CS7IS1: Information Retreival Assignment 1 - Lucene Search Engine

KEVIN MORRIS, Trinity College, Dublin, Ireland

The following document will outline the process by which I created a Search Engine on the Cranfield Index using Lucene. The library trec_eval was used to evaluate the results for a wide range of scores.

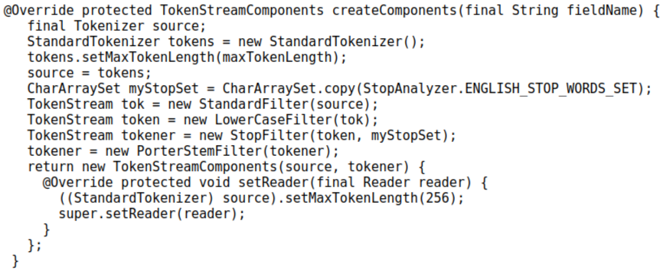
1 CREATING THE INDEX

The first task was to create a Lucene Index on the Cranfield Documents. The main() method parses the command-line parameters, then in preparation for instantiating IndexWriter, opens a Directory, and instantiates StandardAnalyzer and IndexWriterConfig. The Cranfiled File to be indexed is imported from the command line providing the location of the document. The value INDEX_DIRECTORY is the path of the filesystem directory where all index information should be stored.

Inorder to index and process all the docuemtns I created a parser for the Cranfield file .The Cranfield file was a list of documents where each document had a Title (I.), Author (A.), Bibliography (B.) and the Context (W.). The parser would read the file in line by line. Each line would be added to an variable depending on the field. Once the parser looped back to a Title context, a new document would be created.

Upon completion of all the documents being created, they were added to an "IndexWriter" object and also written to a Directory. This directory was analysed applying a a modified Analyzer which I created.

This analyser normalised characters to lower case, removed stop words and stemmed the words. Removing the stop words were words taken from the ENGLISH_STOP_WORDS_SET in the "StopAnalyser" libray. The "PorterStemFilter" library was used to stem the words when being indexed.

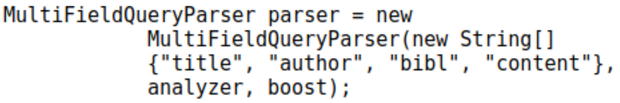


```
@Override protected TokenStreamComponents createComponents(final String fieldName) {
   final Tokenizer source;
   StandardTokenizer tokens = new StandardTokenizer();
   tokens.setMaxTokenLength(maxTokenLength);
   source = tokens;
   CharArraySet myStopSet = CharArraySet.copy(StopAnalyzer.ENGLISH_STOP_WORDS_SET);
   TokenStream tok = new StandardFilter(source);
   TokenStream token = new LowerCaseFilter(tok);
   TokenStream tokener = new StopFilter(token, myStopSet);
   tokener = new PorterStemFilter(tokener);
   return new TokenStreamComponents(source, tokener) {
     @Override protected void setReader(final Reader reader) {
       ((StandardTokenizer) source).setMaxTokenLength(256);
       super.setReader(reader);
     }
   };
 }
```

Fig. 1. Code Snippet of the anlayser used to stem and remove stop words

2 QUERYING THE INDEX

Once I created the Index I began to create a method to search the index. With the multiple fields the index contained for each document I used a "MultiFieldParser" which searched the document for each field. The only issue was that it treats each field equally when searching in the documents. I decided to alter weighting of importance for each field.



```
MultiFieldQueryParser parser = new
            MultiFieldQueryParser(new String[]
            {"title", "author", "bibl", "content"},
            analyzer, boost);
```

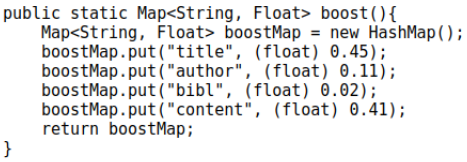
Fig. 2. Code Snippet of MultiFieldParser

In order to determine the weighting for each field, a number of tests were done to find the most accurate weightings to give the best results. From previous experience of searching for books, I would rarely use the Author or Bilbiography to search for a speicfic book so I gave them a lower weighting of importance. The title and context were the main fields so they had a much higher weighting.

Context	**Weighting**
Title	.34
Author	.01
Bibliography	.02
Content	.62

Table 1. Weight of each field

The queries were parsed from the "cran.qry" file. They were then, just like the index file were then stemmed and removed stop words using the customised Analyser. Once all the queries were in the same format as the index file they were then put into the "IndexSearcher" which searched the index for each query and returned all the matching documents. The max amount of results to be returned was set to 30.



```
public static Map<String, Float> boost(){
    Map<String, Float> boostMap = new HashMap();
    boostMap.put("title", (float) 0.45);
    boostMap.put("author", (float) 0.11);
    boostMap.put("bibl", (float) 0.02);
    boostMap.put("content", (float) 0.41);
    return boostMap;
}
```

Fig. 3. Code Snippet of weighting the fields

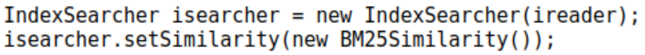
Author's address: Kevin Morris, Trinity College, Dublin, Ireland, morrisk3@tcd.ie.

3 SCORING

The "IndexWriterConfig" has a method called "setSimilarity" which sets the type of Lucene scoring that you would like to use. Similarity defines the components of Lucene scoring. From research the main four lucene scoring methods are:

- BM25
- Boolean
- Classic
- lm_dirichlet

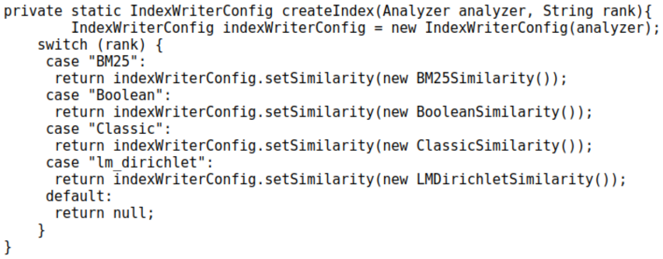
One of the higher-level implementation such as TFIDFSimilarity (also known as Classic Similarity), which implements the vector space model with this API is one of the four I decided to use.



```
IndexSearcher isearcher = new IndexSearcher(ireader);
isearcher.setSimilarity(new BM25Similarity());
```

Fig. 4. Code Snippet of setting the Similarity when Searching

When testing it was necessary to make sure that the indexing and query searching were using the same similarity scoring as they had to be the same in order to get the right results and scores.



```
private static IndexWriterConfig createIndex(Analyzer analyzer, String rank){
        IndexWriterConfig indexWriterConfig = new IndexWriterConfig(analyzer);
    switch (rank) {
     case "BM25":
      return indexWriterConfig.setSimilarity(new BM25Similarity());
     case "Boolean":
      return indexWriterConfig.setSimilarity(new BooleanSimilarity());
     case "Classic":
      return indexWriterConfig.setSimilarity(new ClassicSimilarity());
     case "lm_dirichlet":
      return indexWriterConfig.setSimilarity(new LMDirichletSimilarity());
     default:
      return null;
    }
}
```

Fig. 5. Code Snippet of diffferent Scoring Similarities when Indexing

4 RESULTS

Using trec_eval I was able to get the results of each of the scoring methods. trec_eval evaluates your results against a sample optimal results.

Similarities	MAP	P30	Recip. Rank	Recall
BM25	0.4067	0.1533	0.8262	0.8383
Classic	0.3790	0.1481	0.7991	0.8148
Boolean	0.3006	0.1261	0.7072	0.7260
lm_dirichlet	0.3096	0.1252	0.0.6958	0.7104

Table 2. Results of each similarity lucence scoring

The BM25 similarity is the lucene scoring method that gives the best results and the boolean similarity is giving the worst results when evaluating against the optimal results. The boolean similarity gave the worst results as it only scores a document if it matches the query fully. This leaves gaps in the infroamtion retreived. BM25 is a improved version of Classic similarity as it uses a score for determining it will find a certain document relevant when scoring the documents.

5 CONCLUSION

From creating a Search Engine in Lucene, I was able to gain a valuable insight into the Information Retrieval process in a larger scale. It allowed me to see the significant change in results when improving the analyser to allow for stop word removal and stemming the words. Testing the different scoring methods was a great way to explore the Lucene libraries and methods.