Tristan White
December 4th 2020
CSCE 4200

CSCE 4200 Project 3

The program begins by having the user choose 1 or 2 from an option list. 1 can only be run if 2 is run first. 2 creates the necessary tokens and indexes. The program after selecting 2 begins by going through the given directory making each file name into a string of vectors. From there, the string vector alongside class instances of the word_dictionary and file_dictionary is passed into the tokenizer function. The function reads in each file line by line. Each line is tokenized and tested to see if it will be further added to the necessary data, ignored, or turned into the name of the document.

The first condition tested is if the line contains “<DOCNO>” and “</DOCNO>”. This is to find the name of the document. This name string will be passed into an if statement where it tests to see if the name of the document is different as the previous time it passed around, the vector of accumulated is added to a 2d vector to be used in indexing later.

The remaining lines are run through a tokenizer of various “if” conditions. The first thing it checks after eliminating any whitespace at the front of the string is if the line starts with an ‘<’. This is to catch an HTML tag alongside any tokens that begin with ‘<’ since we aren’t supposed to catch words that have no alphanumeric characters. Next, the tokenizer checks to see if any ‘-’ are inside the string. If the ‘-’ is found, then the token is split at the ‘-’. From here, all non alphanumeric characters are erased. After this the token is put into lowercase and is checked to see if it is a number. If it is a number, then the token is scanned to see if it is a stop word. If it isn’t then the token is placed in 1 of two indexes. One for indexing, and the other for word dictionaries. If no ‘-’, then the whole split string thing doesn’t happen and the tokenizer does the steps after non alphanumeric characters were removed. This will repeat until all files are read.

It should be noted that the program goes through tokenization and stemming twice to save memory the vectors hold. First time tokenizing is for the Collected_tokens and the second time for the words_per_doc. All duplicates are removed from collected_tokens after sorting the entries in alphabetical order. The 2d vectors replicas are not deleted as they will be used to determine the frequencies later.

The process of stemming the two vectors begins by declaring a bool check to determine if the newly stemmed word is a duplicate. The program goes across the collected_tokens vector and has the stemmer stem each element. Then all previous elements which have been stemmed are checked. If there is a previous element that matches, then the bool check is set to true and the loop is broken. An if statement checks to see if the check is false and if it is, the word is passed into the make_dictionary function where it is placed in the map word_dict with the key being the iterator j that is added to each addition.

The indexing begins with the forward index. The 2d vector created earlier has its rows set as document and the columns as words in document. Each column is parched through stemming the element. This is passed to function addDocument along win doc number, and the created word dictionary. The function creates a vector of structs to contain the word id and frequencies of each document. The word passed in gets the id, and checks to see if the word is

already in the vector. If not, then a new doc is created with this word as the first element. If it is in the vector, then the word is passed to function addnode with the key, document number, and the word dictionary. In this, the word goes through the struct vector and compares the keys with the key that was passed in w. If they are the same, the frequency has 1 added to it. If the word isn't in yet, then a new stuct is created and added.

After this, the inverted index is created. From here the function addword is called alongside the word dictionary and file dictionary. A struct is made with the documents and the frequency each word appears in them. First each entry of the word dictionary is gone through getting the key. From here the forward index is parsed. If the word shows up in a doc, then the frequency is copied into the struct alongside the doc id. This struct is put into a vector. After all documents are gone through, then the word id and vector of structs are put inside a pair and then placed into a vector. This vector is the forward index.

Bellow in figure 1, there is information on the time taken to tokenize and index
Info made with 15 documents
Time to make the forward index: ~1222 seconds
Forward index size: 5367

Time to make the inverted index: ~289 seconds
Inverted index size: 33581
Total Time (Tokenizing and Indexing): 1790 Seconds

Total index size: 33581
Main.grels provides 10 relevant items
Figure 1

After this, the program will stop and tell the user to choose '1' to run the search program. When chosen it will call the function query_processor which in turn will call topics_file_reader, and will read in the topics.txt file. The topics.txt file is parsed in the topics_file_reader finding the topic number, then parses the title, and desc of each topic. The user enters 1 for the title to be parsed and 2 for the title and desc to be parsed. To find the tokens, the program uses different string tests to ensure that the right words are taken from the string. The words taken are determined by what the user entered earlier. The program then makes sure the terms aren't stop words, and the remaining words are stemmed with the same porters algorithm used before. These are placed in a vector of pairs with one being the topic number, and the other being a vector of terms.

In order to retrieve documents from the two datasets, we use a cosine score algorithm provided by the book alongside tf-idf weighting to find the weights needed for the algorithm. The program calls the cosine_similarity function to do this. The program goes through each query term adding to the scores[N] array. When this is done, the top K terms, in this case K=10 are printed out to the console and to the file vsm_output.txt. From here the user must manually judge to find out if these results are good. The user can also use the "main.qrels" file to quickly refer to relevant.

The results from parsing topics.txt in figure 2. Results are based on the results in main.qrels. Each variant was tested once.

Topic 352 did not retrieve properly with just “title” or with “title” and “desc”

Topic 353:
With just “title”

Using K = 10 for precision
Precision 3/10
Recall 3/10

With “title” and “desc” there was no relevant information retrieved

Topic 354:
Just “title”
Precision 4/10
Recall = 4/9
With “title” and “desc” there was no relevant information retrieved

Topic 359 did not retrieve properly with just “title” or with “title” and “desc”
Figure 2

As seen in the findings presented above, the search engine seems to get more inaccurate if the query is expanded which is the opposite of what I thought would happen. Regretfully due to time restaurants from other classwork, I was unable to get the narrative function to be parsed properly. There also could be problems with implementing the cosine similarity function such as the wrong variable being used, or not calculating the weights properly.