Documentation Programming Task IR

Team IR is da best

Members: Michael Thane M.Nr.: 204373

Theodora Kyriakou M.Nr.: 223825

Sebastian Hennig M.Nr.: 221088

We will first describe the general Workflow and then go into detail for each step.

# General Workflow

First we read the all relevant files from the Folder that is parsed via the command line. Then these files are parsed into **Doc objects.** These Objects contain all information to create the Index. We then check if there is already an index stored in the folder, which was also passed via command line. If there is, we load that index and if not we create a new index out of these Doc objects. After that it is decided which ranking model is used also dependent on the user input. When all these parameters are set we query the index with the last parameter that is parsed via the command line and print the results.

## Getting Relevant Files

We first get all relevant files (txt and html) from the folder provided in the command line args. We do this using a recursive folder traversal function which reads in all files in a folder when they have the right format and then proceeds to go to any subfolders and do the same thing again until there are no more subfolders left and returns all files found.

## Extracting relevant information

In the next step we extract the important information of the files into Objects. These are named **Docs.** A Doc has the following attributes: title, content and path. We create these Objects using a Factory. The getInstance() method of our factory takes a file path as input and returns a Doc object of that file. To successfully manage this step the files first need to parsed depending on their format.

## Parsing

Text:

To process the different documents we need to parse the content of the txt-File with the method „parseText“. It takes the path of the text as an argument and returns an array with two values, the title of the file and content of the text.

HTML:

In the same manner we are parsing the HTML files. In this case we take the title information which is present in the HTML-file. If it is not present, we will take the filename. For parsing the HTML-content we used the Jsoup-library.

Jsoup Version: [**jsoup-1.11.3.jar**](https://jsoup.org/packages/jsoup-1.11.3.jar)

Once we parsed all the files into our Docs we can use these to create the index.

## Indexing

Lucene Version: 6.6.0

Lucene Analyzer Commons: 6.6.0

Lucene Query Parser: 6.6.0

Following the parsing of the files, we moved on in creating the class file **Indexer.java**, in which we apply the Lucene library functions needed for analyzing the text and creating the index if it doesn‘t already exists.

First of all, we initialize an instance of the class IndexWriter. We are going to need this later in order to create the indexing process. We are creating a c**ustomized constructor for the Indexer class**, in which we will use the path of the directory of the index file as parameter. Inside the indexer class constructor are making use of the IndexWriter constructor whose parameters are the directory of the folder who will contain the index file and the IndexWriterConfig containing the desired analyzer. We chose to declare an instance of the class of EnglishAnalyzer, because by default it contains a lot of indexing filters like the **PorterStemFilter** which we need in our case.

Later, we declare a **closing function** for the index writer which will be needed after the indexing is done, and we are moving on to the creation of a fuction called **getDocument**, which returns a document that includes all the desired information needed for the creation of the index.

In the **getDocument** function we are creating three fields, one for the content of the document, one for its title and one for its directory path. We retrieved these fields from the parameter of our custom class Doc. Then we are adding them to the document that will be returned when we call the function inside the process of indexing, which is implemented with the function **indexFile.**

In the **indexFile** function, we use as a parameter an instance of our class Doc. In the first line of code, there is a message printed for the user that informs him about when every document is being indexed, showing the title of it in the terminal. Then an instance of the Document class is created, in which we attach the getDocument function that has as a parameter the doc variable. In the end, the document that was just created is being added in the Index Writer that we use.

Last but not least, we are calling the indexFile function inside the **createIndex** fuction that we create afterwards. In this function, the documents which were given as input from the user are indexed one by one inside a loop and then the number of docs that were indexed is returned.

## Query

We have created a customized class called **Searcher.java**, in which we are implementing the process of searching for the requested query inside the available documents.

Inside the Searcher customized **constructor** that we create, we declare the **indexSearcher** and the **MultifieldQueryParser** instances that we initialized in the beginning of the class, using the directory path of the index as a parameter that we need, in order to specify where the indexSearcher is going to find the index file for making the search of the query. What should be included in the search is defined inside the declaration of the MultifieldQueryParser, which in our case is both the content and the title.

Afterwards, we are creating **two recursive TopDocs functions** called **search** but taking different parameters as input.

In the first search method, the given query is parsed from string to Query type and the TopDocs results from a search that the indexSearcher implements is returned, based of course on the query and the maximum number of terms.

In the second search method, the query is given immediately without string parsing and together with the sort type that we choose to set in the main method when we call the function, the query search is implemented and the results are returned but this time they are naturally more specific as far as it comes to the sorting.

In the end, a document based on the score of indexSearcher is returned from the function **getDocument**. Also we are creating a **similarity function** that will be needed later in the main method for the ranking models that will be used.

## Ranking Model

In our **main** method, when we create an instance of our custom class Searcher in order to implement the query search, we are using the setSimilarity function created inside the Searcher class. We are determining whether the results will be returned based on the Vector Space Model (ClassicSimilarity) or the OKAPI BM25 ranking model (BM25Similarity). The user has already defined which ranking model he prefers from the path variables given in the terminal, which is stored in argument 2 under the name modeVS. modeVS can be 1 if the user wants the Vector Space Model or 0 if he wants the OKAPI BM25 model.