Text Technologies for Data Science

RuiChen Gao, s1709006

October 30, 2021

1 Introduction

In this coursework, we were asked to implement basic search engine features such as preprocessing, indexing, boolean search and ranked information retrieval. For this report, I will detail the methods I used for each of the tasks with code examples and comments. Moreover, I will briefly comment on the system as a whole, and what have I learned from implementing the system, before discussing challenges encountered and how I can improve the system further for a conclusion.

2 Implementation

The system was implemented using my own machine, under Windows 10 environment with PyCharm 2021.2.2 IDE, code was developed using Python 3.9 and consists of four functions (booleanSearch, phraseSearch, proximitySearch, rankedRetrieval) and also four helper functions (tf, df, weight, score) to be used by the rankedRetrieval function. Everything else were written according to the task order under the same script as functions, including load collection files, preprocess files, build indexes, read queries, execute queries, and lastly generate output files.

2.1 Preprocessing

For preprocessing, collection file and queries were done separately, collection files were loaded at the start and applied preprocessing immediately, but queries were only loaded after indexing, and parts of preprocessing were done within the search functions. Despite the difference, they all contain the following steps:

1. Tokenization
2. Stop-word removal
3. Stemming
4. Casefolding

The entire preprocessing stage for the collection file was done using a one line of list comprehension code; in this way, everything in the collection will be converted to lower case then split by non-words and non-digits, before checking whether each split words is in the stop words list and apply stemming. “re” package was used and stemmer was “Porter Stemmer” from the nltk package, stop words were provided by the coursework specifications, which was the englishST. (The .xml collection file trec.5000.xml was parsed using xml.etree.ElementTree package before any processing could be applied.)

2.2 Indexing

I have used a nested dictionary of nested lists to implement the positional index, the first dictionary maps each unique term to the first list, which stores the document frequency at position 0 and the second dictionary at position 1, the second dictionary maps each document number to the second list, which stores list of indexes that term occurred in the corresponding document.

Preprocessed collection file was then processed using a loop counter i which accounts for each of

the documents in the collection to build the positional index. This positional index was then sorted in to correct order with simple sorting codes. Some example usage of positional index can be seen below for a better representation of the structure:

1. positionalIdx[term][0] will return the document frequency of corresponding term.
2. positionalIdx[term][1] will return a list of dictionaries that each maps the document number to a list of indexes.
3. positionalIdx[term][1][5000] will return a list of indexes, represents the positions of the term that has appeared in document 5000.

The way to build up this positional index was to check each token in each document, whether this token exist in the current positional index, if this is the first time it appears, then add it to the positional index and do corresponding initialisation; if this term appeared in the current positional index before then check whether this term appeared in this particular document before, if so, append to the document list, otherwise create a new mapping of the document and the term before increase the document count by one.

2.3 Boolean search

In order to perform boolean search, a collection matrix was built first using the positional index with the help of simple loops. The boolean search function takes a single query, preprocess it and checks for each term whether they are boolean operators "AND", "OR" or "NOT". If the term is a boolean operator then simply convert it in to boolean operator representation "&", "|" or "∼", otherwise find the row vector of the term in the collection matrix and convert it to a binary string.

After all terms in the single query has been converted, with the help of "eval" function, I can get a maximum-document-number-sized bit vector representing valid/invalid documents with 1 and 0, so if index 7 of the bit vector is an one, then it means that document number 7 should be returned by the query, vice versa. (Some processing to the bit vector also required such as filling in the offset gap with zeros since leading zeros were not returned by "eval".)

2.4 Phrase search

Phrase search takes a single query, preprocesses the query and check for the position of the " sign, since a phrase can come after a boolean operator and does not guarantee to be at the start of the query, which means " sign is the only way of identifying where do the phrases start. This was done by checking the terms in the query one by one, and if the phrase identifier was not found, then the term will be left as it is, but if the term begins with a phrase identifier, phrase search function will remove the phrase identifier for the word and the word after (since it is guaranteed that phrase search queries will only contain two words in the phrase at maximum, so no further checks were needed), also adding an "AND" word in between the two phrase words.

After all terms in the single query has been converted, phrase search runs boolean search and gathers a set of queries that fit the requirement. However, boolean search only performs a simple scan and does not preserve phrase requirements (two words being one after the other); therefore, a list of document ID and their indexes for the two phrase words were gathered from the positional index, then a nested loop (of each document and each index) was performed to check whether each document returned by the boolean search, does contain these two phrase words in the neighbouring positions one after one; early termination of the loop is used here to speed up the system, as we can return the document after we found a single match of the correct phrase word, rather than having to look through the entire document.

2.5 Proximity search

The way of implementing proximity search was very similar to the way I implemented the phrase search, as they only differs in some of the preprocessing decisions and phrase distance checks. For

preprocessing, the proximity serach identifer has changed from " to "#", and the phrase distance can be extracted by tracking the query characters that appeared after "#" and before "("; An "AND" term was put in between the two terms within the query as well.

Then the edited query was then provided to the boolean search function to gather a list of documents that suit the criterion. A check similar to the one used in phrase search was used to make sure the two proximity words were within the corresponding distance of each other. (Using $range(index - distance, index + distance + 1)$ rather than $index + 1$ for the two terms).

2.6 Ranked IR

The ranked retrieval function uses four helper functions that I have created (tf, df, weight, score) to calculate numerical values. I have also removed stop words from the query because it cannot be found in the documents. For each of the query term, I simply retrieved a list of documents that contain the term from the positional index, before joining the lists returned by each term and remove duplicate document numbers, this will give us a list of document numbers that contains at least one of the search term. Then for each of these documents, a score is calculated using the helper functions and appended in to a list of tuples, which stores the document number and its corresponding score, this list is returned after sorting the score in descending order.

2.7 Other

Query files were parsed with a parser written by myself, which uses the characteristics of query symbols (e.g. #) to determine which search to perform, and feed the queries one by one to the search functions and store results, before processes the result to a correct format and output them to the corresponding files.

3 Comments and challenges

1. Boolean search using bit vector and "eval" functions were new to me and very frequently used in the searches, I had to spend a lot of time to research and debug, sometimes the argument data structure are just invalid and the error messages were not very clear to me.

2. I have extended my theoretical knowledge on these searches and ranked IR to actual code writing experiences.

3. To build up an entire system from absolute nothing but text descriptions was new to me as other courses tends to provide skeletons; from setting up the environment, to write up labs and combine them to form this coursework, I have learned a lot of basic skills.

4 Future work

1. Positional index could be stored using a much more efficient format to save processing time, and functionalities such as stochasticity could be added. So when encountering newly unseen words, it can be added to the positional index directly, rather than having to re-run the whole positional index code.

2. Phrase search and proximity search functions can be generalised by some functions to determine the difference and when to apply which.

3. Preprocessing could be done with different rule sets, for some collections such as the internet words, we might not want to remove certain stop words or symbols.

4. This system can only process queries with certain format, an extension to the query parser would make this system more variable.