CMSC 476/676 Information Retrieval
Homework 3 REPORT

JA52979
Sai Teja Challa

Steps to run the program:

In the IDE terminal, You can run the program described below:
python3 index.py "input_direc_path" "output_direc_path".

Here, 'index.py' is the main python program. You must have all libraries installed and your current working directory containing python files in order to run the method described above. The input directory is the output files(tokenized files) of phase 1.

Improvement of preprocessing:

In this phase, I implemented the preprocessing using the same stopwords and the same preprocessing methods from the previous phase where the processing involves removing stopwords, then removing the words that occur only once in the entire corpus, and also the words of length 1. Also, I removed the BM-25 function which calculates the term weight using the BM-25 variant, since I used tf-idf for the term weights.

Term Weighting:

For the term weights calculation, I have implemented TF-IDF, I calculated the product of term frequency(tf) and inverse document frequency(idf), later the TF-IDF score for each term is normalized by dividing each TF-IDF score by the square root of the sum of the squares of all TF-IDF scores in the document, using the same approach from the previous phase.

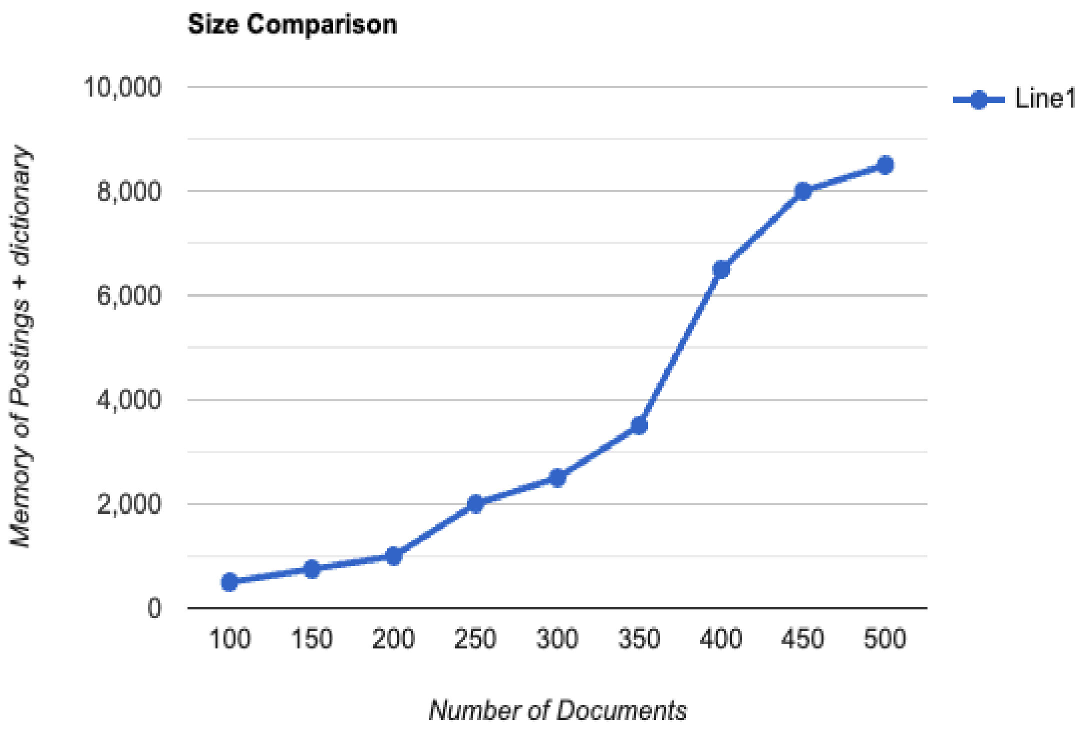
TF-IDF= (TF * IDF)

Dictionary and Postings files creation:

Inorder to build a pair of files i.e,. dictionary and postings files, I created a function "*write_output_files*". So, in the function, I created a dictionary named "*postings*". This dictionary is initialized to hold the postings lists, where the function will store which documents each term appears in and the term's weight in each document. And thereby iterating through the tf_idf_results, which is expected to be a dictionary where each key is a filename (e.g., 001.txt) and each value is another dictionary of terms and their associated TF-IDF weights in that file. It aggregates this data into the postings dictionary, where each term is a key, and the value is a list of tuples containing document IDs and the corresponding TF-IDF weights. After iteration, I created two files for storing dictionary records and postings respectively. Then opening those files, using the 'postings' dictionary, at first the dictionary file is written and then postings file. Since, here an inverted index is built using a postings file and dictionary to maximize the effectiveness of accessing term-related information from a collection of documents. To create a map of word locations inside the index, it is necessary to go through the terms one at a time to fill the dictionary before writing to the posting files. In particular, the dictionary entry for each word specifies the exact byte position in the posts file where the term's detailed postings start, in addition to the number of documents that contain the term. This approach makes sure that every search operation can rapidly consult the dictionary to determine the precise location in the postings file where the term appears. As the postings for a term are being written, the inner loop that writes to the postings file for each term follows the same logically approach.

Therefore, For every term in the postings dictionary, the term is written to the dictionary file, followed by the number of documents that contain the term, and the location of the first record for that word in the postings file. Then for each term, the function iterates over its postings list and writes each document's ID and TF-IDF weight to the postings file.

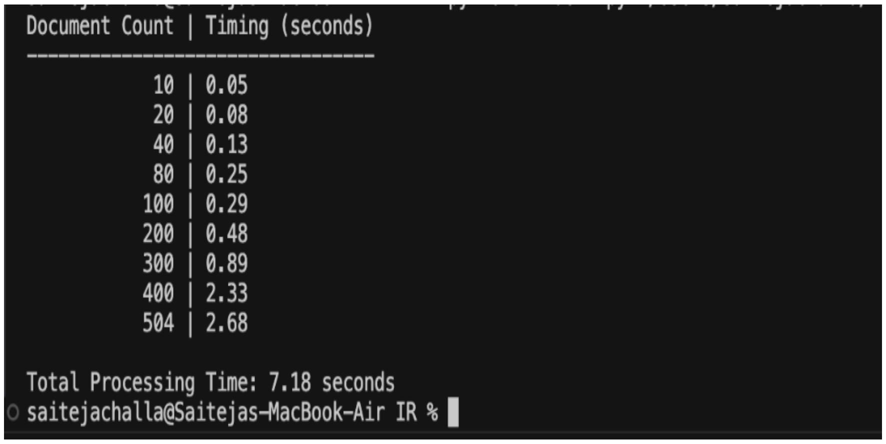
<u>Comparing the size of raw data with the size of dictionary and postings file:</u>
The total size of the postings file and dictionary file together is eight megabytes, whereas the raw data size before any preprocessing is eleven megabytes. Thus, over thirty percent of the memory can be saved. The number of documents and the total amount of RAM used by the posts file and the dictionary file are shown graphically below.



As the number of documents increases, the graph shows that file sizes increase as well. Between 350 and 450 documents, there is a significant spike in memory consumption, and if we were to examine the raw data file sizes from this range, they would be significantly larger than other files in the corpus. Therefore, it is very obvious that the posts file grows in size as the corpus grows.

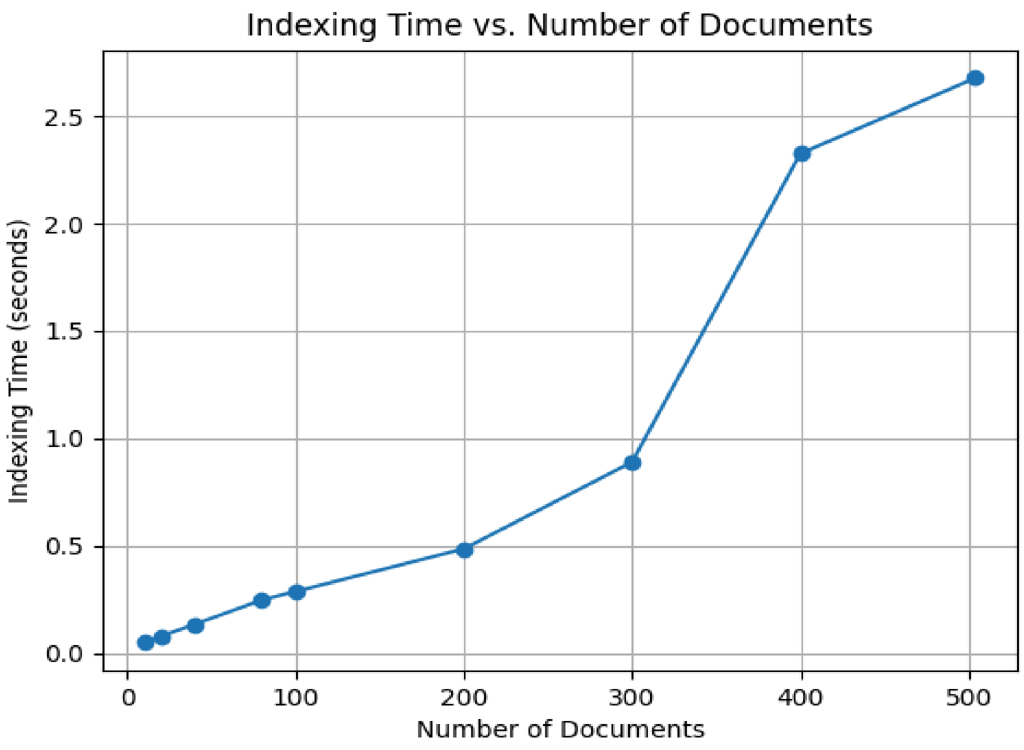
Results:

Below figure depicts the different times of the index function taking on a varying number of documents.



```
Document Count | Timing (seconds)
---------------------------------
            10 | 0.05
            20 | 0.08
            40 | 0.13
            80 | 0.25
           100 | 0.29
           200 | 0.48
           300 | 0.89
           400 | 2.33
           504 | 2.68

Total Processing Time: 7.18 seconds
saitejachalla@Saitejas-MacBook-Air IR %
```



Above graph shows the plot between the time and number of documents processed.

Analysis:

The amount of time needed to preprocess every document, creating the postings, and creating the dictionary file has increased slightly. The number of times we are looping through each dictionary for which we have a document could be the cause of the slight decrease in performance. While creating the postings file, we are iterating through all of the document dictionaries to retrieve the tf-idf scores. We are also iterating through all of these dictionaries to determine how many documents each word appears in. This could have increased the program's complexity, leading to a longer overall execution time.

Hence, upon considering the function which writes postings to the output files, and the compilation of postings and the subsequent file-writing operations, I can say that the overall computational complexity can be approximated as O(T * D) where T is the number of unique terms and D is the average number of documents per term. This shows the overall amount of work performed in comparison with the number of term-document pairs processed..