Information Retrieval System

Marko Mekjavic — s1813308

October 2021

1 Introduction

In this report I will explain the development as well as the functionalities of my Information Retrieval System.

2 Project Objectives

The objective of this project was to develop and implement a simple **Information Retrieval System** that is able to **pre-process** texts from which it can retrieve information; create a **positional inverted index** of terms in the texts; as well as provide the following functionalities: **Boolean Search** for phrases, proximity, and words as well as **Ranked Information Retrieval** for text-free queries based on the ***TFIDS*** technique.

For the purposes of easier understanding and development of the system, I have broken the project into the following components:

1. **Prepossessing** - includes the methods responsible for pre-processing the text data
2. **Indexing** - includes the methods for generating the index
3. **Boolean Querying** - includes the methods responsible for preparation as well as the execution of the boolean search queries
4. **Ranked Querying** - includes the methods responsible for preparation as well as the execution of the ranked search queries
5. **Control** - contains the *main()* method, which controls the operation of the whole IR system by calling all of the sub-components

3 Text Processing

The first objective of the project was to implement the **pre-processing methods** to prepare the text data to be ready for the system to use and operate on. This component is hence used for any pre-processing of texts in the project, which allowed for easier development of other components. The pre-processing component is further divided into three sub-components; *Tokenisation*, *Stopword Removal*, and *Stemming*. The three sub-components are being called from the main preprocessing method, which takes a text in a form of a string as an input and then passes it onto the three subcomponents. The first operation to be performed is tokenisation, after which the stopwords are being removed from the text, and lastly the text is stemmed.

3.1 Tokenisation

The first step is to split the text - decision was made to split it at every non-alphanumerical value, using a simple regular expression `[â-zA-Z0-9_]`, which was inputted to the Python `split()` method.
The text is then split on *hyphens* and *apostrophes* which simplifies the design of the pre-processing. However, the downside of this approach is that some composed words are split and hence their meaning can be miss-understood in some queries. For instance documents containing "middle-east" might still appear in unrelated queries that contain the word "middle". Nevertheless, phrase search is not affected by this approach.
Next steps are the removal of numbers and case-folding. The decision to not split the text on numbers but rather remove them was made because many companies include numbers either in their names or in the names of their products. By simply removing the number the name of the company/product remains intact.

3.2 Stopword Removal

From the list of words produced after tokenisation, we then remove any stopwords that have been provided to us. Since the stopwords provided are not stemmed, the decision was made to first remove the stopwords from the text, and then stem the rest of the remaining text. This way some of the computational overhead was saved.

3.3 Stemming

The last step in pre-processing of the text files is stemming. For obtaining the stem of each word I decided to use the `PorterStemmer` from the `nltk.stem` library. The new list of stemmed words will match words of similar form disregarding their plurality, case or conjugation, and hence also decreases the size of the inverted index.

4 Indexing

The next step is to create an inverted index, which is created dynamically as we are reading through the document collection.

4.1 Algorithm

The algorithm responsible for generating the inverted index contains two steps; **Document Analysis** and **Generating Index**. As an input it takes an already parsed collection of files in a form of an `ElementTree` object, provided by the `xml.tree` library in Python.

4.1.1 Document Analysis

Before we start generating an inverted index, we must first prepare the documents, which have already been parsed into a `ElementTree` object. Since we need to keep track of the words in each document, we need to extract each document's ID as well as their text. In order to accomplish that, we need to look at the following three tags:

- `<DOCNO />` - tag containing the document number
- `<HEADLINE />` - tag containing the headline of the document
- `<TEXT />` - tag containing the actual body of document

For purposes of easier access to the files as well as their identification in the future, I decided to use a dictionary where the key is a document's number, and the value is the corresponding text, which ensures faster access with look-up time of *O(n)*.
As we go through the `ElementTree` object, we extract each document's number from `DOCNO` tag, and its text from `HEADLINE` and `TEXT` tags, which is then pre-processed as described in Section 3.

4.1.2 Generating Index

After we generated a dictionary of documents, we can generate an **inverted index** from it, which stores positions of a word in every document that word occurs. For its implementation I decided to use a dictionary, where keys are the words that appear in the documents, and the values are dictionaries with keys being the document numbers and the values being lists of numbers representing the position of a word in that given document. The positions start from 1 rather than 0.
We generate the index by going over every document and check for every word if it already exists in the index. If not, we update the index by storing the word as a key and its value as a new dictionary with the key being documents number, and the value being an array with the position of the word. On the other hand, if the word is already in the index, we take its value and check whether the current document is already in the inner dictionary. If satisfied, we take its value and append the position of the word to the list. Otherwise, we update the inner dictionary and add the document's number as a new key with a value being a list with the position of the word.
After the index is generated, we can easily output it into a text file with a desired format. The data structure we decided upon, allows easy calculation of the term frequency as we only need to consider the number of documents each word appears in - which corresponds to the number of all keys in the inner dictionary of the given word.

5 Query Processing

The first step in processing the queries is to **read them directly form** the file and **prepare them for execution**. When they are read from the file as strings we must first remove the number indicating the query's position in the file as well as the new line character at the end.

5.1 Singular or Compounded

The first division of queries is into **compounded** and **singular** ones. **Compounded queries** are composed of two or more singular queries, which are connected with a logical connector `AND` or `OR`. The system first checks if the query is a singular or a compounded one by checking if it contains an `AND` or an `OR`.
In case we are dealing with a compound query, it is broken down into components and each component is being analysed as a singular query.

5.2 Singular Query Types

The first step in analysing singular queries is to check if they are negated or not. This is accomplished by checking if the query contains a `NOT`. The second step is to divide queries based on their type. There are three different query types the system can distinguish; `Word`, `Phrase`, `Proximity`. Once the query is being classified, it is being processed accordingly.

5.2.1 Phrase

Phrases are queries that are of the form:

"term1 term2"

They are recognised by detecting *"* as a first and last element, which is then removed and the phrase is split into first and second word. Tuple of the form *(term1, term2)* is created, which is then used for execution of the query. The last step is the pre-processing of each term as explained in Section 3.

5.2.2 Proximity

Proximities are queries that are of the form:

#number(term1,term2)

They are recognised by detecting the # as a first element. The query is then split on every non-alphanumerical character and a tuple of the form *(number, term1, term2)* is created. The last step is the pre-processing of each term as explained in Section 3.

5.2.3 Word

All queries that are not recognised as `Phrases` or `Proximities` are then recognised as words.

6 Query Execution

After the query is processed classified into a type, it can then be executed by the system. In terms of execution it can either be based on a **Boolean** or **Ranked** retrieval technique.

6.1 Boolean Retrieval

When executing boolean queries we must first classify if a query is **singular** or **compounded**. In case of the later the system divides them into the `AND` and `OR` queries. In both examples the system will execute both singular queries and return the result for both. After the results are returned, they are used differently to calculate the end result.

6.1.1 Word

The easiest execution is for queries that are composed of singular words. The system will query the index - which is represented as a dictionary of dictionaries, as explained in Section 4.1.2 - and retrieve only the numbers of documents the word appears in.
In case we are querying only one word and there is more than one result, the system will sort the results based on their document number and return them in an ordered format.

6.1.2 Proximity

When executing a proximity query the system searches the files for *term1* as well as for *term2* and returns their results separately. The results are in the form of a tuple `(document number, position)`. These are then compared and if the two words are positioned within the the number indicated by the proximity, then they are added to the result.

6.1.3 Phrase

When executing a phrase query the system searches the files for *term1* as well as for *term2* and returns their results separately. The results are in the form of a tuple `(document number, position)`. These are then compared and if the two words are positioned one after the other, then they are added to the result.

6.1.4 And

When both results, for first and second query, are returned in a form of a set, an intersection of both sets is being calculated by using the built-in Python method *a.intersection(b).*

6.1.5 Or

When both results, for first and second query, are returned in a form of a set, a union of both sets is being calculated by using the built-in Python method *a.union(b).*

6.1.6 Not

When executing a negated query, the system first executes the query, and when the result is returned in a form of a set, a subtraction operation is performed. From the list of all documents we subtract the ones that are returned when the query is executed.

6.2 Ranked Retrieval

For free-text queries, we search the documents for each of the terms in the query. Then for each document in each posting, we need to calculate their score explained bellow.

6.2.1 Score

There are two metrics we need to consider in order to calculate the score. The first one is called *term frequency* and is a count of how many times a term occurs in a document.
The second measure is the *inverse document frequency*, which is derived from the term's *document frequency* - the number of documents the term appeared in. The more documents the term appears in, the less relative it is when calculating the score. Hence, the higher the *inverse document frequency* of a term, the rarer the term is and therefore has a higher importance when calculating the score.
The *TFIDF term weighting* combines the two measures in the following way:

$$w_{t,d} = (1 + log_{10} tf(t,d)) \cdot log_{10}(\frac{N}{df(t)})$$

We must then calculate the score for every term in the query and sum them up. This sum then represents the score of the document for the whole query:

$$Score(q,d) = \sum_{t \in q \cap d} w_{t,d}$$

The documents are then sorted in a descending order by their query score and returned to the user. All scores are rounded to four decimal places and only up to 150 results per query are returned.

7 Control

The whole functionality of the system is being controlled in the `main()` method, which takes as an argument the name of the *xml* file. Firstly, the *xml* file is being parsed into a `ElementTree` object using the `xml.etree` library in Python. Secondly, the data in the file collection is preprocessed as explained.
Afterwards the index is being generated and the boolean and ranked query search are being performed.

8 Lessons Learned

Implementing an Information Retrieval system from scratch gave me an in-depth understanding as well as an appreciation for the algorithms used. Development of such system also taught me the importance of planning in order to structure and organise the project in a format where individual components can be easily tested and replaced if needed.

9 Challenges Faced

When developing the querying system I had to refactor it quite a few times and this has thought me to really test my ideas before starting implementing them.
Navigating over large sets of data has taught me to discover much more efficient data structures, such as dictionaries and sets, and to use list comprehensions instead of nested for loops as they allowed for easier error spotting.
Lastly I would like to point out the importance of dividing the functionalities into different helper methods as they can then be easily reused and replaced.

10 Improvements

Of course there are several ways of improving the system.

10.1 Document Storage

Storing the whole file collection into a dictionary can be very impractical when dealing with larger collections of files. Keeping them in an *xml* file might be a better solution but it would be harder to perform operations on the file collection.

10.2 Query Parsing

The parsing of queries can also be significantly improved. For instance the current system only expects the phrase to be constructed of two words but in the future it could be refactored so that larger phrases can be queried as well.
Another improvement would also be implementation of querying of nested compounded queries, which currently are not supported.