PredictionIO: Text Manipulation Engine

In the real world, there are many applications that collect text as data. For example, suppose that you have a set of news articles that are categorized based on content, and you wish to automatically assign incoming, uncategorized articles to one of the existing categories. There are a wide array of machine learning models you can use create, or train, a predictive model to assign an incoming article, or query, to an existing category. Before you can use these techniques you must first transform the text data (in this case the set of news articles) into numeric vectors, or feature vectors, that can be used to train your model.

The purpose of this tutorial is to illustrate how you can go about doing this using PredictionIO's platform. The advantages of using this platform include: a dynamic engine that responds to queries in real-time; distributed computing capabilities for handling data processing and model training tasks with large data sets. In particular, we will show you how to:

- import a corpus of text documents into PredictionIO's event server;
- read the imported event data for use in text processing;
- transform document text into a feature vector;
- use the feature vectors to fit a classification model based on Naive Bayes (using Spark MLLib library implementation);
- use the feature vectors to fit a classification model based on Latent Dirichlet Allocation (using Spark MLLib library implementation.
- evaluate the performance of the fitted models;
- yield predictions to queries in real-time using a fitted model.

Prerequisites

Before getting started, please make sure that you have the latest version of PredictionIO installed . You will also need PredictionIO's Python SDK , and the Scikit learn library (http://scikit-learn.org/stable/) for importing a sample data set into the PredictionIO Event Server. Any Python version greater than 2.7 will work for the purposes of executing the `data/import_eventserver.py` script provided with this engine template. Moreover, we emphasize here that this is an engine template written in *Scala* and can be more generally thought of as an SBT project containing all the necessary components.

You should also download the engine template named Modeling Text Data that accompanies this tutorial.

Engine Overview

The engine follows the general DASE architecture which we briefly review here. As a user, you are tasked with collecting data for your web or application, and importing it into PredictionIO's Event Server. Once the data is in the server, it can be read and processed by the engine via the Data Source and Preparation components, respectively. The Algorithm component then trains a predictive model using the processed, or prepared, data. Once we have trained a model, we are ready to deploy our engine and respond to real-time queries via the Serving component. The Evaluation component is used to compute an appropriate metric to test the performance of a fitted model, as well as aid in the tuning of model hyper parameters.

We are working with text data which means our queries, or newly observed documents, are of the form

$$\{\texttt{text : String}\}.$$

In our example, a query would be an incoming news article. Once the engine is deployed it can process the query, and then return a Predicted Result of the form

$$\{\texttt{category : String, confidence : Double}\}.$$

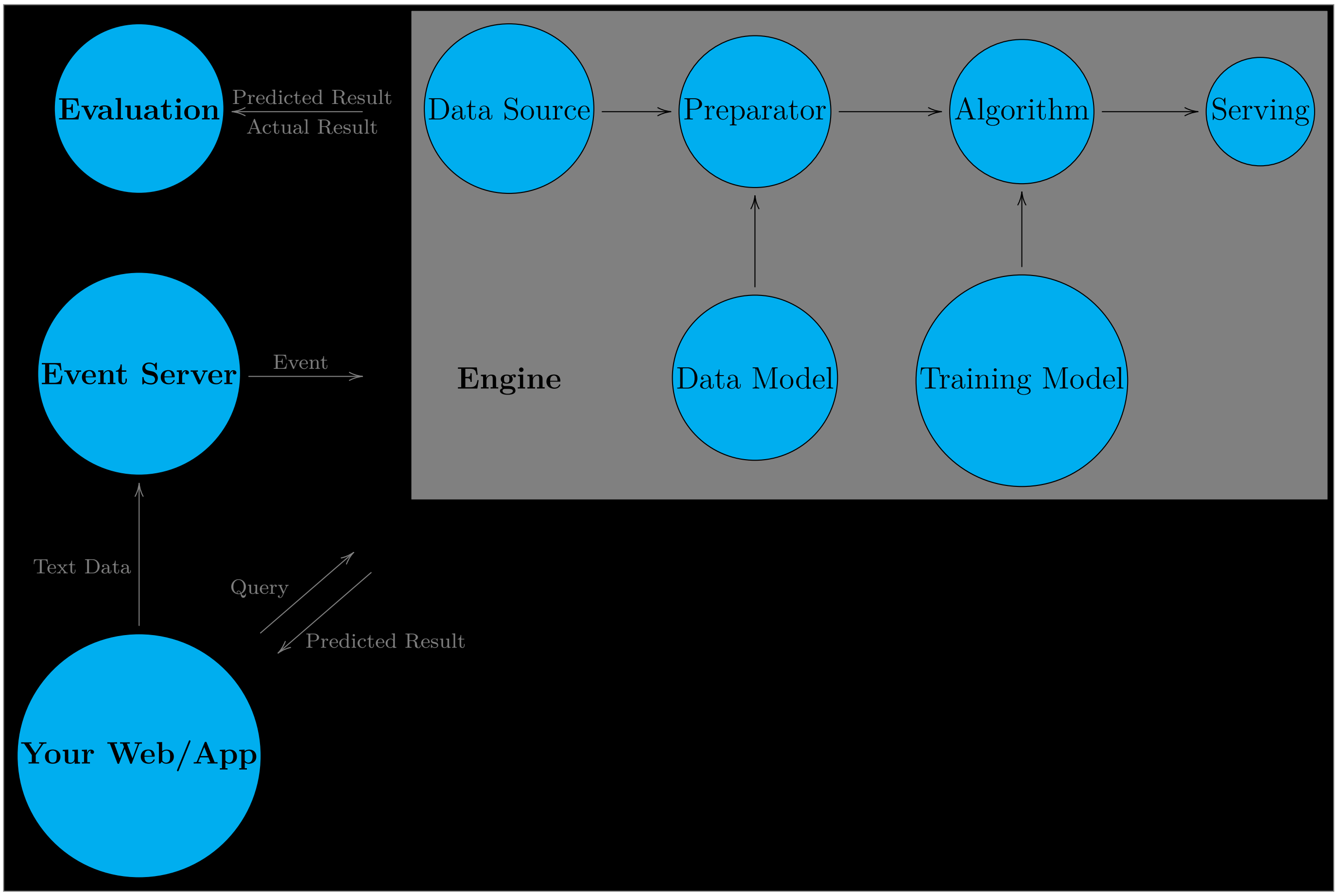
Here category is the model's class assignment for this new text document (i.e. news article) and confidence is a value between 0 and 1 representing our confidence in the category prediction (0 meaning we have no confidence in our prediction). The Actual Result is of the form

$$\{\texttt{category : String}\}.$$

This is used in the evaluation stage when estimating the performance of our predictive model.

In addition to the DASE components, our engine also includes the components DataModel and TrainingModel. DataModel is the muscle in the Preparator stage as it is the component that vectorizes the text data. The TrainingModel component takes the vectorized data and trains a classification model. The two particular Scala classes implemented in the engine template are named SupervisedModel and UnsupervisedModel.

To summarize: (1) we import text data to the Event Server; (2) the engine reads and processes the data, and trains a predictive model; (3) once the model is trained and the engine deployed, your web application can send text queries to which the engine can respond with a predicted result in real time; (4) in the evaluation, the engine produces both predicted results and actual results for the training data and feeds them to the Evaluation component. The figure below shows a graphical representation of the engine architecture just described, as well as its interactions with your web/app and a provided Event Server:



Quick Start

This is a quick start guide in case you want to start using the engine right away. For more detailed information, read the subsequent sections.

1. Create a new application. After the application is created, you will be given an access key for the application.

   ```
   $ pio app new MyTextApp
   ```

2. Import the tutorial data, and be sure to replace `***` with the access key obtained from the latter step.

   ```
   $ python import_eventserver.py --access_key ***
   ```

3. Set the engine parameters in the file `engine.json`. The default settings are shown below.

```
{
  "id": "default",
  "description": "Default settings",
  "engineFactory": "TextManipulationEngine.TextManipulationEngine",
  "datasource": {
    "params": {
      "appName": "marco-testapp",
      "evalK": 5
    }
  },
  "preparator": {
    "params": {
      "nMin": 1,
      "nMax": 2
    }
  },
  "algorithms": [
    {
      "name": "sup",
      "params": {
        "lambda": 0.5
      }
    }
  ]
}
```

4. Build your engine.

```
$ pio build
```

5.a. Train your model and deploy.

```
$ pio train
$ pio deploy
```

5.b. Run the evaluator.

```
$ pio eval
```

Depending on your needs, in steps (5.x.) above, you can configure your Spark settings by typing a command of the form:

```
$ pio command -- --master url --driver-memory {0}G --executor-memory {1}G
--conf spark.akka.framesize={2} --total_executor_cores {3}
```

We only list the latter commands as these are some of the more commonly modified values. See the Spark documentation and the PredictionIO FAQ's for more information

Importing Data

In order to stick with the news article example, we will, for the purpose of this illustration, be importing two different sources of data into PredictionIO's Event Serve: a corpus of news documents that are categorized into a set of topics, as well as a set of stop words. Stop words are words that we do not want to include in our corpus when modeling our text data. Both the data and stop words are imported from the Scikit learn Python library.

For the remainder of the tutorial, we will assume that the present working directory is the engine template root directory. The script used to import the data is `import_eventserver.py` located in the data directory. To actually import the data into our Event Server, we must first create an application which we will name MyTextApp. To do this run the shell command `pio app new MyTextApp`, and take note of your access key. If you forget your access key, you can obtain it by using the command `pio app list`. The following shell output shows the command needed for importing your data (first line), and the resulting output after the data has been successfully imported. Replace `***` with your actual access key, and `???` with the correct location of the port hosting HBase. You can usually leave out the url field as HBase will generally be hosting on port 7070.

```
$ python data/import_eventserver.py --access_key *** --url ???
Importing data.....
Imported 11314 events.
Importing stop words.....
Imported 318 stop words.
```

Data Source: Reading Event Data

Now that our data has been imported into PredictionIO's Event Server, it needs to be read from HBase for it to actually be used by our engine. This is precisely what the DataSource engine component is for. We first explain the classes Observation and Training Data which are defined in `DataSource.scala`. Observation serves as a wrapper for storing the information about a news document needed to train a model. The class member label refers to the label of the category a document belongs to, and text, stores the actual document content. TrainingData is used to store an RDD of Observation objects and our set of stop words.

The class DataSourceParams is used to specify the parameters needed to read and prepare the data for processing. This class is initialized with two parameters appName and evalK. The first parameter specifies your application name (i.e. MyTextApp), which is needed so that the DataSource component knows where to pull the event data from. The second parameter is used for model evaluation and specifies the number of folds to use in cross-validation when we estimate a model performance metric.

Finally, we come to the DataSource class. This is initialized with its corresponding parameter class, and extends PDataSource[TD, E, Q, AR]. This extension means that it **must** implement the method readTraining which returns an instance of type TD which is in this case the class TrainingData. This method completely relies on the defined *private* methods readEventData and readStopWords. Both of these functions read data observations as Event instances, create an RDD containing these events and finally transforms the RDD of events into an object of the appropriate type as seen below:

```
private def readEventData(sc: SparkContext) : RDD[Observation] = {
    //Get RDD of Events.
    PEventStore.find(
      appName = dsp.appName,
      entityType = Some("source"), // specify data entity type
      eventNames = Some(List("documents")) // specify data event name

      // Convert collected RDD of events to and RDD of Observation
      // objects.
    )(sc).map(e => Observation(
      e.properties.get[Double]("label"),
      e.properties.get[String]("text")
    )).cache
  }

  // Helper function used to store stop words from
  // event server.
  private def readStopWords(sc : SparkContext) : Set[String] = {
    PEventStore.find(
      appName = dsp.appName,
      entityType = Some("resource"),
      eventNames = Some(List("stopwords"))

    //Convert collected RDD of strings to a string set.
    )(sc)
      .map(e => e.properties.get[String]("word"))
      .collect
      .toSet
  }
```

Note that readEventData and readStopWords use different entity types and event names, but use the same application name. This is because we imported both our corpus and stop word set using the same access key. These field distinctions are required for distinguishing between the two data types. Also note that these methods require a SparkContext to be passed as a parameter. The method readEval also relies on readEventData and readStopWords, and its function is to prepare the different cross-validation folds needed for evaluating your model and tuning hyper parameters.

Data Model

Our data model implementation is actually just a Scala class taking in as parameters td, nMin, nMax, where td is an object of class TrainingData. The other two parameters are the components of our n-gram window which we will define shortly. In this section, we give an overview of how we go about representing our document strings. It will be easier to explain this process with an example, so consider the document:

$$D := \texttt{"Hello, my name is Marco."}$$

The first thing we need to do is break up D into a list of "allowed tokens." You can think of a token as a terminating sequence of characters that exist in our document (think of a word in a sentence). For example, the list of tokens that appear in D is:

$$\texttt{Hello} \to \texttt{,} \to \texttt{my} \to \texttt{name} \to \texttt{is} \to \texttt{Marco} \to \texttt{.}$$

Now, recall that when we imported our data, we also imported a set of stop words. This set of stop words contains all the words (or tokens) that we do not want to include once we tokenize our documents. Hence, we will call the tokens that appear in D and are not contained in our set of stop words allowed tokens. So, if our set of stop words is $\{\texttt{my}, \texttt{is}\}$, then the list of allowed tokens appearing in D is:

$$\texttt{Hello} \to \texttt{,} \to \texttt{name} \to \texttt{Marco} \to \texttt{.}$$

Preparator : Processing the Data