KIV/IR Semester project

Search Engine

Jakub Šilhavý
A21N0072P
silhavyj@students.zcu.cz

19. 03. 2022

Contents

1 Introduction

This application represents a simple search engine that was developed as a semester project within the KIV/IR module at the University of West Bohemia. Using a boolean expression, it allows the user to search for specific documents within a given set of documents that was inputted into the application. The results of a search can be ranked either by cosine similarity or tf-idf. It also supports automatic language detection. However, for the sake of simplicity, the user can only search for documents in Czech or English.

1.1 User manual

The main purpose of this document is to explain the details of how this assignment was tackled. If you are looking for a user manual on how to use this application, please visit
`https://github.com/silhavyj/KIV-IR-Search-Engine`.

2 Implementation

2.1 Technology

2.1.1 Programming language

The entire application is written in `Java`. This decision was made based on previous experience with this programming language as well as its wide range of library support. The project was developed on Linux using `openjdk 11.0.14 2022-01-18`.

2.1.2 Build system

As for the build system, the project uses `Maven`, primarily because of its convenience when it comes to importing third-party libraries. The version of `Maven` used within this project is `3.6.3`

2.1.3 Dependencies

The entire list of dependencies along with their versions could be found in the `pom.xml` file, which is located in the root directory of the project structure.

- `org.openjfx` (`JavaFX` GUI library)
- `junit` (unit testing)
- `org.apache.opennlp` (stemmer - preprocessing)
- `com.github.pemistahl` (language detection)
- `org.json` (working with the JSON format)
- `org.jsoup` (fetching down articles from the internet)

2.2 Architecture

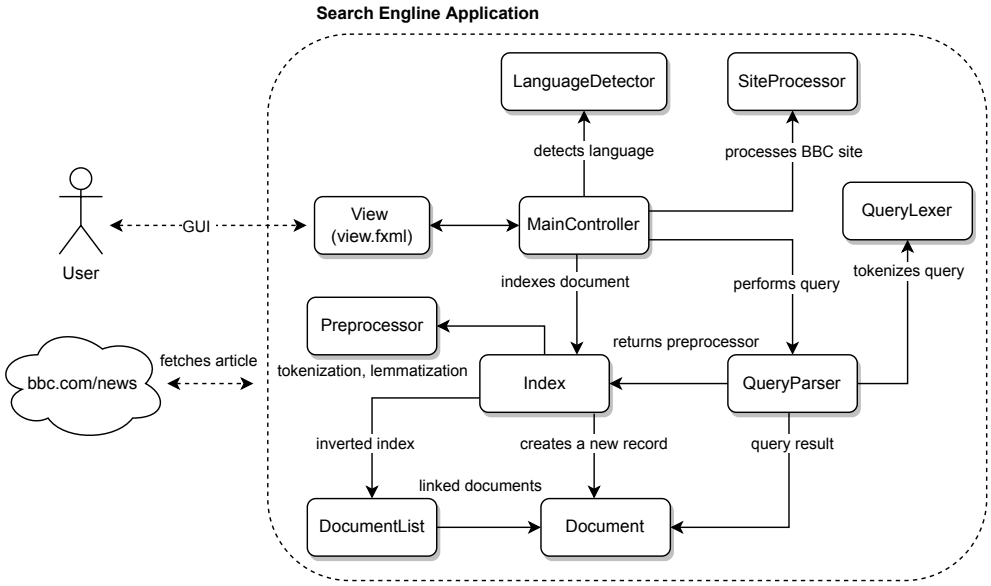

Figure 2.1: Graph of dependencies within different modules

As you can see in the figure above, the user interacts with the system via a GUI interface. The `MainController` class handles all events triggered by the user, e.g. inserting a document, executing a query, etc. With each event, it uses the backend logic, which is independent of the GUI, to perform the desired task. For more information about each class, please see the `Javadoc` documentation.

2.3 Input data

The application expects the documents to be in the JSON format of the following structure.

```
{ "datetime": "",
  "author": "",
  "title": "",
  "article": "",
  "url": ""
}
```

It is worth mentioning that the only obligatory fields are the title and article. The rest is just metadata that will be displayed along with each result of a search. The application distinguishes the documents by their location on the disk, not by their contents. If the user attempts to insert multiple instances of the same file (document), the first document will be indexed and the rest will be discarded.

All documents that come with the application can be found hosted over at `https://cutt.ly/5SzcWSG`.

2.3.1 BBC News

These documents represent articles that were fetched down using a technique called crawling. In total, there are 8,486 articles. All the articles were downloaded from `https://www.bbc.com/news`. Each document contains a URL that leads to its the original location on the website.

2.3.2 TREC

The documents stored in the trec folder were provided with the assignment as a serialized binary file. In order to keep things consistent, they were converted into individual JSON files which have the same structure as the documents downloaded from the BBC News. The overall number of the documents is 81,735.

2.4 Preprocessing

Before a document is indexed, it needs to be preprocessed first. This process involves language detection, tokenization, removing accents, removing stopwords, and stemming.

2.4.1 Language detection

In order to support multiple language, I took advantage of the following library `https://github.com/pemistahl/lingua`. This library supports many different languages. However, because each language needs to have its own stemmer and tokenizer, the number of languages supported in this application was reduces to two - Czech and English.

During debugging, it was discovered that the Czech language gets sometimes confused with the Slovak language. Therefore, the Slovak language and the Czech language are treated as the same.

2.4.2 Stemming

The process of stemming is done separately for both languages. For each language, there is a file containing stopwords of that language. All stop words are excluded from a document before it is indexed. The files can be found in the root folder of the project structure - `stopwords-cs.txt` and `stopwords-en.txt`.

English

As far as English is concerned, I used the following library to do the job `https://github.com/apache/opennlp`. In particular, I used the `PorterStemmer` class.

Czech

As for the Czech language, I used the same class that was used in the second practical of the KIV/IR module. This class can be found over at `https://github.com/dundalek/czech-stemmer`.

2.5 Inverted Index

There is a separate index for each language detected in the documents. Which index will be used for searching is determined by the language detected in the query. Inverted index is represented as a `HashMap` of `LinkledLists`. The key of each chain of documents is a term. The value represents a list of documents where the term occurs. The `Index` class also holds a map of locations of individual files on the disk.

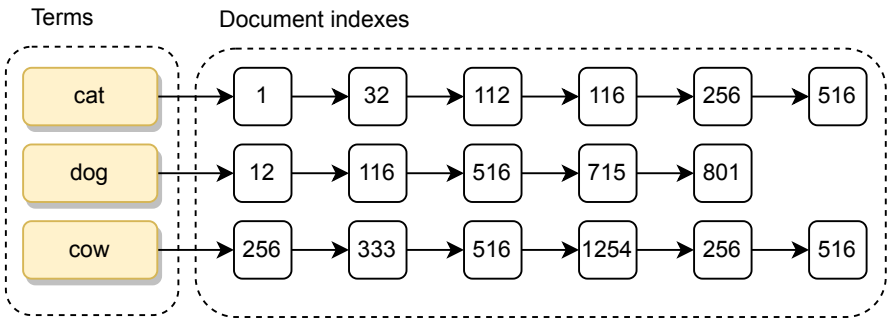

Figure 2.2: Structure of an inverted index

2.6 Search query

2.6.1 Query evaluation

In order to search for documents, I needed to evaluate a boolean expression the user inputs into the application. This expression, as required, supports `AND`, `OR`, and `NOT` operators. Operands represent individual key words the user wants to find in the documents. If two words are not explicitly separated by `AND` or `OR`, they are automatically evaluated as `term1 AND term2`.[1]

In terms of implementation, I decided to represent the operators as `'&'`, `'|'`, and `'!'` respectively. The user can also further specify the priority using a pair of parentheses. Examples of the query syntax can be seen in the highlighted box below.

```
> price & (gas | petrol) & (increase | high) & Northern Ireland
> Biden & !Trump
> Christmas & !(omicron | coronavirus)
```

As for query evaluation, I first converted the infix notation into a postfix notation, which is easier to parse. An explanation of this algorithm can be seen, for example, at `https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/`. An expression written in a postfix notation can be easily evaluated using a `stack`. An algorithm for this can be also found over at `https://www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression/`. Essentially, it all comes down to being able to evaluate a simple expression of one the following forms: `term1 & term2`, `term1 | term2`, or `!term1`.

AND

Calculating `AND` for two given sets of documents means finding the documents that appear in both sets. Given the fact that two sets are represented as sorted linked lists, using two pointers, we can calculate the intersection in linear time.

[1] Each word found in a query is also preprocessed using the same technique that is used for indexing a document.

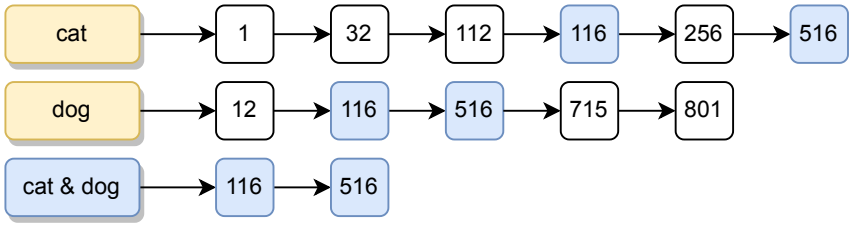

Figure 2.3: Example of `AND` operation

The final list of intersecting documents is also sorted, so it can be reused later on.

OR

The `OR` operation works as a union of two given sets of documents. This operation needs to make sure that there are no duplicities in the final result. With the assumption that the two sets are sorted linked lists, the `OR` operation can be calculated, also, in linear time.

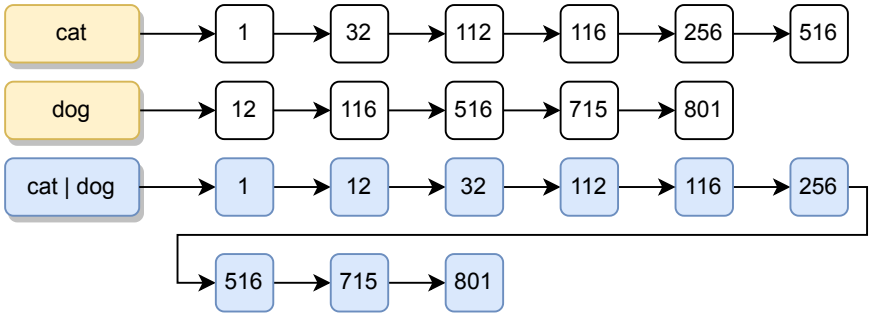

Figure 2.4: Example of `OR` operation

NOT

The `NOT` operation returns a complement of a given set of documents. It uses the `Index` class to retrieve all documents that have been indexed. In other words, it returns all documents except for those which are passed in as a parameter.

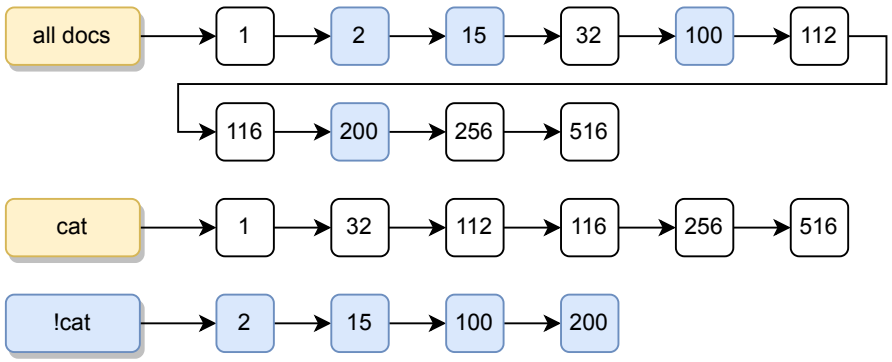

Figure 2.5: Example of `NOT` operation

2.7 Ranking document

2.7.1 Bag of words

There is an in-memory `bag-of-words` representation of each document held int the `Index` class. Essentially, there were two options.

The first option was to create a `bag-of-words` representation for specific documents based on a given query. The advantage of this is that it does not require as much space for the representation to be created. It is simply created upon evaluation of a query and destroyed by `Java`'s garbage collector afterwards. The trade off is that ranking might be a bit slower.

The second option was to create a `bag-of-words` representation for each document prior to a query being evaluated. This technique is much faster as there is no need to load and analyze the documents with each query. However, it does require more memory to store the documents.

In the end, I decided to go with the second option because I prefer fast searching.

2.7.2 Relevant words

In order to rank the results of a search, I needed to represent both the documents and the query the same way. Document representation has been done prior as it is explained in subsection 2.7.1.

As far as a query is concerned, it gets a little bit tricky due to the `NOT` operation. Relevant words are understood as the words the user is intending to find, whether it's using `AND` or `OR`. Let us consider the following example, for instance.

```
> cat & dog
> cat | dog
```

In both cases, the user is interested in occurrences of both words, to a certain degree. Therefore, both words are relevant. When it comes to `NOT` however, they become irrelevant as the user wants to find any document but the ones which contain the words.

```
> !(cat | dog)
```

This query says "find all documents that do not contain the words `cat` and `dog`". In this case, there would be no way to rank the results, as there are no relevant words in the query. The application does not know which documents are more important as they all have the same property - they do not contain the unwanted words. In order to find all relevant words of a query, I simply simplified the expression, so there are no parentheses. Simplification of the previous example would like so:

```
> !cat | !dog
```

You may argue that according to De Morgan's laws (`https://en.wikipedia.org/wiki/De_Morgan%27s_laws`), the simplification is incorrect and that it should be (`!cat & !dog`) instead. You are right, of course. However, as mentioned previously, in both cases, (`cat & dog`) and (`cat | dog`), the set of relevant words is the same.

The algorithm to simplify an expression can be found at `https://cutt.ly/sSQtXrl`. Once an expression is simplified, all we need to do to find the relevant words is to iterate through it and check which words are not prefixed with '`!`'.

2.7.3 Cosine similarity

Cosine similarly is calculates between two `bag-of-words`. The implementation is done according to the formula expressed at `https://en.wikipedia.org/wiki/Cosine_similarity`.

2.7.4 TF-IDF

The second way of raking documents, tf-ifd, is implemented as it is explained in lecture 07 on page 28. The piece of code that does the implementation can be found in `Index.java`.

3 Testing

There are, in total, 67 tests written for this application. Their purposes is to test the functionality of searching, evaluating expression, retrieving relevant words, etc. All tests are located in `main/src/test`. The framework used for testing is `JUnit`.

3.1 Operation

These tests are meant to test the `AND`, `OR`, and `NOT` operations as described in 2.6.1.

3.2 Query

This package tests the validity of a query (2.6.1) as well as retrieving relevant words from a valid query (2.7.2).

3.3 Search

This package tests searching documents using a mock in-memory index.

4 Conclusion

I believe that all compulsory features have been implemented. Besides, standard requirements, I choose to implement the following features as well.

4.0.1 Implemented extra features

- Later data indexing - adding new data to an existing index.
- Detection of query language and indexed documents
- Indexing web content - I enter a web site, the program downloads the data and indexes it directly into the existing index (`https://www.bbc.com/news`)
- GUI/web interface
- Documentation written in TEX
- Custom implementation of query parsing without using an external library

4.0.2 Achieved performance

The application was developed and tested on a laptop with the following parameters.

CPU: Intel i7-4710MQ (8) @ 3.500GHz
GPU: Intel 4th Gen Core Processor
Memory: 7638MiB

The bottle-neck of the application is its performance when it comes to indexing documents. It is due to language detection of each documents as well as the whole process of preprocessing. It takes about 1-2 minutes to index all documents of the `bbc-news` folder (8,486 documents; 47MB). To index the entire `trec-all` folder (81,735 documents; 372MB), it takes about 10-15 minutes to do so.

However, searching does not take more than a couple of milliseconds. The worst case I experienced was roughly around 30 ms. On average though, it takes up to 10 ms to perform a search and rank the results.