**Ambrose Tuscano**

**CMSC 676 Information Retrieval**

**Spring 2020**

**Project 3 Write Up**

**Introduction :**

Once an Information Retrieval system has processed all the input files, it needs to store in which documents each term appears and what weight does it have for the particular document. This makes it easy to retrieve a document with respect to the token’s relevance for it.

The index file includes the token, the document it appears in, and the its starting point in the postings file. The postings files is a list of records, each includes the document ID (in this case, a string representing the filename) and the term weight for that term in that document.

I choose to sort the index file alphabetically by token to provide for easy parsing and possible binary search.

**Implementation:**

The implementation of this program is nearly the same as in previous program for tfidf calculation, with some optimizations which included using ***NLTK*** library function ***word\_tokenize*** for creating tokens, I figured out that it was a efficient method and that, during sorting tokens alphabetically it gave no errors(which previous solution didn’t handle well). Further the weighting code was changed to create a matrix implementation where each token weight was stored as per its value for a particular file in the document folder, so it was easy to access it later for the indexing and posting part.

I created 3 functions to organize the code better.

1. **Preprocessing**

First Function(prepop) looks after the Preprocessing part. Also, while doing so it calculates the term frequency, while doing so it also maintains frequency of the word in the file and number of files in the whole corpus, which contain the particular token. For Term Frequency(tf) we calculate it using particular word frequency in the file upon total number of words tokenized.

1. **Weight calculation**

Second function was used to calculate idf and simultaneously save it as a matrix of tfidf of word relative to particular file. This was done using :

tfidf[file][token] = tf[file][token] \* idf

Having such a format made it easy to use for further indexing. Everything else till this part is same as previous project.

1. **Indexing**

This Function takes the files, the tokens and the tfidf matrix as parameters. I have created a structure to save the token, its documents and posting number, alternative way was to use a an dictionary using Hash key and an array structure, but I figured out that structures like in C language can be better utilized and more efficient. Reference for the same was -https://dbader.org/blog/records-structs-and-data-transfer-objects-in-python

The structure maintained the token, its number of appearances i.e. postings and the index to its starting in posting file.

In the **indpos** function, I loop for every token in every file. I have used the sorted function here so the Indexing is done in a alphabetic ascending order of the tokens. This helps in finding the tokens easily while retrieving due to a binary search element being involved.

The index array is appended with the structure, which holds the token, number of files with its occurrences and its posting index for better search in the posting file.

For each token, the documents are retrieved and its relevant weight to the document is added, this is saved in an array.

1. We Needed to plot graph of the timings needed to index a given batch of file, so I do that in the later part and call the above three functions in the loop which gets the solution done better. Time for each function is recorded.

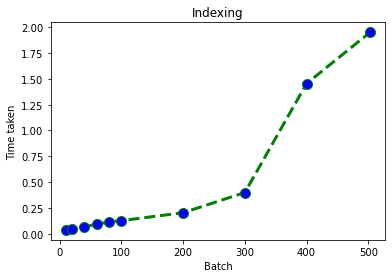
**Conclusion:**

Interestingly of the approximately 313,000 tokens in the unprocessed documents, only around 76,460 remained. I have kept singleton tokens too, they can be eliminated as they might be typing errors or give little meaning. My whole program runs in approximately 49.127525329589844 seconds, that includes the 10 loops it takes for indexing the files in batches for graph calculation.

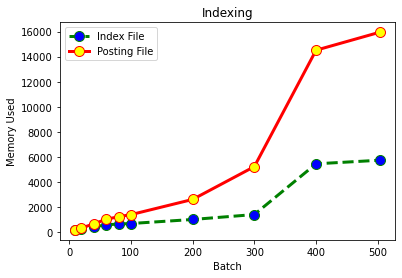
Documents of other languages, such as 373.html which is of Hungarian are indexed too.

The program executes indexing the 503 files along with preprocessing and weight calculation in approximately 22 seconds. Indexing of the 503 files after tokenization and weight calculation is just a little less than 2 seconds. We can see that as number of files increases the time taken increases both for preprocessing and indexing. But it is observed that the time taken overall is mostly for preprocessing and there are little fluctuations in time for indexing(which increase to +1 seconds as the corpus size grows above 400 files). (Refer Fig 1)

Also, the space taken for indexing 400 and 503 files is considerably more then indexing a smaller corpus. For 503 files the programs creates a index file of 5750.75390625 kb and a posting file of 15943.794921875kb. Fig.2 gives the space graph.



**Fig 1: Indexing time**

****

**Fig 2 : Memory Used**