PredictionIO: Modeling Text Data 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 and you want to implement an automatic categorization system that groups existing articles based on content similarity, and assigns future news articles into one of these categories. There are a wide array of machine learning models you can use to first cluster the news articles into categories and subsequently build a predictive model to classify new articles into the learned categories. However, before being able to use these techniques you must first transform the text data (in this case the set of articles) into numeric vectors, or feature vectors, that can be used to create, or train, a 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 distributed data processing and model training for an increase in performance speed, as well as the capacity to use a newly trained predictive model to respond to queries in real-time. 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 Naive Bayes classification model (using Spark MLLib library implementation);
- use the feature vectors to fit a Latent Dirichlet Allocation clustering model (using Spark MLLib library implementation.
- evaluate the performance of the fitted models;
- yield predictions to queries in real-time using a fitted model.

Prerequisites

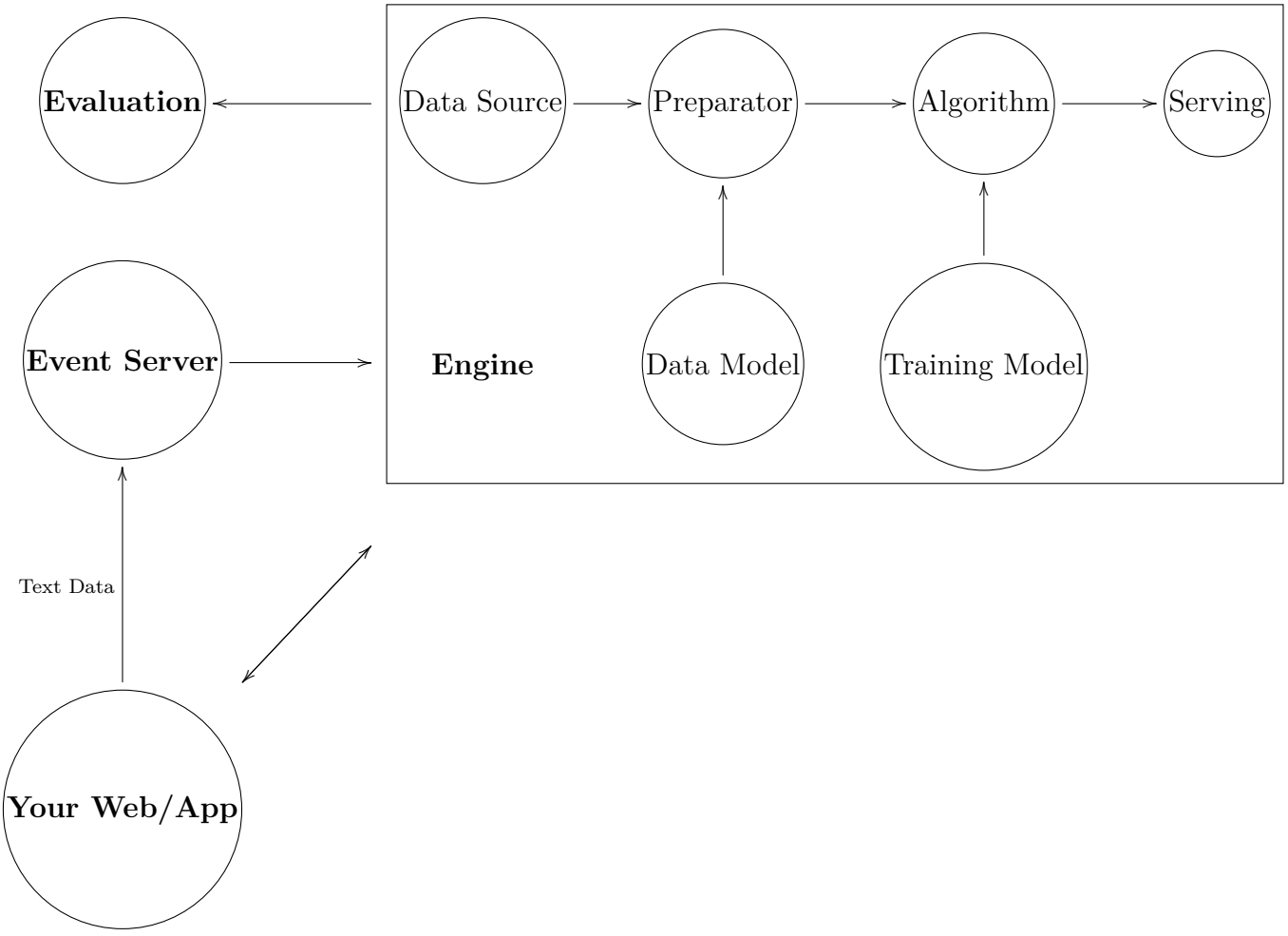
Before getting started, please make sure that you have the latest version of PredictionIO installed (https://docs.prediction.io/install/). You will also need PredictionIO's Python SDK (https://github.com/PredictionIO/PredictionIO-Python-SDK), and the Scikit learn library (http://scikit-learn.org/stable/) 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 (http://templates.prediction.i
that accompanies this tutorial.

Engine Overview

The engine follows the general DASE architecture which we briefly review here. Firstly, as a user, you are charged with collecting data from 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 our engine via the DataSource and Preparation components, respectively. The Algorithm engine 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.

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 which is more of a conceptual framework representing a set of Scala classes that can produce a predictive model. The two particular Scala classes implemented in the engine template are named SupervisedModel and UnsupervisedModel. 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:



Importing Data

In order to stick with the news article example, we will be importing two different sources of data into PredictionIO's: 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 `data/import_eventserver.py`. 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 how the command 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 7077.

```
$ 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.

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. 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 hyperparameters.

Preparator : Preparing the Data

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`, and 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:

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

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:

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