# Tutorial on MapReduce Framework in NLTK

## 1.1. Introduction to MapReduce programming model

Natural Language Processing often involves dealing with large corpus and data sets. MapReduce is a programming model and an associated implementation for processing and generating large data sets.[1] It is a programming model can greatly simplify the writing of distributed computing programs.

Conceptually, MapReduce can be treated as a combination of two functions: map function and reduce function. We can write their prototype in pseudo-code like:

```
map (k1,v1) ➔ list(k2,v2)
reduce (k2,list(v2)) ➔list(v2)
```

Map function receives a key and a value as its argument, and generates a list of keys and values. Reduce function receives a key and a list of values correspond to this key as its argument, and outputs a list of values. Basically, a map function and a reduce function is all the things require programmer to provide to run the distributed computing program.

You may wonder how this very simple interface can lead to great computing power. Actually, it is the simplicity help to get power from distributed computing environment. Complex program structure tends to scale badly in large clusters because the dependency between different parts of program would constraint the parallelism degree of the program. Conversely, the map function in MapReduce is very simple. It assumes input data as a list of independent key and value, thus allowing different map functions to run on different computer.

It would be more concrete to illustrate MapReduce with a metaphor. The election of president is a good one. To count the votes, the simplest way would be to ask an officer to sequentially collect each resident's choice and add the votes each candidate get together. Obviously, this method is too slow. In practice, we would do this job in two steps: (1) set up a lot of ballot boxes, and then every qualified individual would go to vote there; (2) after voting, the officers just need to count the votes in the box and sum them together. This is exactly what MapReduce do: (1) split the job to a lot of small parts and let map operation to process them respectively and (2) finally use reduce operation to combine the result.

MapReduce encapsulates a lot of low-level details below its very simple interface. Knowing how to use this interface should be enough in a lot of cases. But in some cases, low-level details can help to devise a clever solution. Anyone interested in details should take a look at [1].

## 1.2. MapReduce Framework in NLTK

NLTK — the Natural Language Toolkit — is a suite of open source Python modules, data and documentation for research and development in natural language processing[2]. To support large data sets processing in NLP, NLP includes a MapReduce framework. This framework is implemented based on Hadoop[3], a software platform that lets one easily write and run applications that process vast amounts of data.

### 1.2.1.Installation

To run MapReduce Framework in NTLK, you should have installed Hadoop and NLTK. These are not trivial tasks. Hadoop and NLTK project both provide good installation guides on their home pages. You should read these guides before installation.

After installation, it is very important to ensure they work correctly. The testing methods are also provided in their documents respectively.

### 1.2.2.Usage

To take the MapReduce approach to solve your own problems, you should first consider the problem carefully in the MapReduce way. That is, how to express the problem with only map and reduce operations, what data to input and what data to output.

In order to be consistent with the interface of MapReduce, you should provide two python source file to run the MapReduce program. One file implement map operation, the other implement reduce operation. The MapReduce framework in NLTK has a very simple interface for users to use. Briefly speaking, to implement your map/reduce operation, you just need to create an Python source file to define a Mapper/Reducer class, which should inherit the MapperBase/ReducerBase class and implement the map/reduce function. The map/reduce function require you to complete has very simple interface:

map(key, value)

reduce(key, value)

In the most common cases, map and reduce functions both receive key and value as arguments and output key and value, which will be passed into the next round of map and reduce.

## 1.3. Demo MapReduce Program

It is very hard to explain MapReduce without giving some examples. The following chapter will guide you through several demo programs, from simple WordCount example to complex unsupervised HMM tagging example.

### 1.3.1. WordCount

As the name implies, WordCount program just count different word (space separated token) in several files and report the occurrences of each unique word. If we don't want to run this program in distributed environment, it would be very simple to write it in Python. We can just declare a dictionary, and then every time we read a word we fetch the count of this word in dictionary and increase it by one. But suppose we have a very large corpus (1TB for example) to process, it would be too slow to run it on a single machine so that we have to resort to large clusters. Of course we can use MapReduce to solve this problem. But remember that MapReduce is just a tool, you have to find the parallelism possibility conveyed in the problem. Obviously, we can divide the very large data sets to small parts, and then run multiple instance of word counting program to process these small parts and finally combine the results to form an overall report.

It is not very hard to devise the solution, first let's have a look at the mapper file:

```
#!/usr/bin/env python

from hadooplib.mapper import MapperBase

class WordCountMapper(MapperBase):

    def map(self, key, value):
        words = value.split()
        for word in words:
            self.outputcollector.collect(word, 1)

if __name__ == "__main__":
    WordCountMapper().call_map()
```

At line 5, we define a class named WordCountMapper and make it inherit the MapperBase class.

From line 7 to line 10, we implement the map function, which is the only function requires us to complete. As is stated before, map function receives a key and a value as its argument. But what exactly do these arguments represent? It depends the particular applications. For this program, key is useless and ignored; value is a line of text extracted from the data sets. How does this program know about this specification? Because all Mapper classes have an attribute named inputformat, whose default value is set to TextLineInput class. This class defines a static method named read_line, which specify when we read a line from the input file the whole line is returned as value argument of map function and Python None object is returned as its key argument. Of course we can change this behavior, but it is better to illustrate it with an example in the following section.

Now that value contains a string of text, we must split it to get a list of words, that's what exactly line 8 does. After getting the word list, you may be tempted to declare a dictionary and use it to sum the count of each word. This approach is just too overkill. Here, we simply report each word's occurrence. In the map stage, we won't concern ourselves with the problem of how to combine all the results. Line 10 illustrates how to output the result. MapperBase class defines an outputcollector attribute, which defines how to output the result. Users can also change the output behavior according to their needs, but the default behavior satisfies the requirement here. The default action is to take a key argument and a value argument and output them in the same line but separated them with a tab character (i.e. '\t').

To make it more clear, the map function receives a key and a value, processes them and output a different key and value. This interface should be very easy to remember and to use.

Then there is the content of reducer file:

```
1  #!/usr/bin/env python
2
3  from hadooplib.reducer import ReducerBase
4
5  class WordCountReducer(ReducerBase):
6
7      def reduce(self, key, values):
8          sum = 0
9          try:
10             for value in values:
11                 sum += int(value)
12             self.outputcollector.collect(key, sum)
13         except ValueError:
14             #count was not a number, so silently discard this item
15             pass
16
17 if __name__ == "__main__":
```

```
18        WordCountReducer().call_reduce()
```

The class hierarchy and interface of Reducer is very similar to the one of Mapper. Implementing the reduce function is the only mandatory requirement. While map and reduce function have very similar interfaces, they are very different in nature:

1. In reduce function's view, the arguments are one key and one list of values corresponds to this key while in map function's view, the arguments are one key and one value.
2. The key argument in reduce function is guaranteed to be unique, i.e. every unique token will be received by reduce function only once. But in map function, mostly the same token will be received multiple times.
3. There is one more guarantee that reduce function can provide: every value correspond to a key will be fetched into the values argument, which is a list.

You may wonder how propery1, 2 and 3 of reduce function is achieved. This is an implementation detail. But I would like to explain it a bit here, because in NLTK we provide these properties in a clean way: (1) sort the (key, value) pair on key field (actually this is done by Hadoop[4]); (2) group all the values having the same key with groupby function from itertools package. Also, the API document for Reducer interface in Hadoop is a good reference.[4]

According to the above assurance, the reduce function in the code above is much more easy to read. This function merely needs to add all the values belong to the specified word together, just as line 10 and line 11 do.

The code of wordcount program is complete, but our discussion is not. Compared to normal program runs on a single machine, distributed programs is particularly hard to debug and test. So before you run it on a cluster, you may want to ensure it run correctly on a single computer. This is how I test the wordcount program with a shell script:

```
export LC_ALL=C
cat input | ./wordcount_mapper.py | sort | ./wordcount_reducer.py > output
diff output correct_output
```

Finally, it is time to put it onto real distributed environment, the command to execute program is long and takes a bit of time to input. So we provide a runStreaming.sh script for your convenience.

### 3.1.1.Name Similarity

Usually one MapReduce program involves multiple stages of map and reduce step. Here is a little program to find out the most similar name for every given name in a name list. For example, the most similar name for Adam will be Ada. The complete code will not be posted here, because every map and reduce step is fairly easy to code once if you define each map and reduce step properly.

There will always be multiple way to do a job with MapReduce. What provided below is just a possible solution:

1. every name is converted to a pair: (name, the_first_char_of_the_name) in map stage, like Adam ➔ (Adam, A); in reduce stage we just output the pair coming from map stage. You can write a program to do this job but the “cat” program in Unix would be suitable as well.
2. swap the pair coming from the previous round of MapReduce in map stage, like (Adam, A) ➔ (A, Adam). Note that because this map stage is an intermediate stage, it is more convenient to change the inputformat from TextLineInput to KeyValueInput, which breaks a tab-separated-line to a key and a value. In reduce stage, we aggregate all the words having the same key and output them to the same line, like: (A, [Ada, Adam, Adams …]) ➔ (A, Ada Adam Adams).
3. In map stage, we sort the values list, and for every name in the list we output the name before it and the name after it. For example, [Ada, Adam, Adams …] ➔ (Adam, Ada Adams)… Apparently, The most similar name for a given name will be either its predecessor or successor, so we use the reducer to compute the similarity between them and finally output the answer.

This program seems too overkill. It involves 3 rounds of MapReduce and is a little hard to understand. But the main concept express here is: via carefully designed scheme, the original problem would be transformed gradually in small step and finally solved.

## 3.1.2.TF*IDF

Now let’s see a more realistic application of MapReduce: computing the TF*IDF value of every word in a large data set.[5]

Toy program tends to be short and straight-forward, while real program often requires you to play some tricks. This program is such one. Obviously, to calculate TF*IDF, the program needs to acquire the text from the data set file. But on the other hand, the program also needs to know which file the given text is from. But we could only accept one kind of input or else we could not tell them apart. So in this situation, how would we define the input for this program?

There are some tricks we can adopt. The first one is to preprocess the text files in advance: adding a file name before each line of text. But this procedure requires us to do this job in a tedious way: we must preprocess it in local machine, which can waste a lot of time.

Here we adopt a solution which takes advantage of Hadoop, which can submit additional files (stop words, dictionary etc.) with “-file” option. The input for this program is not bulks of text but a list of file names to be processed. But now how to provide the text in data set? We can submit them with “-file” option. Once submitted, they lie in the same namespace or directory as the executable python scripts’. So we can open them as normal files though they are now actually distributed over the cluster.

Once the data input problem is solved, the program should be comparatively easy to write. Basically we just need to output the (term, filename) pair, then count (term, filename) for term frequency and inverse document frequency and finally combine them to calculate the TF*IDF value.

### 3.1.3. Unsupervised HMM Training

Now we provide another typical application of MapReduce program: to iteratively run a program until some condition is satisfied. Unsupervised HMM training [6] [7] and pageRank algorithm both belong to this category.

This unsupervised HMM training program is directly modified from original code in hmm.py file from NLTK. It represents one typical kind of problem: moving codes run in a single computer to clusters. Facing this situation, we should try to extract the code that can be executed simultaneously to put into the map stage.

This program is really hard to explain, but it has nothing to do with MapReduce because this algorithm is complex in nature. We would just sketch the main concept here. Normally we would use multiple sequences to train a HMM. The original code in NLTK computes several local quantities for each sequence, and then combines all these local quantities to get a global quantity and finally infer the parameters of HMM from that global quantity. It is not hard to see that the computation of local quantities should be done in map stage to maximize the parallelism and the combination of local quantities should be done in reduce stage.

There is one more difficulty left: every round of HMM training should get the HMM parameters from the previous round and propagate the new parameters to the next round of training until converged. So how can we pass these parameters around? We can output those parameters to a file, and then fetch this file to local and use the “-file” technique to submit this file in the next round of executing. This procedure requires us to check if the converged condition is satisfied and to execute Hadoop commands. This requirement is not easy to be satisfied with shell scripts, so we write a runStreaming.py python script to do this job.

## 1.4. Reference

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI’04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS,

pages 137–150, 2004.

[2] http://nltk.sourceforge.net/index.php/Main_Page

[3] http://hadoop.apache.org/

[4] http://hadoop.apache.org/core/docs/r0.18.0/api/index.html

[5] http://en.wikipedia.org/wiki/Tf-idf

[6] http://en.wikipedia.org/wiki/Hidden_Markov_model

[7] Lawrence R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," Proceedings of the IEEE, 77 (2), p. 257–286, February 1989