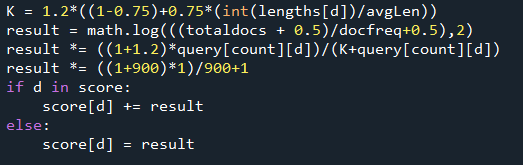
Sallahuddin Khan 20L-2293 Report

# Implementation of BM-25

Code snippet for the BM-25 is as follows:



This code will execute for all the terms in a query and for each term for all the documents found containing the term in the corpus.

In query[count][d] the identifier d is the document and count is the index of the query term.

Here the constants are not labelled but 1.2 and 0.75 are the constants. Score[d] is a dictionary where the scores for each document is being stored. If the score has already been calculated for any query term the value is then added to it.

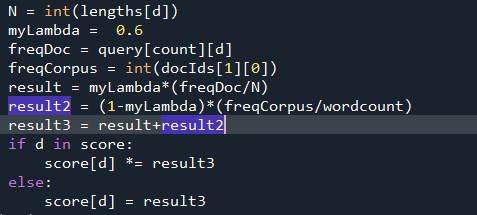
“Lengths” contain the document lengths. DocFreq is the number of documents containing the term.

The cmd command for the execution for this is:



# Implementation of Jelinek Mercer

The implementation for the Jelinek-mercer is as follows:



Lambda is set to 0.6. “freqDoc” is the frequency of the word in that particular document. “freqCorpus” is the frequency of the term in the entire corpus. docIds array contain the exact line of the term\_index where the term was placed. In the first index it contains the total number of occurrences in the corpus.

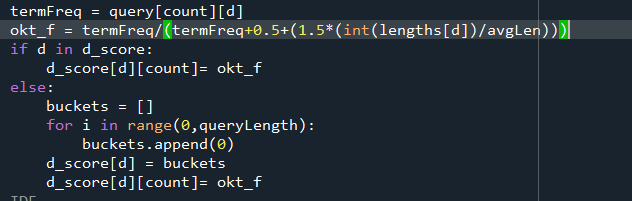
If the term appears in more than one documents the answers are multiplied. Hence, score for a term for each term will be multiplied.

The cmd command for the execution for this is:



# Okapi TF

Over here the score calculation is done in two parts for each query. First a two-dimensional dictionary is made containing the okapi score for each query. This dictionary contains the score for the documents containing the query term for each query term. If the document does not contain the term the index will simply contain zero. The code snippet to find the dictionary of vectors is as follows:



TermFreq is the number of times the term has appeared in a specific document. Length[d] contains the length of the document and avgLen is the average length of the documents.

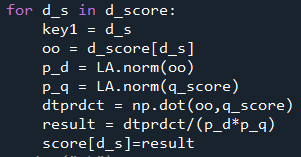
D\_score is the dictionary containing the vectors corresponding to all the documents containing the terms. If the term is not present in the document it will remain to zero that will have no affect on further calculation.

We need a similar vector for the query as well. It is assumed that each query only contains a term once. We treat each as a document an similar calculations will be done:



Now d\_score contains the oktf vectors for the documents and q\_score contains for queries.

Next, we need to calculate the cosine similarity in between the vectors the code is self-explanatory:



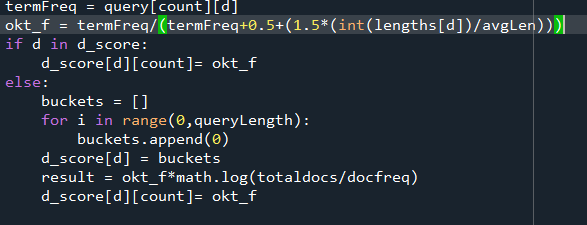
d\_s is the id of the document.

In order to run following is the command:



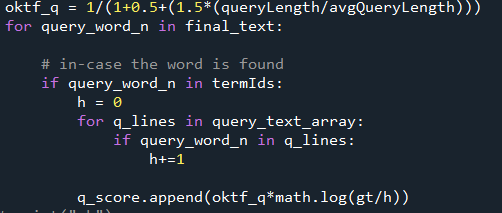
# TF-IDF

The calculation for the TF-IDF is very similar to the Okapi-TF with a difference of one step.



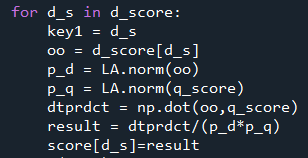
Log of oktf is taken with the total number of documents in the corpus and docFreq is the number of documents containing the term while termFreq is the number of times term has appeared in the document.

Similar for the query:



Extra work has been done to get the number of queries containing the term. Gt contains total amount of words in all queries.

Rest is same as Okapi-TF:

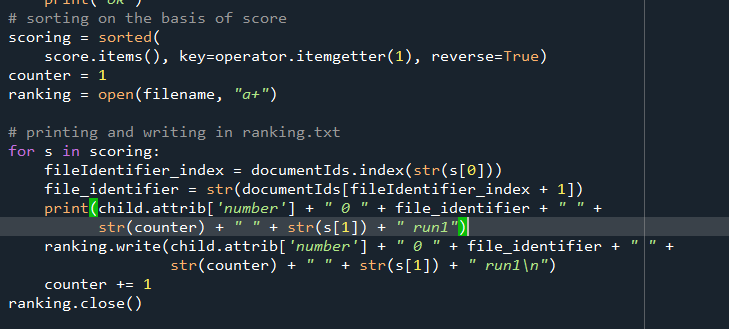


To run following is the command:



The array score contains the score for each document for every query.

Scores are then sorted and printed:



# GAP Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Query No | Okapi TF | TF-IDF | BM25 | Jelinek Mercer |
| 1-204 | 0.008547008547008 | 0.008547008547008 | 0.00793650793657 | 0.02631578947368 |
| 2-214 | 0.526654940014123 | 0.526654940014123 | 0.55461206623643 | 0.05394667808734 |
| 3-216 | 0.378806901540842 | 0.377840800812188 | 0.48729273442390 | 0.19029609210678 |
| 4-221 | 0.316704561132011 | 0.315047238917981 | 0.37757228956896 | 0.08386303232753 |
| 5-227 | 0.139102927362360 | 0.143638836789658 | 0.23076694930930 | 0.04991132288680 |
| 6-230 | 0.269234503029288 | 0.270637268563850 | 0.34315665756156 | 0.00961719244684 |
| 7-234 | 0.737734924576738 | 0.738813511535644 | 0.69920048028277 | 0.08016930273114 |
| 8-243 | 0.254482627252368 | 0.253887987082460 | 0.36148114740089 | 0.09699501455811 |
| 9-246 | 0.116078145245287 | 0.116102622140891 | 0.12493698221503 | 0.00180259546745 |
| 10-250 | 0.439754911051789 | 0.439754911051789 | 0.13634594748647 | 0.00819529668421 |
| Average | 0.318710144975182 | 0.319092512545559 | 0.33233017624218 | 0.06011123167699 |

Following are some of the observations:

* The performance of the queries for Jelinek mercer was significantly poorer than the others. We can see this by looking at the average scoring of the queries.
* Other three methods showed similar results. They showed scores that can be translated to satisfactory.
* 7th query was easier than the others as it showed some good result.
* 1st and 9th query was harder than the others and gave poor scores.
* 3rd query that showed average scores in Okapi, TF-IDF and BM25. Gave relatively better score in Jelinek mercer.
* The seventh query was expected to perform good as it was “dark chocolate health benefits”. All of these words are very common and sparsely scattered in the dataset.
* Third was to perform bad as the query is short and has a full name in it “nicolas cage movies”. However, it performed average deviating from our initial assumptions. Maybe, the word “Movies” had a high influence.
* Similarly, first was expected to and it performed bad. As there was a name of man and military abbreviation. “uss carl winson”
* Fifth query also performed bad. The reason for it being two words were stopwords and were removed. The performance was based on other two words.
* Ninth, did not performed well. As the query had many words. But the number of occurrences In the dataset was not much. Hence the division factor during the queries was high.
* Last query performed average in Okapi-TF and TF-IDF but poor in other two.