Lucene Search Engine

1. Introduction

In the implementation of the Lucene Search Engine, I have used the latest version of Apache Lucene Jar which is 8.7.0. Java 1.8 is used in building the Jar for Lucene Search Engine. In this engine, cran.all.1400 the dataset is used for indexing and then queries from cran.qry are used on the indexed dataset. After running the engine, trec_eval is used to evaluate system performance.

2. Indexing

Before indexing the documents, the index writer is set on CREATE mode so that every time we run the engine, old index files will get overwritten by the new index file resulting in an accurate number of indexed documents. By looking at cran.all.1400 file, we can see there are a total of 5 fields present for each document. So, while creating a document, I used split to index each document separately and to store all the fields in a specific document. The fields for documents are as follows:

- Item(I) – Document number
- Title(T) – Title of Document
- Author(A) – Author of Document
- Words(W) – Content of Document
- Bibliography(B) – Bibliography of Document

3. Analyzer

I have created a custom analyzer class DocAnalyzer which extends Analyzer class. In that class, I have overridden the method createComponents to create a custom analyzer. In that method, I have created a TokenFilter to apply specific filtering on the tokens. I have used stopFilter to remove all stop words from the token. For this, I have used a set of English Stop Words. Then I have used PorterStemFilter to stem all the tokens. But, to use PorterStemFilter your all tokens must be in lowercase. Therefore, I have used LowerCaseFilter to lowercase all the tokens before using any filter.

4. Parsing and Searching

The queries from cran.qry are parsed using MultiFieldParser. I have used MultiFieldParser because there are a total of 4 fields on which query should be parsed to find a relevant document. There are a total of 225 queries present in cran.qry file. Before parsing, each query is trimmed, and all escape characters are ignored. I have used multiple similarities in the engine and then compared them to find the best possible similarity to be used. I have used BM25Similarity, BooleanSimilarity and ClassicSimilarity to score the documents.

scoreDocs is used to retrieve the score of searched queries over the document and then it is stored in the result file.

- BM25Similarity

  It is based on probabilistic retrieval and it scores documents based on the frequency of the query terms.
- ClassicSimilarity

  This similarity extends TFIDFSimilarity. It is based on the Vector Space Model scoring approach which considers the relative frequency of query terms to give an appropriate score to a document.
- BooleanSimilarity

  This similarity gives a score to query terms based on their appearance in the document. It gives a score of value 0 if the term does not appear in the document and it gives a score of value 1 if the term appears in the document.

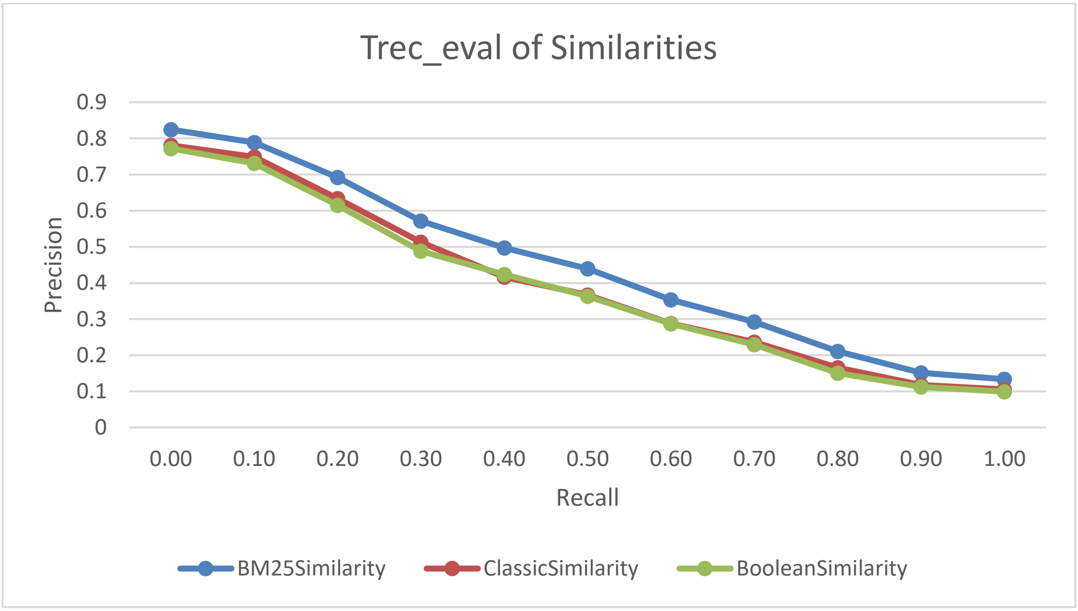
To use these similarities, I have created a property file in which we can define which similarity to be used.

5. Trec_eval Evaluation

Below are the map values of different similarities used:

Similarities	Map Value
BM25Similarity	0.4272
ClassicSimilarity	0.3736
BooleanSimilarity	0.3625

As you can see, BM25Similarity has the highest map value. Therefore, we can say, for this dataset and the logic used, BM25Similarity is the better similarity than remaining.



As you can see from the above graph, BM25Similarity has the higher score at every point. Therefore, we can confirm that BM25Similarity is a better choice.