# 10 Text Mining at Scale

In this chapter, we will go back to some of the libraries we learned about in the previous chapters, but this time, we want to learn to learn how these libraries will scale up with bigdata. We assume that you have a fair bit of an idea about big data, **Hadoop** and **Hive**. We will explore how some of the Python libraries, such as NLTK, scikit-learn, and pandas can be used on a Hadoop cluster with a large amount of unstructured data.

We will cover some of the most common use cases in the context of NLP and text mining, and we will also provide a code snippet that will be helpful for you to get your job done. We will look at three major examples that can capture the vast majority of your text mining problems. We will tell you how to run NLTK at scale to perform some of the NLP tasks that we completed in the initial chapters. We will give you a few examples of some of the text classification tasks that can be done on Big Data.

One other aspect of doing machine learning and NLP at a very high scale is to understand whether the problem is parallelizable or not. We will talk in brief about some of the problems discussed in the previous chapter, and whether these problems are big data problems or not. Or in some case is it even possible to solve this using Big Data.

Since most of the libraries we learned so far are written in Python, let's deal with one of the main questions of how to get Python on Big Data (Hadoop).

By end of the chapter we like reader to have :

* Good understanding about big data related technologies such as Hadoop, Hive and how it can be done using python.
* Step by step tutorial to work with NLTK, Scikit & PySpark on Big Data.

## Different ways of using Python on Hadoop

There are many ways to run a Python process on Hadoop. We will talk about some of the most popular ways through which we can run Python on Hadoop as a streaming MapReduce job, Python UDF in Hive, and Python hadoop wrappers.

### Python streaming

Typically a Hadoop job has to be written in form of a map and reduce function.

User has to write an implementation of map and reduce function for the given task. Commonly these mappers and reducers are implemented in JAVA. At the same time Hadoop provide streaming, you where a user can write a Python mapper and reducer function similar to Java in any other language. I am assuming that you have run a word count example using Python. We will also use the same example using NLTK later in this chapter.

In case you have not, have a look at

[http://www.michael-noll.com/tutorials/writingan-hadoop-mapreduce-program-in-python/](http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/) to know more about MapReduce in Python.

### Hive/Pig UDF

Other way to use Python is by writing a **UDF** (**User Defined Function**) in Hive/Pig. The idea here is that most of the operations we are performing in NLTK are highly parallelizable. For example, POS tagging, Tokenization, Lemmatization, Stop Word removal, and NER can be highly distributable. The reason being the content of each row is independent from the other row, and we don't need any context while doing some of these operations.

So, if we have NLTK and other Python libraries on each node of the cluster, we can write a **user defined function** (**UDF**) in Python, using libraries such as NLTK and scikit. This is one of the easiest way of doing NLTK, especially for scikit on a large scale. We will give you a glimpse of both of these in this chapter.

### Streaming wrappers

There is a long list of wrappers that different organizations have implemented to get Python running on the cluster. Some of them are actually quite easy to use, but all of them suffer from performance bias. I have listed some of them as follows, but you can go through the project page in case you want to know more about them:

* Hadoopy
* Pydoop
* Dumbo
* mrjob

For the exhaustive list of options available for the usage of Python on

Hadoop, go through the article at

[http://blog.cloudera.com/blog/2013/01/a-guide-topython-frameworks-for-hadoop/](http://blog.cloudera.com/blog/2013/01/a-guide-to-python-frameworks-for-hadoop/).

## NLTK on Hadoop

We talked enough about NLTK as a library, and what are some of the most-used functions it gives us. Now, NLTK can solve many NLP problems from which many are highly parallelizable. This is the reason why we will try to use NLTK on Hadoop.

The best way of running NLTK on Hadoop is to get it installed on all the nodes of the cluster. This is probably not that difficult to achieve. There are ways in which you can do this, such as sending the resource files as a streaming argument. However, we will rather prefer the first option.

### A UDF

There are a variety of ways in which we can make NLTK run on Hadoop. Let's talk about one example of using NLTK by doing tokenization in parallel using a Hive UDF.

For this use case, we have to follow these steps:

1. We have chosen a small dataset where only two columns exist. We have to create the same schema in Hive:

|  |  |
| --- | --- |
| **ID** | **Content** |
| UA0001 | "I tried calling you. The service was not up to the mark" |
| UA0002 | "Can you please update my phone no" |
| UA0003 | "Really bad experience" |
| UA0004 | "I am looking for an iPhone" |

1. Create the same schema in Hive. The following Hive script will do this for you:

Hive script

**CREATE TABLE $InputTableName (**

**ID String,**

**Content String**

**)**

**ROW FORMAT DELIMITED**

**FIELDS TERMINATED BY '\t';**

1. Once we have the schema, essentially, we want to get something like tokens of the content in a separate column. So, we just want another column in the $outTable with the same schema, and the added column of tokens: Hive script

**CREATE TABLE $OutTableName (**

**ID String,**

**Content String,**

**Tokens String**

**)**

1. Now, we have the schemas ready. We have to write the UDF in Python to read the table line by line and then apply a tokenize method. This is very similar to what we did in *Chapter 3*, *Part of Speech Tagging*. This is the piece of function that is analogous to all the examples in *Chapter 3*, *Part of Speech Tagging*. Now, if you want to get POS tags, Lemmatization, and HTML, you just need to modify this UDF. Let's see how the UDF will look for our tokenizer:

**>>>import sys**

**>>>import datetime**

**>>>import pickle**

**>>>import nltk**

**>>>nltk.download('punkt') >>>for line in sys.stdin:**

**>>> line = line.strip()**

**>>> print>>sys.stderr, line**

**>>> id, content= line.split('\t')**

**>>> print>>sys.stderr,tok.tokenize(content)**

**>>> tokens =nltk.word\_tokenize(concat\_all\_text)**

**>>> print '\t'.join([id,content,tokens])**

1. Just name this UDF something like: nltk\_scoring.py.
2. Now, we have to run the insert hive query with the TRANSFORM function to apply the UDF on the given content and to do tokenization and dump the tokens in the new column: Hive script

**add FILE nltk\_scoring.py;**

**add FILE english.pickle; #Adding file to DistributedCache INSERT OVERWRITE TABLE $OutTableName**

**SELECT**

**TRANSFORM (id, content)**

**USING 'PYTHONPATH nltk\_scoring.py'**

**AS (id string, content string, tokens string )**

**FROM $InputTablename;**

1. If you are getting an error like this, you have not installed the NLTK and NLTK data correctly:

**raiseLookupError(resource\_not\_found) LookupError:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\* Resource u'tokenizers/punkt/english.pickle' not found. Please use the NLTK Downloader to obtain the resource: >>> nltk.download() Searched in:**

* + **'/home/nltk\_data'**
  + **'/usr/share/nltk\_data'**
  + **'/usr/local/share/nltk\_data'**
  + **'/usr/lib/nltk\_data'**
  + **'/usr/local/lib/nltk\_data'**

1. If you are able to run this Hive job successfully, you will get a table named OutTableName, that will look something like this:

|  |  |  |
| --- | --- | --- |
| **ID** | **Content** |  |
| UA0001 | "I tried calling you, The service was not up to the mark" | [" I", " tried", "calling", "you", "The",  "service" "was", "not", "up", "to", "the", "mark"] |
| UA0002 | "Can you please update my phone no" | ["Can", "you", "please" "update", " my",  "phone" "no"] |
| UA0003 | "Really bad experience" | ["Really"," bad" "experience"] |
| UA0004 | "I am looking for an iphone" | ["I", "am", "looking", "for", "an", "iPhone"] |

### Python streaming

Let's try the second option of Python streaming. We have Hadoop streaming, where we can write our own mapper and reducer functions, and then use Python streaming with mapper.py, as it looks quite similar to our Hive UDF. Here we are using the same example with map-reduce python streaming this will give us a option of choosing a Hive table or using a HDFS file directly. We will just go over the content of the file and tokenize it. We will not perform any reduce operation here, but for learning, I included a dummy reducer, which just dumps it. So now, we can ignore the reducer from the execution command completely. Here is the code for the Mapper.py:

Mapper.py

**>>>import sys**

**>>>import pickle**

**>>>import nltk >>>for line in sys.stdin:**

**>>> line = line.strip()**

**>>> id, content = line.split('\t')**

**>>> tokens =nltk.word\_tokenize(concat\_all\_text)**

**>>> print '\t'.join([id,content,topics])**

Here is the code for the Reducer.py:

Reducer.py

**>>>import sys**

**>>>import pickle**

**>>>import nltk >>>for line in sys.stdin:**

**>>> line = line.strip()**

**>>> id, content,tokens = line.split('\t')**

**>>> print '\t'.join([id,content,tokens])**

The following is the Hadoop command to execute a Python stream:Hive script

**hadoop jar <path>/hadoop-streaming.jar \ -D mapred.reduce.tasks=1 -file <path>/mapper.py \**

**-mapper <path>/mapper.py \**

**-file <path>/reducer.py \**

**-reducer <path>/reducer.py \**

**-input /hdfspath/infile \**

**-output outfile**

## Scikit-learn on Hadoop

The other important use case for big data is machine learning. Specially with Hadoop, scikit-learn is more important, as this is one of the best options we have to score a machine learning model on big data. Large-scale machine learning is currently one of the hottest topics, and doing this in a big data environment such as Hadoop is all the more important. Now, the two aspects of machine learning models are building a model on big data and to build model on a significantly large amount of data and scoring a significantly large amount of data.

To understand more, let's take the same example data we used in the previous table, where we have some customer comments. Now, we can build, let's say, a text classification mode using a significant training sample, and use some of the learnings from *Chapter 6*, *Text Classification* to build a Naive Bayes, SVM, or a logistic regression model on the data. While scoring, we might need to score a huge amount of data, such as customer comments. On the other hand building the model itself on big data is not possible with scikit-learn, we will require tool like spark/Mahot for that. We will take the same step-by-step approach of scoring using a pre-trained model as we did with NLTK. While building the mode on big data will be covered in the next section. For scoring using a pre-trained model specifically when we are working on a text mining kind of problem. We need two main objects (a vectorizer and modelclassifier) to be stored as a serialized pickle object.

Here, pickle is a Python module to achieve serialization by which the object will be saved in a binary state on the disk and can be consumed by loading again.

<https://docs.python.org/2/library/pickle.html>

Build an offline model using scikit on your local machine and make sure you pickle objects. For example, if I use the Naive Bayes example from *Chapter 6*, *Text Classification*, we need to store vectorizer and clf as pickle objects:

**>>>vectorizer = TfidfVectorizer(sublinear\_tf=True, min\_df=in\_min\_df, stop\_words='english', ngram\_range=(1,2), max\_df=in\_max\_df) >>>joblib.dump(vectorizer, "vectorizer.pkl", compress=3)**

**>>>clf = GaussianNB().fit(X\_train,y\_train)**

**>>>joblib.dump(clf, "classifier.pkl")**

The following are the steps for creating a output table which will have all the customer comments for the entire history:

1. Create the same schema in Hive as we did in the previous example. The following Hive script will do this for you. This table can be huge; in our case, let's assume that it contains all the customer comments about the company in the past:

Hive script

**CREATE TABLE $InputTableName (**

**ID String,**

**Content String**

**)**

**ROW FORMAT DELIMITED**

**FIELDS TERMINATED BY '\t';**

1. Build an output table with the output column like the predict and probability score:

Hive script

**CREATE TABLE $OutTableName (**

**ID String, Content String, predict String, predict\_score double )**

1. Now, we have to load these pickle objects to the distributed cache using the addFILE command in Hive:

**add FILE vectorizer.pkl; add FILE classifier.pkl;**

1. The next step is to write the Hive UDF, where we are loading these pickle objects. Now, they start behaving the same as they were on the local. Once we have the classifier and vectorizer object, we can use our test sample, which is nothing but a string, and generate the TFIDF vector out of this. The vectorizer object can be used now to predict the class as well as the probability of the class:

Classification.py

**>>>import sys**

**>>>import pickle**

**>>>import sklearn**

**>>>from sklearn.externals import joblib**

**>>>clf = joblib.load('classifier.pkl')**

**>>>vectorizer = joblib.load('vectorizer.pkl')**

**>>>for line in sys.stdin:**

**>>> line = line.strip()**

**>>> id, content= line.split('\t')**

**>>> X\_test = vectorizer.transform([str(content)])**

**>>> prob = clf.predict\_proba(X\_test)**

**>>> pred = clf.predict(X\_test)**

**>>> prob\_score =prob[:,1]**

**>>> print '\t'.join([id, content,pred,prob\_score])**

1. Once we have written the classification.py UDF, we have to also add this UDF to the distributed cache and then effectively, run this UDF as a TRANSFORM function on each and every row of the table. The Hive script for this will look like this:

Hive script

**add FILE classification.py;**

**INSERT OVERWRITE TABLE $OutTableName**

**SELECT**

**TRANSFORM (id, content)**

**USING 'python2.7 classification.py'**

**AS (id string, scorestringscore string )**

**FROM $Tablename;**

1. If everything goes well, then we will have the output table with the output schema as:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Content** | **Predict** | **Prob\_score** |
| UA0001 | "I tried calling you, The service was not up to the mark" | Complaint | 0.98 |
| UA0002 | "Can you please update my phone no " | No | 0.23 |
| UA0003 | "Really bad experience" | Complaint | 0..97 |
| UA0004 | "I am looking for an iPhone " | No | 0.01 |

So, our output table will have all the customer comments for the entire history, scores for whether they were complaints or not, and also a confidence score. We have choosen a Hive UDF for our example, but the similar process can be done through the Pig and Python steaming in a similar way as we did in NLTK.

This example was to give you a hands-on experience of how to score a machine learning model on Hive. In the next example, we will talk about how to build a machine learning/NLP model on big data.

## PySpark

Let's go back to the same discussion we had of building a machine learning/NLP model on Hadoop and the other where we score a ML model on Hadoop. We discussed second option of scoring in depth in the last section. Instead sampling a smaller data-set and scoring let’s use a larger data-set and build a large-scale machine learning model step-by-step using PySpark. I am again using the same running data with the same schema:

|  |  |  |
| --- | --- | --- |
| **ID** | **Comment** | **Class** |
| UA0001 | I tried calling you, The service was not up to the mark | 1 |
| UA0002 | Can you please update my phone no | 0 |
| UA0003 | Really bad experience | 1 |
| UA0004 | I am looking for an iPhone | 0 |
| UA0005 | Can somebody help me with my password | 1 |
| UA0006 | Thanks for considering my request for | 0 |

Consider the schema for last 10 years worth of comments of the organization. Now, instead of using a small sample to build a classification model, and then using a pretrained model to score all the comments, let me give you a step-by-step example of how to build a text classification model using PySpark.

The first thing that we need to do is we need to import some of the modules. Starting with SparkContext, which is more of a configuration, you can provide more parameters, such as app names and others for this.

**>>>from pyspark import SparkContext**

**>>>sc = SparkContext(appName="comment\_classifcation")**

For more information, go through the article at

[http://spark.apache.org/docs/0.7.3/api/pyspark/ pyspark.context.SparkContext-class.html](http://spark.apache.org/docs/0.7.3/api/pyspark/pyspark.context.SparkContext-class.html).

The next thing is reading a tab delimited text file. Reading the file should be on HDFS. This file could be huge (~Tb/Pb):

**>>>lines = sc.textFile("testcomments.txt")**

The lines are now a list of all the rows in the corpus:

**>>>parts = lines.map(lambda l: l.split("\t"))**

**>>>corpus = parts.map(lambda row: Row(id=row[0], comment=row[1], class=row[2]))**

The part is a list of fields as we have each field in the line delimited on "\t".

Let's break the corpus that has [ID, comment, class (0,1)] in the different RDD objects:

**>>>comment = corpus.map(lambda row: " " + row.comment)**

**>>>class\_var = corpus.map(lambda row:row.class)**

Once we have the comments, we need to do a process very similar to what we did in *Chapter 6*, *Text Classification*, where we used scikit to do tokenization, hash vectorizer and calculate TF, IDF, and tf-idf using a vectorizer.

The following is the snippet of how to create tokenization, term frequency, and inverse document frequency:

**>>>from pyspark.mllib.feature import HashingTF**

**>>>from pyspark.mllib.feature import IDF**

**# https://spark.apache.org/docs/1.2.0/mllib-feature-extraction.html**

**>>>comment\_tokenized = comment.map(lambda line: line.strip().split(" "))**

**>>>hashingTF = HashingTF(1000) # to select only 1000 features**

**>>>comment\_tf = hashingTF.transform(comment\_tokenized)**

**>>>comment\_idf = IDF().fit(comment\_tf)**

**>>>comment\_tfidf = comment\_idf.transform(comment\_tf)** We will merge the class with the tfidf RDD like this:

**>>>finaldata = class\_var.zip(comment\_tfidf)** We will do a typical test, and train sampling:

**>>>train, test = finaldata.randomSplit([0.8, 0.2], seed=0)**

Let's perform the main classification commands, which are quite similar to scikit. We are using a logistic regression, which is widely used classifier. The pyspark.mllib provides you with a variety of algorithms.

For more information on pyspark.mllib visit [https://spark. apache.org/docs/latest/api/python/pyspark.mllib.html](https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html)

The following is an example of Naive bayes classifier:

**>>>from pyspark.mllib.regression import LabeledPoint**

**>>>from pyspark.mllib.classification import NaiveBayes**

**>>>train\_rdd = train.map(lambda t: LabeledPoint(t[0], t[1]))**

**>>>test\_rdd = test.map(lambda t: LabeledPoint(t[0], t[1]))**

**>>>nb = NaiveBayes.train(train\_rdd,lambda = 1.0)**

**>>>nb\_output = test\_rdd.map(lambda point: (NB.predict(point.features), point.label)) >>>print nb\_output**

The nb\_output command contains the final predictions for the test sample. The great thing to understand is that with just less than 50 lines, we built a snippet code for an industry-standard text classification with even petabytes of the training sample.

## Summary

To summarize this chapter, our objective was to apply the concepts that we learned so far in the context of big data. In this chapter, you learned how to use some Python libraries, such as NLTK and scikit with Hadoop. We talked about scoring a machine learning model, or an NLP-based operation.

We also saw three major examples of the most-common use cases. On understanding these examples, you can apply most of the NLTK, scikit and PySpark functions.

This chapter was a quick and brief introduction to NLP and text mining on big data. This is one of the hottest topics, and each term and tool which I talked about in the example snippet could be a book in itself. I tried to give you a hacker's approach, to give you an introduction to big data and text mining on a large scale. I encourage you to read more about some of these big data technologies such as Hadoop, Hive, Pig, and Spark and try to explore some of the examples we gave in this chapter.