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; 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 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 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 we have trained these models, we are ready to deploy our 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.

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

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

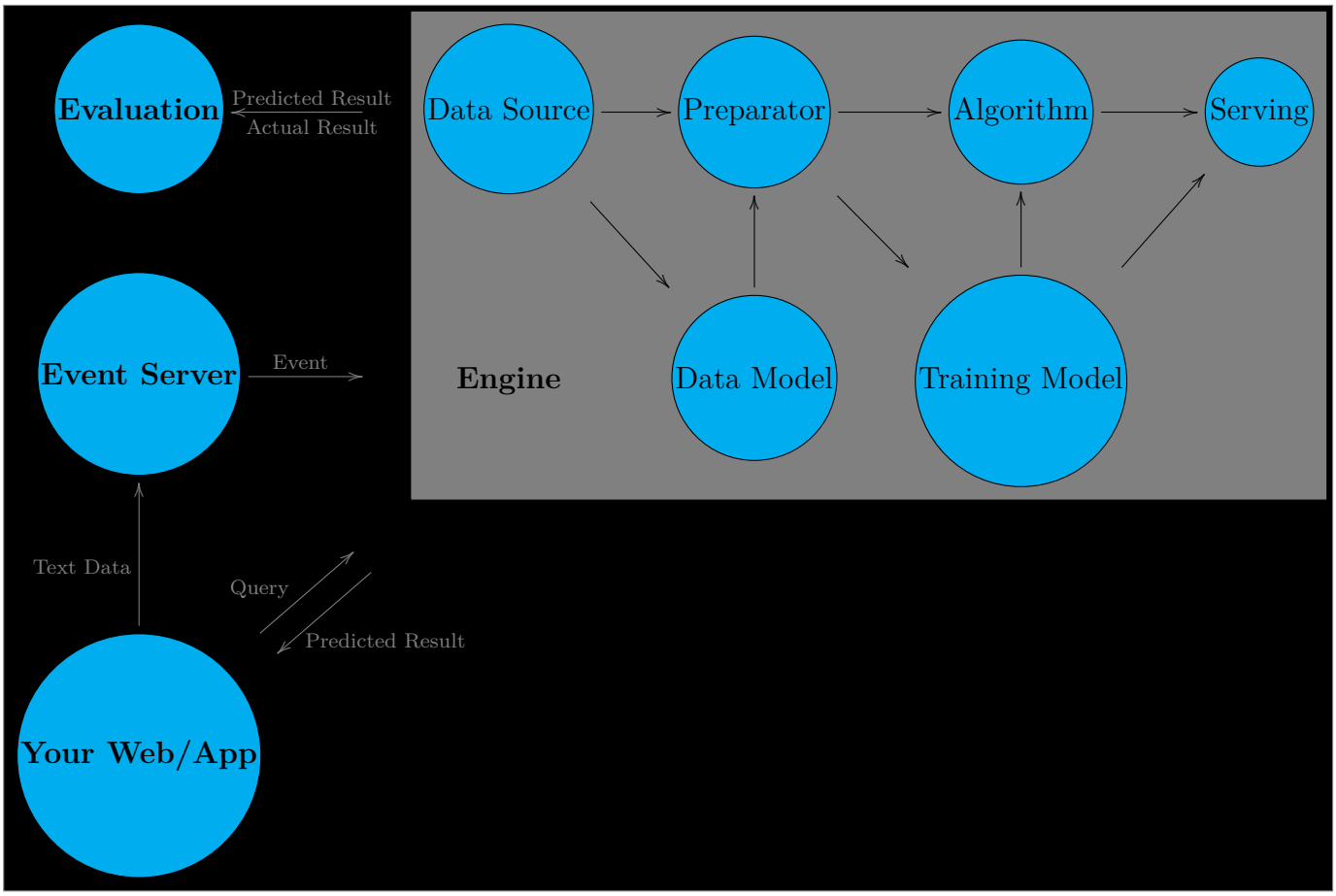
Here category is the model's class assignment for this new text document (i.e. our best guess for this article's categorization), and confidence, a value between 0 and 1 representing our confidence in the category prediction (0 meaning we have no confidence in the prediction). The Actual Result is of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

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

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

In addition to the DASE components, our engine also includes the Data Model and Training Model abstractions. The Data Model refers to the implementation of modeling choices relating to feature extraction and selection. In our example, this includes the vectorization of text and any subsequent feature reduction or transformation procedures one might want to do. The Training Model abstraction refers to any set of Scala classes that take in a set of feature observations and outputs a predictive model. In our example, this includes the NBModel (Naive Bayes) and LDAModel (Latent Dirichlet Allocation) based classes which give us category classification models.

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. 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}
```

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 from the Scikit learn Python library 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 we do not want to include in our corpus when modeling our text data.

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.

Processing the Data

This section deal 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. As you will see, most of the work in the processing takes place in the Data Model stage. This decoupling allows you to focus primarily on development issues regarding your model, and also allows Data Scientists an easy way to incorporate their modeling ideas into the DASE framework.

Data Model : Data Representation

For our Text Manipulation Engine, the Data Model abstraction is implemented as a Scala class taking in the 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 cover the default feature preparation procedure implemented in the engine template. 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 an array "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:

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

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 $\{$`"my"`, `"is"`$\}$, then the list of allowed tokens appearing in D is:

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

We call any function that takes as input a text document and returns an array of allowed tokens a tokenizer. The tokenizer in our Data Model implementation is the private method tokenize. We use the SimpleTokenizer from the OpenNLP 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).

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. The n-gram window is an interval of integers for which we extract grams for each element. nMin and nMax are the smallest and largest integer values in the interval, respectively, for $i = 0, 1, ..., n-1$, and $j = 0, ..., n-2$.

In our implementation, the n-gram extraction and counting procedure is carried out by the private method hash, which returns a Map with keys, n-grams, and values, the number of times each n-gram is extracted from the document. We use OpenNLP's NGramModel class 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 intuitively 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. In our implementation, 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 we need an object that is serializable), and we create a second hash map with n-grams associated to indices. This gives a global index to each n-gram so that each document observation is vectorized in the same manner.

The last two functions we mention 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.

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 is built to do the heavy lifting needed for this data processing. Our Preparator can now simply implement the prepare method which outputs an object of type PreparedData, which in our case serves as a wrapper for an initialized DataModel object. Since our DataModel class requires us to specify the two n-gram window components, we must incorporate the custom class of parameters for our Preparator component, PreparatorParams.

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.

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 Training Model abstraction 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 Your training model classes is accompanied by a corresponding algorithm component. For example, NBModel accompanies NBAlgorithm, and LDAModel, LDAAlgorithm. The implementation details are largely simplified by the Training Model abstraction, since you take care of the modeling implementation details in the model classes. Again this demonstrates how easy it is to incorporate modeling choices into the DASE architecture.

Firstly, we must again initialize a parameter class 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 a Training Model implementation. It must also implement a predict method that vectorizes an object and predicts using the trained model. The vectorization function is implemented by the Data Model, and the categorization (prediction) is handled mainly by the TrainingModel. 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`. We 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, and, in practice, you could have as many as you would like. You simply have to implement a new Training Model and Training Algorithm pair.

Naive Bayes Classification

We will be using the Multinomial Naive Bayes implementation found in the Spark MLLib library. However, recall that the predicted results required in the specs listed in our overview must be of the form.

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

We interpret here the confidence value as the probability that a document belongs to a category given the vectorized data. Now, 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, and θ is a CxD matrix, where C is the number of classes and D, the number of features, giving the log probabilities that parametrize a Multinomial model. Say we are given a document and vectorize it. Letting $\mathbf{x}$ denote this vectorized form, then it can be shown that the vector

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

is a vector with C components that represent the posterior class probabilities given $\mathbf{x}$. This is essentially the motivation behind defining the class NBModel, which initially uses MLLib's NaiveBayesModel in the implementation. The private methods innerProduct and getScores are implemented to do the computation above. Once we have the vector of class probabilities, we categorize the text document to the class with highest posterior probability and both the category as well as the probability of belonging to that category (i.e. our 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 of the topics over the observed bi-grams. 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

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

as the distribution over the topics of 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 $\mathbf{Z}$ to 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:

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

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 our 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. 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. For example, the default engine setting has this set to yield the label from the model predicting with greater confidence.

Evaluation

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))))
  )
}
```

Here we are only assessing the parameter values that show up in the algorithm stage. However, it is plausible that the modeler 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))))
  )
}
```