THE UNIVERSITY *of* EDINBURGH

Text technologies for Data Science

Coursework 1

Student ID: S2258327

1. **Details on the methods you used for tokenisation and stemming**

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.

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).

2. **Details on how you implemented the 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’:

```
'person': [360,
 {'5001': [75, 86, 112, 118],
  '5005': [54],
  '5008': [538],
  '5009': [231],
  '5020': [145],
  '5021': [38],
  '5029': [57],
  '5031': [203],
  '5043': [81],
  '5056': [205],
  '5063': [265],
  '5064': [231],
  '5070': [212],
  '5072': [167],
```

Figure 1: Excerpt of inverted index

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.

3. **Details on how you 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.

4. **A brief commentary on the system as a whole and what you learned from implementing it**

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.

This coursework has been extremely valuable to me; the majority of the things that were implemented were new to me. I come from a electronic engineering background, so I knew how to code before, however, I did not know about concepts like regular expressions, and complex data structures such as the inverted index.

5. **What challenges you faced when implementing it?**

Without any doubt, the main challenge was how to build a parser for the boolean queries. While this was not strictly required, I challenged myself on doing so. This was particularly difficult, because as mentioned, regular expressions were something new to me. Also, the concept of tfidf was somewhat difficult for me to understand, but after some revision and reading, I was able to figure it out.

Furthermore, due to my academic background, as opposed to IR, code efficiency was not something very important. During lab 2 and 3, my code was fairly slow, but after some modifications to my code, I was able to decrease the processing time by a huge ammount.

6. **Any ideas on how to improve and scale your implementation**

In this case, the collection contained 5000 documents so loading the inverted index into memory was not a problem. However, for large-scale applications, the collection size will be much higher, and loading the inverted index into memory would not be possible; in this case, it will be beneficial to store the inverted index on disk, and dinamically retrieve the information for the words present on the query. For scaling the implementation, the system can be uploaded to a website and the inverted index stored on a database.