Report draft:

MODULE Tester

The following project of ours has been executed sequentially as follows, the first step being to process the text of the corpus. Via regular expressions, (with the assistance of imported library “re”), we have compiled the text removing all of its hyperlinks. The tag expressions </a> have been utilised to recognize and remove hyperlinks from the text, replaced by a blank space.

Now comes the portion where for each doc id mentioned in the corpus, a separate document directory has been created to hold the text. Next, we employ a soup data type from the imported library “Beautiful soup”, which acts as an html parser and for each of the document directory created with regard to a specific Doc id, the snippet starts populating the directory with the text of that particular document. After parsing through entire text corpus and segregating texts in their respective doc id file, a set of text files have been created and saved in the documents folder. This is for manual perusal and to observe relevance of data or results of rank retrieval in the later stages.

Each document is specifically referred by its ID.

This procedure is followed by lower casing all words, as well as removing punctuations and replacing them with a space. Also, certain specific apostrophes in a different text font have been separately mentioned and replaced.

This processing of texts to lowercase and remove punctuations has been a specific effort to tokenise the words, via importing library nltk, adding them to dictionary, and specifying it as a “list”, to avoid word repetitions in dictionary.

Using the document id and the formulated tokens, we start filling an inverted index “dataframe df”. We populate for each doc id in a row, the term frequencies of the dictionary in the respective doc id as its columns.

In a process of creating a doc vector, we first enumerate all the words and the documents they appear in, and then sum up and take a log using numpy library function “.log10”.

For each document, we multiply the frequencies of log of terms via “lnc” SMART Notation approach and take square root of their squares’ summations in order to formulate a normalized vector considering all “vector magnitude for each of the terms”.

Mistaken log 0 values have been reinstated as 0.

At the end of this module, the data file, text corpus segregation into doc files, their frequencies, words and vectors of documents have saved in the system via “pickling” feature of converting all these python dataframes into byte stream to store in database “Storage”.

MODULE Query \_tester

The query \_tester module incorporates a free text string as a query from the user. The query is similarly processed as the text used for tokenization, that is, lowercase, and unnecessary punctuations removed and replaced by spaces. The same procedure is rehashed over a query as was for a document, by making a list sequence of query words and considering it as a 0th index of query dictionary.

The second major step to match the query with the document and find the cosine similarity was to get a dummy of the previous dataframe with the mentioned words as columns, and for each of the query item (token) in a single query, the frequency is noted and saved. The buffer populates term frequency for each of the given terms in query, along with noting down query document frequencies of the terms.

A major issue being faced is the presence of terms in query that are not in the text corpus, which would make the vector length of the query unequal to the doc vector. The issue can be either resolved by:

* Brute force to calculate zero frequency of the term in the dictionary.
* A more refined way to eliminate query terms absent in the corpus.

Thereafter an array is created for multiplying respective log of term frequency and idf of document frequency, and normalised vector is obtained [according to “ltc” SMART notation] dividing the magnitude in a similar way as is did with the documents (considering log 0 values as 0.

OBTAINING scores.

Now, each doc vector is imported and multiplied as a dot product with query vector in order to define cosine similarity scores, and thus rank the pages based on iterating through top sorted scores.

Thereby, the output shows/ prints top ‘k’ number of doc ID’s with topmost scores.

Assumptions (till now):

1. Query would hold valid terms.
2. During document scoring, the top k documents are sorted assuming each document will have a different cosine similarity score

Issues:

1. How to make the ranking more accurate and relevant?
2. How to reduce the heuristics of query terms having invalid terms?
3. Can there be more time - efficient process rather than going through all documents?