CS7IS1: Assignment 1 - Lucene Search Engine

JOSEPH FITZPATRICK, Trinity College, Dublin, Ireland

1 INTRODUCTION

This report will outline the process by which I created a Search Engine on the Cranfield Index using lucene.
The results were evaluated using the "Trec Eval" library to test for a wide variety of scores.

2 IMPLEMENTATION

2.1 Preprocessing Documents

The first step in creating the search engine was to process and index all of the documents required. For this I created a simple parser for the entire "Cranfield 1400" file. The parser would read the file in line by line. Each line would then be added to an variable depending on it's context, ie. if it was a Index, Title, Author, Bibliography or Content. Once the context looped around to the starter context, the variables would be used to create a new document.

Once all the documents were created, they were added to an 'IndexWriter' object and then written to a 'Directory' stored on Disk. The directory was then analysed using a custom analyser that I created. The analyser removed stop words, normalised characters to lower case and stemmed the words.

In each of the documents the Title, Author, Bibliography and Content were stored as an indexable field using 'Field.Store.YES', the "Index" of each document was stored as a non-indexable field by using "Field.Store.NO".

```
Document d = new Document()
d.add(new TextField("index", Field.Store.NO))
d.add(new TextField("title", Field.Store.YES))
d.add(new TextField("author", Field.Store.YES))
d.add(new TextField("bib", Field.Store.YES))
d.add(new TextField("content", Field.Store.YES))
```

2.2 Searching the Index

The next step was to begin searching the index. As the index contained a number of fields for each document which need to be searched I used the "MultiFieldQueryParser". This query parser allows you to search across all of the fields at once. It does however treat all of the fields as equal, giving each an equal weighting. I decided to change this and alter the weighting for each field.

Field	Weighting
Index	N/A
Title	0.35
Bibliography	0.01
Author	0.03
Content	0.61

Table 1. Weightings applied to each field in the Index

These score were calculated by performing some small calculations to identify the optimal weightings to give the best results. Rarely do people search for the author of a book or it's bibliography information and therefore, both of those received very low weightings. Giving the higher weightings for the content and title was based on the fact that people search for those more often.

The Search queries were parsed from the "Cran.qry" file. They were then trimmed and parsed using the custom analyser to remove stop words and stem the words. Once the queries were in the correct format for searching they were then fed into the IndexSearcher Object. This searched the index for the queries and returned the top 30 results identified.

2.3 Ranking Results

The results that are returned from the IndexSearcher object are ranked based on their relevancy score. This score indicates how relevant a Document is based on the query. The documents are ranked highest to lowest.

2.4 Custom Analyser

Once I had the initial minimal search engine working I began looking at ways to improve the results I obtained. To do this I used the "Luke" index visualisation tool to view the directory created. With this I could see that the Standard Analyser was lacking in a few areas. With some research on lucene I began to create the analyser by extending the 'StopwordAnalyzerBase' class. This class allowed me to customize the stop-words I wanted to exclude, while also allowing me to extend the analysers functionality. I decided to use the 'English Stop Words Set' as it contained the majority of the words 'Luke' was showing. I also added a lower-case normaliser to the analyser, as well as a stemming filter to stem words.

In the end the analyzer was made from components including LowerCaseFilter, StopFilter (with a custom set of stop-words) and a PorterStemFiler. Each of these components individually provided some marginal gains, however their combination helped to create a more accurate search engine.

3 RESULTS

Providing results for the effectiveness of the search engine was carried out using "Trec Eval". This tool works by performing some

Author's address: Joseph Fitzpatrick, Trinity College, Dublin, Ireland, jfitzpa1@tcd.ie.

calculations against your results and the optimal results. I also compared different scoring approaches in my model.

Scoring	MAP	Recall	Reciprocal Rank	P5
BM25	0.412	0.8320	0.832	0.4587
Boolean	0.3013	0.7257	0.7056	0.3556
Classic/VSM	0.3805	0.8192	0.8024	0.4213

Table 2. Results generated from each of the similarity scoring mechanisms

The Mean Average Precision across all similarity scoring mechanism's was lower than I had expected. With such a low MAP the ranking of the results would be less than ideal. However the Recall of the results was much better. With a recall of above 80% for BM25 the majority of relevant documents would be returned. The reciprocal rank shows how the ranking system was best in the BM25 Similairy Scoring. The P5 score shows that on average the precision for the top 5 results returned was less than ideal, as most people only click on the first few results presented by a search engine, I would have liked a higher score here.

From these results, it's clear that using BM25 similarity scoring produces the best results across the board, and boolean similarity produces the worst results. BM25 is an improved version of the Classic Similarity TF*IDF scoring mechanism usually used within lucene. It uses a probabilistic score for determining if a user will find a certain document relevant when scoring the documents. Boolean Similarity gives the worst results as it only scores a document if it fully matches the query. Therefore no partial results are returned leaving gaps in the information retrieved. The Vector Space Model (VSM) or Classic scoring performed well, however it wasn't as good as the more advanced similarity method which did more than just calculating the TF*IDF.

I was able to adjust the scoring mechanism by changing the Similarity passed into the IndexSearcher as an argument.

```
IndexSearcher is =
          new IndexSearcher(ireader);
is.setSimilarity(new ClassicSimilarity());
```

By changing the ‘ClassicSimilarity‘ object to ‘BM25Similarity‘ or ‘BooleanSimilarity‘, I was able to generate the new results. With each of the new similarities in the search mechanism, I also had to alter the similarity scoring used in creating the index. This could be done in by changing the ‘IndexWriterConfig‘ to use whichever similarity method required.

```
IndexWriterConfig iwConfig =
        new IndexWriterConfig(analyzer);
iwConfig.setSimilarity(new ClassicSimilarity());
iwConfig.setOpenMode(iwConfig.OpenMode.CREATE);
IndexWriter iwriter =
        new IndexWriter(directory, iwConfig);
```

4 CONCLUSION

From writing the Search Engine with the lucene library, I have been able to explore the information retrieval process in greater detail. Being able to compare the impact of different techniques, such as stemming and stop-word removal, showed their significance. As a whole, it take a combination of a lot of different factors to create a search engine capable of returning significant results.

While my implementation produced some good results, it did lack in a few areas. As I already had a predefined set of queries I was able to tailor the documents weighting to be skewed a bit towards those queries rather than general searches. Unfortunately we don't live in a perfect world and have access to a predefined set of queries, so this did limit the search engine as a general searching mechanism.