546: ASSIGNMENT 2
Programming Assignment – Retrieval Models

Shravya Kadur
ID: 32167135

I. System Design

A. Description

The system is built using Python as the primary language. The three main classes in the system are:

1. **IndexerClass** – Designed to handle the indexing part of the system. It begins by parsing the collection, going through each document and extracting each word. The words are then stored in an inverted index, with each word having its own posting list. The mapping between docId and scene; docId and play is also maintained. Based on the user input, this class stores the created inverted index either in compressed or uncompressed form, with a lookup table on disk.

2. **CompressorClass** – This class is specifically responsible for all compression-related tasks. It takes the inverted index generated by the IndexerClass, uses delta encoding to transform the docIds and V-Byte compression to make the storage of the index on disk less expensive. It is also able to reverse-engineer this process to obtain the original inverted index from a compressed index file.

3. **QueryClass** – The query class handles all queries of the system. It generates a file which stores 7 random words in 100 iterations. This is then used to calculate the appropriate Dice pair (the word in the vocabulary with the highest Dice's Coefficient corresponding to each word) for each of the 7x100 words.

4. **PostingClass** – Forms the basis of the posting list for each term. It stores the list of term positions itself, along with the term frequency by document, and document frequency of the term.

5. **RetrievalClass** – The class contains functions that are fundamental for the assignment on retrieval models. It uses Term-At-A-Time to calculate scores using Vector Space and BM25, while using Document-At-A-Time for the Language models, Jelinek-Mercer Smoothing and Dirichlet Smoothing.

B. Design Tradeoffs

- The choice for the Inverted Index was between a clean, easy-to-read list of tuples for each word with (docId, position) pairs and a list of integers with docId, length of the position array, and the elements of the position array themselves. The latter was chosen for ease of storage and compression.
- The term and document frequencies were initially stored with the inverted list but were later moved to memory in order to enable ease of index compression and retrieval of the frequencies on-demand.
- The Term-At-A-Time algorithm was preferred over Document-At-A-Time for Vector Space and BM25 in order to avoid unnecessary iterations through documents that do not contain the query terms.

C. Resolved Issues

- BM25 did not return expected results initially since the function normalized document scores before ranking them. This was resolved by only normalizing for Vector Space scoring.
- The Vector Space and BM25 models did not return expected values for queries with repeated terms. This was resolved by considering each query term once only.

D. Software Libraries

1. **math** – To obtain mathematical log to the base e (natural logarithm).
2. **random** – To generate random indexes in order to populate 7 word queries.
3. **sys** – To read command line arguments and to pass byteorder.
4. **json** – To read the collection data.

II. Q & A

A. Do you expect the results for Q6 to be good or bad? Why?

Q6 contains the query terms “to be or not to be”. Such a query would likely produce results that are bad, and this is due to the terms that make up the query. The terms are all (or mostly) stop words, which have a high occurrence rate in any given document. This then implies that a longer document by default has a high count of these terms and would be scored higher. In addition to this, the terms “to” and “be” occur more than once so they add more weight to stop words.

B. Do you expect the results for a new query *setting the scene* to be good or bad? Why?

The query “setting the scene” has the potential to produce good results in a general scenario. The reason for this is that there is only one stop word, “the”, and the other query terms are not very commonly used. In this context however, the word

"scene" occurs in every document, hence not making it a good term to query along with the one stop word that exists.

C. What will have to change in your implementation to support phrase queries or other structured query operators?

The implementation currently only deals with a single word at a time (or unigrams). To support phrase queries, including a broader model with n-grams such as bigrams and trigrams (or even higher, if needed) would make the implementation more accurate for phrases and other structured query operators. Adding the relevance factor would improve the system significantly as well.

D. How does your system do? Which method appears to be better? On which queries? Justify your answer.

The system does well in identifying occurrences of individual terms in the query, but not the query as a whole. It would do better with relevance judgements and use of n-grams where n>1. The methods that do well in general are the query likelihood models (Dirichlet does the best). Specifically, the methods do well on queries with lesser terms such as "alas", since they consider individual terms only. They do not perform as well on longer queries, and on queries with stop words. This is likely because the query likelihood models rank by the probability that the query occurs in the document. They further use smoothing to account for missing words or words that haven't occurred yet.