bigml-java Documentation

Release master

Jun 25, 2020

Contents

In this tutorial, you will learn how to use the BigML bindings for Java.

CHAPTER 1

Additional Information

For additional information about the API, see the BigML developer's documentation.

1.1 Introduction

BigML makes machine learning easy by taking care of the details required to add data-driven decisions and predictive power to your company. Unlike other machine learning services, BigML creates beautiful predictive models that can be easily understood and interacted with.

These BigML Java bindings allow you to interact with BigML.io, the API for BigML. You can use it to easily create, retrieve, list, update, and delete BigML resources (i.e., sources, datasets, models and, predictions).

This module is licensed under the Apache License, Version 2.0.

1.1.1 Support

Please report problems and bugs to our BigML Java Binding issue tracker.

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

1.1.2 Requirements

JVM 1.6 and above are currently supported by these bindings.

You will also need `maven` to build the package. If you are new to `maven`, please refer to Maven Getting Started Guide.

1.1.3 Installation

To use the latest stable release, include the following `maven` dependency in your project's `pom.xml`.

```
<dependency>
    <groupId>org.bigml</groupId>
    <artifactId>bigml-binding</artifactId>
    <version>1.8.13</version>
</dependency>
```

You can also download the development version of the bindings directly from the Git repository

```
$ git clone git://github.com/bigmlcom/bigml-java.git
```

1.1.4 Authentication

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

This module will look for your username and API key in the `src/main/resources/binding.properties` file. Alternatively, you can respectively set the JVM parameters `BIGML_USERNAME` and `BIGML_API_KEY` with `-D` or use envronment variables.

With that set up, connecting to BigML is a breeze. First, import `BigMLClient`:

```
import org.bigml.binding.BigMLClient;
```

then:

```
BigMLClient api = new BigMLClient();
```

Otherwise, you can initialize directly when instantiating the BigMLClient class as follows:

```
BigMLClient api = new BigMLClient(
    "myusername", "ae579e7e53fb9abd646a6ff8aa99d4afe83ac291", null);
```

These credentials will allow you to manage any resource in your user environment.

In BigML a user can also work for an `organization`. In this case, the organization administrator should previously assign permissions for the user to access one or several particular projects in the organization. Once permissions are granted, the user can work with resources in a project according to his permission level by creating a special constructor for each project. The connection constructor in this case should include the `project ID`:

```
BigMLClient api = new BigMLClient(
    "myusername", "ae579e7e53fb9abd646a6ff8aa99d4afe83ac291",
    "project/53739b98d994972da7001d4a", null, null);
```

If the project used in a connection object does not belong to an existing organization but is one of the projects under the user's account, all the resources created or updated with that connection will also be assigned to the specified project.

When the resource to be managed is a `project` itself, the connection needs to include the corresponding `organization ID`:

```
BigMLClient api = new BigMLClient(
    "myusername", "ae579e7e53fb9abd646a6ff8aa99d4afe83ac291",
    "project/53739b98d994972da7001d4a",
    "organization/53739b98d994972da7025d4a", null);
```

1.1.5 Alternative domains

For Virtual Private Cloud setups, you can change the remote server URL to the VPC particular one by either setting the `BIGML_URL` in `binding.properties` or in the JVM environment. By default, they have the following values:

```
BIGML_URL=https://bigml.io/andromeda/
```

If you are in Australia or New Zealand, you can change them to:

```
BIGML_URL=https://au.bigml.io/andromeda/
```

The corresponding SSL REST calls will be directed to your private domain henceforth.

1.2 Quick Start

This chapter shows how to create a model from a remote CSV file and use it to make a prediction for a new single instance.

Imagine that you want to use this csv file containing the Iris flower dataset to predict the species of a flower whose `sepal length` is `5` and whose `sepal width` is `2.5`. A preview of the dataset is shown below. It has 4 numeric fields: `sepal length`, `sepal width`, `petal length`, `petal width` and a categorical field: `species`. By default, BigML considers the last field in the dataset as the objective field (i.e., the field that you want to generate predictions for).

```
sepal length,sepal width,petal length,petal width,species
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
...
5.8,2.7,3.9,1.2,Iris-versicolor
6.0,2.7,5.1,1.6,Iris-versicolor
5.4,3.0,4.5,1.5,Iris-versicolor
...
6.8,3.0,5.5,2.1,Iris-virginica
5.7,2.5,5.0,2.0,Iris-virginica
5.8,2.8,5.1,2.4,Iris-virginica
```

The typical process you need to follow when using BigML is to:

1. open a connection to BigML API with your user name and API Key
2. create a **source** by uploading the data file
3. create a **dataset** (a structured version of the source)
4. create a **model** using the dataset
5. finally, use the model to make a **prediction** for some new input data.

As you can see, all the steps above share some similarities, in that each one consists of creating a new BigML resource from some other BigML resource. This makes the BigML API very easy to understand and use, since all available operations are orthogonal to the kind of resource you want to create.

All API calls in BigML are asynchronous, so you will not be blocking your program while waiting for the network to send back a reply. This means that at each step you need to wait for the resource creation to finish before you can move on to the next step.

This can be exemplified with the first step in our process, creating a **source** by uploading the data file.

First of all, you need to create the connecting to BigML:

```
import org.bigml.binding.BigMLClient;

// Create BigMLClient with the properties in binding.properties
BigMLClient api = new BigMLClient();
```

You will need to create then a `Source` object to encapsulate all information that will be used to create it correctly, i.e., an optional name for the source and the data file to use:

```
JSONObject args = null;
JSONObject source = api.createRemoteSource(
    "https://static.bigml.com/csv/iris.csv",
    "Iris Source", args);
```

If you do not want to use a remote data file, as you are doing in this example, you can use a local data file by replacing the last line above, as shown here:

```
JSONObject args = null;
JSONObject source = api.createSource(
    "./data/iris.csv", "Iris Source", args);
```

That's all! BigML will create the source, as per our request, and automatically list it in the BigML Dashboard. As mentioned, though, you will need to monitor the source status until it is fully created before you can move on to the next step, which can be easily done like this:

```
while (!api.sourceIsReady(source))
    Thread.sleep(1000);
```

The steps described above define a generic pattern of how to create the resources you need next, i.e., a `Dataset`, a `Model`, and a `Prediction`. As an additional example, this is how you create a `Dataset` from the `Source` you have just created:

```
  // --- create a dataset from the previous source ---
  // Dataset object which will encapsulate the dataset information
  JSONObject args = null;
  args.put("name", "my new dataset");

  JSONObject dataset = api.createDataset(
        (String)source.get("resource"), args, null, null);

  while (!api.datasetIsReady(dataset))
        Thread.sleep(1000);
```

You can easily complete the crreation of a prediction following these steps:

```
    JSONObject model = api.createModel(
        (String)dataset.get("resource"), args, null, null);

    while (!api.modelIsReady(model))
        Thread.sleep(1000);

    JSONObject inputData = new JSONObject();
    inputData.put("sepal length", 5);
    inputData.put("sepal width", 2.5);

    JSONObject prediction = api.createPrediction(
```

(continues on next page)

(continued from previous page)

```
        (String)model.get("resource"), inputData, true,
        args, null, null);
```

After this quick introduction, it should be now easy to follow and understand the full code that is required to create a prediction starting from a data file. Make sure you have properly installed BigML Java bindings as detailed in Requirements.

You can then get the prediction result:

```
prediction = api.getPrediction(prediction);
```

and print the result:

```
String output = (String)Utils.getJSONObject(
    prediction, "object.output");
System.out.println("Prediction result: " + output);

Prediction result: Iris-virginica
```

and also generate an evaluation for the model by using:

```
    JSONObject testSource = api.createSource("./data/test_iris.csv",
        "Test Iris Source", args);

    while (!api.sourceIsReady(source)) Thread.sleep(1000);

    JSONObject testDataset = api.createDataset(
        (String)testSource.get("resource"), args, null, null);

    while (!api.datasetIsReady(dataset)) Thread.sleep(1000);

    JSONObject evaluation = api.createEvaluation(
        (String)model.get("resource"), (String)dataset.get("resource"),
        args, null, null);
```

Setting the `storage` argument in the api client instantiation:

```
BigMLClient api = new BigMLClient(
    "myusername", "ae579e7e53fb9abd646a6ff8aa99d4afe83ac291", "./storage");
```

all the generated, updated or retrieved resources will be automatically saved to the chosen directory.

You can also find a sample API client code from here.

1.3 Fields Structure

1.3.1 Source

BigML automatically generates identifiers for each field. The following example shows how to retrieve the fields, ids, and its types that have been assigned to a source:

```
source = api.getSource(source);
JSONObject fields = (JSONObject) Utils.getJSONObject(
    source, "object.fields");
```

source `fields` object:

```
{
    "000000":{
        "name":"sepal length",
        "column_number":0,
        "optype":"numeric",
        "order":0
    },
    "000001":{
        "name":"sepal width",
        "column_number":1,
        "optype":"numeric",
        "order":1
    },
    "000002":{
        "name":"petal length",
        "column_number":2,
        "optype":"numeric",
        "order":2
    },
    "000003":{
        "name":"petal width",
        "column_number":3,
        "optype":"numeric",
        "order":3
    },
    "000004":{
        "column_number":4,
        "name":"species",
        "optype":"categorical",
        "order":4,
        "term_analysis":{
            "enabled":true
        }
    }
}
```

When the number of fields becomes very large, it can be useful to exclude or filter them. This can be done using a query string expression, for instance:

```
source = api.getSource(source, "limit=10&order_by=name");
```

would include in the retrieved dictionary the first 10 fields sorted by name.

1.3.2 Dataset

If you want to get some basic statistics for each field you can retrieve the `fields` from the dataset as follows to get a dictionary keyed by field id:

```
dataset = api.getDataset(dataset);
JSONOoject fields = (JSONObject) Utils.getJSONObject(
    dataset, "object.fields");
```

dataset `fields` object:

```
{
    "000000": {
        "column_number": 0,
        "datatype": "double",
        "name": "sepal length",
        "optype": "numeric",
        "order": 0,
        "preferred": true,
        "summary": {
            "bins": [
                [4.3, 1],
                [4.425, 4],

                ...snip...

                [7.9, 1]
            ],
            "kurtosis": -0.57357,
            "maximum": 7.9,
            "mean": 5.84333,
            "median": 5.8,
            "minimum": 4.3,
            "missing_count": 0,
            "population": 150,
            "skewness": 0.31175,
            "splits": [
                4.51526,
                4.67252,

                ...snip...

                7.64746
            ],
            "standard_deviation": 0.82807,
            "sum": 876.5,
            "sum_squares": 5223.85,
            "variance": 0.68569
        }
    },

    ...snip...

    "000004": {

        ...snip...

    }
}
```

The field filtering options are also available using a query string expression, for instance:

```
dataset = api.getDataset(dataset, "limit=20");
```

limits the number of fields that will be included in `dataset` to 20.

1.3.3 Model

One of the greatest things about BigML is that the models that it generates for you are fully white-boxed. To get the explicit tree-like predictive model for the example above:

```
model = api.getModel(model);
JSONObject tree = (JSONObject) Utils.getJSONObject(
    model, "object.model.root");
```

model `tree` object:

```
{
    "children":[{
        "children":[{
            "children":[{
                "confidence":0.91799,
                "count":43,
                "id":3,
                "objective_summary":{
                    "categories":[
                        [
                            "Iris-virginica",
                            43
                        ]
                    ]
                },
                "output":"Iris-virginica",
                "predicate":{
                    "field":"000002",
                    "operator":">",
                    "value":4.85
                }
            }, {
                "children":[{
                    "confidence":0.20654,
                    "count":1,
                    "id":5,
                    "objective_summary":{
                        "categories":[
                            [
                                "Iris-versicolor",
                                1
                            ]
                        ]
                    },
                    "output":"Iris-versicolor",
                    "predicate":{
                        "field":"000001",
                        "operator":">",
                        "value":3.1
                    }
                },

                ...snip...

            },

            ...snip...
```

(continues on next page)

(continued from previous page)

```
        },

        ...snip...

    },

    ...snip...
}
```

(Note that we have abbreviated the output in the snippet above for readability: the full predictive model you'll get is going to contain much more details).

Again, filtering options are also available using a query string expression, for instance:

```
model = api.getModel(model, "limit=5");
```

limits the number of fields that will be included in `model` to 5.

1.3.4 Evaluation

The predictive performance of a model can be measured using many different measures. In BigML these measures can be obtained by creating evaluations. To create an evaluation you need the id of the model you are evaluating and the id of the dataset that contains the data to be tested with. The result is shown as:

```
evaluation = api.getEvaluation(evaluation);
JSONObject result = (JSONObject) Utils.getJSONObject(evaluation, "object.result");
```

evaluation `result` object:

```
{
    "class_names":[
        "Iris-setosa",
        "Iris-versicolor",
        "Iris-virginica"
    ],
    "mode":{
        "accuracy":0.33333,
        "average_f_measure":0.16667,
        "average_phi":0,
        "average_precision":0.11111,
        "average_recall":0.33333,
        "confusion_matrix":[
            [50, 0, 0],
            [50, 0, 0],
            [50, 0, 0]
        ],
        "per_class_statistics":[
            {
                "accuracy":0.3333333333333333,
                "class_name":"Iris-setosa",
                "f_measure":0.5,
                "phi_coefficient":0,
                "precision":0.3333333333333333,
                "present_in_test_data":true,
```

(continues on next page)

(continued from previous page)

```
                "recall":1.0
            },
            {
                "accuracy":0.6666666666666667,
                "class_name":"Iris-versicolor",
                "f_measure":0,
                "phi_coefficient":0,
                "precision":0,
                "present_in_test_data":true,
                "recall":0.0
            },
            {
                "accuracy":0.6666666666666667,
                "class_name":"Iris-virginica",
                "f_measure":0,
                "phi_coefficient":0,
                "precision":0,
                "present_in_test_data":true,
                "recall":0.0
            }
        ]
    },
    "model":{
        "accuracy":1,
        "average_f_measure":1,
        "average_phi":1,
        "average_precision":1,
        "average_recall":1,
        "confusion_matrix":[
            [50, 0, 0],
            [0, 50, 0],
            [0, 0, 50]
        ],
        "per_class_statistics":[
            {
                "accuracy":1.0,
                "class_name":"Iris-setosa",
                "f_measure":1.0,
                "phi_coefficient":1.0,
                "precision":1.0,
                "present_in_test_data":true,
                "recall":1.0
            },
            {
                "accuracy":1.0,
                "class_name":"Iris-versicolor",
                "f_measure":1.0,
                "phi_coefficient":1.0,
                "precision":1.0,
                "present_in_test_data":true,
                "recall":1.0
            },
            {
                "accuracy":1.0,
                "class_name":"Iris-virginica",
                "f_measure":1.0,
                "phi_coefficient":1.0,
```

(continues on next page)

(continued from previous page)

```
                "precision":1.0,
                "present_in_test_data":true,
                "recall":1.0
            }
        ]
    },
    "random":{
        "accuracy":0.28,
        "average_f_measure":0.27789,
        "average_phi":-0.08123,
        "average_precision":0.27683,
        "average_recall":0.28,
        "confusion_matrix":[
            [14, 19, 17],
            [19, 10, 21],
            [15, 17, 18]
        ],
        "per_class_statistics":[
            {
                "accuracy":0.5333333333333333,
                "class_name":"Iris-setosa",
                "f_measure":0.2857142857142857,
                "phi_coefficient":-0.06063390625908324,
                "precision":0.2916666666666667,
                "present_in_test_data":true,
                "recall":0.28
            },
            {
                "accuracy":0.4933333333333333,
                "class_name":"Iris-versicolor",
                "f_measure":0.20833333333333331,
                "phi_coefficient":-0.16357216402190614,
                "precision":0.21739130434782608,
                "present_in_test_data":true,
                "recall":0.2
            },
            {
                "accuracy":0.5333333333333333,
                "class_name":"Iris-virginica",
                "f_measure":0.33962264150943394,
                "phi_coefficient":-0.019492029389636262,
                "precision":0.32142857142857145,
                "present_in_test_data":true,
                "recall":0.36
            }
        ]
    }
}
```

where two levels of detail are easily identified. For classifications, the first level shows these keys:

- **class_names**: A list with the names of all the categories for the objective field (i.e., all the classes)
- **mode**: A detailed result object. Measures of the performance of the classifier that predicts the mode class for all the instances in the dataset
- **model**: A detailed result object.
- **random**: A detailed result object. Measures the performance of the classifier that predicts a random class for

all the instances in the dataset.

and the detailed result objects include accuracy, average_f_measure, average_phi, average_precision, average_recall, confusion_matrix and per_class_statistics.

For regressions first level will contain these keys:

- **mean**: A detailed result object. Measures the performance of the model that predicts the mean for all the instances in the dataset.
- **model**: A detailed result object.
- **random**: A detailed result object. Measures the performance of the model that predicts a random class for all the instances in the dataset.

where the detailed result objects include mean_absolute_error, mean_squared_error and r_squared (refer to developers documentation for more info on the meaning of these measures.

1.3.5 Cluster

For unsupervised learning problems, the cluster is used to classify in a limited number of groups your training data. The cluster structure is defined by the centers of each group of data, named centroids, and the data enclosed in the group. As for in the model's case, the cluster is a white-box resource and can be retrieved as a JSON:

```
cluster = api.getCluster("cluster/56c42ea47e0a8d6cca0151a0");
JSONObject result = (JSONObject) Utils.getJSONObject(cluster, "object");
```

cluster object object:

```
{
    "balance_fields":true,
    "category":0,
    "cluster_datasets":{},
    "cluster_models":{},
    "clusters":{
        "clusters":[{
            "center":{
                "000000":6.262,
                "000001":2.872,
                "000002":4.906,
                "000003":1.676,
                "000004":"Iris-virginica"
            },
            "count":100,
            "distance":{
                "bins":[
                    [0.03935, 1],
                    [0.04828, 1],
                    [0.06093, 1 ],
                    ...snip...
                    [0.47935, 1]
                ],
                "maximum":0.47935,
                "mean":0.21705,
                "median":0.20954,
                "minimum":0.03935,
                "population":100,
                "standard_deviation":0.0886,
```

(continues on next page)

(continued from previous page)

```
                "sum":21.70515,
                "sum_squares":5.48833,
                "variance":0.00785
            },
            "id":"000000",
            "name":"Cluster 0"
        }, {
            "center":{
                "000000":5.006,
                "000001":3.428,
                "000002":1.462,
                "000003":0.246,
                "000004":"Iris-setosa"
            },
            "count":50,
            "distance":{
                "bins":[
                    [0.01427, 1],
                    [0.02279, 1],
                    ...snip...
                    [0.41736, 1]
                ],
                "maximum":0.41736,
                "mean":0.12717,
                "median":0.113,
                "minimum":0.01427,
                "population":50,
                "standard_deviation":0.08521,
                "sum":6.3584,
                "sum_squares":1.16432,
                "variance":0.00726
            },
            "id":"000001",
            "name":"Cluster 1"
        }],
        "fields":{
            ...snip...
        }
    },
    "code":200,
    "columns":5,
    "created":"2016-02-17T08:26:12.583000",
    "credits":0.017581939697265625,
    "credits_per_prediction":0.0,
    "critical_value":5,
    "dataset":"dataset/56c42ea07e0a8d6cca01519b",
    "dataset_field_types":{
        "categorical":1,
        "datetime":0,
        "effective_fields":5,
        "items":0,
        "numeric":4,
        "preferred":5,
        "text":0,
        "total":5
    },
    "dataset_status":true,
```

(continues on next page)

(continued from previous page)

```
    "dataset_type":0,
    "description":"",
    "excluded_fields":[],
    "field_scales":{},
    "fields_meta":{
        "count":5,
        "limit":1000,
        "offset":0,
        "query_total":5,
        "total":5
    },
    "input_fields":[
        "000000",
        "000001",
        "000002",
        "000003",
        "000004"
    ],
    "k":2,
    "locale":"en_US",
    "max_columns":5,
    "max_rows":150,
    "model_clusters":false,
    "name":"Iris Source dataset's cluster",
    "number_of_batchcentroids":0,
    "number_of_centroids":0,
    "number_of_public_centroids":0,
    "out_of_bag":false,
    "price":0.0,
    "private":true,
    "project":null,
    "range":[
        1,
        150
    ],
    "replacement":false,
    "resource":"cluster/56c42ea47e0a8d6cca0151a0",
    "rows":150,
    "sample_rate":1.0,
    "scales":{
        "000000":0.18941532079904913,
        "000001":0.35975000221609077,
        "000002":0.08884141152890178,
        "000003":0.20571391803576422,
        "000004":0.15627934742019414
    },
    "shared":false,
    "size":4609,
    "source":"source/56c42e9f8a318f66df007548",
    "source_status":true,
    "status":{
        "code":5,
        "elapsed":1213,
        "message":"The cluster has been created",
        "progress":1.0
    },
    "subscription":false,
```

(continues on next page)

(continued from previous page)

```
    "summary_fields":[],
    "tags":[],
    "updated":"2016-02-17T08:26:24.259000",
    "white_box":false
}
```

(Note that we have abbreviated the output in the snippet above for readability: the full predictive cluster yo'll get is going to contain much more details).

1.3.6 Anomaly Detector

For anomaly detection problems, BigML uses iforest as an unsupervised kind of model that detects anomalous data in a dataset. The information it returns encloses a `top_anomalies` block that contains a list of the most anomalous points. For each, we capture a `score` from 0 to 1. The closer to 1, the more anomalous. We also capture the `row` which gives values for each field in the order defined by `input_fields`. Similarly we give a list of `importances` which match the `row` values. These importances tell us which values contributed most to the anomaly score. Thus, the structure of an anomaly detector is similar to:

```
anomaly = api.getAnomaly("anomaly/56c432728a318f66e4012f82");
JSONObject object = (JSONObject) Utils.getJSONObject(anomaly, "object");
```

anomaly `object` object:

```
{
    "anomaly_seed":"2c249dda00fbf54ab4cdd850532a584f286af5b6",
    "category":0,
    "code":200,
    "columns":5,
    "constraints":false,
    "created":"2016-02-17T08:42:26.663000",
    "credits":0.12307357788085938,
    "credits_per_prediction":0.0,
    "dataset":"dataset/56c432657e0a8d6cd0004a2d",
    "dataset_field_types":{
        "categorical":1,
        "datetime":0,
        "effective_fields":5,
        "items":0,
        "numeric":4,
        "preferred":5,
        "text":0,
        "total":5
    },
    "dataset_status":true,
    "dataset_type":0,
    "description":"",
    "excluded_fields":[],
    "fields_meta":{
        "count":5,
        "limit":1000,
        "offset":0,
        "query_total":5,
        "total":5
    },
    "forest_size":128,
```

(continues on next page)

(continued from previous page)

```
    "id_fields":[],
    "input_fields":[
        "000000",
        "000001",
        "000002",
        "000003",
        "000004"
    ],
    "locale":"en_US",
    "max_columns":5,
    "max_rows":150,
    "model":{
        "constraints":false,
        "fields":{
            ...snip...
        },
        "forest_size":128,
        "kind":"iforest",
        "mean_depth":9.557347074468085,
        "sample_size":94,
        "top_anomalies":[{
            "importance":[
                0.22808,
                0.23051,
                0.21026,
                0.1756,
                0.15555
            ],
            "row":[
                7.9,
                3.8,
                6.4,
                2.0,
                "Iris-virginica"
            ],
            "row_number":131,
            "score":0.58766
        },
        {
            "importance":[
                0.21552,
                0.22631,
                0.22319,
                0.1826,
                0.15239
            ],
            "row":[
                7.7,
                3.8,
                6.7,
                2.2,
                "Iris-virginica"
            ],
            "row_number":117,
            "score":0.58458
        },
        ...snip...
```

(continues on next page)

(continued from previous page)

```
        {
            "importance":[
                0.23113,
                0.15013,
                0.17312,
                0.20304,
                0.24257
            ],
            "row":[
                4.9,
                2.5,
                4.5,
                1.7,
                "Iris-virginica"
            ],
            "row_number":106,
            "score":0.54096
        }],
        "top_n":10,
        "trees":[{
            "root":{
                "children":[{
                    "children":[{
                        "children":[{
                            "children":[{
                                "children":[{
                                    "population":1,
                                    "predicates":[{
                                        "field":"00001f",
                                        "op":">",
                                        "value":35.54357
                                    }]
                                }, {
                                ...snip...
                                }, {
                                    "population":1,
                                    "predicates":[{
                                        "field":"00001f",
                                        "op":"<=",
                                        "value":35.54357
                                    }]
                                }],
                                "population":2,
                                "predicates":[{
                                    "field":"000005",
                                    "op":"<=",
                                    "value":1385.5166
                                }]
                            }],
                            "population":3,
                            "predicates":[{
                                "field":"000020",
                                "op":"<=",
                                "value":65.14308
                            }, {
                                "field":"000019",
                                "op":"=",
```

(continues on next page)

(continued from previous page)

```
                                "value":0
                            }]
                        }],
                        ...snip...
                        "population":105,
                        "predicates":[{
                            "field":"000017",
                            "op":"<=",
                            "value":13.21754
                        }, {
                            "field":"000009",
                            "op":"in",
                            "value":["0"]
                        }]
                    }],
                    "population":126,
                    "predicates":[true, {
                        "field":"000018",
                        "op":"=",
                        "value":0
                    }]
                },
            },
            "training_mean_depth":11.071428571428571
        }
    },
    "name":"Iris Source dataset's anomaly detector",
    "number_of_anomalyscores":0,
    "number_of_batchanomalyscores":0,
    "number_of_public_anomalyscores":0,
    "ordering":0,
    "out_of_bag":false,
    "price":0.0,
    "private":true,
    "project":null,
    "range":[
        1,
        150
    ],
    "replacement":false,
    "resource":"anomaly/56c432728a318f66e4012f82",
    "rows":150,
    "sample_rate":1.0,
    "sample_size":94,
    "shared":false,
    "size":4609,
    "source":"source/56c432638a318f66e4012f7b",
    "source_status":true,
    "status":{
        "code":5,
        "elapsed":617,
        "message":"The anomaly detector has been created",
        "progress":1.0
    },
    "subscription":false,
    "tags":[],
    "top_n":10,
```

(continues on next page)

(continued from previous page)

```
    "updated":"2016-02-17T08:42:42.238000",
    "white_box":false
}
```

(Note that we have abbreviated the output in the snippet above for readability: the full anomaly detector you'll get is going to contain much more details).

The `trees` list contains the actual isolation forest, and it can be quite large usually. That's why, this part of the resource should only be included in downloads when needed. Each node in an isolation tree can have multiple predicates. For the node to be a valid branch when evaluated with a data point, all of its predicates must be true.

1.3.7 Samples

To provide quick access to your row data you can create a `sample`. Samples are in-memory objects that can be queried for subsets of data by limiting their size, the fields or the rows returned. The structure of a sample would be::

Samples are not permanent objects. Once they are created, they will be available as long as GETs are requested within periods smaller than a pre-established TTL (Time to Live). The expiration timer of a sample is reset every time a new GET is received.

If requested, a sample can also perform linear regression and compute Pearson's and Spearman's correlations for either one numeric field against all other numeric fields or between two specific numeric fields.

1.3.8 Correlations

A `correlation` resource contains a series of computations that reflect the degree of dependence between the field set as objective for your predictions and the rest of fields in your dataset. The dependence degree is obtained by comparing the distributions in every objective and non-objective field pair, as independent fields should have probabilistic independent distributions. Depending on the types of the fields to compare, the metrics used to compute the correlation degree will be:

- for numeric to numeric pairs: Pearson's and Spearman's correlation coefficients.
- for numeric to categorical pairs: One-way Analysis of Variance, with the categorical field as the predictor variable.
- for categorical to categorical pairs: contingency table (or two-way table), Chi-square test of independence , and Cramer's V and Tschuprow's Tcoefficients.

An example of the correlation resource JSON structure is:

```
JSONObject correlation =
    api.getCorrelation("correlation/55b7c4e99841fa24f20009bf");
JSONObject object = (JSONObject) Utils.getJSONObject(
    correlation, "object");
```

correlation `object` object:

```
{
    "category": 0,
    "clones": 0,
    "code": 200,
    "columns": 5,
    "correlations": {
        "correlations": [
```

(continues on next page)

(continued from previous page)

```
            {
                "name": "one_way_anova",
                "result": {
                        "000000": {
                            "eta_square": 0.61871,
                            "f_ratio": 119.2645,
                            "p_value": 0,
                            "significant": [True,
                                True,
                                True
                            ]
                        },
                        "000001": {
                            "eta_square": 0.40078,
                            "f_ratio": 49.16004,
                            "p_value": 0,
                            "significant": [True,
                                True,
                                True
                            ]
                        },
                        "000002": {
                            "eta_square": 0.94137,
                            "f_ratio": 1180.16118,
                            "p_value": 0,
                            "significant": [True,
                                True,
                                True
                            ]
                        },
                        "000003": {
                            "eta_square": 0.92888,
                            "f_ratio": 960.00715,
                            "p_value": 0,
                            "significant": [True,
                                True,
                                True
                            ]
                        }
                    },
                }],
               "fields": {
                    "000000": {
                        "column_number": 0,
                        "datatype": "double",
                        "idx": 0,
                        "name": "sepal length",
                        "optype": "numeric",
                        "order": 0,
                        "preferred": True,
                        "summary": {
                            "bins": [[4.3,1], [4.425,4], ..., [7.9,1]],
                            "kurtosis": -0.57357,
                            "maximum": 7.9,
                            "mean": 5.84333,
                            "median": 5.8,
                            "minimum": 4.3,
```

(continues on next page)

(continued from previous page)

```
                            "missing_count": 0,
                            "population": 150,
                            "skewness": 0.31175,
                            "splits': [4.51526, 4.67252, 4.81113, 4.89582, 4.96139, 5.01131, ..., 6.92597, 7.20423, 7.64746],
                            "standard_deviation": 0.82807,
                            "sum": 876.5,
                            "sum_squares": 5223.85,
                            "variance": 0.68569
                        }
                    },
                    "000001": {
                        "column_number": 1,
                        "datatype": 'double',
                        "idx": 1,
                        "name": "sepal width",
                        "optype": "numeric",
                        "order": 1,
                        "preferred": True,
                        "summary": {
                            'counts': [[2,1], [2.2,
                        ...
                    },
                    ....
                    "000004": {
                        "column_number': 4,
                        "datatype": '"string'",
                        "idx": 4,
                        "name": "species",
                        "optype": "categorical",
                        "order": 4,
                        "preferred": True,
                        "summary": {
                            "categories": [["Iris-setosa", 50],
                                           ["Iris-versicolor",50],
                                           ["Iris-virginica", 50]],
                            "missing_count": 0
                        },
                        "term_analysis": {"enabled": True}
                    }
                },
            "significance_levels": [0.01, 0.05, 0.1]
    },
    "created": "2015-07-28T18:07:37.010000",
    "credits": 0.017581939697265625,
    "dataset": "dataset/55b7a6749841fa2500000d41",
    "dataset_status": True,
    "dataset_type": 0,
    "description": "",
    "excluded_fields": [],
    "fields_meta": {
        "count": 5,
        "limit": 1000,
        "offset": 0,
        "query_total": 5,
        "total": 5},
    "input_fields": ["000000", "000001", "000002", "000003"],
```

(continues on next page)

(continued from previous page)

```
    'locale": "en_US",
    "max_columns": 5,
    "max_rows": 150,
    "name": u"iris' dataset correlation",
    "objective_field_details": {
        "column_number": 4,
        "datatype": "string",
        "name": "species",
        "optype": "categorical",
        "order": 4
    },
    "out_of_bag": False,
    "price": 0.0,
    "private": True,
    "project": None,
    "range": [1, 150],
    "replacement": False,
    "resource": "correlation/55b7c4e99841fa24f20009bf",
    "rows": 150,
    "sample_rate": 1.0,
    "shared": False,
    "size": 4609,
    "source": "source/55b7a6729841fa24f100036a",
    "source_status": True,
    "status": {
        "code": 5,
        "elapsed": 274,
        "message": "The correlation has been created",
        "progress": 1.0
    },
    "subscription": True,
    "tags": [],
    "updated": "2015-07-28T18:07:49.057000",
    "white_box": False
}
```

Note that the output in the snippet above has been abbreviated. As you see, the `correlations` attribute contains the information about each field correlation to the objective field.

1.3.9 Statistical Tests

A `statisticaltest` resource contains a series of tests that compare the distribution of data in each numeric field of a dataset to certain canonical distributions, such as the normal distribution or Benford's law distribution. Statistical test are useful in tasks such as fraud, normality, or outlier detection.

- Fraud Detection Tests: Benford: This statistical test performs a comparison of the distribution of first significant digits (FSDs) of each value of the field to the Benford's law distribution. Benford's law applies to numerical distributions spanning several orders of magnitude, such as the values found on financial balance sheets. It states that the frequency distribution of leading, or first significant digits (FSD) in such distributions is not uniform. On the contrary, lower digits like 1 and 2 occur disproportionately often as leading significant digits. The test compares the distribution in the field to Bendford's distribution using a Chi-square goodness-of-fit test, and Cho-Gaines d test. If a field has a dissimilar distribution, it may contain anomalous or fraudulent values.
- Normality tests: These tests can be used to confirm the assumption that the data in each field of a dataset is distributed according to a normal distribution. The results are relevant because many statistical and machine learning techniques rely on this assumption. Anderson-Darling: The Anderson-Darling test computes a test

statistic based on the difference between the observed cumulative distribution function (CDF) to that of a normal distribution. A significant result indicates that the assumption of normality is rejected. Jarque-Bera: The Jarque-Bera test computes a test statistic based on the third and fourth central moments (skewness and kurtosis) of the data. Again, a significant result indicates that the normality assumption is rejected. Z-score: For a given sample size, the maximum deviation from the mean that would expected in a sampling of a normal distribution can be computed based on the 68-95-99.7 rule. This test simply reports this expected deviation and the actual deviation observed in the data, as a sort of sanity check.

- Outlier tests: Grubbs: When the values of a field are normally distributed, a few values may still deviate from the mean distribution. The outlier tests reports whether at least one value in each numeric field differs significantly from the mean using Grubb's test for outliers. If an outlier is found, then its value will be returned.

An example of the statisticaltest resource JSON structure is:

```
JSONObject statisticalTest = api.getStatisticalTest("statisticaltest/
↪55b7c7089841fa25000010ad");
JSONObject object = (JSONObject) Utils.getJSONObject(
    statisticalTest, "object");
```

statisticalTest `object` object:

```
{
    "category": 0,
    "clones": 0,
    "code": 200,
    "columns": 5,
    "created": "2015-07-28T18:16:40.582000",
    "credits": 0.017581939697265625,
    "dataset": "dataset/55b7a6749841fa2500000d41",
    "dataset_status": True,
    "dataset_type": 0,
    "description": "",
    "excluded_fields": [],
    "fields_meta": {
      "count": 5,
      "limit": 1000,
      "offset": 0,
      "query_total": 5,
      "total": 5
    },
    "input_fields": ["000000", "000001", "000002", "000003"],
    "locale": "en_US",
    "max_columns": 5,
    "max_rows": 150,
    "name": u"iris" dataset test",
    "out_of_bag": False,
    "price": 0.0,
    "private": True,
    "project": None,
    "range": [1, 150],
    "replacement": False,
    "resource": "statisticaltest/55b7c7089841fa25000010ad",
    "rows": 150,
    "sample_rate": 1.0,
    "shared": False,
    "size": 4609,
    "source": "source/55b7a6729841fa24f100036a",
    "source_status": True,
```

(continues on next page)

(continued from previous page)

```
    "status": {
      "code": 5,
      "elapsed": 302,
      "message": "The test has been created",
      "progress": 1.0
    },
    "subscription": True,
    "tags": [],
    "statistical_tests": {
      "ad_sample_size": 1024,
      "fields": {
          "000000": {
              "column_number": 0,
              "datatype": "double",
              "idx": 0,
              "name": "sepal length",
              "optype": "numeric",
              "order": 0,
              "preferred": True,
              "summary": {
                  "bins": [[4.3,1], [4.425,4], ..., [7.9, 1]],
                  "kurtosis": -0.57357,
                  "maximum": 7.9,
                  "mean": 5.84333,
                  "median": 5.8,
                  "minimum": 4.3,
                  "missing_count": 0,
                  "population": 150,
                  "skewness": 0.31175,
                  "splits": [4.51526, 4.67252, 4.81113, 4.89582, ..., 7.20423, 7.
↪64746],
                  "standard_deviation": 0.82807,
                  "sum": 876.5,
                  "sum_squares": 5223.85,
                  "variance": 0.68569
                }
            },
            ...
            "000004": {
                "column_number": 4,
                "datatype": "string",
                "idx": 4,
                "name": "species",
                "optype": "categorical",
                "order": 4,
                "preferred": True,
                "summary": {
                    "categories": [ ["Iris-setosa", 50],
                                    ["Iris-versicolor", 50],
                                    ["Iris-virginica", 50]],
                    "missing_count": 0
                },
                "term_analysis": {"enabled": True}
            }
        },
        "fraud": [
          {
```

(continues on next page)

(continued from previous page)

```
            "name": "benford",
            "result": {
                "000000": {
                    "chi_square": {
                        "chi_square_value": 506.39302,
                        "p_value": 0,
                        "significant": [ True, True, True ]
                    },
                    "cho_gaines": {
                        "d_statistic": 7.124311073683573,
                        "significant": [ True, True, True ]
                    },
                    "distribution": [ 0, 0, 0, 22, 61, 54, 13, 0, 0],
                    "negatives": 0,
                    "zeros": 0
                },
                 "000001": {
                    "chi_square": {
                        "chi_square_value": 396.76556,
                        "p_value": 0,
                        "significant": [ True, True, True ]
                    },
                    "cho_gaines": {
                      "d_statistic": 7.503503138331123,
                      "significant": [ True, True, True ]
                    },
                    "distribution": [ 0, 57, 89, 4, 0, 0, 0, 0, 0],
                    "negatives": 0,
                    "zeros": 0
                },
                .....
            }
        }
      ],
      "normality": [
          {
            "name": "anderson_darling",
            "result": {
                "000000": {
                    "p_value": 0.02252,
                    "significant": [False, True, True]
                },
                 "000001": {
                    "p_value": 0.02023,
                    "significant": [False, True, True]
                },
                 "000002": {
                    "p_value": 0,
                    "significant": [True, True, True]
                },
                 "000003": {
                    "p_value": 0,
                    "significant": [True, True, True]
                }
            }
          },
          {
```

(continues on next page)

(continued from previous page)

```
            "name": "jarque_bera",
            "result": {
              "000000": {
                "p_value": 0.10615,
                "significant": [False, False, False]
              },
               "000001": {
                  "p_value": 0.25957,
                  "significant": [False, False, False]
              },
               "000002": {
                  "p_value": 0.0009,
                  "significant": [True, True, True]
              },
               "000003": {
                  "p_value": 0.00332,
                  "significant": [True, True, True]}
              }
          },
          {
            "name": "z_score",
            "result": {
                "000000": {
                    "expected_max_z": 2.71305,
                    "max_z": 2.48369
                },
                "000001": {
                  "expected_max_z": 2.71305,
                  "max_z": 3.08044
                },
                "000002": {
                  "expected_max_z": 2.71305,
                  "max_z": 1.77987
                },
                "000003": {
                  "expected_max_z": 2.71305,
                  "max_z": 1.70638
                }
            }
          }
        ],
        "outliers": [
          {
             "name": "grubbs",
             "result": {
                "000000": {
                    "p_value": 1,
                    "significant": [False, False, False]
                },
                "000001": {
                    "p_value": 0.26555,
                    "significant": [False, False, False]
                },
                "000002": {
                    "p_value": 1,
                    "significant": [False, False, False]
                },
```

(continues on next page)

(continued from previous page)

```
                "000003": {
                    "p_value": 1,
                    "significant": [False, False, False]
                }
            }
        }
    ],
    "significance_levels": [0.01, 0.05, 0.1]
},
"updated": "2015-07-28T18:17:11.829000",
"white_box": False
}
```

Note that the output in the snippet above has been abbreviated. As you see, the `statistical_tests` attribute contains the `fraud`, `normality` and `outliers` sections where the information for each field's distribution is stored.

1.3.10 Logistic Regressions

A logistic regression is a supervised machine learning method for solving classification problems. Each of the classes in the field you want to predict, the objective field, is assigned a probability depending on the values of the input fields. The probability is computed as the value of a logistic function, whose argument is a linear combination of the predictors' values. You can create a logistic regression selecting which fields from your dataset you want to use as input fields (or predictors) and which categorical field you want to predict, the objective field. Then the created logistic regression is defined by the set of coefficients in the linear combination of the values. Categorical and text fields need some prior work to be modelled using this method. They are expanded as a set of new fields, one per category or term (respectively) where the number of occurrences of the category or term is store. Thus, the linear combination is made on the frequency of the categories or terms.

An example of the logisticregression resource JSON structure is:

```
JSONObject logisticRegression =
api.getLogisticRegression("logisticregression/5617e71c37203f506a000001");
JSONObject object = (JSONObject) Utils.getJSONObject(
    logisticRegression, "object");
```

logisticRegression `object` object:

```
{
    "balance_objective": False,
    "category": 0,
    "code": 200,
    "columns": 5,
    "created": "2015-10-09T16:11:08.444000",
    "credits": 0.017581939697265625,
    "credits_per_prediction": 0.0,
    "dataset": "dataset/561304f537203f4c930001ca",
    "dataset_field_types": {
        "categorical": 1,
        "datetime": 0,
        "effective_fields": 5,
        "numeric": 4,
        "preferred": 5,
        "text": 0,
        "total": 5
```

(continues on next page)

(continued from previous page)

```
    },
    "dataset_status": True,
    "description": "",
    "excluded_fields": [],
    "fields_meta": {
        "count": 5,
        "limit": 1000,
        "offset": 0,
        "query_total": 5,
        "total": 5
    },
    "input_fields": ["000000", "000001", "000002", "000003"],
    "locale": "en_US",
    "logistic_regression": {
        "bias": 1,
        "c": 1,
        "coefficients": [   [   "Iris-virginica",
                                 [   -1.7074433493289376,
                                     -1.533662474502423,
                                     2.47026986670851,
                                     2.5567582221085563,
                                     -1.2158200612711925]],
                             [   "Iris-setosa",
                                 [   0.41021712519841674,
                                     1.464162165246765,
                                     -2.26003266131107,
                                     -1.0210350909174153,
                                     0.26421852991732514]],
                             [   "Iris-versicolor",
                                 [   0.42702327817072505,
                                     -1.611817241669904,
                                     0.5763832839459982,
                                     -1.4069842681625884,
                                     1.0946877732663143]]],
        "eps": 1e-05,
        "fields": {
          "000000": {
              "column_number": 0,
              "datatype": "double",
              "name": "sepal length",
              "optype": "numeric",
              "order": 0,
              "preferred": True,
              "summary": {
                  "bins": [[4.3,1],[4.425,4],[4.6,4],...,[7.9,1]],
                  "kurtosis": -0.57357,
                  "maximum": 7.9,
                  "mean": 5.84333,
                  "median": 5.8,
                  "minimum": 4.3,
                  "missing_count": 0,
                  "population": 150,
                  "skewness": 0.31175,
                  "splits": [4.51526, 4.67252, 4.81113, ..., 6.92597, 7.20423, 7.
↪64746],
                  "standard_deviation": 0.82807,
                  "sum": 876.5,
```

(continues on next page)

(continued from previous page)

```
                "sum_squares": 5223.85,
                "variance": 0.68569
            }
        },
        "000001": {
            "column_number": 1,
            "datatype": "double",
            "name": "sepal width",
            "optype": "numeric",
            "order": 1,
            "preferred": True,
            "summary": {
                "counts": [[2,1],[2.2,3],...,[4.2,1],[4.4,1]],
                "kurtosis": 0.18098,
                "maximum": 4.4,
                "mean": 3.05733,
                "median": 3,
                "minimum": 2,
                "missing_count": 0,
                "population": 150,
                "skewness": 0.31577,
                "standard_deviation": 0.43587,
                "sum": 458.6,
                "sum_squares": 1430.4,
                "variance": 0.18998
            }
        },
         "000002": {
            "column_number": 2,
            "datatype": "double",
            "name": "petal length",
            "optype": "numeric",
            "order": 2,
            "preferred": True,
            "summary": {
                "bins": [[1,1],[1.16667,3],...,[6.6,1],[6.7,2],[6.9,1]],
                "kurtosis": -1.39554,
                "maximum": 6.9,
                "mean": 3.758,
                "median": 4.35,
                "minimum": 1,
                "missing_count": 0,
                "population": 150,
                "skewness": -0.27213,
                "splits": [1.25138,1.32426,1.37171,...,6.02913,6.38125],
                "standard_deviation": 1.7653,
                "sum": 563.7,
                "sum_squares": 2582.71,
                "variance": 3.11628
            }
        },
        "000003": {
            "column_number": 3,
            "datatype": "double",
            "name": "petal width",
            "optype": "numeric",
            "order": 3,
```

(continues on next page)

(continued from previous page)

```
              "preferred": True,
              "summary": {
                  "counts": [[0.1,5],[0.2,29],...,[2.4,3],[2.5,3]],
                  "kurtosis": -1.33607,
                  "maximum": 2.5,
                  "mean": 1.19933,
                  "median": 1.3,
                  "minimum": 0.1,
                  "missing_count": 0,
                  "population": 150,
                  "skewness": -0.10193,
                  "standard_deviation": 0.76224,
                  "sum": 179.9,
                  "sum_squares": 302.33,
                  "variance": 0.58101
              }
          },
          "000004": {
              "column_number": 4,
              "datatype": "string",
              "name": "species",
              "optype": "categorical",
              "order": 4,
              "preferred": True,
              "summary": {
                  "categories": [["Iris-setosa",50],
                                  ["Iris-versicolor",50],
                                  ["Iris-virginica",50]],
                  "missing_count": 0
              },
              "term_analysis": {"enabled": True}
          }
      },
      "normalize": False,
      "regularization": "l2"
    },
    "max_columns": 5,
    "max_rows": 150,
    "name": u"iris" dataset"s logistic regression",
    "number_of_batchpredictions": 0,
    "number_of_evaluations": 0,
    "number_of_predictions": 1,
    "objective_field": "000004",
    "objective_field_name": "species",
    "objective_field_type": "categorical",
    "objective_fields": ["000004"],
    "out_of_bag": False,
    "private": True,
    "project": "project/561304c137203f4c9300016c",
    "range": [1, 150],
    "replacement": False,
    "resource": "logisticregression/5617e71c37203f506a000001",
    "rows": 150,
    "sample_rate": 1.0,
    "shared": False,
    "size": 4609,
    "source": "source/561304f437203f4c930001c3",
```

(continues on next page)

(continued from previous page)

```
    "source_status": True,
    "status": {    "code": 5,
                   "elapsed": 86,
                   "message": "The logistic regression has been created",
                   "progress": 1.0},
    "subscription": False,
    "tags": ["species"],
    "updated": "2015-10-09T16:14:02.336000",
    "white_box": False
}
```

Note that the output in the snippet above has been abbreviated. As you see, the `logistic_regression` attribute stores the coefficients used in the logistic function as well as the configuration parameters described in the developers section.

1.3.11 Linear Regressions

A linear regression is a supervised machine learning method for solving regression problems. The implementation is a multiple linear regression that models the output as a linear combination of the predictors. The coefficients are estimated doing a least-squares fit on the training data.

As a linear combination can only be done using numeric values, non-numeric fields need to be transformed to numeric ones following some rules:

- Categorical fields will be encoded and each class appearance in input data will convey a different contribution to the input vector.
- Text and items fields will be expanded to several numeric predictors, each one indicating the number of occurences for a specific term. Text fields without term analysis are excluded from the model.

Therefore, the initial input data is transformed into an input vector with one or may components per field. Also, if a field in the training data contains missing data, the components corresponding to that field will include an additional 1 or 0 value depending on whether the field is missing in the input data or not.

The JSON structure for a linear regression is:

JSONObject linearRegression = api.getLinearRegression("lineqarregression/5617e71c37203f506a000001"); JSONObject object = (JSONObject) Utils.getJSONObject(linearRegression, "object");

linearRegression `object` object:

```
{
    'category': 0,
    'code': 200,
    'columns': 4,
    'composites': None,
    'configuration': None,
    'configuration_status': False,
    'created': '2019-02-20T21:02:40.027000',
    'creator': 'merce',
    'credits': 0.0,
    'credits_per_prediction': 0.0,
    'dataset': 'dataset/5c6dc06a983efc18e2000084',
    'dataset_field_types': {
        'categorical': 0,
        'datetime': 0,
        'items': 0,
```

(continues on next page)

(continued from previous page)

```
        'numeric': 6,
        'preferred': 6,
        'text': 0,
        'total': 6
    },
    'dataset_status': True,
    'datasets': [],
    'default_numeric_value': None,
    'description': '',
    'excluded_fields': [],
    'execution_id': None,
    'execution_status': None,
    'fields_maps': None,
    'fields_meta': {
        'count': 4,
        'limit': 1000,
        'offset': 0,
        'query_total': 4,
        'total': 4
    },
    'fusions': None,
    'input_fields': ['000000', '000001', '000002'],
    'linear_regression': {
        'bias': True,
        'coefficients': [
            [-1.88196],
            [0.475633],
            [0.122468],
            [30.9141]
        ],
        'fields': {
            '000000': {
                'column_number': 0,
                'datatype': 'int8',
                'name': 'Prefix',
                'optype': 'numeric',
                'order': 0,
                'preferred': True,
                'summary': {
                    'counts': [
                        [4, 1],
        ...

        'stats': {
            'confidence_intervals': [
                [5.63628],
                [0.375062],
                [0.348577],
                [44.4112]
            ],
            'mean_squared_error': 342.206,
            'number_of_parameters': 4,
            'number_of_samples': 77,
            'p_values': [
                [0.512831],
                [0.0129362],
                [0.491069],
```

(continues on next page)

(continued from previous page)

```
                [0.172471]
            ],
            'r_squared': 0.136672,
            'standard_errors': [
                [2.87571],
                [0.191361],
                [0.177849],
                [22.6592]
            ],
            'sum_squared_errors': 24981,
            'xtx_inverse': [
                [4242,
                 48396.9,
                 51273.97,
                 568],
                [48396.9,
                 570177.6584,
                 594274.3274,
                 6550.52],
                [51273.97,
                 594274.3274,
                 635452.7068,
                 6894.24],
                [568,
                 6550.52,
                 6894.24,
                 77]
                ],
            'z_scores': [
                [-0.654436],
                [2.48552],
                [0.688609],
                [1.36431]
            ]
        }
    },
    'locale': 'en_US',
    'max_columns': 6,
    'max_rows': 80,
    'name': 'grades',
    'name_options': 'bias',
    'number_of_batchpredictions': 0,
    'number_of_evaluations': 0,
    'number_of_predictions': 2,
    'number_of_public_predictions': 0,
    'objective_field': '000005',
    'objective_field_name': 'Final',
    'objective_field_type': 'numeric',
    'objective_fields': ['000005'],
    'operating_point': {   },
    'optiml': None,
    'optiml_status': False,
    'ordering': 0,
    'out_of_bag': False,
    'out_of_bags': None,
    'price': 0.0,
    'private': True,
```

(continues on next page)

(continued from previous page)

```
    'project': 'project/5c6dc062983efc18d5000129',
    'range': None,
    'ranges': None,
    'replacement': False,
    'replacements': None,
    'resource': 'linearregression/5c6dc070983efc18e00001f1',
    'rows': 80,
    'sample_rate': 1.0,
    'sample_rates': None,
    'seed': None,
    'seeds': None,
    'shared': False,
    'size': 2691,
    'source': 'source/5c6dc064983efc18e00001ed',
    'source_status': True,
    'status': {
        'code': 5,
        'elapsed': 62086,
        'message': 'The linear regression has been created',
        'progress': 1
    },
    'subscription': True,
    'tags': [],
    'type': 0,
    'updated': '2019-02-27T18:01:18.539000',
    'user_metadata': {},
    'webhook': None,
    'weight_field': None,
    'white_box': False
}
```

Note that the output in the snippet above has been abbreviated. As you see, the `linear_regression` attribute stores the coefficients used in the linear function as well as the configuration parameters described in the developers section.

1.3.12 Associations

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.

An example of the association resource JSON structure is:

```
JSONObject association =
     api.getAssociation("association/5621b70910cb86ae4c000000");
JSONObject object = (JSONObject) Utils.getJSONObject(
    sssociation, "object");
```

association `object` object:

```
{
    "associations":{
        "complement":false,
        "discretization":{
            "pretty":true,
            "size":5,
            "trim":0,
            "type":"width"
        },
        "items":[
            {
                "complement":false,
                "count":32,
                "field_id":"000000",
                "name":"Segment 1",
                "bin_end":5,
```

(continues on next page)

(continued from previous page)

```
                "bin_start":null
            },
            {
                "complement":false,
                "count":49,
                "field_id":"000000",
                "name":"Segment 3",
                "bin_end":7,
                "bin_start":6
            },
            {
                "complement":false,
                "count":12,
                "field_id":"000000",
                "name":"Segment 4",
                "bin_end":null,
                "bin_start":7
            },
            {
                "complement":false,
                "count":19,
                "field_id":"000001",
                "name":"Segment 1",
                "bin_end":2.5,
                "bin_start":null
            },
            ...
            {
                "complement":false,
                "count":50,
                "field_id":"000004",
                "name":"Iris-versicolor"
            },
            {
                "complement":false,
                "count":50,
                "field_id":"000004",
                "name":"Iris-virginica"
            }
        ],
        "max_k": 100,
        "min_confidence":0,
        "min_leverage":0,
        "min_lift":1,
        "min_support":0,
        "rules":[
            {
                "confidence":1,
                "id":"000000",
                "leverage":0.22222,
                "lhs":[
                    13
                ],
                "lhs_cover":[
                    0.33333,
                    50
                ],
```

(continues on next page)

(continued from previous page)

```
                "lift":3,
                "p_value":0.000000000,
                "rhs":[
                    6
                ],
                "rhs_cover":[
                    0.33333,
                    50
                ],
                "support":[
                    0.33333,
                    50
                ]
            },
            {
                "confidence":1,
                "id":"000001",
                "leverage":0.22222,
                "lhs":[
                    6
                ],
                "lhs_cover":[
                    0.33333,
                    50
                ],
                "lift":3,
                "p_value":0.000000000,
                "rhs":[
                    13
                ],
                "rhs_cover":[
                    0.33333,
                    50
                ],
                "support":[
                    0.33333,
                    50
                ]
            },
            ...
            {
                "confidence":0.26,
                "id":"000029",
                "leverage":0.05111,
                "lhs":[
                    13
                ],
                "lhs_cover":[
                    0.33333,
                    50
                ],
                "lift":2.4375,
                "p_value":0.0000454342,
                "rhs":[
                    5
                ],
                "rhs_cover":[
```

(continues on next page)

(continued from previous page)

```
                    0.10667,
                    16
                ],
                "support":[
                    0.08667,
                    13
                ]
            },
            {
                "confidence":0.18,
                "id":"00002a",
                "leverage":0.04,
                "lhs":[
                    15
                ],
                "lhs_cover":[
                    0.33333,
                    50
                ],
                "lift":3,
                "p_value":0.0000302052,
                "rhs":[
                    9
                ],
                "rhs_cover":[
                    0.06,
                    9
                ],
                "support":[
                    0.06,
                    9
                ]
            },
            {
                "confidence":1,
                "id":"00002b",
                "leverage":0.04,
                "lhs":[
                    9
                ],
                "lhs_cover":[
                    0.06,
                    9
                ],
                "lift":3,
                "p_value":0.0000302052,
                "rhs":[
                    15
                ],
                "rhs_cover":[
                    0.33333,
                    50
                ],
                "support":[
                    0.06,
                    9
                ]
```

(continues on next page)

(continued from previous page)

```
            }
        ],
        "rules_summary":{
            "confidence":{
                "counts":[
                    [
                        0.18,
                        1
                    ],
                    [
                        0.24,
                        1
                    ],
                    [
                        0.26,
                        2
                    ],
                    ...
                    [
                        0.97959,
                        1
                    ],
                    [
                        1,
                        9
                    ]
                ],
                "maximum":1,
                "mean":0.70986,
                "median":0.72864,
                "minimum":0.18,
                "population":44,
                "standard_deviation":0.24324,
                "sum":31.23367,
                "sum_squares":24.71548,
                "variance":0.05916
            },
            "k":44,
            "leverage":{
                "counts":[
                    [
                        0.04,
                        2
                    ],
                    [
                        0.05111,
                        4
                    ],
                    [
                        0.05316,
                        2
                    ],
                    ...
                    [
                        0.22222,
                        2
                    ]
```

(continues on next page)

(continued from previous page)

```
                ],
                "maximum":0.22222,
                "mean":0.10603,
                "median":0.10156,
                "minimum":0.04,
                "population":44,
                "standard_deviation":0.0536,
                "sum":4.6651,
                "sum_squares":0.61815,
                "variance":0.00287
            },
            "lhs_cover":{
                "counts":[
                    [
                        0.06,
                        2
                    ],
                    [
                        0.08,
                        2
                    ],
                    [
                        0.10667,
                        4
                    ],
                    [
                        0.12667,
                        1
                    ],
                    ...
                    [
                        0.5,
                        4
                    ]
                ],
                "maximum":0.5,
                "mean":0.29894,
                "median":0.33213,
                "minimum":0.06,
                "population":44,
                "standard_deviation":0.13386,
                "sum":13.15331,
                "sum_squares":4.70252,
                "variance":0.01792
            },
            "lift":{
                "counts":[
                    [
                        1.40625,
                        2
                    ],
                    [
                        1.5067,
                        2
                    ],
                    ...
                    [
```

(continues on next page)

(continued from previous page)

```
                        2.63158,
                        4
                    ],
                    [
                        3,
                        10
                    ],
                    [
                        4.93421,
                        2
                    ],
                    [
                        12.5,
                        2
                    ]
                ],
                "maximum":12.5,
                "mean":2.91963,
                "median":2.58068,
                "minimum":1.40625,
                "population":44,
                "standard_deviation":2.24641,
                "sum":128.46352,
                "sum_squares":592.05855,
                "variance":5.04635
            },
            "p_value":{
                "counts":[
                    [
                        0.000000000,
                        2
                    ],
                    [
                        0.000000000,
                        4
                    ],
                    [
                        0.000000000,
                        2
                    ],
                    ...
                    [
                        0.0000910873,
                        2
                    ]
                ],
                "maximum":0.0000910873,
                "mean":0.0000106114,
                "median":0.00000000,
                "minimum":0.000000000,
                "population":44,
                "standard_deviation":0.0000227364,
                "sum":0.000466903,
                "sum_squares":0.0000000,
                "variance":0.000000001
            },
            "rhs_cover":{
```

(continues on next page)

(continued from previous page)

```
                "counts":[
                    [
                        0.06,
                        2
                    ],
                    [
                        0.08,
                        2
                    ],
                    ...
                    [
                        0.42667,
                        2
                    ],
                    [
                        0.46667,
                        3
                    ],
                    [
                        0.5,
                        4
                    ]
                ],
                "maximum":0.5,
                "mean":0.29894,
                "median":0.33213,
                "minimum":0.06,
                "population":44,
                "standard_deviation":0.13386,
                "sum":13.15331,
                "sum_squares":4.70252,
                "variance":0.01792
            },
            "support":{
                "counts":[
                    [
                        0.06,
                        4
                    ],
                    [
                        0.06667,
                        2
                    ],
                    [
                        0.08,
                        2
                    ],
                    [
                        0.08667,
                        4
                    ],
                    [
                        0.10667,
                        4
                    ],
                    [
                        0.15333,
```

(continues on next page)

(continued from previous page)

```
                        2
                    ],
                    [
                        0.18667,
                        4
                    ],
                    [
                        0.19333,
                        2
                    ],
                    [
                        0.20667,
                        2
                    ],
                    [
                        0.27333,
                        2
                    ],
                    [
                        0.28667,
                        2
                    ],
                    [
                        0.3,
                        4
                    ],
                    [
                        0.32,
                        2
                    ],
                    [
                        0.33333,
                        6
                    ],
                    [
                        0.37333,
                        2
                    ]
                ],
                "maximum":0.37333,
                "mean":0.20152,
                "median":0.19057,
                "minimum":0.06,
                "population":44,
                "standard_deviation":0.10734,
                "sum":8.86668,
                "sum_squares":2.28221,
                "variance":0.01152
            }
        },
        "search_strategy":"leverage",
        "significance_level":0.05
    },
    "category":0,
    "clones":0,
    "code":200,
    "columns":5,
```

(continues on next page)

(continued from previous page)

```
    "created":"2015-11-05T08:06:08.184000",
    "credits":0.017581939697265625,
    "dataset":"dataset/562fae3f4e1727141d00004e",
    "dataset_status":true,
    "dataset_type":0,
    "description":"",
    "excluded_fields":[ ],
    "fields_meta":{
        "count":5,
        "limit":1000,
        "offset":0,
        "query_total":5,
        "total":5
    },
    "input_fields":[
        "000000",
        "000001",
        "000002",
        "000003",
        "000004"
    ],
    "locale":"en_US",
    "max_columns":5,
    "max_rows":150,
    "name":"iris' dataset's association",
    "out_of_bag":false,
    "price":0,
    "private":true,
    "project":null,
    "range":[
        1,
        150
    ],
    "replacement":false,
    "resource":"association/5621b70910cb86ae4c000000",
    "rows":150,
    "sample_rate":1,
    "shared":false,
    "size":4609,
    "source":"source/562fae3a4e1727141d000048",
    "source_status":true,
    "status":{
        "code":5,
        "elapsed":1072,
        "message":"The association has been created",
        "progress":1
    },
    "subscription":false,
    "tags":[ ],
    "updated":"2015-11-05T08:06:20.403000",
    "white_box":false
}
```

Note that the output in the snippet above has been abbreviated. As you see, the `associations` attribute stores items, rules and metrics extracted from the datasets as well as the configuration parameters described in the developers section.

1.3.13 Topic Models

A topic model is an unsupervised machine learning method for unveiling all the different topics underlying a collection of documents. BigML uses Latent Dirichlet Allocation (LDA), one of the most popular probabilistic methods for topic modeling. In BigML, each instance (i.e. each row in your dataset) will be considered a document and the contents of all the text fields given as inputs will be automatically concatenated and considered the document bag of words.

Topic model is based on the assumption that any document exhibits a mixture of topics. Each topic is composed of a set of words which are thematically related. The words from a given topic have different probabilities for that topic. At the same time, each word can be attributable to one or several topics. So for example the word "sea" may be found in a topic related with sea transport but also in a topic related to holidays. Topic model automatically discards stop words and high frequency words.

Topic model's main applications include browsing, organizing and understanding large archives of documents. It can been applied for information retrieval, collaborative filtering, assessing document similarity among others. The topics found in the dataset can also be very useful new features before applying other models like classification, clustering, or anomaly detection.

An example of the topicmodel resource JSON structure is:

```
JSONObject topicModel =
    api.getTopicModel("topicmodel/58362aaa983efc45a1000007");
JSONObject object = (JSONObject) Utils.getJSONObject(topicModel, "object");
```

topicModel `object` object:

```
{
    "category": 0,
    "code": 200,
    "columns": 1,
    "configuration": None,
    "configuration_status": False,
    "created": "2016-11-23T23:47:54.703000",
    "credits": 0.0,
    "credits_per_prediction": 0.0,
    "dataset": "dataset/58362aa0983efc45a0000005",
    "dataset_field_types": {
        "categorical": 1,
        "datetime": 0,
        "effective_fields": 672,
        "items": 0,
        "numeric": 0,
        "preferred": 2,
        "text": 1,
        "total": 2
    },
    "dataset_status": True,
    "dataset_type": 0,
    "description": "",
    "excluded_fields": [],
    "fields_meta": {
        "count": 1,
        "limit": 1000,
        "offset": 0,
        "query_total": 1,
        "total": 1
    },
    "input_fields": ["000001"],
```

(continues on next page)

(continued from previous page)

```
    "locale": "en_US",
    "max_columns": 2,
    "max_rows": 656,
    "name": u"spam dataset"s Topic Model ",
    "number_of_batchtopicdistributions": 0,
    "number_of_public_topicdistributions": 0,
    "number_of_topicdistributions": 0,
    "ordering": 0,
    "out_of_bag": False,
    "price": 0.0,
    "private": True,
    "project": None,
    "range": [1, 656],
    "replacement": False,
    "resource": "topicmodel/58362aaa983efc45a1000007",
    "rows": 656,
    "sample_rate": 1.0,
    "shared": False,
    "size": 54740,
    "source": "source/58362a69983efc459f000001",
    "source_status": True,
    "status": {
        "code": 5,
        "elapsed": 3222,
        "message": "The topic model has been created",
        "progress": 1.0
    },
    "subscription": True,
    "tags": [],
    "topic_model": {
        "alpha": 4.166666666666667,
        "beta": 0.1,
        "bigrams": False,
        "case_sensitive": False,
        "fields": {
            "000001": {
                "column_number": 1,
                "datatype": "string",
                "name": "Message",
                "optype": "text",
                "order": 0,
                "preferred": True,
                "summary": {
                    "average_length": 78.14787,
                    "missing_count": 0,
                    "tag_cloud": [["call",72],["ok",36],...,["yijue",2]],
                    "term_forms": {   }
                },
                "term_analysis": {
                    "case_sensitive": False,
                    "enabled": True,
                    "language": "en",
                    "stem_words": False,
                    "token_mode": "all",
                    "use_stopwords": False
                }
            }
```

(continues on next page)

(continued from previous page)

```
        },
        "hashed_seed": 62146850,
        "language": "en",
        "number_of_topics": 12,
        "term_limit": 4096,
        "term_topic_assignments": [
            [0,5,0,1,0,19,0,0,19,0,1,0],
            [0,0,0,13,0,0,0,0,5,0,0,0],
            ...
            [0,7,27,0,112,0,0,0,0,0,14,2]
        ],
        "termset": ["000","03","04",...,"yr","yup","\xfc"],
        "top_n_terms": 10,
        "topicmodel_seed": "26c386d781963ca1ea5c90dab8a6b023b5e1d180",
        "topics": [   {   "id": "000000",
                           "name": "Topic 00",
                           "probability": 0.09375,
                           "top_terms": [   [   "im",
                                                 0.04849],
                                             [   "hi",
                                                 0.04717],
                                             [   "love",
                                                 0.04585],
                                             [   "please",
                                                 0.02867],
                                             [   "tomorrow",
                                                 0.02867],
                                             [   "cos",
                                                 0.02823],
                                             [   "sent",
                                                 0.02647],
                                             [   "da",
                                                 0.02383],
                                             [   "meet",
                                                 0.02207],
                                             [   "dinner",
                                                 0.01898]]},
                          {   "id": "000001",
                              "name": "Topic 01",
                              "probability": 0.08215,
                              "top_terms": [   [   "lt",
                                                    0.1015],
                                                [   "gt",
                                                    0.1007],
                                                [   "wish",
                                                    0.03958],
                                                [   "feel",
                                                    0.0272],
                                                [   "shit",
                                                    0.02361],
                                                [   "waiting",
                                                    0.02281],
                                                [   "stuff",
                                                    0.02001],
                                                [   "name",
                                                    0.01921],
                                                [   "comp",
```

(continues on next page)

(continued from previous page)

```
                                                    0.01522],
                                                [   "forgot",
                                                    0.01482]]},
                          ...
                         {   "id": "00000b",
                             "name": "Topic 11",
                             "probability": 0.0826,
                             "top_terms": [   [   "call",
                                                    0.15084],
                                                [   "min",
                                                    0.05003],
                                                [   "msg",
                                                    0.03185],
                                                [   "home",
                                                    0.02648],
                                                [   "mind",
                                                    0.02152],
                                                [   "lt",
                                                    0.01987],
                                                [   "bring",
                                                    0.01946],
                                                [   "camera",
                                                    0.01905],
                                                [   "set",
                                                    0.01905],
                                                [   "contact",
                                                    0.01781]]
                        }
            ],
        "use_stopwords": False
    },
    "updated": "2016-11-23T23:48:03.336000",
    "white_box": False
}
```

Note that the output in the snippet above has been abbreviated.

The topic model returns a list of top terms for each topic found in the data. Note that topics are not labeled, so you have to infer their meaning according to the words they are composed of.

Once you build the topic model you can calculate each topic probability for a given document by using Topic Distribution. This information can be useful to find documents similarities based on their thematic.

As you see, the `topic_model` attribute stores the topics and termset and term to topic assignment, as well as the configuration parameters described in the developers section.

1.3.14 Time Series

A time series model is a supervised learning method to forecast the future values of a field based on its previously observed values. It is used to analyze time based data when historical patterns can explain the future behavior such as stock prices, sales forecasting, website traffic, production and inventory analysis, weather forecasting, etc. A time series model needs to be trained with time series data, i.e., a field containing a sequence of equally distributed data points in time.

BigML implements exponential smoothing to train time series models. Time series data is modeled as a level component and it can optionally include a trend (damped or not damped) and a seasonality components. You can learn more about how to include these components and their use in the API documentation page.

You can create a time series model selecting one or several fields from your dataset, that will be the ojective fields. The forecast will compute their future values.

An example of the topicmodel resource JSON structure is:

```
JSONObject timeSeries =
    api.getTimeSeries("timeseries/596a0f66983efc53f3000000");
JSONObject object = (JSONObject) Utils.getJSONObject(timeSeries, "object");
```

timeSeries `object` object:

```
{
    "category": 0,
    "clones": 0,
    "code": 200,
    "columns": 1,
    "configuration": None,
    "configuration_status": False,
    "created": "2017-07-15T12:49:42.601000",
    "credits": 0.0,
    "dataset": "dataset/5968ec42983efc21b0000016",
    "dataset_field_types": {
        "categorical": 0,
        "datetime": 0,
        "effective_fields": 6,
        "items": 0,
        "numeric": 6,
        "preferred": 6,
        "text": 0,
        "total": 6
    },
    "dataset_status": True,
    "dataset_type": 0,
    "description": "",
    "fields_meta": {
        "count": 1,
        "limit": 1000,
        "offset": 0,
        "query_total": 1,
        "total": 1
    },
    "forecast": {
      "000005": [
        {
          "lower_bound": [30.14111, 30.14111, ... 30.14111],
          "model": "A,N,N",
          "point_forecast": [68.53181, 68.53181, ..., 68.53181, 68.53181],
          "time_range": {
               "end": 129,
               "interval": 1,
               "interval_unit": "milliseconds",
               "start": 80
          },
          "upper_bound": [106.92251, 106.92251, ... 106.92251, 106.92251]
        },
        {
          "lower_bound": [35.44118, 35.5032, ..., 35.28083],
          "model": "A,Ad,N",
```

(continues on next page)

(continued from previous page)

```
          ...
             66.83537,
             66.9465],
          "time_range": {
              "end": 129,
              "interval": 1,
              "interval_unit": "milliseconds",
              "start": 80
          }
        }
      ]
    },
    "horizon": 50,
    "locale": "en_US",
    "max_columns": 6,
    "max_rows": 80,
    "name": "my_ts_data",
    "name_options": "period=1, range=[1, 80]",
    "number_of_evaluations": 0,
    "number_of_forecasts": 0,
    "number_of_public_forecasts": 0,
    "objective_field": "000005",
    "objective_field_name": "Final",
    "objective_field_type": "numeric",
    "objective_fields": ["000005"],
    "objective_fields_names": ["Final"],
    "price": 0.0,
    "private": True,
    "project": None,
    "range": [1, 80],
    "resource": "timeseries/596a0f66983efc53f3000000",
    "rows": 80,
    "shared": False,
    "short_url": "",
    "size": 2691,
    "source": "source/5968ec3c983efc218c000006",
    "source_status": True,
    "status": {
        "code": 5,
       "elapsed": 8358,
       "message": "The time series has been created",
       "progress": 1.0
    },
    "subscription": True,
    "tags": [],
    "time_series": {
        "all_numeric_objectives": False,
        "datasets": {
          "000005": "dataset/596a0f70983efc53f3000003"},
          "ets_models": {
              "000005": [
                {
                    "aic": 831.30903,
                    "aicc": 831.84236,
                    "alpha": 0.00012,
                    "beta": 0,
                    "bic": 840.83713,
```

(continues on next page)

(continued from previous page)

```
                    "final_state": {    "b": 0,
                                         "l": 68.53181,
                                         "s": [   0]},
                    "gamma": 0,
                    "initial_state": {    "b": 0,
                                           "l": 68.53217,
                                           "s": [   0]},
                    "name": "A,N,N",
                    "period": 1,
                    "phi": 1,
                    "r_squared": -0.0187,
                    "sigma": 19.19535
                },
                {
                    "aic": 834.43049,
                    ...
                    "slope": 0.11113,
                    "value": 61.39
                }
              ]
          },
          "fields": {
              "000005": {
                  "column_number": 5,
                  "datatype": "double",
                  "name": "Final",
                  "optype": "numeric",
                  "order": 0,
                  "preferred": True,
                  "summary": {
                      "bins": [[28.06,1], ...,  [108.335,2]],
                      ...
                      "sum_squares": 389814.3944,
                      "variance": 380.73315
                  }
              }
          },
          "period": 1,
          "time_range": {
              "end": 79,
              "interval": 1,
              "interval_unit": "milliseconds",
              "start": 0
          }
      },
    "type": 0,
    "updated": "2017-07-15T12:49:52.549000",
    "white_box": False
}
```

1.3.15 OptiMLs

An OptiML is the result of an automated optimization process to find the best model (type and configuration) to solve a particular classification or regression problem.

The selection process automates the usual time-consuming task of trying different models and parameters and evaluating their results to find the best one. Using the OptiML, non-experts can build top-performing models.

You can create an OptiML selecting the ojective field to be predicted, the evaluation metric to be used to rank the models tested in the process and a maximum time for the task to be run.

An example of the optiML resource JSON structure is:

```
JSONObject optiML = api.getOptiML("optiml/5afde4a42a83475c1b0008a2");
JSONObject object = (JSONObject) Utils.getJSONObject(optiML, "object");
```

optiML `object` object:

```
{
    "category": 0,
    "code": 200,
    "configuration": None,
    "configuration_status": False,
    "created": "2018-05-17T20:23:00.060000",
    "creator": "mmartin",
    "dataset": "dataset/5afdb7009252732d930009e8",
    "dataset_status": True,
    "datasets": ["dataset/5afde6488bf7d551ee00081c",
                  "dataset/5afde6488bf7d551fd00511f",
                  "dataset/5afde6488bf7d551fe002e0f",
                  ...
                  "dataset/5afde64d8bf7d551fd00512e"],
    "description": "",
    "evaluations": ["evaluation/5afde65c8bf7d551fd00514c",
                     "evaluation/5afde65c8bf7d551fd00514f",
                     ...
                     "evaluation/5afde6628bf7d551fd005161"],
    "excluded_fields": [],
    "fields_meta": {
        "count": 5,
        "limit": 1000,
        "offset": 0,
        "query_total": 5,
        "total": 5
    },
    "input_fields": ["000000", "000001", "000002", "000003"],
    "model_count": {
                    "linearregression": 1,
        "logisticregression": 1,
        "model": 8,
        "total": 9
    },
    "models": ["model/5afde64e8bf7d551fd005131",
                "model/5afde64f8bf7d551fd005134",
                "model/5afde6518bf7d551fd005137",
                "model/5afde6538bf7d551fd00513a",
                "linearregression/5c8f576e1f386f7dc3000048",
                "logisticregression/5afde6558bf7d551fd00513d",
                ...
                "model/5afde65a8bf7d551fd005149"],
    "models_meta": {
        "count": 9,
        "limit": 1000,
        "offset": 0,
        "total": 9
```

(continues on next page)

(continued from previous page)

```
    },
    "name": "iris",
    "name_options": "9 total models (linearregression: 1, logisticregression: 1, model: 8), metric=max_phi, model candidates=18, max. training time=300",
    "objective_field": "000004",
    "objective_field_details": {
        "column_number": 4,
        "datatype": "string",
        "name": "species",
        "optype": "categorical",
        "order": 4
    },
    "objective_field_name": "species",
    "objective_field_type": "categorical",
    "objective_fields": ["000004"],
    "optiml": {
        "created_resources": {
            "dataset": 10,
            "linearregression": 1,
            "logisticregression": 11,
            "logisticregression_evaluation": 11,
            "model": 29,
            "model_evaluation": 29
        },
       "datasets": [   {   "id": "dataset/5afde6488bf7d551ee00081c",
                            "name": "iris",
                            "name_options": "120 instances, 5 fields (1 categorical, 4 numeric), sample rate=0.8"},
                        {   "id": "dataset/5afde6488bf7d551fd00511f",
                            "name": "iris",
                            "name_options": "30 instances, 5 fields (1 categorical, 4 numeric), sample rate=0.2, out of bag"},
                        {   "id": "dataset/5afde6488bf7d551fe002e0f",
                            "name": "iris",
                            "name_options": "120 instances, 5 fields (1 categorical, 4 numeric), sample rate=0.8"},
                        ...
                        {   "id": "dataset/5afde64d8bf7d551fd00512e",
                            "name": "iris",
                            "name_options": "120 instances, 5 fields (1 categorical, 4 numeric), sample rate=0.8"}],
        "fields": {
          "000000": {
               "column_number": 0,
               "datatype": "double",
               "name": "sepal length",
               "optype": "numeric",
               "order": 0,
               "preferred": True,
               "summary": {
                  "bins": [[4.3,1], ..., [7.9,1]],
                  ...
                  "sum": 179.9,
                  "sum_squares": 302.33,
                  "variance": 0.58101
               }
          },
```

(continues on next page)

(continued from previous page)

```
        "000004": {
             "column_number": 4,
             "datatype": "string",
             "name": "species",
             "optype": "categorical",
             "order": 4,
             "preferred": True,
             "summary": {
                "categories": [["Iris-setosa",50],
                                ["Iris-versicolor",50],
                                ["Iris-virginica",50]],
                "missing_count": 0
            },
            "term_analysis": {"enabled": True}
        }
    },
    "max_training_time": 300,
    "metric": "max_phi",
    "model_types": ["model", "linearregression", "logisticregression"],
    "models": [
      {
          "evaluation": {
             "id": "evaluation/5afde65c8bf7d551fd00514c",
             "info": {
                "accuracy": 0.96667,
                "average_area_under_pr_curve": 0.97867,
                ...
                "per_class_statistics": [
                  {
                     "accuracy": 1,
                     "area_under_pr_curve": 1,
                     ...
                     "spearmans_rho": 0.82005
                  }
                ]
             },
             "metric_value": 0.95356,
             "metric_variance": 0.00079,
             "name": "iris vs. iris",
             "name_options": "279-node, deterministic order, operating␣
↪kind=probability"
          },
          "evaluation_count": 3,
          "id": "model/5afde64e8bf7d551fd005131",
          "importance": [    [    "000002",
                                   0.70997],
                              [    "000003",
                                   0.27289],
                              [    "000000",
                                   0.0106],
                              [    "000001",
                                   0.00654]],
          "kind": "model",
          "name": "iris",
          "name_options": "279-node, deterministic order"
      },
      ....
```

(continues on next page)

(continued from previous page)

```
    }
    "private": True,
    "project": None,
    "resource": "optiml/5afde4a42a83475c1b0008a2",
    "shared": False,
    "size": 3686,
    "source": "source/5afdb6fb9252732d930009e5",
    "source_status": True,
    "status": {
          "code": 5,
         "elapsed": 448878.0,
         "message": "The optiml has been created",
         "progress": 1
    },
    "subscription": False,
    "tags": [],
    "test_dataset": None,
    "type": 0,
    "updated": "2018-05-17T20:30:29.063000"
}
```

1.3.16 Fusions

A Fusion is a special type of composed resource for which all submodels satisfy the following constraints: they're all either classifications or regressions over the same kind of data or compatible fields, with the same objective field. Given those properties, a fusion can be considered a supervised model, and therefore one can predict with fusions and evaluate them. Ensembles can be viewed as a kind of fusion subject to the additional constraints that all its submodels are tree models that, moreover, have been built from the same base input data, but sampled in particular ways.

The model types allowed to be a submodel of a fusion are: deepnet, ensemble, fusion, model, logistic regression and linear regression.

An example of the fusion resource JSON structure is:

```
JSONObject fusion = api.getFusion("fusion/59af8107b8aa0965d5b61138");
JSONObject object = (JSONObject) Utils.getJSONObject(fusion, "object");
```

fusion `object` object:

```
{
    "category": 0,
    "code": 200,
    "configuration": null,
    "configuration_status": false,
    "created": "2018-05-09T20:11:05.821000",
    "credits_per_prediction": 0,
    "description": "",
    "fields_meta": {
        "count": 5,
        "limit": 1000,
        "offset": 0,
        "query_total": 5,
        "total": 5
    },
    "fusion": {
```

(continues on next page)

(continued from previous page)

```
        "models": [
            {
                "id": "ensemble/5af272eb4e1727d378000050",
                "kind": "ensemble",
                "name": "Iris ensemble",
                "name_options": "boosted trees, 1999-node, 16-iteration,␣
↪deterministic order, balanced"
            },
            {
                "id": "model/5af272fe4e1727d3780000d6",
                "kind": "model",
                "name": "Iris model",
                "name_options": "1999-node, pruned, deterministic order, balanced"
            },
            {
                "id": "logisticregression/5af272ff4e1727d3780000d9",
                "kind": "logisticregression",
                "name": "Iris LR",
                "name_options": "L2 regularized (c=1), bias, auto-scaled, missing␣
↪values, eps=0.001"
            },
            {
                "id": "linearregression/5c8f576e1f386f7dc3000048",
                "kind": "linearregression",
                "name": "Iris Linear Regression",
                "name_options": "bias"
            }
        ]
    },
    "importance": {
        "000000": 0.05847,
        "000001": 0.03028,
        "000002": 0.13582,
        "000003": 0.4421
    },
    "model_count": {
        "ensemble": 1,
        "linearregression": 1,
        "logisticregression": 1,
        "model": 1,
        "total": 3
    },
    "models": [
        "ensemble/5af272eb4e1727d378000050",
        "model/5af272fe4e1727d3780000d6",
        "linearregression/5c8f576e1f386f7dc3000048",
        "logisticregression/5af272ff4e1727d3780000d9"
    ],
    "models_meta": {
        "count": 3,
        "limit": 1000,
        "offset": 0,
        "total": 3
    },
    "name": "iris",
    "name_options": "3 total models (ensemble: 1, linearregression: 1,␣
↪logisticregression: 1, model: 1)",
```

(continues on next page)

(continued from previous page)

```
    "number_of_batchpredictions": 0,
    "number_of_evaluations": 0,
    "number_of_predictions": 0,
    "number_of_public_predictions": 0,
    "objective_field": "000004",
    "objective_field_details": {
        "column_number": 4,
        "datatype": "string",
        "name": "species",
        "optype": "categorical",
        "order": 4
    },
    "objective_field_name": "species",
    "objective_field_type": "categorical",
    "objective_fields": [
        "000004"
    ],
    "private": true,
    "project": null,
    "resource":"fusion/59af8107b8aa0965d5b61138",
    "shared": false,
    "status": {
        "code": 5,
        "elapsed": 8420,
        "message": "The fusion has been created",
        "progress": 1
    },
    "subscription": false,
    "tags": [],
    "type": 0,
    "updated": "2018-05-09T20:11:14.258000"
}
```

1.4 Resources

1.4.1 Creating Resources

Newly-created resources are returned in a dictionary with the following keys:

- **code**: If the request is successful you will get a `HTTP_CREATED` (201) status code. In asynchronous file uploading `api.createSource` calls, it will contain `HTTP_ACCEPTED` (202) status code. Otherwise, it will be one of the standard HTTP error codes detailed in the documentation.
- **resource**: The identifier of the new resource.
- **location**: The location of the new resource.
- **object**: The resource itself, as computed by BigML.
- **error**: If an error occurs and the resource cannot be created, it will contain an additional code and a description of the error. In this case, **location**, and **resource** will be `None`.

Statuses

Please, bear in mind that resource creation is almost always asynchronous (**predictions** are the only exception). Therefore, when you create a new source, a new dataset or a new model, even if you receive an immediate response from the BigML servers, the full creation of the resource can take from a few seconds to a few days, depending on the size of the resource and BigML's load. A resource is not fully created until its status is `FINISHED`. See the documentation on status codes for the listing of potential states and their semantics. So depending on your application you might need to import the following constants:

```
import org.bigml.binding.resources.AbstractResource;

AbstractResource.FINISHED
AbstractResource.QUEUED
AbstractResource.STARTED
AbstractResource.IN_PROGRESS
AbstractResource.SUMMARIZED
AbstractResource.FINISHED
AbstractResource.UPLOADING
AbstractResource.FAULTY
AbstractResource.UNKNOWN
AbstractResource.RUNNABLE
```

Usually, you will simply need to wait until the resource is in the `FINISHED` state for further processing. If that's the case, the easiest way is calling the `api.xxxIsReady` method and passing as first argument the object that contains your resource:

```
import org.bigml.binding.BigMLClient;

// Create BigMLClient with the properties in binding.properties
BigMLClient api = new BigMLClient();

// creates a source object
JSONObject source = api.createSource("my_file.csv");

// checks that the source is finished and updates ``source``
while (!api.sourceIsReady(source))
    Thread.sleep(1000);
```

In this code, `api.createSource` will probably return a non-finished `source` object. Then, `api.sourceIsReady` will query its status and update the contents of the `source` variable with the retrieved information until it reaches a `FINISHED` or `FAILED` status.

Remember that, consequently, you will need to retrieve the resources explicitly in your code to get the updated information.

Projects

A special kind of resource is `project`. Projects are repositories for resources, intended to fulfill organizational purposes. Each project can contain any other kind of resource, but the project that a certain resource belongs to is determined by the one used in the `source` they are generated from. Thus, when a source is created and assigned a certain `project_id`, the rest of resources generated from this source will remain in this project.

The REST calls to manage the `project` resemble the ones used to manage the rest of resources. When you create a `project`:

```
    import org.bigml.binding.BigMLClient;

    // Create BigMLClient with the properties in binding.properties
    BigMLClient api = new BigMLClient();

    JSONObject project = api.createProject({"name": "my first project"});
```

the resulting resource is similar to the rest of resources, although shorter:

```
{
    "code": 201,
    "resource": "project/54a1bd0958a27e3c4c0002f0",
    "location": "http://bigml.io/andromeda/project/54a1bd0958a27e3c4c0002f0",
    "object": {
        "category": 0,
        "updated": "2014-12-29T20:43:53.060045",
        "resource": "project/54a1bd0958a27e3c4c0002f0",
        "name": "my first project",
        "created": "2014-12-29T20:43:53.060013",
        "tags": [],
        "private": True,
        "dev": None,
        "description": ""
    },
    "error": None
}
```

and you can use its project id to get, update or delete it:

```
    JSONObject project = api.getProject("project/54a1bd0958a27e3c4c0002f0");
    String resource = (String) Utils.getJSONObject(
        project, "resource");
    api.updateProject(resource,
                      {'description': 'This is my first project'});

    api.deleteProject(resource);
```

Important: Deleting a non-empty project will also delete **all resources** assigned to it, so please be extra-careful when using the `api.deleteProject` call.

External Connectors

To create an external connector to an existing database you need to use the `createExternalConnector` method. The only two required parameters are the the name of the external data source to connect to (allowed types are: `elasticsearch`, `postgresql`, `mysql`, `sqlserver`) and the dictionary that contains the information needed to connect to the particular database/table. The attributes of the connection dictionary needed for the method to work will depend on the type of database used.

For instance, you can create a connection to an `Elasticsearch` database hosted locally at port `9200` by calling:

```
    import org.bigml.binding.BigMLClient;

    // Create BigMLClient with the properties in binding.properties
    BigMLClient api = new BigMLClient();
```

(continues on next page)

(continued from previous page)

```
    JSONObject connectionInfo = JSONValue.parse(
        "{\"hosts\": [\"elasticsearch\"]}"
    );
    JSONObject externalConnector = api.createExternalConnector(
        elasticsearch, connectionInfo);
```

Sources

To create a source from a local data file, you can use the `createSource` method. The only required parameter is the path to the data file (or file-like object). You can use a second optional parameter to specify any of the options for source creation described in the BigML API documentation.

Here's a sample invocation:

```
    import org.bigml.binding.BigMLClient;

    // Create BigMLClient with the properties in binding.properties
    BigMLClient api = new BigMLClient();

    JSONObject args = JSONValue.parse(
        "{\"name\": \"my source\",
           \"source_parser\": {\"missing_tokens\": [\"?\""]}}"
    );
    JSONObject source = api.createSource("./data/iris.csv", args);
```

or you may want to create a source from a file in a remote location:

```
source = api.createRemoteSource("s3://bigml-public/csv/iris.csv", args)
```

or using data stored in a local java variable. The following example shows the two accepted formats:

```
    String inline = "[{\"a\": 1, \"b\": 2, \"c\": 3},
                      {\"a\": 4, \"b\": 5, \"c\": 6}]";
    JSONObject args = JSONValue.parse("{\"name\": \"inline source\"}");
    JSONObject source = api.createInlineSource(
        inline, {'name': 'inline source'});
```

As already mentioned, source creation is asynchronous. In both these examples, the `api.createSource` call returns once the file is uploaded. Then `source` will contain a resource whose status code will be either `WAITING` or `QUEUED`.

Datasets

Once you have created a source, you can create a dataset. The only required argument to create a dataset is a source id. You can add all the additional arguments accepted by BigML and documented in the Datasets section of the Developer's documentation.

For example, to create a dataset named "my dataset" with the first 1024 bytes of a source, you can submit the following request:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my dataset\", \"size\": 1024}");
    JSONObject dataset = api.createDataset(source, args);
```

Upon success, the dataset creation job will be queued for execution, and you can follow its evolution using `api.datasetIsReady(dataset)`.

As for the rest of resources, the create method will return an incomplete object, that can be updated by issuing the corresponding `api.getDataset` call until it reaches a `FINISHED` status. Then you can export the dataset data to a CSV file using:

```
api.downloadDataset("dataset/526fc344035d071ea3031d75",
    filename="my_dir/my_dataset.csv");
```

You can also extract samples from an existing dataset and generate a new one with them using the `api.createDataset` method. The first argument should be the origin dataset and the rest of arguments that set the range or the sampling rate should be passed as a dictionary. For instance, to create a new dataset extracting the 80% of instances from an existing one, you could use:

```
JSONObject originDataset = api.createSataset(source);
JSONObject sampleArgs = JSONValue.parseValue("{\"sample_rate\": 0.8}");
JSONObjectdataset = api.createDataset(originDataset, sampleArgs);
```

Similarly, if you want to split your source into training and test datasets, you can set the `sample_rate` as before to create the training dataset and use the `out_of_bag` option to assign the complementary subset of data to the test dataset. If you set the `seed` argument to a value of your choice, you will ensure a deterministic sampling, so that each time you execute this call you will get the same datasets as a result and they will be complementary:

```
    JSONObject originDataset = api.createSataset(source);

    JSONObject trainArgs = JSONValue.parseValue(
        "{\"name\": \"Dataset Name | Training\",
          \"sample_rate\": 0.8,
          \"seed\": \"my seed\"}");
    JSONObject trainDataset = api.createDataset(originDataset, trainArgs);

    JSONObject testArgs = JSONValue.parseValue(
        "{\"name\": \"Dataset Name | Test\",
          \"sample_rate\": 0.8,
          \"seed\": \"my seed\",
          \"out_of_bag\": true}");
    JSONObject testDataset = api.createDataset(originDataset, testArgs);
```

Sometimes, like for time series evaluations, it's important that the data in your train and test datasets is ordered. In this case, the split cannot be done at random. You will need to start from an ordered dataset and decide the ranges devoted to training and testing using the `range` attribute:

```
    JSONObject originDataset = api.createSataset(source);

    JSONObject trainArgs = JSONValue.parseValue(
        "{\"name\": \"Dataset Name | Training\",
          \"range\": [1, 80]}");
    JSONObject trainDataset = api.createDataset(originDataset, trainArgs);

    JSONObject testArgs = JSONValue.parseValue(
        "{\"name\": \"Dataset Name | Test\",
```

(continues on next page)

(continued from previous page)

```
          \"range\": [81, 100]}");
    JSONObject testDataset = api.createDataset(originDataset, testArgs);
```

It is also possible to generate a dataset from a list of datasets (multidataset):

```
    JSONObject dataset1 = api.createDataset(source1);
    JSONObject dataset2 = api.createDataset(source2);
    List datasetsIds = new ArrayList();
    datasetsIds.add(dataset1);
    datasetsIds.add(dataset2);
    JSONObject multidataset = api.createDataset(datasetsIds);
```

Clusters can also be used to generate datasets containing the instances grouped around each centroid. You will need the cluster id and the centroid id to reference the dataset to be created. For instance,

```
    JSONObject cluster = api.createCluster(dataset);
    JSONObject args = JSONValue.parseValue("{\"centroid\": \"000000\"}");
    JSONObject clusterDataset1 = api.createDataset(cluster, args);
```

would generate a new dataset containing the subset of instances in the cluster associated to the centroid id `000000`.

Models

Once you have created a dataset you can create a model from it. If you don't select one, the model will use the last field of the dataset as objective field. The only required argument to create a model is a dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the Models section of the Developer's documentation.

For example, to create a model only including the first two fields and the first 10 instances in the dataset, you can use the following invocation:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my model\",
          \"input_fields\": [\"000000\", \"000001\"],
          \"range\": [1, 10]}");
    JSONObject model = api.createModel(dataset, args);
```

Again, the model is scheduled for creation, and you can retrieve its status at any time by means of `api.modelIsReady(model)`.

Models can also be created from lists of datasets. Just use the list of ids as the first argument in the api call

```
    JSONObject dataset1 = api.createDataset(source1);
    JSONObject dataset2 = api.createDataset(source2);
    List datasetsIds = new ArrayList();
    datasetsIds.add(dataset1);
    datasetsIds.add(dataset2);
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my model\",
          \"input_fields\": [\"000000\", \"000001\"],
          \"range\": [1, 10]}");
    JSONObject model = api.createModel(datasetsIds, args);
```

And they can also be generated as the result of a clustering procedure. When a cluster is created, a model that predicts if a certain instance belongs to a concrete centroid can be built by providing the cluster and centroid ids:

```
    JSONObject cluster = api.createCluster(dataset);
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"model for centroid 000001\",
          \"centroid\": \"000001\"}");
    JSONObject model = api.createModel(cluster, args);
```

if no centroid id is provided, the first one appearing in the cluster is used.

Clusters

If your dataset has no fields showing the objective information to predict for the training data, you can still build a cluster that will group similar data around some automatically chosen points (centroids). Again, the only required argument to create a cluster is the dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the Clusters section of the Developer's documentation.

Let's create a cluster from a given dataset:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my cluster\", \"k\": 5}");
    JSONObject cluster = api.createCluster(dataset, args);
```

that will create a cluster with 5 centroids.

Anomaly detectors

If your problem is finding the anomalous data in your dataset, you can build an anomaly detector, that will use iforest to single out the anomalous records. Again, the only required argument to create an anomaly detector is the dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the Anomaly detectors section of the Developer's documentation.

Let's create an anomaly detector from a given dataset:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my anomaly\"}");
    JSONObject anomaly = api.createAnomaly(dataset, args);
```

that will create an anomaly resource with a `top_anomalies` block of the most anomalous points.

Associations

To find relations between the field values you can create an association discovery resource. The only required argument to create an association is a dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the [Association section of the Developer's documentation](https://bigml.com/api/associations.

For example, to create an association only including the first two fields and the first 10 instances in the dataset, you can use the following invocation:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my association\",
          \"input_fields\": [\"000000\", \"000001\"],
          \"range\": [1, 10]}");
    JSONObject association = api.createAssociation(dataset, args);
```

Again, the association is scheduled for creation, and you can retrieve its status at any time by means of `api.associtionIsReady(association)`.

Associations can also be created from lists of datasets. Just use the list of ids as the first argument in the api call

```
    List datasetsIds = new ArrayList();
    datasetsIds.add(dataset1);
    datasetsIds.add(dataset2);
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my association\",
          \"input_fields\": [\"000000\", \"000001\"],
          \"range\": [1, 10]}");
    JSONObject association = api.createAssociation(dataset, args);
```

Topic models

To find which topics do your documents refer to you can create a topic model. The only required argument to create a topic model is a dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the Topic Model section of the Developer's documentation.

For example, to create a topic model including exactly 32 topics you can use the following invocation:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my topics\",
          \"number_of_topics\": 32}");
    JSONObject topicModel = api.createTopicModel(dataset, args);
```

Again, the topic model is scheduled for creation, and you can retrieve its status at any time by means of `api.topicModelIsReady(topicModel)`.

Topic models can also be created from lists of datasets. Just use the list of ids as the first argument in the api call.

```
    List datasetsIds = new ArrayList();
    datasetsIds.add(dataset1);
    datasetsIds.add(dataset2);
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my topics\",
          \"number_of_topics\": 32}");
    JSONObject topicModel = api.createTopicModel(datasetsIds, args);
```

Time series

To forecast the behaviour of any numeric variable that depends on its historical records you can use a time series. The only required argument to create a time series is a dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the [Time Series section of the Developer's documentation](https://bigml.com/api/timeseries.

For example, to create a time series including a forecast of 10 points for the numeric values you can use the following invocation:

```
JSONObject args = JSONValue.parseValue(
    "{\"name\": \"my time series\",
      \"horizon\": 10}");
JSONObject timeSeries = api.createTimeSeries(dataset, args);
```

Again, the time series is scheduled for creation, and you can retrieve its status at any time by means of `api.timeSeriesIsReady(timeSeries)`.

Time series also be created from lists of datasets. Just use the list of ids as the first argument in the api call

```
List datasetsIds = new ArrayList();
datasetsIds.add(dataset1);
datasetsIds.add(dataset2);
JSONObject args = JSONValue.parseValue(
    "{\"name\": \"my time series\",
      \"horizon\": 10}");
JSONObject timeSeries = api.createTimeSeries(datasetsIds, args);
```

OptiML

To create an OptiML, the only required argument is a dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the OptiML section of the Developer's documentation.

For example, to create an OptiML which optimizes the accuracy of the model you can use the following method

```
JSONObject args = JSONValue.parseValue(
    "{\"name\": \"my optiml\",
      \"metric\": \"accuracy\"}");
JSONObject optiml = api.createOptiML(dataset, args);
```

The OptiML is then scheduled for creation, and you can retrieve its status at any time by means of `api.optiMLIsReady(optiml)`.

Fusion

To create a Fusion, the only required argument is a list of models. You can also include in the request all the additional arguments accepted by BigML and documented in the Fusion section of the Developer's documentation.

For example, to create a Fusion you can use this connection method:

```
List modelsIds = new ArrayList();
modelsIds.add("model/5af06df94e17277501000010");
modelsIds.add("model/5af06df84e17277502000019");
modelsIds.add("deepnet/5af06df84e17277502000016");
modelsIds.add("ensemble/5af06df74e1727750100000d");
JSONObject args = JSONValue.parseValue("{\"name\": \"my fusion\"}");
JSONObject fusion = api.createFusion(modelsIds, args);
```

The Fusion is then scheduled for creation, and you can retrieve its status at any time by means of `api.fusionIsReady(fusion)`.

Fusions can also be created by assigning some weights to each model in the list. In this case, the argument for the create call will be a list of dictionaries that contain the `id` and `weight` keys:

```
    JSONArray models = JSONValue.parseValue(
        "[{\"id\": \"model/5af06df94e17277501000010\", \"weight\": 10},
          {\"id\": \"model/5af06df84e17277502000019\", \"weight\": 20},
          {\"id\": \"deepnet/5af06df84e17277502000016\",\"weight\": 5}]}");
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my weighted fusion\"}");
    JSONObject fusion = api.createFusion(models, args);
```

Predictions

You can now use the model resource identifier together with some input parameters to ask for predictions, using the `createPrediction` method. You can also give the prediction a name:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"sepal length\": 5,
          \"sepal width\": 2.5});
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my prediction\"}");
    JSONObject prediction = api.createPrediction(
        "model/5af272fe4e1727d3780000d6", inputData, args);
```

Predictions can be created using any supervised model (model, ensemble, logistic regression, linear regression, deepnet and fusion) as first argument.

Centroids

To obtain the centroid associated to new input data, you can now use the `createCentroid` method. Give the method a cluster identifier and the input data to obtain the centroid. You can also give the centroid predicition a name:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"pregnancies\": 0,
          \"plasma glucose\": 118,
          \"blood pressure\": 84,
          \"triceps skin thickness\": 47}");
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my centroid\"}");
    JSONObject centroid = api.createCentroid(
        "cluster/56c42ea47e0a8d6cca0151a0", inputData, args);
```

Anomaly scores

To obtain the anomaly score associated to new input data, you can now use the `createAnomalyScore` method. Give the method an anomaly detector identifier and the input data to obtain the score:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"src_bytes\": 350}");
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my score\"}");
    anomaly_score = api.create_anomaly_score(
        "anomaly/56c432728a318f66e4012f82", inputData, args);
```

Association sets

Using the association resource, you can obtain the consequent items associated by its rules to your input data. These association sets can be obtained calling the `createAssociationSet` method. The first argument is the association ID and the next one is the input data.

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"genres\": \"Action$Adventure\"}");
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my association set\"}");
    JSONObject associationSet = api.createAssociationSet(
        "association/5621b70910cb86ae4c000000", inputData);
```

Topic distributions

To obtain the topic distributions associated to new input data, you can now use the `createTopicDistribution` method. Give the method a topic model identifier and the input data to obtain the score:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"Message\": \"The bubble exploded in 2007.\"}");
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my topic distribution\"}");
    JSONObject topicDistribution = api.createTopicDistribution(
        "topicmodel/58362aaa983efc45a1000007", inputData, args);
```

Forecasts

To obtain the forecast associated to a numeric variable, you can now use the `createForecast` method. Give the method a time series identifier and the input data to obtain the forecast:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"Final\": {\"horizon\": 10}}");
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my forecast\"}");
    JSONObject forecast = api.createForecast(
        "timeseries/596a0f66983efc53f3000000", inputData, args);
```

Evaluations

Once you have created a supervised learning model, you can measure its perfomance by running a dataset of test data through it and comparing its predictions to the objective field real values. Thus, the required arguments to create an evaluation are model id and a dataset id. You can also include in the request all the additional arguments accepted by BigML and documented in the Evaluations section of the Developer's documentation.

For instance, to evaluate a previously created model using an existing dataset you can use the following call:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my evaluation\"}");
    JSONObject evaluation = api.createEvaluation(
        "model/5afde64e8bf7d551fd005131",
        "dataset/5afde6488bf7d551ee00081c",
        args);
```

Again, the evaluation is scheduled for creation and `api.evaluationIsReady(evaluation)` will show its state.

Evaluations can also check the ensembles' performance. To evaluate an ensemble you can do exactly what we just did for the model case, using the ensemble object instead of the model as first argument:

```
JSONObject evaluation = api.createEvaluation(
    "ensemble/5af272eb4e1727d378000050",
    "dataset/5afde6488bf7d551ee00081c");
```

Evaluations can be created using any supervised model (including time series) as first argument.

Ensembles

To improve the performance of your predictions, you can create an ensemble of models and combine their individual predictions. The only required argument to create an ensemble is the dataset id:

```
JSONObject ensemble = api.createEnsemble(
    "dataset/5143a51a37203f2cf7000972");
```

BigML offers three kinds of ensembles. Two of them are known as `Decision Forests` because they are built as collections of `Decision trees` whose predictions are aggregated using different combiners (`plurality`, `confidence weighted`, `probability weighted`) or setting a `threshold` to issue the ensemble's prediction. All `Decision Forests` use bagging to sample the data used to build the underlying models.

As an example of how to create a `Decision Forest` with `20` models, you only need to provide the dataset ID that you want to build the ensemble from and the number of models:

```
    JSONObject args = JSONValue.parseValue(
        "{\"number_of_models\": 20}");
    JSONObject ensemble = api.createEnsemble(
        "dataset/5143a51a37203f2cf7000972", args);
```

If no `number_of_models` is provided, the ensemble will contain 10 models.

`Random Decision Forests` fall also into the `Decision Forest` category, but they only use a subset of the fields chosen at random at each split. To create this kind of ensemble, just use the `randomize` option:

```
    JSONObject args = JSONValue.parseValue(
        "{\"number_of_models\": 20,
          \"randomize\": true}");
    JSONObject ensemble = api.createEnsemble(
        "dataset/5143a51a37203f2cf7000972", args);
```

The third kind of ensemble is `Boosted Trees`. This type of ensemble uses quite a different algorithm. The trees used in the ensemble don’t have as objective field the one you want to predict, and they don’t aggregate the underlying models’ votes. Instead, the goal is adjusting the coefficients of a function that will be used to predict. The models’ objective is, therefore, the gradient that minimizes the error of the predicting function (when comparing its output with the real values). The process starts with some initial values and computes these gradients. Next step uses the previous fields plus the last computed gradient field as the new initial state for the next iteration. Finally, it stops when the error is smaller than a certain threshold or iterations reach a user-defined limit. In classification problems, every category in the ensemble’s objective field would be associated with a subset of the `Boosted Trees`. The objective of each subset of trees is adjustig the function to the probability of belonging to this particular category.

In order to build an ensemble of `Boosted Trees` you need to provide the `boosting` attributes. You can learn about the existing attributes in the ensembles’ section of the API documentation, but a typical attribute to be set would be the maximum number of iterations:

```
    args = {'boosting': {'iterations': 20}}
    ensemble = api.create_ensemble('dataset/5143a51a37203f2cf7000972', args)

    JSONObject args = JSONValue.parseValue(
        "{\"boosting\": {\"iterations\": 20}");
    JSONObject ensemble = api.createEnsemble(
        "dataset/5143a51a37203f2cf7000972", args);
```

Linear regressions

For regression problems, you can choose also linear regressions to model your data. Linear regressions expect the predicted value for the objective field to be computable as a linear combination of the predictions.

As the rest of models, linear regressions can be created from a dataset by calling the corresponding create method:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my linear regression\",
          \"objective_field\": \"my_objective_field\"}");
    JSONObject linearRegression = api.createLinearRegression(
        "dataset/5143a51a37203f2cf7000972", args);
```

In this example, we created a linear regression named `my linear regression` and set the objective field to be `my_objective_field`. Other arguments, like `bias`, can also be specified as attributes in arguments dictionary at creation time. Particularly for categorical fields, there are three different available ‘field_codings`options (contrast`, `other` or the `dummy` default coding). For a more detailed description of the` field_codings‘‘ attribute and its syntax, please see the Developers API Documentation.

Logistic regressions

For classification problems, you can choose also logistic regressions to model your data. Logistic regressions compute a probability associated to each class in the objective field. The probability is obtained using a logistic function, whose argument is a linear combination of the field values.

As the rest of models, logistic regressions can be created from a dataset by calling the corresponding create method:

```
JSONObject args = JSONValue.parseValue(
    "{\"name\": \"my logistic regression\",
      \"objective_field\": \"my_objective_field\"}");
JSONObject logisticRegression = api.createLogisticRegression(
    "dataset/5143a51a37203f2cf7000972", args);
```

In this example, we created a logistic regression named `my logistic regression` and set the objective field to be `my_objective_field`. Other arguments, like `bias`, `missing_numerics` and `c` can also be specified as attributes in arguments dictionary at creation time. Particularly for categorical fields, there are four different available ‘field_codings`options (`dummy`,`contrast`,`other`or the`one-hot`default coding). For a more detailed description of the`field_codings‘‘ attribute and its syntax, please see the Developers API Documentation.

Deepnets

Deepnets can also solve classification and regression problems. Deepnets are an optimized version of Deep Neural Networks, a class of machine-learned models inspired by the neural circuitry of the human brain. In these classifiers, the input features are fed to a group of “nodes” called a “layer”. Each node is essentially a function on the input that transforms the input features into another value or collection of values. Then the entire layer transforms an input vector into a new “intermediate” feature vector. This new vector is fed as input to another layer of nodes. This process continues layer by layer, until we reach the final “output” layer of nodes, where the output is the network’s prediction: an array of per-class probabilities for classification problems or a single, real value for regression problems.

Deepnets predictions compute a probability associated to each class in the objective field for classification problems. As the rest of models, deepnets can be created from a dataset by calling the corresponding create method:

```
JSONObject args = JSONValue.parseValue(
    "{\"name\": \"my deepnet\",
      \"objective_field\": \"my_objective_field\"}");
JSONObject deepnet = api.createDeepnet
    "dataset/5143a51a37203f2cf7000972", args);
```

In this example, we created a deepnet named `my deepnet` and set the objective field to be `my_objective_field`. Other arguments, like `number_of_hidden_layers`, `learning_rate` and `missing_numerics` can also be specified as attributes in an arguments dictionary at creation time. For a more detailed description of the available attributes and its syntax, please see the Developers API Documentation.

Batch predictions

We have shown how to create predictions individually, but when the amount of predictions to make increases, this procedure is far from optimal. In this case, the more efficient way of predicting remotely is to create a dataset containing the input data you want your model to predict from and to give its id and the one of the model to the `createBatchPrediction` api call:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my batch prediction\",
          \"all_fields\": true,
          \"header\": true,
          \"confidence\": true}");
    JSONObject batchPrediction = api.createBatchPrediction(
        "model/5af06df94e17277501000010",
        "dataset/5143a51a37203f2cf7000972",
        args);
```

In this example, setting `all_fields` to true causes the input data to be included in the prediction output, `header` controls whether a headers line is included in the file or not and `confidence` set to true causes the confidence of the prediction to be appended. If none of these arguments is given, the resulting file will contain the name of the objective field as a header row followed by the predictions.

As for the rest of resources, the create method will return an incomplete object, that can be updated by issuing the corresponding `api.getBatchPrediction` call until it reaches a `FINISHED` status. Then you can download the created predictions file using:

```
api.downloadBatchPrediction(
    "batchprediction/526fc344035d071ea3031d70",
    "my_dir/my_predictions.csv");
```

that will copy the output predictions to the local file given in the second param.

The output of a batch prediction can also be transformed to a source object using the `createSourceFromBatchPrediction` method in the api:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my_batch_prediction_source\"}");
    api.createSourceFromBatchPrediction(
        "batchprediction/526fc344035d071ea3031d70", null, args);
```

This code will create a new source object, that can be used again as starting point to generate datasets.

Batch centroids

As described in the previous section, it is also possible to make centroids' predictions in batch. First you create a dataset containing the input data you want your cluster to relate to a centroid. The `createBatchCentroid` call will need the id of the input data dataset and the cluster used to assign a centroid to each instance:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my batch centroid\",
          \"all_fields\": true,
          \"header\": true}");
    JSONObject batchCentroid = api.createBatchCrediction(
        "cluster/5af06df94e17277501000010",
        "dataset/5143a51a37203f2cf7000972",
        args);
```

Batch anomaly scores

Input data can also be assigned an anomaly score in batch. You train an anomaly detector with your training data and then build a dataset from your input data. The `createBatchAnomalyScore` call will need the id of the dataset and of the anomaly detector to assign an anomaly score to each input data instance:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my batch anomaly score\",
          \"all_fields\": true,
          \"header\": true}");
    JSONObject batchAnomalyScore = api.createBatchAnomalyScore(
        "anomaly/5af06df94e17277501000010",
        "dataset/5143a51a37203f2cf7000972",
        args);
```

Batch topic distributions

Input data can also be assigned a topic distribution in batch. You train a topic model with your training data and then build a dataset from your input data. The `createBatchTopicDistribution` call will need the id of the dataset and of the topic model to assign a topic distribution to each input data instance:

```
    JSONObject args = JSONValue.parseValue(
        "{\"name\": \"my batch topic distribution\",
          \"all_fields\": true,
          \"header\": true}");
    JSONObject batchTopicDistribution = api.createBatchTopicDistribution(
        "topicmodel/58362aaa983efc45a1000007",
        "dataset/5143a51a37203f2cf7000972",
        args);
```

1.4.2 Reading Resources

When retrieved individually, resources are returned as a dictionary identical to the one you get when you create a new resource. However, the status code will be `HTTP_OK` if the resource can be retrieved without problems, or one of the HTTP standard error codes otherwise.

1.4.3 Listing Resources

You can list resources with the appropriate api method:

```
api.listSources(null);
api.listDatasets(null);
api.listModels(null);
api.listPredictions(null);
api.listEvaluations(null);
api.listEnsembles(null);
api.listBatchPredictions(null);
api.listClusters(null);
api.listCentroids(null);
api.listBatchCentroids(null);
api.listAnomalies(null);
api.listAnomalyScores(null);
api.listBatchAnomalyScores(null);
```

(continues on next page)

(continued from previous page)

```
api.listProjects(null);
api.listSamples(null);
api.listCorrelations(null);
api.listStatisticalTests(null);
api.listLogisticRegressions(null);
api.listLinearRegressions(null);
api.listAssociations(null);
api.listAssociationSets(null);
api.listTopicModels(null);
api.listTopicDistributions(null);
api.listBatchTopicDistributions(null);
api.listTimeSeries(null);
api.listForecasts(null);
api.listDeepnets(null);
api.listScripts(null);
api.listLibraries(null);
api.listExecutions(null);
api.listExternalConnectors();
```

you will receive a dictionary with the following keys:

- **code**: If the request is successful you will get a `HTTP_OK` (200) status code. Otherwise, it will be one of the standard HTTP error codes. See BigML documentation on status codes for more info.
- **meta**: A dictionary including the following keys that can help you paginate listings:
  - **previous**: Path to get the previous page or `None` if there is no previous page.
  - **next**: Path to get the next page or `None` if there is no next page.
  - **offset**: How far off from the first entry in the resources is the first one listed in the resources key.
  - **limit**: Maximum number of resources that you will get listed in the resources key.
  - **total_count**: The total number of resources in BigML.
- **objects**: A list of resources as returned by BigML.
- **error**: If an error occurs and the resource cannot be created, it will contain an additional code and a description of the error. In this case, **meta**, and **resources** will be `None`.

1.4.4 Filtering resources

In order to filter resources you can use any of the properties labeled as *filterable* in the BigML documentation. Please, check the available properties for each kind of resource in their particular section. In addition to specific selectors you can use two general selectors to paginate the resources list: `offset` and `limit`. For details, please check this requests section.

A few examples:

First 5 sources created before April 1st, 2012 ^^

```
api.listSources("limit=5;created__lt=2012-04-1");
```

First 10 datasets bigger than 1MB ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```
api.listDatasets("limit=10;size__gt=1048576");
```

Models with more than 5 fields (columns) ^^

```
api.listModels("columns__gt=5");
```

Predictions whose model has not been deleted ^^

```
api.listPredictions("model_status=true");
```

1.4.5 Ordering Resources

In order to order resources you can use any of the properties labeled as *sortable* in the BigML documentation. Please, check the sortable properties for each kind of resources in their particular section. By default BigML paginates the results in groups of 20, so it's possible that you need to specify the `offset` or increase the `limit` of resources to returned in the list call. For details, please, check this requests section.

A few examples:

Sources ordered by size ^^^^^^^^^^^^^^^^^^^^^^^^

```
api.listSources("order_by=size");
```

Datasets created before April 1st, 2012 ordered by size ^^

```
api.listDatasets("created__lt=2012-04-1;order_by=size");
```

Models ordered by number of predictions (in descending order). ^^

```
api.listModels("order_by=-number_of_predictions");
```

Predictions ordered by name. ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```
api.listPredictions("order_by=name");
```

1.4.6 Updating Resources

When you update a resource, it is returned in a dictionary exactly like the one you get when you create a new one. However the status code will be `HTTP_ACCEPTED` if the resource can be updated without problems or one of the HTTP standard error codes otherwise.

```
    JSONObjects args = new JSONObject();
    args.put("name", "new name");

    api.updateSource(source, args);
    api.updateDataset(dataset, args);
    api.updateModel(model, args);
    api.updatePrediction(prediction, args);
    api.updateEvaluation(evaluation, args);
    api.updateEnsemble(ensemble, args);
    api.updateBatchPrediction(batchPrediction, args);
    api.updateCluster(cluster, args);
    api.updateCentroid(centroid, args);
    api.updateBatchCentroid(batchCentroid, args);
    api.updateAnomaly(anomaly, args);
    api.updateAnomalyScore(anomalyScore, args);
    api.updateBatchAnomalyScore(batchAnomalyScore, args);
    api.updateProject(project, args);
```

(continues on next page)

(continued from previous page)

```
    api.updateCorrelation(correlation, args);
    api.updateStatisticalTest(statisticalTest, args);
    api.updateLogisticRegression(logisticRegression, args);
    api.updateLinearcRegression(linearRegression, args);
    api.updateAssociation(association, args);
    api.updateAssociationSet(associationSet, args);
    api.updateTopicModel(topicModel, args);
    api.updateTopicDistribution(topicDistribution, args);
    api.updateBatchTopicDistribution(batchTopicDistribution, args);
    api.updateTimeSeries(timeSeries, args);
    api.updateForecast(forecast, args);
    api.updateDeepnet(deepnet, args);
    api.updateScript(script, args);
    api.updateLibrary(library, args);
    api.updateExecution(execution, args);
    api.updateExternalConnector(externalConnector, args)
```

Updates can change resource general properties, such as the `name` or `description` attributes of a dataset, or specific properties, like the `missing tokens` (strings considered as missing values). As an example, let's say that your source has a certain field whose contents are numeric integers. BigML will assign a numeric type to the field, but you might want it to be used as a categorical field. You could change its type to `categorical` by calling:

```
    JSONObject args = JSONValue.parseValue(
        "{\"fields\": {\"000001\": {\"optype\": \"categorical\"}}}");
    api.updateSource(source, args);
```

where `000001` is the field id that corresponds to the updated field.

Another usually needed update is changing a fields' `non-preferred` attribute, so that it can be used in the modeling process:

```
    JSONObject args = JSONValue.parseValue(
        "{\"fields\": {\"000001\": {\"preferred\": true}}}");
    api.updateDataset(dataset, args);
```

where you would be setting as `preferred` the field whose id is `000001`.

You may also want to change the name of one of the clusters found in your clustering:

```
    JSONObject args = JSONValue.parseValue(
        "{\"clusters\": {\"000001\": {\"name\": \"my cluster\"}}}");
    api.updateCluster(cluster, args);
```

which is changing the name of the cluster whose centroid id is `000001` to `my_cluster`. Or, similarly, changing the name of one detected topic:

```
    JSONObject args = JSONValue.parseValue(
        "{\"topics\": {\"000001\": {\"name\": \"my topic\"}}}");
    api.updateTopicModel(topicModel, args);
```

You will find detailed information about the updatable attributes of each resource in BigML developer's documentation.

1.4.7 Deleting Resources

Resources can be deleted individually using the corresponding method for each type of resource.

```
    api.deleteSource(source);
    api.deleteDataset(dataset);
    api.deleteModel(model);
    api.deletePrediction(prediction);
    api.deleteEvaluation(evaluation);
    api.deleteEnsemble(ensemble);
    api.deleteBatchPrediction(batchPrediction);
    api.deleteCluster(cluster);
    api.deleteCentroid(centroid);
    api.deleteBatchCentroid(batchCentroid);
    api.deleteAnomaly(anomaly);
    api.deleteAnomalyScore(anomalyScore);
    api.deleteBatchAnomalyScore(batchAnomalyScore);
    api.deleteSample(sample);
    api.deleteCorrelation(correlation);
    api.deleteStatisticalTest(statisticalTest);
    api.deleteLogisticRegression(logisticRegression);
    api.deleteLinearRegression(linearRegression);
    api.deleteAssociation(association);
    api.deleteAssociationSet(associationSet);
    api.deleteTopicModel(topicModel);
    api.deleteTopicDistribution(topicDistribution);
    api.deleteBatchTopicDistribution(batchTopicDistribution);
    api.deleteTimeSeries(timeSeries);
    api.deleteForecast(forecast);
    api.deleteDeepnet(deepnet);
    api.deleteProject(project);
    api.deleteScript(script);
    api.deleteLibrary(library);
    api.deleteExecution(execution);
    api.deleteExternalConnector(externalConnector)
```

Each of the calls above will return a dictionary with the following keys:

- **code** If the request is successful, the code will be a `HTTP_NO_CONTENT` (204) status code. Otherwise, it wil be one of the standard HTTP error codes. See the documentation on status codes for more info.
- **error** If the request does not succeed, it will contain a dictionary with an error code and a message. It will be `None` otherwise.

1.4.8 Public and shared resources

The previous examples use resources that were created by the same user that asks for their retrieval or modification. If a user wants to share one of her resources, she can make them public or share them. Declaring a resource public means that anyone can see the resource. This can be applied to datasets and models. To turn a dataset public, just update its `private` property:

```
    JSONObject args = JSONValue.parseValue(
        "{\"private\": false}");
    api.updateDataset("dataset/5143a51a37203f2cf7000972", args);
```

and any user will be able to download it using its id prepended by `public`:

```
api.getDataset("public/dataset/5143a51a37203f2cf7000972");
```

In the models' case, you can also choose if you want the model to be fully downloadable or just accesible to make predictions. This is controlled with the `white_box` property. If you want to publish your model completely, just use:

```
JSONObject args = JSONValue.parseValue(
    "{\"private\": false, \"white_box\": true}");
api.updateModel("model/5143a51a37203f2cf7000956"'", args);
```

Both public models and datasets, will be openly accessible for anyone, registered or not, from the web gallery.

Still, you may want to share your models with other users, but without making them public for everyone. This can be achieved by setting the `shared` property:

```
    JSONObject args = JSONValue.parseValue(
        "{\"shared\": true}");
    api.updateModel("model/5143a51a37203f2cf7000956", args);
```

Shared models can be accessed using their share hash (propery `shared_hash` in the original model):

```
api.getModel("shared/model/d53iw39euTdjsgesj7382ufhwnD");
```

1.5 Whizzml Resources

Whizzml is a Domain Specific Language that allows the definition and execution of ML-centric workflows. Its objective is allowing BigML users to define their own composite tasks, using as building blocks the basic resources provided by BigML itself. Using Whizzml they can be glued together using a higher order, functional, Turing-complete language. The Whizzml code can be stored and executed in BigML using three kinds of resources: `Scripts`, `Libraries` and `Executions`.

Whizzml `Scripts` can be executed in BigML's servers, that is, in a controlled, fully-scalable environment which takes care of their parallelization and fail-safe operation. Each execution uses an `Execution` resource to store the arguments and results of the process. Whizzml `Libraries` store generic code to be shared of reused in other Whizzml `Scripts`.

1.5.1 Scripts

In BigML a `Script` resource stores Whizzml source code, and the results of its compilation. Once a Whizzml script is created, it's automatically compiled; if compilation succeeds, the script can be run, that is, used as the input for a Whizzml execution resource.

An example of a `script` that would create a `source` in BigML using the contents of a remote file is:

```
    import org.bigml.binding.BigMLClient;

    // Create BigMLClient
    BigMLClient api = new BigMLClient();

    // creating a script directly from the source code. This script creates
    // a source uploading data from an s3 repo. You could also create a
    // a script by using as first argument the path to a .whizzml file which
    // contains your source code.
    JSONObject script = api.createScript(
            "(create-source {\"remote\" \"s3://bigml-public/csv/iris.csv\"})")

    while (!api.scriptIsReady(script))
        Thread.sleep(1000);

    JSONObject object = (JSONObject) Utils.getJSONObject(script, "object");
```

script `object` object:

```
{
    "approval_status": 0,
    "category": 0,
    "code": 200,
    "created": "2016-05-18T16:54:05.666000",
    "description": "",
    "imports": [],
    "inputs": None,
    "line_count": 1,
    "locale": "en-US",
    "name": "Script",
    "number_of_executions": 0,
    "outputs": None,
    "price": 0.0,
    "private": True,
    "project": None,
    "provider": None,
    "resource": "script/573c9e2db85eee23cd000489",
    "shared": False,
    "size": 59,
    "source_code": "(create-source {"remote" "s3://bigml-public/csv/iris.csv"})",
    "status": {
        "code": 5,
        "elapsed": 4,
        "message": "The script has been created",
        "progress": 1.0
    },
    "subscription": True,
    "tags": [],
    "updated": "2016-05-18T16:54:05.850000",
    "white_box": False
}
```

A `script` allows to define some variables as `inputs`. In the previous example, no input has been defined, but we could modify our code to allow the user to set the remote file name as input:

```
    import org.bigml.binding.BigMLClient;

    // Create BigMLClient
    BigMLClient api = new BigMLClient();

    JSONArray inputsList = JSONValue.parse(
        "[{"name": "my_remote_data",
           "type": "string",
           "default": "s3://bigml-public/csv/iris.csv",
           "description": "Location of the remote data"}]"
    );
    JSONObject inputs = new JSONObject();
    inputs.put("inputs", inputsList);

    JSONObject script = api.createScript(
            "(create-source {\"remote\" my_remote_data})",
            inputs)

    while (!api.sctiptIsReady(source))
        Thread.sleep(1000);
```

The `script` can also use a `library` resource (please, see the `Libraries` section below for more details) by including its id in the `imports` attribute. Other attributes can be checked at the API Developers documentation for Scripts.

1.5.2 Executions

To execute in BigML a compiled Whizzml `script` you need to create an `execution` resource. It's also possible to execute a pipeline of many compiled scripts in one request.

Each `execution` is run under its associated user credentials and its particular environment constaints. As `scripts` can be shared, you can execute the same `script` several times under different usernames by creating different `executions`.

As an example of `execution` resource, let's create one for the script in the previous section:

```
    import org.bigml.binding.BigMLClient;

    // Create BigMLClient
    BigMLClient api = new BigMLClient();

    JSONObject execution = api.createExecution("script/573c9e2db85eee23cd000489");

    while (!api.executionIsReady(execution))
        Thread.sleep(1000);

    JSONObject object = (JSONObject) Utils.getJSONObject(execution, "object");
```

execution `object` object:

```
{
    "category": 0,
    "code": 200,
    "created": "2016-05-18T16:58:01.613000",
    "creation_defaults": {    },
    "description": "",
    "execution": {
        "output_resources": [
          {
             "code": 1,
             "id": "source/573c9f19b85eee23c600024a",
             "last_update": 1463590681854,
             "progress": 0.0,
             "state": "queued",
             "task": "Queuing job",
             "variable": ""
          }
        ],
        "outputs": [],
        "result": "source/573c9f19b85eee23c600024a",
        "results": ["source/573c9f19b85eee23c600024a"],
        "sources": [["script/573c9e2db85eee23cd000489", ""]],
        "steps": 16
    },
    "inputs": None,
    "locale": "en-US",
    "name": u"Script"s Execution",
    "project": None,
```

(continues on next page)

(continued from previous page)

```
    "resource": "execution/573c9f19b85eee23bd000125",
    "script": "script/573c9e2db85eee23cd000489",
    "script_status": True,
    "shared": False,
    "status": {
        "code": 5,
        "elapsed": 249,
        "elapsed_times": {
            "in-progress": 247,
            "queued": 62,
            "started": 2
        },
        "message": "The execution has been created",
        "progress": 1.0
    },
    "subscription": True,
    "tags": [],
    "updated": "2016-05-18T16:58:02.035000"
}
```

An `execution` receives inputs, the ones defined in the `script` chosen to be executed, and generates a result. It can also generate outputs. As you can see, the execution resource contains information about the result of the execution, the resources that have been generated while executing and users can define some variables in the code to be exported as outputs. Please refer to the Developers documentation for Executions for details on how to define execution outputs.

1.5.3 Libraries

The `library` resource in BigML stores a special kind of compiled Whizzml source code that only defines functions and constants. The `library` is intended as an import for executable scripts. Thus, a compiled library cannot be executed, just used as an import in other `libraries` and `scripts` (which then have access to all identifiers defined in the `library`).

As an example, we build a `library` to store the definition of two functions: `mu` and `g`. The first one adds one to the value set as argument and the second one adds two variables and increments the result by one.

```
    import org.bigml.binding.BigMLClient;

    // Create BigMLClient
    BigMLClient api = new BigMLClient();

    JSONObject library = api.createLibrary(
        "(define (mu x) (+ x 1)) (define (g z y) (mu (+ y z)))");

    while (!api.libraryIsReady(library))
        Thread.sleep(1000);

    JSONObject object = (JSONObject) Utils.getJSONObject(library, "object");
```

library `object` object:

```
{
    "approval_status": 0,
    "category": 0,
    "code": 200,
    "created": "2016-05-18T18:58:50.838000",
```

(continues on next page)

(continued from previous page)

```
    "description": "",
    "exports": [
        {"name": "m", "signature": ["x"]},
        {"name": "g", "signature": ["z", "y"]}
    ],
    "imports": [],
    "line_count": 1,
    "name": "Library",
    "price": 0.0,
    "private": True,
    "project": None,
    "provider": None,
    "resource": "library/573cbb6ab85eee23c300018e",
    "shared": False,
    "size": 53,
    "source_code": "(define (mu x) (+ x 1)) (define (g z y) (mu (+ y z)))",
    "status": {
        "code": 5,
        "elapsed": 2,
        "message": "The library has been created",
        "progress": 1.0
    },
    "subscription": True,
    "tags": [],
    "updated": "2016-05-18T18:58:52.432000",
    "white_box": False
}
```

Libraries can be imported in scripts. The `imports` attribute of a `script` can contain a list of `library` IDs whose defined functions and constants will be ready to be used throughout the `script`. Please, refer to the API Developers documentation for Libraries for more details.

1.6 Local Resources

All the resources in BigML can be saved in json format and used locally with no connection whatsoever to BigML's servers. This is specially important for all Supervised and Unsupervised models, that can be used to generate predictions in any programmable device. The next sections describe how to do that for each type of resource.

This json can be used just as the remote model to generate predictions. As you'll see in next section, the local `Model` object can be instantiated by giving json as first argument:

```
    import org.bigml.binding.BigMLClient;
    import org.bigml.binding.LocalPredictiveModel;

    // Create BigMLClient with the properties in binding.properties
    BigMLClient api = new BigMLClient();

    // Get remote model
    JSONObject model = api.getModel("model/502fdbff15526876610002615");

    // Create local model
    LocalPredictiveModel localModel = new LocalPredictiVeModel(model);

    // Predict
```

(continues on next page)

(continued from previous page)

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    localModel.predict(inputData);
```

1.6.1 Local Models

You can instantiate a local version of a remote model.

```
    import org.bigml.binding.BigMLClient;
    import org.bigml.binding.LocalPredictiveModel;

    BigMLClient api = new BigMLClient();

    // Get remote model
    JSONObject model = api.getModel("model/502fdbff15526876610002615");

    // Create local model
    LocalPredictiveModel localModel = new LocalPredictiVeModel(model);
```

This will retrieve the remote model information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a Model object that you can use to make local predictions.

Local Predictions

Once you have a local model you can use to generate predictions locally.

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    localModel.predict(inputData);
```

Local predictions have three clear advantages:

- Removing the dependency from BigML to make new predictions.
- No cost (i.e., you do not spend BigML credits).
- Extremely low latency to generate predictions for huge volumes of data.

The default output for local predictions is the prediction itself, but you can also add other properties associated to the prediction, like its confidence or probability, the distribution of values in the predicted node (for decision tree models), and the number of instances supporting the prediction. To obtain a dictionary with the prediction and the available additional properties use the `full=True` argument:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    localModel.predict(inputData, null, null, null, true);
```

that will return:

```
{
    "count": 47,
    "confidence": 0.92444,
    "probability": 0.9861111111111112,
    "prediction": "Iris-versicolor",
    "distribution_unit": "categories",
    "path": ["petal length > 2.45",
             "petal width <= 1.75",
             "petal length <= 4.95",
             "petal width <= 1.65"],
    "distribution": [["Iris-versicolor", 47]]
}
```

Note that the path attribute for the proportional missing strategy shows the path leading to a final unique node, that gives the prediction, or to the first split where a missing value is found. Other optional attributes are next which contains the field that determines the next split after the prediction node and distribution that adds the distribution that leads to the prediction. For regression models, min and max will add the limit values for the data that supports the prediction.

When your test data has missing values, you can choose between last prediction or proportional strategy to compute the prediction. The last prediction strategy is the one used by default. To compute a prediction, the algorithm goes down the model's decision tree and checks the condition it finds at each node (e.g.: 'sepal length' > 2). If the field checked is missing in your input data you have two options: by default (last prediction strategy) the algorithm will stop and issue the last prediction it computed in the previous node. If you chose proportional strategy instead, the algorithm will continue to go down the tree considering both branches from that node on. Thus, it will store a list of possible predictions from then on, one per valid node. In this case, the final prediction will be the majority (for categorical models) or the average (for regressions) of values predicted by the list of predicted values.

You can set this strategy by using the missingStrategy argument with code 0 to use last prediction and 1 for proportional.

```
    import org.bigml.binding.MissingStrategy;

    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    localModel.predict(
        inputData, MissingStrategy.PROPORTIONAL, null, null, true);
```

For classification models, it is sometimes useful to obtain a probability or confidence prediction for each possible class of the objective field. To do this, you can use the predictProbability and predictConfidence methods respectively. The former gives a prediction based on the distribution of instances at the appropriate leaf node, with a Laplace correction based on the root node distribution. The latter returns a lower confidence bound on the leaf node probability based on the Wilson score interval.

Each of these methods take the missingStrategy argument that functions as it does in predict. Note that these methods substitute the deprecated multiple parameter in the predict method functionallity.

So, for example, the following:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3}");
    localModel.predictProbability(inputData);
```

would result in

```
    [{"prediction": "Iris-setosa",
      "probability": 0.0033003300330033},
```

(continues on next page)

(continued from previous page)

```
    {"prediction": "Iris-versicolor",
     "probability": 0.4983498349834984},
    {"prediction": "Iris-virginica",
     "probability": 0.4983498349834984}]
```

The output of `predictConfidence` is the same, except that the output maps are keyed with `confidence` instead of `probability`.

For classifications, the prediction of a local model will be one of the available categories in the objective field and an associated `confidence` or `probability` that is used to decide which is the predicted category. If you prefer the model predictions to be operated using any of them, you can use the `operatingKind` argument in the `predict` method. Here's the example to use predictions based on `confidence`:

```
JSONObject inputData = JSONValue.parseValue(
    "{\"petal length\": 3, \"petal width\": 1}");
localModel.predict(inputData, null, null, "confidence", true, null);
```

Previous versions of the bindings had additional arguments in the `predict` method that were used to format the prediction attributes. The signature of the method has been changed to accept only arguments that affect the prediction itself, (like `missingStrategy`, `operatingKind` and `opreatingPoint`) and `full` which is a boolean that controls whether the output is the prediction itself or a dictionary will all the available properties associated to the prediction.

```
public Prediction predict(
        JSONObject inputData, MissingStrategy missingStrategy,
        JSONObject operatingPoint, String operatingKind, Boolean full,
        List<String> unusedFields) throws Exception {
    ...
}
```

Operating point's predictions

In classification problems, Models, Ensembles and Logistic Regressions can be used at different operating points, that is, associated to particular thresholds. Each operating point is then defined by the kind of property you use as threshold, its value and a the class that is supposed to be predicted if the threshold is reached.

Let's assume you decide that you have a binary problem, with classes `True` and `False` as possible outcomes. Imagine you want to be very sure to predict the `True` outcome, so you don't want to predict that unless the probability associated to it is over `0,8`. You can achieve this with any classification model by creating an operating point:

```
JSONObject operatingPoint = JSONValue.parseValue(
    "{\"kind length\": \"probability\",
      \"positive_class width\": \"True\",
      \"threshold\": 0.8}");
```

to predict using this restriction, you can use the `operatingPoint` parameter:

```
Prediction prediction = localModel.predict(
    inputData, null, operatingPoint, nul, true, null);
```

where `inputData` should contain the values for which you want to predict. Local models allow two kinds of operating points: `probability` and `confidence`. For both of them, the threshold can be set to any number in the `[0, 1]` range.

1.6.2 Local Clusters

You can instantiate a local version of a remote cluster.

```
import org.bigml.binding.BigMLClient;
import org.bigml.binding.LocalCluster;

BigMLClient api = new BigMLClient();

// Get remote cluster
JSONObject cluster = api.getCluster("cluster/502fdbff15526876610002435");

// Create local cluster
LocalCluster localCluster = new LocalCluster(cluster);
```

This will retrieve the remote cluster information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a `LocalCluster` object that you can use to make local centroid predictions.

Local clusters provide also methods for the significant operations that can be done using clusters: finding the centroid assigned to a certain data point, sorting centroids according to their distance to a data point, summarizing the centroids intra-distances and inter-distances and also finding the closest points to a given one. The *Local Centroids* and the *Summary generation* sections will explain these methods.

Local Centroids

Using the local cluster object, you can predict the centroid associated to an input data set:

```
JSONObject inputData = JSONValue.parseValue(
    "{\"pregnancies\": 0, \"plasma glucose\": 118,
      \"blood pressure\": 84, \"triceps skin thickness\": 47,
      \"insulin\": 230, \"bmi\": 45.8,
      \"diabetes pedigree\": 0.551, \"age\": 31,
      \"diabetes\": \"true\"}");
JSONObject centroid = localCluster.centroid(inputData);
```

that will return:

```
{
    "distance": 0.454110207355,
    "centroid_name": "Cluster 4",
    "centroid_id": "000004"
}
```

You must keep in mind, though, that to obtain a centroid prediction, input data must have values for all the numeric fields. No missing values for the numeric fields are allowed unless you provided a `default_numeric_value` in the cluster construction configuration. If so, this value will be used to fill the missing numeric fields.

As in the local model predictions, producing local centroids can be done independently of BigML servers, so no cost or connection latencies are involved.

Another interesting method in the cluster object is `localCluster.closestInCluster`, which given a reference data point will provide the rest of points that fall into the same cluster sorted in an ascending order according to their distance to this point. You can limit the maximum number of points returned by setting the `numberOfPoints` argument to any positive integer.

```
    JSONObject referencePoint = JSONValue.parseValue(
        "{\"pregnancies\": 0, \"plasma glucose\": 118,
          \"blood pressure\": 84, \"triceps skin thickness\": 47,
          \"insulin\": 230, \"bmi\": 45.8,
          \"diabetes pedigree\": 0.551, \"age\": 31,
          \"diabetes\": \"true\"}");
    JSONObject point = localCluster.closestInCluster(inputData, 2, null);
```

The response will be a dictionary (JSONObject) with the centroid id of the cluster an the list of closest points and their distances to the reference point.

```
  {
    "closest": [
        {"distance": 0.06912270988567025,
           "data": {"plasma glucose": "115", "blood pressure": "70",
                    "triceps skin thickness": "30", "pregnancies": "1",
                    "bmi": "34.6", "diabetes pedigree": "0.529",
                    "insulin": "96", "age": "32", "diabetes": "true"}
        },
        {"distance": 0.10396456577958413,
           "data": {"plasma glucose": "167", "blood pressure": "74",
           "triceps skin thickness": "17", "pregnancies": "1", "bmi": "23.4",
           "diabetes pedigree": "0.447", "insulin": "144", "age": "33",
           "diabetes": "true"}
        }
    ],
    "reference": {
      "age": 31, "bmi": 45.8, "plasma glucose": 118,
      "insulin": 230, "blood pressure": 84,
      "pregnancies": 0, "triceps skin thickness": 47,
      "diabetes pedigree": 0.551, "diabetes": "true"},
    "centroid_id": "000000"
  }
```

No missing numeric values are allowed either in the reference data point. If you want the data points to belong to a different cluster, you can provide the `centroid_id` for the cluster as an additional argument.

Other utility methods are `local_cluster.sortedCentroids` which given a reference data point will provide the list of centroids sorted according to the distance to it

```
        "{\"pregnancies\": 1, \"plasma glucose\": 115,
          \"blood pressure\": 70, \"triceps skin thickness\": 30,
          \"insulin\": 96, \"bmi\": 34.6,
          \"diabetes pedigree\": 0.529, \"age\": 32,
          \"diabetes\": \"true\"}");
    JSONObject sortedCentroids = localCluster.sortedCentroids(
      inputData, 2, null);
```

that will return:

```
  {
      "centroids": [{"distance": 0.31656890408929705,
                      "data": {"000006": 0.34571, "000007": 30.7619,
                               "000000": 3.79592, "000008": "false"},
                      "centroid_id": "000000"},
                     {"distance": 0.4424198506958207,
                      "data": {"000006": 0.77087, "000007": 45.50943,
```

(continues on next page)

(continued from previous page)

```
                               "000000": 5.90566, "000008": "true"},
                      "centroid_id": "000001"}],
      "reference": {"age": "32", "bmi": "34.6", "plasma glucose": "115",
                     "insulin": "96", "blood pressure": "70",
                     "pregnancies": "1", "triceps skin thickness": "30",
                     "diabetes pedigree": "0.529", "diabetes": "true"}
  }
```

or `pointsInCluster` that returns the list of data points assigned to a certain cluster, given its `centroid_id`.

```
    JSONObject points = localCluster.pointsInCluster("000000");
```

1.6.3 Local AnomalyDetector

You can also instantiate a local version of a remote anomaly.

```
    import org.bigml.binding.BigMLClient;
    import org.bigml.binding.LocalAnomaly;

    BigMLClient api = new BigMLClient();

    // Get remote anomaly
    JSONObject anomaly = api.getAnomalyDetector(
        "anomaly/502fcbff15526876610002435");

    // Create local anomaly detector
    LocalAnomaly localAnomaly = new LocalAnomaly(anomaly);
```

This will retrieve the remote anomaly detector information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return an `LocalAnomaly` object that you can use to make local anomaly scores.

The anomaly detector object has also the method `filter` that will build the LISP filter you would need to filter the original dataset and create a new one excluding the top anomalies. Setting the `include` parameter to True you can do the inverse and create a dataset with only the most anomalous data points.

Local Anomaly Scores

Using the local anomaly detector object, you can predict the anomaly score associated to an input data set:

```
    JSONObject inputData = JSONValue.parseValue("{\"src_bytes\": 350}");
    double score = localAnomaly.score(inputData);

    0.9268527808726705
```

As in the local model predictions, producing local anomaly scores can be done independently of BigML servers, so no cost or connection latencies are involved.

1.6.4 Local Logistic Regression

You can also instantiate a local version of a remote logistic regression.

```
    import org.bigml.binding.BigMLClient;
    import org.bigml.binding.LocalLogisticRegression;

    BigMLClient api = new BigMLClient();

    // Get remote logistic regression
    JSONObject logistic = api.getLogisticRegression(
        "logisticregression/502fdbff15526876610042435");

    // Create local logistic regression
    LocalLogisticRegression localLogisticRegression =
        new LocalLogisticRegression(logistic);
```

This will retrieve the remote logistic regression information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a `LocalLogisticRegression` object that you can use to make local predictions.

Local Logistic Regression Predictions

Using the local logistic regression object, you can predict the prediction for an input data set:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 2, \"sepal length\": 1.5,
          \"petal width\": 0.5, \"sepal width\": 0.7}");
    localLogisticRegression.predict(inputData, null, null, true);
```

that will return:

```
  {
      "distribution": [
          {"category": "Iris-virginica", "probability": 0.5041444478857267},
          {"category": "Iris-versicolor", "probability": 0.46926542042788333},
          {"category": "Iris-setosa", "probability": 0.02659013168639014}
      ],
      "prediction": "Iris-virginica",
      "probability": 0.5041444478857267
  }
```

As you can see, the prediction contains the predicted category and the associated probability. It also shows the distribution of probabilities for all the possible categories in the objective field.

You must keep in mind, though, that to obtain a logistic regression prediction, input data must have values for all the numeric fields. No missing values for the numeric fields are allowed.

For consistency of interface with the `LocalPredictiveModelModel` class, logistic regressions again have a `predictProbability` method. As stated above, missing values are not allowed, and so there is no `missingStrategy` argument.

Operating point predictions are also available for local logistic regressions and an example of it would be:

```
    JSONObject operatingPoint = JSONValue.parseValue(
        "{\"kind length\": \"probability\",
          \"positive_class width\": \"True\",
          \"threshold\": 0.8}");
    localLogisticRegression.predict(inputData, operatingPoint, null, true);
```

You can check the *Operating point's predictions* section to learn about operating points. For logistic regressions, the only available kind is `probability`, that sets the threshold of probability to be reached for the prediction to be the positive class.

1.6.5 Local Linear Regression

You can also instantiate a local version of a remote linear regression.

```
import org.bigml.binding.BigMLClient;
import org.bigml.binding.LocalinearRegression;

BigMLClient api = new BigMLClient();

// Get remote linear regression
JSONObject linear = api.getLinearRegression(
    "linearregression/502fdbff15526876610042435");

// Create local linear regression
LocalLinearRegression localLinearRegression =
    new LocalLinearRegression(linear);
```

This will retrieve the remote logistic regression information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a `LocalLinearRegression` object that you can use to make local predictions.

Local Linear Regression Predictions

Using the local linear regression object, you can predict the prediction for an input data set:

```
JSONObject inputData = JSONValue.parseValue(
    "{\"petal length\": 2, \"sepal length\": 1.5,
      \"petal width\": 0.5, \"sepal width\": 0.7}");
localLinearRegression.predict(inputData, true);
```

that will return:

```
{
        "prediction": -4.2168344
}
```

To obtain a linear regression prediction, input data can only have missing values for fields that had already some missings in training data.

1.6.6 Local Deepnet

You can also instantiate a local version of a remote Deepnet.

```
import org.bigml.binding.BigMLClient;
import org.bigml.binding.LocalDeepnet;

BigMLClient api = new BigMLClient();
```

(continues on next page)

(continued from previous page)

```
    // Get remote deepnet
    JSONObject deepnet = api.getDeepnet(
        "deepnet/502fdbff15526876610022435");

    // Create local deepnet
    LocalDeepnet localDeepnet = new LocalDeepnet(deepnet);
```

This will retrieve the remote deepnet information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a `LocalDeepnet` object that you can use to make local predictions.

Local Deepnet Predictions

Using the local deepnet object, you can predict the prediction for an input data set:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 2, \"sepal length\": 1.5,
          \"petal width\": 0.5, \"sepal width\": 0.7}");
    localDeepnet.predict(inputData, null, null, true);
```

that will return:

```
  {
      "distribution": [
        {"category": "Iris-virginica", "probability": 0.5041444478857267},
        {"category": "Iris-versicolor", "probability": 0.46926542042788333},
        {"category": "Iris-setosa", "probability": 0.02659013168639014}
      ],
      "prediction": "Iris-virginica",
      "probability": 0.5041444478857267
  }
```

As you can see, the full prediction contains the predicted category and the associated probability. It also shows the distribution of probabilities for all the possible categories in the objective field.

To be consistent with the `LocalPredictiveModelModel` class interface, deepnets have also a `predictProbability` method.

Operating point predictions are also available for local deepnets and an example of it would be:

```
    JSONObject operatingPoint = JSONValue.parseValue(
        "{\"kind length\": \"probability\",
          \"positive_class width\": \"True\",
          \"threshold\": 0.8}");
    localDeepnet.predict(inputData, operatingpoint, null, true);
```

1.6.7 Local Fusion

You can also instantiate a local version of a remote Fusion.

```
import org.bigml.binding.BigMLClient;
import org.bigml.binding.LocalFusion;

BigMLClient api = new BigMLClient();

// Get remote fusion
JSONObject fusion = api.getFusion(
    "fusion/502fdbff15526876610022438");

// Create local fusion
LocalFusion localFusion = new LocalFusion(fusion);
```

This will retrieve the remote deepnet information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a `LocalFusion` object that you can use to make local predictions.

Local Fusion Predictions

Using the local fusion object, you can predict the prediction for an input data set:

```
JSONObject inputData = JSONValue.parseValue(
    "{\"petal length\": 2, \"sepal length\": 1.5,
      \"petal width\": 0.5, \"sepal width\": 0.7}");
localFusion.predict(inputData, null, null, true);
```

that will return:

```
{
    "prediction": "Iris-setosa",
    "probability": 0.45224
}
```

As you can see, the full prediction contains the predicted category and the associated probability.

To be consistent with the `ocalPredictiveModel` class interface, fusions have also a `predict_probability` method.

Operating point predictions are also available with probability as threshold for local fusions and an example of it would be:

```
JSONObject operatingPoint = JSONValue.parseValue(
    "{\"kind length\": \"probability\",
      \"positive_class width\": \"True\",
      \"threshold\": 0.8}");
localFusion.predict(inputData, operatingpoint, null, true);
```

1.6.8 Local Association

You can also instantiate a local version of a remote association resource.

```
import org.bigml.binding.BigMLClient;
import org.bigml.binding.LocalAssociation;

```

(continues on next page)

(continued from previous page)

```
    BigMLClient api = new BigMLClient();

    // Get remote association
    JSONObject association = api.getAssociation(
        "association/502fdcff15526876610002435");

    // Create local association
    LocalAssociation localAssociation = new LocalAssociation(association);
```

This will retrieve the remote association information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return an `LocalAssociation` object that you can use to extract the rules found in the original dataset.

The created `LocalAssociation` object has some methods to help retrieving the association rules found in the original data. The `rules` method will return the association rules. Arguments can be set to filter the rules returned according to its `leverage`, `strength`, `support`, `p_value`, a list of items involved in the rule or a user-given filter function.

```
    List itemList = new ArrayList();
    itemList.add("Edible");
    localAssociation.rules(null, null, 0.3, itemList, null);
```

In this example, the only rules that will be returned by the `rules` method will be the ones that mention `Edible` and their `p_value` is greater or equal to `0.3`.

The rules can also be stored in a CSV file using `rulesCsv`:

```
    List itemList = new ArrayList();
    itemList.add("Edible");
    localAssociation.rulesCsv(
        "/tmp/my_rules.csv", null, null, 0.3, itemList, null);
```

This example will store the rules whose strength is bigger or equal to 0.1 in the `/tmp/my_rules.csv` file.

You can also obtain the list of `items` parsed in the dataset using the `items` method. You can also filter the results by field name, by item names and by a user-given function:

```
    List names = new ArrayList();
    names.add("Brown cap");
    names.add("White cap");
    names.add("Yellow cap");
    localAssociation.items("Cap Color", names, null, null);
```

This will recover the `Item` objects found in the `Cap Color` field for the names in the list, with their properties as described in the developers section.

Local Association Sets

Using the local association object, you can predict the association sets related to an input data set:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"gender\": \"Female\", \"genres\": \"Adventure$Action\",
          \"timestamp\": 993906291, \"occupation\": \"K-12 student\",
          \"zipcode\": 59583, \"rating\": 3}");
    localAssociation.associationSet(inputData, null, null);
```

that returns

```
[
    {"item": {"complement": False,
               "count": 70,
               "field_id": "000002",
               "name": "Under 18"},
      "rules": ["000000"],
      "score": 0.0969181441561211},
    {"item": {"complement": False,
               "count": 216,
               "field_id": "000007",
               "name": "Drama"},
      "score": 0.025050115102862636},
    {"item": {"complement": False,
               "count": 108,
               "field_id": "000007",
               "name": "Sci-Fi"},
      "rules": ["000003"],
      "score": 0.02384578264599424},
    {"item": {"complement": False,
               "count": 40,
               "field_id": "000002",
               "name": "56+"},
      "rules": ["000008",
                "000020"],
      "score": 0.021845366022721312},
    {"item": {"complement": False,
               "count": 66,
               "field_id": "000002",
               "name": "45-49"},
      "rules": ["00000e"],
      "score": 0.019657155185835006}
]
```

As in the local model predictions, producing local association sets can be done independently of BigML servers, so no cost or connection latencies are involved.

1.6.9 Local Topic Model

You can also instantiate a local version of a remote topic model.

```
import org.bigml.binding.BigMLClient;
import org.bigml.binding.LocalTopicModel;

BigMLClient api = new BigMLClient();

// Get remote topicModel
JSONObject topicModel = api.getTopicModel(
    "topicmodel/502fdbcf15526876210042435");

// Create local topicModel
LocalTopicModel localTopicModel = new LocalTopicModel(topicModel);
```

This will retrieve the remote topic model information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a `LocalTopicModel` object that you can use to obtain local topic distributions.

Local Topic Distributions

Using the local topic model object, you can predict the local topic distribution for an input data set:

```
JSONObject inputData = JSONValue.parseValue(
    "{\"Message\": \"Our mobile phone is free\"}");
localTopicModel.distribution(inputData);
```

that returns

```
[
    {"name": "Topic 00", "probability": 0.002627154266498529},
    {"name": "Topic 01", "probability": 0.003257671290458176},
    {"name": "Topic 02", "probability": 0.002627154266498529},
    {"name": "Topic 03", "probability": 0.1968263976460698},
    {"name": "Topic 04", "probability": 0.002627154266498529},
    {"name": "Topic 05", "probability": 0.002627154266498529},
    {"name": "Topic 06", "probability": 0.13692728036990331},
    {"name": "Topic 07", "probability": 0.6419714165615805},
    {"name": "Topic 08", "probability": 0.002627154266498529},
    {"name": "Topic 09", "probability": 0.002627154266498529},
    {"name": "Topic 10", "probability": 0.002627154266498529},
    {"name": "Topic 11", "probability": 0.002627154266498529}
]
```

As you can see, the topic distribution contains the name of the possible topics in the model and the associated probabilities.

1.6.10 Local Time Series

You can also instantiate a local version of a remote time series.

```
import org.bigml.binding.BigMLClient;
import org.bigml.binding.LocalTimeSeries;

BigMLClient api = new BigMLClient();

// Get remote timeSeries
JSONObject timeSeries = api.getTimeSeries(
    "timeseries/502fdbcf15526876210042435");

// Create local timeSeries
LocalTimeSeries localTimeSeries = new LocalTimeSeries(timeSeries);
```

This will create a series of models from the remote time series information, using an implicitly built `BigML()` connection object (see the `Authentication` section for more details on how to set your credentials) and return a `LocalTimeSeries` object that you can use to obtain local forecasts.

Local Forecasts

Using the local time series object, you can forecast any of the objective field values:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"Final\": {\"horizon\": 5},
          \"Assignment\": {\"horizon\": 10, \"ets_models\": {\"criterion\": \"aic\", \
↪"limit\": 2}}}");
    localTimeSeries.forecast(inputData);
```

that returns

```
  {
    "000005": [
        {"point_forecast": [68.53181, 68.53181, 68.53181, 68.53181, 68.53181],
         "model": "A,N,N"}],
     "000001": [{"point_forecast": [54.776650000000004, 90.00943000000001,
                                     83.59285000000001, 85.72403000000001,
                                     72.87196, 93.85872, 84.80786, 84.65522,
                                     92.52545, 88.78403],
                 "model": "A,N,A"},
                {"point_forecast": [55.882820120000005, 90.5255466567616,
                                     83.44908577909621, 87.64524353046498,
                                     74.32914583152592, 95.12372848262932,
                                     86.69298716626228, 85.31630744944385,
                                     93.62385478607113, 89.06905451921818],
                 "model": "A,Ad,A"}]
  }
```

As you can see, the forecast contains the ID of the forecasted field, the computed points and the name of the models meeting the criterion. For more details about the available parameters, please check the API documentation.

1.6.11 Multi Models

Multi Models use a numbers of BigML remote models to build a local version that can be used to generate predictions locally. Predictions are generated combining the outputs of each model.

```
    import org.bigml.binding.BigMLClient;
    import org.bigml.binding.MultiModel;

    BigMLClient api = new BigMLClient();

    JSONArray models = (JSONArray) api.listModels(
        ";tags__in=my_tag").get("objects");

    MultiModel multiModel = new MultiModel(models, null, null);
```

This will create a multi model using all the models that have been previously tagged with `my_tag` and predict by combining each model's prediction. The combination method used by default is `plurality` for categorical predictions and mean value for numerical ones. You can also use `confidence weighted`:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    multiModel.predict(inputData, null, PredictionMethod.PLURALITY, null);
```

that will weight each vote using the confidence/error given by the model to each prediction, or even `probability weighted`:

```
JSONObject inputData = JSONValue.parseValue(
    "{\"petal length\": 3, \"petal width\": 1}");
multiModel.predict(inputData, null, PredictionMethod.PROBABILITY, null);
```

that weights each vote by using the probability associated to the training distribution at the prediction node.

There's also a `threshold` method that uses an additional set of options: threshold and category. The category is predicted if and only if the number of predictions for that category is at least the threshold value. Otherwise, the prediction is plurality for the rest of predicted values.

An example of `threshold` combination method would be:

```
Map options = new HashMap();
options.put("threshold", 3);
options.put("category", "Iris-virginica");
JSONObject inputData = JSONValue.parseValue(
    "{\"petal length\": 0.9, \"petal width\": 1}");
multiModel.predict(inputData, null, PredictionMethod.THRESHOLD, options);
```

When making predictions on a test set with a large number of models, `batch_predict` can be useful to log each model's predictions in a separated file. It expects a list of input data values and the directory path to save the prediction files in.

```
JSONArray inputDataList = JSONValue.parseValue(
    "[{\"petal length\": 3, \"petal width\": 1},
      {\"petal length\": 3, \"petal width\": 5.1}]");
multiModel.batchPredict(inputDataList, "data/predictions");
```

The predictions generated for each model will be stored in an output file in `data/predictions` using the syntax `model_[id of the model]__predictions.csv`. For instance, when using `model/50c0de043b563519830001c2` to predict, the output file name will be `model_50c0de043b563519830001c2__predictions.csv`. An additional feature is that using `reuse=True` as argument will force the function to skip the creation of the file if it already exists. This can be helpful when using repeatedly a bunch of models on the same test set.

```
JSONArray inputDataList = JSONValue.parseValue(
    "[{\"petal length\": 3, \"petal width\": 1},
      {\"petal length\": 3, \"petal width\": 5.1}]");
multiModel.batchPredict(
    inputDataList, "data/predictions", true, null, null, null, null);
```

Prediction files can be subsequently retrieved and converted into a votes list using `batchVotes`:

```
List<MultiVote> batchVotes = multiModel.batchVotes(
    "data/predictions", null);
```

which will return a list of MultiVote objects. Each MultiVote contains a list of predictions, e.g.

```
 [
   {"prediction": "Iris-versicolor", "confidence": 0.34, "order": 0}, {"prediction":
↪"Iris-setosa", "confidence": 0.25, "order": 1}
 ]
```

These votes can be further combined to issue a final prediction for each input data element using the method `combine`

```
for (MultiVote multiVote: batchVotes) {
    HashMap<Object, Object> prediction = multivote.combine();
}
```

Again, the default method of combination is `plurality` for categorical predictions and mean value for numerical ones. You can also use `confidence weighted`:

```
    HashMap<Object, Object> prediction = multivote.combine(
        PredictionMethod.CONFIDENCE, null);
```

or `probability weighted`:

```
    HashMap<Object, Object> prediction = multivote.combine(
        PredictionMethod.PROBABILITY, null);
```

For classification, the confidence associated to the combined prediction is derived by first selecting the model's predictions that voted for the resulting prediction and computing the weighted average of their individual confidence. Nevertheless, when `probability weighted` is used, the confidence is obtained by using each model's distribution at the prediction node to build a probability distribution and combining them. The confidence is then computed as the wilson score interval of the combined distribution (using as total number of instances the sum of all the model's distributions original instances at the prediction node)

In regression, all the models predictions' confidences contribute to the weighted average confidence.

1.6.12 Local Ensembles

You can also instantiate a local version of a remote ensemble resource.

```
    import org.bigml.binding.BigMLClient;
    import org.bigml.binding.LocalEnsemble;

    BigMLClient api = new BigMLClient();

    // Get remote ensemble
    JSONObject ensemble = api.getEnsemble(
        "ensemble/5143a51a37203f2cf7020351");

    // Create local ensemble
    LocalEnsemble localEnsemble = new LocalEnsemble(ensemble);
```

The local ensemble object can be used to manage the three types of ensembles: `Decision Forests` (bagging or random) and the ones using `Boosted Trees`.

The `operatingKind` argument overrides the legacy `method` argument, which was previously used to define the combiner for the models predictions.

Similarly, local ensembles can also be created by giving a list of models to be combined to issue the final prediction (note: only random decision forests and bagging ensembles can be built using this method):

```
    import org.bigml.binding.LocalEnsemble;
    List models = new ArrayList();
    models.add("model/50c0de043b563519830001c2");
    models.add("model/50c0de043b5635198300031b");
    LocalEnsemble localEnsemble = new LocalEnsemble(models, 10);
```

Note: the ensemble JSON structure is not self-contained, meaning that it contains references to the models that the ensemble is build of, but not the information of the models themselves. To use an ensemble locally with no connection to the internet, you must make sure that not only a local copy of the ensemble JSON file is available in your computer, but also the JSON files corresponding to the models in it. This is automatically achieved when you use the `LocalEnsemble(ensemble)` constructor, that fetches all the related JSON files and stores them in an `./storage` directory. Next calls to `Ensemble(ensemble)` will retrieve the files from this local storage, so that internet connection will only be needed the first time an `LocalEnsemble` is built.

On the contrary, if you have no memory limitations and want to increase prediction speed, you can create the ensemble from a list of local model objects. Then, local model objects will only be instantiated once, and this could increase performance for large ensembles.

1.6.13 Local Ensemble's Predictions

As in the local model's case, you can use the local ensemble to create new predictions for your test data, and set some arguments to configure the final output of the `predict` method.

The predictions' structure will vary depending on the kind of ensemble used. For `Decision Forests` local predictions will just contain the ensemble's final prediction if no other argument is used.

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    localEnsemble.predict(inputData, null, null, null, null, false)
```

returns

```
Iris-versicolor
```

The final prediction of an ensemble is determined by aggregating or selecting the predictions of the individual models therein. For classifications, the most probable class is returned if no especial operating method is set. Using `full=True` you can see both the predicted output and the associated probability:

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    localEnsemble.predict(inputData, null, null, null, null, null, true, null)
```

returns

```
  {
      "prediction": "Iris-versicolor",
      "probability": 0.98566
  }
```

In general, the prediction in a classification will be one amongst the list of categories in the objective field. When each model in the ensemble is used to predict, each category has a confidence, a probability or a vote associated to this prediction. Then, through the collection of models in the ensemble, each category gets an averaged confidence, probabiity and number of votes. Thus you can decide whether to operate the ensemble using the `confidence`, the `probability` or the `votes` so that the predicted category is the one that scores higher in any of these quantities. The criteria can be set using the `operatingKind` option (default is set to `probability`):

```
    JSONObject inputData = JSONValue.parseValue(
        "{\"petal length\": 3, \"petal width\": 1}");
    localEnsemble.predict(
        inputData, null, null, null, null, "votes", true, null);
```

Regression will generate a predictiona and an associated error, however `Boosted Trees` don't have an associated confidence measure, so only the prediction will be obtained in this case.

For consistency of interface with the `LocalPredictiveModelModel` class, as well as between boosted and non-boosted ensembles, local Ensembles again have a `predictProbability` method. This takes the same optional arguments as `LocalPredictiveModelModel.predict`: `missingStrategy`.

Operating point predictions are also available for local ensembles and an example of it would be:

```
JSONObject operatingPoint = JSONValue.parseValue(
    "{\"kind length\": \"probability\",
      \"positive_class width\": \"True\",
      \"threshold\": 0.8}");
localEnsemble.predict(
    inputData, null, null, null, operatingPoint, null, true, null)
```

You can check the *Operating point's predictions* section to learn about operating points. For ensembles, three kinds of operating points are available: `votes`, `probability` and `confidence`. `Votes` will use as threshold the number of models in the ensemble that vote for the positive class. The other two are already explained in the above mentioned section.

1.6.14 Rule Generation

You can also use a local predictive model to generate a IF-THEN rule set that can be very helpful to understand how the model works internally.

```
 localModel.rules();

 IF petal_length > 2.45 AND
    IF petal_width > 1.75 AND
        IF petal_length > 4.85 THEN
            species = Iris-virginica
        IF petal_length <= 4.85 AND
            IF sepal_width > 3.1 THEN
                species = Iris-versicolor
            IF sepal_width <= 3.1 THEN
                species = Iris-virginica
    IF petal_width <= 1.75 AND
        IF petal_length > 4.95 AND
            IF petal_width > 1.55 AND
                IF petal_length > 5.45 THEN
                    species = Iris-virginica
                IF petal_length <= 5.45 THEN
                    species = Iris-versicolor
            IF petal_width <= 1.55 THEN
                species = Iris-virginica
        IF petal_length <= 4.95 AND
            IF petal_width > 1.65 THEN
                species = Iris-virginica
            IF petal_width <= 1.65 THEN
                species = Iris-versicolor
IF petal_length <= 2.45 THEN
    species = Iris-setosa
```

1.6.15 Summary generation

You can also print the model from the point of view of the classes it predicts with `localModel.summarize()`. It shows a header section with the training data initial distribution per class (instances and percentage) and the final predicted distribution per class.

Then each class distribution is detailed. First a header section shows the percentage of the total data that belongs to the class (in the training set and in the predicted results) and the rules applicable to all the the instances of that class (if any). Just after that, a detail section shows each of the leaves in which the class members are distributed. They are sorted in descending order by the percentage of predictions of the class that fall into that leaf and also show the full rule chain that leads to it.

```
    Data distribution:
        Iris-setosa: 33.33% (50 instances)
        Iris-versicolor: 33.33% (50 instances)
        Iris-virginica: 33.33% (50 instances)

    Predicted distribution:
        Iris-setosa: 33.33% (50 instances)
        Iris-versicolor: 33.33% (50 instances)
        Iris-virginica: 33.33% (50 instances)

    Field importance:
        1. petal length: 53.16%
        2. petal width: 46.33%
        3. sepal length: 0.51%
        4. sepal width: 0.00%

    Iris-setosa : (data 33.33% / prediction 33.33%) petal length <= 2.45
        · 100.00%: petal length <= 2.45 [Confidence: 92.86%]

    Iris-versicolor : (data 33.33% / prediction 33.33%) petal length > 2.45
        · 94.00%: petal length > 2.45 and petal width <= 1.65 and petal length <= 4.
↪95 [Confidence: 92.44%]
        · 2.00%: petal length > 2.45 and petal width <= 1.65 and petal length > 4.95␣
↪and sepal length <= 6.05 and petal width > 1.55 [Confidence: 20.65%]
        · 2.00%: petal length > 2.45 and petal width > 1.65 and petal length <= 5.05␣
↪and sepal width > 2.9 and sepal length > 6.4 [Confidence: 20.65%]
        · 2.00%: petal length > 2.45 and petal width > 1.65 and petal length <= 5.05␣
↪and sepal width > 2.9 and sepal length <= 6.4 and sepal length <= 5.95 [Confidence:␣
↪20.65%]

    Iris-virginica : (data 33.33% / prediction 33.33%) petal length > 2.45
        · 76.00%: petal length > 2.45 and petal width > 1.65 and petal length > 5.05␣
↪[Confidence: 90.82%]
        · 12.00%: petal length > 2.45 and petal width > 1.65 and petal length <= 5.05␣
↪and sepal width <= 2.9 [Confidence: 60.97%]
        · 6.00%: petal length > 2.45 and petal width <= 1.65 and petal length > 4.95␣
↪and sepal length > 6.05 [Confidence: 43.85%]
        · 4.00%: petal length > 2.45 and petal width > 1.65 and petal length <= 5.05␣
↪and sepal width > 2.9 and sepal length <= 6.4 and sepal length > 5.95 [Confidence:␣
↪34.24%]
        · 2.00%: petal length > 2.45 and petal width <= 1.65 and petal length > 4.95␣
↪and sepal length <= 6.05 and petal width <= 1.55 [Confidence: 20.65%]
```

You can also use `localModel.getDataDistribution()` and `local_model.getPredictionDistribution()` to obtain the training and prediction basic distribution information as a list (suitable to draw histograms or any further processing). The tree nodes' information (prediction, confidence, impurity and distribution) can also be retrieved in a CSV format using the method `localModel.exportTreeCSV()`. The output can be sent to a file by providing a `outputFilePath` argument or used as a list.

Local ensembles have a localEnsemble.summarize() method too, the output in this case shows only the data distribution (only available in Decision Forests) and field importance sections.

For local clusters, the localCluster.summarize() method prints also the data distribution, the training data statistics per cluster and the basic intercentroid distance statistics. There's also a localCluster.statisticsCsv(file_name) method that store in a CSV format the values shown by the summarize() method. If no file name is provided, the function returns the rows that would have been stored in the file as a list.

1.7 Running the tests

There is a test suite using Cucumber available, you may want to run it by execute:

```
$ mvn test
```

or this way, if you want to debug the tests

```
$ mvn -Dmaven.surefire.debug="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,
↪suspend=y,address=8000 -Xnoagent -Djava.compiler=NONE" test
```

or this way, if you want run an specific feature

```
$ mvn test -Dcucumber.options="--glue classpath:org.bigml.binding --format pretty src/
↪test/resources/test_01_prediction.feature"
```