# 第10章 **大规模文本挖掘**

在本章，我们打算再回头来谈谈之前章节中所介绍的一些程序库，但这回要谈的是如何在大数据环境中扩大规模地使用这些库。因此，我们会假设读者对于Hadoop+Hive这样的大数据框架已经有了一定的了解。在此基础之上，我们会对一些Python库进行一些相应的探讨，例如NLTK、scikit-learn和pandas这几个库都可以被应用于带有大规模非结构化数据的Hadoop集群。

我们将会讨论到一些讨论NLP和文本挖掘领域中常见的用例，在这过程中，我们也会给出一些代码片段，以便帮助您完成相关的工作。具体来说，我们要来看三个会涉及到绝大多数文本挖掘问题的主要示例。通过这些示例，我们会告诉您如何通过大规模地执行NLTK来完成本书最初几章中所介绍的那些NLP任务。此外，我们还将通过几个例子来介绍如何在大数据条件下执行文本分类任务。

当然，机器学习和NLP还有另一高度规模化应用的问题就是它们是否可并行化。我们在这里将会简单地讨论一下上一章中的一些问题，看看这些问题是否属于大数据问题。或者是否在某些条件下可以用大数据的方式来解决这些问题。

由于我们到目前为止所学习的大多数库都是用Python编写的，所以如何用Python（Hadoop）来处理大数据也是本章的主要问题之一。

在阅读完本章之后，我们希望读者：

* 能很好地了解Hadoop，Hive这些与大数据相关的技术。并在其条件下使用Python。
* 根据教程一步一步地掌握如何在大数据条件下使用NLTK、Scikit和PySpark。

## 在Hadoop上使用Python的不同方式

在Hadoop上运行一个Python进程的方式有很多种。在这里，我们将会讨论其中一些当前最为流行的方式，并通过这些方式在Hadoop上用Python来实现流式的MapReduce作业[[1]](#footnote-1)、Hive中的Python UDF、以及Python hadoop包装器。

### Python的流操作

通常，一个典型的Hadoop作业必须要被写成map+reduce函数的形式。用户需要根据给定任务来编写相应的map+reduce函数的实现。这些mapper和reducer通常是用Java来实现的。而与此同时Hadoop也为我们提供流式操作的接口，用户可以基于这些接口来写一个Python封装器，并用其它任意一种语言来编写之前由Java所实现的mapper和reducer函数。接下来，我们会来看一个用Python编写的单词计数示例。而且本章稍后还会介绍如何用NLTK库再实现一次。

|  |
| --- |
| 如果您还不太了解情况，可以看看下面链接中的资料  <http://www.michael-noll.com/tutorials/writingan-hadoop-mapreduce-program-in-python/>，以便了解一下Python环境下的MapReduce模式。. |

### Hive/Pig下的UDF

另一种使用Python处理大数据的方式就是在Hive / Pig中编写**UDF（User Defined Function）**。 这种方法的思路认为：我们在NLTK中所执行的大多数操作都是高度可并行化的。譬如说词性标注、标识化处理、词形还原、停用词移除以及NER这些都是可高度分布式执行的操作。因为其中的每一行内容都独立于其它行，所以我们在执行这些操作时不需要根据任何上下文。

因此，如果我们在集群上的每个节点上都部署了NLTK及其它Python库，也可以用Python来编写一些**用户定义函数（UDF）**，以借助NLTK和scikit这些库的功能。这是其引用NLTK最简单的一种方法，对于scikit的大规模引用则更是如此。我们会在本章后续内容中具体介绍这两个库的情况。

### 流式操作的封装器

各种不同组织所实现的封装器可以被列成一份长长的列表，以便能让Python在集群上执行相关的任务。这其中有一些封装使用起来其实相当简单，但问题是它们都有性能较差这个问题。我在下面也列出了一些，如果您想了解它们，可以去这些项目的网站去阅读一下相关介绍：

* Hadoopy
* Pydoop
* Dumbo
* Mrjob

|  |
| --- |
| 如果想查看一份更详细的目前Hadoop上可供选择的Python库列表，读者可以参阅下面这篇文章：  http://blog.cloudera.com/blog/2013/01/a-guide-topython-frameworks-for-hadoop/. |

## Hadoop上的NLTK

我们之前已经从一个库的角度对NLTK进行了充分的讨论，并介绍了它的一只鹅最常用的函数。目前，NLTK已经可以解决许多NLP问题，这其中有许多都是高度可并行化的方案。这也是为什么我们要试着在Hadoop上使用NLTK的原因。

在Hadoop上，运行NLTK的最佳途径就是将其安装在集群的所有节点上。这实现起来并不困难。有几种方式都可以做到这一点，例如我们可以将资源文件以流参数的形式来发送。但通常我们宁愿选择下面的第一个选项。

### 用户定义函数（UDF）

在Hadoop上运行NLTK的方法有很多，我们下面要讨论的实例是如何用NLTK来实现一个Hive UDF，以便完成并行式的标识化处理。

这个用例的操作可以分成以下步骤：

1. 我们在这里选择了一个只包含两列数据的小型数据集，然后在Hive中创建与之相对于的相同数据模式[[2]](#footnote-2)：

|  |  |
| --- | --- |
| **ID** | **内容** |
| 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. 现在，我们在Hive中创建相同的数据模式，该操作的Hive脚本如下：

Hive script

CREATE TABLE $InputTableName (

ID String,

Content String

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t';

1. 数据模式建构完成之后，基本上，下一步要做的就是针对每一个独立列中的内容进行类似于标识化的处理。所以，我们现在就会想在$ outTable中设置另一个具有相同模式的列，并为其添加相应的标识列：

Hive script

CREATE TABLE $OutTableName (

ID String,

Content String,

Tokens String

)

1. 现在模式方面的准备已经完成了，接下来我们要用Python来编写UDF，以逐行读取上面的数据表，并对其执行tokenize()方法。这一过程与我们在第3章“词性标注”所做的事情非常类似。这是一个与第3章“词性标注”中所有示例都很类似的函数。而且，如果您现在还想获得其POS标签、词形但愿以及某HTML标记的话，只需要相应地修改一下这个UDF即可。下面我们就来看看这个UDF是如何查找相关标识的：

>>> 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. 接下来，我们要命名一下这个UDF ，譬如nltk\_scoring.py。
2. 现在，我们得要用TRANSFORM函数来执行Hive的insert查询，以便将上面的UDF应用到给定的内容中，将新列标记化并转储其标识：

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. 如果您在上述过程中遇到了下面这样的错误信息，就说明NLTK机器数据组件没有被正确安装：

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. 如果该Hive作业能够被成功运行，我们就会得到一张名为OutTableName的表，具体如下：

|  |  |  |
| --- | --- | --- |
| **ID** | **内容** |  |
| 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

下面我们来试试第二个选项：Python流。我们根据现有的Hadoop流来编写自己的mapper和reducer函数，然后使用mapper.py来操作Python流，它的用法和Hive UDF非常类似。下面，我们就用同一个例子来示范一下map-reduce函数操作Python流的过程。这种方法为我们提供了一种可以使用Hive表，甚至直接使用HDFS文件的选项。在这里，我们只示范如何读取相关的内容并将其标识化，并不会涉及到任何reduce操作，但为了让学习有一定的完整性，我会在这里纳入一个虚拟的reducer，该函数只负责转储相关结果。也正因为如此，我们在这里可以完全忽略掉来自命令执行环境中的reducer。 下面是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])**

接下来是Reducer.py的代码：

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])**

下面再来看看Hadoop中执行Python流操作的命令：

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

机器学习是大数据领域中另一个非常重要的用例。在这方面Hadoop、scikit-learn这些框架就显得更尤为重要了，因为这是我们在大数据环境下评估机器学习模型的最佳选项之一。由于大型机器学习本身就是当前业界最热门的话题之一，所以在Hadoop这样的大数据环境中做这些事就显得更为重要了。当前，机器学习模型主要有两个方面的问题：在大数据环境下建模，以及针对大量数据的建模和评估。

为了加深对上述概念的理解，我们在这里也将采用之前那张表中的相同示例数据，这其中包含了一些客户的评论信息。现在假设我们要用一份重要的训练样本来构建一个文本分类模式，并利用在*第6章“文本分类”*中学到的知识在该数据上构建出朴素贝叶斯算法、SVM或逻辑回归模型。然后在进行评估的时候，我们可能需要面对的是针对海量数据的评估，譬如说客户评论这样的信息。另一方面，靠scikit-learn库本身来在大数据环境中建模是做不到的，所以我们在这里会需要用到像spark / Mahot这样的工具。另外，我们还会像在使用NLTK时一样用到某种预训练模型，并执行相同的步骤评估方法，我们将会在下一节中介绍其建模过程。特别在处理处理文本挖掘类问题时，使用预训练模型来执行评估是尤为合适的。当然，我们需要将这里的两个主要对象（vectorizer和modelclassifier）存储成某种序列化的pickle对象。

|  |
| --- |
| 这里提到的pickle其实是一个Python模块，主要用于实现序列化，即通过该模块，我们可以将对象以二进制状态存储在磁盘上，并且可以通过二次加载来使用。  <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。

1. 译者注：MapReduce是一种编程方式，主要用于针对大规模数据集的并行计算。 [↑](#footnote-ref-1)
2. 译者注：由于原文针对的是英文环境，所以这里的例句就不翻译了，下同。 [↑](#footnote-ref-2)