can be used later for testing in the following manner:

```
# skytree-server  svm                              \
  --model_in      sdss.lsvm.model.0.1              \
  --testing_in    sdss.test.st                     \
  --labels_out    sdss.test.labels.lsvm.0.1
```

Tuning a Linear SVM

We allow the tuning of the regularization parameter `--lambda` in the SVM formulation. The following command returns the *10*-fold cross-validated scores for the regularization parameter at `0.01,0.1,1.0,10.0` and selects the setting with the best cross-validated accuracy:

```
# skytree-server       svm                \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels  \
  --lambda             0.01,0.1,1,10      \
  --num_folds          10
```

The regularization parameters can also be provided in the form of a range `<min>:<step-size>:<max>`. The following command will tune for the best regularization value among `0.1,0.2,0.3,0.4,0.5`:

```
# skytree-server       svm                \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels  \
  --lambda             0.1:0.1:0.5        \
  --num_folds          10
```

Monte Carlo cross-validation can be used instead of *k*-fold by specifying the `--holdout_ratio` in conjunction with the `--num_folds`. If `--num_folds=10` and `--holdout_ratio=0.2` then a random 20% of the training data would be heldout in each fold as the validation set. A validation set can also be provided by the user using the `--tuning_in` and the `--tuning_labels_in` options in the following way:

```
# skytree-server       svm                \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels  \
  --lambda             0.1:0.1:0.5        \
  --tuning_in           sdss.tune.st      \
  --tuning_labels_in    sdss.tune.labels
```

All of the linear SVM models trained for each of the parameter settings can be saved with the `--tuning_models_out` option. This can only be used when a validation set is provided for tuning (i.e, when the `--tuning_in` and the `--tuning_labels_in` options are specified).

The tuning results with the scores for each of the parameter settings can be saved to a file using the `--tuning_results_out` option. By default, the format for the output file is CSV. Users can change this to JSON format using the `--tuning_results_format` option.

```
# skytree-server          svm                     \
  --training_in           sdss.train.st           \
  --training_labels_in    sdss.train.labels       \
  --lambda                0.1:0.1:0.5             \
  --num_folds             10                      \
  --tuning_results_out    sdss.svm.tune.results \
  --tuning_results_format json
```

Note: The JSON format is of the following form:

```
[
 { "Parameters" : { ... }, "Metrics" : { ... }},
...
 { "Parameters" : { ... }, "Metrics" : { ... }}
]
```

Tuning and Testing a Linear SVM

Following tuning, the model with the best parameter setting can be used for predicted the labels for a test set in the following manner:

```
# skytree-server       svm                     \
  --training_in        sdss.train.st           \
  --training_labels_in sdss.train.labels       \
  --lambda             0.1:0.1:0.5             \
  --num_folds          10                      \
  --testing_in         sdss.test.st            \
  --labels_out         sdss.test.labels.lsvm
```

The linear SVM model with the best parameter setting in tuning can also be saved with the `--model_out` option for later prediction in the following manner:

```
# skytree-server       svm                     \
  --training_in        sdss.train.st           \
  --training_labels_in sdss.train.labels       \
  --lambda             0.1:0.1:0.5             \
  --num_folds          10                      \
  --testing_in         sdss.test.st            \
  --labels_out         sdss.test.labels.lsvm
```

Training and Testing a Non-Linear SVM

The following command creates an SVM model using the radial-basis function (RBF) kernel (also known as the Gaussian kernel) with a bandwidth 10 with a regularization parameter `0.1` and then predicts the labels for a test set using the trained model:

```
# skytree-server       svm                          \
  --training_in        sdss.train.st                \
  --training_labels_in sdss.train.labels            \
  --kernel             rbf                          \
  --rbf_bandwidth      10.0                         \
  --lambda             0.1                          \
  --testing_in         sdss.test.st                 \
  --labels_out         sdss.test.labels.rbf.10.svm.0.1
```

A polynomial kernel K is defined as $K(x, y) = s * x^T y+t)^d$, where d is the degree, s is the scale and t is the offset of the polynomial kernel. A polynomial kernel with degree 2 (with a default scale `1.0` and default offset `0.0`) can be used in the following manner:

```
# skytree-server       svm                          \
  --training_in        sdss.train.st                \
  --training_labels_in sdss.train.labels            \
  --kernel             polynomial                   \
  --polynomial_degree  2                            \
  --lambda             0.1                          \
  --testing_in         sdss.test.st                 \
  --labels_out         sdss.test.labels.poly.2.svm.0.1
```

The bias term in the nonlinear SVM formulation can be removed with the `--exclude_bias_term` option similar to linear SVM.

A trained linear SVM model can be saved for later use in the following way:

```
# skytree-server       svm                          \
  --training_in        sdss.train.st                \
  --training_labels_in sdss.train.labels            \
  --kernel             rbf                          \
  --rbf_bandwidth      1.0                          \
  --lambda             0.1                          \
  --model_out          sdss.lsvm.model.0.1
```

This will save the linear SVM model in a file named `sdss.lsvm.model.0.1`. Once the model is saved, it can be used later for testing in the following manner:

```
# skytree-server  svm                               \
  --model_in      sdss.lsvm.model.0.1               \
  --testing_in    sdss.test.st                      \
  --labels_out    sdss.test.labels.lsvm.0.1
```

The model can also be saved and loaded when running a distributed (polynomial) Skytree Server job.

```
# skytree-server       svm                          \
  --training_in        sdss.train.st            \
  --training_labels_in sdss.train.labels        \
  --kernel             polynomial               \
  --polynomial_degree  3                        \
  --polynomial_offset  1.0                      \
  --polynomial_scale   0.01                     \
  --lambda             0.1                      \
  --model_out          sdss.lsvm.model.0.1      \
  --hosts              host1,host2,host3
```

```
# skytree-server  svm                             \
  --model_in      sdss.lsvm.model.0.1             \
  --testing_in    sdss.test.st                    \
  --labels_out    sdss.test.labels.poly.2.svm.0.1  \
  --hosts         host1,host2,host3
```

Tuning a Non-Linear SVM

Along with the regularization parameter `--lambda`, the user can also tune over various kernel parameters. For the RBF kernel, the `svm` module allows tuning over the bandwidth (via the `--rbf_bandwidth` option). For the polynomial kernel, this module allows tuning over the degree, scale, and the offset (via the `--polynomial_degree`, `--polynomial_scale`, and `--polynomial_offset`).

For example, we can tune over regularization values `0.01,0.1` and RBF kernels with bandwidths `0.1,1.0,10.0` (for a total of 6 parameters settings) with the following:

```
# skytree-server       svm               \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --lambda             0.01,0.1          \
  --kernel             rbf               \
  --rbf_bandwidth      0.1,1,10          \
  --num_folds          10
```

The polynomial kernel parameters can be tuned over in a similar manner. The tuning options can used in a manner similar to the examples presented for the linear SVM.

Tuning and Testing a Non-linear SVM

Following tuning, the non-linear SVM model with the best parameter setting can be used for predicting the labels for a test set in the following way:

```
# skytree-server       svm                      \
  --training_in        sdss.train.st            \
  --training_labels_in sdss.train.labels        \
  --lambda             0.01,0.1                 \
  --kernel             rbf                      \
  --rbf_bandwidth      0.1,1,10                 \
  --num_folds          10                       \
  --testing_in         sdss.test.st             \
  --labels_out         sdss.test.labels.rbf.svm
```

The Probability Threshold

The predicted class label is derived from the probabilities generated for being of a particular class type. For example the probabilities that are output by `svm` are the probability of being class +1.

Typically, if for any given point, the probability of being +1 is > 0.5 then we assign that point the label of +1. This is the default behavior of Skytree Server as well when it is not doing automatic tuning. However, a probability threshold of 0.5 or 50% is not necessarily the best value for highest classification accuracy, or for that matter any scoring metric that utilizes the labels output. Therefore, when Skytree Server is in tuning mode it automatically reports the best probability threshold for any given metric.

Two cases arise depending on the `--testing_objective`.

1. `--testing_objective` is `accuracy` or `fscore` i.e. label based metric is being used to pick the best model. In this case the probability threshold is automatically tuned and applied in tuning as well as testing and stored with the model. It is also reported in the log. Therefore, if `--testing_objective` is accuracy then all the scores and models will use the best possible probability threshold for `accuracy`. Similarly for `fscore`.

2. `--testing_objective` is `gini` or `capture_dev` or `precision_at_k`, i.e. probability based metric is being used to pick the best model.

   a. In this case, if the `--testing_objective` is `gini` or `capture_dev`, then the `--classification_objective` argument comes into play. If `--classification_objective` is `accuracy`, then the model will still be picked based on best `gini` or `capture_dev` but the probability threshold will be tuned for `accuracy` and the labels outputted for the test data will be based on this probability threshold. Similarly, for `fscore`.

   b. In this case, if the `--testing_objective` is `precision_at_k` (where $k \in (0,1)$ is specified via `--k_for_precision` and defaults to 0.1), the `--classification_objective` is not used at all. The probability threshold is chosen to correspond to the top 100*k*-th percentile probability. This implies that any test point with the probability of being +1 higher than this threshold is probably in the top 100*k*-th percentile.

Tuning for Best Accuracy

A testing dataset can be specified via the `--testing_in` option when Skytree Server runs in tuning mode. Skytree Server will first tune for the best parameters and use the model giving the *best Gini* by default for prediction using the test set. If you want to pick the model with the best accuracy rather than the best Gini the `--testing_objective` argument can be used. This is illustrated in the following example:

```
# skytree-server       svm                   \
  --training_in        sdss.train.st         \
  --training_labels_in sdss.train.labels     \
  --lambda             0.1:0.1:0.5           \
  --num_folds          10                    \
  --testing_in         sdss.test.st          \
  --probabilities_out  probabilities.sdss    \
  --labels_out         sdss.test.labels.lsvm \
  --testing_objective  accuracy
```

Smart Search

In the case where users might not know the best configuration options to specify in order to train and tune a model, or in the case where users want to repeat an experiment, SVM allows for hyper-parameter optimization (smart search).

With a series of `--smart_search` options, users only need to specify the training set and the kernel type (if a kernel function other than the default `--kernel=linear` is desired). Based on the kernel type, SVM will automatically search for the best parameter settings without any additional required input from the user. When scoring, smart search will tune over the `--testing_objective` while reporting additional scores.

The example below specifies to try 1000 different parameter settings. In this example, because a kernel type is not specified, smart search will run using `--kernel=linear`. Note that if a different kernel function is desired, then it must be explicitly specified.

```
# skytree-server            svm                 \
  --training_in             sdss.train.st       \
  --training_labels_in      sdss.train.labels   \
  --smart_search                                \
  --smart_search_iterations 1000
```

By default, smart search restricts the parameter space (for example, in `--rbf_bandwidth`, `--lambda`, `--polynomial_offset`, etc.) in order to reduce the training time. If you do not want to restrict the upper bound for these parameters, specify `--limit_parameters=off`.

In some cases, users may want to input values for certain parameters and let SVM smart search tune over the rest. Users can provide open or closed intervals for any tuning parameter using <min>:<max> or <min>: or :<max>. Specific values at regular intervals can also be specified with <min>:<step>:<max>. For example:

- `--rbf_bandwidth=1.0:2.0` can be used to specify an interval.
- `--rbf_bandwidth=1.0:` specifies a lower bound.
- `--rbf_bandwidth=1.0:0.2:2.0` & `--rbf_bandwidth=1.1` can be used to specify exact values to use during smart search.

Tunable intervals or exact values that are explicitly set will be passed to `skytree-server`. All other parameters will be sampled during smart search.

Note: `--smart_search` cannot be used with `--model_in`. Similarly, `--probability_threshold` cannot be specified with `--smart_search` because this option cannot be used during tuning.

Note: Similar to restart in GBT and GBTR, users can restart a smart search run. Refer to *Restarting Smart Search* (page 87) in the GBT section for more information.

PMML Model Export

Trained SVM models can be exported into PMML (Predictive Model Markup Language) format using the `--pmml_out` option. This allows the use of an external code for scoring.

For more information on PMML format, please refer to the PMML 4.2 section on the *Data Mining Group Web site* (http://www.dmg.org/v4-2/GeneralStructure.html).

All options available for SVM are available for PMML output.

The labels column is included in the PMML output. Unlike in the ensemble methods, the labels column by default is defined in terms of `margin >= 0.0` and not in terms of `probability_1 >= 0.5`. If, however, the user provides `--probability_threshold`, or if SVM is being tuned, then the labels column is defined in terms of `probability_1 >= probability_threshold`.

Note: PMML output option is not supported for distributed non-linear SVM.

Notes on SVM Training

1. Datasets with randomly shuffled rows can increase `svm` performance (in terms of both accuracy and efficiency), especially in the distributed setting.
2. For better accuracy, stability, and convergence in the SVM training process, especially with the linear and polynomial kernels, all the features in the training set should be *scaled* to the same range. (For example, all the features can be scaled to be in the `[-1, 1]` range.) This means that the range of the feature values for each feature should be similar. There are two important things to note about scaling:
   a. The scaling coefficients used to scale the training set must also be used to scale the test/tuning/validation set.
   b. Scaling is different from normalization. Scaling just involves dividing each feature value with the maximum absolute feature value.

Metric Learning Methods

Some machine learning methods are independent to how the data is scaled, stretched, shifted, slanted, or rotated. While these *affine invariant* methods can be extremely useful, they are not always the fastest or most accurate means of estimation. The goal of metric learning is to enable more powerful yet scale-sensitive methods to operate on badly scaled data.

For instance, nearest neighbors search is invariant to overall scale, origin shift, and rotation—these either affect distances uniformly or don't affect them at all—but it is not invariant to the scales of individual dimensions. A single dimension with values in the millions will wash out the effect of all other dimensions if their values range from 0 to 1. Metric learning thus aims to learn scale weights for each dimension to optimize the prediction task, which is in this case often classification.

Though the goal is always the same, there are a number of different strategies for optimizing metric weights, including the following:

- Fit (page 117)
- Random (page 118)
- Forward Selection (page 118)
- Backward Selection (page 119)
- Forward/Backward Selection with Fit (page 120)
- Explore (page 120)
- Combination of Metric Learning with Tuning (page 122)

Combining Advanced Nearest Neighbor methods together with the Metric Learning and other Prediction methods can result in models that achieve the highest theoretically attainable classification accuracy.

The weighted nearest neighbors classifier wnnc from the Skytree Server Advanced Nearest Neighbor methods is applied on the SDSS dataset:

First, a baseline is established without metric learning:

```
# skytree-server       wnnc               \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels  \
  --imbalance                             \
  --k_neighbors        50                 \
  --testing_in         sdss.test.st       \
  --probabilities_out  probs
```

```
# skytree-server       score              \
  --true_labels_in     sdss.test.labels   \
  --probabilities_in   probs.class1
```

```
Gini index: 0.99052
```

In this example, some typical parameters are used, without tuning. Refer to the documentation for the Advanced Nearest Neighbor methods for additional details about the wnnc module.

In the following, we will show how to get even higher Gini indices through metric learning. These concepts are generally applicable to any datasets.

Fit

Metric weights can be found by fitting a linear model on the training data. This is specified with `--fit_metric_weights`:

```
# skytree-server           wnnc               \
  --training_in            sdss.train.st      \
  --training_labels_in     sdss.train.labels  \
  --imbalance                                 \
  --k_neighbors            50                 \
  --fit_metric_weights                        \
  --fit_metric_weights_out fit.csv
```

This will compute a decent set of weights and write them to a file called `fit.csv`. You can also try `--fit_method=[1|2]`. Note that the fitted weights are not necessarily optimal for Gini, F-Score, etc. You can also try specifying a threshold with `--fit_threshold 5` (or other values). The fit threshold prunes dimensions that seem unhelpful by weighting them to 0. Lower values are more aggressive, but `--fit_threshold=0` turns pruning off. The default threshold is 8 for `--fit_metric_weights` but is turned off (0) for the `backward_fit` and `forward_fit` methods.

The `--fit_metric_weights` option can also be combined with `--metric_learning` (explained in the next section), in which case metric learning would try to improve on the initial weights obtained by fitting.

Alternately, you can use `--metric_weights_in` to let optimization continue from a user-specified weight vector, or where metric learning left off, even if switching to another metric learning method. Note that `--fit_metric_weights` trumps `---metric_weights_in`.

Random

The simplest metric learning method is to use a different random weight (i.e., sampled uniformly from the interval [0, 1] for each dimension (feature), and use a different random weight vector at every iteration. The random generator can be seeded with `--metric_learning_random_seed` for reproducibility.

```
# skytree-server                wnnc              \
  --training_in                 sdss.train.st     \
  --training_labels_in          sdss.train.labels \
  --imbalance                                     \
  --k_neighbors                 50                \
  --metric_learning                               \
  --metric_learning_method      random            \
  --metric_learning_random_seed 123               \
  --metric_learning_iterations  10                \
  --metric_learning_objective   gini              \
  --optimal_metric_weights_out  random.csv
```

This will perform leave-one-out cross-validation for metric learning with 10 iterations of random metric weight vectors and finally write the best weights found so far (in the above example for F-score) to a file called `random.csv`:

```
cat random.csv
1.1645573427262690e-02
2.6797112819664903e-02
8.5948552910153353e-04
3.4347075288359008e-02
```

Forward Selection

In addition to `random`, there are two other metric learning methods for forward and backward selection.

Forward selection is the concept of finding the best weight vector consisting of only one non-trivial dimension, and then successively adding the single best next dimension. The final weight vector is the optimal subset of all available dimensions given the rule of sticking with the best dimensions for each previous iteration (dimensionality).

Given a simple synthetic example of forward-selection for 4D data that goes through 10 trials:

4 trials in 1D: 1 0 0 0, 0 1 0 0, 0 0 1 0, 0 0 0 1, keep the best, for example, 0 0 1 0

3 trials in 2D: 1 0 1 0, 0 1 1 0, 0 0 1 1, keep the best, for example, 0 0 1 1

2 trials in 3D: 1 0 1 1, 0 1 1 1, might be no improvement on 0 0 1 1

1 trial in 4D : 1 1 1 1, might still be worse than 0 0 1 1

In this example, the `optimal_metric_weights_out` file would represent the weight vector `0 0 1 1`.

Now applied to the SDSS dataset:

```
# skytree-server               wnnc              \
  --training_in                sdss.train.st     \
  --training_labels_in         sdss.train.labels \
  --imbalance                                    \
  --k_neighbors                50                \
  --metric_learning                              \
  --metric_learning_method     forward           \
  --optimal_metric_weights_out forward.csv
```

In this case, we see that all 4 dimensions of the dataset seem important enough to keep, as `forward.csv` contains all weights of 1.0. Note that there are 2^4 = 16 different possible weight vectors by selecting individual dimensions, but only 10 were seen during the path chosen during forward selection, so it is possible that the global optimum is missed, especially in higher dimensionalities.

If `--metric_weights_in` was given, then instead of weights of 0 or 1.0, the optimal weights subset will be taken from the specified weights. This can be very useful, but it also opens up an infinite optimization space and can require a lot of computational resources even if accompanied by a good optimization strategy.

Backward Selection

Backward selection is similar to forward selection, but it starts with the full weight vector (all weights of 1.0 if no `--metric_weights_in` is given), and then tries to reduce the dimensionality one by one.

```
# skytree-server               wnnc              \
  --training_in                sdss.train.st     \
  --training_labels_in         sdss.train.labels \
  --imbalance                                    \
  --k_neighbors                50                \
  --metric_learning                              \
  --metric_learning_method     backward          \
  --optimal_metric_weights_out backward.csv
```

In this case, the `backward.csv` metric vector represents `1 1 1 0`, indicating that for this particular dataset and for `--metric_learning_objective=gini` (the default value), it is better to ignore the 4th dimension. This also shows that forward selection must have skipped this particular weight vector. Backward selection can sometimes be better if most dimensions are expected to be important, especially when starting with a good initial weight vector.

Most likely, the *global optimum* for the weight vector does not just contain values of 0 or 1.0, so it makes sense to use `--metric_weights_in`, or perform more advanced metric learning steps as explained in the sections that follow.

```
# skytree-server               wnnc              \
  --training_in                sdss.train.st     \
  --training_labels_in         sdss.train.labels \
  --imbalance                                    \
  --k_neighbors                50                \
  --metric_weights_in          random.csv        \
  --metric_learning                              \
  --metric_learning_method     backward          \
  --optimal_metric_weights_out random_backward.csv
```

Also note that for backward and `backward_fit` (see the next section), it is possible to terminate the iterations before the weight vector goes all the way down to 1D. This can be done using the `--metric_learning_min_dim_ratio` option. A value of 0.5, for example, would specify that the lowest dimensionality to be tried is half the number of full dimensions.

Forward/Backward Selection with Fit

Forward or backward selection can sometimes help when fitting a new *D*-dimensional weight vector for every iteration during forward or backward selection. For fitting, only dimensions specified by the selection path taken so far will end up with non-zero weights.

This can be obtained by specifying `--metric_learning_method=[forward_fit|backward_fit]`:

```
# skytree-server                wnnc              \
  --training_in                 sdss.train.st     \
  --training_labels_in          sdss.train.labels \
  --imbalance                                     \
  --k_neighbors                 50                \
  --metric_learning                               \
  --metric_learning_method      forward_fit       \
  --optimal_metric_weights_out forward_fit.csv
```

```
# skytree-server                wnnc              \
  --training_in                 sdss.train.st     \
  --training_labels_in          sdss.train.labels \
  --imbalance                                     \
  --k_neighbors                 50                \
  --metric_learning                               \
  --metric_learning_method      backward_fit      \
  --optimal_metric_weights_out backward_fit.csv
```

Note: Weights given by `--metric_weights_in` have no effect on `forward_fit` or `backward_fit`, as they would be overwritten during the first iteration.

Explore

Probably the most powerful metric learning method in Skytree Server is the `explore` method. It can lead to extremely accurate results if given enough iterations. Starting with the best initial weights we obtained earlier, the `explore` method tries to slightly modify each dimension's weight based on advanced Monte Carlo methods, accepting and rejecting updates based on multiple criteria.

```
# skytree-server                 wnnc              \
  --training_in                  sdss.train.st     \
  --training_labels_in           sdss.train.labels \
  --imbalance                                      \
  --k_neighbors                  50                \
  --metric_weights_in            backward.csv      \
  --metric_learning                                \
  --metric_learning_method       explore           \
  --metric_learning_iterations   50                \
  --metric_learning_random_seed 999                \
  --optimal_metric_weights_out  backward_explore_50.csv
```

After 50 `explore` iterations, the weights have changed from `1 1 1 0` to

```
# cat backward_explore_50.csv
  8.3307723740124218e-01
  1.4389604326347196e+00
  1.5398424440030496e+00
  3.6836467139240275e-01
```

and the Gini index has increased significantly from the baseline of 0.99052:

```
# skytree-server       wnnc                     \
  --training_in        sdss.train.st            \
  --training_labels_in sdss.train.labels        \
  --testing_in         sdss.test.st             \
  --imbalance                                   \
  --k_neighbors        50                       \
  --metric_weights_in  backward_explore_50.csv  \
  --probabilities_out  probs
```

```
# skytree-server     score             \
  --true_labels_in   sdss.test.labels  \
  --probabilities_in probs.class1
```

```
Gini index: 0.992115
```

This process can be repeated for many iterations, or by re-starting from previously obtained metric weights:

```
# skytree-server                wnnc                     \
  --training_in                 sdss.train.st            \
  --training_labels_in          sdss.train.labels        \
  --imbalance                                            \
  --k_neighbors                 50                       \
  --metric_weights_in           backward_explore_50.csv  \
  --metric_learning                                      \
  --metric_learning_method      explore                  \
  --metric_learning_iterations  50                       \
  --metric_learning_random_seed 999                      \
  --optimal_metric_weights_out  backward_explore_50.again.csv
```

Again, the Gini index has increased: 0.992293.

Further options to the explore metric learning method are `--explore_persistence` and `--explore_scales_in`. The first regulates the “risk appetite” of the method, while the latter can point to a file indicating the amount of variation to be taken for each dimension. For example, if the weight for the first dimension should be kept constant, but the weight for the 4th dimension is allowed to change a lot, then the following could be done:

```
# echo "0 0.1 0.1 0.5" > explore_scales
```

```
# skytree-server                wnnc                     \
  --training_in                 sdss.train.st            \
  --training_labels_in          sdss.train.labels        \
  --imbalance                                            \
  --k_neighbors                 50                       \
  --metric_weights_in           backward_explore_50.csv  \
  --metric_learning                                      \
  --metric_learning_method      explore                  \
  --metric_learning_iterations  50                       \
  --metric_learning_random_seed 192                      \
  --explore_scales_in           explore_scales           \
  --optimal_metric_weights_out  backward_explore_50.scale.csv
```

Now, the weights will only have changed for the last 3 dimensions.

```
# cat backward_explore_50.scale.csv
  8.3307723740124218e-01
  1.4243593341817145e+00
  1.5702470122195578e+00
  4.4279322735316989e-01
```

Note that the (biased) Gini index has improved again during the last metric learning tuning run, but the (unbiased) testing Gini index has actually decreased:

```
# skytree-server       wnnc                           \
  --training_in        sdss.train.st                  \
  --training_labels_in sdss.train.labels              \
  --testing_in         sdss.test.st                   \
  --imbalance                                         \
  --k_neighbors        50                             \
  --metric_weights_in  backward_explore_50.scale.csv  \
  --probabilities_out  probs
```

```
# skytree-server     score              \
  --true_labels_in   sdss.test.labels   \
  --probabilities_in probs.class1
```

```
Gini index: 0.992166
```

To reiterate, keep in mind that any Ginis, etc., found on the training references and/or tuning data are *biased* as they have been *optimized*. To obtain unbiased estimates of Gini, it is necessary to rerun metric-weighted classification on a validation (test) set (one that played no role in optimization) via `--metric_weights_in`, as shown here.

Combination of Metric Learning with Tuning

Each of the metric learning methods introduced so far can lead to good results. It makes sense to try them all, but keep in mind that `forward`, `forward_fit`, `backward`, and `backward_fit` all have quadratic iteration counts with the number of dimensions. But especially one initial (deterministic) `backward` run (potentially only going down to a relatively large fraction of dimensions) in combination with many successive `explore` iterations can be the shortest path to obtaining a model with high predictive strength.

All metric learning methods can automatically be combined with tuning over the number of neighbors and distance kernel bandwidths (potentially for both classes in binary classification).

It might be helpful to note that all above metric learning methods can be terminated with `Ctrl+C`, and optimal metric weights (so far) will still be written to file.

Alternatively, if iterations are taking too long, you can enable rank approximation. For example, try this:

```
# --rank_error_tol  0.3 \
  --rank_error_prob 0.9
```

Or you can try `--sample_imbalance` if `--imbalance` is also used.

You can also optimize for things other than Gini, by specifying `--metric_learning_objective fscore`. Other available objectives include `accuracy` and `capturedev`.

Don't forget to try different distance weight functions, such as `1/r`, `1/r^2`, `fixed_gaussian`, `gaussian`, `fixed_epan` and `epan`. For `gaussian` and `epan` kernels, bandwidths are important (and can be tuned over).

One such combined example would be as follows:

```
# skytree-server                wnnc                      \
  --training_in                 sdss.train.st             \
  --training_labels_in          sdss.train.labels         \
  --dist_weight                 gaussian                  \
  --imbalance                                             \
  --k_neighbors                 2:2:20                    \
  --k_neighbors                 2:2:20                    \
  --bandwidth                   0.1:0.2:1.0               \
  --bandwidth                   0.1:0.2:1.0               \
  --metric_weights_in           backward_explore_50.csv   \
  --metric_learning                                       \
  --metric_learning_method      explore                   \
  --metric_learning_iterations  50                        \
  --metric_learning_random_seed 192                       \
  --explore_scales_in           explore_scales            \
  --optimal_metric_weights_out  backward_explore_50.tune.csv
```

```
Class -1 k-value(s): 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,
Class -1 bandwidth(s): 0.1, 0.3, 0.5, 0.7, 0.9,
Class 1 k-value(s): 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,
Class 1 bandwidth(s): 0.1, 0.3, 0.5, 0.7, 0.9,
```

Now each of the 50 metric learning iterations will tune over 2500 parameter configurations.

The optimal metric weights and the optimal model parameters (as reported at the end for best Gini, in this example) should lead to an improved model.

```
# skytree-server          wnnc                         \
  --training_in           sdss.train.st                \
  --training_labels_in    sdss.train.labels            \
  --dist_weight           gaussian                     \
  --imbalance                                          \
  --k_neighbors           20                           \
  --k_neighbors           2                            \
  --bandwidth             0.1                          \
  --bandwidth             0.5                          \
  --probability_threshold 0.9810441889047585           \
  --metric_weights_in     backward_explore_50.tune.csv \
  --testing_in            sdss.test.st                 \
  --probabilities_out     probs
```

```
# skytree-server     score              \
  --true_labels_in   sdss.test.labels \
  --probabilities_in probs.class1
```

```
Gini index: 0.994496
```

The Gini index has increased significantly over the baseline results. While this was just a small dataset, all of the above concepts should translate well to larger datasets, leading to models with unprecedented accuracy.

Logistic Regression

The Skytree Server Logistic Regression module implements an L1 regularized, memory-efficient, and fast algorithm for binary classification. Many descriptive statistics on the resulting model are also output.

Information on how to obtain the sample datasets used in the examples can be found in *datasets*. You can run the following command to see the options available for logistic regression:

```
# skytree-server logistic --help
```

You can also refer to *Logistic Regression Options* (page 286) in the Command Reference Appendix for information on the available options.

A Simple Example

Run the Logistic Regression module with the following arguments:

```
# skytree-server       logistic          \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels  \
  --coefficients_out   coeffs
```

The above call will compute the logistic regression model, on the data provided in "`sdss.train.st`". The coefficients for the model, which will contain the intercept coefficient as well, will be output to the file called "`coeffs`".

Excluding the Bias Term

By default, a bias or intercept term is included in the computed model. The bias term adjusts the log-odds up or down by a constant amount, i.e. it is the predicted log-odds when all inputs are exactly 0. It can be excluded (forced equal 0) with `--exclude_bias_term`:

```
# skytree-server       logistic          \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --coefficients_out   coeffs            \
  --exclude_bias_term
```

Excluding the bias term will negatively impact measured model quality, but is appropriate when a given logistic regression model should definitely predict 0.5 when all inputs are 0.

Regularization Term

Skytree Server Logistic Regression is L1-regularized. The default value for the regularization term is set to 1*e* - 6. This value is sufficient to completely fit the model to the data. Higher values can be used to ensure under-fit models that may suit the problem better.

```
# skytree-server       logistic          \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --coefficients_out   coeffs            \
  --regularization     0.001
```

Altering the regularization term will penalize the L1-norm of the coefficients in the logistic regression.

Training and Testing

To predict probabilities for a testing dataset the following simple example can be used.

```
# skytree-server       logistic                \
  --training_in        sdss.train.st           \
  --training_labels_in sdss.train.labels       \
  --coefficients_out   coeffs                  \
  --testing_in         sdss.test.st            \
  --probabilities_out  sdss.test.probabilities \
  --labels_out         sdss.test.labels.out
```

Testing Only

Once the coefficients have been saved, they can be reloaded to predict the labels and probabilities of test datasets as follows.

```
# skytree-server      logistic                 \
  --coefficients_in   coeffs                   \
  --testing_in        sdss.test.st             \
  --probabilities_out sdss.test.probabilities  \
  --labels_out        sdss.test.labels
```

Linear Regression

Linear regression is an approach to modeling the relationship between a scalar variable y and one or more explanatory variables denoted X. For example, we might want to predict the stock market index from the temperature, the air pollution level and highway traffic. We could train a linear regression model from historical data. Obviously the accuracy of the predictor wouldn't be high. Just finding a linear model that has high accuracy is not always what we want. Interpretability of the model over the factors (in this case temperature, air pollution etc) is very important too. For example, we would like to eliminate factors that are not contributing to the prediction. Stepwise regression (http://en.wikipedia.org/wiki/Stepwise_regression) is one way of eliminating factors.

Overview

Linear regression models the relationship between a group of regressors (predictors) $X_1, \cdots X_p$ and the prediction values y, using a linear fit. Given the following data set

$$\left\{y_i, x_{i1}, \cdots, x_{ip}\right\}_{i=1}^{N}$$

the model takes the following form:

$$y = X\beta + \epsilon$$

where

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, X = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_N^T \end{pmatrix}, \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}, \epsilon = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_N \end{pmatrix}$$

X is called the design matrix, and one could include the *bias term* by appending a constant 1 vector to X, that is to add $x_{i0} = 1$ for $i = 1, \cdots N$. In this case, the model has an additional coefficient such that $\beta = (\beta_0, \beta_1, \cdots, \beta_p)^T$. The following assumptions are made:

1. The design matrix X is of full column rank. Otherwise, β is not uniquely defined, leading to numerical instability.
2. ϵ_i is an independent Gaussian noise.
3. The predictors x_i are error-free.

If the design matrix is of full rank, one way of computing the linear model β is to use the ordinary least squares:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

Feature Selection

If the full column rank assumption is not satisfied, we resort to feature selection, which removes redundant predictor features. The following algorithms are typically used.

- Backward elimination with variance inflation factor (VIF) selection, optionally preceded by the correlation-based feature selection and optionally followed by the bidirectional stepwise regression.
- Least absolute shrinkage and selection operator (LASSO).

Supported Features

The following features are currently supported:

- Correlation-based feature pruning.
- Backward elimination with VIF criterion.

- Bidirectional stepwise regression based on the AIC criterion.

Correlation-Based Pruning

The interdependence between one feature and another can be measured by the Pearson product-moment correlation coefficient, which can be thought of as a normalized measure of covariance. It is defined for the feature X_1 and X_2:

$$\rho_{X_1 X_2} = \frac{cov(X_1, X_2)}{\sigma_{X_1}\sigma_{X_2}} = \frac{E\left[(X_1-\mu_{X_1})(X_2-\mu_{X_2})\right]}{\sigma_{X_1}\sigma_{X_2}}$$

where μ_{X_1}, μ_{X_2} are expected values, and σ_{X_1}, σ_{X_2} are population variances of X_1 and X_2 respectively.

Given a data set

$$\{y_1, x_{i1}, \cdots, x_{ip}\}_{i=1}^{N},$$

the corresponding sample correlation coefficient is defined as:

$$r_{X_1, X_2} = \frac{\sum_{i=1}^{N}(X_{1,i}-\overline{X_1})(X_{2,i}-\overline{X_2})}{(N-1)s_{X_1}s_{X_2}}$$

where

$$\overline{X_1} = \frac{1}{N}\sum_{i=1}^{N} X_{1i} \quad \text{and} \quad \overline{X_2} = \frac{1}{N}\sum_{i=1}^{N} X_{2i}$$

are sample means, and

$$s_{X_1} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(X_{1,i}-\overline{X_1})^2} \quad \text{and} \quad s_{X_2} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(X_{2,i}-\overline{X_2})^2}$$

are standard deviations of X_1 and X_2 respectively.

By the Cauchy-Schwarz inequality, it can be shown that $|r_{X_1 X_2}| \leq 1$. A sample correlation coefficient close to 1 implies an increasing linear relationship, while being close to -1 implies a decreasing linear relationship. A sample correlation coefficient close to 0 implies a lack of correlation (but does not imply independence unless X_1 and X_2 are jointly Gaussian).

This pruning method is computationally cheaper than the backward elimination with VIF criteria or the bidirectional stepwise regression; the variances of each feature and the covariances of each pair of features can be precomputed and do not involve matrix-inversion.

Backward Elimination with VIF Selection

Backward elimination with variance inflation factor (VIF) selection is preceded optionally by the correlation-pruning described in the previous section. VIF is a method of quantifying the degree of collinearity. For example, suppose we have the following regression equation with *k* independent variables:

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_k X_k + \epsilon$$

Then the variance inflation factor for $\hat{\beta}_i$ is given by regressing the rest of the variables X_j for $j \neq i$ against X_i.

$$X_i = c_0 + \sum_{j \neq i} a_j X_j + e$$

and computing the following:

$$VIF(\hat{\beta}_i) = \frac{1}{1 - R_i^2}$$

Where R_i is the coefficient of determination.

The feature elimination algorithm consists of two nested loops; the inner loop takes the current set of features that have to be considered for pruning and regresses one of the features at a time against the rest and computes its VIF. If the maximum VIF, after the termination of the inner loop, is more than the pre-specified threshold, then the feature with the corresponding VIF is eliminated. The outer loop continues either until there are only two features in the prune set or the inner loop fails to find a feature with a large VIF. The following is the pseudocode for the algorithm:

Stepwise Regression

Stepwise regression is a greedy feature selection algorithm. It starts with a set of features and considers an action of adding a feature from the set of inactive features and removing a feature from the currently active set of features. The decision of whether to add or remove is based on the maximization of the fitness of the linear model, for which AIC, BIC, and the F-test criterion are widely used. Skytree Server uses foward/backward hybrid stepwise selection with AIC.

Usage Examples

Run the following command to see the options available for linear regression.

```
# skytree-server linear --help
```

You can also refer to *Linear Regression Options* (page 282) in the Command Reference Appendix for information on the available options.

A Simple Example

Run the Linear Regression module with the following arguments:

```
# skytree-server     linear           \
  --references_in    sample_promo.st \
  --prediction_index 0
```

The above call will compute the linear regression model, on the data provided in “`sample_promo.st`”, by using the backward-elimination with VIF criteria algorithm. Notice that the prediction value, also called y, is in this case the first non-metadata column and is indexed by 0, as suggested above. If `--prediction_index` is left out, the first attribute (index 0) is used by default. The rest of the attributes are used to predict y.

Excluding the Bias Term

By default, a bias term is included in the computed model. The bias term adjusts all predictions up or down by a constant amount, i.e. it is the predicted value when all inputs are exactly 0. It can be excluded (forced equal 0) with `--exclude_bias_term`:

```
# skytree-server      linear          \
  --references_in     sample_promo.st \
  --exclude_bias_term
```

Excluding the bias term can negatively impact measured model quality, but it is appropriate when a given regression model should definitely predict 0 when all inputs are 0.

VIF Selection Threshold

The threshold for the VIF selection can be set through the `--vif_threshold` parameter:

```
# skytree-server     linear          \
  --references_in    sample_promo.st \
  --prediction_index 0               \
  --vif_threshold    5.0
```

Altering the VIF threshold affects how readily linear regression will eliminate redundant features. A lower VIF threshold will result in fewer features contributing to the final model.

Outputting the Model

To output model coefficients, use:

```
# skytree-server     linear          \
  --references_in    sample_promo.st \
  --prediction_index 0               \
  --coeffs_out       coefficients.st
```

If the bias term is used in the model, its value will be written to the coefficient with index equal to that of the predicted variable; otherwise, the value at that index is 0. In this example, the bias term is written to the first coefficient.

Outputting Statistics

Linear regression can also output a host of statistics describing the quality of the model. These are available by enabling the `--output_statistics` flag, and are each written to their own files as follows:

```
# skytree-server      linear                    \
  --references_in     sample_promo.st           \
  --coeffs_out        coefficients.st           \
  --std_errors_out    standard_errors.st        \
  --conf_los_out      confidence_band_lows.st   \
  --conf_his_out      confidence_band_highs.st  \
  --t_values_out      t_values.st               \
  --p_values_out      p_values.st               \
  --adj_r_squared_out adjusted_r_squared.st     \
  --f_statistic_out   f_statistic               \
  --r_squared_out     r_squared.st              \
  --sigma_out         sigma.st                  \
  --output_statistics true
```

A More Advanced Example

The following example will run linear regression again with the first feature as the prediction index. Correlation pruning is enabled, which precedes VIF selection and will remove redundant features with correlation coefficients greater than 0.7. Afterward, the algorithm will use bidirectional stepwise optimization to find an optimal, VIF-selected model, where each step requires a minimum improvement of 0.5.

```
# skytree-server          linear          \
  --references_in         sample_promo.st \
  --prediction_index      0               \
  --correlation_pruning                   \
  --correlation_threshold 0.7             \
  --stepwise                              \
  --stepwise_threshold    0.5
```

Generalized Linear Models

The Generalized Linear Model (GLM) is an extension of the linear model, which for data **x** = $(1, x_1, \dots, x_n)$ and response variable y finds linear weights $\beta = (\beta_0, \beta_1, \dots, \beta_n)$ such that

$$\hat{y} = \mathrm{x}^T\beta = \beta_0 + x_1\beta_1 + \cdots + x_n\beta_n$$

is a suitable estimate of y. Optimal model weights are usually chosen such that the estimated responses minimize the sum of squared errors for a training dataset of size m,

$$\sum_{1 \le i \le m} (y_i - \mathbf{x}_i^T\beta)^2$$

This form of estimation is equivalent to maximum likelihood estimation, if one assumes that responses y are sampled from a normal distribution with mean $\mu = \mathbf{x}^T\beta$ and fixed variance.

The above configuration is limited in several ways. Notably, modeling y as a linear combination of the features of **x** means that any constant change in any of the predictors is met with a corresponding constant (though scaled) change in the prediction $\hat{y}$. Consequently, it is not possible to enforce that $\hat{y}$ always remain in some sensible range, e.g., greater than zero, regardless of the predictors. In addition, the assumption that y is normally distributed with fixed variance may be incorrect for the given data. As a result, optimization may be disproportionately influenced by higher-variance regions of the data, sacrificing the performance elsewhere, and may yield predictions that do not align with the underlying distribution's expected value.

In contrast to the above, GLM can fit β for responses y sampled from any of various distributions in the exponential family. Depending on the distribution chosen, this allows for y to have either fixed variance or variance that scales as a function of $\mathbf{x}^T\beta$. In addition, GLM can fit linear models to non-linear transformations of the response

$$g(\mu) = \eta = \mathbf{x}^T\beta$$

for μ the expected value of y and a *link function* g such as the logarithm, square root, or multiplicative inverse. The predicted values are then obtained

$$\hat{y} = g^{-1}(\eta) = g^{-1}(\mathbf{x}^T\beta)$$

GLM Components

There are three key components to any GLM problem:

- A probability distribution family f in the exponential family with expected value $\mu = E_f(y)$
- A linear model $\eta = \mathbf{x}^T\beta$
- A link function g relating $g(\mu) = \eta$

Accordingly, in order to specify a GLM problem, you must choose family function f, link function g, and any parameters needed to train the linear model weights β.

Note: A family function cannot be specified in `glmc` because only a single family (Binomial) is available.

Distribution Families

The different choices for the family in GLM is usually governed by how the distribution family models variance with respect to the mean. This is important because there are situations where responses with larger magnitude are allowed to have larger variance in the modeling process. (For example, when insurance claims are of the order of thousands, a variance/error of the order of hundreds is acceptable, but claims of the order of hundreds can only tolerate a variance in the order of tens.)

Skytree Server supports the following probability distribution families in GLM:

Regression:

- **Gaussian**: The usual least-squares linear regression distribution. Variance is constant with respect to μ.
- **Poisson**: Generally used for positive integral targets. Skytree Server can handle non-integer targets if provided. Variance scales proportionally to μ.
- **Gamma**: For positive targets. Variance scales proportionally to μ^2.
- **Tweedie**: Also known as the compound Poisson-gamma distributions. Provides a continuous spectrum from Poisson distribution to the gamma distribution. Variance scales proportionally to μ^p, where $1 < p < 2$ is the Tweedie exponent.

Classification:

- **Binomial**: Useful when the targets take one of only two values/levels. (Usually a lower value denotes failure of some event of interest while the higher value denotes success.) Hence, it is used in binary classification where the targets are labels. The variance scales proportionally to $\mu(1-\mu)$.

Link Functions

Skytree Server currently supports the following Link Functions:

Regression:

- **Canonical**: This can be used with any `glmr` Family option. This is equivalent to one of the following for each of the paired family functions.
- **Identity**: This can be used with a Gaussian or Poisson family.

$$E_f(y) = \eta$$

- **Square-Root**: This can be used with the Poisson family.

$$E_f(y) = \eta^2$$

- **Inverse**: This can be used with the Gamma Family.

$$E_f(y) = \frac{1}{\eta}$$

- **Log**: This can be used with a Gamma, Poisson, or Tweedie Family.

$$E_f(y) = e^{\eta}$$

Note: Certain link functions, such as Log and Square-Root, are only applicable to non-negative responses.

Classification:

- **Logit**: The Logistic link function can be used with the Binomial family in `glmc`.

$$E_f(y) = 1 / (1 + \exp(-\eta))$$

- **Cloglog**: The Complementary Log-Log function can be used with the Binomial family in `glmc`.

$$E_f(y) = 1 - \exp(-\exp(\eta))$$

The following table shows the accepted combinations of Family/Link Function.

Table 2: *Family/Link Function Combinations*

Family	Link Function					
	Identity	Square Root	Inverse	Log	Logit	Cloglog
Gaussian	X					
Poisson	X	X		X		
Gamma			X	X		
Tweedie				X		
Binomial					X	X

Note: Canonical link functions for supported families:
• Gaussian, identity
• Poisson, log
• Gamma, inverse
• Tweedie, log
• Binomial, logit

Regularization

Both least-squares linear regression and GLM are subject to overfitting the model weights β, in which component coefficients β_j become large but contradictory so as to exploit minor fluctuations in the training data in order to more exactly reproduce the response. While training error may diminish to near zero in this case, testing error tends to worsen and may become extreme. This always occurs if the number of features is sufficiently large with regard to the number of samples; accordingly, it is important to employ a form of *regularization*.

Regularization constitutes a shift towards bias (simpler models) and away from variance (overfitting). It can be thought of as a form of Occam's razor; specifically, it asserts that models featuring large coefficients that yet delicately cancel so as to obtain the observed responses are extremely unlikely and should be disregarded. It accomplishes this by adding a penalty term directly to the optimized value. In the case of least-squares linear regression, this may be written

$$\sum_{1\le i\le n} (y_i - \mathbf{x}_i^T\beta)^2 + \lambda_1 \sum_{1\le j\le d} \left|\beta_j\right| + \frac{\lambda_2}{2} \sum_{1\le j\le d} \beta_j^2$$

where the non-negative λ_1 and λ_2 terms correspond to the L1 and L2 penalties. If λ_2 is non-zero and λ_1 is zero, the above is known as ridge regression. If λ_1 is non-zero and λ_2 is zero, then the above is known as the LASSO. If both λ_1 and λ_2 are non-zero, the above is known as the elastic net method.

L1 and L2 penalties produce models with slightly different characters from one another. In particular, larger L1 penalties will tend to produce sparse models, with many coefficients β_j exactly equal to zero. This can be thought of as optimization completely eliminating irrelevant features, or selecting only one feature from each group of correlated features. On the other hand, L2 regularization tends to assign non-zero weights to all of the features. This can produce models with less overall bias and thus potentially higher accuracy, but also more complexity. Mixing the two penalties results in a compromise between these two effects: moderately sparse models that are nonetheless still sensitive to the slight differences of even correlated features.

Like any parameter, the L1 and L2 penalty factors must be adequately tuned to train good models. Their optimal values may vary as a function of dataset size, the scale of the features, and the distribution of the data.

PMML Model Export

Trained GLM models can be exported into PMML (Predictive Model Markup Language) format using the `--pmml_out` option. This allows the use of an external code for scoring.

For more information on PMML format, please refer to the PMML 4.2 section on the *Data Mining Group Web site* (http://www.dmg.org/v4-2/GeneralStructure.html).

All options available in `glmc` and `glmr` are available for PMML output.

For `glmc`, the labels column is included in the PMML output. Similar to ensemble methods, the labels column by default is defined in terms `probability_1 >= 0.5`.

Usage Examples

Run the following command to see the options available for GLM (classification and regression).

```
# skytree-server glmr --help
# skytree-server glmc --help
```

or refer to *Generalized Linear Model Classification Options* (page 229) and *Generalized Linear Model Regression Options* (page 236) in the Command Reference appendix.

Training and Testing a Model

The following example trains a GLM Regression model using the gaussian-identity Family/Link Function pair.

```
# skytree-server          glmr                \
  --training_in           train.st            \
  --training_targets_in   train.targets       \
  --model_out             glmr.model          \
  --testing_in            test.st             \
  --targets_out           test.pred.targets \
  --family                gaussian            \
  --link                  identity            \
  --l1_penalty            0.01                \
  --l2_penalty            0                   \
  --epsilon               0.001               \
  --max_iterations        1000
```

The model is then loaded and tested.

```
# skytree-server  glmr          \
  --model_in      glmr.model    \
  --testing_in    test.st       \
  --targets_out   test.pred.targets
```

Excluding the Bias Term

By default, a bias term is included in the computed model. The bias term adjusts all predictions up or down by a constant amount, i.e. it is the predicted value when all inputs are exactly 0. It can be excluded (forced equal 0) with `--exclude_bias_term`:

```
# skytree-server         glmc                \
  --training_in          train.st            \
  --training_labels_in   train.labels        \
  --model_out            glmc.model          \
  --testing_in           test.st             \
  --labels_out           test.pred.labels    \
  --link                 cloglog             \
  --l1_penalty           0.01                \
  --l2_penalty           0                   \
  --epsilon              0.001               \
  --max_iterations       1000                \
  --exclude_bias_term
```

In GLM, the inverse of the link function is applied to obtain the final predictions (targets in `glmr` and probabilities in `glmc`). Excluding the bias term can negatively impact measured model quality, but it is appropriate when a given linear model should definitely predict 0 when all inputs are 0 (before the application of the inverse of the link function to obtain the final predictions). You may also want to exclude the bias/intercept term to choose a simpler model when appropriate and move away from overfitting.

Note: This option cannot be used with the following family/link combinations: poisson/identity, poisson/sqrt, and gamma/inverse.

Tuning an Unbiased Poisson-Log Model

The following example tunes an unbiased GLMR model using the poisson-log Family/Link Function pair.

```
# skytree-server           glmr              \
  --training_in            train.st          \
  --training_targets_in    train.targets     \
  --num_folds              5                 \
  --family                 poisson           \
  --link                   log               \
  --exclude_bias_term      on                \
  --l1_penalty             0,0.001,0.01,0.1  \
  --l2_penalty             0,0.001,0.01,0.1  \
  --tuning_results_out     glmr.tuning.json  \
  --tuning_results_format  json
```

Tuning for Yield Scoring

Just as with `gbt` and `rdf`, the `glmc` method allows you to include a `--yield_values_in` option, which specifies the file for calculating yield values during tuning. This file must be the same length as either the training or tuning vector, depending on whether a tuning table or holdouts are used.

Note: The `--yield_values_in` option is not available in `glmr`.

```
# skytree-server           glmc                          \
  --training_in            yield.data.st                 \
  --training_labels_in     yield.data.labels             \
  --num_folds              2                             \
  --model_out              model                         \
  --yield_values_in        yield.data.values             \
  --testing_objective      yield                         \
  --link                   logit                         \
  --l1_penalty             0,0.01,0.1,1                  \
  --l1_penalty             0,0.01,0.1,1                  \
  --tuning_results_out     glmc.yield.tuning.results.json \
  --tuning_results_format  json                          \
  --log                 run.log
```

When specifying the optional `--testing_objective=yield`, the yield value will be used to determine the best model. A `--yield_values_in` file is required if `--testing_objective=yield` is specified; otherwise it is optional.

The logs show the best tuning results for Gini and Yield as well as the parameters used to reproduce these models.

Using the `--yield_values_in` option, the results can be scored as follows:

```
# skytree-server         score              \
  --yield_values_in      yield.data.values  \
  --predicted_labels_in  labels.tuning
```

Smart Search

In the case where users might not know the best configuration options to specify in order to train and tune a model, or in the case where users want to repeat an experiment, GLM allows for hyper-parameter optimization (smart search).

With a series of `--smart_search` options, users need only specify the training set and model validation options (tuning dataset, holdout ratio, number of folds, etc.), and GLM will automatically search for the best parameter settings without any additional required input from the user. When scoring, smart search will tune over the `--testing_objective` while reporting additional scores.

The example below specifies to try 1000 different parameter settings in a GLM Regression model.

```
# skytree-server            glmr                 \
  --training_in             train.st             \
  --training_targets_in     train.targets        \
  --holdout_ratio           0.3                  \
  --num_folds               10                   \
  --smart_search                                 \
  --smart_search_iterations 1000
```

Note: The above example performs smart search with a default testing objective of `mean_absolute_error`. The `--testing_objective` option guides the sequence of parameters used during smart search and should be explicitly stated if the desired objective is not the default.

In some cases, users may want to input values for certain parameters and let Smart Search tune over the rest. Users can provide open or closed intervals for any tuning parameter using <min>:<max> or <min>: or :<max>. Specific values at regular intervals can also be specified with <min>:<step>:<max>. For example:

- `--l1_penalty=0:1` can be used to specify an interval.
- `--l1_penalty=0.1:` specifies a lower bound.
- `--l1_penalty=0.1:0.01:0.001` can be used to specify exact values to use during smart search.

Tunable intervals or exact values that are explicitly set will be passed to `skytree-server`. All other parameters will be sampled during smart search.

Random Permutation Variable Importance

In ensemble methods, it can become difficult to ascertain the importance of a given independent variable in the predictive model especially relative to other independent variables. One method proposed in Random Decision Forest literature is to use random permutations of variables [breiman2001rf].

After each tree is constructed, the values of a single variable in the out-of-sample data are randomly permuted and then this data is tested against the corresponding tree. This is done for all the dimensions and the results aggregated over all trees. Finally, we observe the difference in accuracy from the non-permuted version versus the permuted version for each dimension separately and associate that as the measure of importance for that variable. The higher the difference the more important that variable is. These values are then scaled so that the most important variable gets an importance of 100 [hastie2009elem]. This is achieved by dividing the raw importance values for all the dimensions by the maximum value and then multiplying by 100. Only dimensions which are completely unused in the model will have an importance of 0.

A Simple Example

A basic version of this analysis requires a saved model and an out-of-sample dataset along with its labels (for classification) or targets (for regression) and is illustrated below:

```
# skytree-server       rdf                 \
  --training_in        income.train.st     \
  --training_labels_in income.train.labels \
  --num_trees          10                  \
  --model_out          model
```

```
# skytree-server              whatif                   \
  --model_in                  model                    \
  --testing_in                income.data.st           \
  --variable_importances_out variable_importances.out \
  --testing_labels_in         income.data.labels
```

In the above example the model is applied on the test data and then the data is scrambled one dimension at a time to observe the effect on accuracy. The variable importances will be written to `variable_importances.out`. This output file includes a single column. The first row is the importances of the first column variable in the .st file, the second row is the importance of the second column, and so on. If the model were a regression model such as GBTR or RDFR, then `--testing_labels_in` would be replaced with `--testing_targets_in`.

Multiple Runs

Each dimension in the out-of-sample dataset is scrambled only once. To reduce variance in results from different scrambling permutations the experiment can be automatically run many times and importances averaged over these runs. For large datasets there will be less variance in the results from different runs, and any number of iterations greater than 10 is unnecessary.

```
# skytree-server              whatif                    \
  --model_in                  model                     \
  --testing_in                income.data.st            \
  --variable_importances_out variable_importances.out  \
  --testing_labels_in         income.data.labels        \
  --num_trials                5
```

Partial Dependencies

For some prediction methods, users can specify the `--partial_dependencies_out` option, which creates a file to store the partial dependencies of the relevant features and/or feature pairs in a JSON format. These pairs can then be used to generate partial dependence plots.

The `--partial_dependencies_out` option is available in `automodel`, `gbt`, `gbtr`, `glmc`, `glmr`, and `svm` (linear) when:

- Training along with `--model_out` or `--testing_in`
- Tuning followed by training with `--model_out` or `--testing_in`

The output prints the following for each variable:

- The feature name
- A marker indicating whether the feature is real or categorical
    - For categorical features, the output maps the internal representation of the category value to the actual category name
- The N value-partial dependency pairs

The data displays as follows in the JSON output. Note that `[type, ...]` denotes an array of values of the specified type:

```
[
 {
  "variableNames": ["string", ...],
  "variableValuesReal": [[float, ...], ...],
  "variableValuesCategorical": [["string", ...], ...],
  "variableValuesCategoricalProportion": [[float, ...], ...],
  "dependenceValues": [[float, ...], ...]
 },
 ...
]
```

Chapter 6 Recommendation

The Skytree Server Recommendation method allows you to make high quality recommendations based on scalable state-of-the-art machine learning algorithms.

Examples of use cases:

- Recommend items based on users' rating histories
- Recommend items based on users' purchase histories
- Compute recommendations for new users

The Recommendation method provides the following recommendation algorithm:

- Item-Based Collaborative Filtering (page 139)

Item-Based Collaborative Filtering

Item-based Collaborative Filtering (CF) is a machine learning algorithm that makes predictions about user preferences based on the opinions of other users. The method first computes a similarity matrix consisting of similarity measures between pairs of candidate items. Then, using the computed similarities and the set of items a target user has viewed, bought, or rated, a prediction is made for each candidate item.

The `cf` module can be applied to recommendation problems involving explicit user feedback, e.g., ratings, for items they have seen or used. The module may also be applied to implicit feedback (unary) problems, where ratings are not available but users' visit or purchase histories have been recorded.

The intuition behind item-based CF is that a user would be interested in buying items that are similar to items that the user has previously liked (or bought) and would like to avoid items that are similar to items that the user has not liked. Item-based CF can be used to either make predictions for a given user-item pair or to recommend items to a user.

Let s_{ij} denote the similarity between two items i and j and let $R_{u,i}$ denote the rating (or implicit feedback) given by user u for item i in the ratings (or unary) matrix R with each row corresponding to a user and each column corresponding to an item. Let us assume that all missing entries in the ratings matrix correspond to zero. For any item i, the set of similar items N_i is used to make rating predictions. Let N_i^u denote the set of items similar to item i that have been rated by user u.

For the case of explicit feedback (ratings), the predicted rating p_{u,i} for a user-item pair (u,i) is given by the weighted sum of the ratings of user u for items similar to item i as shown by the following representation.

$$p_{u,i} = \frac{\sum_{j \in N_i^u} s_{ij} \cdot R_{u,j}}{\sum_{j \in N_i^u} |s_{ij}|}$$

Let I_u be the set of items purchased (or rated) by the user u. To make r new recommendations to the user, a candidate set C of items is generated by combining all the similar items for each item $j \in I_u$ and discarding any item already in I_u as shown by the following expression

$$C = \bigcup_{j \in I_u} N_j \backslash I_u$$

Now, the items in the candidate set are sorted according to their predicted ratings, and the top r items are chosen as recommendations for the user u.

For the unary case, each item c (elementof) C is instead scored by summing its similarities to all the items in I_u as $f_c = \Sigma_{j \in I_u} \hat{s}_{cj}$ where $\hat{s}_{cj} = s_{cj}$ if c is among the similar items to j ($c \in N_j$) or zero otherwise. Then the items in the candidate set C are sorted with respect to f_c in a non-increasing order, and the top r items are chosen as recommendations for the user u.

There are multiple ways of computing the similarity between two items. If $\vec{\mathbf{i}}$ and $\vec{\mathbf{j}}$ denote the columns in the ratings matrix R, then the **cosine-similarity** is given by

$$s_{ij} = \frac{\| \vec{\mathbf{i}} . \vec{\mathbf{j}} \|}{\|\vec{\mathbf{i}}\|_2 \, \|\vec{\mathbf{j}}\|_2}$$

where $\|\vec{\mathbf{i}}\|_2 = \sqrt{\Sigma_u R_{u,i}^2}$ denotes the Euclidean norm of the vector $\vec{\mathbf{i}}$.

Different users tend to have different rating scales, but this is not taken into account in cosine-similarity. The **adjusted cosine-similarity** alleviates this problem by offsetting each user's ratings by the average rating of the user.

Let $U_{i,j}$ be the set of users that have rated both items i and j, and let $\overline{R}_u$ be the average rating of user u. Then the adjusted cosine-similarity is given by

$$s_{ij} = \frac{\Sigma_{u \in U_{i,j}} (R_{u,i} - \overline{R}_u)\,(R_{u,j} - \overline{R}_u)}{\sqrt{\Sigma_{u \in U_{i,j}} (R_{u,i} - \overline{R}_u)^2}\,\sqrt{\Sigma_{u \in U_{i,j}} (R_{u,j} - \overline{R}_u)^2}}$$

A slight modification to the adjusted cosine-similarity using norms of the item vectors is given by

$$s_{ij} = \frac{\Sigma_{u \in U_{i,j}} (R_{u,i} - \overline{R}_u)\,(R_{u,j} - \overline{R}_u)}{\|\vec{\mathbf{i}}\|_2 \, \|\vec{\mathbf{j}}\|_2}$$

Another notation of similarity between two items is the **Pearson-r correlation**. Let $\overline{R}_i$ and $\overline{R}_j$ be the average rating of the i^{th} and j^{th} item respectively, and $U_{i,j}$ again denote the set of users who have rated both items i and j. Then the Pearson correlation is given by

$$s_{ij} = \frac{\Sigma_{u \in U_{i,j}} (R_{u,i} - \overline{R}_i)\,(R_{u,j} - \overline{R}_j)}{\sqrt{\Sigma_{u \in U_{i,j}} (R_{u,i} - \overline{R}_i)^2}\,\sqrt{\Sigma_{u \in U_{i,j}} (R_{u,j} - \overline{R}_j)^2}}$$

A slight modification to the Pearson correlation using norms of the item vectors is given by

$$s_{ij} = \frac{\Sigma_{u \in U_{i,j}} (R_{u,i} - \bar{R}_i)(R_{u,j} - \bar{R}_j)}{\| \vec{\mathbf{i}} \|_2 \| \vec{\mathbf{j}} \|_2}$$

While user-item ratings are quite common scenarios, there are various situations where the feedback is more implicit (for example, in terms of number of visits or number of times bought). We refer to this as the unary setting. This setting provides an opportunity to define similarities between items in terms of co-occurrences or conditional probabilities.

The co-occurrence of a pair of items (visited or bought by the same user) across many users might imply that these two items are quite similar. Similarly, the conditional probability that the user will buy item j given that the user has already bought i can potentially define a similarity of item j to i.

Let U_{ij} be the set of all users who have implicitly provided feedback for both items i and j, and let U_i be the set of all users who have provided implicit (unary) feedback for only item i. Then the **co-occurrence** based similarity function is given by

$$s_{i,j} = | U_{i,j} |$$

and the **conditional-probability** based similarity function is given by

$$s_{ij} = \frac{|U_{i,j}|}{|U_i| \cdot |U_j|^{\alpha}}$$

where $|A|$ denotes the cardinality of any set A, and $\alpha \in [0, 1]$ is a damping parameter that can be optimized over for better performance. Note that while the co-occurrence based similarity function is symmetric, the notion of similarity based on the conditional-probability leads to asymmetric relations.

The **Jaccard** similarity function is given by

$$s_{ij} = \frac{| U_{i,j} |}{|U_i| + |U_j| - |U_{i,j}|}$$

Skytree Server has the aforementioned similarity functions as well as various enhancement to these similarity functions for better performance. The similarity function can be specified by the user or can be tuned over for best possible performance. Refer to *Similarity Functions* (page 147) for further details.

To see all available options for the `cf` module, run:

```
# skytree-server cf --help
```

or refer to *Collaborative Filtering Options* (page 215) in the Command Reference appendix.

Introductory Examples

The examples that follow focus on recommendations using datasets provided with the documentation.

Ratings-Based Recommendation

We begin by recommending 10 items for each user and writing the recommendations to the file `recommendations`. The columns included in the output file are `user_id,item_id,rating`.

```
# skytree-server         cf                \
  --training_in          user_item.st      \
  --training_ratings_in user_item.ratings \
  --num_recommendations 10                \
  --recommendations_out recommendations   \
  --format_out           csv
```

In the above command the `user_item.st` file contains pairs of user and items IDs. A user may have rated multiple items and would then appear once per rated item. The `user_item.ratings` file gives the ratings corresponding to the user-item pairs of `user_item.st`.

If recommendations are required for only a subset of users, the `--users_in` option can be used to specify a list of user IDs. If predicted ratings are desired for specific user-item pairs, the `--testing_in` option should be used.

Note: Items rated by the user are ignored when computing recommendations and are not included in the `--recommendations_out` file, but this can be overridden by the `--recurrent` option. (See *Recommendations for New Users* (page 143).)"

The measure used in the computation of the similarity matrix can be specified via the `--similarity_function` option. The matrix can then be written to file by specifying `--similarity_out`. Note that this command does not specify `--format_out` for this run. In this case, the default Mahout output format will be used.

```
# skytree-server         cf                \
  --training_in          user_item.st      \
  --training_ratings_in user_item.ratings \
  --similarity_function cosine            \
  --num_recommendations 10                \
  --recommendations_out recommendations   \
  --similarity_out       similarity_matrix
```

See the *Similarity Functions* (page 147) section that follows for further information regarding the available similarity functions.

Unary Recommendation

If ratings are not available, for example, if only user purchase histories are available, the `--unary` flag should be specified:

```
# skytree-server         cf              \
  --training_in          user_item.st    \
  --unary                                \
  --k_neighbors          20              \
  --num_recommendations 10              \
  --recommendations_out recommendations
```

Here, we've also specified the `--k_neighbors` option. In this case, only items within "k nearest neighbors" of the rated items are considered as recommendation candidates.

Note: The `--mean_impute` option cannot be specified when `--unary` is on. In this case, `--mean_impute` will automatically be turned off, and `--popularity_impute` will be turned on. Conversely, if `--popularity_impute` is turned on with non-unary data, that option will be turned off, and `--mean_impute` will be turned on instead. (Refer to *Mean Imputation* (page 148) and *Popularity Imputation* (page 149) for more information.)

Saving and Loading a Model

Models can be saved, in binary format, and reloaded using the `--model_out` and `--model_in` options, respectively:

```
# skytree-server         cf                 \
  --training_in          user_item.st       \
  --training_ratings_in  user_item.ratings  \
  --similarity_function  cosine             \
  --model_out            model.cf
```

```
# skytree-server         cf                 \
  --model_in             model.cf           \
  --num_recommendations  10                 \
  --recommendations_out  recommendations
```

Recommendations for New Users

You can also compute recommendations for new users (i.e, users who are not included in the training set). This is done using both `--testing_in` and `--testing_ratings_in`, with a data format that is analogous to the one used for training the model. This recommendation also requires that `--recommendations_out` is specified. The example below trains a model (model.test) and then produces recommendations.

```
# skytree-server          cf                        \
  --similarity_function   adjusted_cosine_norm      \
  --training_in           training_ratings.st       \
  --training_ratings_in   training_ratings.ratings  \
  --k_neighbors           0                         \
  --max_sims_per_item     0                         \
  --min_ratings_per_user  0                         \
  --threshold             -0.1                      \
  --compression           on                        \
  --mean_impute           off                       \
  --similarity_out        similarity                \
  --model_out             model.test                \
  --log                   train.log
```

```
# skytree-server          cf                        \
  --model_in              model.test                \
  --testing_in            testing_ratings.st        \
  --testing_ratings_in    testing_ratings.ratings   \
  --recommendations_out   recommendations           \
  --log                   test.log                  \
  --num_recommendations   10                        \
  --k_neighbors           0                         \
  --max_sims_per_item     0                         \
  --min_ratings_per_user 0                        \
  --threshold            -0.1
```

By default, items that were already recommended to and rated by a user are not included in `--ratings_out` file. However, it might make sense to recommend those items to a user more than once. The `--recurrent` option specifies whether previously rated items in the training set can be recommended again to a given user.

```
# skytree-server         cf                     \
  --similarity_function  adjusted_cosine_norm   \
  --training_in          user_item.st           \
  --training_ratings_in  user_item.ratings      \
  --testing_in           user_item.st           \
  --k_neighbors          0                      \
  --max_sims_per_item    0                      \
  --min_ratings_per_user 0                      \
  --threshold            -0.1                   \
  --compression          on                     \
  --mean_impute          off                    \
  --similarity_out       similarity             \
  --model_out            model.test             \
  --log                  run.log                \
  --ratings_out          ratings.test           \
  --recurrent            on
```

Tuning the Model

Use any of the following methods for tuning the model. Refer to *Controlling Accuracy, Speed, and Memory Usage* (page 147) for descriptions of the parameters to be tuned.

- Tuning Using a Holdout Set (page 144)
- Tuning Using K-Fold Cross Validation (page 145)
- Tuning and Recommending (page 145)
- Content-Based Filtering (page 145)

Tuning Using a Holdout Set

To tune the model parameters, one can explicitly specify a tuning dataset via the `--tuning_in` and `--tuning_ratings_in` options. Alternately, a portion of the training data can be automatically held out by providing a `--holdout_ratio`:

```
# skytree-server        cf                \
  --training_in         user_item.st      \
  --training_ratings_in user_item.ratings \
  --k_neighbors         1:1:20            \
  --holdout_ratio       0.2
```

The command above will hold out 20% of the training data for use in evaluating the model.

Upon completion, the model parameters giving the ten best scoring metrics are reported. For each of the best models (one per metric) all parameters are reported as part of an option string that can be used to reproduce the models. All tuning results can be saved to a file using the `--tuning_results_out` option. Users can specify whether the format of this output file is JSON or CSV using the `--tuning_results_format` option. By default, the tuning results output format is CSV.

Tuning Using K-Fold Cross Validation

Tuning can also be done using K-fold cross validation. This is better than using a single hold out set which is randomly generated as above. The downside is that it can be approximately `K` times slower. One strategy is to hone in on approximate parameters using a holdout set and then fine tune the parameters using K-fold cross validation. In the above we replace `--holdout_ratio` with `--num_folds`. The parameters with the best average results over all folds are returned:

```
# skytree-server         cf                 \
  --training_in          user_item.st       \
  --training_ratings_in  user_item.ratings  \
  --k_neighbors          1:1:20             \
  --num_folds            5
```

If both `--num_folds` and `--holdout_ratio` are provided then the algorithm will not do K-fold cross validation but instead repeat `K` times with a new holdout tuning set and return the parameters with the best average results over these `K` runs.

Tuning and Recommending

A testing dataset can be specified via the `--testing_in` option together with a tuning set (either user-supplied or held out). The program will first tune for the best parameters as above and use the model giving the *best mean absolute error* for recommendation. If `--model_out` is specified, the best model will be stored to a file as well.

If a holdout dataset was generated, a new model will first be generated using the full training dataset.

The following command runs the entire process:

```
# skytree-server         cf                 \
  --training_in          user_item.st       \
  --training_ratings_in  user_item.ratings  \
  --similarity_function  all                \
  --k_neighbors          1:1:20             \
  --num_recommendations  10                 \
  --holdout_ratio        0.2                \
  --recommendations_out  recommendations
```

Content-Based Filtering

Content-based filtering relies on the item or user features to produce recommendations rather than relying on the collaborative watching/rating history of the items or users. The `--alpha` parameter is used to find a tradeoff between the two components. Specify `--alpha=0` for pure collaborative filtering, or use `--alpha=1` for pure content-based filtering. When performing content-based filtering, the item-feature-based part of the similarity function is set to cosine or adjusted cosine for the case of unary and non-unary item features, respectively, while the collaborative filtering part of the similarity function is still set by the user. (Unary item features are analogous to unary ratings, in the sense that a given unary feature is either present or non-present for an item, without a numerical value attached. Non-unary item features, on the other hand, have assigned values.) `--item_features_in` specifies the file that includes the item features. This file uses the same format as the user item file, but the first column is the index of the feature. `--item_features_value_in` is the file containing the ratings for these item features. This has the same format as the ratings file. The item features are interpreted as unary when the `--item_features_value_in` flag is not present.

The following command runs the entire process:

```
# skytree-server            cf                   \
  --training_in             user_item.st         \
  --training_ratings_in     user_item.ratings    \
  --alpha                   1                    \
  --item_features_in        item_features.st     \
  --item_features_value_in  item_features.value  \
  --recommendations_out     recommendations
```

Objective Functions

The `cf` module by default selects the tuned model with the best mean absolute error to be used for testing and/or file output. Alternatively, you can specify a different method. In addition to mean absolute error, typical methods can include mean squared error, root mean squared error, and L1/L2 relative error. The `cf` module provides additional alternative objective functions that you can use. Based on the following assumptions:

- Let P_i be the set of top k recommended items for user i (i=1,...N)
- Let S_i be the set of
  - the highest rated k items (or less if the user has rated less than k items)

    or

  - all items in the unary case.
- Let R_i be the size of the intersection between S_i and P_i ($R_i=|S_i \cap P_i|$)
- Let T_i bet the minimum between the size of S_i and k ($T_i=\min|S_i|, k$)

then these metrics are defined as follows:

$$\text{Absolute Precision} = \frac{\sum_i R_i}{kN}$$

$$\text{Relative Precision} = 1\,/\,N \sum_i \left(\frac{R_i}{T_i}\right)$$

$$\text{Hit rate} = 1\,/\,N \sum_i (R_i \neq 0)$$

Use the `--objective_function` option to specify an objective function to use during tuning.

```
# skytree-server        cf             \
  --training_in         user_item.st   \
  --recurrent                          \
  --unary                              \
  --max_sims_per_item   0              \
  --objective_function  hit_rate       \
  --k_for_precision     10             \
  --tuning_in           items.test     \
  --tuning_ratings_in   items.ratings
```

Similarity Functions

Several measures are available to determine the similarity between items. The function can also be tuned by specifying multiple comma-separated values or, to try all valid functions, by specifying `--similarity_function=all`. In the example below, the output is specified to be in json format.

```
# skytree-server        cf                   \
  --training_in         user_item.st         \
  --training_ratings_in user_item.ratings    \
  --similarity_function all                  \
  --k_neighbors         1:1:20               \
  --num_recommendations 10                   \
  --holdout_ratio       0.2                  \
  --recommendations_out recommendations.json \
  --format_out          json
```

The available similarity functions for ratings-based recommendation are:

- `cosine`
- `adjusted_cosine` (default for ratings-based and item features)
- `adjusted_cosine_norm`
- `pearson`
- `pearson_norm`
- `co-occurrence`
- `all` (uses all above similarity functions during tuning)

The available similarity functions for unary recommendation are:

- `cosine` (default for ratings-based and item features)
- `conditional_prob`
- `jaccard`
- `co-occurrence`
- `all` (uses all supported similarity functions for unary case during tuning)

Controlling Accuracy, Speed, and Memory Usage

There are several options to control the accuracy, speed and memory usage for the Item-Based Collaborative Filtering module. For simplicity, the verbiage is directed towards the non-unary case, but the principles apply generally.

Recall that for each user for which recommendations are to be made, recommendations are formed by computing predicted ratings for all the items the user hasn't yet rated, and returning the items with the highest predicted ratings.

The predicted ratings for an unrated item are computed by iterating over rated items and adding the product of the rating and the similarity to the to-be-rated item. Exactly which of the previously rated items are considered in this pair-wise sum significantly affects the accuracy and computational requirements. For example, only the most similar items or only those items matching certain other criteria can be included.

Per default, all relevant items are considered and the highest intrinsic accuracy of the method can be obtained after tuning for all similarity functions, thresholds and number of neighbors (and potentially setting some Boolean flags in the unary case). Note that for big datasets, storing all similarities between items rated by at least two users can lead to large memory requirements (lots of non-zeroes in the similarity matrix), and it might make sense to only store the top 1000 similarities per item for example, without affecting accuracies too much.

The sections that follow provide a list of the options that have the biggest impact on numerical and computational performance.

Similarity Threshold

The (tunable) `--threshold` option affects accuracy (not speed or memory usage), by setting a minimal threshold of similarity values to be considered for making rating predictions and recommendations. The default threshold is `0.0`, which means it excludes dissimilar items (negative similarity). To exclude items that are only slightly similar, a threshold of, e.g., `0.1` can be specified.

Number of Neighbors

The number of neighbors to be used when making rating predictions is specified by the (tunable) option `--k_neighbors` and mostly affects accuracy, slightly affects speed and doesn't affect memory usage. As mentioned previously, together with the similarity threshold, the number of neighbors is a fundamental tuning parameter to obtain high accuracy in predicted ratings. The default value of 0 will include all the available similarities between items the user has rated and the item for which a prediction is made.

Maximum Number of Similarities per Item

The maximum number of similarities retained for each item is specified via the (tunable) option `--max_sims_per_item`. This affects accuracy, speed and memory usage. At the default value of 0, all similar items are retained. Restricting the number of non-zeroes in the similarity matrix can deteriorate accuracy, but can significantly reduce memory usage and increase speed.

Minimum Number of Ratings per User

The minimum number of ratings per user for consideration in the similarity matrix computation can be specified via the (tunable) option `--min_ratings_per_user`. This parameter affects accuracy, speed and memory usage indirectly via the size of the resulting similarity matrix. With a default value of `1`, every rated item will be included.

Mean Imputation

When making recommendations, the option `--mean_impute` (default: on) forces the number of recommendations for every user to be the requested number of recommendations even in the absence of enough meaningful data. This can happen, for example, when a user has not rated enough items or if not enough similar items can be found, possibly due to some of the options mentioned in this section. In that case, the mean global rating will be used to predict user-item ratings.

Note that this option is only used as a last resort to fill in missing recommendations. Existing valid recommendations are never replaced, even if they have a lower predicted rating than the global average rating (a personalized rating is generally expected to be better than a global rating).

Mean imputation affects recommendation accuracy (it allows otherwise missing recommendations to exist) and can affect speed and memory usage if the number of requested recommendations is much larger than the number of otherwise available recommendations.

Note: If `--mean_impute` is turned on with unary data, this option will automatically be turned off, and `--popularity_impute` will be turned on instead.

Popularity Imputation

When making recommendations, the option `--popularity_impute` pads the list of recommendations with the most popular items. This option substitutes and overrides `--mean_impute` in the case of unary ratings (`--unary`=on). In this case, popular items will be appended to the recommendations list with a rating of 0.

Note: If `--popularity_impute` is turned on with non-unary data, this option will automatically be turned off, and `--mean_impute` will be turned on instead.

Biased Prediction

The option `--biased_prediction` (default: `off`) corrects for biased baselines and can lead to higher accuracies without affecting runtime or memory usage. For illustration: If a user's ratings are generally a lot lower or higher than the average rating across all users, then this subjective bias is accounted for when computing similarities between the items this user has rated and items rated by other users. The same can be true for items that get low or high relative ratings. Only either user- or item-based biases can be accounted for at once. For `adjusted_cosine` and `adjusted_cosine_norm` similarity functions, the user's mean rating is taken into account when making predictions (and hence recommendations). For `pearson` and `pearson_norm` similarity functions, the item's mean rating is used instead.

SKYTREE®

Chapter 7 Scoring

The scoring methods consist of the following quality metrics:

- **Classification Scoring:**
  - Confusion Matrix (page 152)
  - Gini Index (page 156)
  - Capture Deviation (page 159)
  - Precision/Recall at Top k (page 162)
- **Regression Scoring:**
  - (root) mean absolute/squared error
  - (relative) L1/L2 error
  - Coefficient of Determination (page 165)
  - Normalized Gini (page 166)
- **Ranking Scoring:**
  - Mean Average Precision (MAP) (page 167)
  - Mean Reciprocal Ranking (MRR) (page 167)
  - Normalized Discounted Cumulative Gain (NDCG) (page 167)
- **Weighted Scoring**
- **Recommendation Scoring**

The Prediction methods allows incorporation of class weights in the scoring module for classification. Currently, only binary classification is supported. The *Class Weights* (page 164) section discusses situations where class weights are useful and how they can be used in the Prediction modules. Finally, Yield Scoring is also available for use when, for example, attempting to determine fraud.

As with every command, you can run the following to see the available options:

```
# skytree-server score --help
```

and

```
# skytree-server score-recommendation --help
```

You can also refer to *Scoring Options* (page 321) and *Recommendation Scoring Options* (page 320) in the Command Reference appendix.

Classification Scoring

For classification methods, predicted probabilities and labels can be compared against known results. The Prediction methods can be used to compute the confusion matrix and derived metrics, Gini, and capture deviation.

Confusion Matrix

The confusion matrix is a table to visualize the performance of certain supervised machine learning methods, such as classifiers. Each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class. For binary classifiers, the confusion matrix summarizes the number of correct and false classifications in both directions (i.e., both true and false positives and negatives).

The confusion matrix is also available with a precision recall (PR) curve when the `--pr_curve_out` option is specified. The output includes the precision, recall, probability threshold, true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), and the percentile (number of points vs. total number of points).

A Simple Example

The weighted nearest neighbors classifier method, wnnc, is applied on the SDSS dataset:

```
# skytree-server       wnnc              \
  --k_neighbors        20                \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st      \
  --labels_out         labels.wnnc       \
  --scores_out         scores.wnnc       \
  --probabilities_out  probs.wnnc
```

Given actual labels `sdss.train.labels` for the training points in `sdss.train.st`, this will produce a file, `labels.wnnc`, containing predicted labels for the test points in `sdss.test.st`.

Run the following command to produce the confusion matrix from the two label files:

```
# skytree-server         score             \
  --true_labels_in       sdss.test.labels \
  --predicted_labels_in labels.wnnc
```

The output will contain the a per-class confusion matrix along with an aggregated confusion matrix:

```
Confusion Matrix:
+-----------------------------------------------+
|       Labels          |    Pred -1 |    Pred 1 |
|===============================================|
|              True -1 |      334   |      65   |
|----------------------+------------+-----------|
|             True 1  |       16    |      2045   |
+-----------------------------------------------+
```

```
Aggregate:
+-------------------------------------------------------------+
|    Class         |    Total     |    Right     |    Wrong     |
|=============================================================|
|               -1 |          399 |          334 |           65 |
|------------------+-------------+-------------+-------------|
|                1 |         2061 |         2045 |           16 |
|------------------+-------------+-------------+-------------|
|          Overall |         2460 |         2379 |           81 |
+-------------------------------------------------------------+
```

The confusion matrix has predictions on the diagonals with the off-diagonals representing the mistakes. In the binary case, the diagonal contains the true positives and true negatives, while the off-diagonals represent the false positives and the false negatives.

The aggregated confusion matrix is the 2x2 matrix in the upper right corner. Note that the target class is defined as the class with label 1 and that class labels are sorted numerically from top to bottom. Hence, the first row of the confusion matrix contains the number of true negatives (TN) and false positives (FP), while the second row contains the number of true positives (TP) and false negatives (FN). The confusion matrix is padded on the left and bottom with simple counts of the total number of points for each label and the number of correctly and wrongly classified points.

Precision Recall Curve

Given true labels and probabilities, use the `--pr_curve_out` option to write a summarized precision recall (PR) curve of a desired size to this file. The output also provides information about the confusion matrix. The PR curve size defaults to 1000, which outputs an entry per 0.1%-tile of the test points.

Using the same `wncc` example from the previous section, run the following command to produce the PR curve output.

```
# skytree-server         score                \
  --true_labels_in       sdss.test.labels     \
  --predicted_labels_in  labels.wnnc          \
  --yield_values_in      sdss.test_values     \
  --classweight          3                    \
  --classweight          7                    \
  --log                  accuracy_cw3_7.log   \
  --probabilities_in     probabilities.class1 \
  --precision_recall_out recall3_7            \
  --pr_curve_out         pr_curve_out_3_7     \
  --pr_curve_size        200
```

The output includes the precision, recall, probability threshold, true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), and the percentile (number of points vs. total number of points).

Note: This option is different than `--precision_recall_out`, which writes the precision recall curve for each of the unique predicted probabilities and does not include confusion matrix information.

Classification Metrics

The output will also contain the following statistics:

```
Classification Accuracy: 0.974797
Recall/Sensitivity (True Positive Rate): 0.992237
False Positive Rate: 0.115288
Precision: 0.978001
F-Score: 0.985067
```

- The `Classification Accuracy` is the number of correctly classified points for both classes (`TP`+`TN`) divided by the total number of points.
- The `Recall/Sensitivity` is defined as the true positive rate: `TP` (points correctly labeled 1) divided by the actual number of points of class 1.
- The `False Positive Rate` is the number of `FP` (points mistakenly labeled 1) divided by the number of points not belonging to class 1.
- The `Precision` is defined as the number of `TP` divided by the number of points classified as belonging to class 1 (`TP`+`FP`).
- The `F-Score` is defined as:

$$\frac{2 \cdot \mathtt{Prescision} \cdot \mathtt{Recall}}{\mathtt{Precision} + \mathtt{Recall}}$$

Optimizing Classification Labels

The `score` module can be used to generate labels when only probabilities are provided. The labels can be generated by specifying a `--probability_threshold`, which can be used to separate the positive and negative classes:

```
# skytree-server           score              \
  --true_labels_in         sdss.test.labels   \
  --probabilities_in       probs.wnnc.class1  \
  --labels_out             labels.wnnc.0.6    \
  --probability_threshold  0.6
```

A new set of labels has been written to `labels.wnnc.0.6`, and the associated confusion matrix and classification metrics have now changed:

```
Confusion Matrix
----------------
Class     Total  Right  Wrong
-1        399    357    42
1         2061   2043   18
Overall   2460   2400   60
```

```
Classification Accuracy: 0.97561
Recall/Sensitivity (True Positive Rate): 0.991266
False Positive Rate: 0.105263
Precision: 0.979856
F-Score: 0.985528
```

A threshold can also be generated automatically to give optimal accuracy or F-Score. To do so, specify either `--classification_objective=accuracy` or `--classification_objective=fscore`:

```
# skytree-server              score             \
  --true_labels_in            sdss.test.labels  \
  --probabilities_in          probs.wnnc.class1 \
  --labels_out                labels.wnnc.0.7   \
  --classification_objective accuracy
```

which gives

```
...
Optimizing classification using objective 'accuracy'
Optimal probability threshold : 0.757778
...
```

```
Confusion Matrix
----------------
Class   Total  Right Wrong
-1      399    377   22
1       2061   2040  21
Overall 2460   2417  43
```

```
Classification Accuracy: 0.98252
Recall/Sensitivity (True Positive Rate): 0.989811
False Positive Rate: 0.0551378
Precision: 0.989331
F-Score: 0.989571
```

Integration Into Other Methods

When the Prediction methods are combined with other Skytree Server methods, then powerful combinations of supervised machine learning algorithms and scoring are possible. For example, the following command will perform automatic optimization of the parameter k (the number of neighbors used for scoring) for highest F-score:

```
# skytree-server              wnnc              \
  --k_neighbors               10:2:30           \
  --training_in               sdss.train.st     \
  --training_labels_in        sdss.train.labels \
  --classification_objective fscore
```

The output will show the confusion matrix obtained for the optimal parameter:

```
=============================================================
Model with the best F-Score
=============================================================
Gini: 0.970118
```

```
Confusion Matrix
----------------
Class     Total Right Wrong
-1        409   379   30
1         2110  2096  14
Overall   2519  2475  44
```

```
Classification Accuracy: 0.982533
Recall/Sensitivity (True Positive Rate): 0.993365
False Positive Rate: 0.0733496
```

```
Precision: 0.985889
F-Score: 0.989613
```

Note that the F-Score is higher than what was achieved in the previous example.

The confusion matrix also accounts for (and indicates) class weights (typically the same class weights are also used for classification):

```
# skytree-server        score            \
  --true_labels_in      sdss.test.labels  \
  --predicted_labels_in labels.wnnc      \
  --classweight         1                \
  --classweight         5
```

```
Confusion Matrix
----------------
Class   Total  Right  Wrong
-1 (*1) 399    353    46
1 (*5)  10305  10225  80
Overall 10704  10578  126
```

```
Classification Accuracy: 0.987948
Recall/Sensitivity (True Positive Rate): 0.992237
False Positive Rate: 0.122807
Precision: 0.995231
F-Score: 0.993731
```

Gini Index

The Gini index is a well-established method to quantify the inequality among values of a frequency distribution, and can be used to measure the quality of a binary classifier. A Gini index of zero expresses perfect equality (or a totally useless classifier), while a Gini index of one expresses maximal inequality (or a perfect classifier).

The Gini index is defined based on either the receiver operating characteristic (ROC) curve or the Lorenz curve. The ROC curve plots the true positive rate (y-axis) as a function of the false positive rate (x-axis), while the Lorenz curve plots the true positive rate (y-axis) as a function of percentiles of the population (x-axis).

Consider a set of points sorted by the predicted scores or probabilities (from high to low) as predicted by the classifier. Starting at (0,0), for the ROC curve, each classification of a point either leads to the curve going up (if the classification was correct) or to the right (if the classification was wrong). A random classifier will draw a diagonal line (in expectation), while a perfect classifier will go straight up from (0,0) to (0,1). For the Lorenz curve, a correct classification will lead to a step up and right, while a wrong classification will lead to a step to the right. A random classifier will still draw a diagonal, but a perfect classifier will go from (0,0) to (*p*,1), where *p* is the probability for a point being in class 1.

The ROC or Lorenz curve represents a collective of models represented by the classifier. The location on the curve is given by the probability threshold of a particular model (i.e., lower probability thresholds for classification typically lead to more true positives, but also to more false positives).

The Gini index itself is independent of the model, and only depends on the ROC or Lorenz curve determined by the distribution of the scores (or probabilities) obtained from the classifier.

To compute the Gini index, you must compute the area under the curve (AUC) and compare it against the area under the curve for a worst possible “diagonal” classifier (AUD), and the area under the curve for the perfect classifier (AUP):

$$\text{Gini} := \frac{\text{AUC} - \text{AUD}}{\text{AUP} - \text{AUD}}$$

Skytree Server features several ways to compute these areas. The default option is exact calculation based on point-by-point integration along the involved curves.

A Simple Example

The examples will use the datasets distributed along with the documentation.

The weighted nearest neighbors classifier, wnnc, method is applied on the SDSS dataset:

```
# skytree-server       wnnc              \
  --k_neighbors        20                \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st      \
  --scores_out         scores.wnnc       \
  --probabilities_out  probs.wnnc
```

Given actual labels `sdss.train.labels` for the training points in `sdss.train.st`, this will produce files containing predicted scores, `scores.wnnc.class-1` and `scores.wnnc.class1` and probabilities `probs.wnnc.class-1` and `probs.wnnc.class1`.

The Gini index can be computed from the scores:

```
# skytree-server   score                \
  --true_labels_in sdss.test.labels     \
  --scores_in      scores.wnnc.class1
```

Similarly, the Gini index can also be computed based on probabilities:

```
# skytree-server     score              \
  --true_labels_in   sdss.test.labels   \
  --probabilities_in probs.wnnc.class1
```

Gini Index Calculation Details

To see more details from the computation of the Gini index, turn on the verbose loglevel with `--loglevel=verbose` option:

```
# skytree-server   score              \
  --true_labels_in sdss.test.labels   \
  --scores_in      scores.wnnc.class1 \
  --loglevel       verbose
```

Now, the printout contains the numerical values of the three areas needed to compute the Gini index:

```
Parameters used for Gini calculation:
  Integration rule: mid
  Number of integration points: 2460
  Area under the ROC curve: AUC=0.987075
  Area under the ROC curve of the perfect classifier: AUP=1
  Area under the diagonal: AUD=0.5
```

```
Gini index: Gini=(AUC-AUD)/(AUP-AUD)=0.974149
```

```
Gini index: 0.974149
```

As mentioned above, the Gini index can be computed based on either the ROC or the Lorenz curve. The default is the ROC curve (`--curve=roc`), but the Lorenz curve can be selected with `--curve=lorenz`:

```
# skytree-server   score               \
  --true_labels_in sdss.test.labels    \
  --scores_in      scores.wnnc.class1  \
  --curve          lorenz              \
  --loglevel       verbose
```

Note that the Gini index is identical for both curves, as the ratio of the areas remains the same under affine transforms.

If the option `--curve_out` is specified, a file is emitted that contains the ROC or Lorenz curve data points for further review or post-processing:

```
# skytree-server   score               \
  --true_labels_in sdss.test.labels    \
  --scores_in      scores.wnnc.class1  \
  --curve          lorenz              \
  --curve_out      curve.st            \
  --loglevel       verbose
```

For this case, the file `curve.st` contains a set of two-dimensional points that represent the Lorenz curve for the classifier defined by the scores and labels given by `scores.wnnc.class1` and `sdss.test.labels`.

Note, again, that other viable classifiers are defined by the scores for the non-target class or by the predicted probabilities (for either class). Those can be passed in as the argument for `--probabilities_in` as well, and they will define the sort order for the dataset. For nearest neighbor classification (nnc), the resulting Gini index will be the same for scores and probabilities, but for a weighted nearest neighbor classifier (wnnc), the results can differ. Refer to the *Advanced Nearest Neighbor Methods* (page 52) section for more information.

Advanced Options

Skytree Server presents a set of advanced options to control how the Gini index is computed.

The set of true labels sorted by the values in the file given by `--probabilities_in` defines the ROC/Lorenz curve. The Gini index depends on the area under that curve (AUC) and the curves defined by the perfect (AUP) and the random (AUD) classifiers (and on how accurately they are computed). Per default, the exact Gini index is computed based on the entire dataset.

For the Lorenz curve, there is an option (`--percentiles_in`) to specify a file containing a set of sample points for the area calculations. Percentiles are given by values between 0 and 100, as seen in the example below.

The area under of the curve given by those sample points is computed as follows. Consider the straight curve given by the two points $(x1, y1)$ and $(x2, y2)$, and let A be the area under that curve above $y = 0$.

If `--percentiles_in` is omitted, exact integration is performed by sampling the curves as finely as possible (i.e., at every data point), and by using the trapezoidal rule which yields $A = (x2 - x1) * (y2 + y1) / 2$, the exact value.

If percentiles are given via the `--percentiles_in` option, then $A_{high} = (x2 - x1) * (y2)$ (rectangular rule using the high point) is used for each section.

Hence, specifying `--percentiles_in=percentiles` for a file containing decile-wise sampling positions created by the command-line (see below)

```
# echo "10 20 30 40 50 60 70 80 90 100" > percentiles
```

will lead to the same results as taking a sum over the decile-wise samples taken from the Lorenz curve, and then dividing by ten to obtain the approximate area under the curves.

Note that sampling based on `--percentiles_in` will typically lead to approximate Gini index values that differ from the exact values obtained without using `--percentiles_in`.

Also note that the shape of the ROC/Lorenz curves will depend on the class weights, since they determine how often each point is counted. Hence, the Gini index will depend on the class weights if percentiles are specified, as the sampling approximations will have different effects. If `--loglevel=verbose` is specified, the percentile-based number of observations and capture rates are printed. Use the following command:

```
# skytree-server   score                \
  --true_labels_in sdss.test.labels     \
  --scores_in      scores.wnnc.class1   \
  --percentiles_in percentiles          \
  --curve          lorenz               \
  --classweight    1                    \
  --classweight    1                    \
  --loglevel       verbose
```

Had the input data been stratified, and the class weights for each class were larger than 1, then the total number of observations would go up. You can observe this with the following command:

```
# skytree-server   score                \
  --true_labels_in sdss.test.labels     \
  --scores_in      scores.wnnc.class1   \
  --percentiles_in percentiles          \
  --curve          lorenz               \
  --classweight    3                    \
  --classweight    5                    \
  --loglevel       verbose
```

Again, class weights will not affect the Gini index, unless `--percentiles_in` is used (and is more granular than sampling at every single data point).

Of course, the class weights used for scoring should be identical to the class weights used for obtaining the probabilities.

Capture Deviation

Skytree Server features the computation of the capture deviation, a conceptually simple but powerful method to quantify the accuracy of predicted probabilities for binary classifiers. It is defined as a normalized sum of piece-wise absolute deviations between the predicted probabilities and the actual probabilities:

$$\text{capture deviation} := \frac{\sum_i |\text{prob}_\text{i}^\text{actual} - \text{prob}_\text{i}^\text{predicted}|}{\sum_i \text{prob}_\text{i}^\text{actual}}$$

The best possible capture deviation value is 0. The actual and predicted probabilities are obtained by counting the actual fraction of points in class 1, and by taking the average over the predicted probabilities to be in class 1:

$$\text{prob}_\text{i}^\text{actual} = \frac{1}{|\Sigma_i|} \sum_{q \in \Sigma_i} \delta(\text{label}(q) == 1)$$

$$\text{prob}_\text{i}^\text{predicted} = \frac{1}{|\Sigma_i|} \sum_{q \in \Sigma_i} \text{prob}_1(q)$$

Note that the sums are taken over a selected number of quantiles Σ_i, and after sorting the points q by the predicted scores or probabilities (whichever file is used as input for `--scores_in`) (highest to lowest).

Skytree Server allows the specification of the number of partitions of the dataset. The default value is 10, leading to ten deciles as the used quantiles for summation.

A Simple Example

For this example, the nearest neighbors classifier method, nnc, is applied on the SDSS dataset:

```
# skytree-server       nnc                 \
  --k_neighbors        20                  \
  --training_in        sdss.train.st       \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st        \
  --scores_out         scores.nnc          \
  --probabilities_out  probs.nnc
```

Given actual labels `sdss.train.labels` for the reference points in `sdss.train.st`, this will produce files containing predicted scores `scores.nnc.class-1`, `scores.nnc.class1` and predicted probabilities `probs.nnc.class-1`, `probs.nnc.class1` for each class..

Now run the following command to compute the capture deviation for the obtained results:

```
# skytree-server     score               \
  --true_labels_in   sdss.test.labels    \
  --scores_in        scores.nnc.class1   \
  --probabilities_in probs.nnc.class1
```

Note that unless the computation of the capture deviation is explicitly disabled with `--capture_deviation=off`, the Prediction methods will automatically determine whether or not the capture deviation can be computed from the given input files.

The output will contain the following low capture deviation score, indicating good prediction performance:

```
Capture deviation: 0.0118874
```

To see more details about the computation, use `--loglevel=verbose`:

```
# skytree-server     score              \
  --true_labels_in   sdss.test.labels   \
  --scores_in        scores.nnc.class1  \
  --probabilities_in probs.nnc.class1   \
  --loglevel         verbose
```

```
Computing capture deviation based on 10 quantiles:
  For top 0.00% to 10.00%: number of observations: 246,         \
    predicted capture rate: 1, actual capture rate: 0.987805
  For top 10.00% to 20.00%: number of observations: 246,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 20.00% to 30.00%: number of observations: 246,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 30.00% to 40.00%: number of observations: 246,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 40.00% to 50.00%: number of observations: 246,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 50.00% to 60.00%: number of observations: 246,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 60.00% to 70.00%: number of observations: 246,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 70.00% to 80.00%: number of observations: 246,        \
    predicted capture rate: 0.992276, actual capture rate: 0.96748
  For top 80.00% to 90.00%: number of observations: 246,        \
    predicted capture rate: 0.469106, actual capture rate: 0.414634
  For top 90.00% to 100.00%: number of observations: 246,       \
    predicted capture rate: 0, actual capture rate: 0.00813008
Mean predicted capture rate: 0.846138, mean actual capture rate: 0.837805
Capture deviation: 0.0118874
```

To use a different number of (equidistant) quantiles, the value can be specified with `--capture_deviation_quantiles`. For example:

```
# skytree-server                 score             \
  --true_labels_in               sdss.test.labels  \
  --scores_in                    scores.nnc.class1 \
  --probabilities_in             probs.nnc.class1  \
  --capture_deviation_quantiles 4                  \
  --loglevel                     verbose
```

```
Computing capture deviation based on 4 quantiles:
  For top 0.00% to 25.00%: number of observations: 615,         \
    predicted capture rate: 1, actual capture rate: 0.995122
  For top 25.00% to 50.00%: number of observations: 615,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 50.00% to 75.00%: number of observations: 615,        \
    predicted capture rate: 1, actual capture rate: 1
  For top 75.00% to 100.00%: number of observations: 615,       \
    predicted capture rate: 0.384553, actual capture rate: 0.356098
Mean predicted capture rate: 0.846138, mean actual capture rate: 0.837805
Capture deviation: 0.00994663
```

Similar to the percentile-based Gini index calculation, the quantile-based capture deviation can also account for class weights (assuming stratified training data):

```
# skytree-server                  score           \
  --true_labels_in                sdss.test.labels  \
  --scores_in                     scores.nnc.class1 \
  --probabilities_in              probs.nnc.class1  \
  --capture_deviation_quantiles 4                 \
  --classweight                   1                 \
  --classweight                   5                 \
  --loglevel                      verbose
```

```
Computing capture deviation based on 4 quantiles:
  For top 0.00% to 25.00%: number of observations: 2678,        \
    predicted capture rate: 1, actual capture rate: 0.99888
  For top 25.00% to 50.00%: number of observations: 2675,       \
    predicted capture rate: 1, actual capture rate: 1
  For top 50.00% to 75.00%: number of observations: 2675,       \
    predicted capture rate: 1, actual capture rate: 1
  For top 75.00% to 100.00%: number of observations: 2676,      \
    predicted capture rate: 0.820889, actual capture rate: 0.852018
Mean predicted capture rate: 0.955222, mean actual capture rate: 0.962724
Capture deviation: 0.00837436
```

Precision/Recall at Top k

For a user specified k (specified via `--k_for_precision_recall`), Skytree Server allows the computation of the precision and recall at the top k for a set of test points with respect to their probabilities of being in the target class. The precision at top k is defined as

$$\text{Precision@top } k := \frac{\text{number of true positives in the top } k \text{ probabilities}}{k}$$

and the top recall at top k is defined as

$$\text{Recall@top } k := \frac{\text{number of true positives in the top } k \text{ probabilities}}{\text{total number of positives in the test set}}$$

The precision and recall at the top k depends on the size of the test set and the number of positives in the test set relative to k. For example, if k is almost as large as the test set, the recall at top k will be very close to 1 regardless of the classifier used. Similarly, if k is chosen to be much higher than the total number of positives in the test set, then the precision at top k will always be very small irrespective of the classifier used.

For this reason, Skytree Server allows the specification of k relative to the size of test set via `--k_for_precision_recall`=K. The input value K should lie between 0 and 1 and implies k = K x n_t where n_t is the test set size. This is also known as precision and recall at the top 100K-th percentile.

It might be the case that all the probabilities are not unique and the kth highest probability might be equal to the $(k + 1)^{th}$, $(k + 2)^{th}$, ..., $k'^{\,th}$ highest probabilities. In this case, we compute precision/recall at the top k'. For test points with equal probabilities, if one point with a particular probability p is included in the top k, then all other points with the same probability p should also be included in the top k. In this case, the numerator and denominator in the computation of the precision (and recall) at the top percentile is appropriately adjusted.

A Simple Example

The examples use the datasets distributed along with the documentation.

The weighted nearest neighbors classifier method, wnnc, is applied on the SDSS dataset:

```
# skytree-server       wnnc              \
  --k_neighbors        20                \
  --training_in        sdss.train.st     \
  --training_labels_in sdss.train.labels \
  --testing_in         sdss.test.st      \
  --probabilities_out  probs.wnnc
```

Given actual labels `sdss.train.labels` for the training points in `sdss.train.st`, this will produce files containing the probabilities `probs.wnnc.class-1` and `probs.wnnc.class1`.

The precision and recall at top *k* can be computed from the probabilities:

```
# skytree-server     score             \
  --true_labels_in   sdss.test.labels   \
  --probabilities_in probs.wnnc.class1
```

If `--k_for_precision_recall` is not specified, it defaults to 0.1, computing precision and recall at the top 10-th percentile. The output contains the computed values of the precision and recall at the top 10-th percentile

```
Precision at top 10-th percentile: 0.987805
Recall at top 10-th percentile: 0.117904
```

The `--k_for_precision_recall` can be specified as follows for any desired *k*:

```
# skytree-server           score             \
  --true_labels_in         sdss.test.labels   \
  --probabilities_in       probs.wnnc.class1  \
  --k_for_precision_recall 0.3
```

to get the following output:

```
Precision at top 30-th percentile: 0.995935
Recall at top 30-th percentile: 0.356623
```

The precision and recall at top k can also be computed based on the scores as well. The precision/recall at top k depends on the ordering of the test points. This ordering is usually computed with respect to the probabilities of the test points to be in the target class. However, if `--scores_in` is provided instead of `--probabilities_in`, the Prediction methods utilize the scores of the test points to define an ordering. The scores for the test points (contained in some file `scores.wnnc.class`) can be used as following:

```
# skytree-server   score             \
  --true_labels_in sdss.test.labels  \
  --scores_in      scores.wnnc.class1
```

Using the `--precision_recall_out` option, you can specify an output file to store the precision/recall information.

```
# skytree-server          score                   \
  --true_labels_in        sdss.test.labels        \
  --probabilities_in      probs.wnnc.class1       \
  --precision_recall_out  precision.recall.out.wnnc
```

The output file includes three columns. The first column shows the precision, the second shows recall, and the third shows the probability threshold at which the values occur.

Class Weights

The training sets are often formed by stratified sampling of the classes. (It is common for the non-target to be downsampled in binary classification.) In this case, the data distribution of the training set is different from the original data distribution. Hence, the probabilities assigned to test points of being in the target class do not correspond to the true data distribution. This also affects the scoring of the classification performed by a model (with respect to the various classification metrics).

Skytree Server allows the computation of the scores with respect to the original data distribution with the `--classweight` option. This option corresponds to the weights of the class and must be repeated for each class in `--training_labels_in`, and its values are used to artificially inflate the impact of the corresponding class (to counter the effects of the stratified sampling).

For example, suppose that the `SDSS` dataset is used, and the non-target class is down-sampled to a tenth of its original size to create the training set `sdss.train.st`. Then the nearest-neighbor classifier can be used as follows with the modified class weights:

```
# skytree-server       nnc                \
  --k_neighbors        20                 \
  --training_in        sdss.train.st      \
  --training_labels_in sdss.train.labels  \
  --testing_in         sdss.test.st       \
  --probabilities_out  probabilities.nnc  \
  --labels_out         labels.nnc         \
  --classweight        10                 \
  --classweight        1
```

Given actual labels `sdss.train.labels` for the training points in `sdss.train.st`, this will produce files containing predicted probabilities `probs.nnc.class-1` and `probs.nnc.class1` and labels `labels.nnc`.

The default values of `--classweight` in the scoring module is 1 for every class. Since class weights are assigned during the training of the model, they should also be assigned during the scoring. The scoring should be done as follows:

```
# skytree-server        score                    \
  --true_labels_in      sdss.test.labels         \
  --probabilities_in    probabilities.nnc.class1 \
  --predicted_labels_in labels.nnc               \
  --classweight         10                       \
  --classweight         1
```

Regression Scoring

For regression methods, predicted regression targets can be compared against testing targets, and typical error metrics such as mean absolute error, mean squared error, root mean squared error, and L1/L2 error are meaningful quantities to consider. Skytree Server calculates all of the above, in addition to the *Coefficient of Determination* (page 165) and *Normalized Gini* (page 166).

A Simple Example

```
# skytree-server         rdfr                  \
  --num_trees            20                    \
  --training_in          kddcup.train.st       \
  --training_targets_in kddcup.train.targets  \
  --testing_in           kddcup.test.st        \
  --targets_out          targets.rdfr
```

Given actual regression targets `kddcup.train.targets` for the training points in `kddcup.train.st`, this will produce a file, `targets.rdfr`, containing predicted labels for the test points in `kddcup.test.st`. Refer to *Ensemble Learning Methods* (page 59) for additional details.

Run the following command to produce the regression error metrics from the two target files:

```
# skytree-server          score                 \
  --true_targets_in       kddcup.test.targets  \
  --predicted_targets_in targets.rdfr
```

The output will contain the following metrics:

- Mean absolute error
- Mean squared error
- Root mean squared error
- L1 error
- L2 error
- Relative L1 error
- Relative L2 error
- Coefficient of determination
- Normalized Gini

Most of them are well known, but we explain the coefficient of determination and normalized Gini in the next sections.

Coefficient of Determination

The coefficient of determination is given by

$$1-\frac{\sum_i (y_i - f_i)^2}{\sum_i (y_i - y_{\text{mean}})^2}$$

where y_i and f_i are the i-th true and predicted targets, respectively and

$$y_{\text{mean}} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

For further information, consult this Web page (http://en.wikipedia.org/wiki/Coefficient_of_determination).

Normalized Gini

For prediction targets P_i, and true targets t_i:

Stable sort permutation $\gamma_p(i)$

such that

$$P_{\gamma_p(i)} > P_{\gamma_p(j)} \text{ if } \gamma_p(i) < \gamma_p(j)$$

and stable sort γ_t

such that

$$t_{\gamma_t(i)} > t_{\gamma_t(j)} \text{ if } \gamma_t(i) < \gamma_t(j)$$

Given the above, normalized Gini is defined as

$$G = \frac{\left(\frac{1}{\tilde{T}} \sum_{i=1}^{n} \sum_{j=1}^{i} t_{\gamma_p(j)}\right) - \tilde{N}}{\left(\frac{1}{\tilde{T}} \sum_{i=1}^{n} \sum_{j=1}^{i} t_{\gamma_t(j)}\right) - \tilde{N}}$$

where

$$\tilde{T} = \sum_{i=1}^{n} t_i$$

and

$$\tilde{N} = \frac{n+1}{2}$$

Normalized Gini with Negative Values

Normalized Gini is ill-defined for datasets with any negative response values. If this scenario is detected, the normalized Gini is calculated with all response values shifted by $-t_{min}$, where t_{min} is the minimum response value. This creates a non-negative set of response values.

Ranking Scoring

For ranking, recommendation, or other Information Retrieval methods, predicted scores can be compared against known results. Skytree Server can be used to compute metrics such as Mean Average Precision, Normalized Discounted Cumulative Gain, Mean Reciprocal Rank, etc.

Some metrics reported are only applicable to problems where the known relevance of the item is binary, i.e., items are marked relevant or not relevant, and there is no varying degree of relevance. Metrics such as Mean Average Precision, Mean Reciprocal Rank, and Precision@K fall into this category. Other metrics such as Normalized Discounted Cumulative Gain are designed to be able to score items that have varying degrees of relevance. For example, the relevance score may belong to the set:

```
0,2,6
```

where 0 would represent irrelevant items, and items with relevance 6 are three times more relevant than items with relevance 2.

Mean Reciprocal Ranking (MRR)

The reciprocal rank of a single query or recommendation is the multiplicative inverse of the rank of the first correct answer. The MRR is simply the mean of such reciprocal ranks across a set of queries.

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_{\text{i}}}$$

Mean Average Precision (MAP)

Precision and recall are single-value metrics based on the whole list of items returned by the system. (Refer to *Precision/Recall at Top k* (page 162).) For systems that return a ranked sequence of items, it is desirable to also consider the order in which the returned items are presented.

By computing a precision and recall at every position in the ranked sequence of items, you can plot a precision-recall curve, plotting precision $p(r)$ as a function of recall r. Average precision computes the average value of $p(r)$ over the interval from r=0 to r=1:

$$\text{AveP} = \int_0^1 p(r)\, dr$$

This is the area under the precision-recall curve. This integral is in practice replaced with a finite sum over every position in the ranked sequence of items. The formula to compute the finite sum is:

$$\text{AveP} = \frac{\sum_{k=1}^{n}(P(k) \times \text{rel}(k))}{\text{number of relevant items}}$$

where k is the rank in the sequence of retrieved items, $P(k)$ is the precision at cut-off k in the list, $\text{rel}(k)$ is an indicator function equaling 1 if the item at rank k is a relevant item; zero otherwise.

Normalized Discounted Cumulative Gain (NDCG)

The premise of DCG, which extends to non-binary relevance scores, is that highly relevant items occurring higher up in the list, i.e. given higher scores by the system should be rewarded logarithmically versus having them lower in the list.

The DCG is then defined as:

$$\text{DCG}_{\text{p}} = \sum_{i=1}^{p}\frac{2^{rel_i}-1}{\log_2(i+1)}$$

Search result lists vary in length depending on the query, so the cumulative gain at each position for a chosen value of p should be normalized across queries. This is done by sorting items of a result list by relevance, producing the maximum possible DCG until position p (also called Ideal DCG (IDCG) until that position). For a query, the normalized discounted cumulative gain, or NDCG, is computed as:

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p}$$

For a set of queries, we take the mean of these different NDCG values to get the average NDCG of the system.

A Simple Example

The ranking functionality in the `gbt` module is applied on the ranking dataset:

```
# skytree-server       gbt                          \
  --training_in        ranking.train.st             \
  --training_labels_in ranking.train.binary.targets \
  --num_trees          10                           \
  --probabilities_out  probabilities                \
  --testing_in         ranking.test.st              \
  --loss_function      ndcg
```

Given actual relevance scores in `ranking.train.binary.targets` for the training points in `ranking.train.st`, this will produce a file, `probabilities`, containing predicted scores for the test points in `ranking.test.st`.

Refer to the *Ensemble Learning Methods* (page 59) section for additional details.

To get the various metrics for the output scores we can run the following:

```
# skytree-server   score-recommendation        \
  --true_labels_in ranking.test.binary.targets \
  --scores_in      probabilities               \
  --group_ids_in   ranking.test.groupids
```

A Non-Binary Example

First, we train a model with non-binary relevance labels.

```
# skytree-server       gbt                   \
  --training_in        ranking.train.st      \
  --training_labels_in ranking.train.targets \
  --num_trees          10                    \
  --probabilities_out  probabilities         \
  --testing_in         ranking.test.st       \
  --loss_function      ndcg
```

Then score the results:

```
# skytree-server   score-recommendation \
  --true_labels_in ranking.test.targets \
  --scores_in      probabilities        \
  --group_ids_in   ranking.test.groupids
```

In this case, Skytree Server automatically notices the non-binary nature of true relevance scores in the file `ranking.test.targets`, which are in the set

```
0,1,5
```

and thus does not compute binary metrics such as MAP and MRR.

Weighted Scoring

There are many situations where the test points themselves have some weight associated with them. A point with a high weight relative to the other points in the test set implies that a mistake on that point will be penalized more severely than a mistake on the rest of the points. This weight for a point might correspond to how commonly this test point occurs or how much this test point is worth. Skytree Server allows the user to provide this weight information to the scoring module to get a “weighted” score which incorporates the effects of these weights for each of the points.

The `--score_weights_in` option should be used with `skytree-server=score` to provide a file containing the weights for each of the test points, each row in the file containing the weight for the corresponding test point. This option can be used for classification and regression scoring.

Weighted Classification Scoring

In classification scoring, the weighted scoring is performed by replacing the count (or cardinality) of a set of points with the sum of the weights of the points in the set. For example, the (unweighted) classification accuracy is computed as

$$\text{Accuracy} = \frac{\text{Number of test points correctly classified}}{\text{Total number of points in the test set}}$$

In contract to this, the weighted accuracy would be compute as

$$\text{Weighted Accuracy} = \frac{\text{Sum of the weights of test points correctly classified}}{\text{Sum of the weights of all the points in the test set}}$$

In a similar spirit, the weighted confusion matrix is created by replacing the number of true positives with the sum of the weights of the true positives (with corresponding changes for true negatives, false positives and false negatives in binary classification). The computation of the remaining classification scoring metrics (namely Gini, Capture Deviation, Precision/Recall at top *100k*-th percentile, F-Score, and Yield) are similarly modified to compute the weighted score.

A Simple Example

Given files containing the true labels, predicted labels, and predicted probabilities of being in the target class for a test set (the predicted labels and probabilities can be obtained by using any one of the classifiers available in `skytree-server`), the usual classification scoring is done as follows:

```
# skytree-server        score                    \
  --true_labels_in      sdss.test.labels         \
  --probabilities_in    probabilities.nnc.class1 \
  --predicted_labels_in labels.nnc
```

If a file containing the weights of the test points is available, it can be specified via the `--score_weights_in` option to perform weighted classification scoring:

```
# skytree-server         score                  \
  --true_labels_in       sdss.test.labels       \
  --score_weights_in     sdss.test.weights      \
  --probabilities_in     probabilities.nnc.class1 \
  --predicted_labels_in labels.nnc
```

Weighted Regression Scoring

Weighted scoring in regression is performed by taking a weighted sum of the errors made at the individual test points. For example, the ℓ_2 regression error for n test points with actual targets y_i and predicted targets $\hat{y}_i$ is computed as

$$\ell_2 \text{ error} = \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2,$$

and the mean squared error (MSE) is computed as

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2$$

If every test point is associated with a weight w_i, then the weighted ℓ_2 error is computed as

$$\text{Weighted } \ell_2 \text{ error} = \sum_{i=1}^{n} w_i \left(y_i - \hat{y}_i\right)^2,$$

and the weighted MSE is computed as

$$\text{Weighted MSE} = \frac{\sum_{i=1}^{n} w_i \left(y_i - \hat{y}_i\right)^2}{\sum_{i=1}^{n} w_i}$$

Similar modifications are done to compute the weighted ℓ_1 regression error, weighted mean absolute regression error, and the weighted coefficient of determination.

A Simple Example

Given files containing the true targets and the predicted targets, (the predicted targets can be obtained by using any one of the regressors available in `skytree-server`), the usual regression scoring is done as follows:

```
# skytree-server          score                \
  --true_targets_in       kddcup.test.targets  \
  --predicted_targets_in targets.rdfr
```

If a file containing the weights of the test points is available, it can be specified via the `--score_weights_in` option to perform weighted regression scoring:

```
# skytree-server          score                \
  --true_targets_in       kddcup.test.targets  \
  --score_weights_in      kddcup.test.weights  \
  --predicted_targets_in targets.rdfr
```

Recommendation Scoring

The `score` module can be used to score recommendations produced in the `--recommendations_out` file.

Note: Recommendation scoring is only available when the `--recommendations_out` file was produced using `--format_out=csv`. In addition, the users present in that recommendations file should also be present in the test file that has the true user-item pairs (`--true_user_item_in` file).

Non-Unary Example

As indicated previously, the following example recommends 10 items for each user and writes the recommendations to the file `recommendations`.

```
# skytree-server        cf                 \
  --training_in         user_item.st       \
  --training_ratings_in user_item.ratings \
  --num_recommendations 10                 \
  --recommendations_out recommendations    \
  --format_out          csv
```

These recommendations can be scored as follows:

```
# skytree-server        score               \
  --true_ratings_in     user_item.ratings  \
  --true_user_item_in   user.item           \
  --recommendations_in  recommendations
```

Unary Example

The `score` module can also be used to score recommendations when ratings are not available. In this case, the recommendations output file is produced using `--unary=on`.

```
# skytree-server        cf              \
  --training_in         user_item.st     \
  --unary                                \
  --k_neighbors         20               \
  --num_recommendations 10               \
  --recommendations_out recommendations \
  --format_out          csv
```

When scoring unary recommendations, the `--true_ratings_in` option is not specified.

```
# skytree-server        score               \
  --true_user_item_in   user.item           \
  --recommendations_in  recommendations
```

Chapter 8 Distributed Module

The Skytree Server Distributed module enables parallel execution of Skytree Server across a compute cluster, a set of computers connected via a network.

Before using the Distributed module, the compute cluster must be configured as described in the "Distributed Installation" section of the *Skytree Server Installation Guide*. In particular, a shared file system must be available and the user must be able to use ssh between any two computers of the cluster via public/private key authentication (i.e., without typing a password). Please consult your IT administrator for assistance in the cluster configuration.

Note: For execution of Skytree Server on Hadoop clusters, please refer to the *YARN/Hadoop Module* (page 181) chapter.

Running Distributed Jobs

Running a distributed Skytree Server job is very similar to running on a single machine. The two important differences are that:

- The `--hosts` option must be provided, followed by a comma-separated list of the computers (hosts) on which Skytree Server will execute.
- All input and output files must reside on a shared file system.

Note: All input and output files must be accessible on each host via the same path, e.g., `/path/to/shared/file/system/data.st` can be used to access the file `data.st` from all hosts. The shared file system should therefore be mounted with the same path on all hosts. If necessary, please contact your IT administrator for assistance.

In a distributed job, one Skytree Server process will run on each host. Hosts may be specified by any resolvable name, e.g., hostname or IP address. In special cases, the same host may be specified more than once. (See note below.)

The `--procs_per_host` option controls the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.

Note: For performance tuning purposes, you may find it helpful to alter the number of threads and the number of processes. A best practice is for the `[number of processes]` x `[number of hosts]` = `[number of cores on the machine]`. A mixture of processes and threads may outperform using only threads. The optimal number depend on the particular problem you're trying to solve and your hardware.

The `--threads` option in a distributed job controls the number of threads per process rather than the total number of threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified, and hosts `host1` and `host2` have, respectively, 4 and 8 threads available, 4 threads will be used for each process.

Note: If a host is repeated, e.g., `--hosts=host1,host1`, it is recommended that the number of threads per process is reduced to avoid overloading the host. Usually, when running *p* processes on a single host, divide the threads available on the host by *p*.

The `--memory` option can be used with GBT/R, RDF/R, GLM/R, and SVM to specify the amount of memory (MB) available for computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation. When tuning using `--smart-search`, Skytree Server automatically avoids parameter configurations that would require more than the given amount of memory. Because avoiding such parameter configurations can alter the results of tuning, Skytree Server will also indicate the number of configurations it had to ignore after tuning is completed. If `--smart-search` cannot generate parameter configurations that fit within the given amount of memory, Skytree Server will fail with an error message.

In the examples below, we run distributed Skytree Server jobs using two processes on the "local" host, where the command is typed. We address that host using the loopback interfaces `localhost` and `127.0.0.1` for simplicity. (Both point to the same machine.) A shared file system is not required in these artificial examples.

Distributed Skytree Server execution is supported for the following algorithms:

- Nearest Neighbors (page 174)
- Ensemble Methods (page 175)
- Item-Based Collaborative Filtering (page 176)
- k-means (page 176)
- Support Vector Machines (page 177)

Nearest Neighbors

The Distributed module can be used with the nn, `nnplus`, nnc, and wnnc modules. To demonstrate, we run a distributed job by simply appending the `--hosts` option to a *Nearest Neighbors Plus* (page 33) example:

```
# skytree-server  nnplus         \
  --k_neighbors   5              \
  --references_in random_1kx6.st \
  --indices_out   neighbors      \
  --distances_out distances      \
  --hosts         localhost,127.0.0.1
```

Ensemble Methods

This section provides examples of how the Distributed module can be used with the following methods:

- Random Decision Forests Classification and Regression (page 175)
- Gradient Boosted Trees Classification and Regression (page 175)
- Fast Gradient Boosted Trees Classification (page 176)

Random Decision Forests Classification and Regression

The `rdf` and `rdfr` modules may also be run across multiple hosts. We illustrate this using an `rdf` example found in the *Ensemble Learning Methods* (page 59) section:

```
# skytree-server       rdf                 \
  --training_in        income.data.st      \
  --training_labels_in income.data.labels \
  --num_trees          100                 \
  --model_out          model.simple        \
  --hosts              localhost,127.0.0.1
```

Gradient Boosted Trees Classification and Regression

The `gbt` and `gbtr` modules allow various tuning parameter combinations to be run simultaneously, with one parameter combination per host. Note that tuning the `--num_trees` presents a special case that will not be distributed among the hosts.

The `--regularization` option in `gbt` or `gbtr` leads to all processes of the job participating in each model's training, tuning and testing. This option is enabled by default for `gbt` and `gbtr`; it is disabled by default for ensemble `gbt` and `gbtr`.

For ensemble GBT (when `--ensemble_size` is used) each individual model is trained and tested using all hosts. Whether training, tuning or testing, the model uses all the specified hosts to build the model.

We illustrate using a `gbt` example (found in the *Tunable Parameters* (page 89) section), which tunes across 8 different parameter sets (disregarding `--num_trees`). Each of the two hosts specified will run four parameter combinations.

```
# skytree-server       gbt                 \
  --training_in        income.data.st      \
  --training_labels_in income.data.labels \
  --holdout_ratio      0.2                 \
  --num_trees          5:5:25              \
  --tree_depth         2,3                 \
  --learning_rate      0.05:0.05:0.2       \
  --regularization     off                 \
  --hosts              localhost,127.0.0.1
```

GBT and GBTR models, in particular, occasionally perform better when the same host is specified multiple times, either via the `--hosts` option or using the `--procs_per_host` option.

In contrast to the previous GBT example, the following ensemble GBT model uses all hosts to run 40 tuning parameter combinations in sequence. It will also use all hosts in the testing phase.

```
# skytree-server       gbt               \
  --training_in        income.data.st    \
  --training_labels_in income.data.labels \
  --testing_in         income.test.st    \
  --ensemble_size      10                \
  --holdout_ratio      0.2               \
  --num_trees          5:5:25            \
  --tree_depth         2,3               \
  --learning_rate      0.05:0.05:0.2     \
  --probabilities_out  probabilities     \
  --labels_out         labels            \
  --regularization     off               \
  --hosts              localhost,127.0.0.1
```

Fast Gradient Boosted Trees Classification

The `--regularization` option is enabled by default for GBT and GBTR and specifies that each model is trained and tested using all hosts. Whether training, tuning or testing, all specified hosts are used to build each model. Note that the data size may exceed that of a single host's memory; only a subset of the data is loaded into memory by each host.

We illustrate using an example in which two hosts train a single GBT model. Note that the commands are similar with GBTR.

```
# skytree-server        gbt                \
  --training_in         income.data.st     \
  --training_labels_in  income.data.labels \
  --regularization_bins 100                \
  --num_trees           50                 \
  --tree_depth          3                  \
  --learning_rate       0.1                \
  --model_out           model.simple       \
  --hosts               localhost,127.0.0.1
```

Item-Based Collaborative Filtering

Item-Based Collaborative Filtering (page 139) can also be run in distributed mode by specifying the `--hosts` option.

```
# skytree-server          cf                  \
  --training_in           user_item.st        \
  --training_ratings_in   user_item.ratings   \
  --similarity_function   cosine              \
  --num_recommendations   10                  \
  --recommendations_out   recommendations     \
  --similarity_out        similarity_matrix   \
  --hosts                 localhost,127.0.0.1
```

k-means

k-means (page 41) can be used to cluster in distributed mode using the default fast exact heuristic method by specifying the `--hosts` option:

```
# skytree-server    kmeans            \
  --references_in   random_100kx6.st  \
  --centroids_out   centroids.csv     \
  --k_clusters      4                 \
  --memberships_out assignments.csv   \
  --hosts           localhost,127.0.0.1
```

k-means can also be used to cluster in distributed mode using Lloyd's algorithm by setting the `--algorithm=lloyds` option and specifying the `--hosts` option. Lloyd's algorithm for k-means allows the data to remain distributed, allowing datasets larger than the memory of a single machine.

```
# skytree-server    kmeans            \
  --references_in   random_100kx6.st  \
  --centroids_out   centroids.csv     \
  --k_clusters      4                 \
  --algorithm       lloyds            \
  --memberships_out assignments.csv   \
  --hosts           localhost,127.0.0.1
```

Support Vector Machines

Details on the usage of `svm` can be found in *SVM Usage Examples* (page 109). A linear SVM model can be trained and tested on distributed data in the following manner:

```
# skytree-server         svm                      \
  --training_in          sdss.train.st            \
  --training_labels_in   sdss.train.labels        \
  --kernel               linear                   \
  --lambda               0.1                      \
  --testing_in           sdss.test.st             \
  --labels_out           sdss.test.labels.lsvm.0.1 \
  --hosts                localhost,127.0.0.1
```

A linear SVM model can also be trained on distributed data and saved for later prediction in the following manner:

```
# skytree-server         svm                      \
  --training_in          sdss.train.st            \
  --training_labels_in   sdss.train.labels        \
  --kernel               linear                   \
  --lambda               0.1                      \
  --model_out            sdss.lsvm.model.0.1      \
  --hosts                localhost,127.0.0.1
```

A saved linear SVM model can be used for making predictions on distributed data as follows:

```
# skytree-server         svm                      \
  --model_in             sdss.lsvm.model.0.1      \
  --testing_in           sdss.test.st             \
  --labels_out           sdss.test.labels.lsvm.0.1 \
  --hosts                localhost,127.0.0.1
```

The tuning of a linear SVM can also be done with distributed data where all processes will be involved in training a linear SVM model on a single parameter setting at a time. The `--tuning_results_out` option can be used to save the tuning results, and the `--tuning_results_format` option allows you to specify whether the format for the tuning results output file is CSV or JSON. Tuning with a user specified (distributed) `--tuning_in` and `--tuning_labels_in` can be also be accomplished in a similar manner.

```
# skytree-server          svm                          \
  --training_in           sdss.train.st                \
  --training_labels_in    sdss.train.labels            \
  --kernel                linear                       \
  --lambda                0.01,0.1,1,10                \
  --num_folds             10                           \
  --hosts                 localhost,127.0.0.1
```

A nonlinear SVM model can be trained, tuned, and tested on distributed data. Both RBF and polynomial nonlinear kernels are supported. For polynomial kernels, a `--polynomial_degree`, `--polynomial_offset`, and `--polynomial_scale` must be specified. For RBF kernels, an `--rbf_bandwidth` must be specified. This is shown in the following example:

```
# skytree-server          svm                          \
  --training_in           sdss.train.st                \
  --training_labels_in    sdss.train.labels            \
  --kernel                rbf                          \
  --rbf_bandwidth         1.0                          \
  --lambda                0.1                          \
  --testing_in            sdss.test.st                 \
  --labels_out            sdss.test.labels.rsvm.0.1 \
  --hosts                 localhost,127.0.0.1
```

Note: Changing the number of threads in linear SVM (single host or multi-host) or in nonlinear SVM (multi-host only) can change results numerically. This will have only minimal effect on scoring when the models are trained with sufficient accuracy. This behavior occurs only when `--threads` > 1.

Status Module

The status module utility allows simple monitoring of hosts in the compute cluster along with Skytree Server processes in progress. It can be run at any time to query the current status of the cluster. Running

```
# skytree-server status --hosts localhost,127.0.0.1
```

returns information of the following form:

```
System status:
-------------------------------------------------------------------------
|              | Avail. |  Avail. |  System  |   Total   |     Free     |
|   Hostname   | Cores  | Threads | Load (%) | Mem. (GB) |   Mem. (GB)  |
-------------------------------------------------------------------------
|        host1 |     4  |       8 |    98.00 |     16.00 |         8.00 |
|        host1 |     4  |       8 |    98.00 |     16.00 |         8.00 |
-------------------------------------------------------------------------
```

```
Active Skytree Server process(es) on host 'host1':
-------------------------------------------------------------------------
|             |                |    CPU    |   Total   |     Runtime     |
|  ProcessID  |    Username    | Usage (%) | Mem. (GB) | [DD-]HH:MM:SS   |
-------------------------------------------------------------------------
|        1234 |  skytree_user |     95.50 |      8.00 |          01:01 |
-------------------------------------------------------------------------
```

```
Active Skytree Server process(es) on host 'host1':
-------------------------------------------------------------------------
|             |                |    CPU     |   Total    |     Runtime    |
|  ProcessID  |    Username    | Usage (%) | Mem. (GB) | [DD-]HH:MM:SS  |
-------------------------------------------------------------------------
|        1234 |   skytree_user |     95.50 |      8.00 |          01:01 |
-------------------------------------------------------------------------
```

If the `--hosts` option is not provided, the list of hosts found in the installation's hosts configuration file is used. (Refer to the *Skytree Server Installation Guide*.) If the list is empty, then only the local host is used.

SKYTREE®

Chapter 9 YARN/Hadoop Module

The Skytree Server YARN/Hadoop module enables execution of Skytree Server on a YARN or Hadoop cluster. We recommend running Skytree Server in a YARN environment rather than the earlier Hadoop version. Running Skytree Server in a YARN environment includes the following benefits:

- YARN provides better control over resources to applications (jobs) that are running on it.
- Machine-learning algorithms are often too complex to be done in a simple MapReduce paradigm. YARN provides better integrations with workloads that include complex algorithms.
- YARN includes a Resource Manager that focuses on scheduling, making it better able to manage large clusters.

Before using the YARN/Hadoop module, the cluster must be configured as described in the *Skytree Server Installation Guide*.

Data Preparation

On a YARN or Hadoop cluster, the data input files loaded by Skytree Server have the same format as elsewhere. In order to create a file that can be loaded by Skytree Server, run the tools described in *Data Preparation* (page 9), and copy the result to the cluster. Alternatively, if the raw data resides on the cluster, you can run a YARN or Hadoop version of the data preparation tools.

A single data preparation tool for YARN and Hadoop, `etl-hadoop.sh`, performs all the necessary steps. This tool runs a sequence of MapReduce jobs, leveraging the distributed processing capabilities of YARN and Hadoop. Most of the options for the other data preparation tools have a corresponding option in this tool.

The `-input_dir` option specifies an input directory. All the files in that directory, except those whose names begin with an underscore (_), become inputs to the tool.

Multiple input directories can be specified. When doing so, an `-input_dir` option must be included for each input directory. (A comma-separated list is not supported.) If a header line exists in any data file in any of the input directories, then all data files in all input directories must have the same header file. Finally, if `-use_column_weights` is specified, then all input directories must have a "`_weights`" file.

The `-output_dir` option is the directory that stores the output files. When including a single input directory, Skytree Server creates a file `output.st` in that directory. When including multiple input files, the output will produce `.st` files with basename prefixes that mirror the input directories' basenames. For example, an input directory named `data_dir` will output `data_dir.st`. This is different than a single input directory, which would output a file named `output.st`. If input directories have matching basenames, then the output files append a "_*n*" to the basename to differentiate the output files. For example, and if one input directory is `/path/to/data_dir`, and another input directory is `/path/to/other/data_dir`, the output will be `data_dir_1.st` and `data_dir_2.st`.

If used with a single input directory, the option -create_labels_file creates a file output.labels, and the option -create_targets_file creates a file output.targets, also in the output directory. When including multiple input files, the output will produce .labels and .targets using the same naming conventions described for the .st output.

There are some other differences between the YARN-based or Hadoop-based tool and the other data preparation tools. For example, the -min_percentiles and -max_percentiles options for generate-headers.sh are not available in YARN and Hadoop. Those options require loading numeric data into main memory, which may not be feasible for the large datasets stored on the clusters. Also, the -ignore_lines option is replaced by -use_column_names. The reason is that, in YARN and Hadoop, it is not always possible to recognize the start of a file. If that option is used, the first line of a file is used as a list of column names. In that case, all files in the input directory must begin with the same list of column names. As a result, options that take a column argument can use those names instead of numbers. Finally, the -skip_bad_lines and -max_bad_lines_per_mapper options are available in YARN and Hadoop only.

You may need to provide memory size hints to YARN and Hadoop for etl-hadoop.sh to run successfully. After the other arguments to the tool, use --, followed by memory parameters. For moderate-sized datasets, try the following parameters:

```
  -Dmapred.job.reduce.memory.mb=2048          \
  -Dmapred.job.map.memory.mb=2048             \
  -Dmapred.job.reduce.memory.physical.mb=2048 \
  -Dmapred.job.map.memory.physical.mb=2048
```

Below are some examples of how to use the etl-hadoop.sh script (omitting, for space reasons, the memory size hints just mentioned).

Note: When running etl-hadoop.sh, the specified -output_dir must not already exist.

```
# etl-hadoop.sh             \
  -input_dir     data1_dir \
  -input_dir     data2_dir \
  -output_dir    etl_out   \
  -delimiter     COMMA     \
  -horizontalize
```

```
# etl-hadoop.sh                 \
  -input_dir          data_dir \
  -output_dir         etl_out  \
  -categorical_number 5,9,13   \
  -use_column_names
```

```
# etl-hadoop.sh                      \
  -input_dir               data_dir \
  -output_dir              etl_out  \
  -ignore_constant_columns          \
  -missing_value           ?        \
  -mean_impute
```

```
# etl-hadoop.sh                 \
  -input_dir          data_dir \
  -output_dir         etl_out  \
  -ignore_columns     2        \
  -delimiter          ,        \
  -categorical_number 3,4,5    \
  -categorical_text   1        \
  -label_index        1        \
  -create_labels_file          \
  -allow_reordering
```

Certain conditions can result in bad lines: too few or too many columns, unrecognized categorical values (including labels), and unparsable numerical values. The -skip_bad_lines option skips all bad lines encountered during the data conversion process. The -max_bad_lines_per_mapper option specifies an upper limit on the number of bad lines, and if more than the specified number of bad lines are found, then etl-hadoop.sh terminates. Note that this option is most applicable when combined with the -header_input_dir option, in case the new data file contains values or lines not seen in header generation. (Refer to *Reusing Headers* (page 185).) If -max_bad_lines_per_mapper is specified without -skip_bad_lines, then etl-hadoop.sh will attempt to parse bad lines. Unrecognized and/or unparsable values or values that do not fit the existing dictionaries will be treated as missing. If a line has fewer columns than expected, it will be padded with missing values. If a line has more columns than expected, extra columns will be truncated. Lines will be skipped only if they cannot be parsed even with these modifications, such as if a line has errors in its libsvm-formatted section.

```
# etl-hadoop.sh                        \
  -input_dir                data3_dir \
  -header_input_dir         etl_out   \
  -output_dir               etl1_out  \
  -ignore_constant_columns            \
  -missing_value            ?         \
  -mean_impute                        \
  -max_bad_lines_per_mapper 10
```

The etl-hadoop.sh script can also pass the -words_columns flag with an accompanying column specification argument. For each possible word, this parameter creates a notional column in the output .st file. The entry will be 1 if the word occurs in a row, and 0 if it does not. Unrecognized words in words columns do not result in bad lines and are ignored. Because most rows don't contain any particular word, the columns are always made sparse. (That is, there is one entry for each word that occurs; otherwise the column is not represented in the output.) For example:

```
# etl-hadoop.sh              \
  -input_dir       data_dir \
  -output_dir      etl_out  \
  -words_columns   6        \
  -rare_words_pct  10       \
  -stop_words_pct  10
```

The -rare_words_pct and -stop_words_pct filter out rare and common words from the .st file. This filtering does not change the number of notional columns, so multiple runs of etl-hadoop.sh with different filtering percentages will yield output files where the column numbers are identical. Also, note that words that can be parsed as numbers become the word "_NUMBER_".

Note: Hadoop ETL queries the Hadoop runtime for the number of reducers that were used during a MapReduce job. To make that query, the Hadoop runtime is passed the names of a "job counter group" and a "counter name." While those names are unlikely to change across Hadoop releases, if they do, the new names can be passed on the command line with a `-D` flag.

For the counter group, the new name can be passed as the value of

```
com.skytree.tools.etl.hadoop.jobCounterGroup
```

while the new counter name is the value of

```
com.skytree.tools.etl.hadoop.launchedReduces
```

You can use the `help` flag to learn all the options available for `etl-hadoop.sh`.

Note: Users running Cloudera (CDH) 4.x may encounter an error, "Found interface org.apache.hadoop.mapreduce.TaskAttemptContext, but class was expected." If this happens, pass the `-cloudera` flag to `etl-hadoop.sh` before the separator.

Note: Hadoop 2.x users may encounter an, "UnsupportedOperationException: This is supposed to be overridden by subclasses," error. If this happens, pass the -hadoop2 flag to `etl-hadoop.sh` before the separator.

A Note Regarding Data Preparation

For cluster configurations with inconsistent `mapred.reduce.tasks` settings in the cluster nodes' `mapred-site.xml` configuration files, an error of the following form may arise in the second MapReduce job submitted by `etl-hadoop.sh`, named "EtlHadoop second-pass schema inference":

```
[INFO] Task Id : attempt_<attempt ID>, Status : FAILED on node <hostname>
java.lang.Throwable: Child Error
at org.apache.hadoop.mapred.TaskRunner.run(TaskRunner.java:275)
Caused by: java.io.IOException: Task process exit with nonzero status of 1.
at org.apache.hadoop.mapred.TaskRunner.run(TaskRunner.java:262)
[INFO] Job <job ID> failed with state FAILED due to: NA
[INFO] Counters: 7
[INFO]   Job Counters
...
[INFO]     Launched map tasks=4
[INFO]     Data-local map tasks=4
[INFO]     Aggregate execution time of reducers(ms)=0
[INFO]     Failed map tasks=1
[ERROR] Hadoop job 'EtlHadoop second-pass schema inference' failed, exiting ...
```

In addition the first (successful) MapReduce job, "EtlHadoop first-pass schema inference", will report counters of the form:

```
[INFO]   Map-Reduce Framework
...
[INFO]   Reduce input records=<reduce input records>
[INFO]   Reduce input groups=<reduce input records>
...
```

```
[INFO]   Map output records=<map input records>
...
```

where <`reduce input records`> and <`map input records`> are not equal.

If the above error is observed, the following flag should be passed to `etl-hadoop.sh` to ensure consistent `mapred.reduce.tasks` setting for the job:

```
-Dmapred.reduce.tasks=<desired number of reducers>
```

Reusing Headers

The `etl-hadoop` script supports reusing headers that were computed from previous runs with the `-header_input_dir` command. This allows future datasets to be converted using the same output format.

If you want to obtain exactly the same output format as the original dataset, then any command line options that were specified in the original dataset must also be specified when using `-header_input_dir`.

```
# etl-hadoop.sh                        \
  -input_dir               data1_dir \
  -input_dir               data2_dir \
  -output_dir              etl_out   \
  -ignore_constant_columns           \
  -missing_value           ?         \
  -mean_impute
```

```
# etl-hadoop.sh                        \
  -input_dir               data3_dir \
  -header_input_dir        etl_out   \
  -output_dir              etl1_out  \
  -ignore_constant_columns           \
  -missing_value           ?         \
  -mean_impute
```

Running Jobs on a YARN/Hadoop Cluster

Running a Skytree Server job on a YARN or Hadoop cluster is similar to running a distributed job. Review the *Distributed Module* (page 173) section before continuing with this section.

There are several important differences between regular distributed jobs and those run on a YARN or Hadoop cluster:

1. Jobs (also referred to as "applications") are submitted to the YARN Resource Manager via the `skytree-yarn` command. Jobs are submitted to Hadoop JobTracker's scheduler via the hadoop command.

2. An output (*job*) directory on the cluster's distributed file system must be specified (and cannot exist prior to job submission). A log file named `skytree-server.log` will automatically be placed inside that job directory, and the `--log` option is not available.

3. The user specifies the number of processes to run using a set of quorum conditions, not by specifying hostnames with the `--hosts` option.

Note: All input file (and directory) paths should be specified as absolute paths to a user-writable path on the system's distributed file system. For most Hadoop clusters, this is inside the user's home directory in HDFS name space (e.g., `/user/<username>/`). For MapR clusters, the MapR-FS filesystem is typically mounted at `/mapr/`, and the user's distributed home directory is at `/mapr/<ClusterName>/user/<username>/` (via the NFS Gateway service).

Note: Output files can be specified by file name only (without a full path), and will be placed in the job directory.

Note: A Skytree Server job on a Hadoop cluster runs one Skytree Server process in each reducer task of a MapReduce job. If the nodes of the cluster are configured with a single reducer slot per node, each process will be free to use all of its node's compute resources. If the nodes are configured with multiple reducer slots, then users must exercise care to consume only an appropriate share of the available resources.

Submitting Jobs

The process for submitting jobs varies based on whether you are using YARN or Hadoop. Refer to the following sections:

- Submitting YARN Jobs (page 186)
- Submitting Hadoop Jobs (page 187)

Submitting YARN Jobs

YARN jobs (also referred to as "applications") are launched by submitting them to YARN's Resource Manager. The Resource Manager allocates appropriate resources to the YARN job and can be tracked on the resource manager's Web UI.

Use the `skytree-yarn` command to submit a job. Skytree Server ships with its own command, ``skytree-yarn``, which is a convenience wrapper over ``yarn``. This sets up the required environment to launch a Skytree Server YARN job correctly.

Note: The `skytree-yarn` command can only be used when submitting jobs. This command expects that the yarn command is in the user's PATH. If it is not, it can be added as follows:

```
 # export PATH=$PATH:<YARN installation path>/bin
```

A sample for submitting a YARN job is follows:

```
# skytree-yarn          rdf                          \
  --training_in         datasets/income.data.st      \
  --training_labels_in  datasets/income.data.labels  \
  --num_trees           10                           \
  --model_out           model                        \
  --processes           2                            \
  --memory              2048                         \
  --cores               2                            \
  --output              job1
```

or more generally:

```
# skytree-yarn  {All skytree methods + parameters}   \
  --processes   {# of skytree processes}             \
  --memory      {Amount of Memory per process in MB} \
  --cores       {# of cores per process}             \
  --output      {job output directory}
```

YARN Parameters

The YARN parameters include those that are also available with `skytree-server`. (See `skytree-server <method> --help` for more information.) The following additional parameters are available for use with YARN only.

- `--processes` (Required): Any YARN job launched has to negotiate the amount of resources it needs to complete. This value informs YARN how many different Skytree workers or threads need to be launched.
- `--memory`: YARN also needs to know the amount of memory for each worker/thread. The value here has to be carefully considered. For best performance, specify the amount of memory per process as close to the actual needed amount as possible. If too little is specified, the job may be killed. If too much is specified, resources may be wasted, or the job may not be able to start.
- `--cores`: Each worker needs access the to cores to use for computation.
- `--output`: All jobs require an output directory (on HDFS). This directory should not exist before the job is started. By default the `skytree-server.log` file is placed here.
- `--queue`: This allows you to specify that a `skytree-yarn` job must be run on a specific queue. If this is specified, then a valid queue name is required, otherwise the job submission will fail. If --queue is not specified, then the job is submitted to the default queue.
- `--filter-stdout`: A flag that specifies to filter stdout to display only the core execution log. A full log with the name `$script_name.<date>-<time>.log` will be written to the current directory.

Important Notes Regarding Skytree YARN Jobs

- The user's home directory is assumed to be /user/{user.name}.
- All relative input files with relative paths are assumed to be under the user's home directory. So `--training_in datasets/data.st` is assumed to be in `/user/{user.name}/datasets/data.st`.
- If the output path is relative, it is also assumed to be relative from the user's home directory. So `--output=job1` will be in `/user/{user.name}/job1`.
- All relative output files are assumed to be under the output directory. So `--model_out=model --output=job1` is assumed to be `job1/model`, and subsequently be in `/user/{user.home}/job1/model`.
- A job output directory should not exist before launch.
- All absolute paths are left as presented.
- The user has read permission on input files and write permission on output files/directories.

Submitting Hadoop Jobs

All jobs must be submitted to the Hadoop JobTracker for dispatch. The JobTracker determines, based on available compute resources in the cluster, when, and where to launch Skytree Server processes.

Job submission commands are of the form:

```
# hadoop com.skytree.SkytreeServer       \
  -Djob_dir=<output directory>           \
  -Doptions=<Skytree Server options>     \
  -Dquorum=<quorum conditions>           \
  [-Dmem_per_process=<reserved memory per process>]
```

Note: We assume that the hadoop command is in the user's `PATH`. If not, it can be added by adding

```
# export PATH=$PATH:<Hadoop installation path>/bin
```

to the user's `$HOME/.bashrc` file (or similar) or by replacing hadoop with the fully pathed command

```
<Hadoop installation path>/bin/hadoop
```

Note: We assume that `<Skytree Server installation path>/lib/*` is in the `HADOOP_CLASSPATH` environment variable, either via the `hadoop-env.sh` file (see the "MapReduce Configuration" section in the *Skytree Server Installation Guide*) or by setting it in the user's `$HOME/.bashrc` file using

```
# export HADOOP_CLASSPATH=<Skytree Server installation path>/lib/*:$HADOOP_CLASSPATH
```

The following error may be reported if the `HADOOP_CLASSPATH` is not correctly set:

```
Error: Could not find or load main class com.skytree.SkytreeServer
```

Instead of setting the `HADOOP_CLASSPATH` environment variable, the hadoop portion of job submission commands can be replaced with:

```
# hadoop jar <Skytree Server installation path>/lib/skytree-server-hadoop.jar
```

We now examine the options specified in the job submission command:

-Djob_dir=<output directory>

The user must specify a job directory that resides on the shared file system. The directory will be created before the job submission and must not exist prior to submission.

The directory will be used for job coordination and must be available to all nodes running Skytree Server. If the argument to a Skytree Server output file option (one ending in _out) is given without a path, it will be written to this directory. The Skytree Server log file will be written to `<output directory>/skytree-server.log`.

-Doptions=<Skytree Server options>

The `<Skytree Server options>` are the same command line arguments that can be passed to Skytree Server when running without a Hadoop MapReduce environment. For example, you can use the following command to execute an RDF job outside the Hadoop cluster:

```
# skytree-server       rdf                 \
  --training_in        income.train.st     \
  --training_labels_in income.train.labels \
  --testing_in         income.test.st      \
  --probabilities_out  probs               \
  --num_trees          100
```

To submit the same job on a Hadoop cluster, run the following instead:

```
# hadoop com.skytree.SkytreeServer             \
  -Djob_dir=<output directory>                 \
  -Doptions="rdf                               \
      --training_in        income.train.st     \
      --training_labels_in income.train.labels \
      --testing_in         income.test.st      \
      --probabilities_out  probs               \
      --num_trees          100"                \
  -Dquorum=<quorum conditions>
```

Note: The quotation marks surrounding the space-separated options are required.

Note: To run distributed jobs, the `--hosts` option cannot be used. Instead, the desired number of processes is specified by the `-Dquorum` option (see below).

Note: On MapR Hadoop distributions, all input file paths without absolute paths, including those specified within an input file (`--input_file`), are interpreted relative to the directory from which the job submission command is executed. On all other Hadoop distributions, all input file paths must be given with absolute paths.

-Dquorum=<quorum conditions>

The quorum conditions specify the necessary compute resources to be allocated to a job before it begins. More specifically, they give the desired number of processes paired with the acceptable amount of time to wait for the scheduler to allocate that number of processes. The format for the argument is

```
<num processes 1>,<time 1>;<num processes 2>,<time 2>,...
```

and is interpreted as to start the job if:

- `<num processes 1>` are allocated to the job within `<time 1>`;
- `<num processes 2>` are allocated to the job within `<time 2>`;
- ...

Recall that the number of Skytree Server processes corresponds to the number of reducers allocated to the job. The `<time>` arguments must be specified with a unit of `d` (days), `h` (hours), `m` (minutes) or `s` (seconds). E.g., `5m` is interpreted as 5 minutes.

The quorum conditions must be specified with decreasing numbers of processes and increasing time periods. If the final condition is not satisfied, the job is killed.

For example, suppose five processes are desired for a particular job, but only two are currently available. If the job is submitted with `-Dquorum="5,30m"` it will wait 30 minutes to obtain five processes, and then the job will be killed. Instead, it is preferable to specify the quorum condition as `-Dquorum="5,5m;4,10m;3,15m;1,2h"`, which will start the job if

- 5 processes are allocated to the job within 5 minutes; or
- 4 processes are allocated to the job within 10 minutes; or
- 3 processes are allocated to the job within 15 minutes; or
- 1 process is allocated to the job within 2 hours; or
- stop trying after 2 hours.

Warning: If a job is killed due to a failure to find a quorum of processes, it is sometimes difficult to determine that the failed quorum condition was the root cause. In these cases it is useful to check the JobTracker's logs for errors reported by the reducers. For more information on how to check reducer log, refer to *Monitoring Jobs* (page 190).

Note that multiple processes can only be specified for the algorithms supported by the Distributed module. For further information regarding their distributed usage, refer to *Distributed Module* (page 173).

-Dmem_per_process=<reserved memory per process>

This option specifies the reserved amount of virtual memory (in MB) for each Skytree Server process, which determines the number of MapReduce slots needed. Remaining slots on a node may be shared with other processes (same or different job). To reserve 4000 MB per process, for example, specify `-Dmem_per_process=4000`.

Note: This option is required if the Capacity Scheduler is enabled and is disallowed otherwise. For information on how to configure Hadoop schedulers, refer to the "MapReduce Configuration" section in the *Skytree Server Installation Guide*.

Warning: For best performance when running under the Capacity Scheduler, all users should specify the reserved amount of memory per process as close to the actually needed amount as possible. If too little is specified, the job may be killed. If too much is specified, resources may be wasted or the job may not be able to start. We recommend consulting the memory usage reports by Skytree Server and/or the Hadoop JobTracker.

Warning: For best performance when running under the Capacity Scheduler, jobs that reserve only a small amount of available memory may also need to be run with a small number of threads via the `--threads` option within the `-Doptions` argument. Otherwise, high CPU loads may result and impede overall job throughput.

Note: A temporary directory output directory will be created in the user's home directory on the Hadoop distributed storage. The directory name is of the form `SkytreeServer-<random characters>` and will be removed upon job completion. In some circumstances, the directory will not be deleted, and the user must manually delete the directory when it is clear that the job has terminated.

Monitoring Jobs

Monitoring a YARN Job

After a YARN job is submitted, its health can be monitored via the Resource Manager. The Resource Manager UI is global and depends on your YARN cluster configuration.

Identifying Your YARN Application

The YARN application ID is printed out immediately after the job is successfully submitted. Below is an example of the log:

```
2014-04-29 20:34:20,568
  - INFO [pool-6-thread-1:n.s.a.y.a.SkytreeYarnClient$ProgressLogger@445]
  - Skytree Application Launched with Id: application_1398210192725_0042
```

In the above example, the Skytree YARN application ID is `application_1398210192725_0042`.

YARN Logs

Each YARN job also writes the logs to the job's output directory.

Monitoring a Hadoop Job

Once a job is submitted, information regarding the job is written to the screen. The information includes:

JobTracker URL for the job

Important information regarding the execution of the job's processes is written to the logs available at this URL. The URL is given in a message similar to:

```
skytree.SkytreeJobRunner: Job submitted. The tracking URL is <URL>.
```

When viewing the job progress, please note that the progress of mappers is not relevant to the execution of the job (as no work is performed during the map phase). In addition, because Skytree Server processes are not traditional reducers, the progress of reducers is not an indication of progress in the Skytree job. Both progress reports should be ignored.

The syslogs of the reducers contain valuable information regarding the execution of each Skytree Server process. They should be consulted, particularly when a job failure occurs. To see the syslog of a reducer, navigate to the job's tracking URL in your browser. Then click on a number in the `Reduce` row of the first table. Next, click on a task in the `Task ID` column. On the next page, under Task logs, click `All`.

Note: A complete list of jobs managed by the JobTracker can be seen using the JobTracker's Web interface (usually at `http://<JobTracker hostname>:50030`).

Hadoop job ID

JobTracker assigns a unique identifier to each submitted job. The job ID is reported in a message similar to:

```
mapred.JobClient: Running job: <job ID>
```

This job ID can be used to track the job using Hadoop's tools and then kill the job as described in *Killing Jobs* (page 192).

Skytree Server log

Skytree Server always writes a log file when running in a Hadoop environment. For convenience, the log messages are repeated to screen by the job submission process.

Note: Occasionally, and particularly for very short jobs, no Skytree Server output is written to screen. In these cases, please consult the Skytree Server log file instead.

Note: The time stamps reported on screen are those of the report time and may lag the actual execution of the job. More accurate times are reported in the Skytree Server log file.

Hadoop counters

For ensemble methods, collaborative filtering and nearest-neighbors methods, and Hadoop counters, in addition to those set by Hadoop itself, will be incremented during execution. These counters include information regarding the progress of tuning iterations and/or the completion of various tasks in the algorithms. The counters appear in a “Skytree Server” group.

Killing Jobs

Killing a YARN Job

Terminating the job submission process (e.g., via **Ctrl+c**) does not kill a job (or application) that has already been submitted to Resource Manager. You must identify the application ID to kill a submitted job. The syntax for this is as follows:

```
# yarn application -kill <application_id>
```

This ID is printed out when the Skytree YARN application gets access to it via YARN. Below is an example of the log:

```
2014-04-29 20:34:20,568
  - INFO [pool-6-thread-1:n.s.a.y.a.SkytreeYarnClient$ProgressLogger@445]
  - Skytree Application Launched with Id: application_1398210192725_0042

2014-04-29 20:34:20,569
  - INFO [pool-6-thread-1:n.s.a.y.a.SkytreeYarnClient$ProgressLogger@446]
  - Application can be killed using command
       'yarn application -kill application_1398210192725_0042'
```

The application ID used to kill the above referenced job is: `application_1398210192725_0042`. The example below shows how to kill this job:

```
# yarn application -kill application_1398210192725_0042
```

Killing a Hadoop Job

Terminating the job submission process (e.g., via **Ctrl+C**) does not kill a job that has already been submitted to JobTracker. To kill a submitted job, identify the job ID (see *Hadoop job ID* (page 191)), and then issue the following command.

```
# hadoop job -kill <job ID>
```

Using Kerberos-Enabled Clusters

Users must generate forwardable Kerberos ticket-granting tickets (TGTs). These can be generated using "`kinit -f`" (or simply `kinit` if the administrator has configured TGTs to be forwardable by default).

Users can run "`klist -f`" to determine if a TGT is forwardable. An `F` flag in the list of flags indicates that a TGT is forwardable. For example:

```
Ticket cache: FILE:/tmp/krb5cc_1000
Default principal: skytree@SKYTREE
Valid starting     Expires            Service principal
01/01/14 00:00:00  01/02/14 00:00:00  krbtgt/SKYTREE@SKYTREE
renew until 01/08/14 00:00:00, Flags: FRI
```

TGTs will be read only when the job is submitted. Because some processes can be time consuming, the ticket must be renewable throughout the duration of job execution. Users should run "`kinit -f`" before submitting a job to ensure maximum TGT lifetime.

Troubleshooting YARN

Below are common errors that you might encounter when using Skytree Server with YARN.

Timeout Logs

A timeout log looks similar to the following:

```
2014-04-29T20:39:21,245Z WARN  n.s.a.y.a.SkytreeYarnClient [ip-172-31-36-199.us-
west-2.compute.internal] [ServiceDelegate
SkytreeYarnClient$SkytreeEventHandler:launchTimeout(SkytreeYarnClient.java:409) -
Requested 4 containers for runnable workers, only got 0 after 42484 ms
```

If you encounter this log, this means that significant time has passed, and Skytree Server has not been allocated enough resource. By default, the Skytree Server application will wait 'forever' for the resources. If that is not the correct behavior, the property can be changed in using the following configuration:

```
# skytree.application.max.timeout.millis=<num>
```

Skytree Server will automatically terminate after the specified number of milliseconds has passed

Not Enough Memory Allocated

Skytree Server's YARN application relies on the user to reserve the proper amount of memory in order for the job to run correctly. If not enough memory is available to the Skytree user, (i.e. Skytree Server uses more than the reserved memory), the application can be killed by YARN. In this case, a log similar to the following can be seen:

```
2014-04-29T20:34:39,922Z WARN  o.a.t.i.a.RunningContainers [ip-172-31-36-199.us-
west-2.compute.internal] [ServiceDelegate] RunningContainers:handleCompleted
(RunningContainers.java:318) - Container container_1398210192725_0042_01_000003
exited abnormally with state COMPLETE, exit code 143
```

followed by

```
2014-04-29T20:34:41,109Z ERROR n.s.a.y.c.AbstractService [ip-172-31-36-199.us-west-
2.compute.internal] [pool-12-thread-1] AbstractService:error
(AbstractService.java:171) - coordinator 7e4fea24-86af-4139-83cc-63073a763ea7-0:
ip-172-31-36-198.us-west-2.compute.internal:60017 failed to respond. Failing
```

If you encounter this log, we recommend allocating more memory to workers (using the `--mem` option).

Java Heap Space and GC Overhead Limit Exceeded

If you encounter the following errors,

```
Error: Java heap space
Error: GC overhead limit exceeded
```

verify that you included a delimiter in the `etl-hadoop.sh` configuration. If the data is tab delimited, for example, and a `-delimiter=TAB` is not specified, then `etl-hadoop.sh` will consider the data to be a single categorical column and will attempt to construct a dictionary the size of the entire dataset. This leads to memory over consumption.

Chapter 10 Streaming Module

The Skytree Server Streaming module allows Skytree Server to listen on a TCP port for incoming test points and reply with predicted outcomes. Both the test points and replies are given in JSON (JavaScript Object Notation) format. Currently, the `gbt`, `gbtr`, `rdf`, and `rdfr` modules are supported.

Running the Streaming Server

Executing Skytree Server in streaming mode requires that the `--port` option specifies the port on which to listen for test data. In addition, `--model_in` must specify a previously trained model to used to predict outcomes for streamed test points.

A third option, `--batch_size`, may also be provided to denote the number of streamed test points to collect before testing with the model. This aggregation of test points may decrease testing time by using the model simultaneously for multiple test points, however this comes at the expense of increasing the turnaround time for a single point (the latency of the response) as a streamed point will not be tested until enough points are received by the server.

Note: The specified port must be greater than or equal to `1024`.

Note: To reduce memory usage, the model file may be queried from disk during prediction.

Streaming Client

The streaming client can communicate with the server via TCP socket on the port specified to the server (via `--port`). Replies will be returned using the same connection. The client should send the test points for which predictions are sought. The format of these points is described in the following section.

JSON Data Format

All input/output is performed via JSON formatted messages. These formats are described in the sections that follow.

Input Data

The format of each input test point message is:

```
{
  "queries":
  {
    "ids":[int],
    "reals":[number],
    "ints":[int]
  }
}
```

The `ids` will be used in the responses by the server to identify the test points' predictions. The `reals` and `ints` correspond to test data transformed as described in *Data Preparation* (page 9) for the model in use.

Note that multiple points may be sent simultaneously by providing an array of point indices in the `ids` field. In that case, the continuous and integral values of the points will be packed into the `reals` and `ints` fields consecutively. For example, if point 1 has integers `[1,2,3]` and point 2 has reals `[4,5,6]` the `ints` field should be given as `[1,2,3,4,5,6]`.

Output Data

The output JSON object will contain the prediction(s) for each point received by the server. The classifier reply format is:

```
{
  "reply":
  {
    "id":[int],
    "label":[int],
    "probability":[number]
  }
}
```

The `id` corresponds to the ids sent to the server in the input message. The `label` and `probability` fields provide the predicted label for the test point and the probability of the point being in the majority class.

The regressor reply format is:

```
{
  "reply":
  {
    "id":[int],
    "target":[number]
  }
}
```

The `id` once again corresponds to a test points `ids` field while the target provides the predicted value of the quantity of interest.

Example

We begin by first training a GBT model and saving it to `model.stream`:

```
# skytree-server       gbt                 \
  --training_in        income.data.st      \
  --training_labels_in income.data.labels \
  --num_trees          20                 \
  --model_out          model.stream
```

The server can then be started using:

```
# skytree-server  gbt            \
  --model_in      model.stream   \
  --port          10000
```

The `income.test.json` file contains the data of `income.test.st` formatted in JSON as described above. That data can be simply sent to the server using the UNIX `netcat` utility. To do so, in a new terminal, run:

```
# cat income.test.json | nc localhost 10000
```

Replies will be written to the terminal in the form:

```
{"reply":{"id":0,"label":-1,"probability":0.0497022}}
{"reply":{"id":1,"label":-1,"probability":0.320625}}
{"reply":{"id":2,"label":-1,"probability":0.341991}}
{"reply":{"id":3,"label":1,"probability":0.788279}}
{"reply":{"id":4,"label":-1,"probability":0.051463}}
...
```

SKYTREE®

Chapter 11 Advanced Usage

The Advanced Usage section provides best practices for obtaining best results with Skytree Server and is intended for advanced users.

Performance Tuning

Skytree Server is designed to optimally utilize system resources by default. Still, for a given machine learning task, the user has several options that can affect runtime and/or memory usage.

Number of Processes

For most methods in Skytree Server, it is best to launch one process per physical server to fully benefit from multithreading on shared memory. For multi-node configurations, one Skytree Server process per compute node should be launched, such that inter-node communication between processes is minimized.

However, for compute servers with non-uniform memory access (NUMA), it might make sense for certain algorithms to launch one (or more) Skytree Server process(es) per physical multi-core processor to maximize the memory locality (assuming affinity to a memory channel) and hence the modeling throughput. For example, for a dual-socket Xeon server with two 8-core processors, there are 16 physical cores and 32 hyperthreading threads available. In that case, it might make sense to launch two Skytree Server processes per node with each 16 threads.

Conventional distributed job launches 2 processes (one per node):

```
# skytree-server       gbt             \
  --training_in        train.st        \
  --training_labels_in labels          \
  --num_trees          100             \
  --model_out          model           \
  --ensemble_size      100             \
  --hosts              node01,node02   \
  --threads            32
```

NUMA-optimized distributed job launches 4 processes (two per node):

```
# skytree-server       gbt                            \
  --training_in        train.st                       \
  --training_labels_in labels                         \
  --num_trees          100                            \
  --model_out           model                         \
  --ensemble_size       100                           \
  --hosts               node01,node01,node02,node02   \
  --threads             16
```

The runtime for the NUMA-optimized job can be slightly less, depending on the problem and the compute hardware.

Note: When running on Hadoop, the NUMA-optimized mode would require running the Capacity Scheduler and reserving less than (or equal to) half the available memory per process, such that two processes can run per node. Refer to *Data Preparation* (page 181) in the YARN/Hadoop section for more information.

Number of Threads per Process

The number of threads should generally be left at the default value, which is the maximum number of threads the server can support without overloading the system. If other compute-intense processes are running on the same server (e.g., in an interactive execution mode without a scheduling system), the user might want to specify fewer threads for each process. If this is a common problem, then the system administrator can set a global default and/or limit on the number of threads as described in the *Skytree Server Installation Guide*.

Number of Cached Trees for Ensemble Members

For ensemble methods, the `--num_cached_trees` option can be used to control the memory usage. Higher values will lead to higher memory usage by Skytree Server. The default value for the number of cached trees (or ensemble members) is the number of threads available. Reduce this value if memory usage is too high.

Validate Header

The *validate header* step can be used to tweak the header generated by the *generate header* step. Once the changes are made to the original header, a new header can be generated based on the changes the user desires. For example, if you accidentally omit a categorical column header, then changing the header and running `validate-header.sh` may be faster than re-running `generate-header.sh`.

The reason this step might be needed is that the header generation process creates a header assuming that all numeric columns (identified as `integer` or `floating`) are of continuous nature. In other words, numeric columns are not categorical or discrete. This is not always true. For example, sometimes in a database a customer's state of residence may be represented as a numeric value from 1 to 50. The header generation process will assume this column to be continuous, but that assumption is not valid. The header generation step has not created dictionaries for these columns and has labeled them as **not** categorical. This needs to be changed, and the `validate-header.sh` utility can be used to do that.

The Process

Tweaking the header is a 2-step process:

1. Edit original header and change `categorical=false` to `categorical=true` for all columns for which dictionaries are to be generated.
2. Run `validate-header.sh` with appropriate arguments to generate a new header.

A Simple Example

The following example illustrates the steps needed. Consider a header that contains the following:

```
# column number: 1 name: age
integer:categorical=false:min=17.0:max=90.0:...
# column number: 2 name: workclass
text:categorical=true;dictionary=...
# column number: 3 name: finalweight
integer:categorical=false:min=12285.0:max=1490400.0:...
# column number: 4 name: education
text:categorical=true;dictionary=...
# column number: 5 name: education-num
integer:categorical=false:min=1.0:max=16.0:
mean=10.078088530363212:sigma=2.5709464361754146
...
```

and was generated by the following command:

```
# generate-header.sh              \
  -file          income.data    \
  -file          income.test    \
  -header_out    income.header  \
  -ignore_lines  1              \
  -ignore_lines  1              \
  -label_index   15             \
  -missing_value ?
```

Assume that we know that column 5 from the header above is actually categorical and not a continuous value and that we want to generate a dictionary for it. We edit the header to indicate that column 5 is categorical by changing `categorical=false` to `categorical=true`:

```
integer:categorical=true:min=1.0:max=16.0:mean=10.078088530363212:sigma=2.5709464361754146
```

Then we run the `validate-header.sh` script as follows:

```
# validate-header.sh                        \
  -file          income.data              \
  -file          income.test              \
  -header_in     income.header            \
  -header_out    income.header.validated  \
  -ignore_lines  1                        \
  -ignore_lines  1                        \
  -missing_value ?
```

to give

```
integer:categorical=true;dictionary="1","10","11","12","13","14","15",
"16","2","3","4","5","6","7","8","9"
```

Notice the only difference in the call to `validate-header.sh` from the call to `generate-header.sh` is the `-header_in` argument. Its value should be the same as the file name of the header generated by `generate-header.sh`. The updated header will be found in `income.header.validated`.

Important Notes

1. This process will even work for floating-point values, but keep in mind that the dictionaries for floating-point numbers can get very large, and these data types may not be the right choice for categorical variables.
2. The above process can be used for all numeric columns that are categorical with one run of `validate-header.sh`. (i.e., There is no need to do multiple runs for each column.)
3. It is highly recommended that you inspect the newly generated header.

Appendix A Troubleshooting

The troubleshooting section attempts to guide users when troubleshooting Skytree Server Machine Learning jobs. It gives an overview over general tips that can help improve productivity and apply to most (if not all) modules in Skytree Server.

For installation and administration details, please refer to the *Skytree Server Installation Guide*. Contact your system administrator for help installing and configuring Skytree Server.

Contact Skytree support (support@skytree.net) if you encounter problems that you cannot resolve.

System Status and Network Speed

The log produced by Skytree Server (or by the `status` module) should be the first thing to examine when troubleshooting any performance issues.

```
# skytree-server status --hosts node01,node02,node03
```

```
-------------------------------------------------------------------------------------------
|          | Avail. |  Avail.  |  System  |    Total   |     Free   |  Network  |  Network   |
| Hostname | Cores  | Threads | Load (%) | Mem. (GB) | Mem. (GB) | Lat. (us) | B/W (MB/s) |
-------------------------------------------------------------------------------------------
|   node01 |     12 |       24 |     1.12 |     47.15 |     41.82 |    origin |     origin |
|   node02 |     12 |       24 |    85.42 |     47.15 |      3.38 |      8.09 |     115.95 |
|   node03 |     12 |       24 |     5.82 |     47.15 |     42.18 |    128.09 |      12.28 |
-------------------------------------------------------------------------------------------
```

In the example log above, the compute node node02 was loaded at 85% and the free memory was very low, suggesting that another process might using system resources. Also, the network connection between node01 and node03 was slow (high latency and low bandwidth), indicating potential configuration problems.

Loglevels

By default, Skytree Server runs in the `default` loglevel. For more verbosity during troubleshooting, we advise to run with `--loglevel=verbose`.

Data Sanity and Ingestion Speed

By default, many methods in Skytree Server support a fast (unchecked) file read for highest file I/O speed during data ingestion via the `--fast_read` option. This is safe if the dataset was created (and sanitized) with the Skytree Server data preparation tools. We suggest you turn off the fast read option (`--fast_read=off`) to perform a thorough check on datasets not created with the Skytree Server data preparation tools to avoid problems that can be potentially hard to detect (e.g., invalid characters interpreted as numbers).

System Load and Memory Usage Watchdog

By default, Skytree Server monitors system resources with a system load and memory usage watchdog service (`--watchdog=on`). Both a low memory threshold (`--watchdog_low_memory_threshold=0.05`) and a high system load threshold (`--watchdog_high_load_threshold=1.5`) can be specified. The default values are fine for most purposes.

High System Load

If the system load is higher than the system load threshold, the log file can contain warning messages generated by the watchdog service as follows:

```
12:37:42 [INFO] Progress: 10%
12:37:47 [WARNING] System load is high! System Load: 178.6 %
12:37:52 [WARNING] System load is high! System Load: 188.7 %
12:37:52 [INFO] Progress: 20%
12:37:57 [WARNING] System load is high! System Load: 198.9 %
12:38:00 [INFO] Progress: 30%
12:38:02 [WARNING] System load is high! System Load: 208.7 %
12:38:02 [WARNING] Try running ps or top in a terminal session to check
         for other processes competing for resources on this machine.
```

On Hadoop systems, this might be a consequence of running multiple jobs on the same node (possible with the Capacity Scheduler), without reducing the number of threads per process (`--threads`). Consider reducing the number of threads per process. On non-Hadoop systems, consider inspecting the compute node for other compute-intense running processes with `top` (and pressing p to sort by processor usage).

High Memory Usage

If the system's available memory is less than what Skytree Server requires to complete the desired machine learning task, the log file can contain warning messages generated by the watchdog service as follows:

```
13:29:34 [INFO] ===================================================
13:29:34 [INFO] Running tuning parameter combination 1 of 125.
13:29:34 [INFO] ===================================================
13:29:50 [WARNING] Memory is running low! Free Memory: 2.254 GB
13:29:52 [INFO] Building 10000 trees for tuning.
13:29:55 [WARNING] Memory is running low! Free Memory: 1.631 GB
13:30:00 [WARNING] Memory is running low! Free Memory: 0.5969 GB
13:30:05 [WARNING] Memory is running low! Free Memory: 0.5976 GB
terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc
```

The above job was killed by the operating system due to insufficient available system memory.

Controlling Memory Usage without Affecting Results

For ensemble methods, the `--num_cached_trees` option can be used to control the memory usage without affecting results; refer to *Number of Cached Trees for Ensemble Members* (page 200).

For nearest neighbors (the `nnplus` and `wnnc` modules), the `--memory_usage` option can be used to control the (relative) memory usage without affecting results, refer to *Controlling Memory Usage* (page 34). Note that this option must be specified in addition to `-Dmem_per_process` for certain Hadoop environments (and both values should be in accordance with each other). See *Submitting Hadoop Jobs* (page 187) for more details.

For Random Decision Forests (the `rdf` module), memory footprint can be reduced when no sampling is done: `--sampling_with_replacement=off --sampling_ratio=1.0`.

Appendix B Command Reference

This appendix describes the commands that are available for use with Skytree Server based on the current module. At any time, you can type `--help` at the command prompt for a list of options available in the current module. For example:

```
# skytree-server nnplus --help
```

Note: When enabling binary features (i.e., turning a feature on), the keywords on, `yes`, `true`, and `1` all have the same effect. Similarly, `off`, no, `false`, and 0 have the same effect when disabling a feature.

Refer to the following sections for more information:

- Nearest Neighbors Plus Options (page 299)
- Random Decision Forests Options (page 302)
- Random Decision Forests Regression Options (page 312)
- Recommendation Scoring Options (page 320)
- Scoring Options (page 321)
- Singular Value Decomposition Options (page 325)
- Status Options (page 327)
- SVM Options (page 328)
- Time Series Options (page 335)
- Two-Point Correlation Options (page 339)
- Weighted Nearest Neighbors Classification Options (page 340)
- What-If Options (page 349)

AutoModel Options

The following tables show the options available in the `automodel` module.

***Table 3:** AutoModel Input Data Options*

Command	Description

`--classweight`	Specify a value for classweights. When used, you must repeat the `--classweight` option for each class in `--training_labels_in`. The values are used to artificially inflate the impact of the corresponding class. If omitted, classweights of 1 are assumed for every class.
	
	Usage:
	`--training_labels_in  sdss.train.labels \`
	`--classweight  10 \`
	`--classweight  1`

`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in  model`

`--smart_search_restart_in`	Optionally specify to load data from a file to continue smart search from saved data.
	
	Usage:
	`--smart_search_iterations  10 \`
	`--smart_search_restart_in  income.restart`

`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  sdss_test.st`

Table 3: *AutoModel Input Data Options (continued)*

Command	Description
`--training_in`	Required. Specify the file containing the training data. **Usage**: `--training_in  sdss.train.st`
`--training_labels_in`	Required for classification problems. Specify the file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`
`--training_point_weights_in`	Optionally specify a file containing the point weights (for training a model) for each point in `--training_in`. By default, each training point gets a point weight of 1. **Usage**: `--training_point_weights_in  income.weights`
`--training_score_weights_in`	When evaluating a model, optionally specify a file containing the score weights for each point in `--training_in`. By default, each point in `--training_in` gets a score weight of 1. `--training_score_weights_in` will only be used if `--tuning_in` is not provided and you use a holdout from the `--training_in` for tuning. Specify `--tuning_score_weights_in` to tune on `--tuning_in` with score weights. **Usage**: `--training_score_weights_in  income.weights`
`--training_targets_in`	Required for regression problems. Specify the file containing the targets of the training data. **Usage**: `--training_targets_in  kddcup.train.targets`
`--yield_values_in`	Optionally specify the file used to calculate yield values during tuning. This must be the same length as either the training or tuning vector, depending on whether you are using holdouts or a tuning table. Note that if `--testing_objective=yield` is specified, then this option is required, and its value will be used to determine the best model. **Usage**: `--yield_values_in  income.yield.st`

Table 4: *AutoModel Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning **Usage**: `--holdout_ratio  0.2`

***Table 4:** AutoModel Model Validation Options (continued)*

Command	Description
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  10`
`--smart_search_iterations`	Optionally specify the number of search rounds to try for tuning. This value must be greater than 0 and defaults to 100. **Usage**: `--smart_search_iterations  500`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	
	Usage:
	`--tuning_in  sdss.tune.st`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_labels_in`	If `--tuning_in` is specified for a classification problem, then also specify a file containing the labels of the tuning data.
	
	Usage:
	`--tuning_in  sdss.tune.st`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_models_out`	File prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`. Note that this generates a single model file storing all of the trees tuned over rather than generating separate lists. To limit the number of trees when using this model, specify the `--num_trees` option when loading this model.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels \`
	`--tuning_models_out  sdss.tune.models`

`--tuning_results_out`	Optionally specify a file to store all tuning results. The output format is JSON. **Usage**: `--tuning_results_out  income.tune.results`

Table 4: *AutoModel Model Validation Options (continued)*

Command	Description
`--tuning_score_weights_in`	Optionally specify a file containing the score weights for each of the tuning points. By default, each tuning point gets a weight of 1. Use this option with `--tuning_in` to tune with score weights. **Usage**: `--tuning_score_weights_in  income.tune.weights`

`--tuning_targets_in`	Specify an file containing the targets of the tuning data. Required for regression problems if `--tuning_in` is provided.
	Usage:
	`--tuning_in  income.tune.st  \`
	`--tuning_targets_in  income.tune.targets`

Table 5: *AutoModel Output Data Options*

Command	Description
`--labels_out`	Optionally specify a file to store computed labels. **Usage**: `--labels_out  results`
`--model_out`	Specify a file to store the trained model. This is required if you are generating a classification model rather than performing testing. **Usage**: `--model_out  sdss.lsvm.model.0.1`
`--output_with_ids`	If enabled, the per-point output for `--labels_out`, `--probabilities_out`, and `--test_point_variable_importances_out` will be prepended with the input file's "`id`" meta field followed by a comma. For example, if the original output is "`a,b,c`" then the new output would be "`ID,a,b,c`" where "`ID`" is an integer (`1, -1`). If an "`id`" meta field is not available in the input, then "`0`" is set as the input "`id`" field (for example, "`0,a,b,c`"). This option is disabled by default. **Usage**: `--output_with_ids  on`
`--partial_dependencies_out`	Specify the file to store the partial dependencies of the relevant features and/or feature pairs in JSON format. These can be used to generate partial dependence plots. **Usage**: `--partial_dependencies_out  dependencies.json`
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Usage**: `--pmml_out  income.pmml`

Table 5: *AutoModel Output Data Options (continued)*

Command	Description
`--probabilities_out`	Optionally specify a file to store computed probabilities. **Usage**: `--probabilities_out  sdss.test.probabilities`

`--smart_search_restart_out`	Optionally specify to save smart search data to a file. This file can then be specified when running additional smart search iterations.
	Usage:
	`--smart_search_iterations  10 \`
	`--smart_search_restart_out  income.data.restart`

`--targets_out`	Optionally specify a file to store computed targets. Used only with regression problems. **Usage**: `--targets_out  targets.gbtr`

`--variable_importances_out`	Optionally specify a file to store intrinsic variable importances. This output file includes a single column. The first row is the importances of the first column variable in the `--training_in` .st file, the second row is the importance of the second column, and so on. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	Usage:
	`--training_in  income.data.st \`
	`--testing_in  income.test.st \`
	`--variable_importances_out  variable_importances.out`

Table 6: *Non-tunable AutoModel Options*

Command	Description
`--classification_objective`	Optionally specify the objective for classification threshold tuning. Specify either `fscore` (default) or `accuracy`. **Usage**: `--classification_objective  accuracy`
`--compression`	Compression for data. Reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`
`--k_for_precision`	Optionally specify *k* as a value between 0 and 1 for precision at the top 100*k*-th percentile. *k* should be greater than 0 and less than 1. *k* defaults to 0.1. **Usage**: `--k_for_precision  0.2`

Table 6: *Non-tunable AutoModel Options (continued)*

Command	Description
`--limit_parameters`	Restricts the parameter space in order to reduce the training time. This option is enabled by default. **Usage**: `--limit_parameters  off`
`--smart_search_seed`	Optionally specify the search seed. If omitted, a time-based seed will be used. **Usage**: `--smart_search_seed  82427`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Classification options: • `gini` (default) • `fscore` • `accuracy` • `capture_dev` • `precision_at_k` • `yield` Note that if `yield` is specified, then `--yield_values_in` must also be specified. Regression options: • `mean_absolute_error` (default) • `mean_squared_error` • `coeff_determination` • `normalized_gini` **Usage**: `--testing_objective  capture_dev`

Table 7: *General AutoModel Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to off. **Usage**: `--fast_read  on`

***Table 7:** General AutoModel Options (continued)*

Command	Description
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`

***Table 7:** General AutoModel Options (continued)*

Command	Description
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Collaborative Filtering Options

The following tables show the options available in the `cf` module.

***Table 8:** CF Input Data Options*

Command	Description
`--item_features_in`	Optionally specify a file with item features intended to improve recommendation accuracy. This file has the same format as the user item file except that the first column is the index of the feature. **Usage**: `--item_features_in item_features.st`

`--item_features_value_in`	Optionally specify a file with the value of item features intended to improve recommendation accuracy. This has the same format as the `--training_ratings_in` file. This option is used with `--alpha`. If `--alpha=1` is specified and this option is not, then item features will be interpreted as unary.
	Usage:
	`--alpha  1 \`
	`--item_features_in  item_features.st \`
	`--item_features_value_in  item_features.value`

`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in  item.model`
`--testing_candidates_in`	Optionally specify a file containing the candidate item data for testing. **Usage**: `--testing_candidates_in  user_item.candidate.test`
`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  user_item.ratings.st`
`--testing_ratings_in`	Optionally specify a file containing the ratings for the new users. **Usage**: `--testing_ratings_in  user_item.ratings.ratings`

Table 8: *CF Input Data Options (continued)*

Command	Description
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in  user_item.st`
`--training_ratings_in`	Specify the file containing the class labels of the training data. This option is required if `--unary=off`. **Usage**: `--training_ratings_in  user_item.ratings`
`--users_in`	Optionally specify a file with user IDs for which to give recommendations. If omitted, use all user IDs found in the training data. **Usage**: `--users_in  sdss.users`

Table 9: *CF Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio  0.2`
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  5`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If this is provided, then `--tuning_ratings_in` is required.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_ratings_in  sdss.tune.ratings`

`--tuning_models_out`	File prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_models_out  sdss.tune.models`

Table 9: *CF Model Validation Options (continued)*

Command	Description

`--tuning_ratings_in`	Optionally specify a file containing the ratings of the tuning data. Required if `--tuning_in` is provided.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_ratings_in  sdss.tune.ratings`

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or JSON. This value defaults to CSV.
	
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible JSON format.
	
	Usage:
	`--tuning_results_out  sdss.tune.results \`
	`--tuning_results_format  json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to JSON using `--tuning_results_format`. **Usage**: `--tuning_results_out  sdss.tune.results`

Table 10: *CF Output Data Options*

Command	Description
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  sdss.lsvm.model.0.1`
`--output_with_ids`	If enabled, the per-point `--ratings_out` output will be prepended with the `--testing_in` file's "id" meta field followed by a comma. For example, if the original output is "a,b,c" then the new output would be "ID,a,b,c". If an "id" meta field is not available in the input, then "-1" is set as the input "id" field (for example, "-1,a,b,c"). This option is disabled by default. **Usage**: `--output_with_ids  on`
`--ratings_out`	Optionally specify a file to store computed ratings. **Usage**: `--ratings_out  sdss.tune.ratings`
`--recommendations_out`	Optionally specify a file to store the recommendations. Note that items rated by the user are ignored when computing recommendations. By default, this file outputs in Mahout format. You can change the output format using the `--format_out` option. The columns in the output are `user_id`,`item_id`,`rating`. **Usage**: `--recommendations_out  recommendations`

Table 10: *CF Output Data Options (continued)*

Command	Description

`--similarity_out`	Optionally specify a file to store the similarity matrix.
	Usage:
	`--similarity_function  cosine \`
	`--similarity_out  similarity_matrix`

Table 11: *Tunable Collaborative Filtering options. Specify a single value, comma-separated list or <min>:<step size>:<max>*

Command	Description
`--alpha`	Tunable parameter for inclusion of item features. Specify `alpha=0` (default) to use pure collaborative filtering, or use `alpha=1` for pure content-based filtering. **Usage**: `--alpha  1`

`--cond_prob_damping`	Damping used for conditional probability. Used only in conjunction with `--similarity_function=conditional_prob`.
	Usage:
	`--similarity_function  conditional_prob \`
	`--cond_prob_damping  10`

`--k_neighbors`	Optionally specify the number of neighbors for collaborative filtering. This value defaults to 0. For 0, all neighbors are used. **Usage**: `--k_neighbors  5`
`--max_sims_per_item`	Optionally specify the maximum number of similarities retained for each item. If 0 (default) is specified, all similar items are retained. Note that by restricting the number of similarities per item, accuracy may deteriorate. **Usage**: `--max_sims_per_item  2`
`--min_ratings_per_user`	Optionally specify the minimum number of ratings per user for consideration in the similarity matrix computation. This parameter affects accuracy, speed and memory usage indirectly via the size of the resulting similarity matrix. With a default value of 1, every rated item will be included. **Usage**: `--min_ratings_per_user  0`

Table 11: *Tunable Collaborative Filtering options. Specify a single value, comma-separated list or <min>:<step size>:<max> (continued)*

Command	Description

`--similarity_function`	Optionally specify the similarity function(s) to use. Select one or multiple (comma-separated) of the following:
	• `cosine` (default for `--unary=on`)
	• `adjusted_cosine` (default for `--unary=off`)
	• `adjusted_cosine_norm`
	• `pearson`
	• `pearson_norm`
	• `co-occurrence`
	• `conditional_prob` (unary only)
	• `jaccard` (unary only)
	• `all` (for tuning)
	Usage:
	`--unary  off \`
	`--similarity_function  co-occurrence \`
	`--training_ratings_in  user_items.ratings`

`--threshold`	Specify to discard item pairs with a similarity value below this. This value defaults to 0. **Usage**: `--threshold  -0.1`

Table 12: *Collaborative Filtering Options*

Command	Description

`--biased_prediction`	If set, correct for biased baseline during prediction. Only for `adjusted_cosine`, `adjusted_cosine_norm`, `pearson`, and `pearson_norm`.
	Usage:
	`--similarity_function  pearson \`
	`--biased_prediction  on`

`--compression`	Compression for data. Reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`

__Table 12:__ Collaborative Filtering Options (continued)

Command	Description

`--format_out`	By default, the `--recommendations_out` file outputs in Mahout format. You can use the `--format_out` option to change this to one of the following formats.
	• `json` - output will be a json file
	• `csv` - output will be a csv file in the format `user_id,item_id,rating`
	Note that the specified format should match the file extension of the `--recommendations_out` file.
	Usage:
	`--recommendations_out  recommendations.json \`
	`--format_out  json`

`--k_for_precision`	Optionally specify *k* as an integer to be used for holdout sets and user-by-user precision at *k*. This value defaults to 5. **Usage**: `--k_or_precision  4`

`--mean_impute`	Specify to perform mean imputation if fewer than `--num_recommendations` recommendations are found for a user. This option cannot be used with `--unary=on`. This value defaults to on, but is automatically turned off when `--unary=on` is specified.
	Usage:
	`--num_recommendations  10 \`
	`--mean_impute  off`

`--metric_weights_in`	Optionally specify a file with metric weights for item feature similarity computation. **Usage**: `--metric_weights_in  random.csv`

`--normalize_similarity`	Specify to normalize rows of the similarity matrix. This value defaults to off. Can only be used in conjunction with `--unary=on`.
	Usage:
	`--unary  on \`
	`--normalize_similarity  on`

`--num_recommendations`	Specify the number of recommendations to compute per user. This option defaults to 10. **Usage**: `--num_recommendations  20`

***Table 12:** Collaborative Filtering Options (continued)*

Command	Description
`--objective_function`	The `cf` module by default selects the tuned model with the best mean absolute error to be used for testing and/or file output. Use the `--objective_function` option to specify an alternate objective function to use during tuning. Values include the following: • `absolute_precision` • `relative_precision` • `hit_rate` • `absolute_error` (default) • `mean_squared_error` • `l1_relative_error` • `l2_relative_error` **Usage**: `--objective_function  relative_precision`

`--popularity_impute`	Specify to push in the most popular items if fewer than `--num_recommendations` recommendations are found. This value defaults to off, but it is automatically turned on if `--unary=on` is specified.
	Usage:
	`--num_recommendations  10 \`
	`--popularity_impute  on`

`--recurrent`	If enabled, then recommendations will be computed for items that were previously rated in the training set. This option is disabled by default. **Usage**: `--recurrent  off`
`--rank_error_tol`	Specify a value between 0 and 1 for the relative rank error tolerance for item feature similarity computation in conjunction with `--max_sims_per_item`. This value defaults to 0. **Usage**: `--rank_error_tol  0.05`
`--unary`	Consider data without ratings, e.g., hit, no hit. Supported similarity functions are: • `cosine` • `conditional_prob` • `jaccard` This option defaults to off. In addition, this option cannot be used with `--mean_impute`. **Usage**: `--unary  on`

Table 13: *General CF Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to off. **Usage**: `--fast_read  on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file  input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`

Table 13: *General CF Options (continued)*

Command	Description
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Convert Data Options

The following tables show the options available in `convert_data.sh`.

Table 14: *Convert Data Input Data Options*

Command	Description
`-file`	Required. Specify the file data input file. **Usage**: `-file  income.data`
`-header_in`	Required. Specify the header input file. **Usage**: `-header_in  income.header`
`-weights`	Specify a file containing a weight vector for columns. The weights must be in CSV format and contain exactly the same number of entries as the number of relevant rows in the header file. **Usage**: `-weights  weights.csv`

Table 15: *Convert Data Output Data Options - Non-Time-Series*

Command	Description
`-data_out`	Required. Specify an output file name. If `-time_series=on`, then specify an output directory. **Usage**: `-data_out  income.data_prep.st`

`-labels_out`	The output labels file. This requires `-label_index`. This cannot be used with `-time_series`.
	Usage:
	`-label_index  15 \`
	`-labels_out  income.data_prep.labels`

`-targets_out`	The output targets file. This requires `-target_index`. This cannot be used with `-time_series`.
	Usage:
	`-target_index  1`
	`-targets_out  income.data_prep.targets`

Table 16: *Convert Data Output Data Options - Time-Series-Only*

Command	Description

`-id_map_file`	The output file with the file name for each ID. This is required for Time Series. This cannot be specified when `-time_series=off`.
	Usage:
	`-time_series  on \`
	`-id_map_file  specimen.map.out`

Table 17: *Convert Data Options - Time-Series-Only*

Command	Description

`-max_open_files`	When `-time_series=on`, specify the maximum number of files to be kept open at the same time.
	Usage:
	`-time_series  on \`
	`-max_open_files  10`

`-time_series`	If enabled, this indicates that data represents a time series. This option is disabled by default. **Usage**: `-time_series  on`

Table 18: Convert Data Additional Options

Command	Description
`-clamp_out_of_range`	If supplied, values beyond the min and max range in the header will be clamped. This option cannot be used with `-ignore_out_of_range`. **Usage**: `-clamp_out_of_range on`
`-extract_from_timestamp`	When a header file includes timestamped data, you can use this option to extract specific date information. Specify a comma-separated list of any of the following. A column is created in the output file for each option that is specified. • `millisecond` • `second` • `minute` • `hour_of_day` • `day_of_week` • `day_of_month` • `week_of_month` • `month` • `day_of_year` • `week_of_year` • `year` **Usage**: `-extract_from_timestamp second,month,day_of_week,year`
`-horizontalize`	If enabled, this creates a column for each possible value of a categorical feature. **Usage**: `-horizontalize on`

`-id_index`	For Time-Series, specify a column number or column name indicating the row identifier. This option is required if `-time_series=on`.
	
	Usage:
	`-time_series on \`
	`-id_index 2`
	
	For non-Time-Series, if this is specified during `generate-header.sh`, then either the same `-id_index` must be specified here (thereby copying the strings into the output file) or the column must be ignored using `-ignore_columns`.
	
	Usage:
	`-id_index 2`

Table 18: Convert Data Additional Options (continued)

Command	Description
`-ignore_columns`	A comma-separated list of column numbers, column names, or ranges of these to be ignored. For Time Series, this represents the columns to ignore in sliding windows. **Usage**: `-ignore_columns  2,6-9,13`
`-ignore_constant_columns`	Specify whether to ignore columns with a single value for all rows. **Usage**: `-ignore_constant_columns  on`
`-ignore_lines`	Specify the number of header lines to ignore. When this option is used with multiple `-file` arguments, it must be specified once for each. When multiple `-ignore_lines` are specified, the first instance of `-ignore_lines` applies to the first `-file` argument, and so on. **Usage**: `-ignore_lines  1`
`-ignore_missing`	If enabled, rows with missing values will be ignored. This value defaults to off. **Usage**: `-ignore_missing  on`
`-ignore_new_words`	Specify whether to ignore words not appearing in the words_columns dictionaries. (Refer to the description for -words_columns (page 246) for more information.) This option is disabled by default. **Usage**: `-ignore_new_words  on`
`-ignore_out_of_range`	If supplied, rows with values beyond the min and max ranges in the header will be ignored. This option cannot be used with `-clamp_out_of_range`. This value is disabled by default. **Usage**: `-ignore_out_of_range  on`
`-label_index`	The column number or name indicating the classification label. For example, a value of 15 identifies column 15 as the label index. Label indices start at 1. **Usage**: `-label_index  10`
`-mean_impute`	When enabled, missing values for numerical features are replaced with mean values, and missing values for categorical features are treated as distinct values. This value defaults to on. **Usage**: `-mean_impute  off`

***Table 18:** Convert Data Additional Options (continued)*

Command	Description
`-normalize`	The normalization method to use. Specify one of the following: • unit: This form makes the data range between 0 and 1. • standard: This gives the data a mean value of 0 and unit variance. **Usage**: `-normalize unit`
`-rare_words_pct`	Filter words less frequent than this percentage. This value defaults to 0.0. **Usage**: `-rare_words_pct 10`
`-sparse_columns`	A comma-separated list of column numbers, column names, or ranges of columns containing sparse data. An `all_columns` keyword can also be specified. **Usage**: `-sparse_columns 11,12`
`-stop_words_pct`	Filter words more frequent than this percentage. This value defaults to 100.0. **Usage**: `-stop_words 10`
`-target_index`	The column number or column name of the regression target. **Usage**: `-target_index 1`

Database Connection Options

The following table shows the options available with the `db-connect.sh` script.

***Table 19:** Database Connect Options*

Command	Description

`-categorical_number`	Comma-delimited list of names of the numerical columns to be considered categorical. When specified, numeric values within those columns are considered as categorical numbers. (For example, 0100, 100, and 100.0 are treated as the same value.) Requires `-header_out`.
	Usage:
	`-header_out header.actual \`
	`-categorical_number 5,9,3`

`-csv_out`	The csv-formatted output file. Note that, while output files are in CSV format, header files are in the same format as those created by `generate-header.sh`. **Usage**: `-csv_out data.actual`

Table 19: *Database Connect Options (continued)*

Command	Description
`-db_url`	REQUIRED. The database to connect to. Please consult your database vendor's documentation to determine how to use your JDBC driver, as these URL formats can change between releases. **Usage**: `-db_url  jdbc:sqlite:ints.db`
`-header_out`	The output header file. **Usage**: `-header_out  header.actual`
`-jdbc_driver`	REQUIRED. The JDBC driver class name. Be sure to consult your database vendor's documentation to determine how to obtain a JDBC driver and how to use it. **Usage**: `-jdbc_driver  org.sqlite.JDBC`

`-labels_column`	Optionally specify the name of the column containing classification labels. Requires `-header_out`.
	
	Usage:
	`-header_out  data.actual \`
	`-labels_column 2`

`-mysql_stream`	When using MySQL database, specify whether to stream results in chunks. **Usage**: `-mysql_stream  on`
`-null_string`	Enter a string that will represent database NULLs in the output. This string defaults to `NULL`. **Usage**: `-null_string  NULLs`

`-password`	If your database requires authentication, use this to specify your password. Note that if `-user` is specified without `-password`, you will be prompted to enter a password.
	
	Usage:
	`-user  auser \`
	`-password  auser.password`

`-postgresql_stream`	When using PostgreSQL, specify whether to stream results in chunks. **Usage**: `-postgresql_stream  on`
`-regard_case`	Specify whether to regard the case of textual data. For example, if this option is enabled, then USD and usd will be treated the same. **Usage**: `-regard_case  on`

Table 19: Database Connect Options (continued)

Command	Description

`-sql_query`	REQUIRED. Specify the SQL query to execute.
	Usage:
	`-sql_query  "SELECT some_int,some_tiny_int, \`
	`             some_small_int, some_big_int \`
	`             FROM int_table"`

`-user`	If your database requires authentication, use this to specify your user name. Note that if `-user` is specified without `-password`, you will be prompted to enter a password. **Usage**: `-user  auser`

Generalized Linear Model Classification Options

The following tables show the options available in the `glmc` module.

Table 20: GLMC Input Data Options

Command	Description
`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in  model`
`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  sdss_test.st`
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in  sdss.train.st`
`--training_labels_in`	Specify the file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`
`--yield_values_in`	Optionally specify the file used to calculate yield values during tuning. This must be the same length as either the training or tuning vector, depending on whether you are using holdouts or a tuning table. Note that if `--testing_objective=yield` is specified, then this option is required, and its value will be used to determine the best model. **Usage**: `--yield_values_in  income.yield.st`

Table 21: *GLMC Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio 0.2`
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed 2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds 10`
`--smart_search`	Specify to use an intelligent tuning technique (instead of naive grid search). No tuning parameters need to be specified when using this flag. However, if desired, users can provide open or closed intervals for any tuning parameter using `<min>:<max>` or `<min>: or :<max>`. Specific values at regular intervals can also be specified with `<min>:<step>:<max>`. Note that this option cannot be used in conjunction with `--model_in` or `--probability_threshold`. **Usage**: `--smart_search on`

`--smart_search_iterations`	When `--smart_search=on`, optionally specify the number of search rounds to try for tuning. This value must be greater than 0 and defaults to 100.
	Usage:
	`--smart_search on \`
	`--smart_search_iterations 500`

`--smart_search_seed`	When `--smart_search=on`, optionally specify the search seed. If omitted, a time-based seed will be used.
	Usage:
	`--smart_search on \`
	`--smart_search_seed 82427`

`--smart_search_restart_in`	When using smart search, optionally specify to load data from a file to continue smart search from saved data.
	Usage:
	`--smart_search on \`
	`--smart_search_iterations 10 \`
	`--smart_search_restart_in income.data.restart`

***Table 21:** GLMC Model Validation Options (continued)*

Command	Description

`--smart_search_restart_out`	When using smart search, optionally specify to save smart search data to a file. This file can then be specified when running additional smart search iterations.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  10 \`
	`--smart_search_restart_out  income.train.restart`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_labels_in`	If `--tuning_in` is specified, then also specify a file containing the labels of the tuning data.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_models_out`	File prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`. Note that this generates a single model file storing all of the trees tuned over rather than generating separate lists. To limit the number of trees when using this model, specify the `--num_trees` option when loading this model.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels \`
	`--tuning_models_out  sdss.tune.models`

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or JSON. This value defaults to CSV.
	
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible JSON format.
	
	Usage:
	`--tuning_results_out  sdss.tune.results \`
	`--tuning_results_format  json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to JSON using `--tuning_results_format`. **Usage**: `--tuning_results_out  sdss.tune.results`

***Table 22:** GLMC Output Data Options*

Command	Description
`--labels_out`	Optionally specify a file to store computed labels. **Usage**: `--labels_out  results`
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  sdss.lsvm.model.0.1`
`--output_with_ids`	If enabled, the per-point output for `--labels_out` and `--probabilities_out` will be prepended with the input file's "`id`" meta field followed by a comma. For example, if the original output is "`a,b,c`" then the new output would be "`ID,a,b,c`" where "`ID`" is an integer (`1, -1`). If an "`id`" meta field is not available in the input, then "`0`" is set as the input "`id`" field (for example, "`0,a,b,c`"). This option is disabled by default. **Usage**: `--output_with_ids  on`
`--partial_dependencies_out`	Specify the file to store the partial dependencies of the relevant features and/or feature pairs in JSON format. These can be used to generate partial dependence plots. This option is available when: • Training along with (model save OR testing) • Tuning followed by training + (model save OR testing) **Usage**: `--partial_dependencies_out  dependencies.json`
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Usage**: `--pmml_out  income.pmml`
`--probabilities_out`	Optionally specify a file to store computed probabilities. **Usage**: `--probabilities_out  sdss.test.probabilities`

***Table 23:** Tunable GLMC Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>.*

Command	Description
`--l1_penalty`	Specify the penalty value for L1 (LASSO) regularization. This value must be >= 0.0. Specifying both L1 and L2 values > 0 will result in Elastic Net regularization. **Usage**: `--l1_penalty  0.1`

Table 23: *Tunable GLMC Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>. (continued)*

Command	Description
`--l2_penalty`	Specify the penalty value for L2 (Ridge) regularization. This value must be > = 0.0 and defaults to 1.0. Specifying both L1 and L2 values > 0 will result in Elastic Net regularization. **Usage**: `--l2_penalty 0.5`

Table 24: *Non-Tunable GLMC Options*

Command	Description
`--classification_objective`	Optionally specify the objective for classification threshold tuning. Specify either `fscore` (default) or `accuracy`. This option cannot be used in conjunction with `--probability_threshold`. **Usage**: `--classification_objective  accuracy`
`--epsilon`	Specify the numerical accuracy to which to train the GLMC model. This value must be > 0.0 and defaults to 0.001. **Usage**: `--epsilon  0.002`
`--exclude_bias_term`	If set, the GLMC formulation will not have a bias term. A bias term is used by default. **Usage**: `--exclude_bias_term  on`
`--k_for_precision`	Optionally specify *k* as a value between `0` and `1` for precision at the top 100*k*-th percentile. *k* defaults to `0.1`. **Usage**: `--k_for_precision  0.2`
`--link`	Specify the link function to use for the GLMC model. Options include: • `logit` (default): logistic • `cloglog`: complementary log-log **Usage**: `--link  cloglog`
`--max_cache_memory`	Specify the maximum allowed memory, in megabytes, used to cache the rows of the kernel matrix (to reduce recomputation). This value defaults to 500MB. **Usage**: `--max_cache_memory  1000`

***Table 24:** Non-Tunable GLMC Options (continued)*

Command	Description
`--max_iterations`	Specify the maximum allowed number of iterations for the training algorithm. This value must be > `0` and defaults to 1000. For GLMC training, this corresponds to the number of passes over the data. **Usage**: `--max_iterations 5000000`
`--probability_threshold`	Optionally specify the probability to be used as the threshold for classification. Cannot be used in conjunction with `--classification_objective` or `--testing_objective`. Similarly, because `--probability_threshold` cannot be used during tuning, this option cannot be specified with `--smart_search`. **Usage**: `--probability_threshold 0.8208208919380429`
`--table_sampling_seed`	Specify the data sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--table_sampling_seed 1359937539`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Specify one of the following: • `gini` (default) • `fscore` • `accuracy` • `capture_dev` • `precision_at_k` • `yield` Cannot be used in conjunction with `--probability_threshold`. Note that if `yield` is specified, then `--yield_values_in` must also be specified. **Usage**: `--testing_objective capture_dev`

***Table 25:** General GLMC Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to `off`. **Usage**: `--fast_read on`

Table 25: *General GLMC Options (continued)*

Command	Description
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file  input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`

Table 25: *General GLMC Options (continued)*

Command	Description
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Generalized Linear Model Regression Options

The following tables show the options available in the `glmr` module.

Table 26: *GLMR Input Data Options*

Command	Description
`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in  model`
`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  kddcup_test.st`
`--training_in`	Specify the file containing the training data. **Usage**: `--training_in  kddcup.train.st`
`--training_targets_in`	Specify the file containing the targets of the training data. **Usage**: `--training_targets_in  kddcup.targets`

***Table 27:** GLMR Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio  0.2`
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  5`
`--smart_search`	Specify to use an intelligent tuning technique (instead of naive grid search). No tuning parameters need to be specified when using this flag. However, if desired, users can provide open or closed intervals for any tuning parameter using <min>:<max> or <min>: or :<max>. Specific values at regular intervals can also be specified with <min>:<step>:<max>. Note that this option cannot be used in conjunction with `--model_in`. **Usage**: `--smart_search  on`

`--smart_search_iterations`	When `--smart_search=on`, optionally specify the number of search rounds to try for tuning. This value must be greater than 0 and defaults to 100.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  500`

`--smart_search_restart_in`	When using smart search, optionally specify to load data from a file to continue smart search from saved data.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  10 \`
	`--smart_search_restart_in  kddcup.restart`

`--smart_search_restart_out`	When using smart search, optionally specify to save smart search data to a file. This file can then be specified when running additional smart search iterations.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  10 \`
	`--smart_search_restart_out  kddcup.train.restart`

Table 27: *GLMR Model Validation Options (continued)*

Command	Description

`--smart_search_seed`	When `--smart_search=on`, optionally specify the search seed. If omitted, a time-based seed will be used.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_seed  82427`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`.
	
	Usage:
	`--tuning_in  kddcup.tune.st \`
	`--tuning_targets_in  kddcup.targets`

`--tuning_models_out`	Optionally specify a file prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`.
	
	Usage:
	`--tuning_in  kddcup.tune.st \`
	`--tuning_models_out  kddcup.tune.models`

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or JSON. This value defaults to CSV.
	
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible JSON format.
	
	Usage:
	`--tuning_results_out  kddcup.tune.results \`
	`--tuning_results_format  json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to JSON using `--tuning_results_format`. **Usage**: `--tuning_results_out  kddcup.results`

`--tuning_targets_in`	Specify an file containing the targets of the tuning data. Required if `--tuning_in` is provided.
	
	Usage:
	`--tuning_in  income.tune.st  \`
	`--tuning_targets_in  income.tune.targets`

Table 28: *GLMR Output data Options*

Command	Description
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  kddcup.model`

Table 28: *GLMR Output data Options (continued)*

Command	Description
`--output_with_ids`	If enabled, the per-point `--targets_out` output will be prepended with the input file's "id" meta field followed by a comma. For example, if the original output is "a,b,c" then the new output would be "ID,a,b,c" where "ID" is an integer (1, -1). If an "id" meta field is not available in the input, then "0" is set as the input "id" field (for example, "0,a,b,c"). This option is disabled by default. **Usage**: `--output_with_ids  on`
`--partial_dependencies_out`	Specify the file to store the partial dependencies of the relevant features and/or feature pairs in JSON format. These can be used to generate partial dependence plots. This option is available when: • Training along (model save OR testing) • Tuning followed by training + (model save OR testing) **Usage**: `--partial_dependencies_out  dependencies.json`
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Note**: This option is not supported for distributed non-linear SVM. **Usage**: `--pmml_out  sdss.pmml`
`--targets_out`	Optionally specify a file to store computed targets. **Usage**: `--targets_out  targets.glmr`

Table 29: *Tunable GLMR Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>.*

Command	Description
`--l1_penalty`	Specify the penalty value for L1 (LASSO) regularization. This value must be >= 0.0. Specifying both L1 and L2 values > 0 will result in Elastic Net regularization. **Usage**: `--l1_penalty  0.1`
`--l2_penalty`	Specify the penalty value for L2 (Ridge) regularization. This value must be > = 0.0 and defaults to 1.0. Specifying both L1 and L2 values > 0 will result in Elastic Net regularization. **Usage**: `--l2_penalty 0.5`

Table 29: *Tunable GLMR Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>. (continued)*

Command	Description

`--tweedie_exponent`	When `--family=tweedie`, specify a value for the exponent. This option has no default value and, therefore, must be explicitly specified when configuring the Tweedie loss function. This value must be > `1.0` and < `2.0`. If users desire a value of `1`, they should specify the Poisson family and Log link function. Similarly, if users desire an exponent equal to `2.0`, they should specify Gamma family along with the Log link function.
	
	Note that users can also tune over this option with values > `1.0` and < `2.0`. When used for tuning, the output CSV will have an extra column for this value.
	
	Usage:
	`--family tweedie \`
	`--tweedie_exponent 1.5`

Table 30: *Non-Tunable GLMR Options*

Command	Description
`--epsilon`	Specify the numerical accuracy to which to train the GLMR model. This value must be > 0.0 and defaults to 0.001. **Usage**: `--epsilon 0.002`
`--exclude_bias_term`	If set, the GLMR formulation will not have a bias term. A bias term is used by default. **Note**: This option cannot be used with the following family-link pairs: Gamma-Inverse, Poisson-Identity, or Poisson-Sqrt. **Usage**: `--exclude_bias_term on`
`--family`	Specify the family function to use for a GLMR model. Options include: • `gamma` • `gaussian` (default) • `poisson` • `tweedie` **Usage**: `--family poisson`

Table 30: *Non-Tunable GLMR Options (continued)*

Command	Description

`--link`	Specify the link function to use for the GLMR model. Options include:
	• `canonical`. This can be use with any `--family` option.
	• `identity`. This can be specified if `--family` is `gaussian` or `poisson`.
	• `inverse`. This can be specified if `--family` is gamma.
	• `log`. This can be specified if `--family` is gamma, `poisson`, or `tweedie`.
	• `sqrt`. This can be specified if `--family` is `poisson`.
	Usage:
	`--family poisson \`
	`--link log`

`--max_cache_memory`	Specify the maximum allowed memory, in megabytes, used to cache the rows of the kernel matrix (to reduce recomputation). This value defaults to 500MB. **Usage**: `--max_cache_memory 1000`
`--max_iterations`	Specify the maximum allowed number of iterations for the training algorithm. This value must be > `0`. For GLMR training, this corresponds to the number of passes over the data. **Usage**: `--max_iterations 5000000`
`--table_sampling_seed`	Specify the data sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--table_sampling_seed 1359937539`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Specify one of the following: • `mean_absolute_error` (default) • `mean_squared_error` • `coeff_determination` • `normalized_gini` **Usage**: `--testing_objective mean_squared_error`

Table 31: *General GLMR Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to `off`. **Usage**: `--fast_read  on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file  input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

Table 31: *General GLMR Options (continued)*

Command	Description
`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_ threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_ threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Generate Header Options

The following tables show the options available in `generate-header.sh`.

Table 32: *Generate Header Input Data Options*

Command	Description
`-file`	Required. Specify the file that contains the data to be analyzed. This can be entered multiple times to include multiple files. **Usage**: `-file  income.data`
`-max_percentiles`	An optional file containing a vector of maximum percentiles, each between `0` and `1`, for clamping. The vector length must be the same as the number of columns in the data files. Note that this option can be memory intensive. **Usage**: `-max_percentiles  income.max`

Table 32: Generate Header Input Data Options (continued)

Command	Description
`-min_percentiles`	A file containing a vector of minimum percentiles, each between `0` and `1`, for clamping. The vector length must be the same as the number of columns in the data files. Note that this option can be memory intensive. **Usage**: `-min_percentiles  income.min`

Table 33: Generate Header Output Data Options

Command	Description
`-header_out`	Required. Specify the output file. **Usage**: `-header_out  income.header`

Table 34: Generate Header Additional Options

Command	Description
`-categorical_number`	Comma-delimited list of column numbers or names, a ranges of columns, or a range of columns with a step size. When specified, numeric values within those columns are considered as categorical numbers. (For example, `0100`, `100`, and `100.0` are treated as the same value.) **Usage**: `-categorical_number  3,5`
`-categorical_text`	Comma-delimited list of column numbers or names, a ranges of columns, or a range of columns with a step size. When specified, numeric values within those columns are considered as categorical text. (For example, `0100`, `100`, and `100.0` are treated as distinct values.) **Usage**: `-categorical_text  4,6`
`-categorical_warn_pct`	Warn if the number of categories exceeds this percentage of items. **Usage**: `-categorical_warn_pct  .20`
`-date_format`	Specify the date format for the timestamp. If this option is not specified, then the timestamp is assumed to be already converted to a UNIX timestamp. **Usage**: `-date_format  "mm/dd/yy"`

***Table 34:** Generate Header Additional Options (continued)*

Command	Description
`-delimiter`	Specify a character that separates columns, or specify one of `TAB`, `COMMA`, `SEMICOLON`, `VBAR`, or `SPACE`. Auto-detected if not specified. Note that Skytree Server allows you to add quotations around text so that it will not be confused as a delimiter (for example, when the delimiter is specified as `COMMA`, and an entry includes a deliberate comma). **Note:** Because multiple spaces can be used to separate columns, it is necessary to explicitly denote empty column values with `""` when using `-delimiter=SPACE`. **Usage**: `-delimiter  TAB`
`-id_index`	Specify a column number that will include non-categorical text without dictionary formation during `convert-data.sh`. **Usage**: `-id_index  2`
`-ignore_columns`	Specify a single or range of column names or numbers to be ignored during the `generate-header.sh` process. Similar to `-id_index`, these columns will be treated as non-categorical text columns, and dictionaries will not be formed. The header file will show type "ignored" for these columns. Note that when specified here, these columns must also be included in the `-ignore_columns` option during `convert-data.sh`. **Usage**: `-ignore_columns  6-9,13`
`-ignore_inconsistent_rows`	Warn, instead of fail, for rows that have too few or too many columns. **Usage**: `-ignore_inconsistent_rows  on`
`-ignore_lines`	The number of header lines to ignore. When this option is used with multiple `-file` arguments, it must be specified once for each. When multiple `-ignore_lines` are specified, the first instance of `-ignore_lines` applies to the first `-file` argument, and so on. **Usage**: `-ignore_lines  1`
`-label_index`	The column number or name indicating the classification label. For example, a value of `15` identifies column 15 as the label index. Label indices start at 1. **Usage**: `-label_index  10`
`-missing_value`	A string signifying a missing value. This ensures that Skytree Server treats these values as missing rather than as text data. **Usage**: `-missing_value  ?`

***Table 34:** Generate Header Additional Options (continued)*

Command	Description
`-num_sparse_input_columns`	Specify the number of sparse columns included in libsvm-formatted data. **Usage**: `-num_sparse_input_columns  4`
`-regard_case`	Specify whether to regard the case of textual data. For example, if this option is enabled, then USD and usd will be treated the same. **Usage**: `-regard_case  on`
`-sparse_suggest_pct`	Suggest columns be made sparse if ratio of zeros or missing values exceeds this percentage. For example, if this value is `50`, then `generate-header.sh` will indicate any columns for which the ratio of zero to non-zero values exceeds 50 percent. **Usage**: `-sparse_suggest_pact  30`
`-time_stamp`	Optionally specify the column name or number that includes a measure of time to be converted to a UNIX timestamp. **Usage**: `-time_stamp  Date`

`-time_zone`	Specify the timezone for the time stamp to be stored in the header. This value defaults to GMT. When specified, this value must be in the format "GMT + *x*:00"
	Usage:
	`-date-format  "dd/mm/yy hh:mm:ss" \`
	`-time_zone  "GMT + 3:00"`

`-trailing_delimiter`	This flag signals that each row has a trailing delimiter. This can be automatically determined unless missing values are indicated by the empty string. **Usage**: `-trailing_delimiter  on`
`-use_column_names`	Use column names from the first line of the first input file. This implies that `-ignore_lines` for that file is at least 1. **Usage**: `-use_column_names  on`
`-words_columns`	For each possible word in the specified column, this parameter creates a notional column in the output `.st` file. The entry will be `1` if the word occurs in a row, and `0` if it does not. Because most rows don't contain any particular word, the columns are always made sparse. (That is, there is one entry for each word that occurs; otherwise the column is not represented in the output.) **Usage**: `-words_columns  6`

Gradient Boosted Trees Options

The following tables show the options available in the gbt module.

Table 35: *GBT Input Data Options*

Command	Description
`--batch_size`	Specify the batch size for streamed test points. A larger batch size can increase throughput but also increase latency. This value defaults to `1`. **Usage**: `--batch_size 2`

`--classweight`	Specify a value for classweights. When used, you must repeat the `--classweight` option for each class in `--training_labels_in`. The values are used to artificially inflate the impact of the corresponding class. If omitted, classweights of `1` are assumed for every class.
	
	Usage:
	`--training_labels_in sdss.train.labels \`
	`--classweight 10 \`
	`--classweight 1`

`--model_in`	Optionally specify a file from which to load the model. Note that this cannot be used with `--smart_search`. **Usage**: `--model_in model`
`--port`	Optionally specify a port number for streaming test data. Must be >= 1024 if specified. **Usage**: `--port 10000`

`--smart_search_restart_in`	When using smart search, optionally specify to load data from a file to continue smart search from saved data.
	
	Usage:
	`--smart_search on \`
	`--smart_search_iterations 10 \`
	`--smart_search_restart_in income.restart`

`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in sdss_test.st`
`--testing_offsets_in`	Specify a file to use as the offset file during testing. If this file is specified with `--model_in`, then you will receive a warning if the input model was trained without offsets. In this case, Skytree Server will continue to run. **Usage**: `--testing_offsets_in income.test.offsets`
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in sdss.train.st`

Table 35: *GBT Input Data Options (continued)*

Command	Description
`--training_labels_in`	REQUIRED. Specify the file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`
`--training_offsets_in`	Specify a file to use as the offset file during training. If this is specified with `--tuning_in`, then `--tuning_offsets_in` must also be specified. **Usage**: `--training_offsets_in  income.test.offsets`
`--training_point_weights_in`	Optionally specify a file containing the point weights (for training a model) for each point in `--training_in`. By default, each training point gets a point weight of 1. **Usage**: `--training_point_weights_in  income.weights`
`--training_score_weights_in`	When evaluating a model, optionally specify a file containing the score weights for each point in `--training_in`. By default, each point in `--training_in` gets a score weight of 1. `--training_score_weights_in` will only be used if `--tuning_in` is not provided and you use a holdout from the `--training_in` for tuning. Please specify `--tuning_score_weights_in` to tune on `--tuning_in` with score weights. **Usage**: `--training_score_weights_in  income.weights`
`--tuning_offsets_in`	Specify a file to use as the offset file during tuning. Note that when this is used with `--tuning_in`, the `--training_offsets_in` file must also be specified. **Usage**: `--tuning_offsets_in  income.tune.offsets`
`--yield_values_in`	Optionally specify the file used to calculate yield values during tuning. This must be the same length as either the training or tuning vector, depending on whether you are using holdouts or a tuning table. Note that if `--testing_objective=yield` is specified, then this option is required, and its value will be used to determine the best model. **Usage**: `--yield_values_in  income.yield.st`

Table 36: *GBT Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio  0.2`

Table 36: *GBT Model Validation Options (continued)*

Command	Description
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  10`
`--smart_search`	Specify to use an intelligent tuning technique (instead of naive grid search). No tuning parameters need to be specified when using this flag. However, if desired, users can provide open or closed intervals for any tuning parameter using `<min>:<max>` or `<min>: or :<max>`. Specific values at regular intervals can also be specified with `<min>:<step>:<max>`. Note that this option cannot be used in conjunction with `--model_in` or `--probability_threshold`. **Usage**: `--smart_search  on`

`--smart_search_iterations`	When `--smart_search=on`, optionally specify the number of search rounds to try for tuning. This value must be greater than `0` and defaults to `100`.
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  500`

`--smart_search_seed`	When `--smart_search=on`, optionally specify the search seed. If omitted, a time-based seed will be used.
	Usage:
	`--smart_search  on \`
	`--smart_search_seed  82427`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_labels_in`	If `--tuning_in` is specified, then also specify a file containing the labels of the tuning data.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

Table 36: *GBT Model Validation Options (continued)*

Command	Description

`--tuning_models_out`	File prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`. Note that this generates a single model file storing all of the trees tuned over rather than generating separate lists. To limit the number of trees when using this model, specify the `--num_trees` option when loading this model.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels \`
	`--tuning_models_out  sdss.tune.models`

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or JSON. This value defaults to CSV.
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible JSON format.
	Usage:
	`--tuning_results_out  sdss.tune.results \`
	`--tuning_results_format  json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to JSON using `--tuning_results_format`. **Usage**: `--tuning_results_out  sdss.tune.results`
`--tuning_score_weights_in`	Optionally specify a file containing the score weights for each of the tuning points. By default, each tuning point gets a weight of 1. Use this option with `--tuning_in` to tune with score weights. **Usage**: `--tuning_score_weights_in  income.tune.weights`

Table 37: *GBT Output Data Options*

Command	Description
`--labels_out`	Optionally specify a file to store computed labels. **Usage**: `--labels_out  results`
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  sdss.lsvm.model.0.1`

***Table 37:** GBT Output Data Options (continued)*

Command	Description
`--output_with_ids`	If enabled, the per-point output for `--labels_out`, `--probabilities_out`, and `--test_point_variable_importances_out` (non-EGBT only) will be prepended with the input file's "`id`" meta field followed by a comma. For example, if the original output is "`a,b,c`" then the new output would be "`ID,a,b,c`" where "`ID`" is an integer (`1, -1`). If an "`id`" meta field is not available in the input, then "`0`" is set as the input "`id`" field (for example, "`0,a,b,c`"). This option is disabled by default. **Usage**: `--output_with_ids on`
`--partial_dependencies_out`	Specify the file to store the partial dependencies of the relevant features and/or feature pairs in JSON format. These can be used to generate partial dependence plots. This option is available when: • Training along with (model save OR testing) • Tuning followed by training + (model save OR testing) **Usage**: `--partial_dependencies_out dependencies.json`
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Notes**: • This option is not supported for multi-class classification in GBT. • This option cannot be used in conjunction with a `--loss_function` that is specified as `map`, `ndgc`, or `mrr`. • The PMML file that is produced requires that missing values are encoded as '?'. This is a reserved symbol for missing values, and as such, should not be used anywhere else in your data. **Usage**: `--pmml_out income.pmml`
`--probabilities_out`	Optionally specify a file to store computed probabilities. **Usage**: `--probabilities_out sdss.test.probabilities`
`--scores_out`	Optionally specify a file to store raw scores that have not been converted to labels, probabilities, or targets. This file can then be specified during training/tuning/testing. **Usage**: `--scores_out income.test.offsets`

Table 37: GBT Output Data Options (continued)

Command	Description

`--smart_search_restart_out`	When using smart search, optionally specify to save smart search data to a file. This file can then be specified when running additional smart search iterations.
	
	Usage:
	`--smart_search on \`
	`--smart_search_iterations 10 \`
	`--smart_search_restart_out income.data.restart`

`--split_importances_buckets`	Optionally specify a number of buckets in split importance calculation. Must be > 1. This value defaults to 10.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split.importances.out \`
	`--split_importances_buckets 4`

`--split_importances_out`	Optionally specify a file to store split importances. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split_importances.out`

`--test_point_variable_ importances_out`	Optionally specify a file to store variable importances for each test point. Requires `--testing_in`.
	
	Usage:
	`--testing_in income.data.st \`
	`--test_point_variable_importances_out income.data.vimps`

`--variable_importances_out`	Optionally specify a file to store intrinsic variable importances. This output file includes a single column. The first row is the importances of the first column variable in the `--training_in` .st file, the second row is the importance of the second column, and so on. Requires `--training_in` and one of `--testing_in` or `--model_ out`.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--variable_importances_out variable_importances.out`

`--variable_importances_out_ as_json`	This option specifies to write variable importances as a JSON file. The output will include feature names, attribute names, and column IDs along with the importance.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--variable_importances_out variable_importances.out`
	`--variable_importances_out_as_json on`

Table 37: *GBT Output Data Options (continued)*

Command	Description

`--visualization_out`	Optionally specify a directory for model visualization (created if it does exist). Requires `--training_in` and one of `--testing_in` or `--model_out`. Cannot be used in conjunction with `--ensemble_size`.
	Usage:
	`--training_in  income.data.st \`
	`--testing_in  income.test.st \`
	`--visualization_out  income`

Table 38: *Tunable GBT Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>.*

Command	Description
`--ensemble_size`	Use `--ensemble_size` GBTs with sampled tables and randomly sampled split dimensions. **Usage**: `--ensemble_size  10`

`--imbalance_scale`	Relative size of all classes with respect to the smallest class. Requires `--imbalance`. This value defaults to 1.0.
	Usage:
	`--imbalance \`
	`--imbalance_scale  2`

`--leaf_nodes`	The number of leaf nodes in the tree (default: 0). Must be >=0. A value of 0 disables the parameter. Only one of `--tree_depth` or `--leaf_nodes` can be non-zero at a time. This option is deprecated. Use `--max_splits=<leaf_Nodes-1>` instead. **Usage**: `--leaf_nodes  5,10,25`
`--learning_rate`	The learning rate. Must be > `0` and <= `1`. This value defaults to `0.1`. **Usage**: `--learning_rate  0.05:0.05:0.2`
`--max_splits`	Specify the number of splits in the tree. This value must be >=`0`. A value of `0` disables this option. Either this value or `--min_node_weight` must be > `0` if `--tree_depth` is explicitly set to 0. If both `--tree_depth` and `--max_splits` are > 0, then Skytree Server finds the best splits that exist within the depth level, up to `--max_splits`. This option cannot be used in conjunction with (deprecated) `--leaf_nodes`. **Usage**: `--max_splits  4`

***Table 38:** Tunable GBT Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>. (continued)*

Command	Description
`--min_node_weight`	The minimum bound on the total weight in each leaf during tree building. This value defaults to 0. Either this value or `--max_splits` must be > 0 if `--tree_depth` is explicitly set to 0. **Usage**: `--min_node_weight  4`
`--num_dimensions`	The number of dimensions to randomly sample at each tree node. This option is only used in conjunction with `--ensemble_size`. If omitted, all dimensions are used. **Usage**: `--num_dimensions  1:1:4`
`--num_trees`	The number of trees in the ensemble. Must be > `0`. **Usage**: `--num_trees  5:5:100`

`--regularization_bins`	The number of bins to use per dimension/attribute of the data. Used only in conjunction with `--regularization`. If omitted (with `--regularization=on`), 200 bins will be used.
	Usage:
	`--regularization  on \`
	`--regularization_bins  100`

`--sampling_ratio`	Per class sampling ratio. You must either specify a single `--sampling_ratio` that will be used for all classes or repeat the `--sampling_ratio` option for each class in `--training_labels_in`. If omitted with an `--ensemble_size` set and `--sample_with_replacement=on`, then full bootstrapping will be used. If an `--ensemble_size` is set and `--sample_with_replacement=off`, then this option is mandatory.
	Usage:
	`--training_labels_in  income.data.labels \`
	`--sampling_ratio  0.25 \`
	`--sampling_ratio  2`

`--tree_depth`	Specifies the depth to which each ensemble is built. If explicitly set to 0, trees will be built to the fullest extent. In this case, you must specify a positive value for either `--max_splits` or `--min_node_weight`. If this value is not set, and values for `--max_splits` and `--min_node_weight` are likewise not set, then this value defaults to 3. If both `--tree_depth` and `--max_splits` are > 0, then Skytree Server finds the best splits that exist within the depth level, up to `--max_splits`. **Usage**: `--tree_depth  0:2:10`

Table 39: *Non-tunable GBT Options*

Command	Description
`--cardinality_based_ dimension_sampling`	Categorical dimensions with larger number of unique values in a given node will have a proportionately higher chance of being selected when sampling dimensions. Continuous features are treated as having one unique value. When `off`, all dimensions have an equal chance of being selected in dimension sampling. **Usage**: `--cardinality_based_dimension_sampling  off`
`--categorical_random_split_ threshold`	If the number of categorical values exceeds this, then random splitting is attempted. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_random_split_threshold  5`
`--categorical_random_split_ tries`	The number of times to try random categorical splits. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_random_split_tries  8`
`--categorical_sampling_seed`	Optionally specify the categorical values sampling random number seed. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_sampling_seed  4`
`--categorical_selection_ method`	Optionally specify one of the following: • `random` • `random_exact` • `one_vs_all` • `one_vs_all_random` • `one_vs_all_exact` • `exact` Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_selection_method  random`
`--classification_objective`	Optionally specify the objective for classification threshold tuning. Specify either `fscore` (default) or `accuracy`. This option cannot be used in conjunction with `--probability_threshold`. **Usage**: `--classification_objective  accuracy`

***Table 39:** Non-tunable GBT Options (continued)*

Command	Description
`--compression`	Compression for data. Reduces computational resource requirements. **NOTE**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`
`--dimension_sampling_seed`	The dimension sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--dimensions_sampling_seed  10`
`--imbalance`	If set, attempt to improve classification for imbalanced classes. Cannot be used in conjunction with `--sampling_ratio`. **Usage**: `--imbalance  off`
`--k_for_precision`	Optionally specify *k* as a value between `0` and `1` for precision at the top 100*k*-th percentile. *k* defaults to `0.1`. **Usage**: `--k_for_precision  0.2`

`--limit_parameters`	When `--smart_search` is enabled, this option restricts the parameter space in order to reduce the training time. This option is enabled by default.
	Usage:
	`--smart_search on`
	`--limit_parameters  off`

`--loss_function`	Specify the method to use when performing ranking for non-ensemble GBT. Options include the following: • `logistic` (default) • `ndcg` • `map` • `mrr` **Note**: PMML output is not available if `--loss_function` is specified as anything other than `logistic` (default). **Usage**: `--loss_function  ndgc`
`--num_cached_trees`	Optionally specify the number of simultaneously cached trees in the ensemble. Higher values lead to higher memory usage. Lower values result in lower memory usage at the expense of increased runtime. This value defaults to the number of available threads. **Usage**: `--num_cached_trees  6`

Table 39: *Non-tunable GBT Options (continued)*

Command	Description
`--probability_threshold`	Optionally specify the probability to be used as the threshold for classification. Cannot be used in conjunction with `--classification_objective` or `--testing_objective`. Similarly, because `--probability_threshold` cannot be used during tuning, this option cannot be specified with `--smart_search`. **Usage**: `--probability_threshold  0.8208208919380429`
`--regularization`	Use the fast decision tree construction heuristic. Multiple nodes can be used when this option is enabled without the need for additional tuning parameters. This value defaults to on for `gbt`/`gbtr`; it defaults to off for ensemble `gbt`/`gbtr`. **Usage**: `--regularization  off`

`--sample_with_replacement`	For each tree, sample the training points using replacement (bootstrap). This option is on by default. If turned `off`, `--sampling_ratio` must be provided.
	
	Usage:
	`--sample_with_replacement  off`
	`--sampling_ratio  0.25`

`--table_sampling_seed`	The data sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--table_sampling_seed  1359937539`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Specify one of the following: • `gini` (default) • `fscore` • `accuracy` • `capture_dev` • `precision_at_k` • `yield` Cannot be used in conjunction with `--probability_threshold`. Note that if `yield` is specified, then `--yield_values_in` must also be specified. **Usage**: `--testing_objective  capture_dev`

`--trim`	If provided, observations with low importance in the model will be pruned in successive iterations. This can lead to significant speedup, but accuracy can suffer. This option defaults to `off`. Cannot be used in conjunction with `--regularization`.
	
	Usage:
	`--regularization  off \`
	`--trim  on`

Table 39: Non-tunable GBT Options (continued)

Command	Description
`--trim_alpha`	Must be > `0` and < `1`. Higher values lead to more aggressive pruning of observations per iterations. Values between `0.05` and `0.2` are common. This option defaults to `0.2`. **Usage**: `--trim_alpha  0.08`

Table 40: General GBT Options

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to off. **Usage**: `--fast_read  on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file  input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

***Table 40:** General GBT Options (continued)*

Command	Description

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	
	Usage:
	`--hosts localhost,127.0.0.1 \`
	`--procs_per_host 3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads 16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold 1`
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to 0.05. **Usage**: `--watchdog_low_memory_threshold 1`

Gradient Boosted Trees Regression Options

The following tables show the options available in the `gbtr` module.

***Table 41:** GBTR Input Data Options*

Command	Description
`--batch_size`	Specify the batch size for streamed test points. A larger batch size can increase throughput but also increase latency. This value defaults to `1`. **Usage**: `--batch_size 2`

Table 41: *GBTR Input Data Options (continued)*

Command	Description

`--classweight`	Specify a value for classweights. When used, you must repeat the `--classweight` option for each class in `--training_labels_in`. The values are used to artificially inflate the impact of the corresponding class. If omitted, classweights of `1` are assumed for every class.
	
	Usage:
	`--training_labels_in  kddcup.train.labels \`
	`--classweight  10 \`
	`--classweight  1`

`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in  model`
`--port`	Optionally specify a port number for streaming test data. Must be >= `1024` if specified. **Usage**: `--port  10000`

`--smart_search_restart_in`	When using smart search, optionally specify to load data from a file to continue smart search from saved data.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  10 \`
	`--smart_search_restart_in  kddcup.restart`

`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  kddcup_test.st`
`--testing_offsets_in`	Specify a file to use as the offset file during testing. If this file is specified with `--model_in`, then you will receive a warning if the input model was trained without offsets. In this case, Skytree Server will continue to run. **Usage**: `--testing_offsets_in  income.test.offsets`
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in  kddcup.train.st`
`--training_offsets_in`	Specify a file to use as the offset file during training. If this is specified with `--tuning_in`, then `--tuning_offsets_in` must also be specified. **Usage**: `--training_offsets_in  income.test.offsets`
`--training_point_weights_in`	Optionally specify a file containing the point weights (for training a model) for each point in `--training_in`. By default, each training point gets a point weight of 1. **Usage**: `--training_point_weights_in  kddcup.weights`

Table 41: *GBTR Input Data Options (continued)*

Command	Description
`--training_score_weights_in`	When evaluating a model, optionally specify a file containing the score weights for each point in `--training_in`. By default, each point in `--training_in` gets a score weight of 1. `--training_score_weights_in` will only be used if `--tuning_in` is not provided and you use a holdout from the `--training_in` for tuning. Please specify `--tuning_score_weights_in` to tune on `--tuning_in` with score weights. **Usage**: `--training_score_weights_in  kddcup.weights`
`--training_targets_in`	REQUIRED. Specify the file containing the targets of the training data. **Usage**: `--training_targets_in  kddcup.targets`
`--tuning_offsets_in`	Specify a file to use as the offset file during tuning. Note that when this is used with `--tuning_in`, the `--training_offsets_in` file must also be specified. **Usage**: `--tuning_offsets_in  income.tune.offsets`

Table 42: *GBTR Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio  0.2`
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  10`
`--smart_search`	Specify to use an intelligent tuning technique (instead of naive grid search). No tuning parameters need to be specified when using this flag. However, if desired, users can provide open or closed intervals for any tuning parameter using `<min>:<max>` or `<min>: or :<max>`. Specific values at regular intervals can also be specified with `<min>:<step>:<max>`. **Usage**: `--smart_search  on`

Table 42: *GBTR Model Validation Options (continued)*

Command	Description

`--smart_search_iterations`	When `--smart_search=on`, optionally specify the number of search rounds to try for tuning. This value must be greater than `0` and defaults to `100`.
	
	Usage:
	`--smart_search on \`
	`--smart_search_iterations 500`

`--smart_search_seed`	When `--smart_search=on`, optionally specify the search seed. If omitted, a time-based seed will be used.
	
	Usage:
	`--smart_search on \`
	`--smart_search_seed 82427`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	
	Usage:
	`--tuning_in sdss.tune.st \`
	`--tuning_labels_in sdss.tune.labels`

`--tuning_models_out`	File prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`. Note that this generates a single model file storing all of the trees tuned over rather than generating separate lists. To limit the number of trees when using this model, specify the `--num_trees` option when loading this model.
	
	Usage:
	`--tuning_in sdss.tune.st \`
	`--tuning_labels_in sdss.tune.labels \`
	`--tuning_models_out sdss.tune.models`

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or `JSON`. This value defaults to `csv`.
	
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible JSON format.
	
	Usage:
	`--tuning_results_out sdss.tune.results \`
	`--tuning_results_format json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to JSON using `--tuning_results_format`. **Usage**: `--tuning_results_out sdss.tune.results`
`--tuning_score_weights_in`	Optionally specify a file containing the score weights for each of the tuning points. By default, each tuning point gets a weight of 1. Use this option with `--tuning_in` to tune with score weights. **Usage**: `--tuning_score_weights_in income.tune.weights`

Table 42: *GBTR Model Validation Options (continued)*

Command	Description

`--tuning_targets_in`	Specify an file containing the targets of the tuning data. Required if `--tuning_in` is provided.
	
	Usage:
	`--tuning_in  income.tune.st  \`
	`--tuning_targets_in  income.tune.targets`

Table 43: *GBTR Output Data Options*

Command	Description
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  sdss.lsvm.model.0.1`
`--output_with_ids`	If enabled, the per-point output for `--targets_out` and `--test_point_variable_importances_out` (non-EGBTR only) will be prepended with the input file's "`id`" meta field followed by a comma. For example, if the original output is "`a,b,c`" then the new output would be "`ID,a,b,c`" where "`ID`" is an integer (`1, -1`). If an "`id`" meta field is not available in the input, then "`0`" is set as the input "`id`" field (for example, "`0,a,b,c`"). This option is disabled by default. **Usage**: `--output_with_ids  on`
`--partial_dependencies_out`	Specify the file to store the partial dependencies of the relevant features and/or feature pairs in JSON format. These can be used to generate partial dependence plots. This option is available when: • Training along with (model save OR testing) • Tuning followed by training + (model save OR testing) **Usage**: `--partial_dependencies_out  dependencies.json`
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Note**: The PMML file that is produced requires that missing values are encoded as '`?`'. This is a reserved symbol for missing values, and as such, should not be used anywhere else in your data. **Usage**: `--pmml_out  income.pmml`
`--scores_out`	Optionally specify a file to store raw scores that have not been converted to labels, probabilities, or targets. This file can then be specified during training/tuning/testing. **Usage**: `--scores_out  income.test.offsets`

Table 43: *GBTR Output Data Options (continued)*

Command	Description

`--split_importances_buckets`	Optionally specify a number of buckets in split importance calculation. Must be > 1. This value defaults to 10.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split.importances.out \`
	`--split_importances_buckets 4`

`--split_importances_out`	Optionally specify a file to store split importances. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split_importances.out`

`--smart_search_restart_out`	When using smart search, optionally specify to save smart search data to a file. This file can then be specified when running additional smart search iterations.
	
	Usage:
	`--smart_search on \`
	`--smart_search_iterations 10 \`
	`--smart_search_restart_out income.data.restart`

`--targets_out`	Optionally specify a file to store computed targets. **Usage**: `--targets_out targets.gbtr`

`--test_point_variable_`
`importances_out`
	Optionally specify a file to store variable importances for each test point. Requires `--testing_in`.
	
	Usage:
	`--testing_in income.data.st \`
	`--test_point_variable_importances_out income.data.vimps`

`--variable_importances_out`	Optionally specify a file to store intrinsic variable importances. This output file includes a single column. The first row is the importances of the first column variable in the `--training_in` .st file, the second row is the importance of the second column, and so on. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--variable_importances_out variable_importances.out`

`--variable_importances_out_`
`as_json`
	This option specifies to write variable importances as a JSON file. The output will include feature names, attribute names, and column IDs along with the importance.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--variable_importances_out variable_importances.out`
	`--variable_importances_out_as_json on`

Table 43: *GBTR Output Data Options (continued)*

Command	Description

`--visualization_out`	Optionally specify a directory for model visualization (created if it does exist). Requires `--training_in` and one of `--testing_in` or `--model_out`. Cannot be used in conjunction with `--ensemble_size`.
	Usage:
	`--training_in  income.data.st \`
	`--testing_in  income.test.st \`
	`--visualization_out  income`

Table 44: *Tunable GBTR Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>.*

Command	Description
`--ensemble_size`	Use `--ensemble_size` GBTRs with sampled tables and randomly sampled split dimensions. **Usage**: `--ensemble_size  10`
`--leaf_nodes`	The number of leaf nodes in the tree (default: 0). Must be >=0. A value of 0 disables the parameter. Only one of `--tree_depth` or `--leaf_nodes` can be non-zero at a time. This option is deprecated. Use `--max_splits=<leaf_nodes-1>` instead. **Usage**: `--leaf_nodes  5,10,25`
`--learning_rate`	The learning rate. Must be > `0` and <= `1`. This value defaults to `0.1`. **Usage**: `--learning_rate  0.05:0.05:0.2`

`--log_clamp`	When `--loss_function` is `pslog`, `gmlog`, or `tdlog`, optionally specify the maximum absolute log value for any terminal node. Requires `--loss_function=pslog`, `--loss_function=gmlog`, or `--loss_function=tdlog`. This value must be > `0` and defaults to `20`.
	Usage:
	`--loss_function  pslog \`
	`--log_clamp  10`

`--max_splits`	Specify the number of splits in the tree. This value must be >=`0`. A value of `0` disables this option. Either this value or `--min_node_weight` must be > `0` if `--tree_depth` is explicitly set to 0. If both `--tree_depth` and `--max_splits` are > 0, then Skytree Server finds the best splits that exist within the depth level, up to `--max_splits`. This option cannot be used in conjunction with (deprecated) `--leaf_nodes`. **Usage**: `--max_splits  4`

***Table 44:** Tunable GBTR Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>. (continued)*

Command	Description
`--min_node_weight`	The minimum bound on the total weight in each leaf during tree building. This value defaults to 0. Either this value or `--max_splits` must be > 0 if `--tree_depth` is explicitly set to `0`. **Usage**: `--min_node_weight  4`
`--num_dimensions`	The number of dimensions to randomly sample at each tree node. This option is only used in conjunction with `--ensemble_size`. If omitted, all dimensions are used. **Usage**: `--num_dimensions  1:1:4`
`--num_trees`	The number of trees in the ensemble. Must be > `0`. **Usage**: `--num_trees  5:5:100`

`--regularization_bins`	The number of bins to use per dimension/attribute of the data. Used only in conjunction with `--regularization`. If omitted (with `--regularization=on`), `200` bins will be used.
	
	Usage:
	`--regularization  on \`
	`--regularization_bins  100`

`--sampling_ratio`	Per class sampling ratio. If `--sample_with_replacement=off`, then this option is mandatory.
	
	Usage:
	`--sampling_ratio  0.25 \`
	`--sampling_ratio  2`

`--tree_depth`	Specifies the depth to which each ensemble is built. If explicitly set to `0`, trees will be built to the fullest extent. In this case, you must specify a positive value for either `--max_splits` or `--min_node_weight`. If this value is not set, and values for `--max_splits` and `--min_node_weight` are likewise not set, then this value defaults to 3. If both `--tree_depth` and `--max_splits` are > 0, then Skytree Server finds the best splits that exist within the depth level, up to `--max_splits`. **Usage**: `--tree_depth  0:2:10`

Table 45: *Non-tunable GBTR Options*

Command	Description
`--approximate_quantiles`	This flag is enabled automatically when `lad` or `huber` is specified for `--loss_function`. It allows for an error tolerance of .001 for quantiles on loss functions. For example, if you specify a `--huber_loss_quantile` of .33, then quantiles ranging from .329 to .331 will also be included. **Usage**: `--approximate_quantiles  off`
`--cardinality_based_dimension_sampling`	Categorical dimensions with larger number of unique values in a given node will have a proportionately higher chance of being selected when sampling dimensions. Continuous features are treated as having one unique value. When `off`, all dimensions have an equal chance of being selected in dimension sampling. **Usage**: `--cardinality_based_dimension_sampling  off`
`--categorical_random_split_threshold`	If the number of categorical values exceeds this, then random splitting is attempted. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_random_split_threshold  5`
`--categorical_random_split_tries`	The number of times to try random categorical splits. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_random_split_tries  8`
`--categorical_sampling_seed`	Optionally specify the categorical values sampling random number seed. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_sampling_seed  4`
`--categorical_selection_method`	Optionally specify one of the following: • `random` • `random_exact` • `one_vs_all` • `one_vs_all_random` • `one_vs_all_exact` • `exact` (default) Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_selection_method  random`

***Table 45:** Non-tunable GBTR Options (continued)*

Command	Description
`--compression`	Specify whether to use compression for data. Enabling this (default) reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`
`--dimension_sampling_seed`	The dimension sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--dimension_sampling_seed  10`

`--huber_loss_quantile`	When `--loss_function=huber`, optionally specify the top percentile of error that should be considered as outliers. Requires `--loss_function=huber`. This value must be between `0` and `1` and defaults to `0.9`.
	Usage:
	`--loss_function  huber \`
	`--huber_loss_quantile  0.8`

`--limit_parameters`	When `--smart_search` is enabled, this option restricts the parameter space in order to reduce the training time. This option is enabled by default.
	Usage:
	`--smart_search on`
	`--limit_parameters  off`

`--loss_function`	Optionally specify a loss function method to use during regression. Available values include the following: • `lad`: Perform least absolute deviation (LAD) regression (default) • `ls`: Perform least-squares regression • `huber`: Perform Huber loss regression • `pslog`: Perform Poisson-log regression with positive integral targets (such as counts) • `gmlog`: Perform Gamma-log regression with positive continuous targets • `gminv`: Perform Gamma-inverse regression with positive continuous targets • `tdlog`: Perform Tweedie-log regression. When `tdlog` is specified, the `--tweedie_exponent` option must also be specified. Users can also tune over the `--tweedie_exponent` option. **Usage**: `--loss_function  ls`

Table 45: *Non-tunable GBTR Options (continued)*

Command	Description
`--num_cached_trees`	Optionally specify the number of simultaneously cached trees in the ensemble. Higher values lead to higher memory usage. Lower values result in lower memory usage at the expense of increased runtime. This value defaults to the number of available threads. **Usage**: `--num_cached_trees  6`
`--regularization`	Use the fast decision tree construction heuristic. Multiple nodes can be used when this option is enabled without the need for additional tuning parameters. This value defaults to `on` for `gbt/gbtr`; it defaults to `off` for ensemble `gbt/gbtr`. **Usage**: `--regularization  off`

`--sample_with_replacement`	For each tree, sample the training points using replacement (bootstrap). This option is on by default. If turned `off`, `--sampling_ratio` must be provided.
	
	Usage:
	`--sample_with_replacement  off \`
	`--sampling_ratio  0.25`

`--table_sampling_seed`	The data sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--table_sampling_seed  1359937539`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Specify one of the following: • `mean_absolute_error` (default) • `mean_squared_error` • `coeff_determination` • `normalized_gini` **Usage**: `--testing_objective  mean_squared_error`

`--tweedie_exponent`	When `--loss_function=tdlog`, specify a value for the exponent. Requires `--loss_function=tdlog`. This option has no default value and, therefore, must be explicitly specified when configuring the Tweedie loss function. This value must be > `1.0` and < `2.0`. If users desire a value of `1`, they should use Poisson distribution (`pslog`). Similarly, if users desire an exponent equal to `2.0`, they should specify Gamma distribution (`gmlog`).
	
	Note that users can also tune over this option with values > `1.0` and < `2.0`. When used for tuning, the output CSV will have an extra column for this value.
	
	Usage:
	`--loss_function  tdlog \`
	`--tweedie_exponent  1.5`

Table 46: *General GBTR Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to `off`. **Usage**: `--fast_read  on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file  input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

Table 46: *General GBTR Options (continued)*

Command	Description
`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads 16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog off`
`--watchdog_high_load_ threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold 1`
`--watchdog_low_memory_ threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold 1`

Hadoop Data Preparation Options

The following tables show the options available in `etl-hadoop.sh`.

Table 47: *Hadoop Data Preparation Options*

Command	Description
`-allow_reordering`	Allow the output file and optional labels and targets files to not respect the order of input files. **Usage**: `-allow_reordering`
`-categorical_number`	Comma-delimited list of column numbers or names, a ranges of columns, or a range of columns with a step size. When specified, numeric values within those columns are considered as categorical numbers. (For example, 0100, 100, and `100.0` are treated as the same value.) **Usage**: `-categorical_number 3,5`

***Table 47:** Hadoop Data Preparation Options (continued)*

Command	Description
`-categorical_text`	Comma-delimited list of column numbers or names, a ranges of columns, or a range of columns with a step size. When specified, numeric values within those columns are considered as categorical text. (For example, `0100`, `100`, and `100.0` are treated as distinct values.) **Usage**: `-categorical_text 4,6`
`-categorical_warn_pct`	Warn if the number of categories exceeds this percentage of items. **Usage**: `-categorical_warn_pct .20`

`-create_labels_file`	Creates a labels file for classification. When used, a `-label_index` value must also be specified.
	
	Usage:
	`-create_labels_file`
	`-label_index 1`

`-create_targets_file`	Creates a targets file for regression. When used, a `-target_index` value must also be specified.
	
	Usage:
	`-create_targets_file`
	`-target_index 1`

`-delimiter`	Specify a character that separates columns, or specify one of `TAB`, `COMMA`, `SEMICOLON`, `VBAR`, or `SPACE`. Auto-detected if not specified. Note that Skytree Server allows you to add quotations around text so that it will not be confused as a delimiter (for example, when the delimiter is specified as `COMMA`, and an entry includes a deliberate comma). **Note:** Because multiple spaces can be used to separate columns, it is necessary to explicitly denote empty column values with `""` when using `-delimiter SPACE`. **Usage**: `-delimiter TAB`
`-header_input_dir`	Specify the directory containing the precomputed schema (such as a previous run's `-output_dir`). **Usage**: `-header_input_dir etl_out`
`-horizontalize`	If enabled, this creates a column for each possible value of a categorical feature. **Usage**: `-horizontalize on`
`-id_index`	Specify a column number or name indicating the row identifier. **Usage**: `-id_index 2`

***Table 47:** Hadoop Data Preparation Options (continued)*

Command	Description
`-ignore_columns`	A comma-separated list of column numbers, column names, or ranges of these to be ignored. **Usage**: `-ignore_columns  2,6-9,13`
`-ignore_constant_columns`	Specify whether to ignore columns with a single value for all rows. **Usage**: `-ignore_constant_columns  on`
`-ignore_missing`	If enabled, rows with missing values will be ignored. This value defaults to `off`. **Usage**: `-ignore_missing  on`

`-input_dir`	Specify one more input directories. When specifying multiple directories, an `-input_dir` option must be included for each input directory. Comma-separated lists are not supported. In addition, if header lines exist in the input directories, then all files in the input directories must have identical header lines.
	Usage:
	`-input_dir  data1_dir`
	`-input_dir  data2_dir`

`-label_index`	The column number or name indicating the classification label. For example, a value of `15` identifies column 15 as the label index. Label indices start at 1. **Usage**: `-label_index  10`
`-max_bad_lines_per_mapper`	Treats unrecognized and/or unparsable values as if they were missing values. **Usage**: `-max_bad_lines_per_mapper  10`
`-missing_value`	A string signifying a missing value. This ensures that Skytree Server treats these values as missing rather than as text data. **Usage**: `-missing_value  ?`
`-mean_impute`	When enabled, missing values for numerical features are replaced with mean values, and missing values for categorical features are treated as distinct values. This value defaults to `on`. **Usage**: `-mean_impute  off`
`-normalize`	The normalization method to use. Specify one of the following: • `unit`: This form makes the data range between 0 and 1. • `standard`: This gives the data a mean value of 0 and unit variance. **Usage**: `-normalize  unit`

Table 47: Hadoop Data Preparation Options (continued)

Command	Description
`-num_sparse_input_columns`	Specify the number of sparse columns included in libsvm-formatted data. **Usage**: `-num_sparse_input_columns  4`
`-output_dir`	The directory that stores the output files. When including a single input directory, Skytree Server creates a file `output.st` in that directory. When including multiple input files, the output will produce `.st` files with basename prefixes that mirror the input directories' basenames. Note that this file name must not already exist. **Usage**: `-output_dir  etl_out`
`-rare_words_pct`	Filter words less frequent than this percentage. This value defaults to `0.0`. **Usage**: `-rare_words_pct  10`
`-regard_case`	Specify whether to regard the case of textual data. For example, if this option is enabled, then USD and `usd` will be treated the same. **Usage**: `-regard_case  on`
`-skip_bad_lines`	Specifies to skips all bad lines encountered during the data conversion process. This is similar to `-ignore_missing`, but this option still allows literal missing values to be imputed. **Usage**: `-skip_bad_lines  on`
`-sparse_columns`	A comma-separated list of column numbers, column names, or ranges of columns containing sparse data. An `all_columns` keyword can also be specified. **Usage**: `-sparse_columns  11,12`
`-sparse_suggest_pct`	Suggest columns be made sparse if ratio of zeros or missing values exceeds this percentage. For example, if this value is `50`, then the output will indicate any columns for which the ratio of zero to non-zero values exceeds 50 percent. **Usage**: `-sparse_suggest_pact  30`
`-stop_words_pct`	Filter words more frequent than this percentage. This value defaults to `100.0`. **Usage**: `-stop_words  10`
`-target_index`	The column number or column name of the regression target. **Usage**: `-target_index  1`

Table 47: *Hadoop Data Preparation Options (continued)*

Command	Description
`-trailing_delimiter`	This flag signals that each row has a trailing delimiter. This can be automatically determined unless missing values are indicated by the empty string. **Usage**: `-trailing_delimiter  on`
`-use_column_names`	Use column names from the first line of the first input file. This implies that `-ignore_lines` for that file is at least 1. **Usage**: `-use_column_names  on`
`-use_column_weights`	Use a comma-delimited weights file in the input director to assign weights to columns. If this option is specified, then all input directories must have a "`_weights`" file. **Usage**: `-use_column_weights  on`
`-words_columns`	For each possible word in the specified column, this parameter creates a notional column in the output `.st` file. The entry will be `1` if the word occurs in a row, and `0` if it does not. Because most rows don't contain any particular word, the columns are always made sparse. (That is, there is one entry for each word that occurs; otherwise the column is not represented in the output.) **Usage**: `-words_columns  6`

Kernel Density Estimation Options

The following table shows the options available in the kde module.

Table 48: *KDE Options*

Command	Description
`--bandwidth`	REQURED. Specify the kernel bandwidth. You must provide either the same number of `--bandwidth` options as classes in `--labels_in`, or just one bandwidth to be used for all classes. **Usage**: `--bandwidth  0.75`
`--densities_out`	Optionally specify a file to store computed densities. **Usage**: `--densities_out  random.densities`
`--kernel`	Specify the function used by KDE/KDA. Enter either `epan` (default) or `gaussian`. **Usage**: `--kernel  gaussian`

***Table 48:** KDE Options (continued)*

Command	Description
`--labels_in`	Optionally specify the file containing reference labels. **Usage**: `--labels_in  reference.labels.st`
`--labels_out`	Optionally specify a file to store classification results when running KDA. **Usage**: `--labels_out  reference.labels`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--prior`	Optionally specify priors for use with KDA. Include as many `--prior` option as there are classes in `--labels_in`. Otherwise, if omitted, KDA assumes priors equal to class ratios. **Usage**: `--prior  2`
`--probabilities_out`	Optionally specify a file to store computed class probabilities when running KDE. **Usage**: `--probabilities_out  probabilities`
`--queries_in`	Optionally specify a file containing query positions. If omitted, KDE/KDA works in "leave-one-out" mode, where every point of the reference set is treated as a query point and its density is computed by excluding the contribution of the point itself. **Usage**: `--queries_in  random_queries.st`
`--references_in`	REQUIRED. Specify the file containing reference data. **Usage**: `--references_in  random_1kx6.st`
`--relative_error`	Maximum relative error for the approximation of KDE/KDA. **Usage**: `--relative_error  0.1`

k-means Options

The following tables show the options available in the kmeans module.

***Table 49:** k-means Input Data Options*

Command	Description

`--centroids_in`	Specify the initial set of centroids for `--run_mode=train`, or centroids for `--run_mode=eval` (in which case, it is required)
	Usage:
	`--run_mode eval \`
	`--centroids_in centroids.st`

`--queries_in`	REQUIRED when `--run_mode=eval`. Specify the file containing the data to be assigned to given clusters.
	Usage:
	`--run_mode eval \`
	`--queries_in sdss.test.st`

`--references_in`	REQUIRED when `--run_mode=train`. Specify the file containing data to be clustered.
	Usage:
	`--run_mode train \`
	`--references_in 3gaussians.st`

***Table 50:** k-means Output Data Options*

Command	Description
`--centroids_out`	Optionally specify a file to store cluster means. This cannot be used in conjunction with `--k_min` and `--k_max`. **Usage**: `--centroids_out centroids.st`

`--coreset_out`	REQUIRED with `--run_mode=coreset`. The coreset will be output to this file. Note that in this output, the points from the table retrain their IDs. Generated points, however, will have an ID of `-1`, indicating that they are not actual points in the dataset.
	Usage:
	`--run_mode coreset \`
	`--coreset_out cores.st`

`--coreset_size`	Can be used with `--run_mode=coreset`. This option specifies the desired size of the coreset output in terms of a fraction of the size of the original data. The coreset size should be > `0` and < `1`. This value defaults to `0.01`.
	Usage:
	`--run_mode coreset \`
	`--coreset_out cores.st \`
	`--coreset_size 0.05`

***Table 50:** k-means Output Data Options (continued)*

Command	Description
`--distortions_out`	Optionally specify a file to store the SQUARED distance to the closest centroid for every point. This cannot be used in conjunction with `--k_min` and `--k_max`. **Usage**: `--distortions_out  distortions.st`
`--memberships_out`	Optionally specify a file to store cluster memberships. This cannot be used in conjunction with `--k_min` and `--k_max`. **Usage**: `--memberships_out  assignments.st`

`--output_with_ids`	If enabled, the per-point output for `--memberships_out` and `--distortions_out` will be prepended with the input file's "id" meta field followed by a comma. For example, if the original output is "`a,b,c`" then the new output would be "`ID,a,b,c`" where "`ID`" is an integer (`1, -1`). If an "`id`" meta field is not available in the input, then "`0`" is set as the input "`id`" field (for example, "`0,a,b,c`").
	This option is disabled by default and requires `--memberships_out` and/or `--distortions_out`.
	Usage:
	`--memberships_out  assignments.st`
	`--output_with_ids  on`

`--progressfile`	Specify a file to report the progress in percents in. This option can only be used in non-distributed mode. **Usage**: `--progressfile  DataCluster`

***Table 51:** k-means Options*

Command	Description
`--algorithm`	Algorithm used to compute clusters. Specify one of the following: • `online`: online k-means (single host only) • `online_fast`: online k-means followed by fast k-means (single host only) • `online_lloyds`: online k-means and followed by Lloyd's algorithm (single host only) • `fast`: fast k-means (multiple hosts allowed) (default) • `lloyds`: Lloyd's algorithm (multiple hosts allowed) • `online_naive`: deprecated, use `online_lloyds` • `naive`: deprecated, use `lloyds` **Usage**: `--algorithm  online_fast`

Table 51: *k-means Options (continued)*

Command	Description
`--clust_movement_thresh`	If in any iteration the largest cluster motion is less than this absolute threshold, k-means will terminate. A value of `0.0` (default) means that k-means will run until full convergence is reached (typically to a local minimum). **Usage**: `--clust_movement_thresh  0.05`
`--compression`	Specify whether to use compression for data. This option reduces computational resource requirements. This value defaults to on. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`

`--epochs`	For `--algorithm=online*`, specify the number of passes through the entire dataset. This value defaults to `1`.
	
	Usage:
	`--algorithm  online_fast \`
	`--epochs  4`

`--initialization`	Choose one of the following options to initialize the centroids: • `random`: pick random points from the dataset (default) • `kmeans++`: compute initial centroids with kmeans++ **Usage**: `--initialization  kmeans++`
`--initialization_seed`	Random number generator seed used for centroid initialization. If omitted, a time-based seed will be used. Results are only reproducible when the same number of hosts is used. **Usage**: `--initialization_seed  2`
`--iterations`	K-means can run in either batch or progressive mode. Use this option to restrict the number of iterations of the batch mode. If `--iterations=i` is omitted, k-means finds exact clusters; otherwise, it terminates after `i` progressive refinements. **Usage**: `--iterations  2`
`--k_clusters`	Specify the number of clusters for k-means. This value must be greater than `1`. To scan for optimum number of clusters, you can use `--k_min` and `--k_max` instead of `--k_clusters`. **Usage**: `--k_clusters  10`

***Table 51:** k-means Options (continued)*

Command	Description

`--k_max`	Specify the maximum number of clusters when scanning for the optimal number of clusters. Must be used in conjunction with `--k_min` and be greater than `--k_min`. This option cannot be used with multiple hosts. This option also cannot be used with `--memberships_out` or `--distortions_out`. If `--k_min` and `--k_max` are not set, then `--k_clusters` will be used.
	
	Usage:
	`--k_min  2 \`
	`--k_max  10 \`
	`--k_step 1`

`--k_min`	Specify the minimum number of clusters when scanning for optimal number of clusters. Must be used in conjunction with `--k_max` and be greater than `1`. This option cannot be used with multiple hosts. This option also cannot be used with `--memberships_out` or `--distortions_out`. If `--k_min` and `--k_max` are not set, then `--k_clusters` will be used.
	
	Usage:
	`--k_min  2 \`
	`--k_max  10 \`
	`--k_step 1`

`--k_step`	Step size for number of clusters when scanning for optimal number of clusters. Must be greater than `0`. This option is used only in conjunction with `--k_min` and `--k_max`. This option defaults to 1.
	
	Usage:
	`--k_min  2 \`
	`--k_max  10 \`
	`--k_step 1`

`--n_restarts`	Specify the number for restarts per k-means training. This can lower the distortion when random initialization of centroids is used. This value defaults to `1`. **Usage**: `--n_restarts  10`

`--online_batch_size`	For `--algorithm=online*`, specify the number of points read in each epoch. If not provided, then the entire dataset is used.
	
	Usage:
	`--algorithm  online_lloyds \`
	`--onlinebatch_size  4`

`--percentage_hold_out`	Specify the percentage of the dataset held out as a validation set when searching for the best number of clusters using `--k_min` and `--k_max`. The value of this percentage corresponds to a fraction and should be between `0` and `1`. This value defaults to `0.2`.
	
	Usage:
	`--k_min  2 \`
	`--k_max  10 \`
	`--k_step 1 \`
	`--percentage_hold_out  0.25`

`--probability`	This option is deprecated and will have no effect. It may be removed in a future release.

Table 51: *k-means Options (continued)*

Command	Description
`--randomize`	Specify whether to randomize the input point order. For k-means in online mode, it is important to have randomized (`i.i.d`) data. You can safely turn this flag off if you know that your data is already `i.i.d`. This value defaults to `on`. **Usage**: `--randomize  off`
`--run_mode`	Specify one of the following for the operating mode. • `train`: find optimal clusters for given reference points and number of clusters (default) • `eval` : find cluster memberships for given query points and centroids • `coreset` : generate a coreset of the given reference points **Usage**: `--run_mode  eval`

Table 52: *General k-means Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to off. **Usage**: `--fast_read  on`

`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. This option is available for `--algorithm=fast` (default) and `--algorithm=lloyds`.
	
	Usage:
	`--algorithm  lloyds \`
	`--hosts  localhost,127.0.0.1`

`--input_file`	If given, load input options from this file. **Usage**: `--input_file  input`
`--log`	If given, write log to this file instead of stdout. **Usage**: `--log  log_run`

Table 52: General k-means Options (continued)

Command	Description
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • verbose: log everything • default: log messages and warnings • warning: log only warnings • silent: no logging **Usage**: `--loglevel  verbose`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_ threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to 1.5. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_ threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to 0.05. **Usage**: `--watchdog_low_memory_threshold  1`

Linear Regression Options

The following table shows the options available in the `linear` module.

***Table 53:** Linear Regression Options*

Command	Description
`--adj_r_squared_out`	Specify the output file for the adjusted r-squared value. **Usage**: `--adj_r_squared_out  adjusted_r_squared.st`
`--algorithm`	The type of algorithm to use. Specify either `fast` (default) or `naive`. **Usage**: `--algorithm  naive`
`--coeffs_in`	Specify the input file for the coefficients. **Usage**: `--coeffs_in  coefficients.st`

`--coeffs_in_bias_term_index`	When `--run_mode=eval` this tells the index position of the bias term if present in the input coefficients. Set to `-1` if not present.
	
	Usage:
	`--run_mode  eval \`
	`--coeffs_in_bias_term_index  -2`

`--coeffs_out`	The output file for the coefficients. **Usage**: `--coeffs_out  coefficients_out.st`
`--conf_his_out`	Specify the output file for the upper bound for the confidence intervals. **Usage**: `--conf_his_out  confidence_band_highs.st`
`--conf_los_out`	Specify the output file for the lower bound for the confidence intervals. **Usage**: `--conf_los_out  confidence_band_lows.st`
`--conf_prob`	Specifies the probability coverage of the confidence intervals. This value defaults to `0.9`. **Usage**: `--conf_prob  0.5`
`--correlation_pruning`	If enabled, the correlation coefficient should be used for initial pruning. This option is disabled by default. **Usage**: `--correlation_pruning  on`
`--correlation_threshold`	During the correlation pruning, each attribute will be pruned if it has an absolute correlation factor greater than this value with any other attributes. This option defaults to `0.9`. **Usage**: `--correlation_threshold  0.7`

Table 53: Linear Regression Options (continued)

Command	Description
`--exclude_bias_term`	If enabled, the bias term will not be included in the linear model. This option is disabled by default. **Usage**: `--exclude_bias_term  on`
`--f_statistic_out`	Specify the output file for the f-statistics. **Usage**: `--f_statistics_out  f_statistic`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--output_statistics`	Specify whether to output the statistics that describe the quality of the model. This option is disabled by default. **Usage**: `--output_statistics  on`
`--p_values_out`	Specify the output file for the p values. **Usage**: `--p_values_out  p_values.st`
`--prediction_index`	Specify the index of the column containing prediction targets. This value defaults to `0`. In this case, the first attribute is used. **Usage**: `--prediction_index  1`

`--predictions_out`	When `--run_mode=eval`, this file will contain the predictions on the query data.
	Usage:
	`--run_mode  eval \`
	`--predictions_out  predictions.st`

Table 53: *Linear Regression Options (continued)*

Command	Description

`--queries_in`	Specify the data file containing the test set to be evaluated. This is REQUIRED when `--run_mode=eval`.
	Usage:
	`--run_mode eval \`
	`--queries_in random_1kx6.st`

`--r_squared_out`	Specify the output file for the r-squared value. **Usage**: `--r_squared_out r_squared.st`

`--references_in`	Specify the data file containing the predictors and the predictions. This is REQUIRED when `--run_mode=train`, which is the default for `--run_mode`.
	Usage:
	`--run_mode train \`
	`--references_in sample_promo.st`

`--run_mode`	Specify one of the following options for the operating mode.
	• `train`: find optimal set of linear regression coefficients (default)
	• `eval`: given coefficients, predict on a set of query points specified by `--queries_in`
	Usage:
	`--run_mode eval \`
	`--queries_in random_1kx6.st`

`--sigma_out`	Specify the output file for the sigma. **Usage**: `--sigma_out sigma.st`
`--std_errors_out`	Specify the output file for the standard errors. **Usage**: `--std_errors_out std_errors.st`
`--stepdirection`	The direction of stepwise regression. Specify one of the following: • `bidir`: use bidirectional stepwise optimization (default) • `forward`: use forward stepwise optimization • `backward`: use backward stepwise optimization **Usage**: `--stepdirection forward`
`--stepwise`	If enabled, the stepwise regression should be used after VIF selection. This option is disabled by default. **Usage**: `--stepwise on`

Table 53: *Linear Regression Options (continued)*

Command	Description
`--stepwise_threshold`	Specify a minimum required improvement in the score needed during stepwise regression in order to continue the selection. This value defaults to 0. **Usage**: `--stepwise_threshold  0.5`
`--t_values_out`	Specify the output file for the t values. **Usage**: `--t_values_out  t_values.st`
`--vif_threshold`	During the pruning stage via variance inflation factor, each attribute will be pruned if the threshold exceeds this value. This value defaults to 8. **Usage**: `--vif_threshold  5.0`

Logistic Regression Options

The following tables show the options available in the `logistic` module.

Table 54: *Logistic Regression Options*

Command	Description
`--coefficients_in`	Specify the input file for coefficients when not training. **Usage**: `--coefficients_in  coeffs.st`
`--coefficients_out`	Specify the output file for coefficients when training. **Usage**: `--coeffecients_out  coeffs`
`--exclude_bias_term`	Specify whether an intercept term (or bias term) is to be used in the regression. This option is disabled by default. **Usage**: `--exclude_bias_term  on`
`--iterations`	Specify the number of iterations to perform. Must be >= 1. If this option is not specified, then the process will iterate until convergence. **Usage**: `--iterations  2`
`--labels_out`	Specify the output file for labels for the test dataset. **Usage**: `--labels_out  sdss.test.labels.out`

***Table 54:** Logistic Regression Options (continued)*

Command	Description
`--probabilities_out`	Specify the output file for probabilities for the test dataset. **Usage**: `--probabilities_out  sdss.test.probabilities`
`--regularization`	Skytree Server logistic regression is L1-regularized. The default value for the regularization term is set to $1e-6$. Use this option to change the regularization value. Specify the regularization parameter. Higher values can be used to ensure under-fit models that may suit the problem better. But note that altering the regularization term will penalize the L1-norm of the coefficients in the logistic regression. **Usage**: `--regularization  0.001`
`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  sdss.test.st`
`--training_in`	Specify a file containing the training data. **Usage**: `--training_in  sdss.train.st`
`--training_labels_in`	Specify a file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`

***Table 55:** General Logistic Regression Options*

Command	Description
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`

Table 55: General Logistic Regression Options (continued)

Command	Description
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold 1`
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold 1`

Network Tester Options

The following table shows the options available in the `network-tester` module.

Table 56: Network Tester Options

Command	Description
`--blocking`	Optionally specify whether to use blocking MPI send/recv. This value defaults to `on`. **Usage**: `--blocking off`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log log_run`

Table 56: *Network Tester Options (continued)*

Command	Description
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--msg_size`	Optionally specify the message size. This option defaults to `1048576`. **Usage**: `--msg_size  2097152`
`--reps`	Optionally specify a number of repetitions. This value defaults to `100`. **Usage**: `--reps  200`
`--sleep`	Optionally specify the time in seconds to sleep between repetitions. This option is turned off (`--sleep=0`) by default. **Usage**: `--sleep  2`

Nearest Neighbors Options

The following table shows the options available in the nn module.

Table 57: *NN Options*

Command	Description
`--algorithm`	Specify the algorithm to find nearest neighbors. Specify `naive`, `fast`, or `fastest` (default). **Usage**: `--algorithm  naive`

Table 57: NN Options (continued)

Command	Description
`--distances_out`	Optionally specify a file to store found neighbor distances (not the squared distances). **Usage**: `--distances_out distances`
`--fast_read`	If set, disable verbose input checks for faster file reads. This option is disabled by default. **Usage**: `--fast_read on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts localhost,127.0.0.1`
`--indices_out`	Optionally specify a file to store found neighbor indices **Usage**: `--indices_out neighbors`
`--k_neighbors`	REQUIRED. Specify the number of neighbors to use for scoring/classification. **Usage**: `--k_neighbors 5`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel verbose`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts localhost,127.0.0.1 \`
	`--procs_per_host 3`

Table 57: *NN Options (continued)*

Command	Description
`--queries_in`	Optionally specify a file containing nearest neighbor search queries. If omitted, `--references_in` is used. **Usage**: `--queries_in  random_1kx6.st`
`--references_in`	REQUIRED file containing the input data. **Usage**: `--references_in  random_1kx6.st`
`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_ threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_ threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Nearest Neighbors Classification Options

The following table shows the options available in the nnc module.

Table 58: *NNC Options*

Command	Description
`--algorithm`	Specify the algorithm to find nearest neighbors. Specify `naive`, `fast`, or `fastest` (default). **Usage**: `--algorithm  naive`

Table 58: *NNC Options (continued)*

Command	Description

`--alpha`	Specify an optional approximation parameter 1 when performing alpha-beta approximation. This option specifies the accuracy and must be in the range (`0, 1`). Larger `--alpha` and `--beta` will improve accuracy somewhat, though smaller values are key for speed.
	
	Usage:
	`--alpha  0.2 \`
	`--beta  5`

`--alpha_beta_random_seed`	Optionally specify the random number seed used for alpha-beta approximation. This option is used only in conjunction with `--alpha`. If omitted, the current time is used.
	
	Usage:
	`--alpha  0.2 \`
	`--beta  5 \`
	`--alpha_beta_random_seed  123`

`--beta`	Specify an optional approximation parameter 2 when performing alpha-beta approximation. This parameter specifies how many samples are taken and must be at least `1`. Larger `--alpha` and `--beta` will improve accuracy somewhat, though smaller values are key for speed.
	
	Usage:
	`--alpha  0.2 \`
	`--beta  5`

`--classification_objective`	Optionally specify an objective for classification threshold tuning, either '`accuracy`' or '`fscore`' (default). This option cannot be used in conjunction with `--probability_threshold`. **Usage**: `--classification_objective  accuracy`

`--classweight`	Optionally specify the class weight for use with nearest neighbor classification. This option impacts the scores and probabilities. You must repeat the `--classweight` option for each class in `--training_labels_in`. If omitted, class weights of `1` are assumed for every class.
	
	Usage:
	`--training_labels_in  sdss.train.labels \`
	`--classweight  10 \`
	`--classweight  1`

`--compression`	Specify whether to use compression for data. Enabling this (default) reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`

Table 58: NNC Options (continued)

Command	Description
`--curve`	Specifies the type of curve used for AUC/Gini calculation. Specify one of the following: • `roc`: Receiver Operations Curve (ROC) (default): ▪ x-axis: false positive rate ▪ y-axis: true positive rate (capture rate) • `lorenz`: Lorenz curve. ▪ x-axis: percentile ▪ y-axis: true positive rate (capture rate) **Usage**: `--curve  lorenz`
`--distances_out`	Optionally specify a file to store found neighbor distances (not the squared distances). **Usage**: `--distances_out  distances`

`--explore_persistence`	Optionally specify a value to set the persistence of the explore method. This is only used in conjunction with `--metric_learning_method=explore`. This value defaults to `0.2` and must be between `0` and `1`.
	Usage:
	`--metric_learning_method  explore \`
	`--explore_scales_in  explore_scales \`
	`--explore_persistence  0.3`

`--explore_scales_in`	Optionally specify a file containing scaling factors for each dimension. Only used in conjunction with `--metric_learning_method=explore`.
	Usage:
	`--metric_learning_method  explore \`
	`--explore_scales_in  explore_scales \`
	`--explore_persistence  0.3`

`--fast_read`	If set, disable verbose input checks for faster file reads. This option is disabled by default. **Usage**: `--fast_read  on`
`--fit_method`	Specify the fitting method used for `--fit_metric_weights` and/or `--metric_learning_method={forward_fit,backward_fit}`. Set to `1` (default) or `2`. **Usage**: `--fit_method  2`
`--fit_metric_weights`	If set, fit initial metric weights. This option is disabled by default. **Usage**: `--ft_metric_weights  on`

Table 58: *NNC Options (continued)*

Command	Description

`--fit_metric_weights_out`	Optionally specify a file for outputting the fit metric weights. This is used only in conjunction with `--fit_metric_weights`.
	
	Usage:
	`--fit_metric_weights  on \`
	`--fit_metric_weights_out  fitweights.csv`

`--fit_reg`	Control parameter for fit method. Use a small positive number between 0 and 1. This value is set to $1e-6$ by default. This is only used in conjunction with `--fit_method=2`.
	
	Usage:
	`--fit_method  2 \`
	`--fit_reg  0.02`

`--fit_threshold`	Specify an optional threshold value for pruning fitted metric weights. The fit threshold prunes dimensions that seem unhelpful by weighting them to 0. Lower values are more aggressive. Set to `0` to disable. By default `--fit_threshold=8.0` for `--fit_metric_weights` but is disabled for `--metric_learning_method=forward_fit`. This option is only used when `--fit_method=1`.
	
	Usage:
	`--fit_method  1 \`
	`--fit_metric_weights  on \`
	`--fit_threshold  4`

`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--indices_out`	Optionally specify a file to store found neighbor indices **Usage**: `--indices_out  neighbors`
`--k_neighbors`	REQUIRED. Specify the number of neighbors to use for scoring/classification. **Usage**: `--k_neighbors  5`
`--labels_out`	Optionally specify a file to output the computed classification labels. **Usage**: `--labels_out  labels`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`

Table 58: *NNC Options (continued)*

Command	Description
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--memory_usage`	Specify the maximum fraction of system memory to use in the range [`0.0,1.0`]. If set to `0.0`, automatic memory management is disabled. By default, Skytree Server makes an effort not to exceed 75% of its host system's memory, breaking computation into smaller chunks if necessary. **Usage**: `--memory_usage  0.80`
`--metric_learning`	Specify whether to perform metric learning. This option is disabled by default. **Usage**: `--metric_learning  on`

`--metric_learning_iterations`	Specify the number of iterations for metric learning. This value defaults to `10` and is only used for `--metric_learning_method={random,explore}`.
	Usage:
	`--metric_learning  on \`
	`--metric_learning_method  random \`
	`--metric_learning_iterations  20 \`

`--metric_learning_method`	Specify the method used for metric learning. This is only used in conjunction with `--metric_learning`. Specify one of the following:
	• `forward`
	• `forward_fit` (default)
	• `backward`
	• `backward_fit`
	• `explore`
	• `random`
	Usage:
	`--metric_learning  on \`
	`--metric_learning_method  explore`

Table 58: *NNC Options (continued)*

Command	Description

`--metric_learning_min_dim_ratio`	Parameter for `--metric_learning_method={backward,backward_fit}`. Specifies the fractional dimensionality at which to end `backward/backward_fit` metric learning. This option defaults to `0` and must be in the range (`0-1`). A value of `0.5`, for example, would specify that the lowest dimensionality to be tried is half the number of full dimensions.
	
	Usage:
	`--metric_learning on \`
	`--metric_learning_method backward \`
	`--metric_learning_min_dim_ratio 0.5`

`--metric_learning_objective`	Specify one of the following objectives for tuning:
	• `gini` (default)
	• `fscore`
	• `accuracy`
	• `capturedev`
	
	Usage:
	`--metric_learning on \`
	`--metric_learning_objective fscore`

`--metric_learning_random_seed`	Optionally specify a random number seed used for metric learning. This is used only in conjunction with `--metric_learning_method={random,explore,forward_fit}`. For `forward` and `forward_fit`, this parameter is only used if `--metric_learning_min_dim_ratio`>`0`. If omitted, the current time is used.
	
	Usage:
	`--metric_learning on \`
	`--metric_learning_method forward_fit \`
	`--metric_learning_min_dim_ratio 0.5 \`
	`--metric_learning_random_seed 123`

`--metric_weights_in`	Optionally specify a file containing the metric weights. Must have as many values as there are dimensions. **Usage**: `--metric_weights random.csv`

`--optimal_metric_weights_out`	Optionally specify a file for outputting the optimal metric weights. Only used in conjunction with `--metric_learning`.
	
	Usage:
	`--metric_learning on \`
	`--optimal_metric_weights_out back.csv`

`--percentiles_in`	Optionally specify a file containing a single line with percentile numbers (e.g., `0.5 1 5 10 20 30 50 100`) to be used as sampling positions for AUC calculations from the ROC/Lorenz curve. A low percentile value refers to high scores. If omitted, all data points are used for integration, yielding the most accurate results. **Usage**: `--percentiles_in percentiles`

Table 58: *NNC Options (continued)*

Command	Description
`--probabilities_out`	Optionally specify a file to output the computed class probabilities. **Usage**: `--probabilities_out  probabilities.nnc`
`--probability_threshold`	Specify an optional probability to be used as the threshold for classification. Cannot be used in conjunction with `--classification_objective`. **Usage**: `--probability_threshold  0.82082089`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--rank_error_prob`	Optionally specify a probability (between `0` and `1`) with which the rank error is guaranteed. Can only be used in conjunction with `--rank_error_tol`.
	
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_prob  0.9`

`--rank_error_random_seed`	Optionally specify a random number seed used for rank approximation. Only used in conjunction with `--rank_error_tol`. If omitted, the current time is used.
	
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_random_seed  123`

`--rank_error_tol`	Optionally specify the rank error tolerance (as a fraction of all neighbors). Specify a value between `0` and `1`. The neighbors found will have a rank error of less than `--rank_error_tol` times the number of all neighbors, guaranteed with the probability given by `--rank_error_prob`. Larger values of `--rank_error_tol` will lead to greater speedups.
	
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_prob  0.9`

`--sample_with_replacement`	For each tree, sample the training points using replacement (bootstrap). This option is `on` by default. If turned off, `--sampling_ratio` must be provided.
	
	Usage:
	`--sample_with_replacement  off \`
	`--sampling_ratio  0.25`

Table 58: *NNC Options (continued)*

Command	Description

`--sampling_ratio`	Per class sampling ratio. You must either specify a single `--sampling_ratio` that will be used for each class, or repeat the `--sampling_ratio` option for each class in `--training_labels_in`. If omitted with `--sample_with_replacement=on`, full bootstrapping will be used. If `--sample_with_replacement=off`, then this option is mandatory.
	
	Usage:
	`--training_labels_in  income.labels \`
	`--sampling_ratio  0.25 \`
	`--sampling_ratio  2`

`--scores_out`	Optionally specify a file to output the computed classification scores. **Usage**: `--scores_out  scores.nn`
`--scoreweight`	Optionally specify the score weight for use with nearest neighbor classification. This value will affect the probabilities. You must repeat the `--scoreweight` option for each class in `--training_labels_in`. If omitted, score weights of `1` are assumed for every class. **Usage**: `--scoreweight  5`
`--smoothing`	Specify the probability smoothing parameter. Must be > `0`. Disabled if set to `0.0` (default). **Usage**: `--smoothing  0.001,0.01`
`--table_sampling_seed`	Optionally specify a random number seed used for table sampling. If omitted, the current time is used. **Usage**: `--table_sampling_seed  1359937539`
`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  sdss_test.st`
`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in  sdss.train.st`

Table 58: *NNC Options (continued)*

Command	Description
`--training_labels_in`	REQUIRED. Specify the file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_labels_in`	If `--tuning_in` is specified, then also specify a file containing the labels of the tuning data.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Nearest Neighbors Plus Options

The following table shows the options available in the `nnplus` module.

Table 59: *NN Plus Options*

Command	Description
`--algorithm`	Specify the algorithm to find nearest neighbors. Specify `naive`, `fast`, or `fastest` (default). **Usage**: `--algorithm  naive`

Table 59: *NN Plus Options (continued)*

Command	Description

`--alpha`	Specify an optional approximation parameter 1 when performing alpha-beta approximation. This option specifies the accuracy and must be in the range (0, 1). Larger `--alpha` and `--beta` will improve accuracy somewhat, though smaller values are key for speed.
	
	Usage:
	`--alpha 0.2 \`
	`--beta 5`

`--alpha_beta_random_seed`	Optionally specify the random number seed used for alpha-beta approximation. This option is used only in conjunction with `--alpha`. If omitted, the current time is used.
	
	Usage:
	`--alpha 0.2 \`
	`--beta 5 \`
	`--alpha_beta_random_seed 123`

`--beta`	Specify an optional approximation parameter 2 when performing alpha-beta approximation. This parameter specifies how many samples are taken and must be at least 1. Larger `--alpha` and `--beta` will improve accuracy somewhat, though smaller values are key for speed.
	
	Usage:
	`--alpha 0.2 \`
	`--beta 5`

`--compression`	Specify whether to use compression for data. Enabling this (default) reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`
`--distances_out`	Optionally specify a file to store found neighbor distances (not the squared distances). **Usage**: `--distances_out distances`
`--fast_read`	If set, disable verbose input checks for faster file reads. This option is disabled by default. **Usage**: `--fast_read on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts localhost,127.0.0.1`
`--indices_out`	Optionally specify a file to store found neighbor indices **Usage**: `--indices_out neighbors`

Table 59: *NN Plus Options (continued)*

Command	Description
`--k_neighbors`	REQUIRED. Specify the number of neighbors to use for scoring/classification. **Usage**: `--k_neighbors 5`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel verbose`
`--memory_usage`	Specify the maximum fraction of system memory to use in the range `[0.0,1.0]`. If set to `0.0`, automatic memory management is disabled. By default, Skytree Server makes an effort not to exceed 75% of its host system's memory, breaking computation into smaller chunks if necessary. **Usage**: `--memory_usage 0.80`
`--metric_weights_in`	Optionally specify a file containing the metric weights. Must have as many values as there are dimensions. **Usage**: `--metric_weights random.csv`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts localhost,127.0.0.1 \`
	`--procs_per_host 3`

`--queries_in`	Optionally specify a file containing nearest neighbor search queries. If omitted, `--references_in` is used. **Usage**: `--queries_in random_1kx6.st`

Table 59: *NN Plus Options (continued)*

Command	Description

`--rank_error_prob`	Optionally specify a probability (between 0 and 1) with which the rank error is guaranteed. Can only be used in conjunction with `--rank_error_tol`.
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_prob  0.9`

`--rank_error_random_seed`	Optionally specify a random number seed used for rank approximation. Only used in conjunction with `--rank_error_tol`. If omitted, the current time is used.
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_random_seed  123`

`--rank_error_tol`	Optionally specify the rank error tolerance (as a fraction of all neighbors). Specify a value between `0` and `1`. The neighbors found will have a rank error of less than `--rank_error_tol` times the number of all neighbors, guaranteed with the probability given by `--rank_error_prob`. Larger values of `--rank_error_tol` will lead to greater speedups.
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_prob  0.9`

`--references_in`	REQUIRED file containing the input data. **Usage**: `--references_in  random_1kx6.st`
`--threads`	Number of threads to use. This value defaults to `24`. **Usage**: `--threads  32`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Random Decision Forests Options

The following tables show the options available in the `rdf` module.

Table 60: *RDF Input Data Options*

Command	Description
`--batch_size`	Specify the batch size for streamed test points. A larger batch size can increase throughput but also increase latency. This value defaults to `1`. **Usage**: `--batch_size  2`

`--classweight`	Specify a value for classweights. When used, you must repeat the `--classweight` option for each class in `--training_labels_in`. The values are used to artificially inflate the impact of the corresponding class. If omitted, classweights of `1` are assumed for every class.
	Usage:
	`--training_labels_in  sdss.train.labels \`
	`--classweight  10 \`
	`--classweight  1`

`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in  model`
`--port`	Optionally specify a port number for streaming test data. Must be >= `1024` if specified. **Usage**: `--port  10000`
`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  sdss_test.st`
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in  sdss.train.st`
`--training_labels_in`	REQUIRED. Specify the file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`
`--training_point_weights_in`	Optionally specify a file containing the point weights (for training a model) for each point in `--training_in`. By default, each training point gets a point weight of `1`. **Usage**: `--training_point_weights_in  income.weights`
`--training_score_weights_in`	When evaluating a model, optionally specify a file containing the score weights for each point in `--training_in`. By default, each point in `--training_in` gets a score weight of 1. `--training_score_weights_in` will only be used if `--tuning_in` is not provided and you use a holdout from the `--training_in` for tuning. Please specify `--tuning_score_weights_in` to tune on `--tuning_in` with score weights. **Usage**: `--training_score_weights_in  income.weights`

Table 60: *RDF Input Data Options (continued)*

Command	Description
`--yield_values_in`	Optionally specify the file used to calculate yield values during tuning. This must be the same length as either the training or tuning vector, depending on whether you are using holdouts or a tuning table. Note that if `--testing_objective=yield` is specified, then this option is required, and its value will be used to determine the best model. **Usage**: `--yield_values_in  income.yield.st`

Table 61: *RDF Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio  0.2`
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  10`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_labels_in`	If `--tuning_in` is specified, then also specify a file containing the labels of the tuning data.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

Table 61: *RDF Model Validation Options (continued)*

Command	Description

`--tuning_models_out`	File prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`. Note that this generates a single model file storing all of the trees tuned over rather than generating separate lists. To limit the number of trees when using this model, specify the `--num_trees` option when loading this model.
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels \`
	`--tuning_models_out  sdss.tune.models`

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or JSON. This value defaults to CSV.
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible JSON format.
	Usage:
	`--tuning_results_out  sdss.tune.results \`
	`--tuning_results_format  json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to JSON using `--tuning_results_format`. **Usage**: `--tuning_results_out  sdss.tune.results`
`--tuning_score_weights_in`	Optionally specify a file containing the score weights for each of the tuning points. By default, each tuning point gets a weight of `1`. Use this option with `--tuning_in` to tune with score weights. **Usage**: `--tuning_score_weights_in  income.tune.weights`

Table 62: *RDF Output Data Options*

Command	Description
`--labels_out`	Optionally specify a file to store computed labels. **Usage**: `--labels_out  results`
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  sdss.lsvm.model.0.1`

Table 62: *RDF Output Data Options (continued)*

Command	Description
`--output_with_ids`	If enabled, the per-point output for `--labels_out` and `--probabilities_out` will be prepended with the input file's "`id`" meta field followed by a comma. For example, if the original output is "`a,b,c`" then the new output would be "`ID,a,b,c`" where "`ID`" is an integer (`1, -1`). If an "`id`" meta field is not available in the input, then "`0`" is set as the input "`id`" field (for example, "`0,a,b,c`"). This option is disabled by default. **Usage**: `--output_with_ids on`
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Notes**: • This option is not supported for multi-class classification in RDF. • The PMML file that is produced requires that missing values are encoded as '?'. This is a reserved symbol for missing values, and as such, should not be used anywhere else in your data. **Usage**: `--pmml_out income.pmml`
`--probabilities_out`	Optionally specify a file to store computed probabilities. **Usage**: `--probabilities_out sdss.test.probabilities`

`--split_importances_buckets`	Optionally specify a number of buckets in split importance calculation. Must be > `1`. This value defaults to `10`.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split.importances.out \`
	`--split_importances_buckets 4`

`--split_importances_out`	Optionally specify a file to store split importances. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split_importances.out`

`--variable_importances_out`	Optionally specify a file to store intrinsic variable importances. This output file includes a single column. The first row is the importances of the first column variable in the `--training_in` .st file, the second row is the importance of the second column, and so on. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--variable_importances_out variable_importances.out`

Table 62: *RDF Output Data Options (continued)*

Command	Description

`--variable_importances_out_as_json`	This option specifies to write variable importances as a JSON file. The output will include feature names, attribute names, and column IDs along with the importance.
	
	Usage:
	`--training_in  income.data.st \`
	`--testing_in  income.test.st \`
	`--variable_importances_out  variable_importances.out`
	`--variable_importances_out_as_json  on`

Table 63: *Tunable RDF Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>.*

Command	Description

`--imbalance_scale`	Relative size of all classes with respect to the smallest class. Requires `--imbalance`. If `--imbalance` is specified, then this value defaults to `1.0`.
	
	Usage:
	`--imbalance \`
	`--imbalance_scale  2`

`--num_dimensions`	The number of dimensions to sample at each node. If omitted, ($\log_2(D) + 1$) is used, where *D* is the number of attributes. **Usage**: `--num_dimensions  1:1:4`
`--num_trees`	The number of trees in the ensemble. Must be > `0`. **Usage**: `--num_trees  5:5:100`

`--sampling_ratio`	Per class sampling ratio. You must either specify a single `--sampling_ratio` that will be used for each class, or repeat the `--sampling_ratio` option for each class in `--training_labels_in`. If omitted with `--sample_with_replacement=on`, full bootstrapping will be used. If `--sample_with_replacement=off`, then this option is mandatory.
	
	Usage:
	`--training_labels_in  income.data.labels \`
	`--sampling_ratio  0.25 \`
	`--sampling_ratio  2`

`--smoothing`	Probability smoothing parameter. Must be >= `0`. Smoothing is disabled when set to `0`. **Usage**: `--smoothing  0,0.001,0.01`
`--tree_depth`	The depth to which each ensemble is built. If set to `0`, trees will be built to the fullest extent. **Usage**: `--tree_depth  0:2:10`

***Table 64:** RDF Options*

Command	Description
`--cardinality_based_dimension_sampling`	Categorical dimensions with larger number of unique values in a given node will have a proportionately higher chance of being selected when sampling dimensions. Continuous features are treated as having one unique value. When `off` (default), all dimensions have an equal chance of being selected in dimension sampling. **Usage**: `--cardinality_based_dimension_sampling  on`
`--categorical_random_split_threshold`	If the number of categorical values exceeds this, then random splitting is attempted. Large values will slow down training time considerably for exhaustive searches of all combinations of O(2^{q-1}). For binary classification problems, though, it is O(q) and, therefore, can be set to larger values. This value defaults to `4`. **Usage**: `--categorical_random_split_threshold  5`
`--categorical_random_split_tries`	The number of times to try random categorical splits. This value defaults to 10. **Usage**: `--categorical_random_split_tries  8`
`--categorical_sampling_seed`	Optionally specify the categorical values sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--categorical_sampling_seed  4`
`--categorical_selection_method`	Optionally specify one of the following: • `random` • `random_exact` (default) • `one_vs_all` • `one_vs_all_random` • `one_vs_all_exact` • `exact` **Usage**: `--categorical_selection_method  exact`
`--classification_objective`	Optionally specify the objective for classification threshold tuning. Specify either `fscore` (default) or `accuracy`. This option cannot be used in conjunction with `--probability_threshold`. **Usage**: `--classification_objective  accuracy`
`--compression`	Compression for data. Reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`

Table 64: *RDF Options (continued)*

Command	Description
`--dimension_sampling_seed`	The dimension sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--dimensions_sampling_seed  10`
`--imbalance`	If set, attempt to improve classification for imbalanced classes. Cannot be used in conjunction with `--sampling_ratio`. **Usage**: `--imbalance  off`
`--impurity`	The type of impurity used to generate the split. Specify `entropy` (default), `gini`, or `misclassification`. **Usage**: `--impurity  gini`
`--k_for_precision`	Optionally specify *k* as a value between `0` and `1` for precision at the top 100*k*-th percentile. *k* defaults to 0.1. **Usage**: `--k_for_precision  0.2`
`--num_cached_trees`	Optionally specify the number of simultaneously cached trees in the ensemble. Higher values lead to higher memory usage. Lower values result in lower memory usage at the expense of increased runtime. This value defaults to the number of available threads. **Usage**: `--num_cached_trees  6`

`--prob_aggregation_method`	Method used for aggregation of individual tree predictions. Specify either `vote` or `average` (default). Only used in conjunction with `--testing_in`.
	Usage:
	`--testing_in  sdss_test.st`
	`--prob_aggregation_method  vote`

`--probability_threshold`	Optionally specify the probability to be used as the threshold for classification. Cannot be used in conjunction with `--classification_objective` or `--testing_objective`. **Usage**: `--probability_threshold  0.8208208919380429`

`--sample_with_replacement`	For each tree, sample the training points using replacement (bootstrap). This option is on by default. If turned `off`, `--sampling_ratio` must be provided.
	Usage:
	`--sample_with_replacement  off`
	`--sampling_ratio  0.25`

Table 64: *RDF Options (continued)*

Command	Description
`--table_sampling_seed`	The data sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--table_sampling_seed 1359937539`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Specify one of the following: • `gini` (default) • `fscore` • `accuracy` • `capture_dev` • `precision_at_k` • `yield` Cannot be used in conjunction with `--probability_threshold`. Note that if `yield` is specified, then `--yield_values_in` must also be specified. **Usage**: `--testing_objective capture_dev`
`--use_gain_ratio`	Optionally specify whether to use information gain ratio vs. information gain as a measure for split criterion. This value defaults to `on`. **Usage**: `--use_gain_ratio off`

Table 65: *General RDF Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to `off`. **Usage**: `--fast_read on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file input`

Table 65: General RDF Options (continued)

Command	Description
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_ threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`

Table 65: *General RDF Options (continued)*

Command	Description
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold 1`

Random Decision Forests Regression Options

The following tables show the options available in the `rdfr` module.

Table 66: *RDFR Input Data Options*

Command	Description
`--batch_size`	Specify the batch size for streamed test points. A larger batch size can increase throughput but also increase latency. This value defaults to `1`. **Usage**: `--batch_size 2`
`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in model`
`--port`	Optionally specify a port number for streaming test data. Must be >= `1024` if specified. **Usage**: `--port 10000`
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in kddcup.train.st`
`--training_point_weights_in`	Optionally specify a file containing the point weights (for training a model) for each point in `--training_in`. By default, each training point gets a point weight of 1. **Usage**: `--training_point_weights_in kddcup.weights`
`--training_score_weights_in`	When evaluating a model, optionally specify a file containing the score weights for each point in `--training_in`. By default, each point in `--training_in` gets a score weight of `1`. `--training_score_weights_in` will only be used if `--tuning_in` is not provided and you use a holdout from the `--training_in` for tuning. Please specify `--tuning_score_weights_in` to tune on `--tuning_in` with score weights. **Usage**: `--training_score_weights_in kddcup.weights`

Table 66: RDFR Input Data Options (continued)

Command	Description
`--training_targets_in`	REQUIRED. File containing the targets of the training data. **Usage**: `--training_targets_in  kddcup.targets`
`--testing_in`	Optional file containing the testing data. **Usage**: `--testing_in  kddcup.test.st`

Table 67: RDFR Model Validation Options

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio  0.2`
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  10`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_models_out`	File prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`. Note that this generates a single model file storing all of the trees tuned over rather than generating separate lists. To limit the number of trees when using this model, specify the `--num_trees` option when loading this model.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels \`
	`--tuning_models_out  sdss.tune.models`

Table 67: *RDFR Model Validation Options (continued)*

Command	Description

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or `JSON`. This value defaults to CSV.
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible `JSON` format.
	Usage:
	`--tuning_results_out  sdss.tune.results \`
	`--tuning_results_format  json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to `JSON` using `--tuning_results_format`. **Usage**: `--tuning_results_out  sdss.tune.results`
`--tuning_score_weights_in`	Optionally specify a file containing the score weights for each of the tuning points. By default, each tuning point gets a weight of `1`. Use this option with `--tuning_in` to tune with score weights. **Usage**: `--tuning_score_weights_in  income.tune.weights`

`--tuning_targets_in`	Specify an file containing the targets of the tuning data. Required if `--tuning_in` is provided.
	Usage:
	`--tuning_in  income.tune.st  \`
	`--tuning_targets_in  income.tune.targets`

Table 68: *RDFR Output Data Options*

Command	Description
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  sdss.lsvm.model.0.1`
`--output_with_ids`	If enabled, the per-point output for `--targets_out` will be prepended with the input file's "`id`" meta field followed by a comma. For example, if the original output is "`a,b,c`" then the new output would be "`ID,a,b,c`" where "`ID`" is an integer (`1`, `-1`). If an "`id`" meta field is not available in the input, then "`0`" is set as the input "`id`" field (for example, "`0,a,b,c`"). This option is disabled by default. **Usage**: `--output_with_ids  on`

Table 68: *RDFR Output Data Options (continued)*

Command	Description
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Notes**: • This option is not supported for multi-class classification in RDF. • The PMML file that is produced requires that missing values are encoded as '?'. This is a reserved symbol for missing values, and as such, should not be used anywhere else in your data. **Usage**: `--pmml_out income.pmml`

`--split_importances_buckets`	Optionally specify a number of buckets in split importance calculation. Must be > 1. This value defaults to 10.
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split.importances.out \`
	`--split_importances_buckets 4`

`--split_importances_out`	Optionally specify a file to store split importances. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--split_importances_out split_importances.out`

`--targets_out`	Optionally specify a file to store computed targets. **Usage**: `--targets_out targets.rdfr`

`--variable_importances_out`	Optionally specify a file to store intrinsic variable importances. This output file includes a single column. The first row is the importances of the first column variable in the `--training_in` .st file, the second row is the importance of the second column, and so on. Requires `--training_in` and one of `--testing_in` or `--model_out`.
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--variable_importances_out variable_importances.out`

`--variable_importances_out_as_json`	This option specifies to write variable importances as a JSON file. The output will include feature names, attribute names, and column IDs along with the importance.
	Usage:
	`--training_in income.data.st \`
	`--testing_in income.test.st \`
	`--variable_importances_out variable_importances.out`
	`--variable_importances_out_as_json on`

Table 69: *Tunable RDFR Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>.*

Command	Description
`--min_node_weight`	The minimum bound on the total weight in each leaf during tree building. This value defaults to `2`. **Usage**: `--min_node_weight  4`
`--num_dimensions`	The number of dimensions to sample at each node. By default, RDFR used (D/3) dimensions, where D is the total number of attributes in the dataset. **Usage**: `--num_dimensions  1:1:4`
`--num_trees`	The number of trees in the ensemble. Must be > `0`. **Usage**: `--num_trees  5:5:100`

`--sampling_ratio`	Per class sampling ratio. You must either specify a single `--sampling_ratio` that will be used for each class, or repeat the `--sampling_ratio` option for each class in `--training_labels_in`. If omitted with `--sample_with_replacement=on`, full bootstrapping will be used. If `--sample_with_replacement=off`, then this option is mandatory.
	Usage:
	`--training_labels_in  income.data.labels \`
	`--sampling_ratio  0.25 \`
	`--sampling_ratio  2`

`--tree_depth`	The depth to which each ensemble is built. If set to `0`, trees will be built to the fullest extent. **Usage**: `--tree_depth  0:2:10`

Table 70: *RDFR Options*

Command	Description
`--cardinality_based_dimension_sampling`	Categorical dimensions with larger number of unique values in a given node will have a proportionately higher chance of being selected when sampling dimensions. Continuous features are treated as having one unique value. When `off`, all dimensions have an equal chance of being selected in dimension sampling. **Usage**: `--cardinality_based_dimension_sampling  off`
`--categorical_random_split_threshold`	If the number of categorical values exceeds this, then random splitting is attempted. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_random_split_threshold  5`

Table 70: *RDFR Options (continued)*

Command	Description
`--categorical_random_split_tries`	The number of times to try random categorical splits. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_random_split_tries 8`
`--categorical_sampling_seed`	Optionally specify the categorical values sampling random number seed. Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_sampling_seed 4`
`--categorical_selection_method`	Optionally specify one of the following: ▪ `random` ▪ `random_exact` ▪ `one_vs_all` ▪ `one_vs_all_random` ▪ `one_vs_all_exact` ▪ `exact` Note that this option is deprecated and may be removed in a future release. **Usage**: `--categorical_selection_method random`
`--compression`	Compression for data. Reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`
`--dimension_sampling_seed`	The dimension sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--dimensions_sampling_seed 10`
`--num_cached_trees`	Optionally specify the number of simultaneously cached trees in the ensemble. Higher values lead to higher memory usage. Lower values result in lower memory usage at the expense of increased runtime. This value defaults to the number of available threads. **Usage**: `--num_cached_trees 6`

`--sample_with_replacement`	For each tree, sample the training points using replacement (bootstrap). This option is on by default. If turned off, `--sampling_ratio` must be provided.
	Usage:
	`--sample_with_replacement off`
	`--sampling_ratio 0.25`

Table 70: RDFR Options (continued)

Command	Description
`--table_sampling_seed`	The data sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--table_sampling_seed 1359937539`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Specify one of the following: • `mean_absolute_error` (default) • `mean_squared_error` • `coeff_determination` • `normalized_gini` **Usage**: `--testing_objective mean_squared_error`

Table 71: General RDFR Options

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to `off`. **Usage**: `--fast_read on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log log_run`

***Table 71:** General RDFR Options (continued)*

Command	Description
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`

__Table 71:__ General RDFR Options (continued)

Command	Description
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold 1`

Recommendation Scoring Options

The following tables show the options available in the `score-recommendation` module.

__Table 72:__ Recommendation Scoring Input Options

Command	Description
`--group_ids_in`	Specify the file containing the ranking groups as predicted. **Usage**: `--group_ids_in ranking.test.groupids`
`--scores_in`	Specify the file containing the classification scores as predicted. **Usage**: `--scores probabilities`
`--true_labels_in`	Specify the file containing the true classification labels. **Usage**: `--true_labels_in ranking.test.kabels`

__Table 73:__ General Recommendation Scoring Options

Command	Description
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel verbose`

Table 73: *General Recommendation Scoring Options (continued)*

Command	Description
`--score_weights_in`	Optionally specify a file containing the score weights for each of the test points. This determines the weight each test point gets in the final score. By default, each test point gets a weight of `1`. **Usage**: `--score_weights_in  ranking.test.weights`

Scoring Options

The following tables show the options available in the `score` module.

Table 74: *Classification Scoring Options*

Command	Description

`--capture_deviation`	Specify whether to automatically compute the capture deviation from the given input files. This option is enabled by default. Requires `--true_labels_in`, `--scores_in`, and `--probabilities_in`.
	Usage:
	`--true_labels_in  sdss.test.labels \`
	`--scores_in  scores.nnc.class1 \`
	`--probabilities_in  probs.nnc.class1 \`
	`--capture_deviation  off`

`--capture_deviation_quantiles`	The number of quantiles to use for the computation of the capture deviation. Defaults to deciles (`10`). **Usage**: `--capture_deviation_quantiles  4`

`--classification_objective`	If provided, this option is used to tune the probability threshold to optimize the given measure. Specify either "`accuracy`" or "`fscore`" (default). Requires `--true_labels_in` and cannot be used in conjunction with `--predicted_labels_in`.
	Usage:
	`--true_labels_in  sdss.test.labels \`
	`--classification_objective  accuracy`

`--classweight`	Optionally specify a classweight. You must repeat the `--classweight` option for each class in `--true_labels_in`. If omitted, classweights of `1` are assumed for every class. **Usage**: `--classweight  5`

Table 74: *Classification Scoring Options (continued)*

Command	Description

`--confusion_matrix`	Specify whether to compute the confusion matrix and associated values. Requires `--true_labels_in` and `--predicted_labels_in`. This value defaults to `on`.
	
	Usage:
	`--true_labels_in  sdss.test.labels \`
	`--probabilities_in  probs.nnc.class1 \`
	`--confusion_matrix  off`

`--curve`	Specifies the type of curve used for AUC/Gini calculation. Specify one of the following: • `roc`: Receiver Operations Curve (ROC) (default): ▪ x-axis: false positive rate ▪ y-axis: true positive rate (capture rate) • `lorenz`: Lorenz curve. ▪ x-axis: percentile ▪ y-axis: true positive rate (capture rate) **Usage**: `--curve  lorenz`
`--curve_out`	Optionally specify a file to which the ROC/Lorenz curve data is written. Note that if sampling points are provided via the `--percentiles_in` option, then the curve's values are only emitted at those sample points. **Usage**: `--curve_out  curve.st`

`--gini`	Specify whether to compute the Gini index. This option defaults to `on` and requires `--true_labels_in` and `--scores_in`.
	
	Usage:
	`--true_labels_in  sdss.test.labels \`
	`--scores_in  scores.nnc.class1 \`
	`--gini  off`

`--k_for_precision_recall`	Optionally specify `k` as a value between `0` and `1` for precision and recall at the top 100k-th percentile. `k` defaults to `0.1`, computing precision and recall at the top 10-th percentile. **Usage**: `--k_for_precision_recall  0.3`

`--labels_out`	Optionally specify a file to which to write labels. Requires `--classification_objective` or `--probability_threshold` for classification.
	
	Usage:
	`--labels_out  classification.labels \`
	`--classification_objective  accuracy`

Table 74: Classification Scoring Options (continued)

Command	Description

`--percentiles_in`	Optionally specify a file containing a single line with percentile numbers (e.g., 0.5 1 5 10 20 30 50 100) to be used as sampling positions for AUC calculations from the Lorenz curve. A low percentile value refers to high scores. If omitted, all data points are used for integration, yielding most accurate results. Requires `--curve=lorenz`.
	Usage:
	`--percentiles_in  percentiles \`
	`--curve  lorenz`

`--pr_curve_out`	Given true labels and probabilities, this option provides a summarized precision recall curve of a desired size (using `--pr_curve_size`, which defaults to 1000) to a specified file. Unlike the `--precision_recall_out` option, this also provides information about the confusion matrix. The output is in CSV format and includes the precision, recall, probability threshold, true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), and the percentile (number of points vs. total number of points).
	Usage:
	`--probabilities_in  probabilities.class1`
	`--true_labels_in   sdss.test.targets`
	`--pr_curve_out  pr_curve.csv`

`--pr_curve_size`	Specify the size of the precision recall curve output. This value defaults to 1000, which outputs an entry per 0.1%-tile of the test points.
	Usage:
	`--probabilities_in  probabilities.class1`
	`--true_labels_in   sdss.test.targets`
	`--pr_curve_out  pr_curve.csv`
	`--pr_curve_size  500`

`--precision_recall_out`	Given true labels and probabilities, this writes the precision recall curve for each of the unique predicted probabilities. The output contains three columns in the following format: <precision>,<recall>,<probability threshold> at which the values occur. **Usage**: `--precision_recall_out  precision.recall`
`--predicted_labels_in`	Specify the file containing the classification labels as predicted. This and/or `--probabilities_in` must be specified for yield deviation scoring. **Usage**: `--predicted_labels_in  labels.wnnc`
`--probabilities_in`	Specify the file containing the classification probabilities. This and/or `--predicted_labels_in` must be specified for yield deviation scoring. **Usage**: `--probabilities_in  probs.wnnc.class1`
`--probability_threshold`	Optionally specify the probability to be used as the threshold for classification. Cannot be used in conjunction with `--predicted_labels_in`. **Usage**: `--probability_threshold  0.82082089`

Table 74: *Classification Scoring Options (continued)*

Command	Description
`--scores_in`	Specify the file containing the classification scores as predicted. **Usage**: `--scores  scores.wnnc.class1`
`--true_labels_in`	Specify the file containing the true classification labels. **Usage**: `--true_labels_in  sdss.test.targets`
`--yield_values_in`	Specify the yield file to be used when calculating the area under the yield. **Usage**: `--yield_values_in  income.yield.st`

Table 75: *Regression Scoring Options*

Command	Description
`--predicted_targets_in`	Specify the file containing the regression targets as predicted. **Usage**: `--predicted_targets_in  targets.rdfr`
`--true_targets_in`	Specify the file containing the true regression targets. **Usage**: `--true_targets_in  kddcup.test.targets`

Table 76: *Recommendation Scoring Options*

Command	Description
`--true_user_item_in`	Specify the file containing the true user-item pairs. **Usage**: `--true_user_item_in  user.item`
`--true_ratings_in`	Specify the file containing the true recommendation ratings. **Usage**: `--true_ratings_in  useritem.ratings`
`--recommendations_in`	Specify the file containing the recommended user-item pairs. User-item pairs must be written as CSV. **Usage**: `--recommendations_in  useritem.recommenation`

Table 76: *Recommendation Scoring Options (continued)*

Command	Description
`--k_for_rec_scoring`	Specify *k* used in precision and hit rate at *k* when scoring the recommendations. This value defaults to 5. **Usage**: `--k_for_rec_scoring  3`

Table 77: *General Scoring Options*

Command	Description
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--score_weights_in`	Optionally specify a file containing the score weights for each of the test points. This determines the weight each test point gets in the final score. By default, each test point gets a weight of `1`. This can be used in both classification and regression scoring. **Usage**: `--score_weights_in  sdss.test.weights`

Singular Value Decomposition Options

The following table shows the options available in the svd module.

Table 78: SVD Options

Command	Description
`--algorithm`	Specify to use one of the following algorithms: • `naive`: standard algorithm • `fast`: fast SVD algorithm (default) • `fastnaive`: computes both fast and naive and compares the results **Usage**: `--algorithm  fastnaive`
`--compute_rec_error`	If enabled, compute the reconstructed matrix and its corresponding reconstruction error. This option is not valid with `--algorithm=naive`. If `--algorithm=fast` or `--algorithm=fastnaive` is specified, then this value defaults to off. **Usage**: `--compute_rec_error  on`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--lsv_out`	Specify the output file for the left singular vectors. **Usage**: `--lsv_out  lsv_out`
`--mean_center`	Specify whether to mean-center the data before SVD, thereby performing principal component analysis (PCA). This option defaults to `off`. **Usage**: `--mean_center  on`
`--rec_matrix_out`	If given, compute and store the reconstructed matrix in this file. This option is only valid when `--compute_rec _error` is enabled. **Usage**: `--rec_matrix_out  reconstructed_out`
`--references_in`	Specify the file containing the input matrix. **Usage**: `--references_in  pm_100x40.st`

***Table 78:** SVD Options (continued)*

Command	Description
`--relative_error`	Specify the permitted relative error of the reconstructed matrix. This is only used when `--algorithm=fast` (which is the default algorithm). In this case, the `--relative_error` value defaults to `0.1`. **Usage**: `--relative_error  0.001`
`--rsv_out`	Specify the output file for the transposed right singular vectors. **Usage**: `--rsv_out  rsv_out`
`--sv_out`	Specify the output file for the singular values. **Usage**: `--sv_out  sv_out`
`--svd_rank`	Output at most this many of the leading singular vectors. If this is not specified, then let the algorithm decide. **Usage**: `--svd_rank  3`

Status Options

The following table shows the options available in the `status` module.

***Table 79:** General Status Options*

Command	Description
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`

Table 79: *General Status Options (continued)*

Command	Description

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

SVM Options

The following tables show the options available in the svm module.

Table 80: *SVM Input Data Options*

Command	Description
`--model_in`	Optionally specify a file from which to load the model. **Usage**: `--model_in  sdss.lsvm.model.0.1`

`--smart_search_restart_in`	When using smart search, optionally specify to load data from a file to continue smart search from saved data.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  10 \`
	`--smart_search_restart_in  sdss.train.restart`

`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  sdss.test.st`
`--training_in`	Specify the file containing the training data. **Usage**: `--training_in  sdss.train.st`
`--training_labels_in`	Specify the file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`
`--yield_values_in`	Optionally specify the file used to calculate yield values during tuning. This must be the same length as either the training or tuning vector, depending on whether you are using holdouts or a tuning table. Note that if `--testing_objective=yield` is specified, then this option is required, and its value will be used to determine the best model. **Usage**: `--yield_values_in  income.yield.st`

***Table 81:** SVM Model Validation Options*

Command	Description
`--holdout_ratio`	Optionally specify the fraction of the table to be held out for tuning. **Usage**: `--holdout_ratio  0.2`
`--holdout_seed`	Optionally specify the holdout sampling random number seed. This value also serves as the seed for `--num_folds`. If omitted, a time-based seed will be used. **Usage**: `--holdout_seed  2`
`--num_folds`	Optionally specify the number of folds for k-fold cross-validation. If `--holdout_ratio` is also specified, then Monte Carlo cross-validation (with randomly drawn holdout sets) is instead performed with the specified number of restarts. **Usage**: `--num_folds  5`
`--smart_search`	Specify to use an intelligent tuning technique (instead of naive grid search). No tuning parameters need to be specified when using this flag. However, if desired, users can provide open or closed intervals for any tuning parameter using <min>:<max> or <min>: or :<max>. Specific values at regular intervals can also be specified with <min>:<step>:<max>. Note that this option cannot be used in conjunction with `--model_in` or `--probability_threshold`. **Usage**: `--smart_search  on`

`--smart_search_iterations`	When `--smart_search=on`, optionally specify the number of search rounds to try for tuning. This value must be greater than 0 and defaults to 100.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  500`

`--smart_search_seed`	When `--smart_search=on`, optionally specify the search seed. If omitted, a time-based seed will be used.
	
	Usage:
	`--smart_search  on \`
	`--smart_search_seed  82427`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_labels_in`	Optionally specify a file containing the labels of the tuning data. Required if `--tuning_in` is provided.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

Table 81: SVM Model Validation Options (continued)

Command	Description

`--tuning_models_out`	Optionally specify a file prefix for storage of all tuning models. Used only in conjunction with `--tuning_in`.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels \`
	`--tuning_models_out  sdss.tune.models`

`--tuning_results_format`	When outputting tuning results, specify whether the format is CSV or `JSON`. This value defaults to CSV.
	
	Note: The CSV format may change. If your environment includes any automation that relies on parsing this file, then we recommend using the extensible `JSON` format.
	
	Usage:
	`--tuning_results_out  sdss.tune.results \`
	`--tuning_results_format  json`

`--tuning_results_out`	Optionally specify a file to store all tuning results. By default, the output format is CSV, but this can be changed to `JSON` using `--tuning_results_format`. **Usage**: `--tuning_results_out  sdss.tune.results`

Table 82: SVM Output data Options

Command	Description
`--labels_out`	Optionally specify a file to store computed labels. **Usage**: `--labels_out  sdss.test.labels.lsvm`
`--model_out`	Optionally specify a file to store the trained model. **Usage**: `--model_out  sdss.lsvm.model.0.1`
`--output_with_ids`	If enabled, the per-point `--labels_out` output will be prepended with the input file's "id" meta field followed by a comma. For example, if the original output is "a,b,c" then the new output would be "ID,a,b,c" where "ID" is an integer (1, -1). If an "id" meta field is not available in the input, then "0" is set as the input "id" field (for example, "0,a,b,c"). This option is disabled by default. **Usage**: `--output_with_ids  on`

Table 82: *SVM Output data Options (continued)*

Command	Description
`--partial_dependencies_out`	Specify the file to store the partial dependencies of the relevant features and/or feature pairs in JSON format. These can be used to generate partial dependence plots. This option is available when: • Training along with (model save OR testing) • Tuning followed by training + (model save OR testing) **Note**: This option can only be used with linear `svm` models. **Usage**: `--partial_dependencies_out  dependencies.json`
`--pmml_out`	Specify the file name to use when exporting models to PMML format. **Note**: This option is not supported for distributed non-linear SVM. **Usage**: `--pmml_out  sdss.pmml`
`--probabilities_out`	Optionally specify a file to store computed probability estimates for test points. **Usage**: `--probabilities_out  probabilities.svm`

`--smart_search_restart_out`	When using smart search, optionally specify to save smart search data to a file. This file can then be specified when running additional smart search iterations.
	Usage:
	`--smart_search  on \`
	`--smart_search_iterations  10 \`
	`--smart_search_restart_out  sdss.train.restart`

Table 83: *Non-Tunable SVM Options*

Command	Description
`--classification_objective`	Optionally specify the objective for classification threshold tuning. Specify either `fscore` or `accuracy`. This option cannot be used in conjunction with `--probability_threshold`. **Usage**: `--classification_objective  accuracy`
`--compression`	Specify whether to perform compression for data. Reduces computational resource requirements. This option defaults to `on`. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`

Table 83: *Non-Tunable SVM Options (continued)*

Command	Description
`--epsilon`	Specify the numerical accuracy to which to train the SVM. Defaults to 1e-3. **Usage**: `--epsilon 0.002`
`--exclude_bias_term`	If set, the SVM formulation will not have a bias term. A bias term is used by default. **Usage**: `--exclude_bias_term on`
`--k_for_precision`	Optionally specify k as a value between `0` and `1` for precision at the top 100k-th percentile. k defaults to `0.1`. **Usage**: `--k_for_precision 0.2`
`--kernel`	Specify the kernel function to be used for the SVM. Specify `linear` (default), `rbf` (Gaussian), or `polynomial`. **Note**: An SVM smart search run is based on a specified kernel type. If `--kernel` is not specified, smart search will run using `--kernel=linear`. If a different kernel function is desired in smart search, then it must be explicitly specified. **Usage**: `--kernel rbf`

`--limit_parameters`	When `--smart_search` is enabled, this option restricts the parameter space in order to reduce the training time. This option is enabled by default.
	
	Usage:
	`--smart_search on`
	`--limit_parameters off`

`--max_cache_memory`	Specify the maximum allowed memory, in megabytes, used to cache the rows of the kernel matrix (to reduce recomputation). This value defaults to 500MB for nonlinear SVM. **Usage**: `--max_cache_memory 1000`
`--max_iterations`	Specify the maximum allowed number of iterations for the training algorithm. This value must be > `0`. For linear SVM training, this corresponds to the number of passes over the data. **Usage**: `--max_iterations 5000000`
`--probability_threshold`	Optionally specify the probability to be used as the threshold for classification. Note that this cannot be used with `--smart_search`. **Usage**: `--probability_threshold 0.8208208919380429`

Table 83: *Non-Tunable SVM Options (continued)*

Command	Description
`--table_sampling_seed`	Specify the data sampling random number seed. If omitted, a time-based seed will be used. **Usage**: `--table_sampling_seed  1359937539`
`--testing_objective`	Optionally specify the objective for selecting the test model from a set of tuned models. Specify one of the following: • `gini` (default) • `fscore` • `accuracy` • `capture_dev` • `precision_at_k` • `yield` Cannot be used in conjunction with `--probability_threshold`. Note that if `yield` is specified, then `--yield_values_in` must also be specified. **Usage**: `--testing_objective  capture_dev`

Table 84: *Tunable SVM Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>.*

Command	Description
`--lambda`	The regularization parameter in the SVM formulation. This option is required when training or tuning without smart search. If `--smart_search` is enabled, then this value is optional. **Usage**: `--lambda  0.1`

`--polynomial_degree`	Specify the degree of the polynomial kernel. This can only be used in conjunction with `--kernel=polynomial`. Must be an integer > 1. This option is required if training or tuning with `--kernel=polynomial`.
	Usage:
	`--kernel  polynomial \`
	`--polynomial_degree  2`

`--polynomial_offset`	Specify the offset of the polynomial kernel. This can only be used in conjunction with `--kernel=polynomial`. Must be >= 0. This value defaults to 0.
	Usage:
	`--kernel  polynomial \`
	`--polynomial_offset  2`

Table 84: *Tunable SVM Options. Specify a single value, comma-separated list, or <min>:<step size>:<max>. (continued)*

Command	Description

`--polynomial_scale`	Specify the scale of the polynomial kernel. This can only be used in conjunction with `--kernel=polynomial`. Must be > 0. This value defaults to 1.0.
	
	Usage
	`--kernel polynomial \`
	`--polynomial_scale 0.5`

`--rbf_bandwidth`	Specify the bandwidth of the RBF kernel. Must be >= 0. This option is required if training or tuning with `--kernel=rbf`.
	
	Usage:
	`--kernel rbf \`
	`--rbf_bandwidth 1.0`

Table 85: *General SVM Options*

Command	Description
`--fast_read`	If set, disable verbose input checks for faster file reads. This option defaults to `off`. **Usage**: `--fast_read on`
`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts localhost,127.0.0.1`
`--input_file`	If given, load input options from this file. **Usage**: `--input_file input`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel verbose`

Table 85: General SVM Options (continued)

Command	Description
`--memory`	Specifies the amount of memory allocated for Skytree Server computation on each host. When training, testing, or tuning via grid-search, if Skytree Server cannot operate within the given amount of memory, it fails with an error message indicating the amount of memory required for computation.. **Usage**: `--memory 10240`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Note**: Changing the number of threads in linear SVM (single host or multi-host) or in nonlinear SVM (multi-host only) can change results numerically. This will have only minimal effect on scoring when the models are trained with sufficient accuracy. In addition, this behavior occurs only when specifying `--threads` > `1`. **Usage**: `--threads  16`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_ threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_ threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Time Series Options

The following tables show the options available in `time_slicer.sh`.

Table 86: Time Series Input Data Options

Command	Description
`-input_file`	Required. Specify the Time Series data input file. **Usage**: `-input_file  ~/input_data`
`-bounds`	Optionally specify a file from which to read upper and lower bounds rather than explicitly specifying `-lower_bounds` and `-upper_bounds`. **Usage**: `-bounds  ~/bounds.txt`

Table 87: Time Series Output Data Options

Command	Description
`-output_file`	Required. Specify the output file for time series data. **Usage**: `-output_file  ~/specimen.slice`

Table 88: Loess Smoothing Options

Command	Description
`-accuracy`	Specify the accuracy for Loess Smoothing. This value defaults to 1e-12. **Usage**: `-accuracy  1`
`-bandwidth`	Specify the bandwidth for Loess Smoothing. **Usage**: `-bandwidth  0.2`
`-iterations`	Specify the number of iterations for Loess Smoothing. This value defaults to `2`. **Usage**: `-iterations  3`
`-use_smoother`	Specify to use Loess Smoothing instead of linear regression. When specified, this option extracts a number of features equal to `-window_width*4` if `-time_gap=off` and `-degree*4` if `-time_gap=on`. This option defaults to off. **Usage**: `-use_smoother  on`

Table 89: *Time Series Options*

Command	Description
`-date_format`	Format for the time stamp. Time stamp is assumed to be already converted if this option is not specified. Note that quotes (" ") are required if the date format includes spaces. **Usage**: `-date_format "yyyy-mm-dd hh:mm:ss"`
`-degree`	Specify the degree of the polynomial to be fit over the window. If `-use_smoother` is specified, this option configures the number of features to be created by smoothing. **Usage**: `-degree 2`
`-delimiter`	Specify a character that separates columns, or specify one of `TAB`, `COMMA`, `SEMICOLON`, `VBAR`, or `SPACE`. Auto-detected if not specified. Note that Skytree Server allows you to add quotations around text so that it will not be confused as a delimiter (for example, when the delimiter is specified as `COMMA`, and an entry includes a deliberate comma). **Note:** Because multiple spaces can be used to separate columns, it is necessary to explicitly denote empty column values with "" when using `-delimiter=SPACE`. **Usage**: `-delimiter TAB`
`-ID_column`	The column index or name for grouping time series. **Usage**: `-ID_column ID`
`-ignore_columns`	Specify the column names, numbers, or ranges of these to ignore in the sliding window. **Usage**: `-ignore_columns 3,date`
`-lower_bounds`	Comma separated list of minima for each sensor/column. This value defaults to minus infinity. **Usage**: `-lower_bounds ,,0,`
`-missing_char`	Specify the character or string of characters that represents entries with missing values. This defaults to ?. **Usage**: `-missing_char N`
`-num_sparse_input_columns`	Specify the number of sparse columns included in libsvm-formatted data. **Usage**: `-num_sparse_input_columns 4`

Table 89: *Time Series Options (continued)*

Command	Description

`-previous_n`	Specify the number of previous elements to transpose. Based on the column specified with `-transpose`, this option specifies the previous "*n*" values within the window that will be turned into columns. Note that this does not include the last value.
	
	This option defaults to `-window_width` when `-time_gap` is turned off (default). If `-time_gap` is enabled, then this option must be explicitly set.
	
	Usage:
	`-time_gap  on \`
	`-transpose  4 \`
	`-previous_n  2`

`-time_gap`	If this option is enabled, the window width and window increment are defined in terms of time stamp gap. This value defaults to off, implying that the window is defined in terms of the number of rows. **Usage**: `-time_gap  on`
`-time_stamp`	Specify a time stamp value. **Usage**: `-time_stamp  Date`
`-time_zone`	Specify the timezone for the time stamp. This value defaults to GMT. When specified, this value must be in the format "GMT + *x*:00" **Usage**: `-time_zone  "GMT + 3:00"`
`-transpose`	Specify the column or columns to be transposed. **Usage**: `-transpose  4,lastupdated`
`-upper_bounds`	Comma separated list of maxima for each sensor/column. This value defaults to plus infinity. **Usage**: `-upper_bounds  0,0,,0`
`-use_column_names`	Specify whether to use the column names from the first row. This value defaults to `off`. **Usage**: `-use_column_names  on`
`-window_increment`	Specify how much to slide the window at each step. This value defaults to the value of `--window_width`. **Usage**: `-window_increment  1`
`-window_width`	REQUIRED. Specify the width of the sliding window. **Usage**: `-window_width  2`

***Table 89:** Time Series Options (continued)*

Command	Description
`-write_column_descriptors`	Specify whether to write column descriptors for output. **Usage**: `-write_column_descriptors  on`
`-write_window_descriptors`	Specify whether to write column descriptors for output. **Usage**: `-write_window_descriptors  on`

Two-Point Correlation Options

The following table shows the options available in the `two-pt` module.

***Table 90:** Two-Point Correlation Options*

Command	Description
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--queries_in`	Optionally specify the data for which the two-pt correlation is to be found against the references. If this option is not specified, then `--references_in` is used. **Usage**: `--queries_in  adult_test_transformed_stratified_6k.st`
`--radius`	Specify the radius (NOT the squared radius) for the two-point correlation. **Usage**: `--radius  1`
`--references_in`	REQUIRED file containing the input data. **Usage**: `--references_in  adult_train_transformed_stratified_10k.st`

Weighted Nearest Neighbors Classification Options

The following table shows the options available in the wnnc module.

Table 91: *WNNC Options*

Command	Description
`--algorithm`	Specify the algorithm to find nearest neighbors. Specify `naive`, `fast`, or `fastest` (default). **Usage**: `--algorithm naive`

`--alpha`	Specify an optional approximation parameter 1 when performing alpha-beta approximation. This option specifies the accuracy and must be in the range (`0, 1`). Larger `--alpha` and `--beta` will improve accuracy somewhat, though smaller values are key for speed.
	Usage:
	`--alpha 0.2 \`
	`--beta 5`

`--alpha_beta_random_seed`	Optionally specify the random number seed used for alpha-beta approximation. This option is used only in conjunction with `--alpha`. If omitted, the current time is used.
	Usage:
	`--alpha 0.2 \`
	`--beta 5 \`
	`--alpha_beta_random_seed 123`

`--bandwidth`	Optionally specify the bandwidth used for gaussian and epan kernels. For `--dist_weight=gaussian/epan`, this option is relative to the furthest neighbor distance. For `fixed_gaussian`/`fixed_epan`, it is an absolute bandwidth. If omitted, bandwidths of 1.0 are assumed for every class.
	Usage:
	`--dist_weight epan \`
	`--bandwidth 0.75 \`
	`--bandwidth 1.10`

`--beta`	Specify an optional approximation parameter 2 when performing alpha-beta approximation. This parameter specifies how many samples are taken and must be at least `1`. Larger `--alpha` and `--beta` will improve accuracy somewhat, though smaller values are key for speed.
	Usage:
	`--alpha 0.2 \`
	`--beta 5`

`--classification_objective`	Optionally specify an objective for classification threshold tuning, either '`accuracy`' or '`fscore`' (default). This option cannot be used in conjunction with `--probability_threshold`. **Usage**: `--classification_objective accuracy`

Table 91: *WNNC Options (continued)*

Command	Description

`--classweight`	Optionally specify the class weight for use with nearest neighbor classification. This option impacts the scores and probabilities. You must repeat the `--classweight` option for each class in `--training_labels_in`. If omitted, class weights of 1 are assumed for every class.
	
	Usage:
	`--training_labels_in  sdss.train.labels \`
	`--classweight  10 \`
	`--classweight  1`

`--compression`	Specify whether to use compression for data. Enabling this (default) reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`
`--curve`	Specifies the type of curve used for AUC/Gini calculation. Specify one of the following: • `roc`: Receiver Operations Curve (ROC) (default): ▪ x-axis: false positive rate ▪ y-axis: true positive rate (capture rate) • `lorenz`: Lorenz curve. ▪ x-axis: percentile ▪ y-axis: true positive rate (capture rate) **Usage**: `--curve  lorenz`
`--dist_weight`	Distance weighting method used for scoring. Specify one of the following: • `1/r` : weight by the inverse distance (default) • `1/r^2` : weight by the inverse distance squared • `gaussian`: weight by the multivariate normal distribution • epan: weight by the Epanechnikov kernel • `fixed_gaussian`: weight by the multivariate normal distribution • `fixed_epan`: weight by the Epanechnikov kernel **Usage**: `--dist_weight  epan`
`--distances_out`	Optionally specify a file to store found neighbor distances (not the squared distances). **Usage**: `--distances_out  distances`

Table 91: *WNNC Options (continued)*

Command	Description

`--explore_persistence`	Optionally specify a value to set the persistence of the explore method. This is only used in conjunction with `--metric_learning_method=explore`. This value defaults to `0.2` and must be between `0` and `1`.
	
	Usage:
	`--metric_learning_method explore \`
	`--explore_scales_in explore_scales \`
	`--explore_persistence 0.3`

`--explore_scales_in`	Optionally specify a file containing scaling factors for each dimension. Only used in conjunction with `--metric_learning_method=explore`.
	
	Usage:
	`--metric_learning_method explore \`
	`--explore_scales_in explore_scales \`
	`--explore_persistence 0.3`

`--fast_read`	If set, disable verbose input checks for faster file reads. This option is disabled by default. **Usage**: `--fast_read on`
`--fit_method`	Specify the fitting method used for `--fit_metric_weights` and/or `--metric_learning_method={forward_fit,backward_fit}`. Set to `1` (default) or `2`. **Usage**: `--fit_method 2`
`--fit_metric_weights`	If set, fit initial metric weights. This option is disabled by default. **Usage**: `--fit_metric_weights on`

`--fit_metric_weights_out`	Optionally specify a file for outputting the fit metric weights. This is used only in conjunction with `--fit_metric_weights`.
	
	Usage:
	`--fit_metric_weights on \`
	`--fit_metric_weights_out fit_weights.csv`

`--fit_reg`	Control parameter for fit method. Use a small positive number between 0 and 1. This value is set to 1*e* - 6 by default. This is only used in conjunction with `--fit_method=2`.
	
	Usage:
	`--fit_method 2 \`
	`--fit_reg 0.02`

Table 91: *WNNC Options (continued)*

Command	Description

`--fit_threshold`	Specify an optional threshold value for pruning fitted metric weights. The fit threshold prunes dimensions that seem unhelpful by weighting them to 0. Lower values are more aggressive. Set to 0 to disable. By default `--fit_threshold=8.0` for `--fit_metric_weights` but is disabled for `--metric_learning_method=forward_fit`. This option is only used when `--fit_method=1`.
	Usage:
	`--fit_method  1 \`
	`--fit_metric_weights  on \`
	`--fit_threshold  4`

`--hosts`	Comma-separated list of hosts on which to run the distributed version of Skytree Server. **Usage**: `--hosts  localhost,127.0.0.1`
`--imbalance`	If set, attempt to improve nearest neighbor classification for imbalanced classes. This option is enabled by default. **Usage**: `--imbalance  off`

`--imbalance_scale`	Specify the relative size of all classes with respect to the smallest class. This option requires `--sample_imbalance`.
	Usage:
	`--sample_imbalance  on \`
	`--imbalance_scale  2`

`--indices_out`	Optionally specify a file to store found neighbor indices **Usage**: `--indices_out  neighbors`

`--k_neighbors`	REQUIRED. Specify the number of neighbors to use for scoring/classification.
	Usage:
	`--k_neighbors  20 \`
	`--k_neighbors  10`

`--labels_out`	Optionally specify a file to output the computed classification labels. **Usage**: `--labels_out  labels`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`

Table 91: WNNC Options (continued)

Command	Description
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel verbose`
`--memory_usage`	Specify the maximum fraction of system memory to use in the range `[0.0,1.0]`. If set to `0.0`, automatic memory management is disabled. By default, Skytree Server makes an effort not to exceed 75% of its host system's memory, breaking computation into smaller chunks if necessary. **Usage**: `--memory_usage 0.80`
`--metric_learning`	Specify whether to perform metric learning. This option is disabled by default. **Usage**: `--metric_learning on`

`--metric_learning_iterations`	Specify the number of iterations for metric learning. This value defaults to `10` and is only used for `--metric_learning_method={random,explore}`.
	Usage:
	`--metric_learning on \`
	`--metric_learning_method random \`
	`--metric_learning_iterations 20`

`--metric_learning_method`	Specify the method used for metric learning. This is only used in conjunction with `--metric_learning`. Specify one of the following:
	• `forward`
	• `forward_fit` (default)
	• `backward`
	• `backward_fit`
	• `explore`
	• `random`
	Usage:
	`--metric_learning on \`
	`--metric_learning_method explore`

Table 91: *WNNC Options (continued)*

Command	Description

`--metric_learning_min_dim_ratio`	Parameter for `--metric_learning_method={backward,backward_fit}`. Specifies the fractional dimensionality at which to end `backward/backward_fit` metric learning. This option defaults to `0` and must be in the range (`0-1`). A value of `0.5`, for example, would specify that the lowest dimensionality to be tried is half the number of full dimensions.
	
	Usage:
	`--metric_learning on \`
	`--metric_learning_method backward \`
	`--metric_learning_min_dim_ratio 0.5`

`--metric_learning_objective`	Specify one of the following objectives for tuning:
	• `gini` (default)
	• `fscore`
	• `accuracy`
	• `capturedev`
	
	Usage:
	`--metric_learning on \`
	`--metric_learning_objective fscore`

`--metric_learning_random_seed`	Optionally specify a random number seed used for metric learning. This is used only in conjunction with `--metric_learning_method={random,explore,forward_fit}`. For `forward` and `forward_fit`, this parameter is only used if `--metric_learning_min_dim_ratio`>0. If omitted, the current time is used.
	
	Usage:
	`--metric_learning on \`
	`--metric_learning_method forward_fit \`
	`--metric_learning_min_dim_ratio 0.5 \`
	`--metric_learning_random_seed 123`

`--metric_weights_in`	Optionally specify a file containing the metric weights. Must have as many values as there are dimensions. **Usage**: `--metric_weights random.csv`

`--optimal_metric_weights_out`	Optionally specify a file for outputting the optimal metric weights. Only used in conjunction with `--metric_learning`.
	
	Usage:
	`--metric_learning on \`
	`--optimal_metric_weights_out random.csv`

`--percentiles_in`	Optionally specify a file containing a single line with percentile numbers (e.g., 0.5 1 5 10 20 30 50 100) to be used as sampling positions for AUC calculations from the ROC/Lorenz curve. A low percentile value refers to high scores. If omitted, all data points are used for integration, yielding the most accurate results. **Usage**: `--percentiles_in percentiles`

Table 91: *WNNC Options (continued)*

Command	Description
`--probabilities_out`	Optionally specify a file to output the computed class probabilities. **Usage**: `--probabilities_out  probabilities.wnnc`
`--probability_threshold`	Specify an optional probability to be used as the threshold for classification. Cannot be used in conjunction with `--classification_objective`. **Usage**: `--probability_threshold  0.82082089`

`--procs_per_host`	Specify the number of processes for each `--host`. If the `--procs_per_host` option is not provided, then Skytree Server will perform a single process on each host. If `--procs_per_host` is provided without `--hosts`, then the specified number of processes will be used on `localhost`.
	
	Usage:
	`--hosts  localhost,127.0.0.1 \`
	`--procs_per_host  3`

`--rank_error_prob`	Optionally specify a probability (between `0` and `1`) with which the rank error is guaranteed. Can only be used in conjunction with `--rank_error_tol`.
	
	Usage:
	`--rank_error_tol 0.05  \`
	`--rank_error_prob  0.9`

`--rank_error_random_seed`	Optionally specify a random number seed used for rank approximation. Only used in conjunction with `--rank_error_tol`. If omitted, the current time is used.
	
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_random_seed  123`

`--rank_error_tol`	Optionally specify the rank error tolerance (as a fraction of all neighbors). Specify a value between `0` and `1`. The neighbors found will have a rank error of less than `--rank_error_tol` times the number of all neighbors, guaranteed with the probability given by `--rank_error_prob`. Larger values of `--rank_error_tol` will lead to greater speedups.
	
	Usage:
	`--rank_error_tol  0.05 \`
	`--rank_error_prob  0.9`

`--sample_imbalance`	If set, attempt to improve classification for imbalanced classes. This option cannot be used in conjunction with `--sampling_ratio`. If this option is enabled, then `--imbalance_scale` can be specified. This option is disabled by default. **Usage**: `--sample_imbalance  on`

Table 91: *WNNC Options (continued)*

Command	Description

`--sample_with_replacement`	For each tree, sample the training points using replacement (bootstrap). This option is `on` by default. If turned `off`, `--sampling_ratio` must be provided.
	
	Usage:
	`--sample_with_replacement  off \`
	`--sampling_ratio  0.25`

`--sampling_ratio`	Per class sampling ratio. You must either specify a single `--sampling_ratio` that will be used for each class, or repeat the `--sampling_ratio` option for each class in `--training_labels_in`. If omitted with `--sample_with_replacement=on`, full bootstrapping will be used. If `--sample_with_replacement=off`, then this option is mandatory.
	
	Usage:
	`--training_labels_in  income.data.labels \`
	`--sampling_ratio  0.25 \`
	`--sampling_ratio  2`

`--scores_out`	Optionally specify a file to output the computed classification scores. **Usage**: `--scores_out  scores.wnnc`
`--scoreweight`	Optionally specify the score weight for use with nearest neighbor classification. This value will affect the probabilities. You must repeat the `--scoreweight` option for each class in `--training_labels_in`. If omitted, score weights of `1` are assumed for every class. **Usage**: `--scoreweight  5`
`--smoothing`	Specify the probability smoothing parameter. Must be >= `0`. Disabled if set to `0.0` (default). **Usage**: `--smoothing  0.001,0.01`
`--table_sampling_seed`	Optionally specify a random number seed used for table sampling. If omitted, the current time is used. **Usage**: `--table_sampling_seed  1359937539`
`--testing_in`	Optionally specify a file containing the testing data. **Usage**: `--testing_in  sdss_test.st`

Table 91: *WNNC Options (continued)*

Command	Description
`--threads`	Optionally specify the number of per-process threads. For single host execution, the default number of threads is set to the maximum available on the host. In distributed mode, the number of threads available to each host is the smallest maximum available number of threads among the hosts. For example, if `--hosts=host1,host2` is specified and hosts "host1" and "host2" have, respectively, 4 and 8 threads available, 4 threads will be used for each process. **Usage**: `--threads  16`
`--training_in`	REQUIRED. Specify the file containing the training data. **Usage**: `--training_in  sdss.train.st`
`--training_labels_in`	REQUIRED. Specify the file containing the class labels of the training data. **Usage**: `--training_labels_in  sdss.train.labels`

`--tuning_in`	Optionally specify a file containing the tuning data. Cannot be provided together with `--num_folds` or `--holdout_ratio`. If specified, then `--tuning_labels_in` must also be specified.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--tuning_labels_in`	If `--tuning_in` is specified, then also specify a file containing the labels of the tuning data.
	
	Usage:
	`--tuning_in  sdss.tune.st \`
	`--tuning_labels_in  sdss.tune.labels`

`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

What-If Options

The following tables show the options available in the `whatif` module.

***Table 92:** What-If Input Data Options*

Command	Description
`--model_in`	REQUIRED model file. **Usage**: `--model_in  model`
`--testing_in`	REQUIRED file containing the testing data. **Usage**: `--testing_in  income.data.st`
`--testing_labels_in`	Optionally specify the file containing the class labels of the testing data. This option is required for classifier models. **Usage**: `--testing_labels_in  income.data.labels`
`--testing_targets_in`	Optionally specify a file containing the targets of the testing data. This option is required for regressor models. **Usage**: `--testing_targets_in  income.data.targets`

***Table 93:** Variable Importances Options. Supported methods include GBT, GBTR, RDF, and RDFR*

Command	Description
`--num_trials`	Specify the number of importance calculations over which to average. This value defaults to `1`. **Usage**: `--num_trials  5`
`--shuffle_seed`	Specify the data shuffling random number seed. If omitted, a time-based seed will be used. **Usage**: `--shuffle_seed  10584`
`--variable_importances_out`	REQUIRED output file for variable importances. **Usage**: `--variable_importances_out  importances.out`

Table 94: General What-If Options

Command	Description
`--compression`	Specify whether to use compression for data. Enabling this (default) reduces computational resource requirements. **Note**: Unlike other commands, the = symbol is mandatory when specifying this option. **Usage**: `--compression=off`
`--fast_read`	If set, disable verbose input checks for faster file reads. This option is disabled by default. **Usage**: `--fast_read  on`
`--log`	If given, write log to this file instead of `stdout`. **Usage**: `--log  log_run`
`--loglevel`	Optionally specify the level of log detail. Specify one of the following: • `verbose`: log everything • `default`: log messages and warnings • `warning`: log only warnings • `silent`: no logging **Usage**: `--loglevel  verbose`
`--num_cached_trees`	Specify the number of simultaneously cached trees to be used for ensemble methods. **Usage**: `--num_cached_trees  6`
`--threads`	Number of threads to use. This value defaults to `24`. **Usage**: `--threads  32`
`--watchdog`	If set, monitor system resources and warn if they are running low. Note that this is enabled by default. **Usage**: `--watchdog  off`
`--watchdog_high_load_threshold`	The watchdog warns if the (normalized) system load is higher than the specified value. This value defaults to `1.5`. **Usage**: `--watchdog_high_load_threshold  1`

Table 94: General What-If Options (continued)

Command	Description
`--watchdog_low_memory_threshold`	The watchdog warns if the amount of available system memory is less than the specified fraction. This value defaults to `0.05`. **Usage**: `--watchdog_low_memory_threshold  1`

Appendix C Bibliography

[donoho2000high] High-dimensional data analysis: Donoho, D.L. “The curses and blessings of dimensionality”, AMS Math Challenges Lecture 2000.

[thompson1995stepwise] Thompson, B. “Stepwise regression and stepwise discriminant analysis need not apply here: A guidelines editorial” Educational and Psychological Measurement 1995

[tibishirani1996regression] Tibshirani, R “Regression shrinkage and selection via the lasso”, Journal of the Royal Statistical Society Series B 1996

[Wasserman2004all] Wasserman, L.A. “All of statistics” Springer 2004

[Wasserman2006all] Wasserman, L.A. “All of nonparametric statistics” Springer 2006

[henley1996k] “A k-nearest-neighbour classifier for assessing consumer credit risk”, Henley, WE and Hand, DJ, The Statistician, volume 45 1996.

[baesens2003benchmarking] “Benchmarking state-of-the-art classification algorithms for credit scoring”, Baesens, B, et al., Journal of the Operational Research Society, vol 54, 2003.

[steinwart2008support] “Support vector machines”, Steinwart, I. and Christmann, A. Springer Verlag 2008

[min2005bankruptcy] “Bankruptcy prediction using support vector machine with optimal choice of kernel function parameters”, Min, J.H. and Lee, Y.C. in Expert Systems with Applications volume 28, 2005

[van-bayesian] “Bayesian Kernel Based Classification for Financial Distress Detection”, Van Gestel, et al., in European Journal of Operational Research, vol 172, 2006.

[hamerly2003learning] “Learning the k in k-means”, Hamerly, G. and Elkan, C. In Advances in Neural Information Processing Systems 2003.

[zheng-tobacco] “Tobacco Distribution Based on Improved K-means Algorithm”, Zheng, B. and Tang, F. and Yang, R.I, in IEEE/INFORMS 2009

[reynolds1999relationship] “A relationship customer typology”, Reynolds, K.E. and Beatty, S.E, in Journal of Retailing, vol 75, 2009

[breiman2001rf] Leo Breiman. “Random Forests”, Machine Learning 2001.

[hastie2009elem] Trevor Hastie et al., “Elements of Statistical Learning” 2nd Ed., Springer 2009.

[breiman1984cart] Leo Breiman et al., “Classification and Regression Trees”, Chapman and Hall 1984.

[friedman2001gbt] Jerome H. Friedman, “Greedy Function Approximation: A Gradient Boosting Machine”, The Annals of Statistics, Oct. 2001.

Index