Information Retrieval Programming project -4 Report

Introduction:

The main aim of this project phase 4 is to create a command line retrieval engine on the top of the inverted index that was developed in project 3. The retrieval engine retrieves the words given as query in the command line by matching the words in the query with the words in the inverted index. The project was developed using python as programming language. Linux is used as operating system. The following is the way to execute the code,

$python3 retrieve.py files wts files <"Query term weights with query word">

retrieve.py – the python code

files – input directory

Query- query given for search

Input :

The inputs for the project are the 503 input html files which has to go some preprocessing to extract only the content leaving the html tags from those 503 files. The content of the input files are tokenized and given as input in order to calculate the term weight and to create the index for the terms. The input files are from 001 to 503 html files with .html extension. In project four , the input also involves the query for which the same term weights are calculated.

Output:

Output for this project phase 4 is the name of the files that contain the given query words. Output is displayed in the command line interface as list of files for the given word.

Implementation work:

The input files containing the html files are used to extract only the content of those pages by importing and utilizing the html2text python module. This extracted content is tokenized using nltk tokenizer available in python. I have removed the list of stop words from the tokens by iterating it through the list of stop words in a separate file stoplist.txt. This improves the quality of tokens in the documents . The stop words removed tokens in the documents are stored in a nested dictionary containing files names as keys , values are tokens and frequency. This dictionary is iterated and the length , average length and total length of the documents are calculated. I have implemented two separate classes one for tokenizing the documents and one for calculating the term so that the program is modularized in a better way. I have also created a separate class for index that have a nested dictionary to store the term, document id and the first location of the word in the document The program is design using object oriented principles. I have created the index using python nested dictionary that has keys term , doc id and value as the first location of term in the document.

Since the phase 4 requires a retrieval engine , I implemented the command retrieval engine using a python library called docopt . The given query word goes through all the preprocessing as the files went through in previous phases of the programming project . The term weights are calculated for the query and are stored in a dictionary as query terms and its weights.

Term weights formula:

The term weight is calculated by the taking a product of term frequency and inverse document frequency.

Term frequency = Frequency of a term in a document / total frequency of the words in all the documents

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

The above is the formula for the calculating the term frequency in the documents where

$n_{i,j}$-number of terms i in a document of j

$\sum_k n_{i,j}$ - Total frequency of words in all documents

$tf_{i,j}$ - Term frequency

Inverse Document Frequency :

Inverse Document Frequency is the logarithmic division of n documents in the whole collection by the ni which is the number of documents in which the term occurs . Idf is computed for the entire corpus whereas tf is computed on a document by document basis.

$$idf_j = log\left[\frac{n}{df_j}\right]$$

The Term weighting product is calculated by taking the product of term frequency and the idf. The formula is shown below .

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

Indexing:

The index is built using the nested dictionary in python . For each term, the id of document which it appears in as well as normalized weight of the term is stored. A counter variable was used to record the last line number written in file. The document frequency for a term was obtained from the inverse document dictionary (inverse_doc_freq[term] returns number of documents that contain that term) built in phase two. After going through all the tokens, the list of tuples was sorted according in alphabetical order of terms and the result was written to the dictionary.

Similarity scores:

The similarity scores for the query is obtained by taking the dot product of the query vector and the row corresponding to the document in the term document matrix. Since ,the term document matrix was implemented using nested dictionary , the document in which the term appears can be obtained by retrieving the values from the keys in the dictionary. The score is calculated by taking the dot product between the query term weight and the term weights in the term document matrix and summing up the similarity score of the query with each and every related documents retrieved .The summed value is stored in another dictionary with doc id as keys.

Algorithm complexity and result:

The outer loop is iterated for every query term while the inner loop is for every document present in the term document matrix. The complexity is calculated by taking the product of number of query term times the number of documents that contains the query terms.

Result:

After the calculation of the similarity scores , the scores that are non zero are stored in list containing the document id and the score and are sorted in descending order based on the scores and this sorted list is displayed as output.

Output:

Below is the output I acquired for the following query words,
sophia@sophia-VirtualBox:~/Desktop/Kennedy_Sophia_HW3$ python3 retrieve.py wts files "1.0 diet"
[('353', 15098.38464691065), ('152', 8195.900062784407), ('050', 3397.8835676960357), ('252', 2297.9823960758367), ('009', 2091.6619951897706), ('263', 2061.7811095442025), ('018', 149.40442822784075)]

sophia@sophia-VirtualBox:~/Desktop/Kennedy_Sophia_HW3$ python3 retrieve.py wts files "1.0 international 1.0 affairs"
[('433', 37521.080436799835), ('434', 30815.204195588114), ('435', 29576.349702589807), ('437', 24195.586730599567), ('436', 21878.49753481162), ('348', 14684.284431201278), ('340',

14397.649307684449), ('345', 14387.765337908007), ('351', 12766.980273225578), ('361', 11986.614092624255)]

sophia@sophia-VirtualBox:~/Desktop/Kennedy_Sophia_HW3$ python3 retrieve.py wts files "1.0 Zimbabwe"

sophia@sophia-VirtualBox:~/Desktop/Kennedy_Sophia_HW3$ python3 retrieve.py wts files "1.0 computer 1.0 network "

[('435', 64629.869502379195), ('434', 62975.90738878355), ('437', 47393.082970007505), ('433', 43119.35147378963), ('436', 18577.997043209372), ('376', 12845.855884951789), ('355', 12335.65869880152), ('043', 11163.72060784249), ('368', 10865.684627814117), ('282', 7508.991259528444)]

sophia@sophia-VirtualBox:~/Desktop/Kennedy_Sophia_HW3$ python3 retrieve.py wts files "1.0 identity 1.0 theft "

[('043', 28023.557240577116), ('360', 16189.627469188612), ('348', 14129.071789734413), ('243', 9196.418501687129), ('303', 9015.723619027298), ('309', 8270.753488763088), ('019', 7256.326077339482), ('308', 6447.9542338612955), ('383', 5298.798181857992), ('272', 4989.714829939862)]

sophia@sophia-VirtualBox:~/Desktop/Kennedy_Sophia_HW3$ python3 retrieve.py wts files "1.0 hydrotherapy "

[('273', 2016.640327938001)]

sophia@sophia-VirtualBox:~/Desktop/Kennedy_Sophia_HW3$ python3 retrieve.py wts files "1.0 this 1.0 is 1.0 the 1.0 information 1.0 retrieval"

[('249', 20091.72560668682), ('027', 18751.996520821765), ('267', 6564.605604459136), ('364', 6347.587464660435), ('376', 5139.918778102687), ('365', 5135.876372457305), ('341', 4926.681880308808), ('349', 4724.056297334056), ('377', 4662.91491194766), ('369', 4654.830100656896)]

I have sorted the term weights in decreasing order.

Conclusion :

Thus in project phase 4, a retrieval engines is implemented on the top of the inverted index which was developed in project phase 3.