COSC 6339
Big Data Analytics
Homework – 2

Priscilla Imandi
1619570

Problem Description
To find the similarity between documents
- i) Find the 1000 most popular words in the collection of documents
- ii) Create an inverted index for the words from part i)
- iii) Calculate the similarity matrix from part ii)
- iv) Getting the top 10 similar documents

Solution Explanation
- **i)** **<u>Finding the 1000 popular words</u>**

  - **Reading the file :**

    ```
    fileData = sc.wholeTextFiles(inputfile, use_unicode= False)
    #Trimming the full-filename
    fileData_base = fileData.map(lambda(filename,content) : (ntpath.basename(filename),content) )
    ```

    Here, the whole folder can be read using the command sc.wholeTextFiles.
    "inputfile" is the filepath of the directory of the input dataset
    "ntpath.basename" will trim the filepath to the final filename

  - **Pre-processing the data :**

    ```
    fileData_base.map(lambda(file, content) :(file, re.sub('[^a-z| |0-9-]', '', content.
    ```

    Now only the alphanumeric characters and hypen will be considered and all other characters are eliminated. This helps us in forming proper words, also hyphenated words will be a part of the text.

  - **Changing the structure of the data :**

    ```
    fileData_preprocess.flatMap(
    ```

    The RDD is in the form of a list of lists, using flat map eliminates this and gives us tuples of words and lengths.

  - **Splitting and Removing Stop Words :**

    ```
    (file, content) : [(word,1) for word in content.split(" ") if not word in stopwordsList])
    ```

    For every word add a value 1 to specify that the word appears one time.
    Check if the word is not in the stop word list(list containing pre-positions etc)

  - **Reducer Step :**

    ```
    #Counting the words
    fileData_reduced = fileData_split.reduceByKey(add, numPartitions=1)
    ```

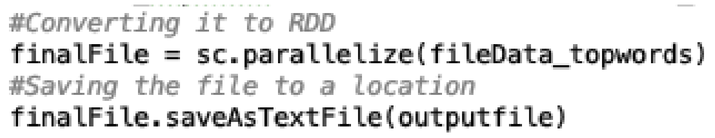
    This the reducer step – where the word is the key and for every key the values are added.

  - **Finding the top 1000 words:**

    ```
    #Finding the top 1000 words - by index 1 - which has count of the word
    fileData_topwords = fileData_reduced.takeOrdered(1000, key = lambda x: -x[1])
    #fileData_sorted = fileData_reduced.sortByValue(False).map(lambda x : x[0]).top(1000)
    fileData_topwords = [i[0] for i in fileData_topwords]
    ```

Sort by value(word counts) in descending order, which gives the top 1000 words. Now, remove the count and consider only the list of words.

- **Convert to RDD and Save:**

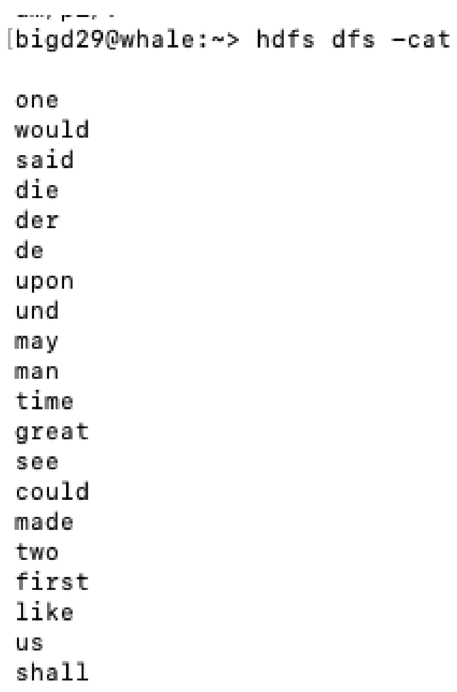


```
#Converting it to RDD
finalFile = sc.parallelize(fileData_topwords)
#Saving the file to a location
finalFile.saveAsTextFile(outputfile)
```

Since the words are in the form of a list, we cannot save it to a text file, because HDFS works with RDDs.

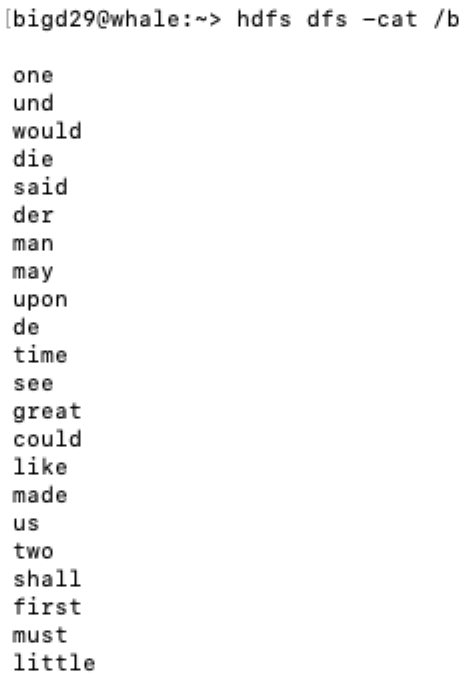
Output File:

Medium Dataset



```
bigd29@whale:~> hdfs dfs -cat

one
would
said
die
der
de
upon
und
may
man
time
great
see
could
made
two
first
like
us
shall
```

Large Dataset



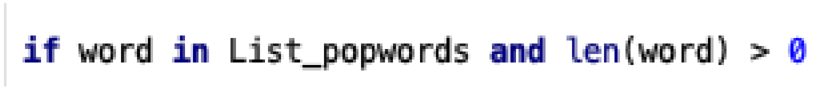
```
bigd29@whale:~> hdfs dfs -cat /b

one
und
would
die
said
der
man
may
upon
de
time
see
great
could
like
made
us
two
shall
first
must
little
```

- A part of the output file can be seen above both from medium as well as big datasets.
- The output is in the form of 4 separate part files in the output folder
- The location of the output file is **/bigd29/output_hw2/big/p1/51/part-r-00000** on the HDFS file system.
- Each file contains 250 words each as we have chosen to take 1000 words
- These words are arranged in descending order of their count – which can be seen for the small dataset shown below (modified version).

```
('the', 684)
('to', 465)
('and', 391)
('of', 347)
('a', 266)
('you', 213)
('in', 194)
('that', 186)
('for', 181)
('or', 174)
('is', 167)
('this', 153)
('not', 143)
('i', 138)
('it', 137)
('by', 114)
('if', 113)
('be', 103)
('his', 102)
('but', 102)
('she', 101)
```

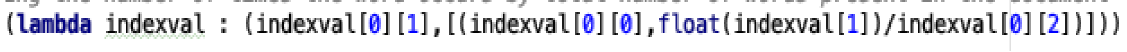
ii) <u>Creating the Inverted Index</u>

- **Collecting Popular Words**
  Read the output files from part-1, split them based on the space delimiter and add the words to a list
- Using the same process as part-1, read the entire folder of the input dataset and and pre-process it. And check if the word is present in the list of popular words.



```
if word in List_popwords and len(word) > 0
```

- Apply the reduce operation and get the word count. Now map each word and document to the corresponding weight. This is done by dividing the number of times the word occurs by the total number of words in the document.



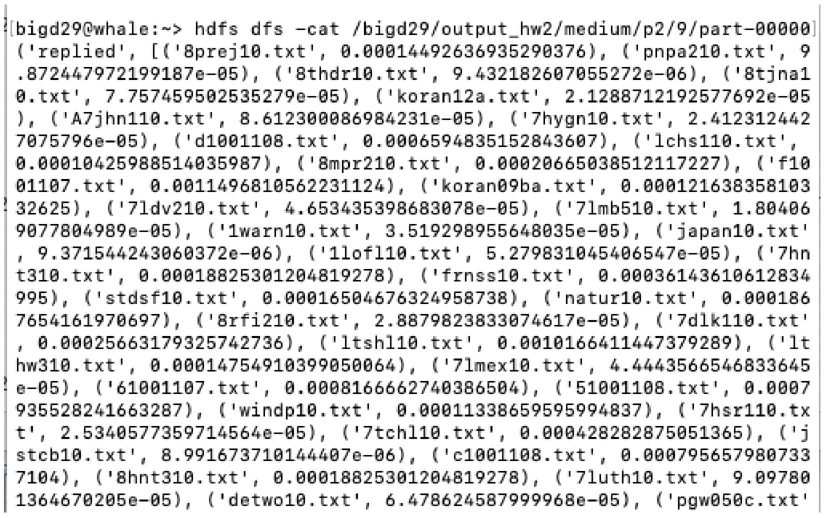
```
(lambda indexval : (indexval[0][1],[(indexval[0][0],float(indexval[1])/indexval[0][2])]))
```

  Word weight = word count / total number of words in document
- Reduce by add key – Now the key would be the word and the value would be a list of tuples of all the documents and their respective weights.

Output File:
Large Dataset

```
bigd29@whale:/home2/cosc6339/bigd29/homework2/source>
hdfs dfs -cat /bigd29/output_hw2/big/p2/65/part-00000
('replied', [('8prej10.txt', 0.00014492636935290376),
('7rwrl10.txt', 2.303324849420138e-05), ('lvymr10.txt'
, 4.816955684007707e-05), ('7gent10.txt', 0.0007383053
584818596), ('8thdr10.txt', 9.432182607055272e-06), ('
7deng10.txt', 2.992220227408737e-05), ('dmess10.txt',
0.00043584854263143557), ('7hygn10.txt', 2.41231244270
75796e-05), ('lchs110.txt', 0.00010425988514035987), (
'bvsmt10.txt', 0.00011156978690170702), ('fltws10.txt'
, 0.0005752987631076593), ('koran09ba.txt', 0.00012163
835810332625), ('7nctl10.txt', 0.0003618206816701643),
 ('8inqt10.txt', 0.0002029467873523562), ('8pasw10a.tx
t', 1.7635129177321224e-05), ('8vnmm10.txt', 9.8934801
96550473e-05), ('7hnt310.txt', 0.00018825301204819278)
, ('rmadv10.txt', 0.000540385899106774), ('lttaf10.txt
', 5.538862040793719e-05), ('19rus10.txt', 0.000128919
```

Medium Dataset



```
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium/p2/9/part-00000
('replied', [('8prej10.txt', 0.00014492636935290376), ('pnpa210.txt', 9
.872447972199187e-05), ('8thdr10.txt', 9.432182607055272e-06), ('8tjna1
0.txt', 7.757459502535279e-05), ('koran12a.txt', 2.1288712192577692e-05
), ('A7jhn110.txt', 8.612300086984231e-05), ('7hygn10.txt', 2.412312442
7075796e-05), ('d1001108.txt', 0.0006594835152843607), ('lchs110.txt',
0.00010425988514035987), ('8mpr210.txt', 0.00020665038512117227), ('f10
01107.txt', 0.0011496810562231124), ('koran09ba.txt', 0.000121638358103
32625), ('7ldv210.txt', 4.653435398683078e-05), ('7lmb510.txt', 1.80406
9077804989e-05), ('1warn10.txt', 3.519298955648035e-05), ('japan10.txt'
, 9.371544243060372e-06), ('1lofl10.txt', 5.279831045406547e-05), ('7hn
t310.txt', 0.00018825301204819278), ('frnss10.txt', 0.00036143610612834
995), ('stdsf10.txt', 0.00016504676324958738), ('natur10.txt', 0.000186
7654161970697), ('8rfi210.txt', 2.8879823833074617e-05), ('7dlk110.txt'
, 0.00025663179325742736), ('ltshl10.txt', 0.0010166411447379289), ('lt
hw310.txt', 0.00014754910399050064), ('7lmex10.txt', 4.4443566546833645
e-05), ('61001107.txt', 0.0008166662740386504), ('51001108.txt', 0.0007
935528241663287), ('windp10.txt', 0.00011338659595994837), ('7hsr110.tx
t', 2.5340577359714564e-05), ('7tchl10.txt', 0.0004282828287505136 5), ('j
stcb10.txt', 8.991673710144407e-06), ('c1001108.txt', 0.000795657980733
7104), ('8hnt310.txt', 0.00018825301204819278), ('7luth10.txt', 9.09780
1364670205e-05), ('detwo10.txt', 6.478624587999968e-05), ('pgw050c.txt'
```

- A part of the output file can be seen above.
- In the above picture – ‘replied’ is the key. And the values following it are a list of tuples of document and the respective weight of ‘replied’

iii) <u>Finding the similarity Matrix</u>

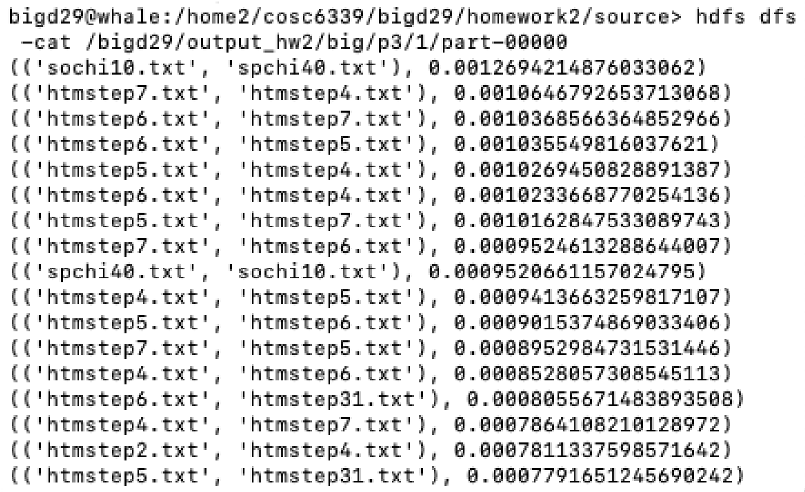
- Collecting Inverted Index Values
  Read the output files from part-2, split them based on a \n character and store them in a RDD
- Now after processing we have the data in the processable format but data is same as output generated by part-2.
- The converted list also called as postings has 2 values for every row – At index 0 it has the word and at index 1 it has the list of tuples of documents and their respective weights.
- For every word we perform the following operation

  ```
  sim = ((doc_weight[i][0],doc_weight[j][0]),weight_i*weight_j)
  ```

  In the above line, doc_weight datastructure contains all list of tuples of documents and their respective weights.
  Now at index 0, i.e doc_weight[x]**[0]**, we have the names of the documents. And the index 1 has weights. weight_i corresponds to weight of document i and weight_j to document j. Multiplying these weights would give the similarity between those documents
- Now we have a matrix of document tuples and their corresponding weights for various words(words are not written into the RDD).
- Now, perform a reduce operation, this would add all the similarity values of various words for all the pair of documents.
- This would be our Similarity index

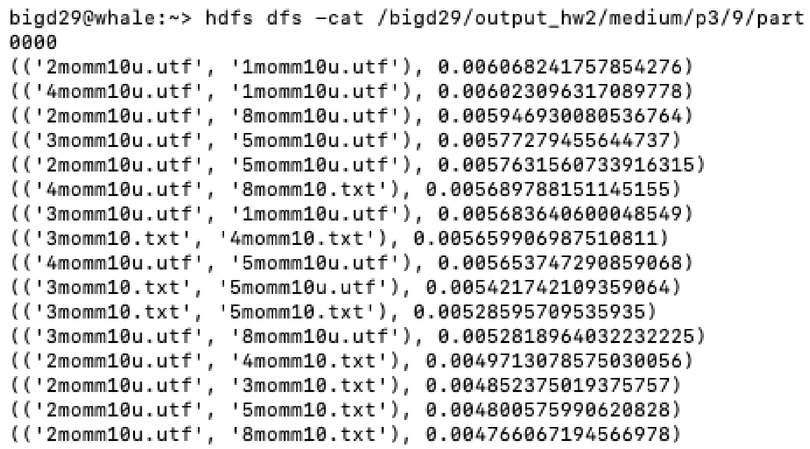
Output File:

Large Dataset



```
bigd29@whale:/home2/cosc6339/bigd29/homework2/source> hdfs dfs
 -cat /bigd29/output_hw2/big/p3/1/part-00000
(('sochi10.txt', 'spchi40.txt'), 0.0012694214876033062)
(('htmstep7.txt', 'htmstep4.txt'), 0.0010646792653713068)
(('htmstep6.txt', 'htmstep7.txt'), 0.0010368566364852966)
(('htmstep6.txt', 'htmstep5.txt'), 0.001035549816037621)
(('htmstep5.txt', 'htmstep4.txt'), 0.0010269450828891387)
(('htmstep6.txt', 'htmstep4.txt'), 0.0010233668770254136)
(('htmstep5.txt', 'htmstep7.txt'), 0.0010162847533089743)
(('htmstep7.txt', 'htmstep6.txt'), 0.0009524613288644007)
(('spchi40.txt', 'sochi10.txt'), 0.0009520661157024795)
(('htmstep4.txt', 'htmstep5.txt'), 0.0009413663259817107)
(('htmstep5.txt', 'htmstep6.txt'), 0.0009015374869033406)
(('htmstep7.txt', 'htmstep5.txt'), 0.0008952984731531446)
(('htmstep4.txt', 'htmstep6.txt'), 0.0008528057308545113)
(('htmstep6.txt', 'htmstep31.txt'), 0.0008055671483893508)
(('htmstep4.txt', 'htmstep7.txt'), 0.0007864108210128972)
(('htmstep2.txt', 'htmstep4.txt'), 0.0007811337598571642)
(('htmstep5.txt', 'htmstep31.txt'), 0.0007791651245690242)
```

Medium Dataset



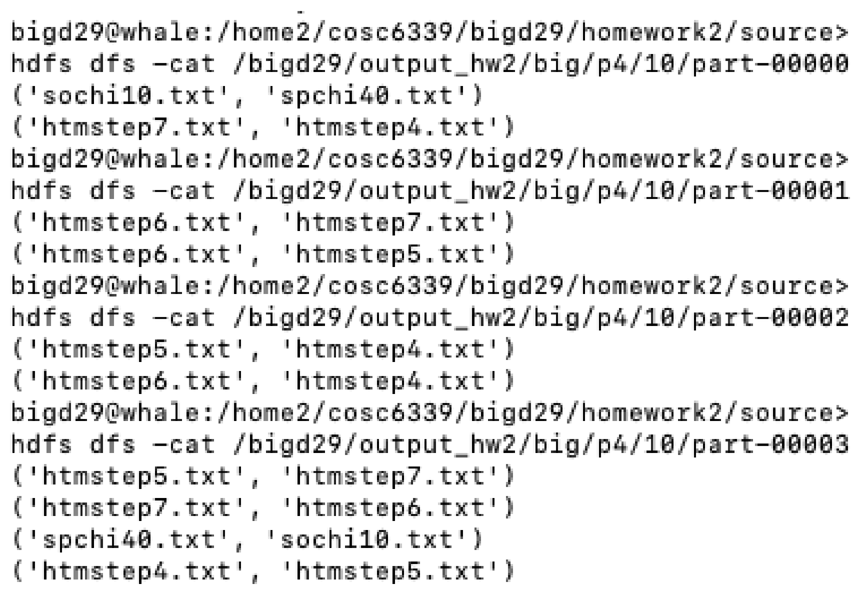
```
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium/p3/9/part
0000
(('2momm10u.utf', '1momm10u.utf'), 0.006068241757854276)
(('4momm10u.utf', '1momm10u.utf'), 0.006023096317089778)
(('2momm10u.utf', '8momm10u.utf'), 0.005946930080536764)
(('3momm10u.utf', '5momm10u.utf'), 0.00577279455644737)
(('2momm10u.utf', '5momm10u.utf'), 0.0057631560733916315)
(('4momm10u.utf', '8momm10.txt'), 0.005689788151145155)
(('3momm10u.utf', '1momm10u.utf'), 0.005683640600048549)
(('3momm10.txt', '4momm10.txt'), 0.005659906987510811)
(('4momm10u.utf', '5momm10u.utf'), 0.005653747290859068)
(('3momm10.txt', '5momm10u.utf'), 0.005421742109359064)
(('3momm10.txt', '5momm10.txt'), 0.00528595709535935)
(('3momm10u.utf', '8momm10u.utf'), 0.0052818964032232225)
(('2momm10u.utf', '4momm10.txt'), 0.0049713078575030056)
(('2momm10u.utf', '3momm10.txt'), 0.004852375019375757)
(('2momm10u.utf', '5momm10.txt'), 0.004800575990620828)
(('2momm10u.utf', '8momm10.txt'), 0.004766067194566978)
```

- It can be seen that the document pair(tuple) and the similarity weight are seen.
- Even here, the files for the bigger dataset will be larger when compared to the medium dataset because the pairs(combinations) of documents will be more in the bigger dataset.

iv) <u>**Finding the top 10 Similar Documents**</u>

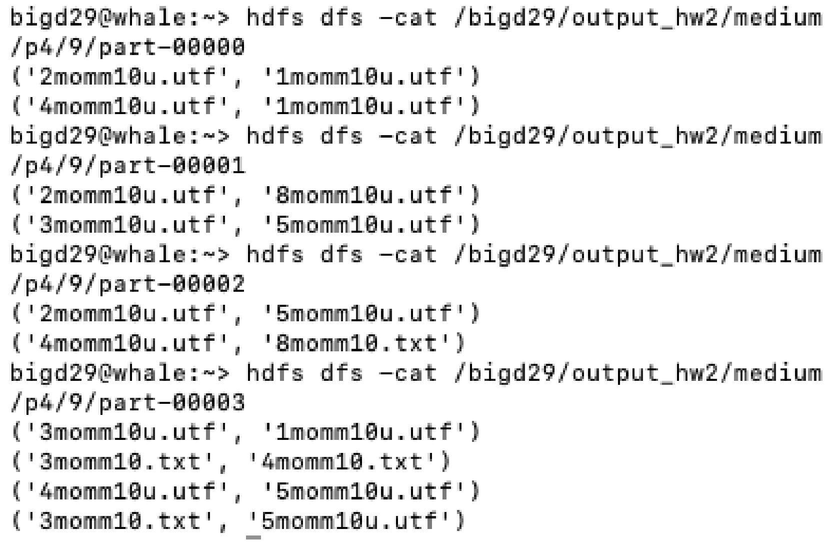
- Collecting Similarity Matrix Values
  Read the output files from part-3, process them and store them in a RDD
- Get the top 10 pairs based on highest weights
- Save the output

Output File:
Large Dataset



```
bigd29@whale:/home2/cosc6339/bigd29/homework2/source>
hdfs dfs -cat /bigd29/output_hw2/big/p4/10/part-00000
('sochi10.txt', 'spchi40.txt')
('htmstep7.txt', 'htmstep4.txt')
bigd29@whale:/home2/cosc6339/bigd29/homework2/source>
hdfs dfs -cat /bigd29/output_hw2/big/p4/10/part-00001
('htmstep6.txt', 'htmstep7.txt')
('htmstep6.txt', 'htmstep5.txt')
bigd29@whale:/home2/cosc6339/bigd29/homework2/source>
hdfs dfs -cat /bigd29/output_hw2/big/p4/10/part-00002
('htmstep5.txt', 'htmstep4.txt')
('htmstep6.txt', 'htmstep4.txt')
bigd29@whale:/home2/cosc6339/bigd29/homework2/source>
hdfs dfs -cat /bigd29/output_hw2/big/p4/10/part-00003
('htmstep5.txt', 'htmstep7.txt')
('htmstep7.txt', 'htmstep6.txt')
('spchi40.txt', 'sochi10.txt')
('htmstep4.txt', 'htmstep5.txt')
```

Medium Dataset

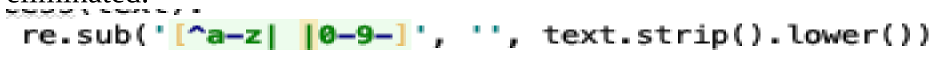


```
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium
/p4/9/part-00000
('2momm10u.utf', '1momm10u.utf')
('4momm10u.utf', '1momm10u.utf')
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium
/p4/9/part-00001
('2momm10u.utf', '8momm10u.utf')
('3momm10u.utf', '5momm10u.utf')
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium
/p4/9/part-00002
('2momm10u.utf', '5momm10u.utf')
('4momm10u.utf', '8momm10.txt')
bigd29@whale:~> hdfs dfs -cat /bigd29/output_hw2/medium
/p4/9/part-00003
('3momm10u.utf', '1momm10u.utf')
('3momm10.txt', '4momm10.txt')
('4momm10u.utf', '5momm10u.utf')
('3momm10.txt', '5momm10u.utf')
```

Challenges and Findings:

i) First, when the code is being written for the short list used for code development – the data had different text when compared to the bigger dataset.

ii) As there were multiple special characters in the large dataset which weren't handled in the small dataset, the output printed was gibberish. Using Regular expressions helped solve the problem as we need not hardcode all the special characters to be eliminated.



```
re.sub('[^a-z| |0-9-]', '', text.strip().lower())
```

Some characters like hyphen cannot be eliminated as they are also a part of a word.

But this has lead to ‘-‘ and ‘--‘ also being considered as a word

iii) And after incorporating this the developmental code yielded good results.

iv) Also, handling the RDD is tricky because it cannot be done in the normal way and to check the output after every step

v) We can observe that in Part-1 both the documents produce almost the same words – showing that either:
    - i) The medium dataset is a subset of the large dataset
    - ii) The above words are popular in general and can be found in any literature

vi) In part-2 the output files for the big dataset are much larger because the number of files in the bigger dataset are more and every word will have more number of entries.

vii) The weights of each of the word is very small, this is because of the vast number of words in every document. So, as per the weight calculation formula the denominator is very large compared to the numerator and hence the low value.

viii) In part-3 we can see that even different file formats are compared – like .utf is compared to .txt. This checks only for the content of the file which is what is needed.

Specifications of the System:

<u>Whale Cluster :</u>

i) Nodes:
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 Ethernet
- 4xDDR InfiniBand HCAs (not used at the moment)

ii) Network Interconnect

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

iii) Storage

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

<u>Spark in Python:</u>

i) It is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing

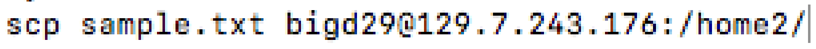
ii) It can be used with Scala, Python, R and SQL shells.

iii) Spark 2.3.1
iv) Hadoop 3.0.3
v) Open JDK 64-Bit Server VM, 1.8.0_171

Execution of the code

i) File Systems:
   - First, it is important to understand that there are 2 systems here
     Local Linux system
     HDFS system
   - Now the source code files are stored in the local Linux system from where we run the job.
   - The data file which contains the text file records and the output location will be on the HDFS system.

ii) Copying the source code file from the local Desktop to Remotely connected Linux system.
Scp <local-location>   Username@Host: <Remote-Location>
Enter the password when prompted for it



```
scp sample.txt bigd29@129.7.243.176:/home2/
```

iii) Submitting a job:
spark-submit --master yarn --conf spark.ui.port=4060 --num-executors 15 --executor-cores 4 --driver-cores 4 part1.py /bigd29/output_hw2/big/p1/50 /cosc6339_hw2/large-dataset/
The above command is used to run the job, which indicates the number of executors used, the number of executor-cores that can be allotted, name of the source file, hdfs location of the data file, hdfs location of the output directory

iv) Viewing the output:
hdfs dfs -cat /bigd29/output_hw2/ big/p1/50 /part-00000

The above command has the location of the output file.

Results & Finding:

Executors are JVM instances that are run on every node.
The executor cores are number of cores that can run the tasks parallelly for each executor.

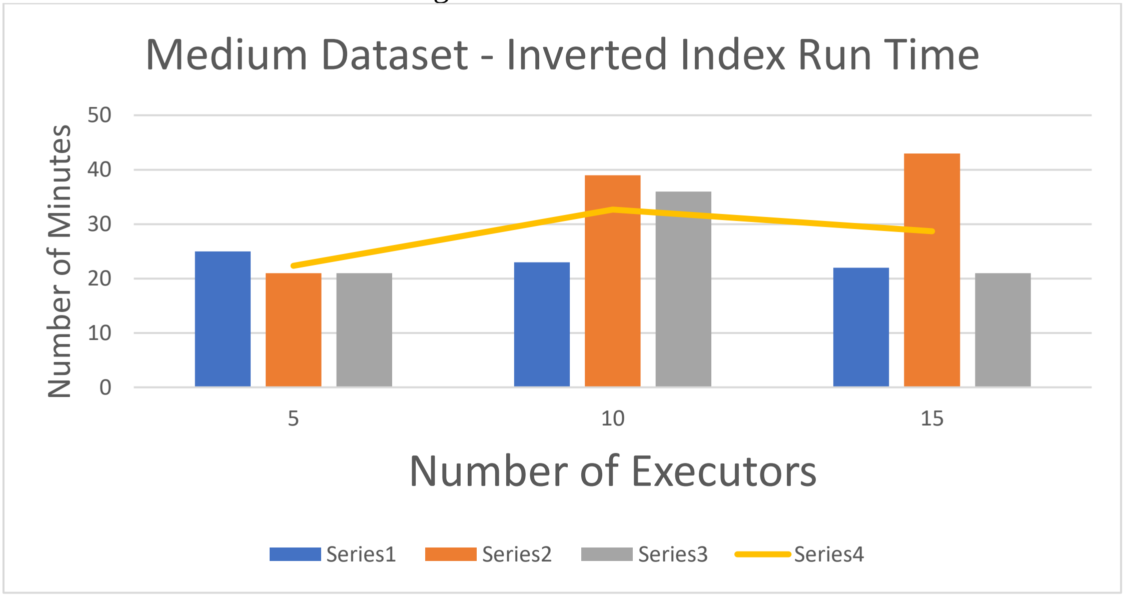
i) <u>Finding the Inverted Index for the dataset :</u>

Medium Dataset

Runs\Number of Executors	5	10	15
1	25	23	22
2	21	39	43
3	21	36	21
Average	22.3333333	32.6666667	28.6666667
Min	21	23	21

Units in seconds – Table showing the number of seconds the job ran for different runs with different number of executors.

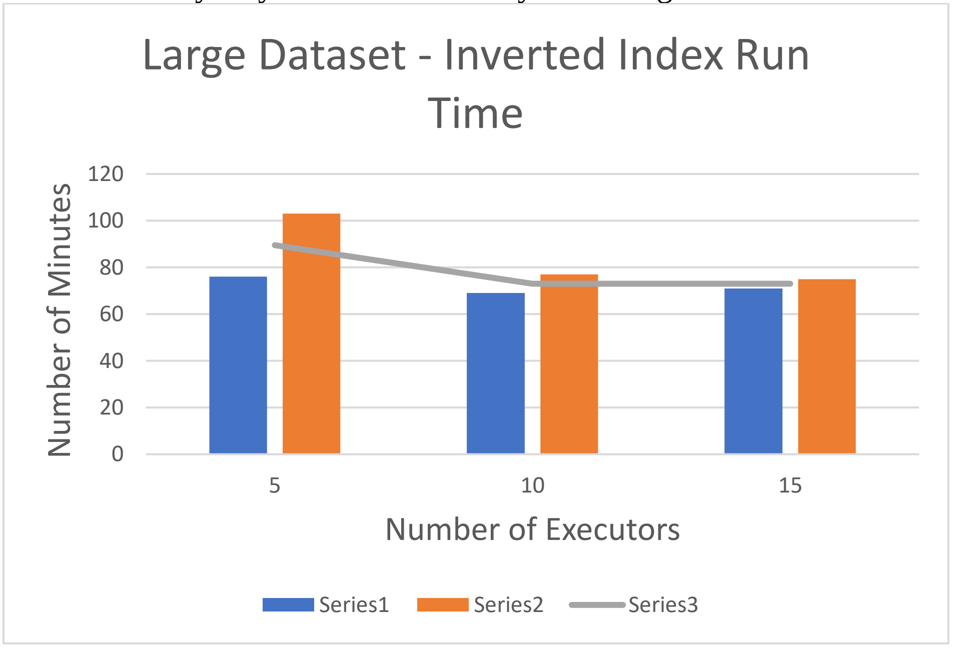
- From 15 to 10 to 5 we can see that the time is constant in this case – this is mostly because of the load on the cluster at the time the jobs were run
- But we can expect that with increase in the number of executors there is decrease in time taken to run
- But from the graph below we can see that there is a decrease in the time from 10 to 15.
- Also I have taken an additional measurement 18 executors which completed the task in 18 minutes showing the downward trend in time taken.



Large Dataset

Runs\Number of Executors	5	10	15
1	76	69	71
2	103	77	75
Average	89.5	73	73
Min	76	69	71

- Here the downward trend can be seen much better with 5 executors consuming around 1/3 time more than the 15 executors.
- Though there is no much significant difference between 10 and 15 executors.
- We can also assume that for every dataset there is an optimum number of executor cores beyond which we can only see slight improvement in time
- The Minimum values are almost equal in the case of 10 and 15 showing that they may not benefit more by increasing the number of executors.



ii) Finding the Similarity Matrix:

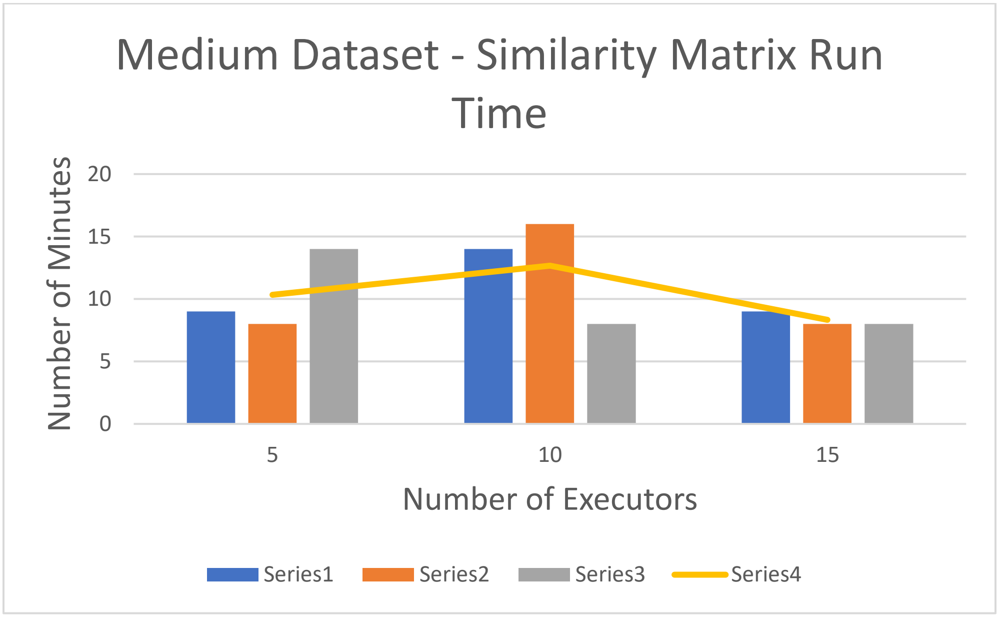
Medium Dataset

Runs\Number of Executors	5	10	15

1	9	14	9
2	8	16	8
3	14	8	8
Average	10.3333333	12.6666667	8.33333333
Min	8	8	8

Units in seconds – Table showing the number of seconds the job ran for different runs with different number of executors.

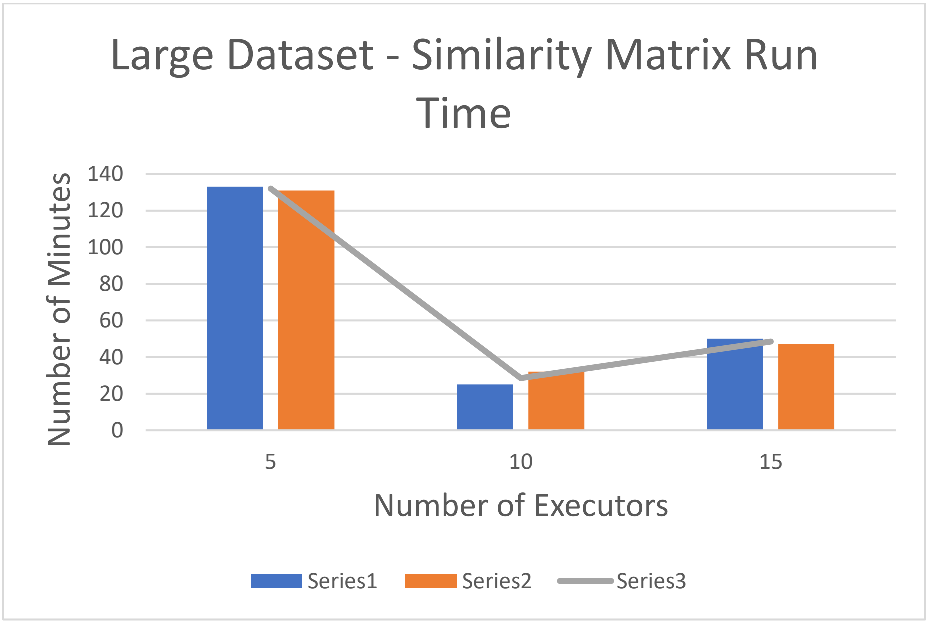
- From 15 to 10 to 5 we can see that the time is constant in this case – this is mostly because of the load on the cluster at the time the jobs were run
- But we can expect that with increase in the number of executors there is decrease in time taken to run
- But from the graph below we can see that there is a decrease in the time from 10 to 15.
- Also I have taken an additional measurement 18 executors which completed the task in 8 minutes showing the downward trend in time taken but with a slight decrease for every task



Large Dataset

Runs\Number of Executors	5	10	15
1	133	25	50
2	131	32	47
Average	132	28.5	48.5
Min	131	25	47

- This is a good example of the increase in time from 5 to 10 which is almost 5 times.
- This steep drop can be due to :
    - i) The cluster load during the times the jobs were run
    - ii) The job is performed in an optimized way at 10 executors and higher.
- But 15 executors has higher timing showing that allotting more resources to a job is always not the best idea
- For every job, there has to be a peak point where it is performed well and then parallelization beyond that point is not as useful.



In conclusion, we can say that increasing the number of resources and parallelizing the jobs does help in increasing the efficiency and decreasing the run time.
But, this is also depends on :

- The cluster specifications on which we are performing the job
- The extend of parallelization the task is implementing
- The current load on the cluster.
- The efficiency of the algorithm we are using.
- The number of executors we are using
- The size of the input file and various other factors.