Introduction

I used Java 1.8 and Apache Lucene 8.11.2 to index the Cranfield Collection[1], search the generated index and score my search engine using the queries provided in the Cranfield Collection using the scoring functions from the Vector Space Model, Best Matching 25, LMDirichlet and LMJelinekMercer. I used the StandardAnalyzer as well as creating my own CranfieldAnalyzer. Furthermore, I used a standard QueryParser as well as a MultiFieldQueryParser. My program begins from `Main.java` and a user can specify which operation they want to perform (indexing, searching or evaluating), the analyzer they would like, the scoring type to use and the parser they would like. These are done via command line arguments which are documented in `README.md`.

Parsing

In `Indexer.java`, I parse each line in the Cranfield Collection corpus `cran.all.1400` into a list of strings and iterate through this list. I also created a list of documents for our index. The text after ".A" represents the author, the text after ".B" represents bibliographic information and the text after ".W" represents an abstract. When a line starts with any of these, we add a `TextField` to the current document with the accumulated text for the previous field such as ".T" for the title. If any of these fields such as author are mentioned in the abstract, we keep them in the abstract field. The text after ".I" represents the identifier (starting from 1) which is the start of the document. Therefore, when a line starts ".I", we add a `TextField` to the current document with the text accumulated from the previous document's abstract (which is always before the ".I"), add this document to our list of documents and create a new document for the next field (this works as long as we are not on line 1, in which case there is nothing before it so we only create a new document). If the line does not start with any of these fields, we simply accumulate the text from the previous lines to use when we meet any of the identifiers mentioned previously.

Indexing

In `Indexer.java`, The `IndexWriter` uses the list of documents after parsing to write to the `./index` folder and also uses the analyzer requested by the user. The `StandardAnalyzer` in Lucene does processing such as tokenization, making the text all lowercase and removing stop words using the default `STOP_WORDS_SET`. My own `CranfieldAnalyzer` extends the Analyzer class in Lucene but instead uses a unique set of stop words[2], converts words to the possessive form using a `EnglishPossessiveFilter`, uses the porter stemming algorithm with a `PorterStemFilter` and performs more additional aggressive English stemming using a `SnowballFilter`. From my testing, other analyzers did not provide a significantly better performance boost.

Searching

In `Searcher.java`, one type of searching is for the command line using `searchForCommandLine()` which outputs the graded top 5 relevant documents along with their score. Another is for scoring purposes using `searchForScoring()` which writes the results of a query to `./evaluation/scores` to be used by trec_eval. Both types of searching use `IndexSearcher.Search()` to get an array of ScoreDoc. For searching, the standard QueryParser has built-in features such as wildcards but this doesn't benefit us for our

relatively simple searches in our corpus. However, the MultiFieldQueryParser allows weighing to be provided for each document field to boost the importance of each. From my testing, I found that a weighting of `0.3f` for title, `0.7f` for abstract and `0.05f` for department yields the highest results, with the author field having no weighting as I learned that it doesn't have an impact on the corpus itself.

Scoring

In `Scorer.java`, I use queries in `QRelsCorrectedforTRECeval`[3] with my `score()` function to write the results of scoring using `Searcher.java` into `./evaluation/scores`. The `BM25Similarity` (BM25) uses TF-IDF meaning it rewards text frequency. The `ClassicSimilarity` (Vector Space Model) calculates the cosine similarity between query vectors and document vectors to find relevance. The `LMDirichletSimilarity` smooths the text frequencies in a document using a Dirichlet distribution to handle rare text in a document. The `LMJelinekMercerSimilarity` uses linear interpolation to combine probabilities from the collection model and document model to calculate relevance.

Evaluation

After running `./run.sh`, the image `visualization.png` is created using the JFreeChart library[4]. We can observe that the standard analyzer always has a better set recall (≈ 0.9961) compared to the `CranfieldAnalyzer` (≈ 0.9629) in blue, which is caused by the `StandardAnalyzer` producing a higher number of results compared to the `CranfieldAnalyzer`. Therefore, a lower precision since precision is [relevant retrieved results/total retrieved results] but a higher recall since recall is [relevant retrieved results/total relevant results]. Also, we notice that the MAP values are higher for the `CranfieldAnalyzer` ($0.3396 \leq \text{MAP} \leq 0.4401$) compared to the `StandardAnalyzer` ($0.3065 \leq \text{MAP} \leq 0.4082$) in red. This is caused by the manual addition of stopwords, aggressive stemming with porter stemming algorithms and snowball filters. The best engine in this case is the `CranfieldAnalyzer` with `BM25` scoring and the `MultiFieldQueryParser`, which makes sense as we can specify weightings and reward text frequency for search results which suits the Cranfield Collection. The LMDirichletSimilarity and LMJelinekMercerSimilarity do the worst, which is most likely because they either assume a Dirichlet distribution for text frequency or do not capture dependencies between text (unsuitable for our corpus).

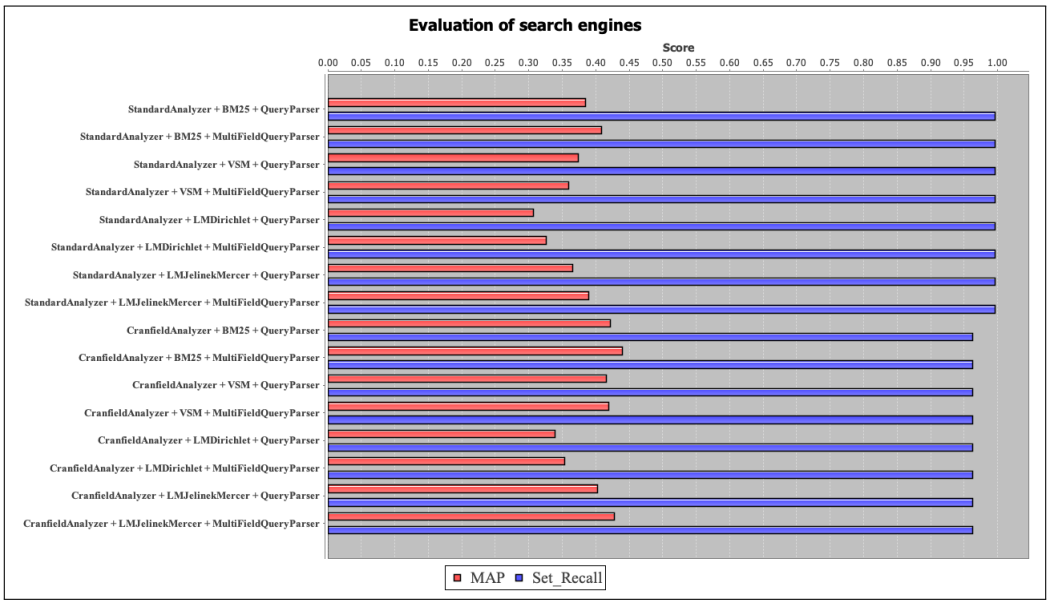

Figure 1: Visualization of the MAP (mean average precision) and set recall values for various combinations of analyzers, scoring methods and parsers.

References

[1] https://ir.dcs.gla.ac.uk/resources/test_collections/cran/
[2] https://www.ranks.nl/stopwords
[3] Google Drive folder with corrected QRELs file for trev_eval (accessed by tcd email only)
[4] https://www.jfree.org/jfreechart/