BM-25 RANKING ALGORITHM

Dinesh Kannan

kannan.d@husky.neu.edu

INTRODUCTION:

The goal of this module is to implement the BM-25 (or Okapi) ranking algorithm that computes a weight for the relevance of a document based on a search query. This algorithm leverages probabilistic retrieval practices in a way that it includes **document and query term weights to** the **binary independence model**. For the relevance score, the algorithm takes into consideration each query term along with their frequencies. Document related information like term frequencies are obtained from an inverted index that is pre-computed by the search engine.

The formula that computes the BM-25 score for a document for a search query is given by,

$$\sum_{i \epsilon Q} \log \frac{(r_i + 0.5)/(R - r_i + 0.5)}{(n_i - r_i + 0.5)/(N - n_i - R + r_i + 0.5)} \cdot \frac{(k_1 + 1)f_i}{(K + f_i)} \cdot \frac{(k_2 + 1)qf_i}{k_2 + qf_i}$$

The R and r are the relevance information that we have. Q is the search query for which the relevance is needed and i refers to a query term in the search query Q.

MODULE SPECIFICS:

According to the problem statement given several things in the BM-25 formula is pre assumed.

- The corpus under consideration is the standard corpus under the name ‘tccorpus.txt’
- There is no prior relevance info for the algorithm, hence the value of R and r’s is 0
- The value of **k1 = 1.2, k2 = 100, b =0.75**
- The number only terms in the corpus is ignored (not indexed)

The stemmed tokens from ‘tccorpus.txt’ is indexed and stored in the file **‘index.out’**

The file **‘queries.txt’** contains a list of queries with individual terms stemmed as the corpus.

The file '**index.out**' contains a tuple with the following elements,

- Inverted Index of all the stemmed documents in the corpus along with DOC_ID and term frequencies
- A List of unique documents in the corpus
- A Dictionary with the DOC_ID as key and length of the document as the value

The BM-25 algorithm is implemented in the file **bm25.py** which takes in the following as command line arguments,

- Inverted index file, which in our case is **'index.out'**
- Queries file containing the list of queries, in our case it is **'queries.txt'**
- Maximum number of document results

DESIGN:

IMPLEMENTATION:

The program has been split into several modules, each serving a specific functionality.

- bm25
- indexer
- queryProcessor
- converter

1. **bm25 :** This is the main module which implements the bm25 algorithm. This module takes in the index, queries and maximum results as inputs and outputs the DOC_IDs ranked according to their bm25 scores along with the query_id. This module makes calls to converter and queryProcessor modules to convert the received index and queries from string to appropriate python data types for easy manipulation.

2. **indexer :** This module takes in the corpus as a file and converts it into an inverted index. In the index, the document_id and term frequency is stored as tuple against a token occurring in the corpus. The tokens from the file is already stemmed. As per problem statement, number only tokens are ignored and hence not indexed.

3. **queryProcessor :** This module takes in the queries file as input and produces two lists with its two functions, queries() and queryProcessor(). queries() returns a list of queries parsed from the file, queryProcessor() returns a dictionary of all queries mapped against its individual query terms.

4. **converter :** This module takes in the index file as input and returns the result as a tuple of thee elements as discussed in the "Module Specifics" section above. This module uses eval function of AST (Abstract Syntax Trees) provided by python for evaluating a string as an expression.

CONCEPTUAL MODELLING/ARCHITECTURE:

As discussed above, this implementation is modular where each module serves a specific functionality. Each module has several functions inside which performs a specific task. To better understand the conceptual modelling and flow of this implementation, **call trees** of **bm25** and **indexer** modules are depicted below.

Call trees represent calls from one function to another along the arrow. Further description of the function and its task is provided in the source code file. The module and function is represented in each circle.

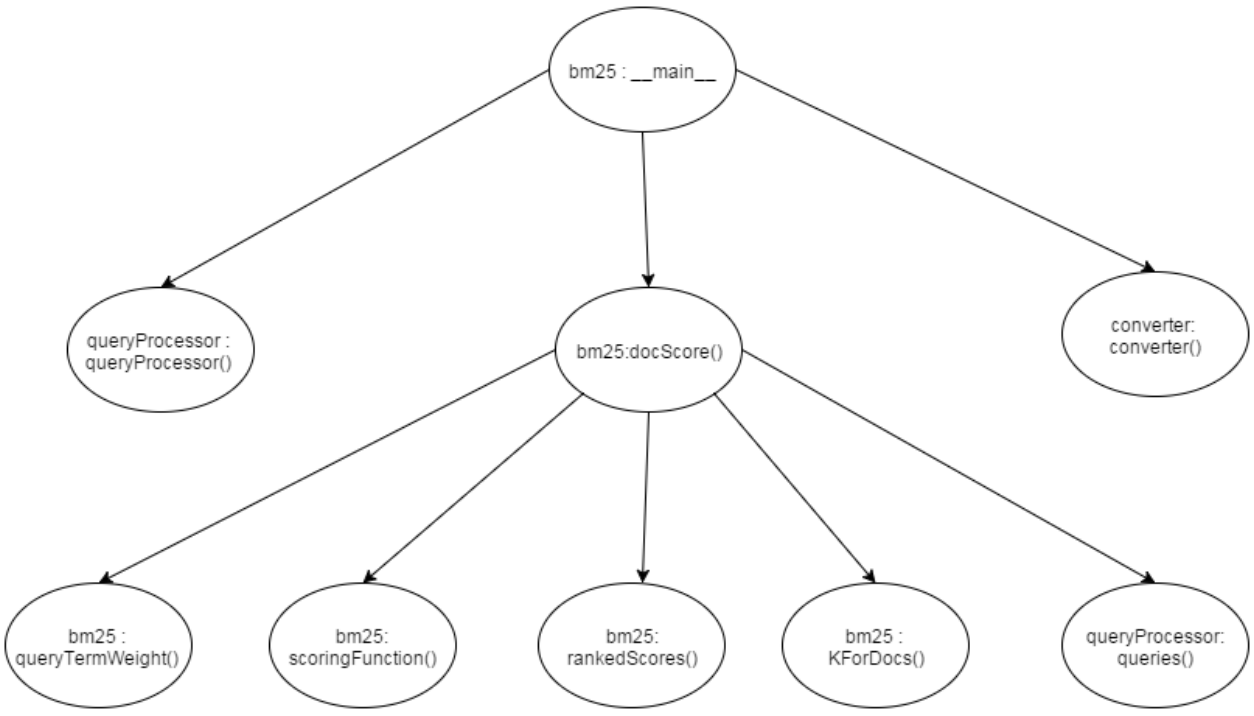

Fig 1: Call Tree of bm25 Module

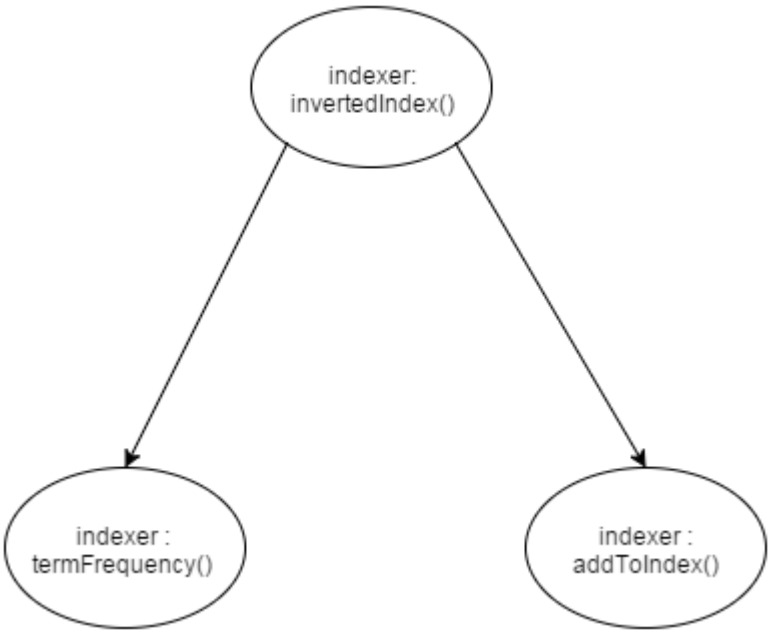

Fig 2: Call Tree of indexer module

RESULTS:

As given in the problem statement, the sorted documents by their bm25 score along with the Query_ID for the run,

bm25 index.out queries.txt 100 > results.eval

can be found in the **results.eval** file provided in the zip file.