IR Programming Assignment 1

1. User Guide

The user is required to at least have Java version 7 or higher to run the code. The user can run the program in the following manner:

java -jar IR_P01.jar [path_to_document_folder]

In the following section we try to explain the flow of the overall program, its implementation, logic and sections of the code catering particular sections of our task.

2. Process Flow

2.1 Document Processing

Step 1: We start by providing the path to the set of test documents to be indexed. In our task we are trying to index only html and text files and hence the code only accepts these files and the rest of the files will be ignored/ a message stating not supported will be displayed.

Step 2: The next step is to parse the documents and to internally create tokens for that documents. Tokenization is the process of separating the sentence of the document into individual words called tokens. These tokens will be further helpful to build our dictionary.

Step 3: Once the tokenization is done, we need to stem the tokens so that we can identify the base form of the token which will be further added to our dictionary. This process is called stemming where we try to reduce the tokens to their base form so that different form of the same word is indexed together.

Step 4: All the above process refers to preprocessing at the end of which we have set of useful words and with the help of pointer we store the information about which documents these words are present. This is called indexing where the token along with the document in which the token is present is stored.

Note: In general while pre-processing, another process called stop word removal is performed where we remove most frequent words which occur in all documents which are not useful for document identification. However, in our code, we have skipped the process of stop word elimination because for our project we couldn't see recognizable difference in the size of the file indexed with stop word elimination and the file indexed without stop word elimination.

Also, since we were unclear about the domain under consideration, we did not consider the stop word elimination.

2.2 Query Processing

The query is to be processed in the same way documents are processed. The rules which were followed while pre-processing the documents are also applied to the query and the query undergoes tokenization and stemming in a similar fashion.

3. Implementation and Overall Logic

We've used the Lucene libarary as instructed. The lucene version is 8.3.0

3.1 Dependencies:

- org.apache.lucene (version 8.3.0) is the main component of the program.
  - lucene-core
  - lucene-analyzers-common
  - lucene-queryparser
- org.jsoup(version 8.1.3)
  - jsoup

3.2 Overall Logic

3.2.1 Index Directory:

This is the location of the index files where lucene stores the index fields of the parsed documents. Here we are creating a new directory every time the application is run. This is to avoid redundant indexing of documents.

3.2.2 Analyzer:

Lucene provides different types of analyzers including *StandardAnalyzer*, *EnglishAnalyzer*, *SimpleAnalyzer*, *WhiteSpaceAnalyzer*, *StopAnalyzer* etc. However, we've implemented a custom Analyzer, *TokenAnalyzer*, by extending the base class Analyzer by adding *LowerCaseFilter* and *PorterStemFilter*. The *LowerCaseFilter* converts the words into lowercase and the *PorterStemFilter*[4] stems the tokens into base words.

```
@Override
protected TokenStreamComponents createComponents(String fieldName) {
            final StandardTokenizer src = new StandardTokenizer();
            src.setMaxTokenLength(255);
```

```
            TokenStream tok = new LowerCaseFilter(src);
            tok = new PorterStemFilter(tok);
            return new TokenStreamComponents(src, tok);
        }
```

3.2.3 Index Writer and Document Parser:

Lucene provides an index writer which is configurable using an *IndexWriterConfig* to add the tokenized documents to the index directory. Here we configure this index writer with the *CustomAnalyzer* and the location of the index directory. We then use this configured index writer to index the files in the folders and sub folders available in the path specified by the user. [1]Shows how to index a simple file by adding different fields.

We also use Jsoup module to read html documents and identify <*title*/>, <*body*/> and <*summary*/> tags and add them as fields for indexing the document.

```
case HTML:
        org.jsoup.nodes.Document htmlDoc = Jsoup.parse(source, "UTF-8");

        String title = htmlDoc.getElementsByTag("title").text();
        String summary = !(htmlDoc.getElementsByTag("summary").isEmpty())
                            ? htmlDoc.getElementsByTag("summary").text()
                            : htmlDoc.body().text().substring(0, 50);
        String contents = htmlDoc.body().text();

        indexDoc.add(new TextField("title", title, Field.Store.YES));
        indexDoc.add(new TextField("summary", summary, Field.Store.YES));
        indexDoc.add(new TextField("contents", contents, Field.Store.NO));
        break;
```

3.2.4 Indexing Documents:

The following fields are added while indexing a document:

filepath – absolute path to the location of the document being indexed.

date – last modified date of the document being indexed.

Index – is the pseudo random *UID* generated for each document using *UUID.randomUUID*()

For text files:

Contents – string content of the file.

For HTML files:

Title – is the contents of the title tag.

Contents – is the contents of the body tag.

Summary – is the contents of summary tag. If this tag does not exist in the file then the first 50 characters are added as the summary of the file.

3.2.5 Index Reader:

Index reader is used to read the index directory and is injected to create a lucene index searcher. Here we are using *DirectoryReader* provided by Lucene as the index reader. [1]

3.2.6 Index Searcher:

Lucene provides an *IndexSearcher* to perform the document lookup by using the configured index reader.

3.2.7 Searching:

Concurrently search the indexed documents on “contents”, “title” and “date” fields. This query can be visualized as “contents: <text> OR title:<text> OR date:<text>”. [3]

3.2.8 Query Parser:

The goal of our application is to perform document search using multiple fields used for indexing concurrently. Hence we are using *MultiFieldQueryParser*[3] which constructs a boolean query with a term query for each of the fields specified. By default this query parser will perform an OR operation on these term queries. But it also performs an OR operation inside a term query with respect to the tokens obtained after tokenizing the search query. Therefore we override this default behavior to perform an AND operation inside each term query while performing an OR on available term queries.

```
MultiFieldQueryParser queryParser = new MultiFieldQueryParser(new String[] {
"contents", "title", "date" },
                    analyzer);
                    queryParser.setDefaultOperator(Operator.AND);
                    Query searchQuery = queryParser.parse(query);
                    indexSearcher.search(searchQuery, this.docCollector);
                    searchResults = docCollector.topDocs().scoreDocs;
```

3.2.9 Document Relevance Ranking:

Lucene provides a relevance score collector, *TopScoreDocCollector*[2], which computes the relevance scores and then ranks the documents accordingly. This can be configured to

return paginated results. We have configured this by setting the NUM_OF_HITS and TOTAL_HITS_THRESHOLD values.

3.3 Implementation and Code Structure

3.3.1 Flow Diagram

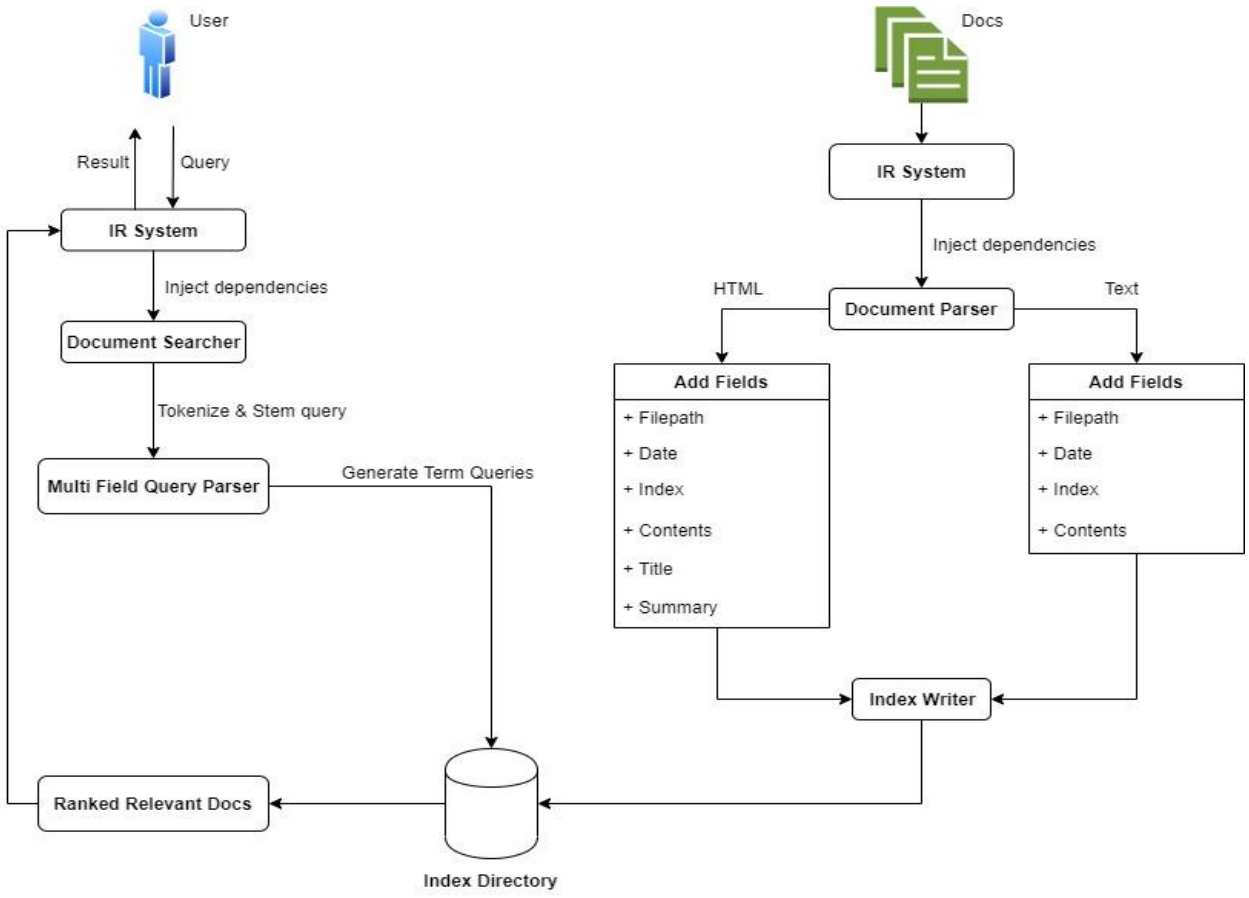

Fig 1. Process Flow

Our code is structured in the following format and the functionality is divided into different logic blocks which are:

Entry Point: InformationRetrievalSystem.java defines the main() method which is the entry point of our application.

Dependency Injection*: __inject_dependencies*() is invoked from the main() which initializes the dependencies and instantiates the *DocumentParser* and *DocumentSearcher* instances.

Document Parsing: DocumentParser.java encapsulates the parser for reading the text and html files available in the folders and sub-folders. *parseDocuments*() member is invoked to perform this task.

Document Searching: DocumentSearcher.java encapsulates the searcher for searching the indexed files. Search begins by invoking search() member function of DocumentSearcher.

Token Analyzer: TokenAnalyzer.java implements the custom analyzer used by our application.

4. Expected Output

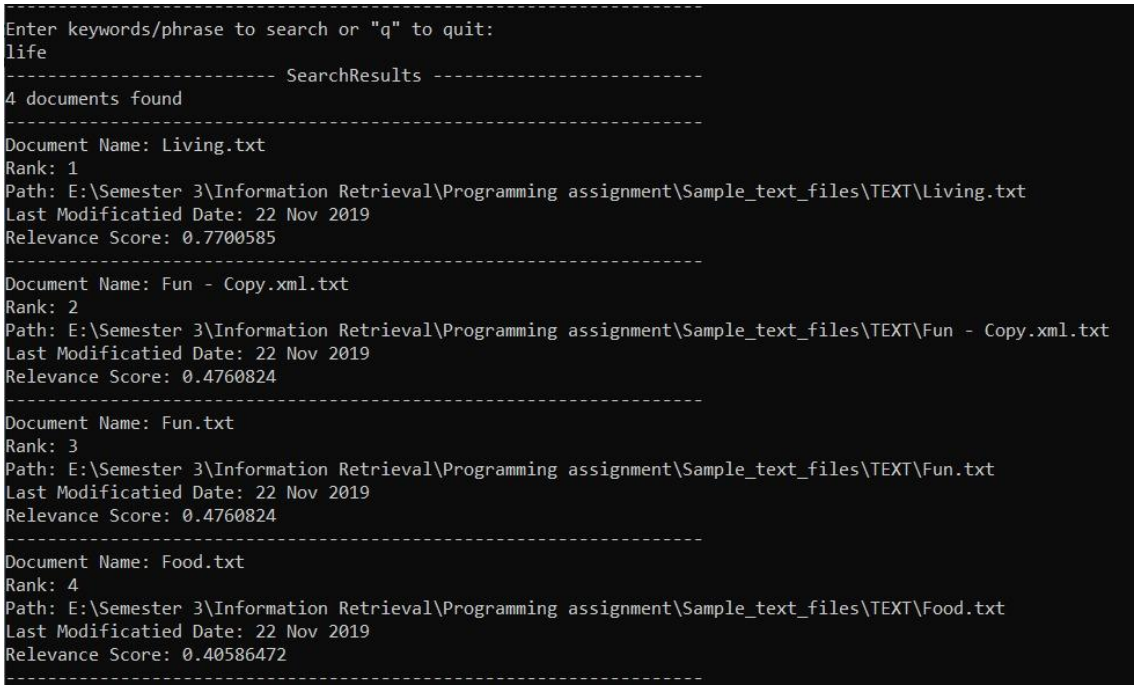

Fig 2: Expected Output

5. References

[1] lucenetutorial.com/sample-apps/textfileindexer-java.html

[2] lucenetutorial.com

[3] stackoverflow.com/questions/2005084/how-to-specify-two-fields-in-lucene-queryparser

[4] PorterStemFilter