Information Retrieval (CMSC 476/676)

Homework 3 – Report

Done by – Anushka Dhekne (CMSC VD19739)

The objective of this project is to build an index for the collection of documents. The index is also called as an inverted file, where, instead of looking up a file to see what terms occur in it, we look up a term to see the files in which the term occurs. The input directory contains files generated as output from Phase 2. The output directory contains two files –

1. dictionary.txt – In this, the records are stored in the alphabetical order and the dictionary should contain 3 fields – the word, the number of documents that contain that word (this should correspond to the number of records that word gets in the postings list) and the location of the first record for that word in the postings file.
2. postings.txt – This file should contain two fields – the document id, the normalized weight of the word in the document.

I have used Python programming language to complete this assignment and Visual Studio Code as the code editor. This report also discusses the improvements in implementation while processing the documents from Homework2 to Homework3. It also includes screenshots of the output files showing, the graph showing time required to process each set of documents and the efficiency of the overall program.

Firstly, the 'processing_documents' function processes each document in the input directory which contains the .txt format files. It converts the file contents to lowercase and removes the stop words from its contents by accessing the stop words list. It also counts the frequency of each token using the 'Counter' library from Python and extracts the document IDs using regular expressions. Finally, it returns a dictionary, where the document IDs are the keys and token counters are values.

The next function – 'calculate_term_weights' calculates the inverted document frequency for each term based on the document frequency which was calculated from the previous function. It stores these document frequencies in a dictionary.

The 'build_index' function iterates over each of the documents, calculating its tf-idf score for each term and storing the document IDs and its respective tf-idf value in the inverted index. Thus, this function creates a dictionary and constructs the inverted index. The function also initializes the postings list and adds the dictionary contents (Document IDs and TF-IDF Scores) to the postings list. It sorts the postings list by document IDs and then by its tf-idf scores.

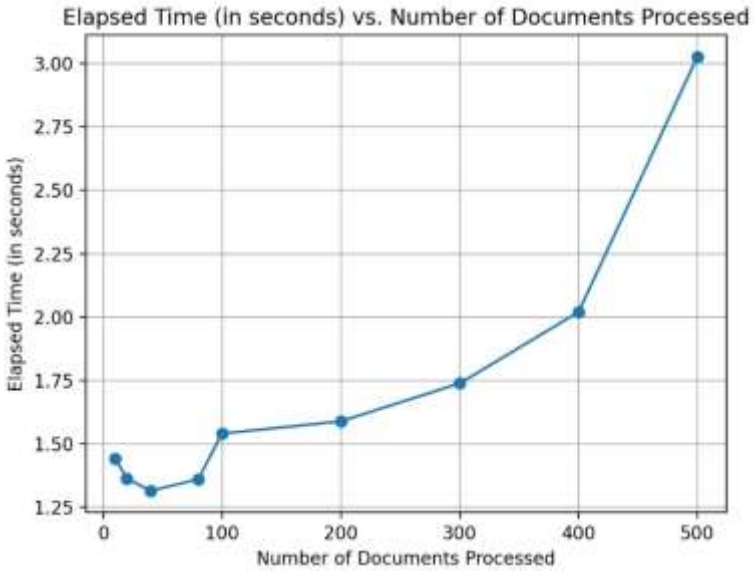
Then, the 'write_to_index' function writes the inverted index to a dictionary file . In this file, each line contains a term, the number of documents containing this term and the position of the postings list in the postings file. This function starts the system time, then passes arguments to each of the functions mentioned above and runs these functions before it ends the system time to calculate the total time required to process these functions or to complete the processing of the documents.

The next function – 'plot_elapsed_time' is used to plot a graph of Elapsed Time (in seconds) versus the Number of Documents Processed. And finally, in the 'main' function, we parse the command line arguments by creating an argument parser object. The time taken by each set of documents (in sets of 10, 20, 40, 80, 100, 200, 300, 400 and 500) is also tracked, along with the total time taken to process all 503 documents from the input directory.

Compared to Phase 2, several code enhancements have been implemented in Phase 3 of the project. Indexing the term documents and sorting these documents have made it easier to fetch or lookup the terms or their weights in the list of input documents from the input directory. Stop word removal from the text during tokenization has improved the indexing process and quality by filtering out the unnecessary and uncommon words from the documents. Tokenization done using the 're.findall' regular expression method helps to identify the word boundaries and extract tokens, which ensures accurate tokenization as compared to the previous method which involved simple splitting by whitespaces.

The code efficiency of the indexing process was evaluated based on the 'time' Python library in terms of execution time on a Linux machine. The time required to process different sets of input documents was measured and its scalability was analyzed. The results show that as the number of documents increase, the time required to process them also increases. Thus, this shows a linear data relationship between the two. Initially, the rate of increase in elapsed time as per the number of documents processing had slightly slowed down, but it increased by around a second for each set of one hundred documents processing. However, this trend keeps fluctuating by 1 or 2 seconds and is not constant. The images shown below depict the results more accurately.

The graph generated by the program is shown below –



The graph shows the relationship between the Number of Documents Processed against the Elapsed Time to process a set of 10,20,40,80,100,200,300,400,500 and 503 documents. The trend of the graph shows that the algorithm is suitable of handling large files and/or a large corpus of files.

The result of the program which shows the total time required to process 503 documents is given below. Once we close the window of the above graph, we get the result as shown below.

```
PS D:\SEM_4\Project\phase3\final_submission> python AnushkaDhekne_Phase3_VD19739.py D:\SEM_4\Project\phase3\final_s
ubmission\input_files D:\SEM_4\Project\phase3\final_submission\output_files
Time taken to process 10 documents: 1.44 seconds
Time taken to process 20 documents: 1.36 seconds
Time taken to process 40 documents: 1.32 seconds
Time taken to process 80 documents: 1.36 seconds
Time taken to process 100 documents: 1.54 seconds
Time taken to process 200 documents: 1.59 seconds
Time taken to process 300 documents: 1.74 seconds
Time taken to process 400 documents: 2.02 seconds
Time taken to process 500 documents: 3.03 seconds
Total time required to process all 503 documents: 15.40 seconds
PS D:\SEM_4\Project\phase3\final_submission>
```

If ‘n’ in the number of documents in the corpus, ‘m’ is the average number of words in each document, ‘d’ is the number of documents, then –

1) The overall time complexity of the code is – O(n.m + dlogd)
2) The overall space complexity of the code is – O(n.m + d)

The overall size of the output files depends on a variety of factors like the number of documents in the input directory, the vocabulary list, the average document length, and the indexing from the postings list as well. As the number of input files increases, the size of the output files also increases (but the increase in the output files size is less than linear as compared to the input files size), since, the vocabulary, postings list also increases accordingly. But the inverted index system allows to restrict the exponential increase in storage requirements as it compresses the storage needs.

Thus, Phase 3 of the project helps to understand the inverted index creation for file indexing and information retrieval purposes. The inverted indexing algorithm is highly scalable, and it scales well for a larger corpus as the compression techniques showed by the inverted index system effectively manages the increasing size of the input files. Thus, large documents can be handled efficiently and in a robust manner.