PredictionIO: Text Classification 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 to 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; separation of concerns, which offers code re-use and maintainability, and distributed computing capabilities for scalability and efficiency. Moreover, it is easy to incorporate non-trivial data modeling tasks into the DASE architecture allowing Data Scientists to focus on tasks related to modeling. We will exemplify these ideas in this tutorial, and, in particular, 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 (we will be modeling each document using n-grams and the tf-idf transformation);
- use the feature vectors to fit a classification model based on Multinomial 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 Text Classification Engine that accompanies this tutorial.

Engine Overview

The engine follows the 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 uses the processed, or prepared, data to train a set of predictive models. Once you have trained these models, you are ready to deploy your engine and respond to real-time queries via the Serving component which combines the results from different fitted models. 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.

This engine template is meant to handle text classification which means you will be working with text data. This means that a query, or newly observed documents, will be of the form

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

In the running 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

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

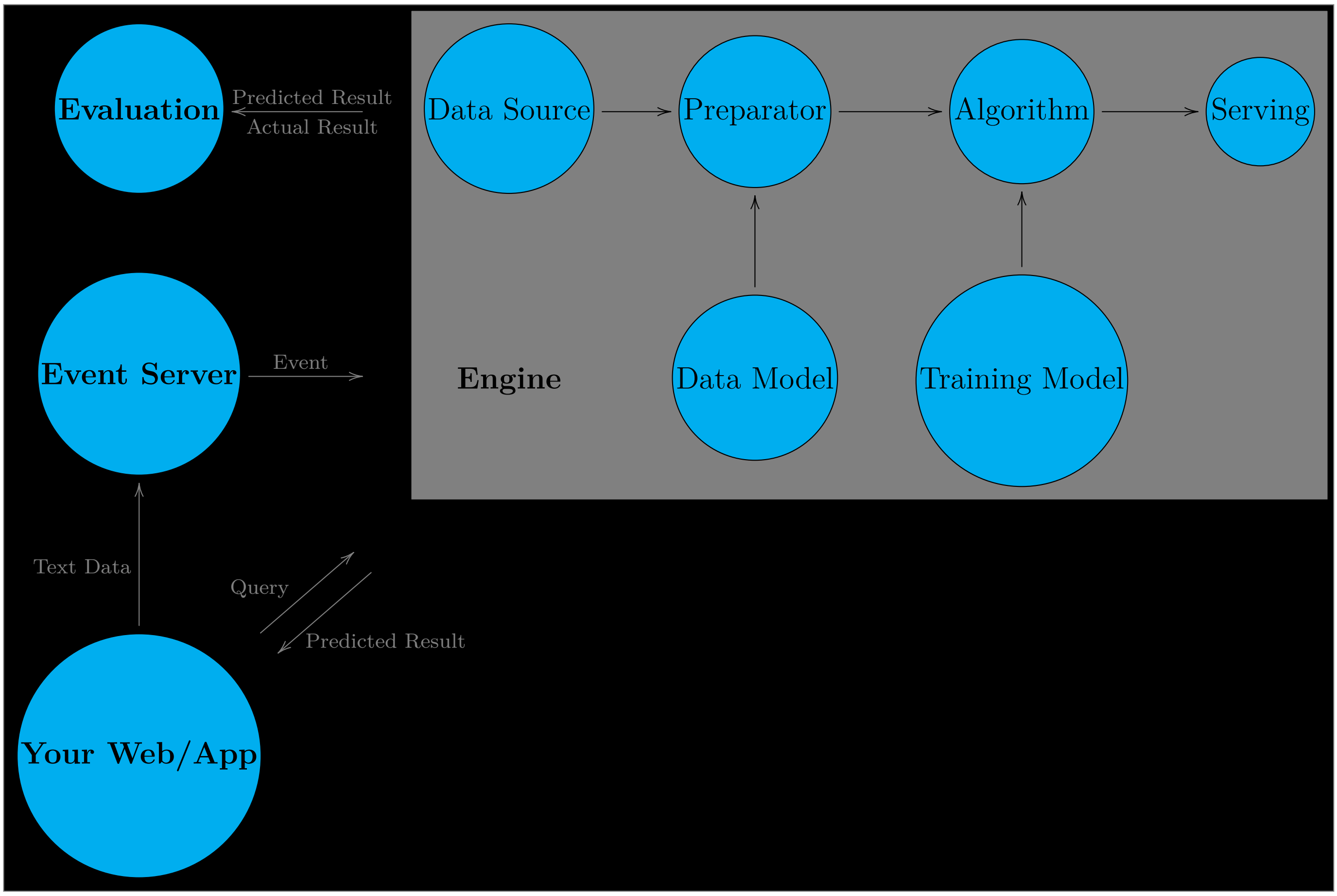
Here category is the model's class assignment for this new text document (i.e. the best guess for this article's categorization), and confidence, a value between 0 and 1 representing your confidence in the category prediction (0 meaning you have no confidence in the prediction). The Actual Result is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

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

This is used in the evaluation stage when estimating the performance of your predictive model (how well does the model predict categories).

In addition to the DASE components, we also introduce the Data Model and Training Model abstractions. The Data Model abstraction refers to the set of Scala classes dealing with the implementation of modeling choices relating to **feature extraction**, **preparation**, and/or **selection**. For this illustration, this only includes the vectorization of text and t.f.-i.d.f. processing which is entirely implemented in the PreparedData class. The Training Model abstraction refers to any set of classes that individually take in a set of feature observations and output a predictive model. This predictive model is leveraged by the Algorithm component to produce prediction results to queries in real-time. In the engine template, this abstraction is implemented in the NBModel (Naive Bayes) and LDAModel (Latent Dirichlet Allocation) classes. **Please note that these are conceptual abstractions that are designed to make engine development easier by decoupling class functionality.** Keeping these abstractions in mind will help you in the future with debugging, and also make it easier to incorporate different modeling ideas into your engine.

The figure below shows a graphical representation of the engine architecture just described, as well as its interactions with your web/app, the provided Event Server, and the defined abstractions:



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. If you have forgotten your access key, use the command `pio app list` to retrieve it.

   ```
   $ 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. Evaluate your training model and tune parameters.

```
$ pio eval
```

5.a. Train your model and deploy.

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

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}
```

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

Importing Data

For the remainder of the tutorial, your present working directory is assumed to be the engine template root directory. For illustration purposes, the script `import_eventserver.py` (located in the data directory) is included for importing two different sources of sample data into PredictionIO's Event Server: a corpus of text documents that are categorized into a set of topics, as well as a set of stop words. Stop words are words that you do not want to include in the corpus when modeling a set of text data.

Now, refer to the quick start guide for the commands used to import your data. Once the data has been successfully imported you should see the following output:

```
Importing data.....
Imported 11314 events.
Importing stop words.....
Imported 318 stop words.
```

This data import process greatly exemplifies the advantages of using PredictionIO's Event Server for data storage. It allows you to import data from different sources and store it using the same server. The event-style format allows for a standardized method of storing data which facilitates the process of reading in your data and incorporating different data sources. For example, the provided data script imports both the text observations and stop words which are going to inevitably differ in nature. In short, PredictionIO's Event Server is yet another abstraction that exacerbates your development productivity, as well as the ability to focus on the modeling stages involved in building your predictive engine.

Data Source: Reading Event Data

Now that the data has been imported into PredictionIO's Event Server, it needs to be read from storage to be used by the engine. This is precisely what the DataSource engine component is for, which is implemented in the template script `DataSource.scala`. The class Observation serves as a wrapper for storing the information about a news document needed to train a model. The attribute label refers to the label of the category a document belongs to, and text, stores the actual document content as a string. The class TrainingData is used to store an RDD of Observation objects along with the 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 estimating a model performance metric.

The final and most important ingredient is the DataSource class. This is initialized with its corresponding parameter class, and extends PDataSource. This **must** implement the method readTraining which returns an instance of type 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 the sample import script imports two different data types, documents and stop words. These field distinctions are required for distinguishing between the two. 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.

Processing the Data

This section deals with the aspect of transforming the data you read in the previous stage into something that you can actually use to train a predictive model.

Preparator : Data Processing in DASE

Recall that the Preparator stage is used for doing any prior data processing needed to fit a predictive model. In line with the separation of concerns, the Data Model implementation, PreparedData, is built to do the heavy lifting needed for this data processing. The Preparator must simply implement the prepare method which outputs an object of type PreparedData. This requires you to specify two n-gram window components (defined in the following section), a custom class of parameters for the Preparator component, PreparatorParams, must be incorporated.

The simplicity of this stage implementation truly exemplifies one of the benefits of using the PredictionIO platform. For developers, it is easy to incorporate different classes and tools into the DASE framework so that the process of creating an engine is greatly simplified which helps increase your productivity. For data scientists, the load of implementation details you need to worry about is minimized so that you can focus on what is important to you: training a good predictive model.

The following section actually explains the meat of the conversion process from text documents to feature vectors.

Data Model : Ideas Behind the PreparedData Implementation

For this Text Classification Engine, the Data Model abstraction is implemented in Prepared Data which takes the parameters td, nMin, nMax, where td is an object of class TrainingData. The other two parameters are the components of the model n-gram window which will be defined shortly. It will be easier to explain the preparation process with an example, so consider the document:

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

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

```
val A = Array("Hello", ",", "my", "name", "is", "Marco", ".")
```

Recall that a set of stop words was also imported in the previous sections. This set of stop words contains all the words (or tokens) that you do not want to include once documents are tokenized. Those tokens that appear in D and are not contained in the set of stop words will be called allowed tokens. So, if the set of stop words is {`"my"`, `"is"`}, then the list of allowed tokens appearing in D is:

```
val A = Array("Hello", ",", "name", "Marco", ".")
```

All of this functionality is implemented in the private method tokenize of the DataModel class. This method uses the SimpleTokenizer from OpenNLP's library to implement this (note that you must add the Maven dependency declaration to your `build.sbt` file to incorporate this library into your engine). You may want to extend this functionality so that all letters in your document are transformed to lowercase so that,for example, the tokens `""Hello"` and `"hello"` are considered equivalent. This is a **modeling choice** that is not implemented in the engine template. However, you may choose to include this to reduce the number of features you create.

The next step in the data representation is to take the array of allowed tokens and extract a set of n-grams and a corresponding value indicating the number of times a given n-gram appears. The set of n-grams for n equal to 1 and 2 in the running example is the set of elements of the form [A(i)] and [A(j), A($j+1$)], respectively. In the general case, the set of n-grams extracted from A will be of the form [A(i), A($i+1$), ..., A($i+n-1$)] for $i = 0, 1, 2, ..., \texttt{A.size} - n$. The n-gram window is an interval of integers for which you want to extract grams for each element in the interval. nMin and nMax are the smallest and largest integer values in the interval, respectively. The default model only includes unigrams and bigrams ($n = 1$ and $n = 2$, respectively).

The n-gram extraction and counting procedure is carried out by the private method hash, which, given a document, returns a Map with keys, n-grams, and values, the number of times each n-gram is extracted from the document. OpenNLP's NGramModel class is used to extract n-grams.

The next step is, once all of the observations have been hashed, to collect all n-grams and compute their corresponding t.f.-i.d.f. value. The t.f.-i.d.f. transformation is a transformation that helps to give less weight to those n-grams that appear with high frequency across all documents, and vice versa. This helps to leverage the predictive power of those words that appear rarely, but can make a big difference in the categorization of a given article. The private method createUniverse outputs an RDD of pairs, where an n-gram is matched with it's i.d.f. value. This RDD is collected as a HashMap (this will be used in future RDD computations so that this object should be serializable), and then a second HashMap is created with n-grams associated to indices. This gives a global index to each n-gram and ensures that each document observation is vectorized in the same manner.

The last two functions that will be mentioned are the methods you will actually use for the data transformation. The method transform takes a document and outputs a sparse vector (MLLib implementation). The transformData simply transforms the TrainingData input (a corpus of documents) into a set of vectors that can now be used for training. The method transform is used both to transform the training data and future queries. It is important to note that using all 11,314 news article observations without any pre-processing of the documents results in over 1 million unigram and bigram features, so that a sparse vector representation is necessary to save some serious computation time.

Training The Model

This section will guide you through the two Training Model implementations that come with this engine template. Recall that the Training Model abstraction refers to an arbitrary set Scala Class that outputs a predictive model (i.e. implements some method that can be used for prediction). The general problem this engine template is tackling is text classification, so that our domain is restricted to implementations producing classifiers. In particular, the two classification models that are implemented in this engine template are based on Multinomial Naive Bayes and Latent Dirichlet Allocation using t.f.-i.d.f. vectorized text.

Algorithm Component

Each of Training Model implementations in this engine are accompanied by a corresponding algorithm component. In particular, NBModel accompanies NBAlgorithm, and LDAModel, LDAAlgorithm. The Training Model abstraction allows you to take care of the modeling implementation details in the model classes. Again this demonstrates how easy it is to incorporate different modeling choices into the DASE architecture.

Each algorithm component then follows a very general form. Firstly, a parameter class must again be initialized to feed in the corresponding Algorithm model parameters. For example, NBAlgorithm incorporates NBAlgorithmParams which holds the appropriate additive smoothing parameter lambda for MLLib's NaiveBayesModel.

The main class of interest in this component is the class that extends P2LAlgorithm. This class must implement a method named train which will output your Training Model implementation. It must also implement a predict method that takes in a query and your model, and uses the fitted model to classify the query text. The vectorization function is implemented by a PreparedData object, and the categorization (prediction) is handled model. Again, this demonstrates the facility with which different models can be incorporated into PredictionIO's DASE architecture.

Turn your attention to the TextManipulationEngine object defined in the script `Engine.scala`. You can see here that the engine is initialized by specifying the DataSource, Preparator, and Serving classes, as well as a Map of algorithm names to Algorithm classes. This tells the engine which algorithms to run. In practice, you can have as many as you would like, you simply have to implement a new Training Model and Training Algorithm pair.

Naive Bayes Classification

This Training Model class only uses the Multinomial Naive Bayes implementation found in the Spark MLLib library. However, recall that the predicted results required in the specifications listed in the overview are of the form:

{`category: String, confidence: Double`}.

The confidence value should really be interpreted as the probability that a document belongs to a category given its vectorized form. Note that MLLib's Naive Bayes model has the class members pi (π) and theta(θ). π is a vector of log prior class probabilities, which really shows your prior beliefs regarding the probability that an arbitrary document belongs in a category. θ is a CxD matrix, where C is the number of classes and D, the number of features, giving the log probabilities that parametrize the Multinomial likelihood model assumed for each class. The multinomial model is easiest to think about as a problem of throwing balls into bins according to some probability distribution. The model treats each n-gram as a bin, and the corresponding t.f.-i.d.f. value as the number of balls thrown into it.

Letting $\mathbf{x}$ be a vectorized text document, then it can be shown that the vector

$$\frac{\exp\left(\pi + \theta\mathbf{x}\right)}{||\exp\left(\pi + \theta\mathbf{x}\right)||}$$

is a vector with C components that represent the posterior class membership probabilities for the document given $\mathbf{x}$. That is, our belief regarding what category this document belongs to after observing its vectorized form. This is the motivation behind defining the class NBModel which uses Spark MLLib's NaiveBayesModel, but implements a separate prediction method. The private methods innerProduct and getScores are implemented to do the computation above.

Once you have a vector of class probabilities, you can classify the text document to the category with highest posterior probability, and, finally, return both the category as well as the probability of belonging to that category (i.e. the confidence in the prediction). This is implemented in the method predict.

LDA Based Classification

This section may get a bit more mathematically intensive, as we will need to do a few derivations to make the model precise. In particular, we will be using Latent Dirichlet Allocation as an unsupervised learning method to first cluster the data into nClust different groups, or topics. The Naive Bayes model from the latter section will then be fit for each cluster. When we go back to make a prediction after observing a new vectorized document $\mathbf{x}$, we will first use the fitted LDA model to extract a probability distribution of the document over the topics, and then use these to classify with a mixture of the fitted Naive Bayes models.

Latent Dirichlet Allocation is implemented in MLLib, and a trained model exists as an object of type LDAModel. This object has a method topicsMatrix which returns a CxD matrix T where C is the total number of unique n-grams and D is equal to nClust, the number of topics. The columns of T give inferred probability distributions (implicitly assuming a multinomial model here) of the topics over the observed unigrams and bigrams. The derivation of this model follows from similar reasoning used to derive the prediction rule presented in Naive Bayes. Letting $\mathbf{x}$ denote a vectorized document, we can interpret the C-dimensional vector

$$\frac{\exp\left[\log(T)^t\mathbf{x}\right]}{||\exp\left[\log(T)^t\mathbf{x}\right]||}$$

as the posterior distribution over the topics given the vectorized document $\mathbf{x}$. Now, for the i^{th} topic we will have a vector of class membership probabilities derived from the i^{th} fitted Naive Bayes model:

$$\frac{\pi_i + \theta_i \mathbf{x}}{||\pi_i + \theta_i \mathbf{x}||}.$$

Letting $\mathbf{Z}(\mathbf{x})$ be the matrix whose i^{th} column vector is the vector above, then it can be shown that the following vector of probabilities is a mixture distribution over the class membership probabilities obtained from the Naive Bayes models fit for each topic:

$$\mathbf{Z}(\mathbf{x}) \left(\frac{\exp\left[\log(T)^t \mathbf{x}\right]}{|| \exp\left[\log(T)^t \mathbf{x}\right] ||} \right).$$

We can now make our prediction using the prediction rule from the latter section.

It now suffices to say how we to cluster the initial N documents. Letting $\mathbf{X}$ be the matrix whose rows are the N vectorized documents. Then the matrix $\exp\left[\mathbf{X}\log(T)\right]$, satisfies the property that each row vector is the (unnormalized) distribution over topics of each respective document. We can now assign topic labels to documents by assigning the column number of the entry corresponding to the document with largest value.

Serving: Delivering the Final Prediction

The serving component is the final stage in the engine, and in a sense, the most important. This is the final stage in which you combine the results obtained from the different models you choose to run. The Serving class extends the LServing class which must implement a method called serve. This takes a query and an associated sequence of predicted results, which contains the predicted results from the different algorithms that are implemented in your engine, and combines the results to yield a final prediction. It is this final prediction that you will receive after sending a query.

For example, you could choose to slightly modify the implementation to return class probabilities coming from a mixture of model estimates for class probabilities, or any other technique you could conceive for combining your results. The default engine setting has this set to yield the label from the model predicting with greater confidence.

Evaluation: Model Assessment and Selection

A predictive model needs to be evaluated to see how it will generalize to future observations. PredictionIO uses cross-validation to perform model performance metric estimates needed to assess your particular choice of model. The script `Evaluation.scala` available with the engine template exemplifies what a usual evaluator setup will look like. First, you must define an appropriate metric. In the engine template, since the topic is text classification, the default metric implemented is category accuracy.

Second you must define an evaluation object (i.e. extends the class Evaluation). Here, you must specify the actual engine and metric components that are to be used for the evaluation. In the engine template, the specified engine is the TextManipulationEngine object, and metric, Accuracy. Lastly, you must specify the parameter values that you want to test in the cross validation. You see in the following block of code:

```
object EngineParamsList extends EngineParamsGenerator {

  // Set data source and preparator parameters.
  private[this] val baseEP = EngineParams(
    dataSourceParams = DataSourceParams(appName = "marco-MyTextApp", evalK = Some(5))
    preparatorParams = PreparatorParams(nMin = 1, nMax = 2)
  )

  // Set the algorithm params for which we will assess an accuracy score.
  engineParamsList = Seq(
    baseEP.copy(algorithmParamsList = Seq(("nb", NBAlgorithmParams(0.5)))),
    baseEP.copy(algorithmParamsList = Seq(("nb", NBAlgorithmParams(1.5)))),
    baseEP.copy(algorithmParamsList = Seq(("nb", NBAlgorithmParams(5))))
  )
}
```

In the latter block of code, only the parameter values that show up in the algorithm stage are assessed. However, it is plausible that you may want to assess parameter values in the Data Model stage. This can easily be done by placing the preparator parameter choices as follows:

```
object EngineParamsList extends EngineParamsGenerator {

  // Set data source and preparator parameters.
  private[this] val baseEP = EngineParams(
    dataSourceParams = DataSourceParams(appName = "marco-MyTextApp", evalK = Some(5))
  )

  // Set the algorithm params for which we will assess an accuracy score.
  engineParamsList = Seq(
    baseEP.copy(preparatorParams = ("preparator", PreparatorParams(nMin = 1, nMax = 2)
      algorithmParamsList = Seq(("nb", NBAlgorithmParams(0.5)))),
    baseEP.copy(preparatorParams = ("preparator", PreparatorParams(nMin = 2, nMax = 3)
      algorithmParamsList = Seq(("nb", NBAlgorithmParams(1.5)))),
    baseEP.copy(preparatorParams = ("preparator", PreparatorParams(nMin = 1, nMax = 3)
      algorithmParamsList = Seq(("nb", NBAlgorithmParams(5))))
  )
}
```

Engine Deployment

Once an engine is ready for deployment it can interact with your web application in real-time. This section will cover how to send and receive queries from your engine, gather more data, and re-training your model with the newly gathered data.

Sending Queries

Recall that one of the greatest advantages of using the PredictionIO platform is that once your engine is deployed, you can respond to queries from your web application in real-time. Recall that our queries are of the form

{`"text" : "..."`}.

To actually send a query you can use our REST API by typing in the following shell command:

```
curl -H "Content-Type: application/json" -d '{ "user": "1", "num": 4 }' \
http://localhost:8000/queries.json}
```

There are a number of SDK's you can use to send your queries and obtain a response. Recall that our predicted response is of the form

{`"category" : "class", "confidence" : 1.0`}

which is what you should see upon inputting the latter command for querying.

Gathering More Data and Retraining Your Model

The importing data section that is included in this tutorial uses a sample data set for illustration purposes, and uses the PredictionIO Python SDK to import the data. However, there are a variety of ways that you can import your collected data (via REST or other SDKs).

As you continue to collect your data, it is quite easy to retrain your model once you actually import your data into the Event Server. You simply repeat the steps listed in the Quick Start guide. We re-list them here again:

1. Build your engine.

   ```
   $ pio build
   ```

2.a. Evaluate your training model and tune parameters.

   ```
   $ pio eval
   ```

2.b. Train your model and deploy.

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