Information Retrieval Phase 3 Report

PROGRAM STRUCTURE

1. Open and read through all the html files
2. Using BeautifulSoup library to parse through all the .html files and extract only the text from it
3. Tokenise the extracted words
4. Handles cases such as special characters, numerical inside a word, and also decimal numbers
5. Converts all the words to lower case
6. Reads the text file stopwords.txt and stores the given specific stop words in a list
7. The extracted words are now passed into a counter to get the word and its frequency value
8. Checks the given conditions that is
    a. Words of length 1
    b. Words with frequency 1
    c. Words that match the stop words
9. Updates the counter filtering out the words from those three conditions
10. Computes the term frequency and stores the value in a global dictionary with file name as key and another dictionary as value, containing word and tf
11. Computes the value that gives the occurrence count of any given word in number of files
12. Computes the inverse document frequency and stores the value in a global dictionary with file name as key and another dictionary containing word and idf as value
13. With these values the tf-idf is calculated for each word in the global dictionary
14. With the tf-idf values we create a Document Term Matrix
15. With the help of TDM and tf-idf dictionaries we create the posting file.

EXPLANATION

The program is built on top of the previous submitted logic with few minor changes and with better efficiency.

The developed program successfully pareses through all of the html files with the help of beautiful soup library and tokenises all of the words. The main basis of tokenisation is spacing. The inbuilt function '.split' is used to split the words extracted.

The program handles punctuations and special characters using regular expression. We give a set of predefined symbols that is filtered out of the extracted and tokenised words. Then all the tokenised words are converted to lower case.

The program then reads the given set of stop words from a text files and filters out the words based on the three given conditions. Then the counter is updated accordingly with the new values. The globally declared dictionary has the file name as the key and a counter of word and its frequency as its value.

The **compute_tf** function in the program then takes this dictionary as the parameter and computes the term frequency. It first computes the word count of every file and stores

the file name and its word count in a dictionary. Then with the word count of that word and total count of words in that file term frequency is calculated. This is done for each and every word in the dictionary and the dictionary is accordingly updated with the new set of values.

Then the **doc_containing** function takes a word as its parameter and calculates the count of documents that word exists in and returns the count.

The **compute_idf** takes the dictionary as an argument and computes the inverse document frequency of the words. It first checks the number of documents to be checked and stores that length in the variable num_doc. For every word in the dictionary it calculates idf with the formula, log of number of documents divided by the word count(calculated using the doc_containing function). With these values it updates another dictionary with the key as file name and another dictionary as its value with the word and its idf.

The function **compute_tf_idf** takes the dictionary as its argument and then multiplies the values of tf and idf to compute the tf-idf of every word. It then writes the obtained value to a text file.

The function **calculate_tdm** forms a document term matrix from the tdf-dictionary. And populates an another dictionary with words as its key and a dictionary with files names as its keys and tf-idf values of that word as its value.

The function **querying** takes in a query and is iterated for every query term, for every document in which the query term in present. It first filters out the stop words from the query input. The time complexity is roughly number of query terms times the number of the documents which contain the term. Then outputs a ranked list of documents based on the query.

The program takes about 21 seconds to complete its execution

IMPLEMENTATION

I extended the program from Phase 3 for implementing the retrieval engine. So I could reuse some of the data structures such as term document dictionary which made the task easier. The retrieval engine takes in queries which are list of words with their term weights and runs the same preprocessing method on them as was used in previous phases. After this step, it was converted to a dictionary with query terms and weights. The retrieval engine takes in queries which are list of words with their term weights and runs the same preprocessing method on them as was used in previous phases.

query = "1.0 international 1.0 affairs"

The parameter is a string with query term weights and query terms. The output is a ranked list of documents with respect to the query given. The order is based on the document-query similarity scores. Only the first ten documents with highest similarity scores are displayed as a list of tuples with document id and similarity score as first and second elements of the tuple respectively.

RESULTS

diet
[('018', 0.33586740206109045),
('009', 0.022968510679065603),
('252', 0.0131670382556083)]

international
affairs
[('219', 0.17027167643752877),
('138', 0.1384970295805471),
('143', 0.10933976019516876),
('125', 0.08777980748062846),
('161', 0.0814688409297336),
('133', 0.07921243846711823),
('117', 0.06425119928994454),
('197', 0.061783061522920645),
('205', 0.060567214102280074),
('247', 0.05129519614094337)]

Zimbabwe
No results found

EXECUTION

The program was coded in PyCharm. It can be run on any IDE that supports Python.
Imports,

1. Import os
2. Import glob
3. Import math
4. from collections import Counter
5. from bs4 import BeautifulSoup
6. Import time
7. Import operator