

# PredictionIO: Modeling Text Data Engine

The purpose of this tutorial is to illustrate how to model text data using PredictionIO's engine platform. We will be largely mirroring the tutorial provided for Scikit's Learn package title "Working With Text Data." However, we will be providing instructions for building the model using PredictionIO's engine platform. The advantages of using this framework include the usage of Apache Spark for distributed computation which is crucial for computation with large data sets, as well as the capacity to use a newly trained predictive model to respond to queries in real-time.

We will assume the user is running the PredictionIO version 0.9.2, and meets all minimum computing requirements. To download PredictionIO, follow the instructions on the Getting Started tutorial.

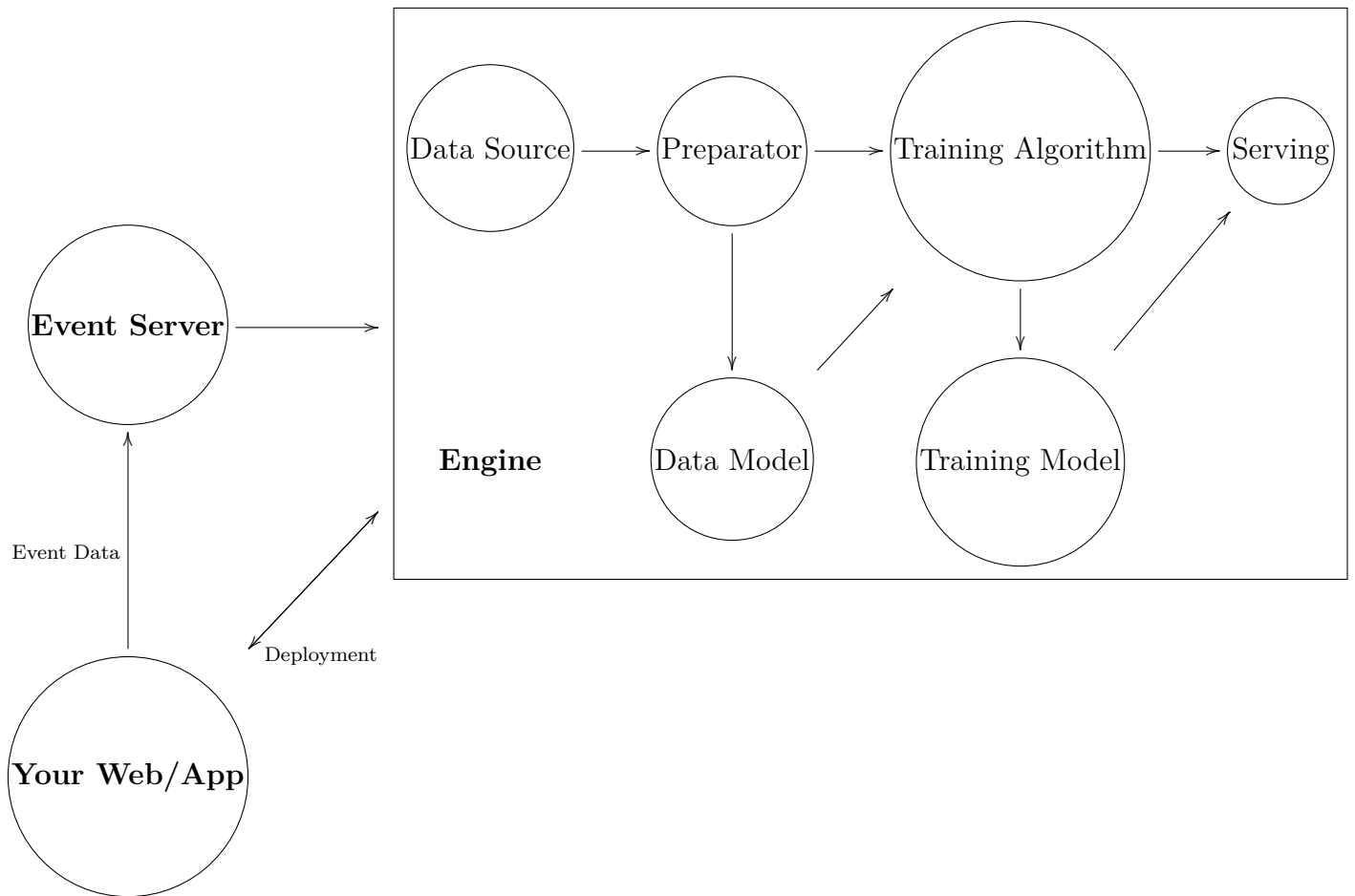
## Engine Overview

As a user, we are charged with collecting data and importing it into an event server. The data is then read and processed by our engine via the **DataSource** and **Preparation** components. 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 problem we will solve using PredictionIO's framework is the following: given a set of documents represented as strings, we want to build a predictive model that either (i), given a categorization of our data documents, classifies new documents into a particular category, or (ii) groups the documents into different classes based on the document content, or . To do this we will transform our strings into numeric vectors using Apache's OpenNLP library so that we may apply existing machine learning methods for (i) classification and (ii) clustering.

Our engine includes all DASE components, a Data Model implementation which deals with data representation, and a Training Model component which is more of a conceptual framework representing a set of Scala classes that can produce a predictive model. Figure 1 below demonstrates the general structure of our engine, as well as its interactions with your web/app and a provided Event Server. We emphasize here that the engine template itself is really just an SBT project.

We begin by covering the data collection stage in which we will obtain a corpus of text documents from Scikit Learn's datasets module. For this purpose, we will use the PredictionIO Python SDK for importing our documents as events into PredictionIO's event server. You can pull the necessary SDK files from PredictionIO's Github repository, if required.



**Figure 1:** Representation of PredictionIO framework for our Modeling Text Data Engine as a directed acyclic graph.

## Importing Data

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. The definition of stop words will be made precise later in the tutorial, however, for now just think of these as words that we do not want to include in our corpus when modeling the text data. Also, for the remainder of the tutorial we will assume that the present working directory is the engine template root directory.

The script in the used to import the data is named `import_eventserver.py`. The script begins with the following import statements:

```

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction import text
import predictionio
import argparse

```

The first line imports a function `fetch_20newsgroups` which is used to fetch the text data; the second, imports an object `text` containing a list of English stop words as an attribute, and, the last two lines import the `predictionio` and `argparse` Python modules, respectively. The following lines bind the data and stop words to the respective variables `twenty_train` and `stop_words`.

```
twenty_train = fetch_20newsgroups(subset = 'train',
                                  shuffle=True,
                                  random_state=10)
stop_words = text.ENGLISH_STOP_WORDS
```

You can run the last few lines of code interactively in your Python interpreter to get a better feel for the data, and in particular, note that `twenty_train` and `stop_words` are implemented as a dictionary and a frozen set, respectively. This is shown in the following output:

```
>>> twenty_train.keys()
dict_keys(['DESCR', 'target', 'data', 'target_names', 'filenames'])
>>> type(stop_words)
<class 'frozenset'>
```

We are interested in the `target`, `target_names`, and `data` keys of `twenty_train`. The `target` attribute is a list of length equal to the number of observations (11,314) consisting of class labels corresponding to the document's given category; the `target_names`, a list of strength of length 20 which give the corresponding category names; and the `data` attribute, a list containing our document strings. We note that you can take any sublist of `twenty_train.target_names` and use this to specify the optional parameter `categories` for the function `fetch_20newsgroups`. This will also return a dictionary of the same form, except it will only include those documents whose class label corresponds to one of the selected categories.

The corpus itself is imported into PredictionIO's Event Server via the function:

```
def import_events(client):
    train = ((twenty_train.target_names[k],
               float(twenty_train.target[k]),
               twenty_train.data[k])
              for k in range(len(twenty_train.data)))
    count = 0
    print('Importing data.....')
    for elem in train:
        count += 1
        client.create_event(
```

```

        event = "documents",
        entity_id = count,
        entity_type = "source",
        properties = {
            "category": elem[0],
            "label": elem[1],
            "text": elem[2]
        })
    print("Imported {0} events.".format(count))

```

The parameter `client` is an instance of the class `predictionio.EventClient`. The first line in the body of the function binds a generator consisting of corresponding document-class-label tuples to the variable `train`. We also initialize a counter in order to report the number of events that are imported. The for loop iterates over the generator `train`, and creates an event using the given data. The parameters `event`, `entity_id`, `entity_type` help identify a given event when reading it in. The properties are the relevant parts of the data that we want to use for training and returning predictions.

The subsequently defined function `import_stopwords` works similarly. However, since this data is used separately from the document corpus, we must be sure to modify the `event_type` field from the one specified in `import_events`.

Our main tool from the PredictionIO Python SDK that we will use to import our data into the event server is the class `EventClient` and its method `create_event`. The function that implements this in the script `MyTextEngine/data/import_eventserver.py` with respect to the `20newsgroups` dataset is `import_events`. As fore mentioned, we will also be importing Scikit's library of English stop words into PredictionIO's Event Server. This is locally stored as a frozen set of strings, as can be verified by typing the following into your Python interpreter:

```

>>> from sklearn.feature_extraction import text
>>> text.ENGLISH_STOP_WORDS

```

Again, the class `EventClient` is the SDK component that allows us import the stop word data into the PredictionIO event server. The function used to import the stop word events is `import_stopwords` which is defined in the same script. With these two functions under our belt, we are ready to begin importing data. First, make sure PredictionIO is running, you can check this by typing `pio status` on your terminal. If it is not running, start it with the command `pio-start-all`. Now, let's create a new application named `MyTextApp` by entering the command `pio app new MyTextApp` in your shell. You should see some printed output which includes your newly created access key. At this point make sure you are in the template root directory, and import the data set using the following command:

```
python data/import_eventserver.py --access_key *****
```

where you will replace \*\*\*\*\* with your actual access key. If the data is successfully imported, you should see the following output:

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

Our data is now sitting in PredictionIO's Event Server and ready to be used by our engine. The following section will get you acquainted with the different engine components that are implemented in this engine template.

# Engine Components

## Data Source

## Preparator

## 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:

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

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:

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

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

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