**Tokenizer.**

The tokenizer first reads the documents in our corpus one-by-one and then tokenizes the text in that document. These tokens are stored in a list and then sent to a function which cleans the tokens for example removes punctuations. These tokenized documents are then written into a new file and stored in the system.

**Indexer.**

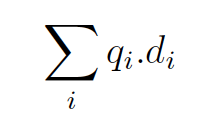
After tokenizing the corpus, we then index the documents. We store the index in the format: *term -> (d, tf)*, where d represents the document ID and tf represents the term frequency. This index is represented in the form of a hash map within a hash map in our programs. The term is the key of the hash map where another hash map is stored. The key of the latter hash map is the document ID (stored as an integer) and this point to the term frequency of the indexed term in that document. This index serves as the input to all the models explained below wherever required.

**Lucene.**

The program uses Lucene Version 4.7.2. We first import the three external jar files. We use the inbuilt SimpleAnaylzer, IndexReader, IndexSearcher, TopScoreDocCollector, and QueryParser classes from the references to the external libraries to tokenize, index, and rank the documents according to the given queries.

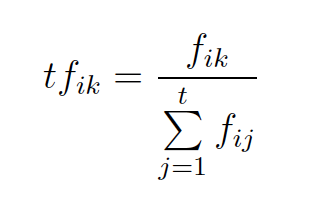
**tf.idf**

The tf.idf program first loads the inverted index in a hash map. Similarly, the query terms are loaded in a different hash map. To calculate the tf.idf score of a document we take the product of the term weights of the document and the term weight of the query for the terms that occur in the query. These products are then summed to get the final tf.idf score.

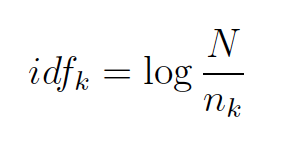


where qi represents the query weight and di represents the document weights.

To calculate the term weight for the document and the query, we simply calculate the tf for that term in the document and multiply it by the idf. We use the formula given below to calculate tf of a term:



where tfik is the term frequency weight of term k in document Di, and fik is the number of occurrences of term k in the document. To calculate the idf score we use

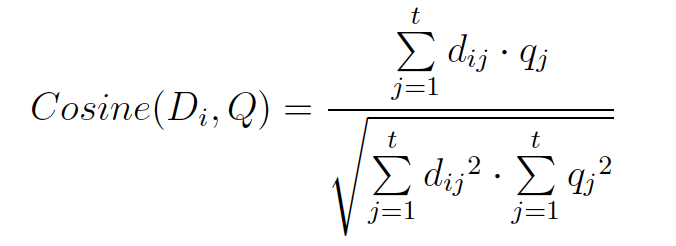


where idfk is the inverse document frequency weight for term k, N is the number of documents in the collection, and nk is the number of documents in which term k occurs.

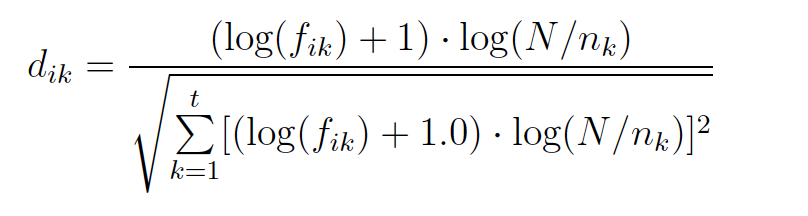
The tf.idf score is computed for the documents and query separately and stores in a different hash maps. These hash maps are then use to multiply the computed scores of the query and the document and give the final document scores.

**Cosine Similarity Vector Space Model.**

The implementation of this model is similar to the tf.idf implementation. We represent the document and query as a vector for all the query terms. These are stored in two different hash maps. The following formula is used to compute the dot product and compute the final score of the document.



The multiplication of the two vectors stored in the hash map gives us the numerator. To normalize the score, we need to calculate the term weights of all the terms in the documents as well. This is pre-calculated and stored in a different hash map. To calculate the term weight for the document and the query, we use the following formula:



Thus, we take the square of all the document weight vectors and query weight vectors represented as a hash map in the program. These two values are then multiplied and the square root of this value gives us the final cosine score of the document for the given query stored in a hash map.

**Stopping.**

Here we are given a list of common words which does not contribute towards the document scoring. We use the BM25 model and then load a common words given in a list. Then while loading the index, we ignore the words that are present in the common words list and not index them at all. Similarly, while loading the query terms list, we check whether that word exists in the common word list. Thus, we run the BM25 without taking into consideration these words and retrieve the documents.

**Stemming.**

For this process, we are given a separate corpus and query list. We first tokenize the corpus and store the tokens of one document in a different file. The indexer then indexes these tokenized documents and gives a new index. We then use this index and feed it to the BM25 search engine and run the stemmed queries to retrieve the document ranked list.

Query-by-Query analysis.

For query: Applied stochastic processes (appli stochast process)

The top five retrieved documents are:

With stemming –

CACM-1696

CACM-0268

CACM-1410

CACM-2535

CACM-1194

Without stemming –

CACM-1696

CACM-0268

CACM-1410

CACM-2882

CACM-1540

For query: portable operating systems (portabl oper system)

The top five retrieved documents are:

With stemming –

CACM-3127

CACM-1591

CACM-1680

CACM-1033

CACM-3068

Without stemming –

CACM-3127

CACM-2246

CACM-3068

CACM-1930

CACM-1461

Here we can see, that for the first query there is not much difference between the ranking is not much different from each other. Hence we could say that the effect of stemming is rather limited. However, stemming could have an adverse effect on the effectiveness of the search engine as seen in the second query as it retrieves non-relevant documents at higher ranks (considering the relevance information as given in the camc.rel file). Thus, we should avoid stemming, at least for languages such as English where the effect is limited and stemming could adversely affect the search results.