COSC 6339

Fall 2018

Big Data Analytics

2nd Assignment

Jaivardhan Singh Shekhawat

Problem Description:

Duplicate Detection:

- **Discovery of multiple representations of the same real-world object**
- **Problems:**
    1. **Representations are not identical (Fuzzy Duplicates)**
    2. **Data sets are large: quadratic complexity if comparing every pair of records**
- **Similarity measures:**
    1. **Domain-dependant vs. domain independent solutions**
    2. **Avoid comparisons by partitioning**
    3. **Parallel computing**

Part 1:

a. **Write a pyspark code to determine the 1,000 most popular words in the document collection provided. Please ensure that your code removes any special symbols, converts everything to lower case (and if possible: remove stop words, destemming, etc.).**
    - **Stop word removal can be done either by using some nltk or creating your own list of words to be removed (e.g. and, or, the, it,…)**

Spark with python language combines to become Pyspark. Apache Spark is an open-source distributed general-purpose cluster-computing framework. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Cluster is nothing but a platform to install Spark. One can run Spark on distributed mode on the cluster. In the cluster, there is master and n number of workers. It schedules and divides resource in the host machine which forms the cluster. The prime work of the cluster manager is to divide resources across applications. It works as an external service for acquiring resources on the cluster.

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to fetch text files from the path given into *text* variable. Then, to split each word of data, we need to use *split()* function, and storing to *words* variable, lower casing the words simultaneously by *lower()* function. Then, to remove punctuation, we replaced punctuations with space. Then, to remove stopwords, we imported *stopwords* from *nltk.corpus*, for removing that '*english'* stopwords from words we are getting. After that, we made word pair and given value of 1 to each, so that later we can add the values to join them using *add* imported from *operator*. Then, we swap the key and value, so that we sort by key using *sortByKey()* function and since we need to sort descending order, so we passed *False* through that function. And then we need only words, not the count of that, so we removed them. As required, we need to give first 1000 popular words, so we took first 1000 using *take()* function. Finally, to save the output in a file of hdfs cluster, we used *saveAsTextFile()* function.

How to run code:

spark-submit --master yarn code2_1.py /cosc6339_hw2/gutenberg-500/ /bigd45/output_filename

How to see output:

hdfs dfs -cat /bigd45/output_filename/part-00000

Part 2:

a. Write a pyspark code to create an inverted index for the 1,000 words determined in Part 1. The inverted index is supposed to be of the form:
term1: doc1: weight1_1, doc2: weight2_1, doc3: weight3_1,…
term2: doc1: weight1_2, doc2: weight2_2, doc3: weight3_2,…
…

where weight x_y is: no. of occurrences of term x in document y / total number of words in document y

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to fetch text files from the *path* given into *text* variable. Then, to split each word of data, we need to use *split()* function, and storing to *words* variable, lower casing the words simultaneously by *lower()* function. Then, to remove punctuation, we replaced punctuations with space. Then, to remove stopwords, we imported *stopwords* from *nltk.corpus*, for removing that '*english'* stopwords from words we are getting. After that, we made word pair and given value of 1 to each, so that later we can add the values to join them using *add* imported from *operator*. Then, we swap the key and value, so that we sort by key using *sortByKey()* function and since we need to sort descending order, so we passed *False* through that function. And then we need only words, not the count of that, so we removed them. As required, we need to give first 1000 popular words, so we took first 1000 using *take()* function. And stored in array *finalWords*. Then, using *for* loop, we fetched words into that variable. Then, as required we need to give filenames also with words, so we fetched filenames, stemming "/" and preceding it, so that we have only filename, instead of whole path of file, so for that we used *split()* function. Now, to remove punctuation, we used *string.punctuation* and filtering it using *filter()* function. Now, to count words of file(s), we need to use *len()* function by splitting the words. Then, we checked particular(from popular 1000 words) word is present or not using *checkWords()* function. Now, to add the values of word and file which has that word, we used *add* imported from *operator*. Since, word and file are key, and fraction is value, so we interchanged the form of that. We changed to word as a key and file(s) and fraction as values. Then, we grouped the key using *groupByKey()* function. Then map the values using *mapByValues()* function, passing *list* through it. Finally, to save the output in a file of hdfs cluster, we used *saveAsTextFile()* function.

Table 1 and graph 1.1 represents the execution time taken by code developed by this part.

How to run code:

spark-submit --master yarn –num-executors 15 code2_2.py /cosc6339_hw2/gutenberg-500/ /bigd45/output_filename

How to see output:

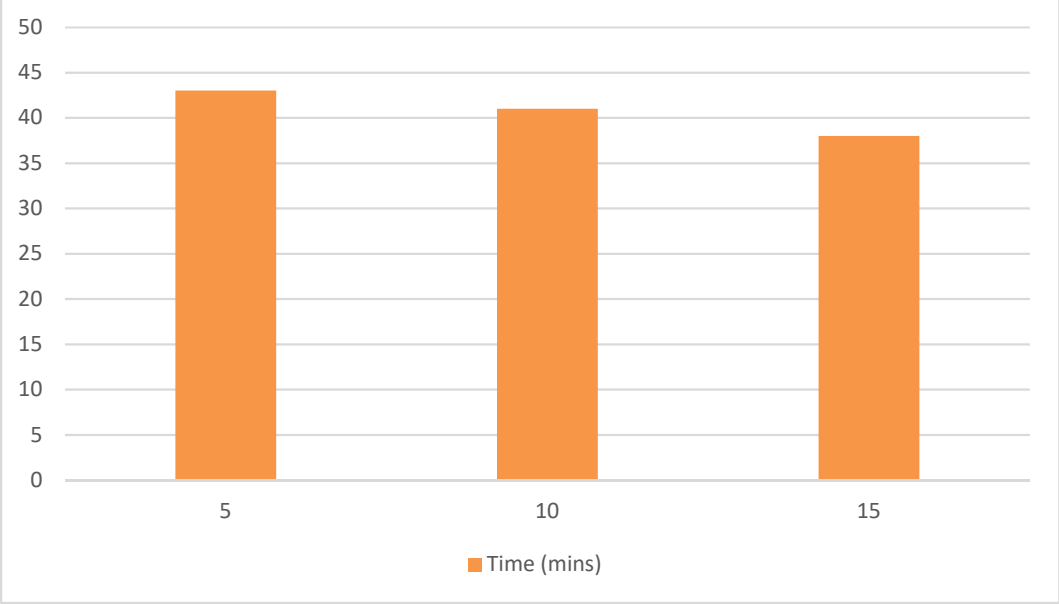
hdfs dfs -cat /bigd45/output_filename/part-00000

b. Measure the execution time of the code for the large data set for 5, 10 and 15 executors

Table 1: Execution time of code developed

Executors Used	Time (in minutes)	Avg. Time (in min.)
5	42	43
	43	
10	41	41
	40	
15	37	38
	38	

Graph 1.1: Graph representation of table above



Part 3:

a. Write a pyspark code to calculate the similarity matrix S with each entry of S being :
S (docx, docy) = $_{t\epsilon V}(weight_{t\_docx} \times weight_{t\_docy})$
With V being the vocabulary (determined in part 1) and the weights having been determined in part 2

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to fetch text files from the *path* given into *text* variable. Then, to split each word of data, we need to use *split()* function, and storing to *words* variable, lower casing the words simultaneously by *lower()* function. Then, to remove punctuation, we replaced punctuations with space. Then, to remove stopwords, we imported *stopwords* from *nltk.corpus*, for removing that '*english'* stopwords from words we are getting. After that, we made word pair and given value of 1 to each, so that later we can add the values to join them using *add* imported from *operator*. Then, we swap the key and value, so that we sort by key using *sortByKey()* function and since we need to sort descending order, so we passed *False* through that function. And then we need only words, not the count of that, so we removed them. As required, we need to give first 1000 popular words, so we took first 1000 using *take()* function. And stored in array *finalWords*. Then, using *for* loop, we fetched words into that variable. Then, as required we need to give filenames also with words, so we fetched filenames, stemming "/" and preceding it, so that we have only filename, instead of whole path of file, so for that we used *split()* function. Now, to remove punctuation, we used *string.punctuation* and filtering it using *filter()* function. Now, to count words of file(s), we need to use *len()* function by splitting the words. Then, we checked particular(from popular 1000 words) word is present or not using *checkWords()* function. Now, to add the values of word and file which has that word, we used *add* imported from *operator*. Since, word and file are key, and fraction is value, so we interchanged the

form of that. We changed to word as a key and file(s) and fraction as values. Then, we grouped the key using *groupByKey()* function. Then map the values using *mapByValues()* function, passing *list* through it. Now, we need to define a function named *func_similarity*, which takes the 2 documents and get the fraction of similarity between them by multiplying the fraction of similarity of both. Then, *sim_rdd* is defined to pass that function into the result which we got in the part 2 of this assignment and using *filter* function so that no 2 same documents are compared. Finally, to save the output in a file of hdfs cluster, we used *saveAsTextFile()* function.

Table 2 and graph 2.1 represents the execution time taken by code developed by this part.

How to run code:

spark-submit --master yarn –num-executors 15 code2_3.py /cosc6339_hw2/gutenberg-500/ /bigd45/output_filename

How to see output:

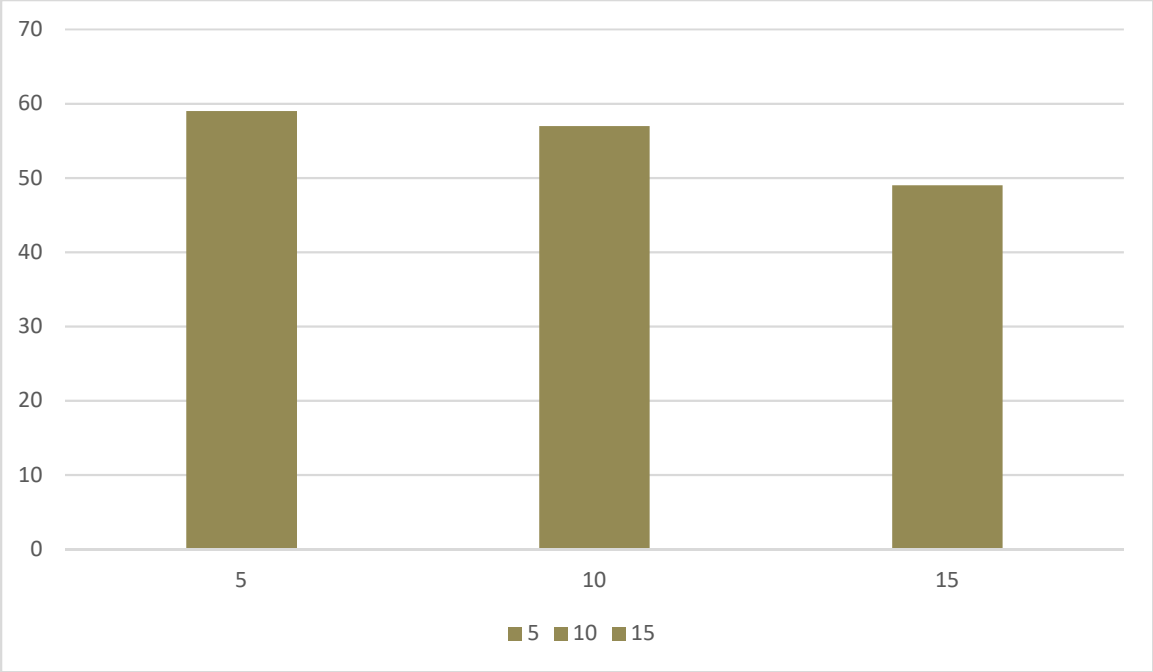
hdfs dfs -cat /bigd45/output_filename/part-00000

b. Measure the execution time for the large data set for 5, 10 and 15 executors

Table 2: Execution time of code developed

Executors Used	Time (in minutes)	Avg. Time (in min.)
5	59	59
	58	
10	57	57
	58	
15	49	49
	50	

Graph 2.1: Graph representation of table above



Part 4:

a. Provide a list of the 10 most similar (or identical) pair of documents from the large data set

Solution Strategy:

First, we need to set application name, port number and executor memory. Then, we need to fetch text files from the *path* given into *text* variable. Then, to split each word of data, we need to use *split()* function, and storing to *words* variable, lower casing the words simultaneously by *lower()* function. Then, to remove punctuation, we replaced punctuations with space. Then, to remove stopwords, we imported *stopwords* from *nltk.corpus*, for removing that '*english'* stopwords from words we are getting. After that, we made word pair and given value of 1 to each, so that later we can add the values to join them using *add* imported from *operator*. Then, we swap the key and value, so that we sort by key using *sortByKey()* function and since we need to sort descending order, so we passed *False* through that function. And then we need

only words, not the count of that, so we removed them. As required, we need to give first 1000 popular words, so we took first 1000 using *take()* function. And stored in array *finalWords*. Then, using *for* loop, we fetched words into that variable. Then, as required we need to give filenames also with words, so we fetched filenames, stemming "/" and preceding it, so that we have only filename, instead of whole path of file, so for that we used *split()* function. Now, to remove punctuation, we used *string.punctuation* and filtering it using *filter()* function. Now, to count words of file(s), we need to use *len()* function by splitting the words. Then, we checked particular(from popular 1000 words) word is present or not using *checkWords()* function. Now, to add the values of word and file which has that word, we used *add* imported from *operator*. Since, word and file are key, and fraction is value, so we interchanged the form of that. We changed to word as a key and file(s) and fraction as values. Then, we grouped the key using *groupByKey()* function. Then map the values using *mapByValues()* function, passing *list* through it. Now, we need to define a function named *func_similarity*, which takes the 2 documents and get the fraction of similarity between them by multiplying the fraction of similarity of both. Then, *sim_rdd* is defined to pass that function into the result which we got in the part 2 of this assignment and using *filter* function so that no 2 same documents are compared. Then, we swaped key and value so that we can sort according to values using *sortByKey()* function, and then swaped again to get the required format as result. Since, we required only top 10 similar documents, so we used *take()* function passing 10 as an argument. Then, used *sc.parallelize()* function because list is not able to save in text file. Finally, to save the output in a file of hdfs cluster, we used *saveAsTextFile()* function.

How to run code:

```
spark-submit --master yarn –num-executors 15 code2_4.py
/cosc6339_hw2/gutenberg-500/ /bigd45/output_filename
```

How to see output:

```
hdfs dfs -cat /bigd45/output_filename/part-00000
```

Output:

((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/1momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/4momm10u.utf'), 0.0556874658201321)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/3momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/4momm10u.utf'), 0.0545867891212528)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/1momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/5momm10u.utf'), 0.05455522452335854)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/3momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/5momm10u.utf'), 0.0532747945665852)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/2momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/5momm10u.utf'), 0.0526475789887901)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/2momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/4momm10u.utf'), 0.0519885854655852)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/5momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/4momm10u.utf'), 0.0516558465224568)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/8momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/5momm10u.utf'), 0.0481669752132020)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/8momm10u.utf', u'hdfs://whale-hdfs-

master:54310/cosc6339_hw2/gutenberg-500/2momm10u.utf'), 0.0465485258545685)
((u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/8momm10u.utf', u'hdfs://whale-hdfs-master:54310/cosc6339_hw2/gutenberg-500/3momm10u.utf'), 0.0449524158412358)

Resources Used:

Cluster:

50 Appro 1522H nodes (whale-001 to whale-057), each node with

- two 2.2 GHz quad-core AMD Opteron processor (8 cores total)
- 16 GB main memory
- Gigabit Ehternet
- 4xDDR InfiniBand HCAs (not used at the moment)

Network Interconnect

- 144 port 4xInfiniBand DDR Voltaire Grid Director ISR 2012 switch (donation from TOTAL)
- two 48 port HP GE switch

Storage

- 4 TB NFS /home file system (shared with crill)
- ~7 TB HDFS file system (using triple replication)

Analysis:

After execution of code in cluster and from table I derived that as the number of executors increases on a job, the execution time (average time to execute the code developed) reduces.