CS7IS3: Report for Assignment 2

Shravani Kulkarni
kulkarsh@tcd.ie
Trinity College Dublin

Kavithvajen Kamaraj
kamarajk@tcd.ie
Trinity College Dublin

George Chavady
chavadyg@tcd.ie
Trinity College Dublin

Anusha Lihala
lihalaa@tcd.ie
Trinity College Dublin

ABSTRACT

The paper explores a number of document search and retrieval approaches implemented using the open source library Apache Lucene on a collection of news articles. Many approaches were tried but only few resulted in scores above a standard Lucene implementation. The highest scoring approach achieved a MAP score of 0.3477 and involved implementing custom Analyzers and QueryParsers, and boosting the relevant terms in the query topics. Surprisingly, use of the NGramTokenFilter and a custom AbbreviationsFilter in the custom analyzer had detrimental and non-existent effect respectively on the MAP score.

KEYWORDS

Information Retrieval, Lucene, Indexer, Query Parser, Analyzer

1 INTRODUCTION

Apache Lucene is an open-source java library that provides indexing and query searching features. In our project, we have used Lucene core's 8.4.1 version [4]. We structured the project with different classes for each of the tasks of the search engine. We implemented four indexers for each of the four categories of the documents, a custom analyser, a query parser and the main class for calling functions from all of these classes. We have utilized trec_eval for evaluating our search engine.

The remainder of this paper is organized as follows: Section 2 covers the dataset in detail, section 3 explains the implementation focussing on the indexer, query parser and analyser used and section 4 illustrates the results obtained.

2 DATASET

The dataset is an "aggregated collection of news articles from the Financial Times Limited (1991, 1992, 1993, 1994), the Federal Register (1994), the Foreign Broadcast Information Service (1996) and the Los Angeles Times (1989, 1990)." The original data was reformatted by NIST to preserve as much of the original structure as possible, but also provide enough consistency to allow simple decoding of the data. The format used involves labeled bracketing, expressed in the style of SGML (Standard Generalized Markup Language). Some of the most common fields across the different document sources include;
`DOC, DOCNO, PROFILE, DATE, HEADLINE,`
`TEXT, PUB and PAGE`

Each document is bracketed by <DOC> tags. The <DOCNO> tags are used to identify the documents via a unique document number.

The error-checking performed on the dataset by NIST included automated checks for control characters, special symbols, foreign language characters, for correct matching of the begin and end tags, and for complete DOC and DOCNO fields. The errors that were not corrected and remain in the dataset include fragment sentences, strange formatting around tables or other "non-textual" items, misspellings, missing fields (that are generally missing from the data), etc.

3 IMPLEMENTATION

Indexer

The documents were categorized into four types with each having a different structure. The directory structures and the tag names in the documents were different across all the documents types. Hence, we implemented four different indexers to cater to the different parsing in each of the document types. After careful examination, we decided to extract three fields to add into the indexer: document number, headline and text. For some of the documents we also took into account the other tags that carried important meanings. For instance, "SUBJECT" and "GRAPHIC" tag in LA Times were appended at the end of the text while indexing. Similar was followed for some other important tags in the documents.

We utilized the `Jsoup` library to read the documents and segregate the contents in it [3]. `Jsoup` made the parsing of XML structure in the document very easy. The `getElementsByTag()` method by `Jsoup` extracted the tag, which we passed to our functions to replace newline characters and remove the opening and closing tags.

Query Parser

Query parsing stands as an interface between the users and the document they seek. So, its role in information retrieval and web search is critical, wonderful and often acutely frustrating. A search engine cannot reach its highest level of performance without a successful and intelligent query parser. Each topic in the query was structured with the following contents:

- `num`: The query number.
- `title`: The title of the query.
- `desc`: It provides more information about the query.
- `narr`: The narrative specifies what documents are relevant and what are irrelevant.

The query was parsed to store each of its parts into a separate variable. Several combinations of title, description and narrative were parsed and the query string that returned the best results was used. Some of the notable techniques used to improve the relevance score were:

- Using a list of stop words which included words curated after careful analysis of the content in the query. E.g., "would", "discuss", "mention", "documents", "describe"
  Also, replacing and removing certain words such as “Narrative” and "Description" also gave better results.
- Cleaning the query by removing special characters such as ‘?’, ‘/’, ‘-’, ‘(‘ and ‘)’ as these have special meanings in the query strings.
- Implemented a `MultiFieldQueryParser` that used the score boosting technique.
  E.g.,
  Boosting the headline to 0.05
  Boosting the text to 0.95
- Using a combination of title, description and narrative for the query string. We also boosted these to get better results.
  E.g., title ˆ 4, description ˆ 3 and narrative ˆ 1
- Parsing the narrative such that sentences containing phrases like “is relevant” and “are relevant” are appended to the query string and the sentences containing the “not relevant” and “irrelevant” keywords are appended after "-" or the prohibit operator. Certain “not relevant” and “irrelevant” keywords were followed by “unless”. The content after the unless was also appended in the relevant query string.

Analyzer

Instead of using the `StandardAnalyzer` provided by Lucene, we implemented our Analyzer called `'MyAnalyzer'` by subclassing Lucene’s Analyzer class. However, we used the standard implementations `StandardTokenizer` and `ClassicFilter` before performing custom processing on the generated token stream. The filters applied after `ClassicFilter` were `TrimFilter`, `LowerCaseFilter`, `StopFilter` and `PorterStemFilter` and `EnglishPossessiveFilter` in that order. The `LowerCaseFilter` changes the whole stream into the lower case for further analysis and the `TrimFilter` removes all the heading and trailing white spaces. The set of stopwords used with `StopFilter` can be found in the Appendix. Along with the standard list of stop word, we also added a few stopwords that were specific to the dataset provided (E.g. "relevant", "mention" etc.). The `PorterStemFilter` is used for converting words with common stems as they tend to have similar meanings. The `EnglishPossessiveFilter` is used for removing trailing ’s from the words.

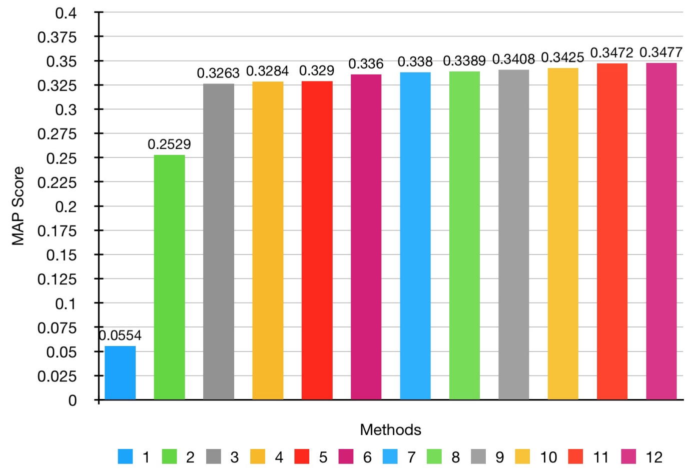

Figure 1: Changes in methods vs MAP

4 RESULTS

The ‘trec_eval’ tool was used as an evaluation tool to compare the results obtained by our search engine with the ground truth. The main metric we used was the Mean Average Precision (MAP). Table 1 and Figure 1 indicate how some changes to our methods resulted in massive jumps in the MAP score.

Changes in the anlyzer

At first, using simple indexing, querying and a StandardAnalyzer the search engine only had a MAP score of 0.0554. Switching to the English Analyser and using the BM25 Similarity Index yielded better result as the score then bumped up to 0.3263. English Analyser could only get us that far and to make more improvements we switched back to the Custom Analyser and made some significant changes to the query parser. By taking inspiration from [5] we also tried the approach of not adding stop words, adding stemming and adding feedback to our engine. This yielded better results in their case and it provided a MAP score of 0.3340 for our search engine. Adding English Possessive filter in the analyzer bumped up the MAP value to 0.338. Additionally, using multi similarity of filterLMJelinekMercer Similarity(0.6f) and BM25Similarity further helped with a higher score of 0.3472.

Boosting the multiple fields in the query parser

After switching the analyser for the Custom Analyser, boosting the scores of the headlines in every document to 0.1 and the text to 0.9, and considering the DOCTITLE in the FR94 collection as the headline allowed for a score of 0.2529. Jimmy et al. concluded that exploratory search queries turn out to be vague and thus boosting the title does not result in the correct outcome [2]. For these kind of queries, there is very less probability that the queries exactly match document titles. However, in our case the the queries provided detailed explanation of the kinds of document required and using boosting for title proved useful.

Analyzing the "narrative" in the given queries

Although the title and the description were used from the query, we also carefully analyzed the narrative to get better results. Also, increasing the weight of certain parts of the query by duplicating data points we felt was more important, that is, while computing the query, the title was replicated thrice, and the description was replicated twice while leaving the narrative as it is. This method enabled us to reach a score of 0.3284. We used the query parser to look for the words like “relevant”, “irrelevant”, etc. in the narrative of the query and used the relevant parts to form our queries which resulted in an increased score of 0.3290. We then found some corner cases that were not covered in the query parser to deal with the relevancy/irrelevancy in narratives, fixing them and removing the stop words “relevant” and “document” gave a new high score of 0.3380.

Considering other related content from the documents in the dataset

Initially we were only taking the headline and the text from the documents for indexing. We tried parsing the SGML comments and surprisingly not ignoring them yields a better score which was not something we anticipated. Coupled with boosting the headlines to 0.05 and text to 0.95 greatly, this new combination gave a score of 0.3360. However, since we were not exactly sure why parsing the SGML comments gave a good score, we tried the latest setup by ignoring the SGML comments and the score dropped to 0.3310. Hence, we decided to submit two search engines for this assignment to see which one performs better with the second half of the QRels file. Furthermore, we also tried to add other important tags from the documents like "GRAPHIC", "SUBJECT" and "FOOTNOTE" that eventually lead to the score of 0.3477.

Other approaches

A lot of experimentation was done with Ngrams, fuzzy word searches, boosting different parts and also trying different analysers, tokenizers, filters and similarity indexes [1]. None of them had a significantly positive effect on the scores as you can see in Table 2.

Major takeaways

- A properly tweaked custom analyser is always the best way to deal with a known dataset.
- Careful analysis of the structure of dataset gives proper insights for designing the search engine. For instance removing the stopwords specific to the data available (E.g. "mention", "relevant", "discuss").
- Making right use of the operators available in the lucene library helps. For instance, boosting the right terms and addition of prohibit operator for non relevant keywords can improve the accuracy.
- The IR system we built is not generic enough to handle any kind of data thrown at it. It’s made only to work with the given datasets.

S.No	Methods	MAP Score
1	Standard Analyser without query parsing for relevancy/irrelevancy	0.0554
2	Standard Analyser + Boost Score(Headline - 0.1; Text - 0.9) + Considered DOCTITLE as headline in FR94	0.2529
3	English Analyser + BM25Similarity	0.3263
4	Custom Analyser + Increased weight of terms by data replication (3T,2D,1N)	0.3284
5	QueryParser - looked for relevant and irrelevant parts	0.3290
6	Parsed the SGML comments (<!– –>) + Boost score(Headline - 0.05; Text 0.95)	0.3360
7	Fixed keywords to consider the case "a relevant" appears in description + Removed the words "relevant" and "document"	0.3380
8	Added English Possessive Filter	0.3389
9	Parsed the GRAPHIC tag in LATimes	0.3408
10	Used the LMJelinekMercer Similarity(0.6f)	0.3425
11	Used LMJelinekMercer Similarity(0.6f) & BM25 Similarity	0.3472
12	Parsed the SUBJECT tag in LATimes	0.3477

Table 1: Changes that improved the score massively

Methods	MAP Score
Fuzzy word searches	0.1792
Abbreviations token filter	0.1878
Boost Score (Headline - 0.4; Text - 0.6)	0.2799
Edge-NGram Token Filter (min - 2; max - 3)	0.0301
Edge-NGram Token Filter (min - 2; max - 5)	0.0294
Edge-NGram Tokenizer (min - 2; max - 5)	0.0046
NGram Token Filter (max - 2; max - 3)	0.0005
NGram Tokenizer	Memory issue
Fixed Shingle Filter	0.1243
KStem Filter	0.3246

Table 2: Changes that did not yield good results

5 CONCLUSION

Building efficient search engines is a complex task that entails indexing the documents and searching the index using query strings. Efficiency is brought in by properly parsing both the content that is to be indexed as well as the query string that is to be searched. Proper analysis of the content within the documents and the queries can help us parse them efficiently. The tools and libraries provided by Lucene exhibit peak performance when the combined together with the aforementioned aspects of properly parsing the content before performing the index and search operations. The Lucene search engine built in this project is able to obtain a maximum MAP score of 0.3477.

REFERENCES

[1] DOUG SPARLING. 2015. Full Text Search of Dialogues with Apache Lucene: A Tutorial — Toptal Developers. https://www.toptal.com/database/full-text-search-of-dialogues-with-apache-lucene [Online; accessed 16-April-2020].

[2] Jimmy, Guido Zuccon, and Bevan Koopman. 2016. Boosting Titles Does Not Generally Improve Retrieval Effectiveness. In *Proceedings of the 21st Australasian Document Computing Symposium (ADCS '16)*. Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3015022.3015028

[3] Jonathan Hedley. 2020. jsoup: Java HTML Parser. https://jsoup.org/ [Online; accessed 16-April-2020].

[4] The Apache Software Foundation. 2020. Apache LuceneTM 8.4.1 Documentation. https://lucene.apache.org/core/8_4_1/index.html [Online; accessed 16-April-2020].

[5] Andrew Trotman, Antti Puurula, and Blake Burgess. 2014. Improvements to BM25 and Language Models Examined. , 8 pages. https://doi.org/10.1145/2682862.2682863