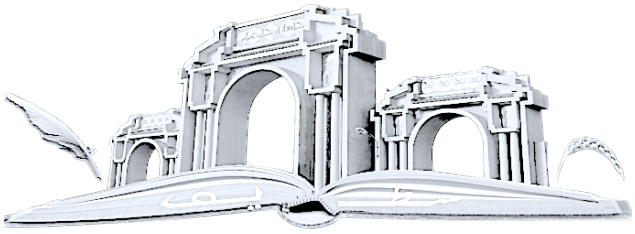
Report:

Azzouz AbdelDjouad Aymen
Fellaouine Mohamed Abderrahim
Faculty of Sciences, University of Ferhat Abbes



A Simple IR Tool

Based on lectures 3-7 of this course (Information Retrieval) and the guided work we were asked to implement on labs 0-3, we were able to build a simple search engine, or let us say search engines. This IR tool includes preprocessing on the collection and the queries provided by Dr. Harrag Fouzi, creating positional inverted index and using it to perform different search engines:

- Boolean search
- Phrase search
- Proximity search
- Ranked IR based on TFIDF

- details on the methods we used for Preprocessing the text:

Based on our tests, results differ when we apply preprocessing or use different preprocessing tools, so for our results to be accurate, we applied the same preprocessing tools on the collection and on the queries:

1. **Tokenization**

   Tokenization is a key (and mandatory) aspect of working with text data, Tokenization is a way of separating a piece of text into smaller units called tokens(in our case words).

For the tokenisation of the documents, I am using a regular expression with the pattern r'\d+\.\d+(?:bn|m|km)?|\w+'. This regular expression will match alphanumeric characters, underscores, and any numbers (including decimal numbers). Aditionally, it will match numbers containing the words ‘bn’, ‘m’ and ‘km’, which are abreviations for billion, million and kilometers, respectively. The decision for this, is that the given collection comes from the Financial Times, and terms like 7.5m and 10bn are very common.

2. **Stopping and Stemming**

Regarding the stemming, the porter stemmer from the library NLTK was used.

It is worth mentioning that even though stemming is applied after the stopwords removal, words that one may consider as stopwords, can still appear on the inverted index. For example, in the collection of documents, there is a company called AT Mays, and when this is tokenized and stemmed, it results in [at, may], so the word may is saved into the inverted index (‘may’ as in the stopword ‘may’, will still be removed).

3. **inverted index**

The first step to create the inverted index, was to parse the XML collection, and for this task, I used the ElementTree XML API. Then, I iterated through each document on the collection, combined the headlines and text of each document, and proceeded to tokenize them using the command re.findall and the regular expresion mentioned above. Furthermore, each token that was not in the list of stopwords was stemmed using the porter stemmer. Next, in that same loop, I iterated through each token on the document, and stored all of the tokens that appeared in the document, together with their position number and document frequency. I did this using a defaultdict of lists. Figure 1 shows an excerpt of the inverted index for the word ‘person’:
Where 360 is the document frequency, the elements on the left are the document Ids and the elements on the right are the positions of the word person in that document. For example, in the document 5001, the word person appeared four times, in position 75, 86, 122 and 118.

- details on how we implemented the four search functions:

For the boolean searches (including proximity and phrase searches), I implemented a parser that can recognize and categorize each type of query. This parser makes use of regular expressions and python sets. AND, OR and NOT operations were implemented using the intersection, union and difference set functions, respectively. For proximity and phrase searches, a function that is called inside the parser was implemented.

Proximity and phrase searches: For this, I simply used the algorithm provided on the lectures. To implement this in python, given two words (word a and word b), I used the inverted index to find the documents and positions of where those two words appeared. Then, I created a temporary dictionary with format {documentID: ([positions of word a], [positions of word b])}, and for each occurrence of word a, I iterated through all the occurences of word b and calculated (position of b – position of a) == distance; if this is true, the documentID is stored. This is repeated for all the documents where both word a and word b appeared.

For phrase search, the distance is set to 1, and the order of the words matter (no absolute value). On the other hand, for proximity search, abs(position b – position a) == distance was computed, and the value of distance is adjusted according to the query prompted by the user.

Boolean searches: As mentioned, a parser to process boolean queries was implemented; this parser can handle a variety of cases. For word queries without spaces (such as 'Scotland'), the parser simply returns all the documents where that word was present. Similarly, queries with AND/OR are also recognized, and the parser uses sets intersection/union for these cases. Furthermore, for AND queries, the parser can also detect appereances of the NOT operator. Combinations of the OR and NOT operators are not implemented simply because this type of query does not make much sense, and was not required for any of the queries provided; however, it can be easily implemented if required. Aditionally, the parser can also deal with combinations of phrase queries and AND(with NOT)/OR operators. In the case of proximity searches, the parser does not handle AND/OR operators, simply because it was not required, but this can be easily implemented if required.

Ranked searches: When a ranked query is entered, a dictionary containing the tfidf of each word in the query is created. Similarly, for each word in the query, the tfidf of that word for all the documents ID is calculated using the inverted index, and if the word does not appear in that document, it is set to 0. Then, to calculate the scores of each document, for each word in the query, if the word

exists in the document, compute $\sum tfidf_{query}[word] * tfidf[doc][word]$ and append the score of that document to a list. This process is repeated for all of the documents in the collection, and the results are sorted by the descending order of score.

It is worth mentioning that the tfidf of the queries and documents is calculated as given in the coursework instructions, and for both, no normalization is applied.

- a brief commentary on the system:

Overall, the system's operations are derived from the inverted index which is created from a collection, it serves as a base for all of the implemented functions. Also, having a parser for the boolean queries makes the system very compact and easy to implement and use. By no means this is a perfect system, and much improvements can be implemented.

- what challenges you faced when implementing it:

This humble project was a such an educative experience for us, where we did not only learn the theory behind search engines, but we got to experience building ones.
They say you can never truly learn if you don't practice and with practice comes challenges and we had our share of them. We ran sometimes into a wall because we kept complicating our work, fortunately we were able to go back and simplify it.

- any ideas on how to improve and scale your implementation:

Improving this project can only come by testing and each time learning from the results, ideas that improve our system must be kept, otherwise discarded.

- We definitely must start with preprocessing, as it has such a direct impact on our results.
- A cleaner code can lead to better results (Faster, more optimized, more accurate)
- We can implement a graphical interface for easy interaction and better impression, so maybe a simple web application can achieve this.
- For scaling the implementation, the system can be uploaded to a website and the inverted index stored on a database.