# 第10章 **大规模文本挖掘**

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.

在本章中，我们将回到我们在前面章节中学到的一些库，但是这一次，我们想学习如何使用bigdata来扩展这些库。我们假设你有一个关于大数据，Hadoop和Hive的想法。我们将探讨一些Python库，例如NLTK，scikit-learn和pandas可以用在具有大量非结构化数据的Hadoop集群上。

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.

我们将讨论在NLP和文本挖掘的上下文中的一些最常见的用例，我们还将提供一个代码片段，将有助于您完成工作。我们将看看可以捕捉绝大多数文本挖掘问题的三个主要示例。我们将告诉你如何运行NLTK大规模执行一些我们在最初的章节中完成的NLP任务。我们将给你一些可以对大数据进行的文本分类任务的几个例子。

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.

以非常高的规模进行机器学习和NLP的另一方面是理解问题是否是可并行化的。我们将简要地讨论前一章讨论的一些问题，以及这些问题是否是大数据问题。或者在某些情况下，甚至可以使用大数据解决这个问题。

因为到目前为止我们学习的大多数库都是用Python编写的，所以我们来处理一个关于如何获取Python上的大数据（Hadoop）的主要问题之一。

通过本章结束，我们喜欢读者有：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).

因为到目前为止我们学习的大多数库都是用Python编写的，所以我们来处理一个关于如何获取Python上的大数据（Hadoop）的主要问题之一。

By end of the chapter we like reader to have :

通过本章结束，我们喜欢读者有：

* 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.
* 良好的了解大数据相关技术，如Hadoop，Hive和如何使用python。
* 一步一步的教程与NLTK，Scikit和PySpark在大数据上合作。

## 在Hadoop上使用Python的不同方式

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.

有很多方法可以在Hadoop上运行Python进程。 我们将讨论一些最流行的方式，通过它们我们可以在Hadoop上运行Python作为流式MapReduce作业，Hive中的Python UDF和Python hadoop包装器。

### Python的流操作

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.

通常，Hadoop作业必须以map和reduce函数的形式写入。用户必须为给定任务编写map和reduce函数的实现。通常，这些映射器和reducer在JAVA中实现。同时Hadoop提供流式处理，你在哪里用户可以写任何其他语言的Java映射器和reducer函数类似于Java。我假设你使用Python运行一个字数示例。我们也将在本章后面使用相同的示例使用NLTK。

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.

通常，Hadoop作业必须以map和reduce函数的形式写入。用户必须为给定任务编写map和reduce函数的实现。通常，这些映射器和reducer在JAVA中实现。同时Hadoop提供流式处理，你在哪里用户可以写任何其他语言的Java映射器和reducer函数类似于Java。我假设你使用Python运行一个字数示例。我们也将在本章后面使用相同的示例使用NLTK。

|  |
| --- |
| In case you have not, have a look at  http://www.michael-noll.com/tutorials/writingan-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.

使用Python的其他方法是在Hive / Pig中编写UDF（用户定义函数）。 这里的想法是，我们在NLTK中执行的大多数操作是高度可并行化的。 例如，POS标记，令牌化，语音化，停止词删除和NER可以高度可分发。 原因是每行的内容独立于其他行，并且我们在执行这些操作时不需要任何上下文。

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.

因此，如果我们在集群的每个节点上都有NLTK和其他Python库，我们可以使用Python中的用户定义函数（UDF），使用NLTK和scikit等库。 这是执行NLTK的最简单的方法之一，特别是对于大规模的scikit。 我们将在本章中向您瞥见这两个。

### 流操作封装器

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:

有一个很长的包装器列表，不同的组织已经实现，以使Python在集群上运行。 其中一些实际上很容易使用，但是他们都遭受性能偏差。 我列出了一些如下，但你可以通过项目页面，如果你想了解更多关于他们：

* 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/. |

## Hadoop上的NLTK

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.

我们充分讨论了NLTK作为一个库，以及它为我们提供的一些最常用的功能。 现在，NLTK可以解决许多NLP问题，许多是高度可并行化的。 这就是为什么我们将尝试在Hadoop上使用NLTK的原因。

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.

在Hadoop上运行NLTK的最佳方法是将其安装在集群的所有节点上。 这可能不是很难实现。 有一些方法可以做到这一点，例如将资源文件作为流参数发送。 然而，我们宁愿选择第一个选项。

### 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.

有多种方法可以使NLTK在Hadoop上运行。 让我们谈谈使用Hive UDF通过并行执行标记化使用NLTK的一个示例。

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

对于此用例，我们必须按照以下步骤操作：

1. We have chosen a small dataset where only two columns exist. We have to create the same schema in Hive: 我们选择了一个只有两列的小数据集。 我们必须在Hive中创建相同的模式：

|  |  |
| --- | --- |
| **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: 在Hive中创建相同的模式。 以下Hive脚本将为您执行此操作：

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 $outTable with the same schema, and the added column of tokens: Hive script一旦我们有了模式，本质上，我们想要在一个单独的列中得到像内容的令牌。 所以，我们只想在$ outTable中有另一个具有相同模式的列，以及添加的令牌列：Hive脚本

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: 现在，我们已经准备好了模式。 我们必须在Python中编写UDF以逐行读取表，然后应用tokenize方法。 这与我们在第3章“语音标记”部分中所做的非常相似。 这是一个类似于第3章“语音标签部分”中所有示例的函数。 现在，如果你想得到POS标签，Lemmatization和HTML，你只需要修改这个UDF。 让我们看看UDF如何查找我们的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. 5.只是命名这个UDF像：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  
   6.现在，我们必须使用TRANSFORM函数运行insert hive查询，以便对给定内容应用UDF，并在新列中进行标记化和转储令牌：Hive脚本

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: 7.如果您收到此类错误，则说明您尚未正确安装NLTK和NLTK数据：

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: 8.如果您能够成功运行此Hive作业，您将获得一个名为OutTableName的表，如下所示：

|  |  |  |
| --- | --- | --- |
| **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的流操作

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:

让我们试试Python流的第二个选项。 我们有Hadoop流，我们可以编写自己的mapper和reducer函数，然后使用mapper.py的Python流，因为它看起来非常类似于我们的Hive UDF。 这里我们使用与map-reduce python流相同的示例，这将给我们选择一个Hive表或直接使用HDFS文件。 我们将只讨论文件的内容并将其标记化。 我们不会在这里执行任何reduce操作，但是为了学习，我包括一个虚拟reducer，它只是转储它。 所以现在，我们可以完全忽略来自执行命令的reducer。 这里是Mapper.py的代码：

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

## Hadoop 上的Scikit-learn

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.

大数据的另一个重要用例是机器学习。特别是使用Hadoop，scikit-learn更重要，因为这是我们为大数据评分机器学习模型的最佳选择之一。大型机器学习是目前最热门的话题之一，在大数据环境（如Hadoop）中这样做更加重要。现在，机器学习模型的两个方面是在大数据上构建模型，并在大量数据上构建模型并对大量数据进行打分。

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.

为了了解更多，让我们采用我们在上表中使用的相同示例数据，其中我们有一些客户评论。现在，我们可以使用重要的训练样本构建文本分类模式，并使用第6章中的一些学习文本分类来构建数据上的朴素贝叶斯，SVM或逻辑回归模型。在评分时，我们可能需要对大量数据进行评分，例如客户评论。另一方面，用大数据构建模型本身是不可能与scikit-learn，我们将需要工具像spark / Mahot的。我们将使用与NLTK一样使用预训练模型的相同的逐步方法进行评分。而在大数据上构建模式将在下一节中介绍。特别是当我们正在处理文本挖掘类问题时，使用预训练模型进行评分。我们需要两个主要对象（vectorizer和modelclassifier）作为序列化的pickle对象存储。

|  |
| --- |
| 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:

使用scikit在本地机器上构建离线模型，并确保选择对象。 例如，如果我使用第6章“文本分类”中的朴素贝叶斯示例，我们需要将vectorizer和clf存为pickle对象：

>>>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:

以下是创建输出表的步骤，其中包含整个历史记录的所有客户注释：

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: 1.在Hive中创建与上一个示例中相同的模式。 以下Hive脚本将为您执行此操作。 这个表可以是巨大的; 在我们的例子中，让我们假设它包含过去公司的所有客户评论：

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: 2.使用输出列构建输出表，如预测和概率分数：

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: 现在，我们必须使用Hive中的addFILE命令将这些pickle对象加载到分布式缓存：

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: 下一步是写Hive UDF，我们在这里加载这些pickle对象。 现在，他们开始行为与他们在当地的一样。 一旦我们有了分类器和矢量化器对象，我们可以使用我们的测试样本，这只是一个字符串，并生成TFIDF向量。 vectorizer对象现在可以用来预测类以及类的概率：

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: 一旦我们写了classification.py UDF，我们还必须将这个UDF添加到分布式缓存，然后有效地在表的每一行上运行这个UDF作为TRANSFORM函数。 这样的Hive脚本将如下所示：

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: 6.如果一切顺利，那么我们将输出表与输出模式：

|  |  |  |  |
| --- | --- | --- | --- |
| **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.

因此，我们的输出表将包含整个历史的所有客户评论，是否是投诉的分数，以及置信分数。 我们为我们的示例选择了一个Hive UDF，但是类似的过程可以通过Pig和Python通过类似于NLTK中的方式进行。

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.

这个例子是给你一个如何在Hive上学习机器学习模型的实践经验。 在下一个例子中，我们将讨论如何在大数据上构建机器学习/ NLP模型。

## 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:

让我们回到我们在Hadoop上构建一个机器学习/ NLP模型的讨论，另一个我们在Hadoop上得到一个ML模型。 我们在最后一节深入讨论了第二个评分选项。 相反，对较小的数据集和评分进行抽样，我们使用更大的数据集，并使用PySpark逐步构建大型机器学习模型。 我再次使用相同的运行数据具有相同的模式：

|  |  |  |
| --- | --- | --- |
| **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.

考虑组织的最近10年的评价的模式。 现在，不是使用一个小样本来构建分类模型，然后使用预训练模型对所有注释进行评分，让我给你一个分步示例，说明如何使用PySpark构建文本分类模型。

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.

我们需要做的第一件事是，我们需要导入一些模块。 从SparkContext开始，这是一个更多的配置，你可以提供更多的参数，如应用程序名称和其他。

**>>>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. |

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

接下来是阅读制表符分隔的文本文件。 读取文件应该在HDFS上。 这个文件可能很大（〜Tb / Pb）：

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

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

这些行现在是语料库中所有行的列表：

>>>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".

该部分是字段列表，因为我们有“\ t”分隔的行中的每个字段。

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

让我们打破在不同RDD对象中具有[ID，注释，类（0,1）]的语料库：

>>>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.

一旦我们得到了注释，我们需要做一个非常类似于第6章文本分类的过程，在这里我们使用scikit进行标记化，散列矢量化和使用矢量化计算TF，IDF和tf-idf。

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

以下是如何创建标记化，术语频率和反向文档频率的代码段：

>>>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:

我们将类与tfidf RDD合并，如下所示：

>>>finaldata = class\_var.zip(comment\_tfidf)

We will do a typical test, and train sampling:

我们将做一个典型的测试，并训练取样：

>>>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.

让我们执行主分类命令，这与scikit非常相似。 我们使用逻辑回归，这是广泛使用的分类器。 pyspark.mllib为您提供了各种算法。

|  |
| --- |
| For more information on pyspark.mllib visit https://spark. apache.org/docs/latest/api/python/pyspark.mllib.html |

The following is an example of Naive bayes classifier:

以下是Naive bayes分类器的示例：

>>>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.

nb\_output命令包含测试样本的最终预测。 最好的理解是，只有不到50行，我们为一个行业标准的文本分类构建了一个片段代码，甚至PB级的训练样本。

## 本章小结

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.

总结本章，我们的目标是应用我们迄今为止在大数据的背景下学到的概念。 在本章中，您学习了如何使用一些Python库，如NLTK和scikit与Hadoop。 我们谈论了评分机器学习模型或基于NLP的操作。

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.

我们还看到了最常见用例的三个主要例子。 理解这些示例，您可以应用大多数NLTK，scikit和PySpark函数。

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.

本章是对大数据的NLP和文本挖掘的快速简要介绍。 这是最热门的主题之一，我在示例代码片段中谈到的每个术语和工具本身都可能是一本书。 我试图给你一个黑客的方法，给你一个大规模的大数据和文本挖掘的介绍。 我鼓励你阅读更多关于一些大数据技术，如Hadoop，Hive，Pig和Spark，并尝试探索我们在本章中提供的一些例子。