bigml-csharp Documentation

Release latest

Oct 26, 2020

Contents

The core information in BigML resources is immutable to ensure traceability and reproducibility. However, some helpful properties like name, description, category and tags can be modified. For sources and datasets contained data itself is immutable, but you may need to change some properties, like parsing formats or field types to ensure that data is correctly handled.

CHAPTER 1

Updating a source

Sources describe the structure inferred by BigML from the uploaded data. This structure includes field names and types, locale, missing values, etc. You can learn more about all the available source attributes at https://bigml.com/api/sources#sr_source_properties. Some of them can be updated to ensure that your data is correctly interpreted. For instance, you could upload a source where a column contains only a subset of integers. BigML will consider this a numeric field. However, in your domain these integers could be the code associated to a category. Then, the field should be handled like a categorical field. You can change the type assigned to the field by calling:

```
using BigML;
using System;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;

namespace Demo
{
    /// <summary>
    /// Updates a source stored in BigML.
    ///
    /// See complete Sources documentation at
    /// https://bigml.com/api/sources
    /// </summary>
    class UpdateSource
    {
        static async void Main()
        {
            // New BigML client in production mode with username and API key
            Console.Write("user: ");
            var User = Console.ReadLine();
            Console.Write("key: ");
            var ApiKey = Console.ReadLine();
            var client = new Client(User, ApiKey);

            // change the id to your source
            string sourceId = "source/57d7240228eb3e69f3000XXX";
```

(continues on next page)

(continued from previous page)

```
            dynamic data = new JObject();
            data["fields"] = new JObject();
            data["fields"]["000000"] = new JObject();
            data["fields"]["000000"].optype = OpType.Categorical.ToString().ToLower();

            // Apply changes
            client.Update<Source>(sourceId, data);
        }
    }
}
```

where `sourceId` is the variable that contains the ID of the source to be updated and we change the type of the field whose ID is `000000` to categorical. The IDs for the fields can be found in the `fields` attribute of the source structure, which contains the properties of each field keyed by its ID.

CHAPTER 2

Updating a dataset

Datasets are the starting point for all models in BigML and contain a serialized version of your data where each field has been summarized and some basic statistics computed. According to its contents, each field gets an attribute called `preferred` whose value is a boolean. The value is set to `true` when BigML thinks that this field will be useful as predictor when creating a model. Fields like IDs, constants or with unique values are marked as `preferred = false` as they are not usually useful when modeling. However, this attribute can be changed if you still want to include them as possible predictors for the model.

Another commonly updated attribute in a dataset is the `objective_field`, that is, the field to be predicted in decision trees, ensembles or logistic regressions. By default, the last field in your dataset is used as objective field. The following example shows how to update the objective field to the first field in your dataset, whose ID is `000000`, and to include or exclude several fields from further analysis by changing their `preferred` attribute.

```
using BigML;
using System;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;

namespace Demo
{
    /// <summary>
    /// Update some properties of a Dataset stored in BigML
    ///  * Change objective field
    ///  * Mark one field as preferred
    ///  * Exclude one field
    ///
    /// See complete Dataset documentation at
    /// https://bigml.com/api/datasets
    /// </summary>
    class UpdateDataset
    {
        static async void Main()
        {
            // New BigML client in production mode with username and API key
            Console.Write("user: "); var User = Console.ReadLine();
```

(continues on next page)

(continued from previous page)

```
            Console.Write("key: "); var ApiKey = Console.ReadLine();
            var client = new Client(User, ApiKey);

            // Update this string with your dataset Id
            string datasetId = "dataset/57d7283b28eb3e69f1000XXX";

            dynamic data = new JObject();
            data["fields"] = new JObject();
            // Mark one field as preferred
            data["fields"]["000000"] = new JObject();
            data["fields"]["000000"].preferred = true;
            // Exclude these two fields
            data["fields"]["00003f"] = new JObject();
            data["fields"]["00003f"].preferred = false;
            data["fields"]["000041"] = new JObject();
            data["fields"]["000041"].preferred = false;
            // Update Objective field
            data["objective_field"] = new JObject();
            data["objective_field"]["id"] = "000003";

            // Apply changes
            client.Update<DataSet>(datasetId, data);
        }
    }
}
```

CHAPTER 3

Batch predictions

This chapter builds Quickstart on and shows how to create a **batch prediction** for multiple instances using an existing **ensemble**. Using an `ensemble` instead on a `model` is just a variation so you can see how flexible and orthogonal BigML API is.

To create a `Batch prediction`, you will need an `ensemble` and a `dataset` containing all the data instances you want predictions for. You can easily create both resources by following the steps detailed in Quickstart. In the following code, you will only focus on the last step in the process, i.e., creating the `Batch prediction`.

```
using BigML;
using System;
using System.Threading.Tasks;

namespace Demo
{
  /// <summary>
  /// This example creates a batch prediction using a dataset and an ensemble
  ///  stored in BigML.
  ///
  /// See complete API developers documentation at https://bigml.com/api
  /// </summary>
  class CreatesBatchPrediction
  {
    static async void Main()
    {

      // --- New BigML client using user name and API key ---
      Console.Write("user: ");
      var user = Console.ReadLine();
      Console.Write("key: ");
      var apiKey = Console.ReadLine();
      var client = new Client(user, apiKey);

      // --- Create a Batch Prediction from a previously created ensemble ---
      // The ensemble id and the dataset id which will be used to create a batch
```

(continues on next page)

(continued from previous page)

```
      // prediction.
      string modelId = "ensemble/54ad6d0558a27e2ddf000XXX";
      string datasetId = "dataset/54ad6d0558a27e2ddf000YYY";

      // Batch prediction object which will encapsulate all required information
      BatchPrediction batchPrediction;
      // setting the parameters to be used in the batch prediction creation
      var parameters = new BatchPrediction.Arguments();
      // the "model" parameter can be a Model, an Ensemble or a Logisticregression
      parameters.Add("model", modelId);
      parameters.Add("dataset", datasetId);
      // optionally, BigML can create a dataset with all results
      parameters.Add("output_dataset", true);
      // start the remote operation
      batchPrediction = await client.CreateBatchPrediction(parameters);
      string batchId = batchPrediction.Resource;
      // wait for the batch prediction to be created
      while ((batchPrediction = await client.Get<BatchPrediction>(batchId))
                                          .StatusMessage
                                          .NotSuccessOrFail())
      {
        await Task.Delay(5000);
      }
      Console.WriteLine(batchPrediction.OutputDatasetResource);
    }
  }
}
```

In the code above, if you want to use a `model` or a `logistic regression` instead of an `ensemble`, all you have to do is specify the model's or logistic regression's Id for the "model" parameter.

CHAPTER 4

Local predictions

In addition to making predictions remotely on BigML servers, the BigML C# bindings also provide support for making predictions locally. This means that you can download your `model`, `ensemble`, or `logistic regression` resource and use it to make predictions offline, i.e., without accessing the network.

The following code snippet shows how you can retrieve a `model` created in BigML and use it for local predictions:

```
using BigML;
using System;
using System.Threading.Tasks;

namespace Demo
{
  /// <summary>
  /// This example retrieves a model previously created in BigML and uses
  /// it to make a prediction locally.
  ///
  /// See complete API developers documentation at https://bigml.com/api
  /// </summary>
  class RetrieveModelPredictLocally
  {
    static async void Main()
    {
      // --- New BigML client using user name and API key ---
      Console.Write("user: ");
      var User = Console.ReadLine();
      Console.Write("key: ");
      var ApiKey = Console.ReadLine();
      var client = new Client(User, ApiKey);

      // --- Retrieve an existing model whose Id is known ---
      Model model;
      string modelId = "model/575085112275c16672016XXX";
      while ((model = await client.Get<Model>(modelId))
                                   .StatusMessage
```

(continues on next page)

(continued from previous page)

```
                                   .StatusCode != Code.Finished)
      {
        await Task.Delay(5000);
      }
      Model.LocalModel localModel = model.ModelStructure();

      // --- Specify prediction inputs and calculate the prediction ---
      Dictionary<string, dynamic>
      inputData = new Dictionary<string, dynamic>();
      inputData.Add("sepal length", 5);
      inputData.Add("sepal width", 2.5);
      Model.Node prediction = localModel.predict(inputData);
    }
  }
}
```

CHAPTER 5

Resources management

You can list BigML's resources using the client listing functions. In addition using `LINQ` xpressions you can retrieve, filter and order a collection of them.

5.1 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.

```
string User = "myuser";
string ApiKey = "80bf873537c19260370a1debf995eb57dd63cXXX";
var client = new Client(User, ApiKey);

// --- Select sources starting at tenth position
Ordered<Source.Filterable, Source.Orderable, Source> result
      = (from s in client.ListSources()
        where s.Offset == 10
        select s);

// Wait for the results
var sources = await result;

foreach (var src in sources)
{
  // print results
  Console.WriteLine(src.ToString());
}
```

5.2 Ordering resources

In order to sort 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.

```
  string User = "myuser";
  string ApiKey = "80bf873537c19260370a1debf995eb57dd63cXXX";
  var client = new Client(User, ApiKey);

  // --- Select 50 sources and order by creation date
  // from more recent to less recent
  Ordered<Source.Filterable, Source.Orderable, Source> result
        = (from s in client.ListSources()
          where s.Limit == 50
          orderby s.Created descending
          select s);

  // Wait for the results from the server
  var sources = await result;

  foreach (var src in sources)
  {
    // do work with each resource: e.g. print results
    Console.WriteLine(src.ToString());
  }
```

5.3 Updating resources

In general, the core information in BigML resources is immutable. This is so to ensure traceability and reproducibility. Therefore, only non-essential properties that are available as auxiliar data, like `name`, `description`, `tags` and `category`, can be modified. They are listed in the API help section.

The following is an example about how to update the tags in a model. The `Update` method in the `Client` class receives as arguments the ID of the resource that is to be updated and the properties that should be changed. The ID of a resource follow the schema `resource_type/alphanumeric24digits00ID` and can be found in the `Resource` method of the corresponding class and is the final part of the URL that points to the resource view in the Dashboard.

```
using BigML;
using System;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;

namespace Demo
{
  /// <summary>
  /// This example updates a Model previously created.
  ///
  /// See complete API developers documentation at https://bigml.com/api
  /// </summary>
  class UpdatesModelTags
  {
```

(continues on next page)

(continued from previous page)

```
    static async void Main()
    {
      string User = "myuser";
      string ApiKey = "80bf873537c19260370a1debf995eb57dd63cXXX";
      var client = new Client(User, ApiKey);

      // Get the Model
      Model m;
      string modelId = "model/57f65df5421aa9efdf000YYY";
      while ((m = await client.Get<Model>(modelId))
                              .StatusMessage.NotSuccessOrFail())
      {
        await Task.Delay(5000);
      }

      // Set the tags and update the Model
      JArray tags = new JArray();
      tags.Add("cool");
      tags.Add("production");
      JObject changes = new JObject();
      changes["tags"] = tags;
      m = await client.Update<Model>(m.Resource, changes);
    }
  }
}
```

Shareable resources allow you to modify their `privacy` and for resources included in projects the `project` is also updatable. Due to their special nature, some resources like `Source` or `Script` have other properties that can be updated, e.g. the locale used in `source_parser` of a source or `inputs` types or descriptions of the arguments for a script. Other that that, if you want to modify a property of your existing resource you will need to create a new one. That's for instance the case when you want to add data to a previously created model. You just upload your new data to create a new `source`, generate a `dataset` from it and merge it with the existing dataset that contained the old data. The consolidated dataset can now be used to rebuild a new `model` based on the entire data set.

5.4 Removing resources

All the resources stored in BigML can be removed individually using their ID. All BigML resources IDs follow the schema `resource_type/alphanumeric24digits00ID` and this is the only required argument of the `client.Delete` function.

```
  // Remove a known model
  string resourceID = "model/57f65df5421aa9efdf000YYY"
  Response rs = await client.Delete(resourceID);
```

CHAPTER 6

Projects

A project is an abstract resource that helps you group related BigML resources. You can assign any resource to a pre-existing project (except for projects themselves). When a user assigns a source or a dataset to a project, all the subsequent resources created using it will belong to the same project.

When using projects, resources will be organized in your Dashboard in project folders. If you change the project in the project selection bar the shown resources will change accordingly. To show all the resources or mix resources from different projects unselect any current project and set selector as “All”.

6.1 Creating a project

To create a project you don’t need to provide any information and BigML will give a default name to it. However, it’s recommended to set a meaningful name, description or tags to make it easy to work with.

```
using BigML;
using System;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;

namespace Demo
{
  /// <summary>
  /// This example creates a new project, uploads a local file that generates a
  /// source and includes it in the previosly created project.
  ///
  /// See complete API developers documentation at https://bigml.com/api
  /// </summary>
  class CreatesProjectAddSource
  {
    static async void Main()
    {
      string User = "myuser";
      string ApiKey = "80bf873537c19260370a1debf995eb57dd63cXXX";
```

(continues on next page)

(continued from previous page)

```
      var client = new Client(User, ApiKey);

      // Create a new project
      Project.Arguments pArgs = new Project.Arguments();
      pArgs.Add("name", "My tests");
      Project p = await client.CreateProject(pArgs);

      string projectID = p.Resource;

      // Create a new source
      Source s = await client.CreateSource("C:/Users/files/data.csv", "Data");
      while ((s = await client.Get<Source>(s.Resource))
                        .StatusMessage.NotSuccessOrFail())
      {
        await Task.Delay(5000);
      }

      // Set the project and update the Source
      JObject changes = new JObject();
      changes["project"] = projectID;
      s = await client.Update<Source>(s.Resource, changes);
    }
  }
}
```

Every resource stored in BigML can be moved from one project to another or included in a certain project[#]_. As in the previous example, you only need to make an update operation on the resource giving the projectID. For instance, you can create a bunch of models in your "Tests" project and move the best one to your "Production" project. In the Dashboard, resources can be moved from the list view or from the detail view.

6.2 Removing a project

One of the benefits of using projects for your work is that removing a project implies removing all the resources included. Thus, you won't need to remove them one by one.

```
using BigML;
using System;
using System.Threading.Tasks;

namespace Demo
{
  /// <summary>
  /// This example removes a project
  ///
  /// See complete API developers documentation at https://bigml.com/api
  /// </summary>
  class RemovesProject
  {
    static async void Main()
    {
      string User = "myuser";
      string ApiKey = "80bf873537c19260370a1debf995eb57dd63cXXX";
      var client = new Client(User, ApiKey);

```

(continues on next page)

(continued from previous page)

```
            // This is the project with my tests
            string projectID = "project/57f65df5421aa9efdf000YYY";

            // Remove the project (and its resources)
            Response rs = await client.Delete(projectID);
        }
    }
}
```

.. [#] Once a resource is included in a project, it can be moved to another one but it must remain associated to one of the existing projects.

CHAPTER 7

WhizzML

WhizzML is a domain-specific language for automating Machine Learning workflows, implementing high-level Machine Learning algorithms, and easily sharing them with others. WhizzML offers out-of-the-box scalability, abstracts away the complexity of underlying infrastructure, and helps analysts, developers, and scientists reduce the burden of repetitive and time-consuming analytics tasks.

7.1 Create a script

WhizzML's main elements are scripts that contain sentences to create, transform and manage BigML's resources. To know more about its syntax you can refer to the examples, tutorials and references in WhizzML Page. The following one is a simple example about how to create a script that allows to import a source using WhizzML. The core of the script is its `source_code` and contains the code that will be executed when running the script.

```
// setting the parameters to be used in script creation
Script.Arguments scArgs = new Script.Arguments();
scArgs.Add("source_code",
           "(create-source {\"remote\" \"https://static.bigml.com/csv/iris.csv\"})");
scArgs.Add("name", "add a remote file");
Script sc = await client.CreateScript(scArgs);
```

7.2 Create an execution

This section shows how to create an execution of a script stored in BigML. You only need to know the Id of the script you want to execute and provide the input parameters for the actual execution.

```

// --- Retrieve an existing script whose Id is known ---
string scriptId = "script/50a2eac63c19200bd1000XXX";

Execution.Arguments exArgs = new Execution.Arguments();
```

(continues on next page)

(continued from previous page)

```
exArgs.Add("script", scriptId);
exArgs.Add("name", "my script execution");
Execution exec = await client.CreateExecution(exArgs);
```

That's all! BigML will create the execution as requested. That generates an `execution` resource that automatically appears in the BigML Dashboard executions listings. Once the execution reaches a `finished` status, the results are returned in the `Results` property of the `Execution` object.

CHAPTER 8

Where to go from here

The previous chapters introduced the basic philosophy of the BigML C# bindings and provided a few code samples showing how you can use them to create several kinds of predictions. That was already a wealth of information, but there is much more to learn about BigML REST API and the BigML C# bindings. In the following sections, you will find a few pointer to help you learn about all the features that BigML provides to developers, how to contribute to BigML C# bindings, and more.

8.1 Discovering BigML

BigML is keen to provide all of its users as much information as possible to help them make the best out of its functionality. On BigML web site, you can find thorough guides to all of the features its Dashboard Web UI offers, a number of tutorials ranging from general introduction to ML-techniques to how machine learning can be applied to real-life problems such as churn prediction, loan risk prediction, etc.

8.2 BigML REST API

BigML provides a rich REST API developers can use to bring the power of machine-learning to their apps. As a proof of the flexibility and power of BigML REST API, it may be interesting to know that BigML web site is entirely based upon them! BigML REST API includes support for advanced machine learning algorithms such as decision trees, ensembles, clusters, anomaly detectors, association rules, and more. Additionally, BigML provides a powerful workflow automation platform based on BigML platform-agnostic DSL for machine learning, WhizzML. You can discover all that BigML REST API has to offer on BigML web site.

8.3 Contributing to bindings

BigML C# bindings are open-source, and pretty much a constant work in progress given the pace with which BigML feature set keeps growing. If you ever happen to find a bug, or would like to provide a fix or an improvement, you are welcome to clone BigML C# bindings repository on GitHub and send us a pull request.

Also, get in touch for general feedback and to tell us what features you consider most important to have implemented in BigML C# bindings.

CHAPTER 9

BigML C# Bindings

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

9.1 Additional Information

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

CHAPTER 10

Introduction

10.1 Requirements and installation

BigML C# bindings use the *Newtonsoft.Json* DLL that was released as part of .NET Silverlight Framework, and you will need to explicitly install it in your system, if you have not already.

To install *Newtonsoft.Json* you can use Visual Studio Package Manager. In your Visual Studio IDE, go to the Package Manager console (Tools > Library Package Manager > Package Manager Console) and type the following command:

```
Install-Package Newtonsoft.Json -Version 12.0.3
```

If the installation is successful, you should see a message like the following one:

```
'Newtonsoft.Json 12.0.3' was successfully added to <your project
name here>.
```

Once you have the *Newtonsoft.Json* package installed, you can use Visual Studio Package Manager to install BigML C# bindings:

```
Install-Package BigML
```

10.2 Authentication

To access BigML using the bindings, you first create a new *client* object by passing your user name and API Key. This is how you can initialize a new *Client* object by retrieving user name and API Key from your standard input:

```
// New BigML client using user name and API key.
Console.Write("user: ");
var user = Console.ReadLine();
Console.Write("key: ");
var apiKey = Console.ReadLine();
var client = new Client(user, apiKey);
```

The *client* object encapsulates your credentials and provides methods for most of the operations available through the BigML API[1] .

10.3 Connecting to a Virtual Private Cloud

If you are using Managed Virtual Private Cloud (VPC), you can specify your VPC URL when instantiating your *client*:

```
var client = new Client(userName, apiKey, vpcDomain: "yourVPC.vpc.bigml.io");
```

BigML VPC provides transparent, exclusive access to resizable computing and data storage capacity in the cloud without needing to install or configure any hardware or software.

[1] You can find your API Key in your BigML account information panel. If needed, you can also create additional API Keys and restrict the privileges that are associated with each of them.

CHAPTER 11

Getting started

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 a **remote CSV file** <https://static.bigml.com/csv/iris.csv> containing the Iris flower dataset to predict the species of a flower based on its morphological characteristics. 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 you want to predict).

```
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 will need to create 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:

```
var parameters = new Source.Arguments();
parameters.Add("name", "my new source");
parameters.Add("remote",
   "https://static.bigml.com/csv/iris.csv");
Source source = await client.CreateSource(parameters);
```

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:

```
var filePath = "./iris.csv";
string name = "Iris file";
Source source = await client.CreateSource(filePath, name);
```

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:

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
Dataset dataset;
// setting the parameters to be used in dataset creation
var parameters = new Dataset.Arguments();
parameters.Add("name", "my new dataset");
// using the source ID as argument
parameters.Add("source", source.Resource);
dataset = await client.CreateDataset(parameters);
// checking the dataset status
while ((dataset = await client.Get<Dataset>(dataset))
                            .StatusMessage
                            .NotSuccessOrFail())
{
await Task.Delay(5000);
}
```

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 C# bindings as detailed in [Requirements and installation](#requirements-and-installation).

```
using BigML;
using System;
using System.Threading.Tasks;

namespace Demo
{
  /// <summary>
  /// This example creates a prediction using a model created with the data
  /// stored in a remote file.
  ///
  /// See complete API developers documentation at https://bigml.com/api
  /// </summary>
  class CreatesPrediction
```

(continues on next page)

(continued from previous page)

```
  {
    static async void Main()
    {
      // --- New BigML client using user name and API key ---
      Console.Write("user: ");
      var user = Console.ReadLine();
      Console.Write("key: ");
      var apiKey = Console.ReadLine();
      var client = new Client(user, apiKey);

      // --- create a source from the data in a remote file ---

      // setting the parameters to be used in source creation
      var parameters = new Source.Arguments();
      parameters.Add("name", "my new source");
      // uploading a remote file
      parameters.Add("remote", "https://static.bigml.com/csv/iris.csv");
      // if you need to upload a local file, change last line to
      // parameters.Add("file", "iris.csv");
      // Source object which will encapsulate the source information
      Source source = await client.CreateSource(parameters);
      // API calls are asynchronous, so you need to check that the source is finally
      // finished. To learn about the possible states for
      // BigML resources, please see http://bigml.com/api/status_codes
      while ((source = await client.Get<Source>(source))
                                 .StatusMessage
                                 .NotSuccessOrFail())
      {
        await Task.Delay(5000);
      }

      // --- create a dataset from the previous source ---
      // setting the parameters to be used in dataset creation
      var parameters = new Dataset.Arguments();
      parameters.Add("name", "my new dataset");
      // using the source ID as argument
      parameters.Add("source", source.Resource);
      // Dataset object which will encapsulate the dataset information
      Dataset dataset = await client.CreateDataset(parameters);
      // checking the dataset status
      while ((dataset = await client.Get<Dataset>(dataset))
                                  .StatusMessage
                                  .NotSuccessOrFail())
      {
        await Task.Delay(5000);
      }

      // --- create a model from the previous dataset ---
      // setting the parameters to be used in model creation
      var parameters = new Model.Arguments();
      parameters.Add("name", "my new model");
      // using the dataset ID as argument
      parameters.Add("dataset", dataset.Resource);
      // Model object which will encapsulate the model information
      Model model = await client.CreateModel(parameters);
      // checking the model status
      while ((model = await client.Get<Model>(model))
```

(continues on next page)

(continued from previous page)

```
                                    .StatusMessage
                                    .NotSuccessOrFail())
      {
        await Task.Delay(5000);
      }

      // --- create a prediction using the model ---
      // setting the parameters to be used in prediction creation
      var parameters = new Prediction.Arguments();
      // using the model ID as argument
      parameters.Add("model", model.Resource);
      // set INPUT DATA for prediction: {'petal length': 5, 'sepal width': 2.5}
      parameters.InputData.Add("petal length", 5);
      parameters.InputData.Add("sepal width", 2.5);

      // SET MISSING STRATEGY and NAME
      parameters.Add("missing_strategy", 1); //Proportional
      parameters.Add("name", "prediction w/ PROPORTIONAL");
      // Prediction object which will encapsulate the prediction information
      Prediction prediction = await client.CreatePrediction(parameters);
      // checking the prediction status
      while ((prediction = await client.Get<Prediction>(prediction))
                                         .StatusMessage
                                         .NotSuccessOrFail())
      {
          await Task.Delay(2000);
      }
      Console.WriteLine("------------------------------\nMissing strategy PROPORTIONAL
↪");
      Console.WriteLine("Prediction: " + prediction.GetPredictionOutcome<string>());
      Console.WriteLine("Confidence: " + prediction.Confidence);


      // Test same input_data, but with missing_stategy = 0 (default value)
      // UPDATE MISSING STRATEGY and NAME
      parameters.Update("missing_strategy", 0); //Last prediction
      parameters.Update("name", "prediction w/ LAST PREDICTION");
      prediction = await client.CreatePrediction(parameters);
      while ((prediction = await client.Get<Prediction>(prediction))
                                         .StatusMessage
                                         .NotSuccessOrFail())
      {
          await Task.Delay(2000);
      }

      Console.WriteLine("------------------------------\nMissing strat. LAST␣
↪PREDICTION");
      Console.WriteLine("Prediction: " + prediction.GetPredictionOutcome<string>());
      Console.WriteLine("Confidence: " + prediction.Confidence);
      Console.WriteLine("------------------------------");
    }
  }
}
```