Information Retrieval: Assignment 1

Nils Charlet
Sam Legrand

November 2020

1 Introduction

Apache Lucene is a text search engine that is widely used by major companies in order to power searching functionality on their websites. It consists of an extensive set of search algorithms and components that are very adaptable, such as a powerful query parser and ranked search. It is fast and space efficient, indexes are only about 20 to 30 percent the size of the original text that is being indexed. It also has some cross-platform advantages, being written in Java, but it is also implemented in other languages while using the same index format. [1] [4] Numerous major companies use Lucene to power searching functionality on their websites [14]: these include Apple, AOL, Comcast, and Disney.
A ton of information in this report was taken from the Lucene API documentation [8].

2 Index

The index that is stored in Lucene is an inverted index. Thus, each term is mapped onto the document numbers that contain it. It is important to note that documents are not simply stored as text in Lucene. A document is a sequence of fields, with each field containing some value/text. Each field can be tokenized in order to derive the terms that have to be stored in the index.
Lucene also supports dynamic document libraries through the use of segments. These are sub-indexes which can be searched separately or merged into a single index. Thus, when a library gets updated, the newly added documents can be added to a new segment which gets merged into the existing index. Note that document numbers may change when segments are merged.
The index also stores some additional information such as term frequency and proximity data for each term in the dictionary. The term frequency data can be used for optimization. The proximity data basically stores the position of the term in each document. Therefore, the index is by default a positional index. However, positional data can be omitted while adding fields to documents by changing the field type.

2.0.1 Custom indexes

Even though there are no other indexes provided by Lucene itself, it is easy to expand upon the existing implementation in order to end up with more advanced indexes. This is because the indexing pipeline itself contains multiple components (e.g. a tokenizer for tokenizing fields). Lucene provides base classes for these components, for which the implementation can be adapted. For example, a user can implement a k-gram index by adapting the tokenizer so that it generates k-grams for terms and adapting the query parser in order to convert wildcards to boolean queries and post-filter the resulting terms against the query.

3 Score models

Lucene contains lots of scoring models that can be used for ranking the documents. The default score model is 'BM25Similarity' [6]. The historical 'ClassicSimilarity' is based on the vector space model and thus uses tf-idf scoring. The documentation recommends using the optimized version of the Okapi BM25 model however, as it is generally considered superior to tf-idf.
We will briefly show an example where BM25 is performing better than tf-idf, using our document retrieval implementation (see Section 5). Our dataset consists of a subset of Stack Overflow posts. First, we use tf-idf and perform two searches, one using the query 'timestamp', the other using 'a timestamp'. 'a' is a word we expect in almost every document, and it does not hold much value. We would expect this to not affect the query results much. The following table shows the top 10 highest rated results for the given query using tf-idf.

	timestamp	a timestamp
1	If I have a PHP string...	If I have a PHP string...
2	Aging Data Structure in C#	SQL Server 2000: Is there...
3	SQL Server 2000: Is there...	Aging Data Structure in C#
4	Why does VS 2005 keep giving me...	Dealing with PHP server and...
5	Dealing with PHP server and...	Reading a C/C++ data structure in...
6	Reading a C/C++ data structure in...	Always Commit the same file with SVN
7	Why Doesn't My Cron Job Work Properly?	Inheritance in database?
8	Always Commit the same file with SVN	Why does VS 2005 keep giving me...
9	Inheritance in database?	Why Doesn't My Cron Job Work Properly?
10	Automatically incremented revision...	Best Practices for securing a...

We can see that while the top 3 contains the same results albeit with #2 and #3 switched, result 4 of 'timestamp' is put at #8 when using 'a timestamp'. This is a very large difference and it does not sound reasonable that adding 'a' to the query makes this result much less relevant. There are also some smaller differences where results are moved a few places.

Let's now look at the top 10 results of the same queries when using BM25 (default parameters: k1 = 1.2: Controls non-linear term frequency normalization (saturation), b = 0.75: Controls to what degree document length normalizes tf values).

	timestamp	a timestamp
1	If I have a PHP string...	If I have a PHP string...
2	Why does VS 2005 keep giving me...	Why does VS 2005 keep giving me...
3	Aging Data Structure in C#	Aging Data Structure in C#
4	SQL Server 2000: Is there...	SQL Server 2000: Is there...
5	Dealing with PHP server and...	Dealing with PHP server and...
6	Reading a C/C++ data structure in...	Reading a C/C++ data structure in...
7	Why Doesn't My Cron Job Work Properly?	Why Doesn't My Cron Job Work Properly?
8	Always Commit the same file with SVN	Always Commit the same file with SVN
9	Inheritance in database?	Inheritance in database?
10	Automatically incremented revision...	Automatically incremented revision...

Here we see that when using BM25, both these queries give the same top 10. This shows that BM25 is not sensitive to stop words which add very little to the query such as the articles 'a', 'an', and 'the'. Furthermore, the #1 result is the same as when using tf-idf for both queries, however there are some new and removed results, and some results are in a different place. It is hard to objectively judge whether these results are more relevant than those generated using tf-idf, however knowing that BM25 is not sensitive to these stop words is assuring.

Other possible scoring models include:

- Boolean scoring, where term scores solely depend on the query boost
- Language based models with different kinds of smoothing methods
- Information based models (using information gain)
- Other types of independence models

A full list can be found in the Lucene documentation [9].
There are also some base classes provided in order to implement a custom scoring model that uses a certain paradigm (e.g. tf-idf normalization, information gain, ...).

4 Search features

The classic query parser that is included with Lucene has an extensive list of supported features [7]:

- Field search: find queries in a specific field instead of the default field

- Wildcards: ? denotes a single character, * denotes multiple characters
- Regular expressions: supported syntax can be found at https://lucene.apache.org/core/8_6_3/core/org/apache/lucene/util/automaton/RegExp.html?is-external=true
- Range search: search for all fields values within a specified range
- Proximity search: search for words within a certain distance of each other
- Fuzzy search: search for a similar spelling of the word (optionally providing the maximum Levenshtein distance). Especially useful for words with similar spelling and (mostly) the same meaning (gray/grey, eat/eats, ...).
- Boolean operators: AND/+/OR/NOT/- operators
- Grouping: group clauses using parentheses
- Term boosting: boost the relevance of a term in the query

There are also a variety of other query classes [15]:

- Phrase query: matches sequence of terms with slop factor (maximum number of positions between terms) (default: 0) For example, the query "large house" can give "large expensive house" as a match with slop factor = 1.
- Multiphrase query: similar to phrase query but can allow multiple terms for each position, for example the previous query "large house" could be adapted to "large house/building" and could now match "large expensive house" and "large cheap building".
- Boolean query: allows you to combine multiple clauses (= query + operator) into a more complex query. Operators include
  - SHOULD: Use this operator when a clause can occur in the result set, but is not required. If a query is made up of all SHOULD clauses, then every document in the result set matches at least one of these clauses.
  - MUST: Use this operator when a clause is required to occur in the result set and should contribute to the score. Every document in the result set will match all such clauses.
  - FILTER: Use this operator when a clause is required to occur in the result set but should not contribute to the score. Every document in the result set will match all such clauses.
  - MUST NOT: Use this operator when a clause must not occur in the result set. No document in the result set will match any such clauses.
- ...

Some features, such as regular expressions and wildcards, are implemented by matching the terms of the dictionary to a state automaton that is generated from the query [5]. This results in some limitations. Wildcards are only applicable to single term queries and cannot be used as the first character of a search, as the matching of terms would become too slow [12]. An additional implementation of a permuterm index [11] or k-gram index would therefore allow for more flexibility and would result in much faster wildcard searches, at the cost of storage space. The fuzzy search could be seen as a form of spell correction but is not very advanced in that respect (no automatic correction when no results are found, maximum edit distance of 2, ...). For more advanced forms of spell correction, adaptations in tokenization (e.g. adding context-sensitive information for each term) and query parsing are necessary.

5 Document retrieval system

In this section, we will describe the implementation of a document retrieval system for a large dataset with Lucene. The dataset used is a dump of Stack Overflow. The code can be inspected in the Github repository [2]. Our code was bootstrapped from the example code of the Lucene documentation [8]. All code sources can be found under the 'reference code' directory.

5.1 Data preprocessing

The dataset is a large XML file which contains all Stack Overflow posts since 2008. Posts can be of different types (identified by a type id), we will mainly look at questions and answers. In order to be able to index documents, we first have to combine questions and answers that correspond to each other into documents. During this process, we select only relevant information for our document:

- Title of the question
- Creation date of the post
- Last edit date of the post
- Name of last editor of the post
- Content (body) of the post
- Tags of the question

This process is implemented in `preprocess.py` and makes use of the lxml library which buffers the input file into memory in order to limit memory usage [13]. After running `python3 ./dataset/preprocess.py`, we end up with a large number of XML files, with each file representing a single document.

5.2 Indexing

We've implemented the indexing of the documents using the `StandardAnalyzer` (default policy for extracting index terms from text) and default configuration of the `IndexWriter` (uses the BM25Similarity score model). We add documents to the index using this index writer through a custom function. This function reads every single document file and generates a virtual document for it, which is added to the index. For each document the attribute values are inserted into corresponding fields, which are added to the virtual document. Each attribute value is added to an 'All' field that is not physically stored.

5.3 Search

Querying and searching the index is implemented through the default `StandardAnalyzer`, `Scanner`, `QueryParser` and `IndexSearcher`. The `IndexSearcher` reads from the previously generated index. During indexing, we added a virtual 'All' field to every document that is linked to all information of that document. This allows us to use the 'All' field as a default field for the query parser. Thus, querying without explicitly defining a field will search for all information in every document.

5.4 Case-sensitive search

Due to the fact that the StandardAnalyzer converts tokens to their lowercase representation [10], search is not case sensitive in our current implementation. Both 'Why' and 'why' searches for the 'Title' field provide the same results:

	Title:"Why"	Title:"why"
1	Why functional languages?	Why functional languages?
2	Why is String.Format static?	Why is String.Format static?
3	Why aren't Enumerations Iterable?	Why aren't Enumerations Iterable?
4	Why does volatile exist?	Why does volatile exist?
5	FF3 WinXP != FF3 Ubuntu - why?	FF3 WinXP != FF3 Ubuntu - why?
6	Why should I learn Lisp?	Why should I learn Lisp?
7	Why does int main() compile?	Why does int main() compile?
8	Why go 64 bit OS?	Why go 64 bit OS?
9	Why doesn't **find** find anything?	Why doesn't **find** find anything?
10	(Why) should I use obfuscation?	(Why) should I use obfuscation?

A case-sensitive search could be useful in many instances (e.g. finding usernames which are case-sensitive, finding names of programming languages, modules etc.).
In order to provide a case-sensitive search, we have to implement a custom analyzer that doesn't convert the tokenized terms to lowercase. This custom analyzer is based on an example that was found online [3]. This custom analyzer overrides the `createComponents()` method of the base `Analyzer` class and uses

only the `StandardTokenizer` without any filters (see `./src/CustomAnalyzer.java`). When we now use this analyzer while indexing and searching we get the following results for the previously used queries:

	Title:"Why"	Title:"why"
1	Why functional languages?	FF3 WinXP != FF3 Ubuntu - why?
2	Why is String.Format static?	C# switch statement limitations - why?
3	Why aren't Enumerations Iterable?	What is reflection and why is it useful?
4	Why does volatile exist?	Virtual functions in constructors, why do...
5	Why should I learn Lisp?	What is cool about generics, why use them?
6	Why doesn't **find** find anything?	In Javascript, why is the "this" operator...
7	Why does int main() compile?	Perl: why is the if statement slower than "and"?
8	Why go 64 bit OS?	When do you use Java's @Override annotation...
9	(Why) should I use obfuscation?	In Cocoa do you prefer NSInteger or int, and...
10	Why does HttpCacheability.Private...	What is a magic number, and why is it...

Hence, case-sensitive search has been successfully implemented.

5.5 Using our program

Please make sure that Lucene version 8.6.3 is installed.
In order to generate an index from the original dataset, the following steps have to be executed:

- Put the `Posts.xml` dump file in the `./dataset` folder
- Run the command `python3 ./dataset/preprocess.py n` where `n` represents the number of documents to be processed (processing a subset of the data, if `n` is omitted or ¡= 0 the whole dump file will be processed)
- Run `IndexGen.java`

The index will be created in the index folder.

The code in `Search.java` contains a text based interface that allows you to make searches through the index. Upon startup, a brief explanation of the possible commands is given.

- `help`: show the brief explanation shown at startup
- `search` {`query`}: perform a new search using the given query. This shows you the the 'pagesize' topmost results. The page size defaults to 10 entries per page.
- `pagesize` {`n`}: allows the user to change the number of results per page. After changing the page size, the current page is adjusted to still show the topmost result of the previously selected page. For instance, having 10 results per page and being on page 6 will show result 51-60. Then

changing the size to 15 results per page will bring you to page 4, showing you results 45-59, still showing result 51.

- `next`: go to the next page
- `prev`: go to the previous page
- `seek {n}`: go to the page containing result `n`.
- `view`: show the same results again
- `select {n}`: choose a specific result and view it in full detail. This command will create an HTML file called `result.html` and open it in your default browser. This page contains the Stack Overflow question and all its responses, including extra data such as when they were last edited and by whom.
- `quit`: close the program

References

[1] Apache lucene: a high-performance and full-featured text search engine library. https://dzone.com/articles/apache-lucene-a-high-performance-and-full-featured.

[2] Github repository. https://github.com/SamLegrand/lucene.

[3] Guide to lucene analyzers. https://www.baeldung.com/lucene-analyzers.

[4] Linkedin's galene search architecture built on apache lucene. https://lucidworks.com/post/linkedins-galene-search-architecture-built-on-apache-lucene/.

[5] Lucene automaton query api documentation. https://lucene.apache.org/core/8_6_3/core/org/apache/lucene/search/AutomatonQuery.html.

[6] Lucene bm25 api documentation. https://lucene.apache.org/solr/8_6_3/solr-core/org/apache/solr/search/similarities/BM25SimilarityFactory.html.

[7] Lucene classic query parser api documentation. https://lucene.apache.org/core/8_6_3/queryparser/index.html?org/apache/lucene/queryparser/classic/package-summary.html.

[8] Lucene core api documentation. https://lucene.apache.org/core/8_6_3/core/index.html.

[9] Lucene similarities documentation. https://lucene.apache.org/core/8_6_3/core/index.html?org/apache/lucene/search/similarities/package-summary.html.

[10] Lucene standardanalyzer. https://lucene.apache.org/core/8_6_3/core/org/apache/lucene/analysis/standard/StandardAnalyzer.html.

[11] Lucene wildcard query and permuterm index. http://sujitpal.blogspot.com/2011/10/lucene-wildcard-query-and-permuterm.html.

[12] Lucene wildcard query api documentation. https://lucene.apache.org/core/8_6_3/core/org/apache/lucene/search/WildcardQuery.html.

[13] Parsing xml and html with lxml. https://lxml.de/3.2/parsing.html.

[14] Powered by lucene. https://cwiki.apache.org/confluence/display/lucene/PoweredBy.

[15] Search and scoring in lucene. https://lucene.apache.org/core/8_6_3/core/org/apache/lucene/search/package-summary.html#package.description.