Information Retrieval & Web Search - Group 10

Sébastien Dunne Fulmer
dunnefus@tcd.ie
19333374

Eamon Phelan
phelanec@tcd.ie
02406390

Aadesh Milind Rasal
rasala@tcd.ie
22301280

Prathamesh Sai Sankar
saisankp@tcd.ie
19314123

Yifan Zhu
zhuyi@tcd.ie
18300717

Abstract

Creating an efficient search engine on a large corpus is a difficult task. However, through the use of advanced information retrieval techniques, one can increase the mean average precision (MAP) values of a search engine considerably. In this research paper, we discuss the techniques we implemented in Apache Lucene to index a large collection (2GB in size) and search within the index efficiently to develop an effective search engine. In terms of indexing, we used techniques such as stop-word removal and synonym filters using custom analyzers with stemming. For scoring, we used similarities such as BM25, Classic, LMDirichlet and LMJelinekMercer. For searching, we used BoostQuery objects to apply weights to different topics, with narratives not given any weighting if they contained "not relevant". By implementing these advanced techniques, our highest MAP score was 0.3366 using a CustomAnalyzer, LMJelinekMercer Similarity and QueryParser.

1 Introduction

For creating an efficient search engine, we approached this project with a focus on simplicity while maintaining accuracy. We started the project by trying out as many different information retrieval techniques as possible that we thought would be suitable for our large corpus. After creating resulting scoring files using different techniques, we evaluated them using trec_eval to get MAP values. By testing different information retrieval techniques, we were able to keep the techniques that had a cumulatively positive effect on the MAP scores from our evaluation and discard the ones that didn't. After synthesising a general approach from experimentation, we developed our implementation by extending the default standard functionality of Apache Lucene. Finally, we tweaked our implementation to optimise remaining areas such as weightings and synonyms.

2 Approach taken

Over the course of our time working on this project, we changed our approach continuously to adapt to new findings and research papers we read. The approaches we took for each task while creating our search engines are explained in depth below.

2.1 Parsing

For parsing our queries, our goal was to make it as simple as possible so we could focus on advanced techniques instead. Our queries were complicated since they consisted of topics, so it was essential that we focused on applying information retrieval techniques instead of manually parsing each element. Therefore, we decided to use the Jsoup library[1] for parsing the documents and getting elements by tags. By doing so, we were able to easily access element tags in the topics from our queries instead of using complicated regex.

2.2 Indexing

Similar to parsing, we wanted to simplify the process of getting element tags from the corpus when we indexed our collection. Our collection was complicated with 4 sub-types of data from The Financial Times Limited, The Federal Register, The Foreign Broadcast Information Service and The Los Angeles Times. We also decided to use the Jsoup library for getting elements by tags in our corpus which abstracted the complexity of messy regex statements that could lead to bugs.

2.3 Searching

For our Searcher class that generates queries from topics, we used a BoostQuery object to apply weightings to the title, description and narrative tags in the topic. We noticed in the topics that the narrative sometimes contained "not relevant", therefore we decided to only apply a weighting to a narrative if it did not contain this. Using an index searcher from Lucene, we produced the top 1000 documents for a topic.

2.4 Scoring

For scoring, topics are parsed by <top> elements using Jsoup and resulting scoring files are put in the “evaluation” folder. We can iterate through the topics and search the topic using our Searcher class. The searcher for our index requires a Similarity to produce scores. The Similarity scoring implemented in this project includes:

1. BM25 Similarity
2. Classic Similarity (Vector space model)
3. LMDirichlet Similarity
4. LMJelinekMercer Similarity

3 Implementation

After deciding on the general approach we would take, we spent the majority of our time working on the implementation of our search engine. For each step of implementation, we encountered issues and overcame them which are explained in depth below.

3.1 Parsing

Our parsing logic is primarily implemented in our ParserUtility class, which is used by our Indexer class when indexing the corpus to parse the formats used by each collection of source documents. Although the corpus provides SGML definition files for formats of each collection, SGML is not a common format and there are no widely available SGML parsers available for Java. The format can diverge sufficiently from XML for an XML parser to not be compatible. Instead of writing our own SGML parser, we checked to ensure that the documents were also valid XML and used Jsoup's XML parser. Using the information provided in the SGML definition files and accompanying documentation, we wrote a method for each document collection to parse its structure into a Lucene Document which could then be passed back to our Indexer. We attempted to map each collection's fields to a standard set of field names in order to make query construction more straightforward.

3.2 Indexing

Our indexing logic is primarily implemented in our Indexer class. Since our code allows the user to specify the Analyzer to be used as a command line parameter, the index method takes an Analyzer object as a parameter in order to delegate the Analyzer selection logic up to the Main class. When invoked, the Indexer instantiates a new Lucene IndexWriter with the provided Analyzer, pointed at the './index/analyzerName' directory where analyzerName is the name of the Analyzer that was used for the indexing. We used different directory paths for each Analyzer so we don't have to re-index the collection (i.e. delete and populate the same folder) every time. The Indexer then iterates through each of the four collections and finds all of the documents contained within the collection by walking the paths in each collection's directory. It then leverages the previously described parsing logic to convert the raw document content for each document in a collection into structured Lucene documents, which are then added to the IndexWriter.

3.3 Searching

Our searching logic is primarily implemented in our Searcher class. Similar to the Indexer, the Searcher is designed so that an Analyzer can be passed into it for more configuration. However, it also takes a Similarity and QueryParser object since these are also relevant to the evaluation. The Searcher instantiates an IndexSearcher using the provided Similarity and stores the provided QueryParser for use in individual searches. The Searcher can then either start an interactive command line session which reads queries in from the standard input and returns a small number of results (which is useful for manual testing) or generate results based on a topic which is used later in the scoring process. When searching based on a topic, the Searcher uses different fields from the topic with different weights to construct a query and retrieves a larger number of results which is necessary for evaluation. After manual experimentation, we applied a weighting of 0.9f for the title, 0.5f for the description, and 0.3f for the narrative (if it didn't contain "not relevant"). Finally, the Searcher returns an ArrayList of these results in a suitable format for trec_eval which is used for evaluation.

3.4 Scoring

Our scoring logic is primarily implemented in our Scorer class. Similar to the Searcher, the Scorer is designed so that an Analyzer, Similarity and QueryParser can be passed into it for configuration. The Scorer parses the topics from the provided file using Jsoup's XML parser into instances of our Topic class. The Topic class takes the provided XML element representing the topic and parses the number, title, description and narrative fields for use while performing basic cleaning on the fields by removing characters not allowed by the QueryParsers and removing extraneous labels. The Scorer uses the Searcher for each of the parsed Topics in order to generate a full set of results for evaluation and writes them into the correct format for evaluation, with a filename based on the provided Analyzer, Similarity and QueryParser in order to allow for evaluating multiple potential configurations at once. We then have a separate shell script for evaluating these files using trec_eval.

4 Advanced techniques

Our search engine is implemented with advanced information retrieval techniques in mind that go beyond the default standard Lucene install. The advanced techniques we used and how they are implemented are explained in depth below.

4.1 Stop-words

Stop-words include the most commonly used words in any language. Since we are dealing with news articles in the English language, we use a set of common English stop-words[2] with a StopFilter which removes these common words for the search engine to focus on more important content and retrieve more relevant documents. For example, if a user tries to search "how to develop an information retrieval system", our list of stop-words in "./collection/stopwords.txt" contains "an", "how" and "to" which can be removed from our search, leading to a more effective search with a focus on the main target of the query (i.e. "develop", "information", "retrieval", "system").

4.2 Synonyms

We added synonym expansion to our custom Analyzer by adding a SynonymFilter to the tokenstream. The WordNet database[3] was used as a source for synonyms. Initially, we used this Analyzer for both index creation and querying but found it to unexpectedly cause a substantial drop in MAP scores (≈ -0.03). Further experimentation resulted in an implementation that only expanded synonyms at query time, and only on the shorter query elements. Attempting to expand synonyms on the longer "narrative" element was found to drop performance (≈ -0.1). A summary of the impact of synonyms on the maximum MAP score can be seen in figure 1.

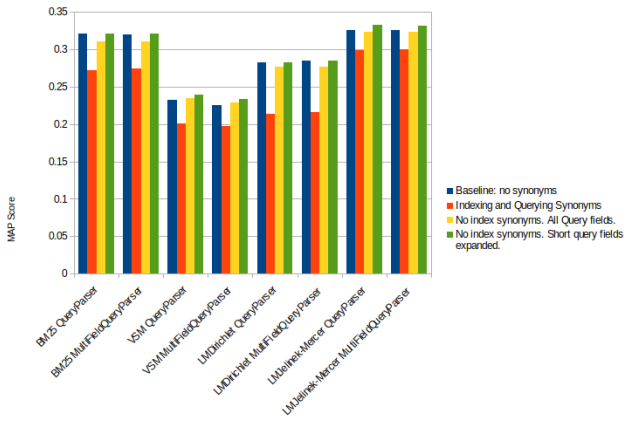

Figure 1. Impact of using synonyms on our search engines.

4.3 Weightings

We experimented with various weightings in two different places in our project:

1. Searcher class
2. Main class

In the Searcher, we assigned different weights to each attribute of the Topic (title, description and narrative) and then generated the query. In the Main class, we set weights to three common fields (title, content and TEXT) between four document types (FT, FR94, FBIS and LATIMES) when creating the MultiFieldQueryParser. For experimenting with these weights, we iterated with different weights in a loop to choose the hyper-parameters that gave the highest MAP value. In particular, we inspected the trends of the MAP values when we increase or decrease the weight for a particular field in order to look for the weighting that gave us the highest MAP value. For each weighting, we compared all the combinations of the Analyzer, the Similarity and the QueryParser to check which combination gave us the highest MAP value. By inspecting trends instead of trying out every weight numerically possible, we found the weighting and combination that generated the highest MAP value in a reasonable time without the need for an inefficient "brute-force" approach.

Table 1. Key for experiments

Abbreviation	Part of search engine
CA	CustomAnalyzer
LMJM	LMJelinek-Mercer
QP	QueryParser
MFQP	MultiFieldQueryParser

MAP	combination	title	description	narrative
0.3257	CA LMJM QP	1f	0.3f	0.4f
0.3282	CA LMJM QP	1f	0.4f	0.4f
0.3286	CA LMJM QP	1f	0.5f	0.4f
0.3283	CA LMJM QP	1f	0.6f	0.4f
0.3248	CA LMJM QP	1f	0.5f	0.5f
0.3289	CA LMJM MFQP	1f	0.5f	0.3f
0.3257	CA LMJM MFQP	1f	0.5f	0.2f
0.3293	CA LMJM QP	0.9f	0.5f	0.3f
0.3277	CA LMJM MFQP	0.8f	0.5f	0.3f

Table 2. Weighting for the fields in the topic

QP MAP	MFQP MAP	combination	title	content	TEXT
0.3293	0.3260	CA LMJM	0.1f	1f	0.1f
0.3293	0.3257	CA LMJM	0.1f	0.9f	0.1f
0.3293	0.3262	CA LMJM	0.1f	1f	0.2f
0.3293	0.3262	CA LMJM	0.1f	1f	0.3f
0.3293	0.3267	CA LMJM	0.1f	1f	0.4f
0.3293	0.3266	CA LMJM	0.1f	1f	0.5f

Table 3. Weighting for the fields in the documents

QP MAP	MFQP MAP	combination	title	content	TEXT
0.3329	0.3297	CA LMJM	0.1f	1f	0.5f
0.3329	0.3300	CA LMJM	0.1f	1f	0.6f
0.3329	0.3297	CA LMJM	0.1f	1f	0.7f
0.3329	0.3257	CA LMJM	0.2f	1f	0.6f

Table 4. Weighting for the fields in the documents (with synonyms)

4.4 Custom Analyzer

Our custom analyzer extends the original analyzer class and overrides the createComponents() method. Our implementation starts with breaking input text into tokens using the StandardTokenizer. Furthermore, we constructed a chain of filters which performs classic tokenization, removes English possessives (i.e. ‘s) from words, converts unicode text to ASCII by removing diacritical marks, applies synonym expansion, converts all tokens into the lower case to maintain consistency, removes trailing and leading whitespaces, removes stop-words from the generated token stream and finally applies the porter stemming algorithm to reduce words

to their root form. Stop-words we used here are not from the default set provided by Apache Lucene, instead we used a list of common English words which was mentioned in more detail in section 4.1.

5 Limitations

Our main source of concern regarding limitations of our implementation was over-fitting to both the topics and corpus. We knew that we could reach MAP values that are higher but it would risk over-fitting our search engine to particular topics. This is because the choices made in our CustomAnalyzer may not apply to a broader range of topics i.e. stop-words may be over-optimised to the chosen topics and synonyms may have an adverse impact. In particular, one potential impact of synonyms is that of changing language usage. The most recent release of the WordNet database is from 2005[3]. If the implementation were to be used for interactive querying for instance, the shift in language usage may make some synonyms redundant and harmful to effectiveness. The Lucene Similarity implementation we found to be most effective was the Jelinek-Mercer method, which we read about in a paper by Chengxiang et al. [4]. We customised this by experimentally determining a nonstandard lambda parameter of 0.65 optimised performance. Larger values of lambda are more effective for longer queries and our implementation would be hampered by a change in query length. Additionally, when considering the corpus, our customisation of weights for the MultiFieldQueryParser may not be effective in a real-world scenario where documents are being added to a corpus over time.

6 Conclusion

In this research project using Apache Lucene, we created a search engine that can index a large collection and search within the index efficiently. We tried 4 different similarities for scoring, 2 types of parsers, 3 types of analyzers and implemented advanced information retrieval techniques such as stop-word removal with a StopFilter as well as other filters including an EnglishPossessiveFilter, ASCIIFoldingFilter, TrimFilter, SynonymFilter and a PorterStemFilter. We experimented using different weightings for BoostQuery objects as well as the MultiFieldQueryParser. By implementing these and reading information retrieval papers related to our project, we achieved an impressive MAP value of 0.3366 by using the following combination:

- CustomAnalyzer (ClassicFilter, EnglishPossessiveFilter, ASCIIFoldingFilter, TrimFilter, StopFilter, SynonymFilter and PorterStemFilter)
- LMJelinek-Mercer (with lamda of 0.65f)
- QueryParser (default)
- BoostQuery for topics such as title, description and narrative with weightings 0.9f, 0.5f and 0.3f respectively.
- Stop-words from countwordsfree.com[2]
- Synonyms from WordNet[3]

```
-- Evaluating the StandardAnalyzer --
Search Engine: StandardAnalyzer BM25 QueryParser
map                     all     0.2642
Search Engine: StandardAnalyzer BM25 MultiFieldQueryParser
map                     all     0.2639
Search Engine: StandardAnalyzer VSM QueryParser
map                     all     0.1951
Search Engine: StandardAnalyzer VSM MultiFieldQueryParser
map                     all     0.1862
Search Engine: StandardAnalyzer LMDirichlet QueryParser
map                     all     0.2531
Search Engine: StandardAnalyzer LMDirichlet MultiFieldQueryParser
map                     all     0.2508
Search Engine: StandardAnalyzer LMJelinek-Mercer QueryParser
map                     all     0.2723
Search Engine: StandardAnalyzer LMJelinek-Mercer MultiFieldQueryParser
map                     all     0.2739
-- Evaluating the EnglishAnalyzer --
Search Engine: EnglishAnalyzer BM25 QueryParser
map                     all     0.3207
Search Engine: EnglishAnalyzer BM25 MultiFieldQueryParser
map                     all     0.3209
Search Engine: EnglishAnalyzer VSM QueryParser
map                     all     0.2342
Search Engine: EnglishAnalyzer VSM MultiFieldQueryParser
map                     all     0.2175
Search Engine: EnglishAnalyzer LMDirichlet QueryParser
map                     all     0.2956
Search Engine: EnglishAnalyzer LMDirichlet MultiFieldQueryParser
map                     all     0.2934
Search Engine: EnglishAnalyzer LMJelinek-Mercer QueryParser
map                     all     0.3272
Search Engine: EnglishAnalyzer LMJelinek-Mercer MultiFieldQueryParser
map                     all     0.3257
-- Evaluating the CustomAnalyzer --
Search Engine: CustomAnalyzer BM25 QueryParser
map                     all     0.3283
Search Engine: CustomAnalyzer BM25 MultiFieldQueryParser
map                     all     0.3281
Search Engine: CustomAnalyzer VSM QueryParser
map                     all     0.2454
Search Engine: CustomAnalyzer VSM MultiFieldQueryParser
map                     all     0.2272
Search Engine: CustomAnalyzer LMDirichlet QueryParser
map                     all     0.2954
Search Engine: CustomAnalyzer LMDirichlet MultiFieldQueryParser
map                     all     0.2938
Search Engine: CustomAnalyzer LMJelinek-Mercer QueryParser
map                     all     0.3366
Search Engine: CustomAnalyzer LMJelinek-Mercer MultiFieldQueryParser
map                     all     0.3333
```

Figure 2. Final results with a max MAP of 0.3366

References

[1] Jsoup: Java HTML/XML Parser
[2] List of stop-words from countwordsfree.com
[3] Synonyms from WordNet. Princeton University. 2005
[4] A study of smoothing methods for language models applied to Ad Hoc information retrieval. Chengxiang Zhai and John Lafferty. 2001