BigML Documentation

Release 3.12.0

The BigML Team

Jul 29, 2017

Contents

BigMLer makes BigML even easier.

BigMLer wraps BigML's API Python bindings to offer a high-level command-line script to easily create and publish datasets and models, create ensembles, make local predictions from multiple models, clusters and simplify many other machine learning tasks.

BigMLer is open sourced under the Apache License, Version 2.0.

CHAPTER 1

BigMLer subcommands

In addition to the `BigMLer` simple command, that covers the main functionality, there are some additional subcommands:

Usual workflows' subcommands

`bigmler cluster`:

Used to generate clusters and centroids' predictions See *Cluster subcommand*.

`bigmler anomaly`:

Used to generate anomaly detectors and anomaly scores. See *Anomaly subcommand*.

`bigmler sample`:

Used to generate samples of data from your existing datasets. See *Sample subcommand*.

`bigmler association`:

Used to generate association rules from your datasets. See *Association subcommand*.

`bigmler logistic-regression`:

Used to generate logistic regression models and predictions. See *Logistic-regression subcommand*.

`bigmler topic-model`:

Used to generate topic models and topic distributions. See *Topic Model subcommand*.

`bigmler time-series`:

Used to generate time series and forecasts. See *Time Series subcommand*.

`bigmler project`:

Used to generate and manage projects for organization purposes. See *Project subcommand*.

Management subcommands

`bigmler delete`:

Used to delete the remotely created resources. See *Delete subcommand*.

`bigmler.export`:

Used to generate the code you need to predict locally with no connection to BigML. See :ref: *bigmler-export*.

Reporting subcommands

`bigmler report`:

Used to generate reports for the analyze subcommand showing the ROC curve and evaluation metrics of cross-validations. See *Report subcommand*.

Model tuning subcommands

`bigmler analyze`:

Used for feature analysis, node threshold analysis and k-fold cross-validation. See *Analyze subcommand*.

Scripting subcommands

`bigmler reify`:

Used to generate scripts to reproduce the existing resources in BigML. See *Reify subcommand*.

`bigmler execute`:

Used to create WhizzML libraries or scripts and execute them. See *Execute subcommand*.

`bigmler whizzml`:

Used to create WhizzML packages of libraries or scripts based on the information of the `metadata.json` file in the package directory. See *Whizzml subcommand*

CHAPTER 2

Quick Start

Let's see some basic usage examples. Check the *installation* and *authentication* sections below if you are not familiar with BigML.

Basics

You can create a new model just with

```
bigmler --train data/iris.csv
```

If you check your dashboard at BigML, you will see a new source, dataset, and model. Isn't it magic?

You can generate predictions for a test set using

```
bigmler --train data/iris.csv --test data/test_iris.csv
```

You can also specify a file name to save the newly created predictions

```
bigmler --train data/iris.csv --test data/test_iris.csv --output predictions
```

If you do not specify the path to an output file, BigMLer will auto-generate one for you under a new directory named after the current date and time (e.g., *MonNov1212_174715/predictions.csv*). With `--prediction-info` flag set to `brief` only the prediction result will be stored (default is `normal` and includes confidence information). You can also set it to `full` if you prefer the result to be presented as a row with your test input data followed by the corresponding prediction. To include a headers row in the prediction file you can set `--prediction-header`. For both the `--prediction-info full` and `--prediction-info brief` options, if you want to include a subset of the fields in your test file you can select them by setting `--prediction-fields` to a comma-separated list of them. Then

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --prediction-info full --prediction-header \
        --prediction-fields 'petal length','petal width'
```

will include in the generated predictions file a headers row

```
petal length,petal width,species,confidence
```

and only the values of `petal length` and `petal width` will be shown before the objective field prediction `species`.

A different `objective field` (the field that you want to predict) can be selected using

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --objective 'sepal length'
```

If you do not explicitly specify an objective field, BigML will default to the last column in your dataset. You can also use as selector the field column number instead of the name (when –no-train-header is used, for instance).

Also, if your test file uses a particular field separator for its data, you can tell BigMLer using `--test-separator`. For example, if your test file uses the tab character as field separator the call should be like

```
bigmler --train data/iris.csv --test data/test_iris.tsv \
        --test-separator '\t'
```

The model's predictions in BigMLer are based on the mean of the distribution of training values in the predicted node. In case you would like to use the median instead, you could just add the `--median` flag to your command

```
bigmler --train data/grades.csv --test data/test_grades.csv \
        --median
```

Note that this flag can only be applied to regression models.

If you don't provide a file name for your training source, BigMLer will try to read it from the standard input

```
cat data/iris.csv | bigmler --train
```

or you can also read the test info from there

```
cat data/test_iris.csv | bigmler --train data/iris.csv --test
```

BigMLer will try to use the locale of the model both to create a new source (if the `--train` flag is used) and to interpret test data. In case it fails, it will try `en_US.UTF-8` or `English_United States.1252` and a warning message will be printed. If you want to change this behaviour you can specify your preferred locale

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --locale "English_United States.1252"
```

If you check your working directory you will see that BigMLer creates a file with the model ids that have been generated (e.g., FriNov0912_223645/models). This file is handy if then you want to use those model ids to generate local predictions. BigMLer also creates a file with the dataset id that has been generated (e.g., TueNov1312_003451/dataset) and another one summarizing the steps taken in the session progress: `bigmler_sessions`. You can also store a copy of every created or retrieved resource in your output directory (e.g., TueNov1312_003451/model_50c23e5e035d07305a00004f) by setting the flag `--store`.

Remote Predictions

All the predictions we saw in the previous section are computed locally in your computer. BigMLer allows you to ask for a remote computation by adding the `--remote` flag. Remote computations are treated as batch computations. This means that your test data will be loaded in BigML as a regular source and the corresponding dataset will be created

and fed as input data to your model to generate a remote `batch prediction` object. BigMLer will download the predictions file created as a result of this `batch prediction` and save it to local storage just as it did for local predictions

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --remote --output my_dir/remote_predictions.csv
```

This command will create a source, dataset and model for your training data, a source and dataset for your test data and a batch prediction using the model and the test dataset. The results will be stored in the `my_dir/remote_predictions.csv` file. If you prefer the result not to be dowloaded but to be stored as a new dataset remotely, add `--no-csv` and `to-dataset` to the command line. This can be specially helpful when dealing with a high number of scores or when adding to the final result the original dataset fields with `--prediction-info full`, that may result in a large CSV to be created as output.

In case you prefer BigMLer to issue one-by-one remote prediction calls, you can use the `--no-batch` flag

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --remote --no-batch
```

Remote Sources

You can create models using remote sources as well. You just need a valid URL that points to your data. BigML recognizes a growing list of schemas (**http**, **https**, **s3**, **azure**, **odata**, etc). For example

```
bigmler --train https://test:test@static.bigml.com/csv/iris.csv

bigmler --train "s3://bigml-public/csv/iris.csv?access-key=[your-access-key]&secret-
↪key=[your-secret-key]"

bigmler --train azure://csv/diabetes.csv?AccountName=bigmlpublic

bigmler --train odata://api.datamarket.azure.com/www.bcn.cat/BCNOFFERING0005/v1/
↪CARRegistration?$top=100
```

Can you imagine how powerful this feature is? You can create predictive models for huge amounts of data without using you local CPU, memory, disk or bandwidth. Welcome to the cloud!!!

Ensembles

You can also easily create ensembles. For example, using bagging is as easy as

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --number-of-models 10 --sample-rate 0.75 --replacement \
        --tag my_ensemble
```

To create a random decision forest just use the *–randomize* option

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --number-of-models 10 --sample-rate 0.75 --replacement \
        --tag my_random_forest --randomize
```

The fields to choose from will be randomized at each split creating a random decision forest that when used together will increase the prediction performance of the individual models.

To create a boosted trees' ensemble use the *–boosting* option

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --boosting --tag my_boosted_trees
```

or add the ‘‘–boosting-iterations‘ limit

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --booting-iterations 10 --sample-rate 0.75 --replacement \
        --tag my_boosted_trees
```

Once you have an existing ensemble, you can use it to predict. You can do so with the command

```
bigmler --ensemble ensemble/51901f4337203f3a9a000215 \
        --test data/test_iris.csv
```

Or if you want to evaluate it

```
bigmler --ensemble ensemble/51901f4337203f3a9a000215 \
        --test data/iris.csv --evaluate
```

There are some more advanced options that can help you build local predictions with your ensembles. When the number of local models becomes quite large holding all the models in memory may exhaust your resources. To avoid this problem you can use the `--max_batch_models` flag which controls how many local models are held in memory at the same time

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --number-of-models 10 --sample-rate 0.75 --max-batch-models 5
```

The predictions generated when using this option will be stored in a file per model and named after the models' id (e.g. *model_50c23e5e035d07305a00004f__predictions.csv"). Each line contains the prediction, its confidence, the node's distribution and the node's total number of instances. The default value for ‘‘max-batch-models‘* is 10.

When using ensembles, model's predictions are combined to issue a final prediction. There are several different methods to build the combination. You can choose `plurality`, `confidence weighted`, `probability weighted` or `threshold` using the `--method` flag

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --number-of-models 10 --sample-rate 0.75 \
        --method "confidence weighted"
```

For classification ensembles, the combination is made by majority vote: `plurality` weights each model's prediction as one vote, `confidence weighted` uses confidences as weight for the prediction, `probability weighted` uses the probability of the class in the distribution of classes in the node as weight, and `threshold` uses an integer number as threshold and a class name to issue the prediction: if the votes for the chosen class reach the threshold value, then the class is predicted and plurality for the rest of predictions is used otherwise

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --number-of-models 10 --sample-rate 0.75 \
        --method threshold --threshold 4 --class 'Iris-setosa'
```

For regression ensembles, the predicted values are averaged: `plurality` again weights each predicted value as one, `confidence weighted` weights each prediction according to the associated error and `probability weighted` gives the same results as `plurality`.

As in the model's case, you can base your prediction on the median of the predicted node's distribution by adding `--median` to your BigMLer command.

It is also possible to enlarge the number of models that build your prediction gradually. You can build more than one ensemble for the same test data and combine the votes of all of them by using the flag `combine_votes` followed by the comma separated list of directories where predictions are stored. For instance

```
bigmler --train data/iris.csv --test data/test_iris.csv \
        --number-of-models 20 --sample-rate 0.75 \
        --output ./dir1/predictions.csv
bigmler --dataset dataset/50c23e5e035d07305a000056 \
        --test data/test_iris.csv  --number-of-models 20 \
        --sample-rate 0.75 --output ./dir2/predictions.csv
bigmler --combine-votes ./dir1,./dir2
```

would generate a set of 20 prediction files, one for each model, in `./dir1`, a similar set in `./dir2` and combine all of them to generate the final prediction.

Making your Dataset and Model public or share it privately

Creating a model and making it public in BigML's gallery is as easy as

```
bigmler --train data/iris.csv --white-box
```

If you just want to share it as a black-box model just use

```
bigmler --train data/iris.csv --black-box
```

If you also want to make public your dataset

```
bigmler --train data/iris.csv --public-dataset
```

You can also share your datasets, models and evaluations privately with whomever you choose by generating a private link. The `--shared` flag will create such a link

```
bigmler --dataset dataset/534487ef37203f0d6b000894 --shared --no-model
```

and the link will be listed in the output of the command

```
bigmler --dataset dataset/534487ef37203f0d6b000894 --shared --no-model
[2014-04-18 09:29:27] Retrieving dataset. https://bigml.com/dashboard/dataset/
↪534487ef37203f0d6b000894
[2014-04-18 09:29:30] Updating dataset. https://bigml.com/dashboard/dataset/
↪534487ef37203f0d6b000894
[2014-04-18 09:29:30] Shared dataset link. https://bigml.com/shared/dataset/
↪8VPwG7Ny39g1mXBRD1sKQLuHrqE
```

or can also be found in the information pannel for the resource through the web interface.

Content

Before making your model public, probably you want to add a name, a category, a description, and tags to your resources. This is easy too. For example

```
bigmler --train data/iris.csv --name "My model" --category 6 \
        --description data/description.txt --tag iris --tag my_tag
```

Please note:

- You can get a full list of BigML category codes here.
- Descriptions are provided in a text file that can also include markdown.
- Many tags can be added to the same resource.
- Use `--no_tag` if you do not want default BigMLer tags to be added.
- BigMLer will add the name, category, description, and tags to all the newly created resources in each request.

Projects

Each resource created in BigML can be associated to a `project`. Projects are intended for organizational purposes, and BigMLer can create projects each time a `source` is created using a `--project` option. For instance

```
bigmler --train data/iris.csv --project "my new project"
```

will first check for the existence of a project by that name. If it exists, will associate the source, dataset and model resources to this project. If it doesn't, a new `project` is created and then associated.

You can also associate resources to an existing `project` by specifying the option `--project-id` followed by its id

```
bigmler --train data/iris.csv --project-id project/524487ef37203f0d6b000894
```

Note: Once a `source` has been associated to a `project`, all the resources derived from this `source` will be automatically associated to the same `project`.

You can also create projects or update their properties by using the *bigmler project* subcommand.

Using previous Sources, Datasets, and Models

You don't need to create a model from scratch every time that you use BigMLer. You can generate predictions for a test set using a previously generated model

```
bigmler --model model/50a1f43deabcb404d3000079 --test data/test_iris.csv
```

You can also use a number of models providing a file with a model/id per line

```
bigmler --models TueDec0412_174148/models --test data/test_iris.csv
```

Or all the models that were tagged with a specific tag

```
bigmler --model-tag my_tag --test data/test_iris.csv
```

You can also use a previously generated dataset to create a new model

```
bigmler --dataset dataset/50a1f441035d0706d9000371
```

You can also input the dataset from a file

```
bigmler --datasets iris_dataset
```

A previously generated source can also be used to generate a new dataset and model

```
bigmler --source source/50a1e520eabcb404cd0000d1
```

And test sources and datasets can also be referenced by id in new BigMLer requests for remote predictions

```
bigmler --model model/52af53a437203f1cfe0001f0 --remote \
        --test-source source/52b0cbe637203f1d3e0015db

bigmler --model model/52af53a437203f1cfe0001f0 --remote \
        --test-dataset dataset/52b0fb5637203f5c4f000018
```

Evaluations

BigMLer can also help you to measure the performance of your supervised models (decision trees, ensembles and logistic regressions). The simplest way to build a model and evaluate it all at once is

```
bigmler --train data/iris.csv --evaluate
```

which will build the source, dataset and model objects for you using 80% of the data in your training file chosen at random. After that, the remaining 20% of the data will be run through the model to obtain the corresponding evaluation.

The same procedure is available for ensembles:

```
bigmler --train data/iris.csv --number-of-models 10 --evaluate
```

and for logistic regressions:

```
bigmler logistic-regression --train data/iris.csv --evaluate
```

You can use the same procedure with a previously existing source or dataset

```
bigmler --source source/50a1e520eabcb404cd0000d1 --evaluate
bigmler --dataset dataset/50a1f441035d0706d9000371 --evaluate
```

The results of an evaluation are stored both in txt and json files. Its contents will follow the description given in the Developers guide, evaluation section and vary depending on the model being a classification or regression one.

Finally, you can also evaluate a preexisting model using a separate set of data stored in a file or a previous dataset

```
bigmler --model model/50a1f43deabcb404d3000079 --test data/iris.csv \
        --evaluate
bigmler --model model/50a1f43deabcb404d3000079 \
        --test-dataset dataset/50a1f441035d0706d9000371 --evaluate
```

As for predictions, you can specify a particular file name to store the evaluation in

```
bigmler --train data/iris.csv --evaluate --output my_dir/evaluation
```

Cross-validation

If you need cross-validation techniques to ponder which parameters (like the ones related to different kinds of pruning) can improve the quality of your models, you can use the `--cross-validation-rate` flag to settle the part of your training data that will be separated for cross validation. BigMLer will use a Monte-Carlo cross-validation variant,

building `2*n` different models, each of which is constructed by a subset of the training data, holding out randomly `n%` of the instances. The held-out data will then be used to evaluate the corresponding model. For instance, both

```
bigmler --train data/iris.csv --cross-validation-rate 0.02
bigmler --dataset dataset/519029ae37203f3a9a0002bf \
        --cross-validation-rate 0.02
```

will hold out 2% of the training data to evaluate a model built upon the remaining 98%. The evaluations will be averaged and the result saved in json and human-readable formats in `cross-validation.json` and `cross-validation.txt` respectively. Of course, in this kind of cross-validation you can choose the number of evaluations yourself by setting the `--number-of-evaluations` flag. You should just keep in mind that it must be high enough to ensure low variance, for instance

```
bigmler --train data/iris.csv --cross-validation-rate 0.1 \
        --number-of-evaluations 20
```

The `--max-parallel-evaluations` flag will help you limit the number of parallel evaluation creation calls.

```
bigmler --train data/iris.csv --cross-validation-rate 0.1 \
        --number-of-evaluations 20 --max-parallel-evaluations 2
```

Configuring Datasets and Models

What if your raw data isn't necessarily in the format that BigML expects? So we have good news: you can use a number of options to configure your sources, datasets, and models.

Most resources in BigML contain information about the fields used in the resource construction. Sources contain information about the name, label, description and type of the fields detected in the data you upload. In addition to that, datasets contain the information of the values that each field contains, whether they have missing values or errors and even if they are `preferred` fields or non-preferred (fields that are not expected to convey real information to the model, like user IDs or constant fields). This information is available in the "fields" attribute of each resource, but BigMLer can extract it and build a CSV file with a summary of it.

```
bigmler --source source/50a1f43deabcb404d3010079 \
        --export-fields fields_summary.csv \
        --output-dir summary
```

By using this command, BigMLer will create a `fields_summary.csv` file in a `summary` output directory. The file will contain a headers row and the fields information available in the source, namely the field column, field ID, field name, field label and field description of each field. If you execute the same command on a dataset

```
bigmler --dataset dataset/50a1f43deabcb404d3010079 \
        --export-fields fields_summary.csv \
        --output-dir summary
```

you will also see the number of missing values and errors found in each field and an excerpt of the values and errors.

But then, imagine that you want to alter BigML's default field names or the ones provided by the training set header and capitalize them, even to add a label or a description to each field. You can use several methods. Write a text file with a change per line as follows

```
bigmler --train data/iris.csv --field-attributes fields.csv
```

where `fields.csv` would be

```
0,'SEPAL LENGTH','label for SEPAL LENGTH','description for SEPAL LENGTH'
1,'SEPAL WIDTH','label for SEPAL WIDTH','description for SEPAL WIDTH'
2,'PETAL LENGTH','label for PETAL LENGTH','description for PETAL LENGTH'
3,'PETAL WIDTH','label for PETAL WIDTH','description for PETAL WIDTH'
4,'SPECIES','label for SPECIES','description for SPECIES'
```

The number on the left in each line is the *column number* of the field in your source and is followed by the new field's name, label and description.

Similarly you can also alter the auto-detect type behavior from BigML assigning specific types to specific fields

```
bigmler --train data/iris.csv --types types.txt
```

where `types.txt` would be

```
0, 'numeric'
1, 'numeric'
2, 'numeric'
3, 'numeric'
4, 'categorical'
```

Finally, the same summary file that could be built with the `--export-fields` option can be used to modify the updatable information in sources and datasets. Just edit the CSV file with your favourite editor setting the new values for the fields and use:

```
bigmler --source source/50a1f43deabcb404d3010079 \
        --import-fields summary/fields_summary.csv
```

to update the names, labels, descriptions or types of the fields with the ones in the `summary/fields_summary.csv` file.

You could also use this option to change the `preferred` attributes for each of the fields. This transformation is made at the dataset level, so in the prior code it will be applied once a dataset is created from the referred source. You might as well act on an existing dataset:

```
bigmler --dataset dataset/50a1f43deabcb404d3010079 \
        --import-fields summary/fields_summary.csv
```

In order to update more detailed source options, you can use the `--source-attributes` option pointing to a file path that contains the configuration settings to be modified in JSON format

```
bigmler --source source/52b8a12037203f48bc00000a \
        --source-attributes my_dir/attributes.json --no-dataset
```

Let's say this source has a text field with id `000001`. The `attributes.json` to change its text parsing mode to full field contents would read

```
{"fields": {"000001": {"term_analysis": {"token_mode": "full_terms_only"}}}}
```

you can also reference the fields by its column number in this JSON structures. If the field to be modified is in the second column (column index starts at 0) then the contents of the `attributes.json` file could be as well

```
{"fields": {"1": {"term_analysis": {"token_mode": "full_terms_only"}}}}
```

The `source-attributes` JSON can contain any of the updatable attributes described in the developers section You can specify the fields that you want to include in the dataset by naming them explicitly

```
bigmler --train data/iris.csv \
        --dataset-fields 'sepal length','sepal width','species'
```

or the fields that you want to include as predictors in the model

```
bigmler --train data/iris.csv --model-fields 'sepal length','sepal width'
```

You can also specify the chosen fields by adding or removing the ones you choose to the list of preferred fields of the previous resource. Just prefix their names with + or − respectively. For example, you could create a model from an existing dataset using all their fields but the `sepal length` by saying

```
bigmler --dataset dataset/50a1f441035d0706d9000371 \
        --model-fields -'sepal length'
```

When evaluating, you can map the fields of the evaluated model to those of the test dataset by writing in a file the field column of the model and the field column of the dataset separated by a comma and using *–fields-map* flag to specify the name of the file

```
bigmler --dataset dataset/50a1f441035d0706d9000371 \
        --model model/50a1f43deabcb404d3000079 --evaluate \
        --fields-map fields_map.txt
```

where `fields_map.txt` would contain

```
0, 1
1, 0
2, 2
3, 3
4, 4
```

if the first two fields had been reversed.

Finally, you can also tell BigML whether your training and test set come with a header row or not. For example, if both come without header

```
bigmler --train data/iris_nh.csv --test data/test_iris_nh.csv \
        --no-train-header --no-test-header
```

Splitting Datasets

When following the usual proceedings to evaluate your models you'll need to separate the available data in two sets: the training set and the test set. With BigMLer you won't need to create two separate physical files. Instead, you can set a `--test-split` flag that will set the percentage of data used to build the test set and leave the rest for training. For instance

```
bigmler --train data/iris.csv --test-split 0.2 --name iris --evaluate
```

will build a source with your entire file contents, create the corresponding dataset and split it in two: a test dataset with 20% of instances and a training dataset with the remaining 80%. Then, a model will be created based on the training set data and evaluated using the test set. By default, split is deterministic, so that every time you issue the same command will get the same split datasets. If you want to generate different splits from a unique dataset you can set the `--seed` option to a different string in every call

```
bigmler --train data/iris.csv --test-split 0.2 --name iris \
        --seed my_random_string_382734627364 --evaluate
```

Advanced Dataset management

As you can find in the BigML's API documentation on datasets besides the basic name, label and description that we discussed in previous sections, there are many more configurable options in a dataset resource. As an example, to publish a dataset in the gallery and set its price you could use

```
{"private": false, "price": 120.4}
```

Similarly, you might want to add fields to your existing dataset by combining some of its fields or simply tagging their rows. Using BigMLer, you can set the `--new-fields` option to a file path that contains a JSON structure that describes the fields you want to select or exclude from the original dataset, or the ones you want to combine and the Flatline expression to combine them. This structure must follow the rules of a specific languange described in the Transformations item of the developers section

```
bigmler --dataset dataset/52b8a12037203f48bc00000a \
        --new-fields my_dir/generators.json
```

To see a simple example, should you want to include all the fields but the one with id `000001` and add a new one with a label depending on whether the value of the field `sepal length` is smaller than 1, you would write in `generators.json`

```
{"all_but": ["000001"], "new_fields": [{"name": "new_field", "field": "(if (< (f \
↪"sepal length\") 1) \"small\" \"big\")"}]}
```

Or, as another example, to tag the outliers of the same field one coud use

```
{"new_fields": [{"name": "outlier?", "field": "(if (within-percentiles? \"sepal␣
↪length\" 0.5 0.95) \"normal\" \"outlier\")"}]}
```

You can also export the contents of a generated dataset by using the `--to-csv` option. Thus,

```
bigmler --dataset dataset/52b8a12037203f48bc00000a \
        --to-csv my_dataset.csv --no-model
```

will create a CSV file named `my_dataset.csv` in the default directory created by BigMLer to place the command output files. If no file name is given, the file will be named after the dataset id.

A dataset can also be generated as the union of several datasets using the flag `--multi-dataset`. The datasets will be read from a file specified in the `--datasets` option and the file must contain one dataset id per line.

```
bigmler --datasets my_datasets --multi-dataset --no-model
```

This syntax is used when all the datasets in the `my_datasets` file share a common field structre, so the correspondence of the fields of all the datasets is straight forward. In the general case, the multi-dataset will inherit the field structure of the first component dataset. If you want to build a multi-dataset with datasets whose fields share not the same column disposition, you can specify which fields are correlated to the ones of the first dataset by mapping the fields of the rest of datasets to them. The option `--multi-dataset-attributes` can point to a JSON file that contains such a map. The command line syntax would then be

```
bigmler --datasets my_datasets --multi-dataset \
        --multi-dataset-attributes my_fields_map.json \
        --no-model
```

and for a simple case where the second dataset had flipped the first and second fields with respect to the first one, the file would read

where `dataset/53330bce37203f222e00004b` would be the id of the second dataset in the multi-dataset.

Model Weights

To deal with imbalanced datasets, BigMLer offers three options: `--balance`, `--weight-field` and `--objective-weights`.

For classification models, the `--balance` flag will cause all the classes in the dataset to contribute evenly. A weight will be assigned automatically to each instance. This weight is inversely proportional to the number of instances in the class it belongs to, in order to ensure even distribution for the classes.

You can also use a field in the dataset that contains the weight you would like to use for each instance. Using the `--weight-field` option followed by the field name or column number will cause BigMLer to use its data as instance weight. This is valid for both regression and classification models.

The `--objective-weights` option is used in classification models to transmit to BigMLer what weight is assigned to each class. The option accepts a path to a CSV file that should contain the `class`,‘‘weight‘‘ values one per row

```
bigmler --dataset dataset/52b8a12037203f48bc00000a \
        --objective-weights my_weights.csv
```

where the `my_weights.csv` file could read

```
Iris-setosa,5
Iris-versicolor,3
```

so that BigMLer would associate a weight of `5` to the `Iris-setosa` class and `3` to the `Iris-versicolor` class. For additional classes in the model, like `Iris-virginica` in the previous example, weight `1` is used as default. All specified weights must be non-negative numbers (with either integer or real values) and at least one of them must be non-zero.

Predictions' missing strategy

Sometimes the available data lacks some of the features our models use to predict. In these occasions, BigML offers two different ways of handling input data with missing values, that is to say, the missing strategy. When the path to the prediction reaches a split point that checks the value of a field which is missing in your input data, using the `last prediction` strategy the final prediction will be the prediction for the last node in the path before that point, and using the `proportional` strategy it will be a weighted average of all the predictions for the final nodes reached considering that both branches of the split are possible.

BigMLer adds the `--missing-strategy` option, that can be set either to `last` or `proportional` to choose the behavior in such cases. Last prediction is the one used when this option is not used.

```
bigmler --model model/52b8a12037203f48bc00001a \
        --missing-strategy proportional --test my_test.csv
```

Models with missing splits

Another configuration argument that can change models when the training data has instances with missing values in some of its features is `--missing-splits`. By setting this flag, the model building algorithm will be able to include the instances that have missing values for the field used to split the data in each node in one of the stemming branches. This will, obviously, affect also the predictions given by the model for input data with missing values. Here's an example to build a model using missing-splits and predict with it.

```
bigmler --dataset dataset/52b8a12037203f48bc00023b \
        --missing-splits --test my_test.csv
```

Fitering Sources

Imagine that you have create a new source and that you want to create a specific dataset filtering the rows of the source that only meet certain criteria. You can do that using a JSON expresion as follows

```
bigmler --source source/50a2bb64035d0706db0006cc --json-filter filter.json
```

where `filter.json` is a file containg a expression like this

```
["<", 7.00, ["field", "000000"]]
```

or a LISP expression as follows

```
bigmler --source source/50a2bb64035d0706db0006cc --lisp-filter filter.lisp
```

where `filter.lisp` is a file containing a expression like this

```
(< 7.00 (field "sepal length"))
```

For more details, see the BigML's API documentation on filtering rows.

Multi-labeled categories in training data

Sometimes the information you want to predict is not a single category but a set of complementary categories. In this case, training data is usually presented as a row of features and an objective field that contains the associated set of categories joined by some kind of delimiter. BigMLer can also handle this scenario.

Let's say you have a simple file

```
color,year,sex,class
red,2000,male,"Student,Teenager"
green,1990,female,"Student,Adult"
red,1995,female,"Teenager,Adult"
```

with information about a group of people and we want to predict the `class` another person will fall into. As you can see, each record has more than one `class` per person (for example, the first person is labeled as being both a `Student` and a `Teenager`) and they are all stored in the `class` field by concatenating all the applicable labels using `,` as separator. Each of these labels is, 'per se', an objective to be predicted, and that's what we can rely on BigMLer to do.

The simplest multi-label command in BigMLer is

```
bigmler --multi-label --train data/tiny_multilabel.csv
```

First, it will analyze the training file to extract all the `labels` stored in the objective field. Then, a new extended file will be generated from it by adding a new field per label. Each generated field will contain a boolean set to `True` if the associated label is in the objective field and `False` otherwise

```
color,year,sex,class - Adult,class - Student,class - Teenager
red,2000,male,False,True,True
green,1990,female,True,True,False
red,1995,female,True,False,True
```

This new file will be fed to BigML to build a `source`, a `dataset` and a set of `models` using four input fields: the first three fields as input features and one of the label fields as objective. Thus, each of the classes that label the training set can be predicted independently using one of the models.

But, naturally, when predicting a multi-labeled field you expect to obtain all the labels that qualify the input features at once, as you provide them in the training data records. That's also what BigMLer does. The syntax to predict using multi-labeled training data sets is similar to the single labeled case

```
bigmler --multi-label --train data/tiny_multilabel.csv \
        --test data/tiny_test_multilabel.csv
```

the main difference being that the ouput file `predictions.csv` will have the following structure

```
"Adult,Student","0.34237,0.20654"
"Adult,Teenager","0.34237,0.34237"
```

where the first column contains the `class` prediction and the second one the confidences for each label prediction. If the models predict `True` for more than one label, the prediction is presented as a sequence of labels (and their corresponding confidences) delimited by `,`.

As you may have noted, BigMLer uses `,` both as default training data fields separator and as label separator. You can change this behaviour by using the `--training-separator`, `--label-separator` and `--test-separator` flags to use different one-character separators

```
bigmler --multi-label --train data/multilabel.tsv \
        --test data/test_multilabel.tsv --training-separator '\t' \
        --test-separator '\t' --label-separator ':'
```

This command would use the `tab` character as train and test data field delimiter and `:` as label delimiter (the examples in the tests set use `,` as field delimiter and ':' as label separator).

You can also choose to restrict the prediction to a subset of labels using the `--labels` flag. The flag should be set to a comma-separated list of labels. Setting this flag can also reduce the processing time for the training file, because BigMLer will rely on them to produce the extended version of the training file. Be careful, though, to avoid typos in the labels in this case, or no objective fields will be created. Following the previous example

```
bigmler --multi-label --train data/multilabel.csv \
        --test data/test_multilabel.csv --label-separator ':' \
        --labels Adult,Student
```

will limit the predictions to the `Adult` and `Student` classes, leaving out the `Teenager` classification.

Multi-labeled predictions can also be computed using ensembles, one for each label. To create an ensemble prediction, use the `--number-of-models` option that will set the number of models in each ensemble

```
bigmler --multi-label --train data/multilabel.csv \
        --number-of-models 20 --label-separator ':' \
        --test data/test_multilabel.csv
```

The ids of the ensembles will be stored in an `ensembles` file in the output directory, and can be used in other predictions by setting the `--ensembles` option

```
bigmler --multi-label --ensembles multilabel/ensembles \
        --test data/test_multilabel.csv
```

or you can retrieve all previously tagged ensembles with `--ensemble-tag`

```
bigmler --multi-label --ensemble-tag multilabel \
        --test data/test_multilabel.csv
```

Multi-labeled resources

The resources generated from a multi-labeled training data file can also be recovered and used to generate more multi-labeled predictions. As in the single-labeled case

```
bigmler --multi-label --source source/522521bf37203f412f000100 \
        --test data/test_multilabel.csv
```

would generate a dataset and the corresponding set of models needed to create a `predictions.csv` file that contains the multi-labeled predictions.

Similarly, starting from a previously created multi-labeled dataset

```
bigmler --multi-label --dataset source/522521bf37203f412fac0135 \
        --test data/test_multilabel.csv --output multilabel/predictions.csv
```

creates a bunch of models, one per label, and predicts storing the results of each operation in the `multilabel` directory, and finally

```
bigmler --multi-label --models multilabel/models \
        --test data/test_multilabel.csv
```

will retrieve the set of models created in the last example and use them in new predictions. In addition, for these three cases you can restrict the labels to predict to a subset of the complete list available in the original objective field. The `--labels` option can be set to a comma-separated list of the selected labels in order to do so.

The `--model-tag` can be used as well to retrieve multi-labeled models and predict with them

```
bigmler --multi-label --model-tag my_multilabel \
        --test data/test_multilabel.csv
```

Finally, BigMLer is also able to handle training files with more than one multi-labeled field. Using the `--multi-label-fields` option you can settle the fields that will be expanded as containing multiple labels in the generated source and dataset.

```
bigmler --multi-label --multi-label-fields class,type \
        --train data/multilabel_multi.csv --objective class
```

This command creates a source (and its corresponding dataset) where both the `class` and `type` fields have been analysed to create a new field per label. Then the `--objective` option sets `class` to be the objective field and

only the models needed to predict this field are created. You could also create a new multi-label prediction for another multi-label field, `type` in this case, by issuing a new BigMLer command that uses the previously generated dataset as starting point

```
bigmler --multi-label --dataset dataset/52cafddb035d07269000075b \
        --objective type
```

This would generate the models needed to predict `type`. It's important to remark that the models used to predict `class` in the first example will use the rest of fields (including `type` as well as the ones generated by expanding it) to build the prediction tree. If you don't want this fields to be used in the model construction, you can set the `--model-fields` option to exclude them. For instance, if `type` has two labels, `label1` and `label2`, then excluding them from the models that predict `class` could be achieved using

```
bigmler --multi-label --dataset dataset/52cafddb035d07269000075b \
        --objective class
        --model-fields=' -type,-type - label1,-type - label2'
```

You can also generate new fields applying aggregation functions such as `count`, `first` or `last` on the labels of the multi label fields. The option `--label-aggregates` can be set to a comma-separated list of these functions and a new column per multi label field and aggregation function will be added to your source

```
bigmler --multi-label --train data/multilabel.csv \
        --label-separator ':' --label-aggregates count,last \
        --objective class
```

will generate `class - count` and `class - last` in addition to the set of per label fields.

Multi-label evaluations

Multi-label predictions are computed using a set of binary models (or ensembles), one for each label to predict. Each model can be evaluated to check its performance. In order to do so, you can mimic the commands explained in the `evaluations` section for the single-label models and ensembles. Starting from a local CSV file

```
bigmler --multi-label --train data/multilabel.csv \
        --label-separator ":" --evaluate
```

will build the source, dataset and model objects for you using a random 80% portion of data in your training file. After that, the remaining 20% of the data will be run through each of the models to obtain an evaluation of the corresponding model. BigMLer retrieves all evaluations and saves them locally in json and txt format. They are named using the objective field name and the value of the label that they refer to. Finally, it averages the results obtained in all the evaluations to generate a mean evaluation stored in the `evaluation.txt` and `evaluation.json` files. As an example, if your objective field name is `class` and the labels it contains are `Adult,Student`, the generated files will be

```
```

Generated files:

MonNov0413_201326

- evaluations
- extended_multilabel.csv
- source
- evaluation_class_student.txt

- models
- evaluation_class_adult.json
- dataset
- evaluation.json
- evaluation.txt
- evaluation_class_student.json
- bigmler_sessions
- evaluation_class_adult.txt

You can use the same procedure with a previously existing multi-label source or dataset

```
bigmler --multi-label --source source/50a1e520eabcb404cd0000d1 \
        --evaluate
bigmler --multi-label --dataset dataset/50a1f441035d0706d9000371 \
        --evaluate
```

Finally, you can also evaluate a preexisting set of models or ensembles using a separate set of data stored in a file or a previous dataset

```
bigmler --multi-label --models MonNov0413_201326/models \
        --test data/test_multilabel.csv --evaluate
bigmler --multi-label --ensembles MonNov0413_201328/ensembles \
        --dataset dataset/50a1f441035d0706d9000371 --evaluate
```

High number of Categories

In BigML there's a limit in the number of categories of a categorical objective field. This limit is set to ensure the quality of the resulting models. This may become a restriction when dealing with categorical objective fields with a high number of categories. To cope with these cases, BigMLer offers the –max-categories option. Setting to a number lower than the mentioned limit, the existing categories will be organized in subsets of that size. Then the original dataset will be copied many times, one per subset, and its objective field will only keep the categories belonging to each subset plus a generic `***** other *****` category that will summarize the rest of categories. Then a model will be created from each dataset and the test data will be run through them to generate partial predictions. The final prediction will be extracted by choosing the class with highest confidence from the distributions obtained for each model's prediction ignoring the `***** other ******` generic category. For instance, to use the same `iris.csv` example, you could do

```
bigmler --train data/iris.csv --max-categories 1 \
        --test data/test_iris.csv --objective species
```

This command would generate a source and dataset object, as usual, but then, as the total number of categories is three and –max-categories is set to 1, three more datasets will be created, one per each category. After generating the corresponding models, the test data will be run through them and their predictions combined to obtain the final predictions file. The same procedure would be applied if starting from a preexisting source or dataset using the `--source` or `--dataset` options. Please note that the `--objective` flag is mandatory in this case to ensure that the right categorical field is selected as objective field.

`--method` option accepts a new `combine` value to use such kind of combination. You can use it if you need to create a new group of predictions based on the same models produced in the first example. Filling the path to the model ids file

```
bigmler --models my_dir/models --method combine \
        --test data/new_test.csv
```

the new predictions will be created. Also, you could use the set of datasets created in the first case as starting point. Their ids are stored in a `dataset_parts` file that can be found in the output location

```
bigmler --dataset my_dir/dataset_parts --method combine \
        --test data/test.csv
```

This command would cause a new set of models, one per dataset, to be generated and their predictions would be combined in a final predictions file.

CHAPTER 3

Advanced subcommands in BigMLer

Analyze subcommand

In addition to the main BigMLer capabilities explained so far, there's a subcommand `bigmler analyze` with more options to evaluate the performance of your models. For instance

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --cross-validation --k-folds 5
```

will create a k-fold cross-validation by dividing the data in your dataset in the number of parts given in `--k-folds`. Then evaluations are created by selecting one of the parts to be the test set and using the rest of data to build the model for testing. The generated evaluations are placed in your output directory and its average is stored in `evaluation.txt` and `evaluation.json`.

Similarly, you'll be able to create an evaluation for ensembles. Using the same command above and adding the options to define the ensembles' properties, such as `--number-of-models`, `--sample-rate`, `--randomize` or `--replacement`

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --cross-validation --k-folds 5 --number-of-models 20
                --sample-rate 0.8 --replacement
```

More insights can be drawn from the `bigmler analyze --features` command. In this case, the aim of the command is to analyze the complete set of features in your dataset to single out the ones that produce models with better evaluation scores. In this case, we focus on `accuracy` for categorical objective fields and `r-squared` for regressions.

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --features
```

This command uses an algorithm for smart feature selection as described in this blog post that evaluates models built by using subsets of features. It starts by building one model per feature, chooses the subset of features used in the model that scores best and, from there on, repeats the procedure by adding another of the available features in the dataset to the chosen subset. The iteration stops when no improvement in score is found for a number of repetitions

that can be controlled using the `--staleness` option (default is `5`). There's also a `--penalty` option (default is `0.1%`) that sets the amount that is substracted from the score per feature added to the subset. This penalty is intended to mitigate overfitting, but it also favors models which are quicker to build and evaluate. The evaluations for the scores are k-fold cross-validations. The `--k-folds` value is set to `5` by default, but you can change it to whatever suits your needs using the `--k-folds` option.

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --features --k-folds 10 --staleness 3 --penalty 0.002
```

Would select the best subset of features using 10-fold cross-validation and a `0.2%` penalty per feature, stopping after 3 non-improving iterations.

Depending on the machine learning problem you intend to tackle, you might want to optimize other evaluation metric, such as `precision` or `recall`. The `--optimize` option will allow you to set the evaluation metric you'd like to optimize.

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --features --optimize recall
```

For categorical models, the evaluation values are obtained by counting the positive and negative matches for all the instances in the test set, but sometimes it can be more useful to optimize the performance of the model for a single category. This can be specially important in highly non-balanced datasets or when the cost function is mainly associated to one of the existing classes in the objective field. Using "–optimize-category" you can set the category whose evaluation metrics you'd like to optimize

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --features --optimize recall \
                --optimize-category Iris-setosa
```

You should be aware that the smart feature selection command still generates a high number of BigML resources. Using `k` as the `k-folds` number and `n` as the number of explored feature sets, it will be generating `k` datasets (`1/k``th of the instances each), and ``k * n` models and evaluations. Setting the `--max-parallel-models` and `--max-parallel-evaluations` to higher values (up to `k`) can help you speed up partially the creation process because resources will be created in parallel. You must keep in mind, though, that this parallelization is limited by the task limit associated to your subscription or account type.

As another optimization method, the `bigmler analyze --nodes` subcommand will find for you the best performing model by changing the number of nodes in its tree. You provide the `--min-nodes` and `--max-nodes` that define the range and `--nodes-step` controls the increment in each step. The command runs a k-fold evaluation (see `--k-folds` option) on a model built with each node threshold in you range and tries to optimize the evaluation metric you chose (again, default is `accuracy`). If improvement stops (see the –staleness option) or the node threshold reaches the `--max-nodes` limit, the process ends and shows the node threshold that lead to the best score.

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --nodes --min-nodes 10 \
                --max-nodes 200 --nodes-step 50
```

When working with random forest, you can also change the number of `random_candidates` or number of fields chosen at random when the models in the forest are built. Using `bigmler analyze --random-fields` the number of `random_candidates` will range from 1 to the number of fields in the origin dataset, and BigMLer will cross-validate the random forests to determine which `random_candidates` number gives the best performance.

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --random-fields
```

Please note that, in general, the exact choice of fields selected as random candidates might be more important than their actual number. However, in some marginal cases (e.g. datasets with a high number noise features) the number of

random candidates can impact tree performance significantly.

For any of these options (`--features`, `--nodes` and `--random-fields`) you can add the `--predictions-csv` flag to the `bigmler analyze` command. The results will then include a CSV file that stores the predictions obtained in the evaluations that gave the best score. The file content includes the data in your original dataset tagged by k-fold and the prediction and confidence obtained. This file will be placed in an internal folder of your chosen output directory.

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --features --output-dir my_features --predictions-csv
```

The output directory for this command is `my_features` and it will contain all the information about the resources generated when testing the different feature combinations organized in subfolders. The k-fold datasets' IDs will be stored in an inner `test` directory. The IDs of the resources created when testing each combination of features will be stored in `kfold1`, `kfold2`, etc. folders inside the `test` directory. If the best-scoring prediction models are the ones in the `kfold4` folder, then the predictions CSV file will be stored in a new folder named `kfold4_pred`.

Report subcommand

The results of a `bigmler analyze --features` or `bigmler analyze --nodes` command are a series of k-fold cross-validations made on the training data that leads to the configuration value that will create the best performant model. However, the algorithm maximizes only one evaluation metric. To see the global picture for the rest of metrics at each validation configuration you can build a graphical report of the results using the `report` subcommand. Let's say you previously ran

```
bigmler analyze --dataset dataset/5357eb2637203f1668000004 \
                --nodes --output-dir best_recall
```

and you want to have a look at the results for each `node_threshold` configuration. Just say:

```
bigmler report --from-dir best_recall --port 8080
```

and the command will traverse the directories in `best_recall` and summarize the results found there in a metrics comparison graphic and an ROC curve if your model is categorical. Then a simple HTTP server will be started locally and bound to a port of your choice, `8080` in the example (`8085` will be the default value), and a new web browser window will be started to show the results. You can see an example built on the well known diabetes dataset.

The HTTP server will create an auxiliary `bigmler/reports` directory in the user's home directory, where symbolic links to the reports in each output directory will be stored and served from.

Cluster subcommand

Just as the simple `bigmler` command can generate all the resources leading to finding models and predictions for a supervised learning problem, the `bigmler cluster` subcommand will follow the steps to generate clusters and predict the centroids associated to your test data. To mimic what we saw in the `bigmler` command section, the simplest call is

```
bigmler cluster --train data/diabetes.csv
```

This command will upload the data in the `data/diabetes.csv` file and generate the corresponding `source`, `dataset` and `cluster` objects in BigML. You can use any of the generated objects to produce new clusters. For instance, you could set a subgroup of the fields of the generated dataset to produce a different cluster by using

```
bigmler cluster --dataset dataset/53b1f71437203f5ac30004ed \
                --cluster-fields="-blood pressure"
```

that would exclude the field `blood pressure` from the cluster creation input fields.

Similarly to the models and datasets, the generated clusters can be shared using the `--shared` option, e.g.

```
bigmler cluster --source source/53b1f71437203f5ac30004e0 \
                --shared
```

will generate a secret link for both the created dataset and cluster that can be used to share the resource selectively.

As models were used to generate predictions (class names in classification problems and an estimated number for regressions), clusters can be used to predict the subgroup of data that our input data is more similar to. Each subgroup is represented by its centroid, and the centroid is labelled by a centroid name. Thus, a cluster would classify our test data by assigning to each input an associated centroid name. The command

```
bigmler cluster --cluster cluster/53b1f71437203f5ac30004f0 \
                --test data/my_test.csv
```

would produce a file `centroids.csv` with the centroid name associated to each input. When the command is executed, the cluster information is downloaded to your local computer and the centroid predictions are computed locally, with no more latencies involved. Just in case you prefer to use BigML to compute the centroid predictions remotely, you can do so too

```
bigmler cluster --cluster cluster/53b1f71437203f5ac30004f0 \
                --test data/my_test.csv --remote
```

would create a remote source and dataset from the test file data, generate a `batch centroid` also remotely and finally download the result to your computer. If you prefer the result not to be dowloaded but to be stored as a new dataset remotely, add `--no-csv` and `to-dataset` to the command line. This can be specially helpful when dealing with a high number of scores or when adding to the final result the original dataset fields with `--prediction-info full`, that may result in a large CSV to be created as output.

The k-means algorithm used in clustering can only use training data that has no missing values in their numeric fields. Any data that does not comply with that is discarded in cluster construction, so you should ensure that enough number of rows in your training data file has non-missing values in their numeric fields for the cluster to be built and relevant. Similarly, the cluster cannot issue a centroid prediction for input data that has missing values in its numeric fields, so centroid predictions will give a "-" string as output in this case.

You can change the number of centroids used to group the data in the clustering procedure

```
bigmler cluster --dataset dataset/53b1f71437203f5ac30004ed \
                --k 3
```

And also generate the datasets associated to each centroid of a cluster. Using the `--cluster-datasets` option

bigmler cluster –cluster cluster/53b1f71437203f5ac30004f0 –cluster-datasets "Cluster 1,Cluster 2"

you can generate the datasets associated to a comma-separated list of centroid names. If no centroid name is provided, all datasets are generated.

Similarly, you can generate the models to predict if one instance is associated to each centroid of a cluster. Using the `--cluster-models` option

bigmler cluster –cluster cluster/53b1f71437203f5ac30004f0 –cluster-models "Cluster 1,Cluster 2"

you can generate the models associated to a comma-separated list of centroid names. If no centroid name is provided, all models are generated. Models can be useful to see which features are important to determine whether a certain instance belongs to a concrete cluster.

Anomaly subcommand

The `bigmler anomaly` subcommand generates all the resources needed to buid an anomaly detection model and/or predict the anomaly scores associated to your test data. As usual, the simplest call

```
bigmler anomaly --train data/tiny_kdd.csv
```

uploads the data in the `data/tiny_kdd.csv` file and generates the corresponding `source`, `dataset` and `anomaly` objects in BigML. You can use any of the generated objects to produce new anomaly detectors. For instance, you could set a subgroup of the fields of the generated dataset to produce a different anomaly detector by using

```
bigmler anomaly --dataset dataset/53b1f71437203f5ac30004ed \
                --anomaly-fields="-urgent"
```

that would exclude the field `urgent` from the anomaly detector creation input fields. You can also change the number of top anomalies enclosed in the anomaly detector list and the number of trees that the anomaly detector iforest uses. The default values are 10 top anomalies and 128 trees per iforest:

```
bigmler anomaly --dataset dataset/53b1f71437203f5ac30004ed \
                --top-n 15 --forest-size 50
```

with this code, the anomaly detector is built using an iforest of 50 trees and will produce a list of the 15 top anomalies.

Similarly to the models and datasets, the generated anomaly detectors can be shared using the `--shared` option, e.g.

```
bigmler anomaly --source source/53b1f71437203f5ac30004e0 \
                --shared
```

will generate a secret link for both the created dataset and anomaly detector that can be used to share the resource selectively.

The anomaly detector can be used to assign an anomaly score to each new input data set. The anomaly score is a number between 0 (not anomalous) and 1 (highest anomaly). The command

```
bigmler anomaly --anomaly anomaly/53b1f71437203f5ac30005c0 \
                --test data/test_kdd.csv
```

would produce a file `anomaly_scores.csv` with the anomaly score associated to each input. When the command is executed, the anomaly detector information is downloaded to your local computer and the anomaly score predictions are computed locally, with no more latencies involved. Just in case you prefer to use BigML to compute the anomaly score predictions remotely, you can do so too

```
bigmler anomaly --anomaly anomaly/53b1f71437203f5ac30005c0 \
                --test data/my_test.csv --remote
```

would create a remote source and dataset from the test file data, generate a `batch anomaly score` also remotely and finally download the result to your computer. If you prefer the result not to be dowloaded but to be stored as a new dataset remotely, add `--no-csv` and `to-dataset` to the command line. This can be specially helpful when dealing with a high number of scores or when adding to the final result the original dataset fields with `--prediction-info full`, that may result in a large CSV to be created as output.

Similarly, you can split your data in train/test datasets to build the anomaly detector and create batch anomaly scores with the test portion of data

```
bigmler anomaly --train data/tiny_kdd.csv --test-split 0.2 --remote
```

or if you want to apply the anomaly detector on the same training data set to create a batch anomaly score, use:

```
bigmler anomaly --train data/tiny_kdd.csv --score --remote
```

To extract the top anomalies as a new dataset, or to exclude from the training dataset the top anomalies in the anomaly detector, set the

`--anomalies-dataset` to `in` or `out` respectively:

```
bigmler anomaly --dataset dataset/53b1f71437203f5ac30004ed \
                --anomalies-dataset out
```

will create a new dataset excluding the top anomalous instances according to the anomaly detector.

Sample subcommand

You can extract samples from your datasets in BigML using the `bigmler sample` subcommand. When a new sample is requested, a copy of the dataset is stored in a special format in an in-memory cache. This sample can then be used, before its expiration time, to extract data from the related dataset by setting some options like the number of rows or the fields to be retrieved. You can either begin from scratch uploading your data to BigML, creating the corresponding source and dataset and extracting your sample from it

```
bigmler sample --train data/iris.csv --rows 10 --row-offset 20
```

This command will create a source, a dataset, a sample object, whose id will be stored in the `samples` file in the output directory, and extract 10 rows of data starting from the 21st that will be stored in the `sample.csv` file.

You can reuse an existing sample by using its id in the command.

```
bigmler sample --sample sample/53b1f71437203f5ac303d5c0 \
               --sample-header --row-order-by="-petal length" \
               --row-fields "petal length,petal width" --mode linear
```

will create a new `sample.csv` file with a headers row where only the `petal length` and `petal width` are retrieved. The `--mode linear` option will cause the first available rows to be returned and the `--row-order-by="-petal length"` option returns these rows sorted in descending order according to the contents of `petal length`.

You can also add to the sample rows some statistical information by using the `--stat-field` or `--stat-fields` options. Adding them to the command will generate a `stat-info.json` file where the Pearson's and Spearman's correlations, and linear regression terms will be stored in a JSON format.

You can also apply a filter to select the sample rows by the values in their fields using the `--fields-filter` option. This must be set to a string containing the conditions that must be met using field ids and values.

```
bigmler sample --sample sample/53b1f71437203f5ac303d5c0 \
               --fields-filter "000001=&!000004=Iris-setosa"
```

With this command, only rows where field id `000001` is missing and field id `000004` is not `Iris-setosa` will be retrieved. You can check the available operators and syntax in the samples' developers doc . More available options can be found in the *Samples subcommand Options* section.

Reify subcommand

This subcommand extracts the information in the existing resources to determine the arguments that were used when they were created, and generates scripts that could be used to reproduce them. Currently, the language used in the scripts will be `Python`. The usual starting point for BigML resources is a `source` created from inline, local or remote data. Thus, the script keeps analyzing the chain of calls that led to a certain resource until the root `source` is found.

The simplest example would be:

```
bigmler reify --id source/55d77ba60d052e23430027bb
```

that will output:

```
"""Python code to reify source/55d77ba60d052e23430027bb

"""

from bigml.api import BigML
api = BigML()

source1 = api.create_source("iris.csv", {"name": "my source"})
api.ok(source1)
```

According to this output, the source was created from a file named `iris.csv` and was assigned a `name`. This script will be stored in the command output directory and named *reify.py*‘ (you can specify a different name and location using the `--output` option).

When creating sources from data, field types are inferred from the contents of the first lines in the uploaded file. Sometimes, these field types must be adapted and the `source` fields attributes are updated. You can also change other fields attributes, like their name, label or description. In order to make sure that the right fields information is reproduced, add the `--add-fields` flag:

```
bigmler reify --id source/55d77ba60d052e23430027bb --add-fields \
              --output my_dir/reify_source.py
```

```
"""Python code to reify source/55d77ba60d052e23430027bb

"""

from bigml.api import BigML
api = BigML()

source1 = api.create_source("iris.csv")
api.ok(source1)

source1 = api.update_source(source1, \
    {'fields': {u'000004': {'optype': u'categorical', 'name': u'species'},
                u'000002': {'optype': u'numeric', 'name': u'petal length'},
                u'000003': {'optype': u'numeric', 'name': u'petal width'},
                u'000000': {'optype': u'numeric', 'name': u'sepal length'},
                u'000001': {'optype': u'numeric', 'name': u'sepal width'}}
    }
)
api.ok(source1)
```

Other resources will have more complex workflows and more user-given attributes. Let’s see for instance the script

to generate an evaluation from a train/test split of a source that was created using the `bigmler --train data/iris.csv --evaluate` command:

```
bigmler reify --id evaluation/55d919850d052e234b000833
```

```
"""Python code to reify evaluation/55d919850d052e234b000833

"""

from bigml.api import BigML
api = BigML()

source1 = api.create_source("iris.csv", {'category': 12,
    'description': u'Created using BigMLer',
    'name': u'BigMLer_SunAug2315_025314',
    'tags': [u'BigMLer', u'BigMLer_SunAug2315_025314']})
api.ok(source1)

dataset1 = api.create_dataset(source1,
    {'name': u'BigMLer_SunAug2315_025314',
     'tags': [u'BigMLer', u'BigMLer_SunAug2315_025314']})
api.ok(dataset1)

model1 = api.create_model(dataset1,
    {'seed': u'BigML, Machine Learning made easy',
     'sample_rate': 0.8, 'name': u'BigMLer_SunAug2315_025314'})
api.ok(model1)

evaluation1 = api.create_evaluation(model1, dataset1,
    {'seed': u'BigML, Machine Learning made easy', 'sample_rate': 0.8,
     'out_of_bag': True, 'name': u'BigMLer_SunAug2315_025314'})
api.ok(evaluation1)
```

As you can see, BigMLer has added a default `category`, `name`, `description`, `tags`, has built the model on 80% of the data and used the `out_of_bag` attribute for the evaluation to use the remaining part of the dataset test data.

Execute subcommand

This subcommand creates and executes scripts in WhizzML (BigML's automation language). With WhizzML you can program any specific workflow that involves Machine Learning resources like datasets, models, etc. You just write a script using the directives in the reference manual and upload it to BigML, where it will be available as one more resource in your dashboard. Scripts can also be shared and published in the gallery, so you can reuse other users' scripts and execute them. These operations can also be done using the *bigmler execute* subcommand.

The simplest example is executing some basic code, like adding two numbers:

```
bigmler execute --code "(+ 1 2)" --output-dir simple_exe
```

With this command, bigmler will generate a script in BigML whose source code is the one given as a string in the `--code` option. The script ID will be stored in a file called `scripts` in the `simple_text` directory. After that, the script will be executed, so a new resource called `execution` will be created in BigML, and the corresponding ID will be stored in the `execution` file of the output directory. Similarly, the result of the execution will be stored in `whizzml_results.txt` and `whizzml_results.json` (in human-readable format and JSON respectively) in

the directory set in the `--output-dir` option. You can also use the code stored in a file with the `--code-file` option.

Adding the `--no-execute` flag to the command will cause the process to stop right after the script creation. You can also compile your code as a library to be used in many scripts by setting the `--to-library` flag.

```
bigmler execute --code-file my_library.whizzml --to-library
```

Existing scripts can be referenced for execution with the `--script` option

```
bigmler execute --script script/50a2bb64035d0706db000643
```

or the script ID can be read from a file:

```
bigmler execute --scripts simple_exe/scripts
```

The script we used as an example is very simple and needs no additional parameter. But, in general, scripts will have input parameters and output variables. The inputs define the script signature and must be declared in order to create the script. The outputs are optional and any variable in the script can be declared to be an output. Both inputs and outputs can be declared using the `--declare-inputs` and `--declare-outputs` options. These options must contain the path to the JSON file where the information about the inputs and outputs (respectively) is stored.

```
bigmler execute --code '(define addition (+ a b))' \
                --declare-inputs my_inputs_dec.json \
                --declare-outputs my_outputs_dec.json \
                --no-execute
```

in this example, the `my_inputs_dec.json` file could contain

```
[{"name": "a",
  "default": 0,
  "type": "number"},
 {"name": "b",
  "default": 0,
  "type": "number",
  "description": "second number to add"}]
```

and `my_outputs_dec.json`

```
[{"name": "addition",
  "type": "number"}]
```

so that the value of the `addition` variable would be returned as output in the execution results.

Additionally, a script can import libraries. The list of libraries to be used as imports can be added to the command with the option `--imports` followed by a comma-separated list of library IDs.

Once the script has been created and its inputs and outputs declared, to execute it you'll need to provide a value for each input. This can be done using `--inputs`, that will also point to a JSON file where each input should have its corresponding value.

```
bigmler execute --script script/50a2bb64035d0706db000643 \
                --inputs my_inputs.json
```

where the `my_inputs.json` file would contain:

```
[["a", 1],
 ["b", 2]]
```

For more details about the syntax to declare inputs and outputs, please refer to the Developers documentation.

You can also provide default configuration attributes for the resources generated in an execution. Add the `--creation-defaults` option followed by the path to a JSON file that contains a dictionary whose keys are the resource types to which the configuration defaults apply and whose values are the configuration attributes set by default.

```
bigmler execute --code-file my_script.whizzml \
                --creation-defaults defaults.json
```

For instance, if `my_script.whizzml` creates an ensemble from a remote file:

```
(define file "s3://bigml-public/csv/iris.csv")
(define source (create-and-wait-source {"remote" file}))
(define dataset (create-and wait-dataset {"source" source}))
(define ensemble (create-and-wait-ensemble {"dataset" dataset}))
```

and `my_create_defaults.json` contains

```
{
    "source": {
    "project": "project/54d9553bf0a5ea5fc0000016"
    },
    "ensemble": {
    "number_of_models": 100, "sample_rate": 0.9
    }
}
```

the source created by the script will be associated to the given project and the ensemble will have 100 models and a 0.9 sample rate unless the source code in your script explicitly specifies a different value, in which case it takes precedence over these defaults.

Whizzml subcommand

This subcommand creates packages of scripts and libraries in WhizzML (BigML's automation language) based on the information provided by a `metadata.json` file. These operations can also be performed individually using the *bigmler execute* subcommand, but *bigmler whizzml* reads the components of the package, and for each component analyzes the corresponding `metadata.json` file to identify the kind of code (script or library) that it contains and creates the corresponding resource in BigML. The `metadata.json` is expected to contain the name, kind, description, inputs and outputs needed to create the script. As an example,

```
{
  "name": "Example of whizzml script",
  "description": "Test example of a whizzml script that adds two numbers",
  "kind": "script",
  "source_code": "code.whizzml",
  "inputs": [
      {
          "name": "a",
          "type": "number",
          "description": "First number"
      },
      {
          "name": "b",
          "type": "number",
          "description": "Second number"
```

```
        }
    ],
    "outputs": [
        {
            "name": "addition",
            "type": "number",
            "description": "Sum of the numbers"
        }
    ]
}
```

describes a script whose code is to be found in the `code.whizzml` file. The script will have two inputs `a` and `b` and one output: `addition`.

In order to create this script, you can type the following command:

```
bigmler whizzml --package-dir my_package --output-dir creation_log
```

and bigmler will:

- look for the `metadata.json` file located in the `my_package` directory.
- parse the JSON, identify that it defines a script and look for its code in the `code.whizzml` file
- create the corresponding BigML script resource, adding as arguments the ones provided in `inputs`, `outputs`, `name` and `description`.

Packages can contain more than one script. In this case, a nested directory structure is expected. The `metadata.json` file for a package with many components should include the name of the directories where these components can be found:

```
{
  "name": "Best k",
  "description": "Library and scripts implementing Pham-Dimov-Nguyen k selection␣
↪algorithm",
  "kind": "package",
  "components":[
    "best-k-means",
    "cluster",
    "evaluation",
    "batchcentroid"
  ]
}
```

In this example, each string in the `components` attributes list corresponds to one directory where a new script or library (with its corresponding `metadata.json` descriptor) is stored. Then, using `bigmler whizzml` for this composite package will create each of the component scripts or libraries. It will also handle dependencies, using the IDs of the created libraries as imports for the scripts when needed.

Delete subcommand

You have seen that BigMLer is an agile tool that empowers you to create a great number of resources easily. This is a tremedous help, but it also can lead to a garbage-prone environment. To keep a control of each new created remote resource use the flag *–resources-log* followed by the name of the log file you choose.

```
bigmler --train data/iris.csv --resources-log my_log.log
```

Each new resource created by that command will cause its id to be appended as a new line of the log file.

BigMLer can help you as well in deleting these resources. Using the *delete* subcommand there are many options available. For instance, deleting a comma-separated list of ids

```
bigmler delete \
       --ids source/50a2bb64035d0706db0006cc,dataset/50a1f441035d0706d9000371
```

deleting resources listed in a file

```
bigmler delete --from-file to_delete.log
```

where *to_delete.log* contains a resource id per line.

As we've previously seen, each BigMLer command execution generates a bunch of remote resources whose ids are stored in files located in a directory that can be set using the `--output-dir` option. The `bigmler delete` subcommand can retrieve the ids stored in such files by using the `--from-dir` option.

```
bigmler --train data/iris.csv --output my_BigMLer_output_dir
bigmler delete --from-dir my_BigMLer_output_dir
```

The last command will delete all the remote resources previously generated by the fist command by retrieving their ids from the files in `my_BigMLer_output_dir` directory.

You can also delete resources based on the tags they are associated to

```
bigmler delete --all-tag my_tag
```

or restricting the operation to a specific type

```
bigmler delete --source-tag my_tag
bigmler delete --dataset-tag my_tag
bigmler delete --model-tag my_tag
bigmler delete --prediction-tag my_tag
bigmler delete --evaluation-tag my_tag
bigmler delete --ensemble-tag my_tag
bigmler delete --batch-prediction-tag my_tag
bigmler delete --cluster-tag my_tag
bigmler delete --centroid-tag my_tag
bigmler delete --batch-centroid-tag my_tag
bigmler delete --anomaly-tag my_tag
bigmler delete --anomaly-score-tag my_tag
bigmler delete --batch-anomaly-score-tag my_tag
bigmler delete --project-tag my_tag
bigmler delete --association-tag my_tag
```

You can also delete resources by date. The options `--newer-than` and `--older-than` let you specify a reference date. Resources created after and before that date respectively, will be deleted. Both options can be combined to set a range of dates. The allowed values are:

- dates in a YYYY-MM-DD format
- integers, that will be interpreted as number of days before now
- resource id, the creation datetime of the resource will be used

Thus,

```
bigmler delete --newer-than 2
```

will delete all resources created less than two days ago (now being 2014-03-23 14:00:00.00000, its creation time will be greater than 2014-03-21 14:00:00.00000).

```
bigmler delete --older-than 2014-03-20 --newer-than 2014-03-19
```

will delete all resources created during 2014, March the 19th (creation time between 2014-03-19 00:00:00 and 2014-03-20 00:00:00) and

```
bigmler delete --newer-than source/532db2b637203f3f1a000104
```

will delete all resources created after the `source/532db2b637203f3f1a000104` was created.

You can also combine both types of options, to delete sources tagged as `my_tag` starting from a certain date on

```
bigmler delete --newer-than 2 --source-tag my_tag
```

And finally, you can filter the type of resource to be deleted using the `--resource-types` option to specify a comma-separated list of resource types to be deleted

```
bigmler delete --older-than 2 --resource-types source,model
```

will delete the sources and models created more than two days ago.

You can simulate the a delete subcommand using the `--dry-run` flag

```
bigmler delete --newer-than source/532db2b637203f3f1a000104 \
               --source-tag my_source --dry-run
```

The output for the command will be a list of resources that would be deleted if the `--dry-run` flag was removed. In this case, they will be sources that contain the tag `my_source` and were created after the one given as `--newer-than` value. The first 15 resources will be logged to console, and the complete list can be found in the `bigmler_sessions` file.

By default, only finished resources are selected to be deleted. If you want to delete other resources, you can select them by choosing their status:

```
bigmler delete --older-than 2 --status failed
```

would remove all failed resources created more than two days ago.

Export subcommand

The `bigmler export` subcommand is intended to help generating the code needed for the models in BigML to be integrated in other applications. To produce a prediction using a BigML model you just need a function that receives as argument the new test case data and returns this prediction (and a confidence). The *bigmler export* subcommand will retrieve the JSON information of your existing decision tree model in BigML and will generate from it this function code and store it in a file that can be imported or copied directly in your application.

Obviously, the function syntax will depend on the model and the language used in your application, so these will be the options we need to provide:

```
bigmler export --model model/532db2b637203f3f1a001304 \
               --language javascript --output-dir my_exports
```

This command will create a javascript version of the function that produces the predictions and store it in a file named *model_532db2b637203f3f1a001304.js* (after the model ID) in the *my_exports* directory.

Models can currently exported in *Python*, *Javascript* and *R*. For models whose fields are numeric or categorical, the command also supports creating *MySQL* functions and *Tableau* separate expressions for both the prediction and the confidence.

Project subcommand

Projects are organizational resources and they are usually created at source-creation time in order to keep together in a separate repo all the resources derived from a source. However, you can also create a project or update its properties independently using the `bigmler project` subcommand.

```
bigmler project --name my_project
```

will create a new project and name it. You can also add other attributes such as `--tag`, `--description` or `--category` in the project creation call. You can also add or update any other attribute to the project using a JSON file with the `--project-attributes` option.

```
bigmler project --project-id project/532db2b637203f3f1a000153 \
                --project-attributes my_attributes.json
```

Association subcommand

Association Discovery is a popular method to find out relations among values in high-dimensional datasets.

A common case where association discovery is often used is market basket analysis. This analysis seeks for customer shopping patterns across large transactional datasets. For instance, do customers who buy hamburgers and ketchup also consume bread?

Businesses use those insights to make decisions on promotions and product placements. Association Discovery can also be used for other purposes such as early incident detection, web usage analysis, or software intrusion detection.

In BigML, the Association resource object can be built from any dataset, and its results are a list of association rules between the items in the dataset. In the example case, the corresponding association rule would have hamburguers and ketchup as the items at the left hand side of the association rule and bread would be the item at the right hand side. Both sides in this association rule are related, in the sense that observing the items in the left hand side implies observing the items in the right hand side. There are some metrics to ponder the quality of these association rules:

- Support: the proportion of instances which contain an itemset.

For an association rule, it means the number of instances in the dataset which contain the rule's antecedent and rule's consequent together over the total number of instances (N) in the dataset.

It gives a measure of the importance of the rule. Association rules have to satisfy a minimum support constraint (i.e., min_support).

- Coverage: the support of the antedecent of an association rule.

It measures how often a rule can be applied.

- Confidence or (strength): The probability of seeing the rule's consequent

under the condition that the instances also contain the rule's antecedent. Confidence is computed using the support of the association rule over the coverage. That is, the percentage of instances which contain the consequent and antecedent together over the number of instances which only contain the antecedent.

Confidence is directed and gives different values for the association rules Antecedent → Consequent and Consequent → Antecedent. Association rules also need to satisfy a minimum confidence constraint (i.e., min_confidence).

- Leverage: the difference of the support of the association

rule (i.e., the antecedent and consequent appearing together) and what would be expected if antecedent and consequent where statistically independent. This is a value between -1 and 1. A positive value suggests a positive relationship and a negative value suggests a negative relationship. 0 indicates independence.

Lift: how many times more often antecedent and consequent occur together than expected if they where statistically independent. A value of 1 suggests that there is no relationship between the antecedent and the consequent. Higher values suggest stronger positive relationships. Lower values suggest stronger negative relationships (the presence of the antecedent reduces the likelihood of the consequent)

As to the items used in association rules, each type of field is parsed to extract items for the rules as follows:

- Categorical: each different value (class) will be considered a separate item.
- Text: each unique term will be considered a separate item.
- Items: each different item in the items summary will be considered.
- Numeric: Values will be converted into categorical by making a

segmentation of the values. For example, a numeric field with values ranging from 0 to 600 split into 3 segments: segment 1 → [0, 200), segment 2 → [200, 400), segment 3 → [400, 600]. You can refine the behavior of the transformation using discretization and field_discretizations.

The `bigmler association` subcommand will discover the association rules present in your datasets. Starting from the raw data in your files:

```
bigmler association --train my_file.csv
```

will generate the `source`, `dataset` and `association` objects required to present the association rules hidden in your data. You can also limit the number of rules extracted using the `--max-k` option

```
bigmler association --dataset dataset/532db2b637203f3f1a000103 \
                    --max-k 20
```

With the prior command only 20 association rules will be extracted. Similarly, you can change the search strategy used to find them

```
bigmler association --dataset dataset/532db2b637203f3f1a000103 \
                    --search-strategy confidence
```

In this case, the `confidence` is used (the default value being `leverage`).

Logistic-regression subcommand

The `bigmler logistic-regression` subcommand generates all the resources needed to buid a logistic regression model and use it to predict. The logistic regression model is a supervised learning method for solving classification problems. It predicts the objective field class as logistic function whose argument is a linear combination of the rest of features. The simplest call to build a logistic regression is

```
bigmler logistic-regression --train data/iris.csv
```

uploads the data in the `data/iris.csv` file and generates the corresponding `source`, `dataset` and `logistic regression` objects in BigML. You can use any of the generated objects to produce new logistic regressions. For instance, you could set a subgroup of the fields of the generated dataset to produce a different logistic regression model by using

```
bigmler logistic-regression --dataset dataset/53b1f71437203f5ac30004ed \
               --logistic-fields="-sepal length"
```

that would exclude the field `sepal length` from the logistic regression model creation input fields. You can also change some parameters in the logistic regression model, like the `bias` (scale of the intercept term), `c` (the strength of the regularization map) or `eps` (stopping criteria for solver).

```
bigmler logistic-regression --dataset dataset/53b1f71437203f5ac30004ed \
                            --bias 1 --c 5 --eps 0.5
```

with this code, the logistic regression is built using an independent term of 1, the step in the regularization is 5 and the difference between the results from the current and last iterations is 0.5.

Similarly to the models and datasets, the generated logistic regressions can be shared using the `--shared` option, e.g.

```
bigmler logistic-regression --source source/53b1f71437203f5ac30004e0 \
                            --shared
```

will generate a secret link for both the created dataset and logistic regressions, that can be used to share the resource selectively.

The logistic regression can be used to assign a prediction to each new input data set. The command

```
bigmler logistic-regression \
        --logistic-regression logisticregression/53b1f71435203f5ac30005c0 \
        --test data/test_iris.csv
```

would produce a file `predictions.csv` with the predictions associated to each input. When the command is executed, the logistic regression information is downloaded to your local computer and the logistic regression predictions are computed locally, with no more latencies involved. Just in case you prefer to use BigML to compute the predictions remotely, you can do so too

```
bigmler logistic-regression
        --logistic-regression logisticregression/53b1f71435203f5ac30005c0 \
        --test data/my_test.csv --remote
```

would create a remote source and dataset from the test file data, generate a `batch prediction` also remotely and finally download the result to your computer. If you prefer the result not to be dowloaded but to be stored as a new dataset remotely, add `--no-csv` and `to-dataset` to the command line. This can be specially helpful when dealing with a high number of scores or when adding to the final result the original dataset fields with `--prediction-info full`, that may result in a large CSV to be created as output.

Topic Model subcommand

Using this subcommand you can generate all the resources leading to finding a `topic model` and its `topic distributions`. These are unsupervised learning models which find out the topics in a collection of documents and will then be useful to classify new documents according to the topics. The `bigmler topic-model` subcommand will follow the steps to generate `topic models` and predict the `topic distribution`, or distribution of probabilities for the new document to be associated to a certain topic. As shown in the `bigmler` command section, the simplest call is

```
bigmler topic-model --train data/spam.csv
```

This command will upload the data in the `data/spam.csv` file and generate the corresponding `source`, `dataset` and `topic model` objects in BigML. You can use any of the intermediate generated objects to produce new topic models. For instance, you could set a subgroup of the fields of the generated dataset to produce a different topic model by using

```
bigmler topic-model --dataset dataset/53b1f71437203f5ac30004ed \
                    --topic-fields="-Message"
```

that would exclude the field `Message` from the topic model creation input fields.

Similarly to the models and datasets, the generated topic models can be shared using the `--shared` option, e.g.

```
bigmler topic-model --source source/53b1f71437203f5ac30004e0 \
                    --shared
```

will generate a secret link for both the created dataset and topic model that can be used to share the resource selectively.

As models were used to generate predictions (class names in classification problems and an estimated number for regressions), topic models can be used to classify a new document in the discovered list of topics. The classification is run by computing the probability for the document to belonging to the topic group. The command

```
bigmler topic-model --topic-model topicmodel/58437a277e0a8d38ec028a5f \
                    --test data/my_test.csv
```

would produce a file `topic_distributions.csv` where each row will contain the probabilities associated to each topic for the corresponding test input. When the command is executed, the topic model information is downloaded to your local computer and the distributions are computed locally, with no more latencies involved. Just in case you prefer to use BigML to compute the topic distributions remotely, you can do so too

```
bigmler topic-model --topic-model topicmodel/58437a277e0a8d38ec028a5f \
                    --test data/my_test.csv --remote
```

would create a remote source and dataset from the test file data, generate a `batch topic distribution` also remotely and finally download the result to your computer. If you prefer the result not to be dowloaded but to be stored as a new dataset remotely, add `--no-csv` and `to-dataset` to the command line. This can be specially helpful when dealing with a high number of scores or when adding to the final result the original dataset fields with `--prediction-info full`, that may result in a large CSV to be created as output.

Time Series subcommand

Using this subcommand you can generate all the resources leading to a `time series` and its `forecasts`. The `time series` is a supervised learning model that works on an ordered sequence of data to extract the patterns needed to make `forecasts`. The `bigmler time-series` subcommand will follow the steps to generate `time series` and predict the `forecasts` for every numeric field in the original dataset that has been set as objective field. As shown in the `bigmler` command section, the simplest call is

```
bigmler time-series --train data/grades.csv
```

This command will upload the data in the `data/grades.csv` file and generate the corresponding `source`, `dataset` and `time series` objects in BigML. You can use any of the intermediate generated objects to produce new time series. For instance, you could set a subgroup of the numeric fields in the dataset to be used as objective fields using the `--objectives` option.

```
bigmler time-series --dataset dataset/53b1f71437203f5ac30004ed \
                    --objectives "Assignment,Final"
```

its value is expected to be a comma-separated list of fields.

Similarly to the models and datasets, the generated clusters can be shared using the `--shared` option, e.g.

```
bigmler time-series --source source/53b1f71437203f5ac30004e0 \
                    --shared
```

will generate a secret link for both the created dataset and time series that can be used to share the resource selectively.

As models were used to generate predictions (class names in classification problems and an estimated number for regressions), time series can be used to generate forecasts, that is, to predict the value of each objective field up till the user-given horizon. The command

```
bigmler time-series --time-series timeseries/58437a277e0a8d38ec028a5f \
                    --horizon 10
```

would produce a file `forecast_000001.csv` with ten rows, one per point, and as many columns as ETS models the time series contains.

When the command is executed, the time series information is downloaded to your local computer and the forecasts are computed locally, with no more latencies involved. Just in case you prefer to use BigML to compute the forecasts remotely, you can do so too

```
bigmler time-series --time-series timeseries/58437a277e0a8d38ec028a5f \
                    --horizon 10 --remote
```

would create a remote forecast with the specified horizon. You can also specify more complex inputs for the forecast. For instance, you can set a different horizon to each objective field and you can give some criteria to select the models used in the forecast. All of this can be done using the `--test` option pointing to a JSON file that should contain the input to be used in the forecast as described in the API documentation. As an example, let's set a horizon of 5 points for the `Final` field and select the first model in the time series array of ETS models, and also forecast 7 points for the `Assignment` field using the model with less `aic` (the one used by default). The command call should then be:

```
bigmler time-series --time-series timeseries/58437a277e0a8d38ec028a5f \
                    --test test.json
```

and the `test.json` file should contain the following JSON:

```
{"Final": {"horizon": 5, "ets_models": {"indices": [0]}},
 "Assignment": {"horizon": 7}}
```

CHAPTER 4

Additional Features

Using local models to predict

Most of the previously described commands need the remote resources to be downloaded to work. For instance, when you want to create a new model from an existing dataset, BigMLer is going to download the dataset JSON structure to extract the fields and objective field information, and only then ask for the model creation. As mentioned, the `--store` flag forces BigMLer to store the downloaded JSON structures in local files inside your output directory. If you use that flag when building a model with BigMLer, then the model is stored in your computer. This model file contains all the information you need in order to make new predictions, so you can use the `--model-file` option to set the path to this file and predict the value of your objective field for new input data with no reference at all to your remote resources. You could even delete the original remote model and work exclusively with the locally downloaded file

```
bigmler --model-file my_dir/model_532db2b637203f3f1a000136 \
        --test data/test_iris.csv
```

The same is available for clusters

```
bigmler cluster --cluster-file my_dir/cluster_532db2b637203f3f1a000348 \
                --test data/test_diabetes.csv
```

anomaly detectors

```
bigmler anomaly --anomaly-file my_dir/anomaly_532db2b637203f3f1a00053a \
                --test data/test_kdd.csv
```

logistic regressions

```
bigmler logistic-regression --logistic-file my_dir/logisticregression_
↪532db2b637203f3f1a00053a \
                --test data/test_diabetes.csv
```

topic models

```
bigmler topic-model --topic-model-file my_dir/topicmodel_532db2b637203f3f1a00053a \
                    --test data/test_spam.csv
```

time series

```
bigmler time-series --time-series-file my_dir/timeseries_532db2b637203f5f1a00053a \
                    --horizon 20
```

Even for ensembles

```
bigmler --ensemble-file my_dir/ensemble_532db2b637203f3f1a00053b \
        --test data/test_iris.csv
```

In this case, the models included in the ensemble are expected to be stored also in the same directory where the local file for the ensemble is. They are downloaded otherwise.

Resuming Previous Commands

Network connections failures or other external causes can break the BigMLer command process. To resume a command ended by an unexpected event you can issue

```
bigmler --resume
```

BigMLer keeps track of each command you issue in a `.bigmler` file and of the output directory in `.bigmler_dir_stack` of your working directory. Then `--resume` will recover the last issued command and try to continue work from the point it was stopped. There's also a `--stack-level` flag

```
bigmler --resume --stack-level 1
```

to allow resuming a previous command in the stack. In the example, the one before the last.

Building reports

The resources generated in the execution of a BigMLer command are listed in the standard output by default, but they can be summarized as well in a `Gazibit` format. Gazibit is a platform where you can create interactive presentations in a flexible and dynamic way. Using BigMLer's `--reports gazibit` option you'll be able to generate a `Gazibit` summary report of your newly created resources. In case you use also the `--shared` flag, a second template will be generated where the links for the shared resources will be used. Both reports will be stored in the `reports` subdirectory of your output directory, where all of the files generated by the BigMLer command are. Thus,

```
bigmler --train data/iris.csv --reports gazibit --shared \
        --output-dir my_dir
```

will generate two files: `gazibit.json` and `gazibit_shared.json` in a `reports` subdirectory of your `my_dir` directory. In case you provide your `Gazibit token` in the `GAZIBIT_TOKEN` environment variable, they will also be uploaded to your account in `Gazibit`. Upload can be avoided, by using the `--no-upload` flag.

User Chosen Defaults

BigMLer will look for `bigmler.ini` file in the working directory where users can personalize the default values they like for the most relevant flags. The options should be written in a config style, e.g.

```
[BigMLer]
dev = true
resources_log = ./my_log.log
```

as you can see, under a `[BigMLer]` section the file should contain one line per option. Dashes in flags are transformed to undescores in options. The example would keep development mode on and would log all created resources to `my_log.log` for any new `bigmler` command issued under the same working directory if none of the related flags are set.

Naturally, the default value options given in this file will be overriden by the corresponding flag value in the present command. To follow the previous example, if you use

```
bigmler --train data/iris.csv --resources-log ./another_log.log
```

in the same working directory, the value of the flag will be preeminent and resources will be logged in `another_log.log`. For boolean-valued flags, such as `--dev` itself, you'll need to use the associated negative flags to overide the default behaviour. Than is, following the former example if you want to override the dev mode used by default you should use

```
bigmler --train data/iris.csv --no-dev
```

The set of negative flags is:

`--no-debug`	as opposed to `--debug`
`--no-dev`	as opposed to `--dev`
`--no-train-header`	as opposed to `--train-header`
`--no-test-header`	as opposed to `--test-header`
`--local`	as opposed to `--remote`
`--no-replacement`	as opposed to `--replacement`
`--no-randomize`	as opposed to `--randomize`
`--no-no-tag`	as opposed to `--no-tag`
`--no-public-dataset`	as opposed to `--public-dataset`
`--no-black-box`	as opposed to `--black-box`
`--no-white-box`	as opposed to `--white-box`
`--no-progress-bar`	as opposed to `--progress-bar`
`--no-no-dataset`	as opposed to `--no-dataset`
`--no-no-model`	as opposed to `--no-model`
`--no-clear-logs`	as opposed to `--clear-logs`
`--no-store`	as opposed to `--store`
`--no-multi-label`	as opposed to `--multi-label`
`--no-prediction-header`	as opposed to `--prediction-header`
`--batch`	as opposed to `--no-batch`
`--no-balance`	as opposed to `--balance`
`--no-multi-dataset`	as opposed to `--multi-dataset`
`--unshared`	as opposed to `--shared`
`--upload`	as opposed to `--no-upload`
`--fast`	as opposed to `--no-fast`
`--no-no-csv`	as opposed to `--no-csv`
`--no-median`	as opposed to `--median`
`--no-score`	as opposed to `--score`
`--server`	as opposed to `--no-server`

CHAPTER 5

Support

Please report problems and bugs to our BigML.io issue tracker.

Discussions about the different bindings take place in the general BigML mailing list. Or join us in our Campfire chatroom.

CHAPTER 6

Requirements

Python 2.7 and 3 are currently supported by BigMLer.

BigMLer requires bigml 4.11.2 or higher. Using proportional missing strategy will additionally request the use of the numpy and scipy libraries. They are not automatically installed as a dependency, as they are quite heavy and exclusively required in this case. Therefore, they have been left for the user to install them if required.

Note that using proportional missing strategy for local predictions can also require numpy and scipy libraries. They are not installed by default. Check the bindings documentation for more info.

CHAPTER 7

BigMLer Installation

To install the latest stable release with pip

```
$ pip install bigmler
```

You can also install the development version of bigmler directly from the Git repository

```
$ pip install -e git://github.com/bigmlcom/bigmler.git#egg=bigmler
```

For a detailed description of install instructions on Windows see the *BigMLer on Windows* section.

CHAPTER 8

BigML Authentication

All the requests to BigML.io must be authenticated using your username and API key and are always transmitted over HTTPS.

BigML module will look for your username and API key in the environment variables `BIGML_USERNAME` and `BIGML_API_KEY` respectively. You can add the following lines to your `.bashrc` or `.bash_profile` to set those variables automatically when you log in

```
export BIGML_USERNAME=myusername
export BIGML_API_KEY=ae579e7e53fb9abd646a6ff8aa99d4afe83ac291
```

Otherwise, you can initialize directly when running the BigMLer script as follows

```
bigmler --train data/iris.csv --username myusername \
        --api-key ae579e7e53fb9abd646a6ff8aa99d4afe83ac291
```

For a detailed description of authentication instructions on Windows see the *BigMLer on Windows* section.

CHAPTER 9

BigMLer on Windows

To install BigMLer on Windows environments, you'll need Python for Windows (v.2.7.x) installed.

In addition to that, you'll need the `pip` tool to install BigMLer. To install pip, first you need to open your command line window (write `cmd` in the input field that appears when you click on `Start` and hit `enter`), download this python file and execute it

```
c:\Python27\python.exe ez_setup.py
```

After that, you'll be able to install `pip` by typing the following command

```
c:\Python27\Scripts\easy_install.exe pip
```

And finally, to install BigMLer, just type

```
c:\Python27\Scripts\pip.exe install bigmler
```

and BigMLer should be installed in your computer. Then issuing

```
bigmler --version
```

should show BigMLer version information.

Finally, to start using BigMLer to handle your BigML resources, you need to set your credentials in BigML for authentication. If you want them to be permanently stored in your system, use

```
setx BIGML_USERNAME myusername
setx BIGML_API_KEY ae579e7e53fb9abd646a6ff8aa99d4afe83ac291
```

Remember that `setx` will not change the environment variables of your actual console, so you will need to open a new one to start using them.

CHAPTER 10

BigML Development Mode

Also, you can instruct BigMLer to work in BigML's Sandbox environment by using the parameter `--dev`

```
bigmler --train data/iris.csv --dev
```

Using the development flag you can run tasks under 1 MB without spending any of your BigML credits.

CHAPTER 11

Using BigMLer

To run BigMLer you can use the console script directly. The `--help` option will describe all the available options

```
bigmler --help
```

Alternatively you can just call bigmler as follows

```
python bigmler.py --help
```

This will display the full list of optional arguments. You can read a brief explanation for each option below.

CHAPTER 12

Optional Arguments

General configuration

`--username`	BigML's username. If left unspecified, it will default to the values of the `BIGML_USERNAME` environment variable
`--api-key`	BigML's api_key. If left unspecified, it will default to the values of the `BIGML_API_KEY` environment variable
`--dev`	Uses FREE development environment. Sizes must be under 16MB though
`--debug`	Activates debug level and shows log info for each https request

Basic Functionality

`--train` *TRAINING_SET*	Full path to a training set. It can be a remote URL to a (gzipped or compressed) CSV file. The protocol schemes can be http, https, s3, azure, odata
`--test` *TEST_SET*	Full path to a test set. A file containing the data that you want to input to generate predictions
`--objective` *OBJECTIVE_FIELD*	The column number of the Objective Field (the field that you want to predict) or its name
`--output` *PREDICTIONS*	Full path to a file to save predictions. If unspecified, it will default to an auto-generated file created by BigMLer. It overrides `--output-dir`
`--output-dir` *DIRECTORY*	Directory where all the session files will be stored. It is overriden by `--output`
`--method` *METHOD*	Prediction method used: `plurality`, `"confidence weighted"`, `"probability weighted"`, `threshold` or `combined`
`--pruning` *PRUNING_TYPE*	The pruning applied in building the model. It's allowed values are `smart`, `statistical` and `no-pruning` The default value is `smart`
`--missing-strategy` *STRATEGY*	The strategy applied predicting when a missing value is found in a model split. It's allowed values are `last` or `proportional`. The default value is `last`
`--missing-splits`	Turns on the missing_splits flag in model creation. The model splits can include in one of its branches the data with missing values
`--evaluate`	Turns on evaluation mode
`--resume`	Retries command execution
`--stack-level` *LEVEL*	Level of the retried command in the stack
`--cross-validation-rate` *RATE*	Fraction of the training data held out for Monte-Carlo cross-validation
`--number-of-evaluations` *NUMBER_OF_EVALUATIONS*	Number of runs that will be used in cross-validation
`--max-parallel-evaluations` *MAX_PARALLEL_EVALUATIONS*	Maximum number of evaluations to create in parallel
`--project` *PROJECT_NAME*	Project name for the project to be associated to newly created sources
`--project-id` *PROJECT_ID*	Project id for the project to be associated to newly created sources
`--no-csv`	Causes the output of a batch prediction, batch centroid or batch anomaly score not to be downloaded as a CSV file
`--to-dataset`	Causes the output of a batch prediction, batch centroid or batch anomaly score to be stored remotely as a new dataset
`--median`	Predictions for single models are returned based on the median of the distribution in the predicted node

Content

`--name` *NAME*	Name for the resources in BigML.
`--category` *CATEGORY*	Category code. See full list.
`--description` *DESCRIPTION*	Path to a file with a description in plain text or markdown
`--tag` *TAG*	Tag to later retrieve new resources
`--no-tag`	Puts BigMLer default tag if no other tag is given

Data Configuration

`--no-train-header`	The train set file hasn't a header
`--no-test-header`	The test set file hasn't a header
`--field-attributes` *PATH*	Path to a file describing field attributes One definition per line (e.g., 0,'Last Nam
`--types` *PATH*	Path to a file describing field types. One definition per line (e.g., 0, 'numeric')
`--test-field-attributes` *PATH*	Path to a file describing test field attributes. One definition per line (e.g., 0,'Last
`--test-types` *PATH*	Path to a file describing test field types. One definition per line (e.g., 0, 'numeric
`--dataset-fields` *DATASET_FIELDS*	Comma-separated list of field column numbers to include in the dataset
`--model-fields` *MODEL_FIELDS*	Comma-separated list of input fields (predictors) to create the model
`--source-attributes` *PATH*	Path to a file containing a JSON expression with attributes to be used as argume
`--dataset-attributes` *PATH*	Path to a file containing a JSON expression with attributes to be used as argume
`--model-attributes` *PATH*	Path to a file containing a JSON expression with attributes to be used as argume
`--ensemble-attributes` *PATH*	Path to a file containing a JSON expression with attributes to be used as argume
`--evaluation-attributes` *PATH*	Path to a file containing a JSON expression with attributes to be used as argume
`--batch_prediction-attributes` *PATH*	Path to a file containing a JSON expression with attributes to be used as argume
`--json-filter` *PATH*	Path to a file containing a JSON expression to filter the source
`--lisp-filter` *PATH*	Path to a file containing a LISP expression to filter the source
`--locale` *LOCALE*	Locale code string
`--fields-map` *PATH*	Path to a file containing the dataset to model fields map for evaluation
`--test-separator` *SEPARATOR*	Character used as test data field separator
`--prediction-header`	Include a headers row in the prediction file
`--prediction-fields` *TEST_FIELDS*	Comma-separated list of fields of the test file to be included in the prediction file
`--max-categories` *CATEGORIES_NUMBER*	Sets the maximum number of categories that will be used in a dataset. When mo
`--new-fields` *PATH*	Path to a file containing a JSON expression used to generate a new dataset with
`--node-threshold`	Maximum number or nodes to grow the tree with
`--balance`	Automatically balance data to treat all classes evenly
`--weight-field` *FIELD*	Field name or column number that contains the weights to be used for each insta
`--shared`	Creates a secret link for every dataset, model or evaluation used in the command
`--reports`	Report formats: "gazibit"
`--no-upload`	Disables reports upload
`--dataset-off`	Sets the evaluation mode that uses the list of test datasets and extracts one each t
`--args-separator`	Character used as separator in multi-valued arguments (default is comma)
`--no-missing-splits`	Turns off the missing_splits flag in model creation.

Remote Resources

`--source` *SOURCE*	BigML source Id
`--dataset` *DATASET*	BigML dataset Id
`--datasets` *PATH*	Path to a file containing a dataset Id
`--model` *MODEL*	BigML model Id
`--models` *PATH*	Path to a file containing model/ids. One model per line (e.g., model/4f824203ce80053)
`--ensemble` *ENSEMBLE*	BigML ensemble Id
`--ensembles` *PATH*	Path to a file containing ensembles Ids
`--test-source` *SOURCE*	BigML test source Id (only for remote predictions)
`--test-dataset` *DATASET*	BigML test dataset Id (only for remote predictions)
`--test-datasets` *PATH*	Path to the file that contains datasets ids used in evaluations, one id per line.
`--source` *SOURCE*	BigML source Id
`--dataset` *DATASET*	BigML dataset Id
`--remote`	Computes predictions remotely (in batch mode by default)
`--no-batch`	Remote predictions are computed individually
`--no-fast`	Ensemble's local predictions are computed storing the predictions of each model in a separate local file before combining them (the default is –fast, that keeps in memory each model's prediction)
`--model-tag` *MODEL_TAG*	Retrieve models that were tagged with tag
`--ensemble-tag` *ENSEMBLE_TAG*	Retrieve ensembles that were tagged with tag

Ensembles

`--number-of-models` *NUMBER_OF_MODELS*	Number of models to create
`--sample-rate` *SAMPLE_RATE*	Sample rate to use (a float between 0.01 and 1)
`--replacement`	Use replacement when sampling
`--max-parallel-models` *MAX_PARALLEL_MODELS*	Max number of models to create in parallel
`--max-batch-models` *MAX_BATCH_MODELS*	Max number of local models to be predicted from in parallel. For ensembles with a number of models over it, predictions are stored in files as they are computed and retrived and combined eventually
`--randomize`	Use a random set of fields to split on
`--combine-votes` *LIST_OF_DIRS*	Combines the votes of models generated in a list of directories
`--ensemble-sample-rate` *RATE*	Ensemble sampling rate for bagging
`--ensemble-sample-seed` *SEED*	Value used as seed in ensembles random selections
`--ensemble-sample-no-replacement`	Don't use replacement when bagging
`--boosting`	Create a boosted ensemble
`--boosting-iterations` *ITERATIONS*	Maximum number of iterations used in boosted ensembles.
`--early-holdout` *HOLDOUT*	The portion of the dataset that will be held out for testing at the end of every iteration in boosted ensembles (between 0 and 1)
`--no-early-out-of-bag`	Causes the out of bag samples not to be tested after every iteration in boosted ensembles.
`--learning-rate` *RATE*	It controls how aggressively the boosting algorithm will fit the data in boosted ensembles (between 0 and 1)
`--no-step-out-of-bag`	Causes the out of bag samples not to be tested after every iteration to choose the gradient step size in boosted ensembles.

If you are not choosing to create an ensemble, make sure that you tag your models conveniently so that you can then retrieve them later to generate predictions.

Multi-labels

`--multi-label`	Use multiple labels in the objective field
`--labels`	Comma-separated list of labels used
`--training-separator` *SEPARATOR*	Character used as field separator in train data field
`--label-separator` *SEPARATOR*	Character used as label separator in the multi-labeled objective field

Public Resources

`--public-dataset`	Makes newly created dataset public
`--black-box`	Makes newly created model a public black-box
`--white-box`	Makes newly created model a public white-box
`--model-price`	Sets the price for a public model
`--dataset-price`	Sets the price for a public dataset
`--cpp`	Sets the credits consumed by prediction

Notice that datasets and models will be made public without assigning any price to them.

Local Resources

`--model-file` *PATH*	Path to a JSON file containing the model info
`--ensemble-file` *PATH*	Path to a JSON file containing the ensemble info

Fancy Options

`--progress-bar`	Shows an update on the bytes uploaded when creating a new source. This option might run into issues depending on the locale settings of your OS
`--no-dataset`	Does not create a model. BigMLer will only create a source
`--no-model`	Does not create a model. BigMLer will only create a dataset
`--resources-log` *LOG_FILE*	Keeps a log of the resources generated in each command
`--version`	Shows the version number
`--verbosity` *LEVEL*	Turns on (1) or off (0) the verbosity.
`--clear-logs`	Clears the `.bigmler`, `.bigmler_dir_stack`, `.bigmler_dirs` and user log file given in `--resources-log` (if any)
`--store`	Stores every created or retrieved resource in your output directory

Analyze subcommand Options

`--cross-validation`	Sets the k-fold cross-validation mode
`--k-folds`	Number of folds used in k-fold cross-validation (default is 5)
`--features`	Sets the smart selection features mode
`--staleness` *INTEGER*	Number of iterations with no improvement that is considered the limit for the analysis to stop (default is 5)
`--penalty` *FLOAT*	Coefficient used to penalyze models with many features in the smart selection features mode (default is 0.001). Also used in node threshold selection (default is 0)
`--optimize` *METRIC*	Metric that is being optimized in the smart selection features mode or the node threshold search mode (default is accuracy)
`--optimize-category` *CATEGORY*	Category whoese metric is being optimized in the smart selection features mode or the node threshold search mode (only for categorical models)
`--nodes`	Sets the node threshold search mode
`--min-nodes` *INTEGER*	Minimum number of nodes to start the node threshold search mode (default 3)
`--max-nodes` *INTEGER*	Maximum number of nodes to end the node threshold search mode (default 2000)
`--nodes-step` *INTEGER*	Step in the node threshold search iteration (default 50)
`--exclude-features` *FEATURES*	Comma-separated list of features in the dataset to be excluded from the features analysis
`--score`	Causes the training set to be run through the anomaly detector generating a batch anomaly score. Only used with the `--remote` flag.

Report Specific Subcommand Options

`--from-dir`	Path to a directory where BigMLer has stored its session data and created resources used in the report
`--port`	Port number for the HTTP server used to visualize graphics in `bigmler report`
`--no-server`	Not starting HTTP local server to show the reports

Cluster Specific Subcommand Options

`--cluster` *CLUSTER*	BigML cluster Id
`--clusters` *PATH*	Path to a file containing cluster/ids. One cluster per line (e.g., cluster/4f824203ce80051)
`--k` *NUMBER_OF_CENTROIDS*	Number of final centroids in the clustering
`--no-cluster`	No cluster will be generated
`--cluster-fields`	Comma-separated list of fields that will be used in the cluster construction
`--cluster-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the cluster creation call
`--cluster-datasets` *CENTROID_NAMES*	Comma-separated list of centroid names to generate the related datasets from a cluster. If no CENTROID_NAMES argument is provided all datasets are generated
`--cluster-file` *PATH*	Path to a JSON file containing the cluster info
`--cluster-seed` *SEED*	Seed to generate deterministic clusters
`--centroid-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the centroid creation call
`--batch-centroid-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the batch centroid creation call
`--cluster-models` *CENTROID_NAMES*	Comma-separated list of centroid names to generate the related models from a cluster. If no CENTROID_NAMES argument is provided all models are generated
`--summary-fields` *SUMMARY_FIELDS*	Comma-separated list of fields to be kept for reference but not used in the cluster bulding process

Anomaly Specific Subcommand Options

`--anomaly` *ANOMALY*	BigML anomaly Id
`--anomalies` *PATH*	Path to a file containing anomaly/ids. One anomaly per line (e.g., anomaly/4f824203ce80051)
`--no-anomaly`	No anomaly detector will be generated
`--anomaly-fields`	Comma-separated list of fields that will be used in the anomaly detector construction
`--top-n`	Number of listed top anomalies
`--forest-size`	Number of models in the anomaly detector iforest
`--anomaly-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the anomaly creation call
`--anomaly-file` *PATH*	Path to a JSON file containing the anomaly info
`--anomaly-seed` *SEED*	Seed to generate deterministic anomalies
`--anomaly-score-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the anomaly score creation call
`--batch-anomaly-score-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the batch anomaly score creation call
`--anomalies-datasets` *[in |out]*	Separates from the training dataset the top anomalous instances enclosed in the top anomalies list and generates a new dataset including them (`in` option) or excluding them (`out` option).

.._sample_options:

Samples Subcommand Options

`--sample` *SAMPLE*	BigML sample Id
`--samples` *PATH*	Path to a file containing sample/ids. One sample per line (e.g., sample/4f824203ce80051)
`--no-sample`	No sample will be generated
`--sample-fields` *FIELD_NAMES*	Comma-separated list of fields that will be used in the sample detector construction
`--sample-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the sample creation call
`--fields-filter` *QUERY*	Query string that will be used as filter before selecting the sample rows. The query string can be built using the field ids, their values and the usual operators. You can see some examples in the developers section
`--sample-header`	Adds a headers row to the sample.csv output
`--row-index`	Prepends acolumn to the sample rows with the absolute row number
`--occurrence`	Prepends a column to the sample rows with the number of occurences of each row. When used with –row-index, the occurrence column will be placed after the index column
`--precision`	Decimal numbers precision
`--rows SIZE`	Number of rows returned
`--row-offset` *OFFSET*	Skip the given number of rows
`--row-order-by` *FIELD_NAME*	Field name whose values will be used to sort the returned rows
`--row-fields` *FIELD_NAMES*	Comma-separated list of fields that will be returned in the sample
`--stat-fields` *FIELD_NAME,FIELD_NAME*	Two comma-separated numeric field names that will be used to compute their Pearson's and Spearman's correlations and linear regression terms
`--stat-field` *FIELD_NAME*	Numeric field that will be used to compute Pearson's and Spearman's correlations and linear regression terms against the rest of numeric fields in the sample
`--unique`	Repeated rows are removed from the sample

Logistic regression Subcommand Options

`--logistic-regression` *LOGISTIC_R*	BigML logistic regression Id
`--logistic-regressions` *PATH*	Path to a file containing logisticregression/ids. One logistic regression per line (e.g., logisticregression/4f824203ce80051)
`--no-logistic-regression`	No logistic regression will be generated
`--logistic-fields` *LOGISTIC_FIELDS*	Comma-separated list of fields that will be used in the logistic regression construction
`--normalize`	Normalize feature vectors in training and prediction inputs
`--no-missing-numerics`	Avoids the default behaviour, which creates a new coefficient for missings in numeric fields. Missing rows are discarded.
`--no-bias`	Avoids default behaviour. The logistic regression will have no intercept term.
`--no-balance-fields`	Avoids default behaviour. No automatic field balance.
`--field-codings` *FIELD_CODINGS*	Numeric encoding for categorical fields (default one-hot encoding)
`--c` *C*	Strength of the regularization step
`--eps` *EPS*	Stopping criteria for solver.
`--logistic-regression-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the logistic regression creation call
`--logistic-regression-file` *PATH*	Path to a JSON file containing the logistic regression info

Topic Model Subcommand Options

`--topic-model` *TOPIC_MODEL*	BigML topic model Id
`--topic-models` *PATH*	Path to a file containing topicmodel/ids. One topic model per line (e.g., topicmodel/4f824203ce80051)
`--no-topic-model`	No topic model will be generated
`--topic-fields` *TOPIC_FIELDS*	Comma-separated list of fields that will be used in the topic model construction
`--bigrams`	Use bigrams in topic search
`--case-sensitive`	Use case sensitive tokenization
`--excluded-terms` *EXCLUDED_TERMS*	Comma-separated list of terms to be excluded from the analysis
`--use-stopwords`	Use stopwords in the analysis.
`--topic-model-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the topic model creation call
`--topic-model-file` *PATH*	Path to a JSON file containing the topic model info

Time Series Subcommand Options

`--time-seriers` *TIME_SERIES*	BigML time series Id
`--time-series-set` *PATH*	Path to a file containing timeseries/ids One time series per line (e.g., timeseries/4f824203ce80051)
`--no-time-series`	No time series will be generated.
`--objectives` *OBJECTIVES*	Comma-separated list of fields that will be used in the time series as objective fields
`--time-series-attributes` *PATH*	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) to be used in the time series creation call
`--time-series-file` *PATH*	Path to a JSON file containing the time series info
`--all-numeric-objectives`	When used, all the numeric fields in the dataset are considered objective fields
`--default-numeric-value` *DEFAULT*	The value used by default if a numeric field is missing. Spline interpolation is used by default and other options are “mean”, “median”, “minimum”, “maximum” and “zero”
`--error` *TYPE*	Type of error considered: 1 - Additive, 2 - Multiplicative
`--period` *PERIOD*	Expected period
`--seasonality` *SEASONALITY*	Type of seasonality: 0 - None, 1 - Additive, 2 - Multiplicative
`--trend` *TREND*	Type of trend: 0 - None, 1 - Additive, 2 - Multiplicative
`--range` *RANGE*	Comma-separated pair of values that set the range limits
`--damped-trend`	When set damping is used in trend
`--forecast`	When set, the time series default forecast is produced
`--horizon` *HORIZON*	Set to an integer, is the number of points in the forecast
`--time-start` *START*	Time starting point coordinate
`--time-end` *END*	Time ending point coordinate
`--time-unit` *UNIT*	Unit for the time interval. The options are described in the API documentation
`--time-interval` *INTERVAL*	Time interval between two rows

Reify Subcommand Options

`--id` *RESOURCE_ID*	ID for the resource to be reified
`--language` *SCRIPTING_LANG*	Language to be used for the script. Currently only Python is available
`--output` *PATH*	Path to the file where the script will be stored
`--add-fields`	Causes the fields information to be added to the source arguments

Execute Subcommand Options

`--code` *SOURCE_CODE*	WhizzML source code to be executed
“–code-file ‘‘ *PATH*	Path to the file that contains Whizzml source code
`--creation-defaults` *RESOURCE_DEFAULTS*	Path to the JSON file with the default configurations for created resources. Please, see details in the API Developers documentation
`--declare-inputs` *INPUTS_DECLARATION*	Path to the JSON file with the description of the input parameters. Please, see details in the API Developers documentation
`--declare-outputs` *OUTPUTS_DECLARATION*	Path to the JSON file with the description of the script outputs. Please, see details in the API Developers documentation
`--embedded-libraries` *PATH*	Path to a file that contains the location of the files to be embedded in the script as libraries
`--execution` *EXECUTION_ID*	BigML execution ID
`--execution-file` *EXECUTION_FILE*	BigML execution JSON structure file
`--execution-tag` *EXECUTION_TAG*	Select executions tagged with EXECUTION_TAG
`--executions` *EXECUTIONS*	Path to a file containing execution/ids. Just one execution per line (e.g., execution/50a20697035d0706da0004a4)
`--imports` *LIBRARIES*	Comma-separated list of libraries IDs to be included as imports in scripts or other libraries
`--input-maps` *INPUT_MAPS*	Path to the JSON file with the description of the execution inputs for a list of scripts
`--inputs` *INPUTS*	Path to the JSON file with the description of the execution inputs. Please, see details in the API Developers documentation
`--libraries` *LIBRARIES*	Path to a file containing libraries/ids. Just one library per line (e.g., library/50a20697035d0706da0004a4)
`--library` *LIBRARY*	BigML library Id.
`--library-file` *LIBRARY_FILE*	BigML library JSON structure file.
`--library-tag` *LIBRARY_TAG*	Select libraries tagged with tag to be deleted
`--outputs` *OUTPUTS*	Path to the JSON file with the names of the output parameters. Please, see details in the API Developers documentation
`--script` *SCRIPT*	BigML script Id.
`--script-file` *SCRIPT_FILE*	BigML script JSON structure file.
`--script-tag` *SCRIPT_TAG*	Select script tagged with tag to be deleted
`--scripts` *SCRIPTS*	Path to a file containing script/ids. Just one script per line (e.g., script/50a20697035d0706da0004a4).
`--to-library`	Boolean that causes the code to be compiled and stored as a library

Whizzml Subcommand Options

`--package-dir` *DIR*	Directory that stores the package files
“–embed-libs ‘‘	I causes the subcommand to embed the libraries code in the package scripts instead of creating libraries and importing them

Delete Subcommand Options

Project Specific Subcommand Options

`--project-attributes`	Path to a JSON file containing attributes for the project

Association Specific Subcommand Options

`--association-attributes`	Path to a JSON file containing attributes (any of the updatable attributes described in the developers section) for the association
`--max-k` K	Maximum number of rules to be found
`--search-strategy` STRATEGY	Strategy used when searching for the associations. The possible values are: confidence, coverage, leverage, lift, support

Prior Versions Compatibility Issues

BigMLer will accept flags written with underscore as word separator like `--clear_logs` for compatibility with prior versions. Also `--field-names` is accepted, although the more complete `--field-attributes` flag is preferred. `--stat_pruning` and `--no_stat_pruning` are discontinued and their effects can be achived by setting the actual `--pruning` flag to `statistical` or `no-pruning` values respectively.

Running the Tests

To run the tests you will need to install lettuce

```
$ pip install lettuce
```

and set up your authentication via environment variables, as explained above. With that in place, you can run the test suite simply by

```
$ cd tests
$ lettuce
```

CHAPTER 13

Building the Documentation

Install the tools required to build the documentation

```
$ pip install sphinx
```

To build the HTML version of the documentation

```
$ cd docs/
$ make html
```

Then launch `docs/_build/html/index.html` in your browser.

CHAPTER 14

Additional Information

For additional information, see the full documentation for the Python bindings on Read the Docs. For more information about BigML's API, see the BigML developer's documentation.

CHAPTER 15

How to Contribute

Please follow the next steps:

1. Fork the project on github.
2. Create a new branch.
3. Commit changes to the new branch.
4. Send a pull request.

For details on the underlying API, see the BigML API documentation.