Information retrieval - Apache Lucene

Quentin De Haes - S0172574 - quentin.dehaes@student.uantwerpen.be
Stijn Rosaer - S0172195 - stijn.rosaer@student.uantwerpen.be

November 2020

1 Lucene

1.1 Indexing

The first major functionality of Lucene is indexing. A Lucene Index consist of all documents [2] allowed to be searched. Each of these documents exists of a set of predefined fields. A field has a unique name, a data type, a value and a FieldType. These fieldtypes state whether these fields should be indexed or not, as wel as whether or not we should store their value for a return on search.[1]

In order to write an index, we require an IndexWriter class [3]. This class takes an IndexWriterConfig to set many configurations within the writer. Some examples of configuartions that can be edited are the similarity and analyzer that will be used.

The Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text, many different Analyzers exist, the StandardAnalyzer, SimpleAnalyzer and EnglishAnalyzer to give some examples. These are the ones we focus one in this project and tested in more depth.[4]

These analyzers each possess a tokenizer [5]. The standardAnalyzer uses the standardtokenizer which implements the Word Break rules from the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29 [6]. The standardtokenizer [7] has many filters, the standardFilter normalizes the tokens extracted by the standardAnalyzer, the LowerCaseFilter normalizes token text to lower case and the StopFilter removes stop words from a token stream. These stopwords are defined in a user provided list of

stopwords. A SimpleAnalyzer [4] filters a letterTokenizer [8] with a LowerCaseFilter, it does not use a stopFilter.

The EnglishAnalyzer [4] filters a StandardTokenizer with StandardFilter, EnglishPossessiveFilter, LowerCaseFilter, StopFilter, and PorterStemFilter.The EnglishPossessiveFilter removes possessives (trailing 's) from words, the PorterStemFilter uses the Porter stemming algorithm to transform the token stream.

Similarity [9] defines the components of Lucene scoring. It thus states which calculations are taken to acquire a score for each document. At indexing time, the indexer allows the Similarity implementation to set a per-document value for each field. There exists a BM25Similarity, TFIDFSimilarity, BooleanSimilarity [12] and more.

The TFIDFSimilarity [10] uses the termFrequency and the inverse document frequency to determine the score of a document. The term frequency $tf_{t,d}$ of term t in document d is defined as the number of times that t occurs in d. The IDF weight can be computed as $idf_t = log_{10}\frac{N}{df_t}$ with df_t the document frequency, the number of documents that t occurs in and N the total number of document in the collection. Tf-idf weight of a term is thus the product of its tf weight and its idf weight.

$$w_{t,d} = (1 + log(tf_{t,d})).log\frac{N}{df_t}$$

The OkapiSimilarity or BM25Similarity [11] uses idf weighting of the query terms present in the document as described above. It also takes the termf frequency and document length into account.

$$RSV_d = \sum_{t \in q} log\left[\frac{N}{df_t}\right] . \frac{(k_1 + 1)tf_{td}}{k_1((1 - b) + b.(L_d/L_{ave})) + tf_{td}}$$

- tf_{td}: term frequency in document d
- $l_d(Lave)$: length of document d (average document length in whole colleciton)
- k_1: tuning parameter controlling document term frequency scaling (default 1.2)
- b: tuning parameter controlling the scaling by document length (default: 0.75)

1.2 Searching

While indexing is certainly an integral part of using Lucene, noone will use Lucene for indexing alone without using it's searching functionality afterwards. In order to start searching, we'll need to parse our string that we would like to query into something that can be used by Lucene, a Query object.

One option for doing so is using a queryparser, or more specifically the MultiFieldQueryParser [13], where we pass a list of all fields we require to search this query on. Another option that we explored is the WildcardQuery. This allows the user to query the index based on a regex like string with wildcards. One major difference between the results is that the MultiFieldQueryParser returns a score for each document, whereas the WildcardQuery[14] returns score 1 for a document if the query results in a match for this document.

When scoring[15] the plethora of documents, we use a similarity to calculate said score. It is important that the same similarity and analyzer are used as the ones from indexing. Otherwise, an error or incorrect scoring may occure. Beside the already explained scoring models TFIDF and BM25/okapi, there is also the option the use the MultiSimularity, PerFieldSimularity that uses a different simularity for each field in the document and a SimularityBase. These are not further specified in this report but are provided by Lucene.

A nice to have feature of Lucene is the possibility to get an explanation[16] of how a document received their score given the different parameters. This is done by creating an Explanation for a specific document id, using the searcher. The result will depend on the used similarity and analyzer since it explains the value based on the formula used for scoring.

Lucene provides the possibility to perform spell checking using a the SpellChecker [17]. This takes as input a dictionary or file with correct words that will be indexed. Next, a sugention can be made for a misspeld word with a given minimum accuracy score.

2 Comparison

A good way to determine the feasability of Lucene for indedexing and retrieval capabilities to large datasets is testing its performace. We did this by comparing two different parameters to give a good overview of the best similarity algorithm and analyzers.

To create a good testing environment, we used the stackoverflow dump from where we took 1000 questions with there answers. This subset was used for every test to make it consistent.

We found that a good way to show accuracy is querying the index on the title of the question, not searching on the title field. If the top-k retrieved documents contain the document with that title, we can conclude that the retrieval was successful. It is important to know that we assume that the title is a unique identifier for a document and we always expoct only one document to be correct. This can be true for testing purposes but in real life scenarios might nog be true.

The accuracy can be calculated by taking the amount of correct retrievals (in the top-k results) divided by k times 1000.

2.1 Similarities and Analyzers

2.1.1 ClassicSimilarity (TFIDF)

2.1.1.1 StandardAnalyzer

k	Found in first k	Accuracy
3	796	26.5333%
10	913	9.13%
20	960	4.8%
50	981	1.962%

2.1.1.2 SimpleAnalyzer

k	Found in first k	Accuracy
3	804	26.7999%
10	915	9.15%
20	960	4.8%
50	983	1.966%

2.1.1.3 EnglishAnalyzer

k	Found in first k	Accuracy
3	832	27.7333%
10	941	9.41%
20	971	4.855%
50	992	1.984%

2.1.2 Okapi BM25 similarity

2.1.2.1 StandardAnalyzer

k	Found in first k	Accuracy
3	906	30.2%
10	954	9.54%
20	973	4.865%
50	986	1.972%

2.1.2.2 SimpleAnalyzer

k	Found in first k	Accuracy
3	903	30.1%
10	959	9.59%
20	974	4.87%
50	990	1.98%

2.1.2.3 EnglishAnalyzer

k	Found in first k	Accuracy
3	926	30.8666%
10	977	9.77%
20	988	4.94%
50	994	1.988%

2.1.3 Boolean similarity

2.1.3.1 StandardAnalyzer

k	Found in first k	Accuracy
3	757	25.2333%
10	841	8.41%
20	876	4.38%
50	913	1.826%

2.1.3.2 SimpleAnalyzer

k	Found in first k	Accuracy
3	767	25.5666%
10	848	8.48%
20	882	4.41%
50	918	1.836%

2.1.3.3 EnglishAnalyzer

k	Found in first k	Accuracy
3	817	27.2333%
10	897	8.97%
20	925	4.625%
50	959	1.918%

2.1.4 Conclusion

It is very clear that okapi/BM25 similarity is superiour in comparison to the two other tested similarity scoring algorithms. The documentation of Lucene also indicates that BM25 is newer and gives a better result than the classic similartiy.

The classic similarity is an implementation of the TFIDF algorithm wich uses a Boolean model and the vector space model.

When comparing the different analyzer, it is very clear that the English analyzer is the best. It is specificly optimized for English text. This is probably because our dataset consists of English text.
The standard analyzer is the most commonly used analyzer and recognises

almost everything. This is also why it is not as well performing as the others. It converts everything to lower case, applies a stop-filter if provides and The simple analyzer consists of a letter tokenizer and a lowercase filter. It splitses everyting on everything that is not a letter and user lowercase.

2.2 Field comparison

2.2.1 Question field

k	Found in first k	Accuracy
3	874	29.1333%
10	920	9.2%
20	942	4.71%
50	959	1.918%

2.2.2 Answers field

k	Found in first k	Accuracy
3	721	24.0333%
10	824	8.24%
20	866	4.33%
50	904	1.808%

2.3 Tags influence

2.3.1 With tags

k	Found in first k	Accuracy
3	926	30.8666%
10	977	9.77%
20	988	4.94%
50	994	1.988%

2.3.2 Without tags

k	Found in first k	Accuracy
3	908	30.2666%
10	963	9.63%
20	975	4.875%
50	985	1.97%

2.4 Conclusion

As we can see, Lucene is very useful tool for indexing and retrieving operations on large datasets. It is very important to build the right configuration by determining the optimal analyzer and similarity. We also noticed that the correct fields for retrieval give a major difference in accuracy, as well as the amount of top documents that should be returned to minimize false positives and false negatives.

Since we only used 1000 files for testing, we performed a test on all 1.2 milion files that we generated by preprocessing as well, using the settings of the best result. We noticed that there was no significant difference between the 1000 files, but that the indexing and searching took a bit longer.

3 Information

Our implementation of a search engine using Lucene can be found on github: `https://github.com/stijnrosaer/project-IR` Our implementation used a subset of the stack overflow dataset, however, due to the sheer size of this dataset, we decided to not include it in our repo. When running the program, we run main with a number of arguments.

The first argument can be 1 of 3 things: index, search and benchmark. each of these has a different use. index is used to index the files, search is used to query upon the files, and benchmark is used to compare the viability of the different analyzers, similarities, ... These arguments can each have a number of flags.

3.0.1 –documents (-d)

This flag is only used by the index command. It is a required flag that denotes the location of the documents to be indexed after the flag.

3.0.2 –similarity (-s)

Is the optional flag that is used by all 3 arguments. It states which similarity is to be used ,this should be the same similarity in both index and search (or benchmark). Three different values can be given to this flag:

- "classic": The standard similarity, which is also the one used when this argument isn't given. It uses the TFIDF algorithm to calculate the document scores.

- "okapi": The okapi, or BM25 similarity.
- "boolean": the boolean similarity.

3.0.3 –analyzer

Is the optional flag that says which analyzer we'll be using. This is a flag used by all 3 arguments. It should be identical in both index and search (or benchmark). Three different values can be given along with this flag, a more in depth explanation between the difference of the analyzers is given earlier in this paper:

- "standard": The standard analyzer.
- "simple": The simple analyzer.
- "english": the English variant of the language analyzer, since this analyzer gave the best results during our benchmark, it is also chosen as standard analyzer when the flag isn't given.

3.0.4 –query (-q)

This is a required flag only used in search. It passes the query. If the query is more than one word, the query must be surrounded by quotation marks to denote it's start and ending.

3.0.5 –field (-f)

Is the optional flag that is used to denote in which fields we will be searching the query, it can be used in both search and benchmark. Any combination of the following values can be given: "title", "tags", "question"and "answers", when the flag is not given, the fields "tags", "question" and "answers" are used.

3.0.6 –amount (-a)

Is the optional flag that states how many of the top documents should be returned.Only search has a use for this flag. When the flag is not given a standardvalue of 20 documents is given.

References

[1] Apache Lucene. "Interface IndexableField". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/4_4_0/core/org/apache/Lucene/index/IndexableField.html

[2] Apache Lucene. "Class Document". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/4_0_0/core/org/apache/Lucene/document/Document.html

[3] Apache Lucene. "Class IndexWriter". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/7_4_0/core/org/apache/Lucene/index/IndexWriter.html

[4] baeldung. "Guide to Lucene Analyzers". *baeldung.com* Retrieved November 2020, from https://www.baeldung.com/Lucene-analyzers

[5] Apache Lucene. "Class Tokenizer". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/7_3_1/core/org/apache/Lucene/analysis/Tokenizer.html

[6] Unicode. "UNICODE TEXT SEGMENTATION" *unicode.org* Retrieved November 2020, from https://unicode.org/reports/tr29/

[7] Apache Lucene. "Class StandardTokenizer". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/6_6_0/core/org/apache/Lucene/analysis/standard/StandardTokenizer.html

[8] Apache Lucene. "Class LetterTokenizer". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/7_1_0/analyzers-common/org/apache/Lucene/analysis/core/LetterTokenizer.html

[9] Apache Lucene. "Class Similarity". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/8_0_0/core/org/apache/Lucene/search/similarities/Similarity.html

[10] Apache Lucene. "Class TFIDFSimilarity". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/8_0_0/core/org/apache/Lucene/search/similarities/TFIDFSimilarity.html

[11] Apache Lucene. "Class BM25Similarity". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/8_0_0/core/org/apache/Lucene/search/similarities/BM25Similarity.html

[12] Apache Lucene. "Class BooleanSimilarity". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/8_0_0/core/org/apache/Lucene/search/similarities/BooleanSimilarity.html

[13] Apache Lucene. "Class MultiFieldQueryParser". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/7_1_0/queryparser/org/apache/Lucene/queryparser/classic/MultiFieldQueryParser.html

[14] Apache Lucene. "Class WildcardQuery". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/7_3_1/core/org/apache/Lucene/search/WildcardQuery.html

[15] Apache Lucene. "scoring". *apache.Lucene.com* Retrieved November 2020, from https://Lucene.apache.org/core/3_5_0/scoring.html

[16] Chris Perks. "Explaining Lucene explain". *chrisperks.co* Rertieved November 2020, from https://chrisperks.co/2017/06/06/explaining-Lucene-explain/

[17] Apache Lucene. "Class SpellChecker". *apache.Lucene.com* Retrieved November 2020, from https://lucene.apache.org/core/6_0_1/suggest/org/apache/lucene/search/spell/SpellChecker.html