Text Technologies for Data Science
Assignment1 - Report

s1813674

31st October 2021

Abstract

The goal of this coursework is to implement a simple IR tool, that Pre-processes text through tokenisation, stop-word removal and stemming, creates a positional inverted index and uses that positional inverted index to perform multiple searches (Boolean search, phrase search, proximity search and ranked IR based on TFIDF)

1 The Trec-5000 Data-set

As an input, we were provided with a trec dataset with around 5000 documents in XML format. For the trec collection, we include the headline and text fields to the index. For positions of terms, we start counting from the headline and continue with the text. To read this file, we use an XML parser from the xml.etree.ElementTree python library.
In our implementation, we use a function to split the input xml file into multiple text input files keeping only the Headline and Text items.

2 Pre-Processing

For the Pre-Processing, we use a function that takes a document name as an argument. The function opens the document and creates an output file that will contain the pre-processed content. We read the input file line by line and perform the pre-processing tasks.

- Tokenization: For this step we use a regex expression that only keeps word characters (w+).
- Stemming: To Stem we use a Snowball Stemmer from the nltk python library
- Stopping: We make sure to not consider the words in the stop words list provided.
- Case Folding: We transform every term to its lowercase version.

We do this line by line starting with the tokenization, then converting to lowercase, then removing the stop words and finally we end by stemming. Note that in the interest of consistency, we also apply the same regex to the list of stop words such that for example you'll becomes youll.

3 Inverted Index

The inverted Index construction relies on the pre-processed documents. We loop through each of these documents extracting the file number from the documentName. The inverted index is a dictionary containing the term, it's document frequency, and then a mapping of docIDs and positional parameters. E.g

```
inv_index["term"] = {1,{150: [1,2,3]}}
\\ Where 1 is the document frequency, 150 is the DOC ID
\\ and [1, 2, 3] is the positions of "term" in that doc
```

For each term, we first check if the term has existed in that DocID before. If it has we just append the position. Otherwise, we initialize a new positional index for that DocID and increment the document frequency by 1.
If this is the first encounter, i.e, first time this term appears, we initialize the list to an empty list, the document frequency to 1, the doc lists to empty dict, and add the DocID with the term position to the newly created dict.

4 The Search Functions

For our Simple IR, we have 4 main search functions (each with their helpers that we won't mention for conciseness).

4.1 Phrase Search

This function is called when the query input is of the form "term1 term2". It takes the query string as an argument and also has a flag argument (simply because it will slightly differ when called from another function).
Before performing the phrase search, we perform tokenization, case folding and stemming to the query while making sure to keep the query number.
If both term exist in the positional index, we get the docs containing the terms separately and keep the intersection of the lists. For each doc in the intersection, we compare the term positions making sure that the difference between the positions is equal to 1. If it's equal to 1, we consider the search successful and add the doc to the list of results.

4.2 Proximity Search

This function works exactly like the phrase search but with an input of the form #proximity(term1, term2). The only difference is that we don't require

the difference between the terms to be equal to 1. Instead, we require it to be less than or equal to the #proximity given.

4.3 Boolean Search

This fuction takes care of queries containing "OR", "AND" and "NOT" operators (Note that for queries where it's just a single word, we automatically look in the inverted index).
We start with the case when there's an AND in the query. We separate the two terms (LHS and RHS of "AND). We then loop through the terms. The first check we perform is checking if the term is negated. If it is, we check if it's a phrase search. If it is a phrase search we call the appropriate function and then call a helper not function. Otherwise, we directly call the helper not function. If the term is not negated, we also check if it's a phrase Search and either call the phraseSearch function or simply look in the index if it's a single term. Once we got the document numbers relevant for each terms, we take the intersection and return that as the result.
We do the same thing if it's an "OR" operation, only this time, instead of taking the intersection of the two lists, we take the union and return it.
Note that, as per coursework guidelines, boolean queries will not contain more than one "AND" or "OR" operator at a time. But a mix between phrase query and one logical "AND" or "OR" operator can be there . Also "AND" or "OR" can be mixed with NOT.

4.4 Ranked Search

For this function, we rely on this formula for calculating a TFIDF score.

$$Score(q,d) = \sum_{t \in q \cap d} (1 + \log_{10} tf(t,d)) \times \log_{10} \left(\frac{N}{df(t)} \right)$$

Same as before, we start by preprocessing the terms in the query and getting a list of all the documents that exist in our collection.
The first step, is looping through all the terms in the query, and getting the document frequency from the index. Then for each term we loop through all the documents and we compute the term frequency (number of times the term appears in the document).
If both the document frequency and the term frequency are greater than 0 then we compute the inverse document frequency such that $idf = \log(N/df)$. We then compute the score (from the formula) and add the term score to the total score for the document.
Once we have scores for all the documents, we sort them in descending order and only return the 150 top documents for each query (Note that, the coursework asks for no normalization).

5 The System

For our implementation, we created a SearchEngine Object with various global variables and functions. The system works like this:

- Create a Search Engine Object
- Call the splittingDocs() function to split the XML file into multiple input files
- For each input file, we preprocess it's content that are going to be saved in a new output file
- Once the preprocessing is done, we delete the input documents as we don't need them anymore
- We then generate the inverted index using the outputted files.
- Once the index is generated, we delete the output documents as we don't need them anymore and write the index to a file.
- We then run the Boolean queries by using the queries.boolean.txt file. Our function will detect the kind of query we need to run and call the right search function. Once the searches are done, we write the results to a file (results.boolean.txt)
- Lastly, we run the ranked queries by using the queries.ranked.txt file. Once the searches are done, we write the results to a file (results.ranked.txt)

6 Challenges and Things I learned

What I learned through this coursework is that it's very important to consider every edge case. The Boolean search function was one of the toughest part simply because it needs to take a lot of things into consideration. It is also very challenging to work with a large corpus of files because we need to stay time and space efficient. However, working with an inverted index is very useful because it gets rid of the need to have documents available.

7 Possible Improvements

I think there are a lot of possible improvement to that simple IR. The first improvement would be getting rid of the need to create input/output files as it's a long process that also requires a lot of memory especially if there are even more documents even though we only run this once. Another possible improvement would be for the boolean search. The current implementation is a bit hard to scale (say we have multiple AND and ORs). It would be interesting to implement a system with some kind of precedence "scores" for operators so we can handle ANY type of query.