CMSC 476/676 Information Retrieval
Homework 2 REPORT
JA52979

Steps to run the program:

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

Here, 'calcweights.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.

Implementation:

Improvement of preprocessing:
For this phase of the project, to improve the preprocessing, I made a list to store all the stopwords given. As mentioned, making use of the list, I removed stopwords, then removed the words that occur only once in the entire corpus, and also the words of length 1. Thus, the resulting number of tokens for all the documents have been slightly decreased, making my term weights more effective.

Term Weighting:
For the 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.

TF-IDF= (TF * IDF)

For calculating the term frequency, i.e., the number of documents that a particular token is appearing in. I calculated using the below formula,

TF= (Freq. of token within a doc) / (Total number of tokens in the same document)

Similarly, for calculating the inverse document frequency, using the below formula, calculations are made.

IDF= log(Total number of documents / Number of documents containing the term)

I have implemented the above logic using the 'calculate_tf_idf' function. By maintaining a two dimensional dictionary, where I stored all the tokens and each token maintains its own dictionary of all the documents it appears in and its frequency in each document it appears in. Later, corresponding document frequency of all the tokens is stored in a dictionary. The above mentioned function takes 'document_tokens' dictionary containing mappings of filenames to lists of tokens from each document, 'df' dictionary containing mappings of each term to its document frequency and total number of documents as inputs. Thus using all these inputs and

formulas each individual term weights are obtained, which is a dictionary mapping filenames to another dictionary of terms and their corresponding TF-IDF scores.

BM25:

For the BM25, I calculated the product of inverse document frequency(idf) and term frequency(tf).

BM25= (IDF * TF)

Using the above inverse document frequency formula,

IDF= log(Total number of documents / Number of documents containing the term)

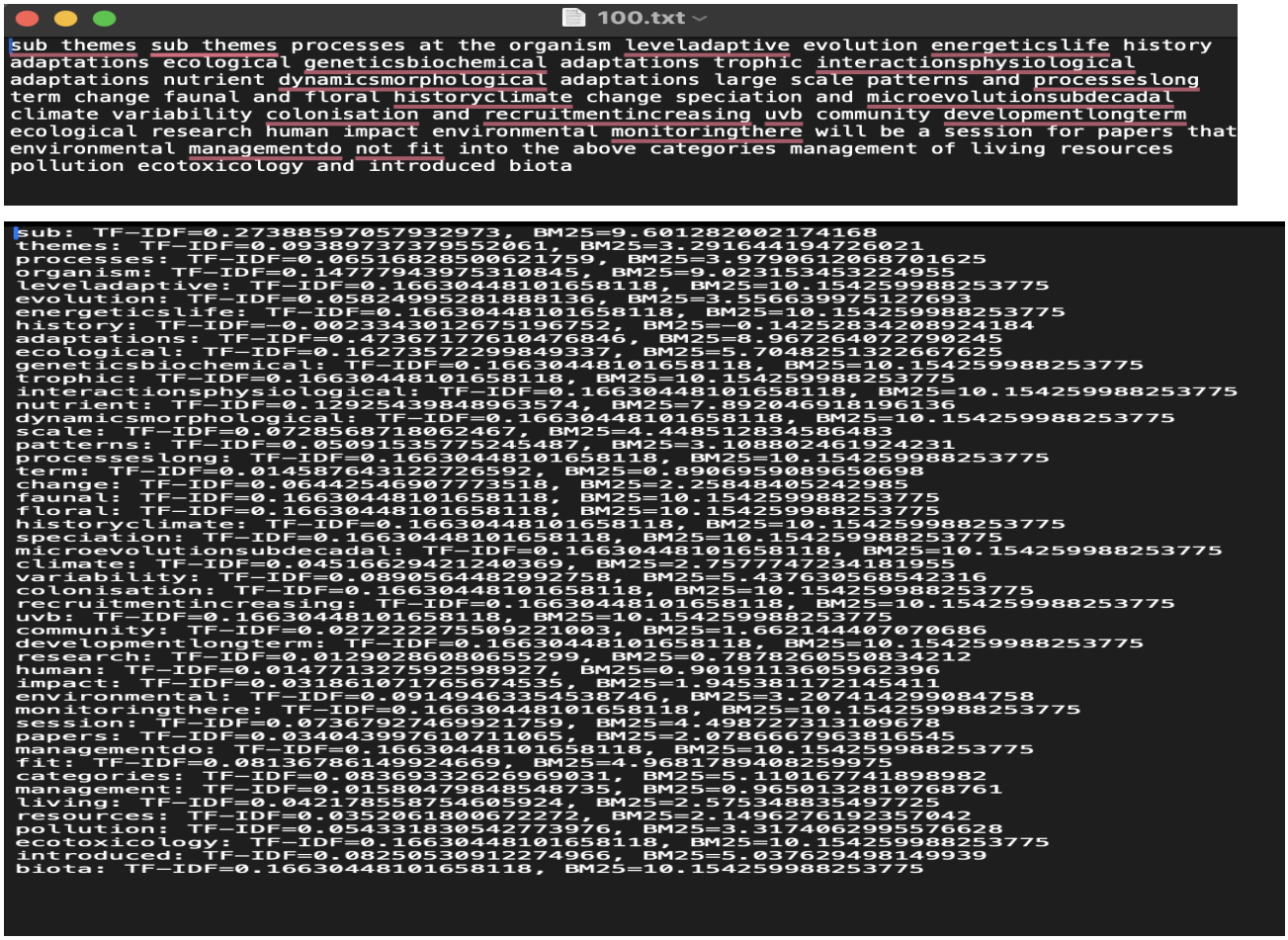
For term frequency, I used below formula,

TF= ((tf * (k1 + 1)) / (tf + k1 * (1 - b + (b * (|D| / avgdl))))

Using the same dictionaries mentioned in TF-IDF calculation, using the IDF and term frequency which is already calculated. Additionally, we need to calculate the average length of the documents, which we determine by multiplying the length of all documents in the corpus by the total number of documents in the corpus. To normalize the impact of a document's size on a term weight, we use this particular formula (length of document/average length of documents). In BM25, there are two tuning factors: k and b. term frequency scaling is controlled by the tuning factor k, where a larger k value indicates a higher priority for term frequency. To avoid that, I've used the recommended value of 1.2 for the k. The tuning factor b is used for managing the document length scaling where shorter documents are assigned greater weight when the value of b is set higher. In this case, the suggested value of 0.75 is used for b.

The following are the outputs I have got for the document 6 & 100..

Above figure depicts the file-006 input and output files.



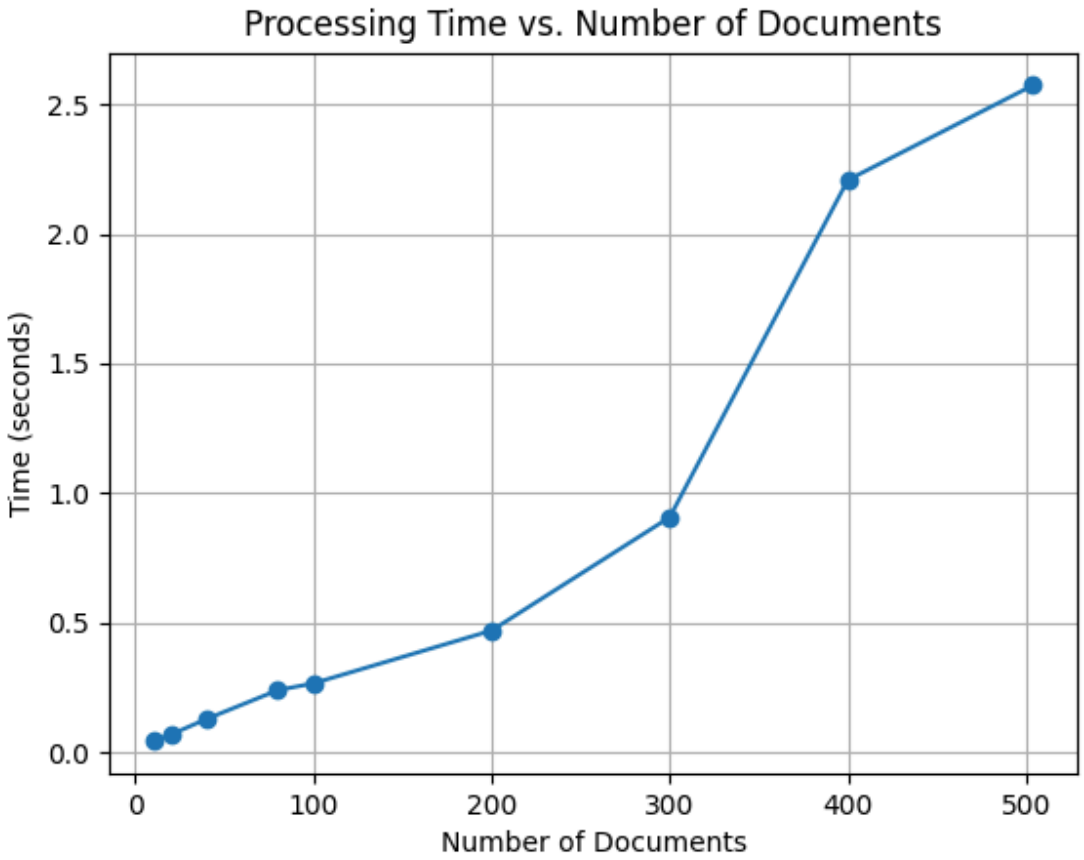
Above figure depicts the file-100 input and output files.

Finally, after calculating the corresponding term weights, I maintained two dictionaries for storing the tf_idf and bm25 results. Using those dictionaries , respective term weights are written to the respective individual file. Also, during processing the documents, considering the mentioned document count set, respective timestamps are calculated and stored in a list. Using that list and the document count set, a graph is plotted.

Below figure shows the timings of term weight calculation on a varying number of documents.

Document Count	Timing (seconds)
10	0.05
20	0.07
40	0.12
80	0.25
100	0.26
200	0.49
300	0.91
400	2.20
504	2.60

Total Processing Time: 6.97 seconds



The graph above depicts the plot between the time and number of documents processed.

Analysis:

From the graph, we can say that the processing of documents for TF-IDF and BM25 scoring is linearly scalable with the number of documents and the number of tokens within those documents. The execution times for varying numbers of documents have increased significantly, due to a significant rise in program complexity. Also, to run the program effectively, I have used a lot of temporary lists and dictionaries in addition to performing repeated reads and writes.

Thus, I can say that the overall time complexity can be approximated as O(d×t), where d is the number of documents and t is the average number of tokens per document. Since it involves processing of documents including tf-idf and bm25 scores calculation.

References:

[1] https://en.wikipedia.org/wiki/Tf%E2%80%93idf
[2] https://en.wikipedia.org/wiki/Okapi_BM25
[3]https://www.elastic.co/blog/practical-bm25-part-3-considerations-for-picking-b-and-k1-in-elasticsearch