Fuzzy Duplicate Detection in a Big Data Text Corpus

Objective

The objective of this project is to come up with a mechanism to detect duplicate objects, where the term 'duplicate' means same or mostly similar representations of the same object in a document collection. The project is divided into four parts:

1. Generate a vocabulary of the 1,000 most popular words in the document collection. Proper pre-processing steps like stopword removal need to be performed.
2. Generate an inverted index for the 1,000-word vocabulary.
3. Compute the similarity matrix for the inverted index generated in part 2.
4. Provide a list of 10 most identical documents in the document collection.

We also compare different file formats which can be used to store data in Big Data applications. The file formats being considered are text, AVRO, parquet and parquet with snappy compression. We write the inverted index to the file format in question in Part 2 and read data from this file format to memory in Part 3 which gives us an estimate of the format's read and write efficiencies for our application. We also compare the file sizes of the inverted index saved to disk.

The document collection used for this homework is the 'Gutenberg Dataset', which is a collection of 3,036 English books written by 142 authors. However, all calculations were performed on the medium subset of the dataset which had 519 books. We used the pyspark framework to implement Spark functionalities in Python 2.7 on the University of Houston's whale cluster.

Solution Strategy

Part 1:

In Part 1, we need to generate a list of (word, wordcount) tuples for the 1,000 most popular words in the given document collection. The code for this part of the document can be found in "part1_wordcount.py". We generate a Resilient Distributed Database (RDD) from the entire document collection using pyspark's textFile function. On using textFile(), the generated RDD has a line from a file in the document collection stored as a string in each row. This is opposed to the wholeTextFiles function, which stores a key value pair of the document path and the entire document as a string in a single row of the RDD. Since we are interested in the 1,000 most popular words in the entire document collection and are not concerned about 'which word belongs to which document' kind of relation, using textFile() with a wildcard '*' is the way to go as it offers a greater scope for parallelization.

After generating the RDD, we use the defined 'clean(line)' function on each row of the RDD. The clean(line) function does the following:

- Retains only the alphabets in the string by imposing the regex '[^a-zA-Z]+'. Special characters are removed in this step. Numbers are generally not a good feature to measure the similarity of two text documents. Hence, they are discarded as well.
- Converts the string to lower case and splits it into a list of words.
- Retains only those words which are not part of a pre-defined set of stopwords. (Stopword removal).
- Applies the Snowball Stemmer on each word and returns this list of words.

After passing each row of the RDD through our clean(line) function, we filter out words which are of length 1 or are over 20. Words of length 1 do not contain enough information for our purpose and words of length over 20 are just absurd anomalies.

We flatten out the result of above mentioned transformations using flatMap() after which each row in the RDD becomes a word. Mapping the word in each row to a (word, 1) tuple and adding them by key using reduceByKey() give us a (word, wordcount) tuple for each unique word in the RDD. We can extract the top 1,000 tuples using takeOrdered() and eventually save it as a file 'vocabulary.out' to be used for Part 2 of the project.

Part 2:

The code for Part 2 can be found in "part2_inverted_index_<file_format>.py". We first store the contents of the file 'vocabulary.out', generated by Part 1 into an RDD. We then collect its keys, store them as a set and broadcast this set using pyspark's broadcast().

We generate an RDD from the entire document collection using wholeTextFiles(). Each row in the RDD is a key value pair of the document path and the contents of the document as a string. In line 35, we extract the file name from the document path, and on the contents of the document, we apply a similar sequence of operations as the clean(line) function in Part 1 but this time, retaining only those words which are part of our broadcasted vocabulary. The filter() function is to remove .utf files, whose value becomes an empty list after the above transformations. Files with .utf extension need to be decoded to become human readable and were not dealt with. Hence, they were filtered out.

After the above transformation, each row in the RDD is a key value pair of the file name and the list of words that appear in the file which are part of our vocabulary. Now we calculate the normalized term frequency (ntf) for each word in the list of words.

Term frequency of a word 'x' in a document 'y' is given by:

$$tf_{xy} = \text{number of occurrences of x in y / total number of words in y}$$

The denominator of the above formula is the length of the document. While dealing with multiple documents, (as we will in Part 3) it is easy to see that term frequency will be biased towards documents of shorter length and will punish longer documents. This could lead to incorrect similarity measurements in a skewed document collection. To negate this problem, I used normalized term frequency throughout this homework.

Normalized term frequency of a word 'x' in a document 'y' is given by:

ntf_{xy} = number of occurrences of x in y / number of occurrences of the most common word in y

Normalized term frequency can be calculated using the user-defined ntf() function. It takes a list of words as an input parameter and returns a list of (word, ntf_{word}) tuples. The RDD is now of the form (document, list of (word, ntf_{word}) tuples for all unique words in the document). Flattening the RDD to (word, [document, ntf_{word} in that document]) tuples and concatenating the values by key using reduceByKey() gives us the inverted index in the required format. We eventually save this RDD as different file extensions to be used in Part 3 and compare the write times of these different file extensions.

Part 3:

The code for Part 3 can be found in "part3_similarity_matrix_<file_type>.py". We define a function pair(x) which takes a list 'x' whose elements are of the form (doc1, ntf_{doc1}) and returns a list 'xy' of length len(x)C_2 whose elements are of the form, ((doc1, doc2), ntf_{doc1}*ntf_{doc2}) for all combinations of documents in list 'x'. We are only interested in the combinations of documents and not the cartesian product because the similarity matrix is symmetric and computing elements in the any one of the lower/upper diagonals is enough.

We generate an RDD from "inverted.txt" generated by Part 2 using textFile(). Each row in the RDD is a string of the file. To convert this string to a (key, value) tuple we use literal_eval() from Python's "ast" library. Now each row in the RDD is a key value pair where the key is a word and the value is a list of tuples (documents in which the word appears, ntf_{word} in that document). Using pyspark's map(), we apply the pair(x) function on the value of the tuple in each row of the RDD and using reduceByKey() with the addition function gives us the similarity metric for each pair of documents. This RDD can be thought of as the flattened lower/upper diagonal of our similarity matrix. We finally save the RDD in different file types to be used for part 4 of the project and also measure read and write times of different file types.

Part 4:

The code for Part 4 can be found in "part4_output.py". We use pyspark's textFile() to generate an RDD from the "matrix.txt" result of part 3. Each row of the RDD is a line in the file stored as a string. We use literal_eval() to convert string to tuples. We use pyspark's takeOrdered() to take the top 10 pairs with the highest similarity metric.

Command to run the Code:

The command to run the code is the following:

spark-submit --master yarn –deploy-mode cluster –num-executors <num> <python file>

Default values of all other parameters of spark-submit were used.

Results

Resources Used:

The cluster used for this homework was the whale cluster provided by the Parallel Software Technologies Laboratory at the University of Houston. The technical specifications of the cluster as mentioned on their web page are:

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)

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 – another cluster at PSTL)
- ~7 TB HDFS file system (using triple replication)

On the software side, we used Python 2.7.13 and Spark 2.3.1.

Measurements Performed

All measurements were performed on the medium dataset. Both Part 2 and Part 3 were run twice with 5, 10 and 15 executors respectively. The execution time calculated is the time difference between the job start time and job finish time as reported on the cluster's official webpage http://whale.cs.uh.edu:8088/cluster. Table 1.1 shows the execution times calculated for each run of the Spark job and the number of executors used and the outputs were saved as text files.

Number of Executors	Execution Times	Average Execution Time
5	16min 52s, 16min	16min 26s (986s)
10	14min 21s, 14min 40s	14min 30.5s (870.5s)
15	8min 45s, 8min 20s	8min 32.5s (512.5s)

Table 1.1(a) Execution time per number of executors for Part 2 using text file

Number of Executors	Execution Times	Average Execution Time
5	3min 52s, 4min 6s	3min 59s (239s)
10	3min 37s, 3min 6s	3min 21.5s (201.5s)
15	1min 58s, 2min 28s	2min 13s (133s)

Table 1.1(b) Execution time per number of executors for Part 3 using text file

Fig. 1.1 shows a graph drawn between the number of executors and the average execution time of the job.

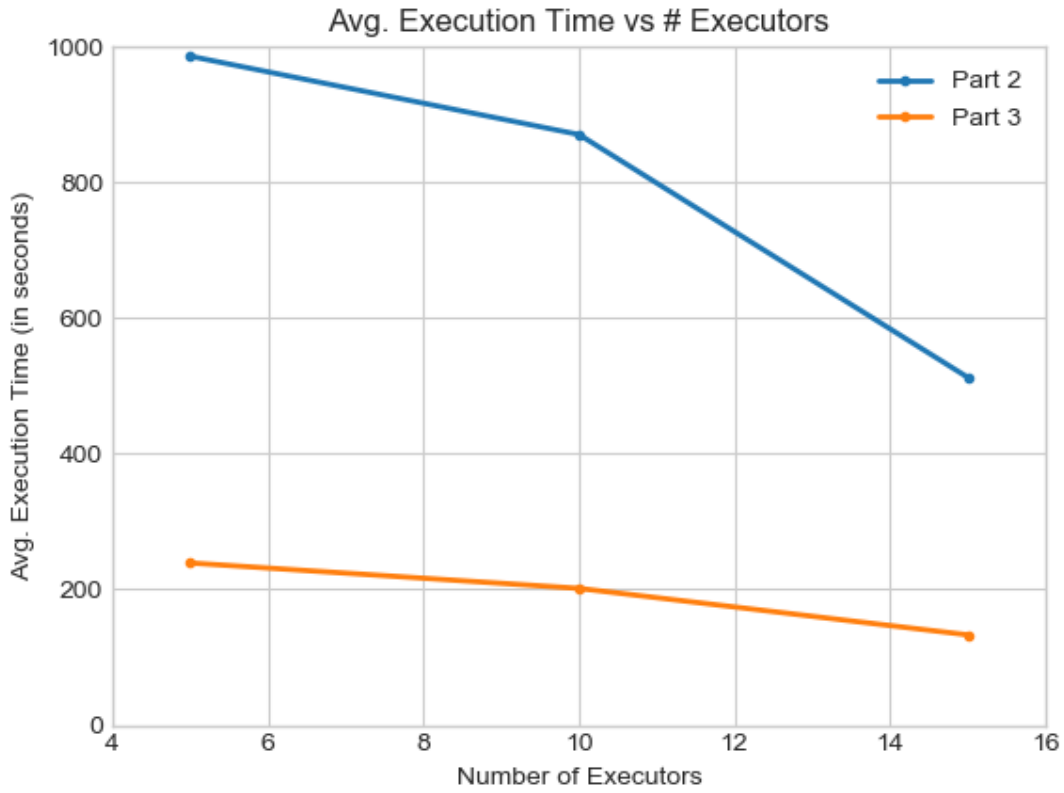

Fig 1.1 Graph between average execution time and number of executors used.

Table 1.2 shows the execution times calculated for each run of the Part 2 job and the file format used when run on 5 executors.

File format	Execution time	Average Execution time
Text	16m52s, 16m, 15m4s, 14m47s, 15m4s	15min 33.40s
AVRO	36m14s, 38m28s, 35m56s, 36m4s, 37m27s	36min 49.80s
Parquet	38m43s, 36m58s, 36m16s, 37m46s, 35m50s	37min 06.06s
Parquet (snappy compressed)	34m52s, 36m55s, 36m19s, 35m38s, 35m51s	35min 55.00s

Table 3.1. Execution times of part 2 job with different file formats and 5 executors.

Table 1.3 shows the execution times calculated for each run of the Part 3 job of homework 2 and the file format used when run on 5 executors.

File format	Execution time	Average Execution time
Text	3min52s, 4min6s, 3min49s, 3min55s, 3min57s	3min 55.80s
AVRO	3min2s, 3min6s, 3min12s, 3min17s, 3min6s	3min 08.60s
Parquet	3min9s, 3min6s, 3min2s, 3min8s, 3min1s	3min 05.20s
Parquet (snappy compressed)	3min8s, 3min4s, 3min21s, 3min10s, 3min7s	3min 10.00s

Table 3.2. Execution times of part 3 job of homework 2 with different file formats.

And Table 1.4 shows the file sizes saved to disk of different file formats. The file size was obtained using the command “hdfs dfs -du -s -h <directory>”.

File format	Inverted Index (output of part 2)	Similarity Matrix (output of part 3)
Text	16.9 MB	10.3 MB
AVRO	4.8 MB	3.3 MB
Parquet	1.8 MB	2.0 MB
Parquet (snappy compressed)	1.8 MB	2.0 MB

Table 1.4. File sizes of outputs saved to disk using different file formats.

Fig. 1.2 and Fig 1.3 show the execution time of part 2 and part 3 jobs against the different file formats respectively when run on 5 executors.

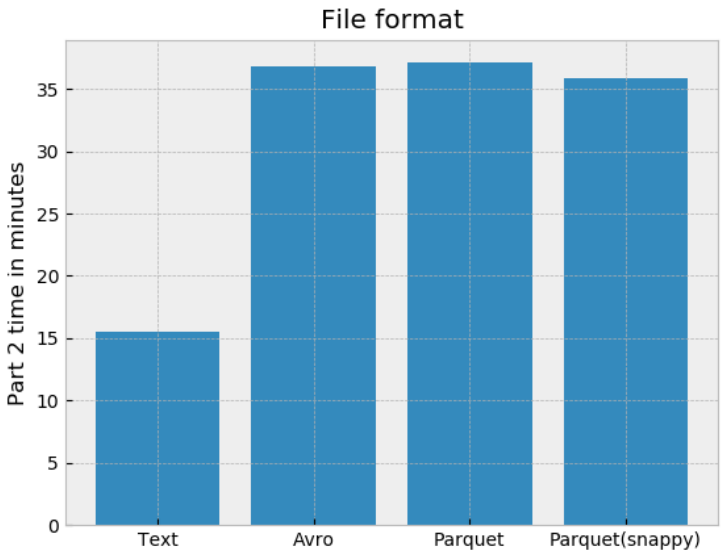

Fig. 1.2. Execution time of Part 2 job vs File format when run on 5 executors

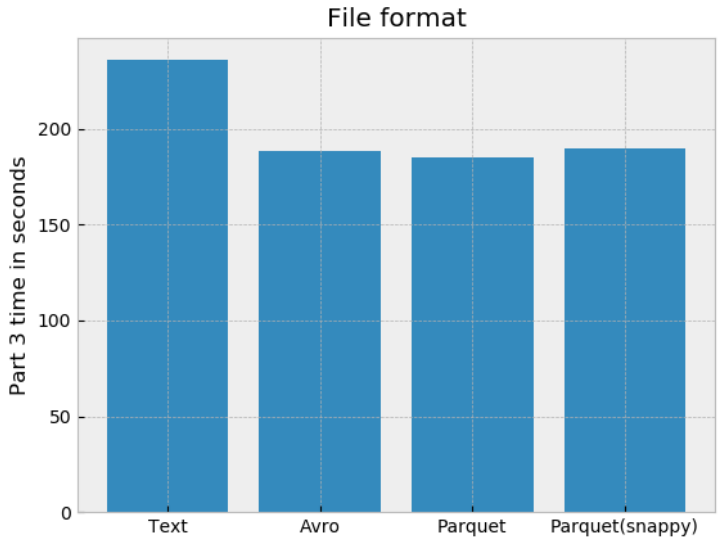

Fig. 1.3. Execution time of Part 3 job vs File format when run on 5 executors

Conclusions

The execution time of the Spark job decreases as we increase the number of executors as expected. Judging from graph 1.1, we can safely say that Part 2 benefits more

from parallelism than Part 3 as is evident by the steeper decrease in execution time when we increase the number of executors to 15. This is because the algorithm for Part 2 is inherently more parallel than the one for Part 3.

AVRO and Parquet are “write-once-read-many” file formats and as expected the write times of the inverted index (part 2 job) for parquet and AVRO file formats is more than double that of using the text file format.

Since AVRO and Parquet use encoding schemes to compress the data, their outputs occupy very less disk space as evident from Table 1.4.

AVRO and Parquet perform better than text file format on the part 3 job however the improvement isn’t very impressive. This could be because of suboptimal handling of data by the code and the overhead of converting data from dataframe to RDD and vice versa.