Corsework1

s2066387

October 25, 2020

1 Pre-processing

I implement tokenisation, stopword removing and stemming in order. For tokenisation, I use regex formula to get rid of all symbols except for words and numbers. I keep the numbers because the given XML file mainly contains information about finance and economics where numbers are of great importance. Meanwhile I delete all punctuation for simplicity. As for stemming, I use stem function from stemming.porter2 as required.

2 Inverted Index

I use cElementTree to parse the headlines and text from the XML to a list. And then store the inverted index into a dictionary of dictionary called term_dic. The key of the dictionary is the term after pre-processing. And the value of the dictionary is a dictionary with the document number as the key and the position which the term appears as the value. It may not be the fastest way for storing the positional inverted index, but it's easy to implement.

3 Search Functions

Since boolean search, phrase search and proximity search can be mixed with each other, I write a function called query_request() to deal with these query inputs as a whole. And I write two separate functions namely word_index() and phrase_index() to look for the document indexes containing either single term or multiple terms. Here phrase_index() can deal with both phrase search and proximity search with input variable "distance" equals 1 for phrase search and a given distance for proximity search. At the same time, a function called bool_search() deals with logical operators.

First of all, I process the query input using the same pre-processing method mentioned above. Two special cases are worth mentioning here. One is that the punctuation "#" is left in the query for positioning the proximity search. The other one is that logical operators "and", "or" and "not" are deleted from the stopwords collection.

Seven cases of boolean search can appear in the query: "A and B", "A or B", "A and not B", "A or not B", "not A and B", "not A or B" and "not A". With this in mind, I use three flags to represent whether the three logical operators appear in the query. If none of the three operators appears, then I determine by its length whether the query is proximity search, phrase search or single word search and operate the corresponding function to look for the document indexes. If at least one of the logical operator appears, I split the query into two lists accordingly without these operators. For each list, I use the same method as above to find the document indexes and then operate logical operators on them at the end.

The ranked IR based on TFIDF is implemented as a nested for-loop. It's intuitive to loop through each query term to find the document frequency(df). Then query_request() is applied on each term. The result as a list of document indexes is looped through to find the frequency of the term(tf) in each document. For each term in a given document we get a score by the TFIDF term weighting fomula and add them up to get the final ranking score for that document. Finally the documents with weight are ranked and limited to the top 150.

4 Commentary

The whole system takes around 1 min and 10s to complete. The organization of the programming is very clear are can be used in more sophisticated systems. Firstly we need to pre-process the collection of documents and apply the same method on the query inputs. And then the positional inverted indexes of the processed terms are stored in a dictionary of dictionary for easy assess of the search functions. For each type of the search functions, multiple cases are discussed and some useful functions are introduced. The result is not just related to the type of search function, but also related to how we pre-precess the given documents and how we store them. Some small mistakes may cause the error of the whole system. All these functions work as a whole so each of them should be coherent and concise. And special attention is paid for the consistency of document pre-processing and query pre-processing.

This coursework is just a rough implementation of simple query of IR but I've learnt a lot from it. The implementation of each part is not very complicated but we have to take efficiency and effectiveness into consideration. After many times of debugging, I have a clear idea of where the bugs are most likely to appear, which is a valuable experience for future work. Also, I've practiced the essential programming skills in Python and gotten to learn some knowledge about algorithms and data structures. When building the system, many advanced topics come to my mind: what if two or more "and" and "or" appears in the query? What if the proximity search contains more than two terms? How to measure whether the user likes the result? I find these topics really interesting and I'm eager to go deeper on them later on.

5 Challenges and Improvement

It has been a while since I learnt Python in school so I spent quit a lot of time on writing the functions in lab2. The biggest challenge is to take every case into consideration. Boolean search, proximity search and phrase search may be mixed with each other in a query. Even for boolean search itself, there are several kinds of combinations of logical operators. It's quite common to forget some of the cases. Besides, sometimes the program run smoothly, but the output is different from others. So I have to write many test functions to figure out where the difference comes from.

For further improvement and larger scale of documents, several aspects could be taken into consideration which are listed as follows:

· Some special cases of tokenisation may be discussed. For instance, hyphens can be broken up and removed without changing the meaning of the two words in most cases. However, in some special cases it may lead to completely different meaning after stemming: "co-operative" breaks up to "co" and "operative" and "co" is removed after stemming, resulting in the term becoming "operative" thus causing inaccurate in query search.

· There may be faster way to store the positional inverted index. Perhaps delta-encoding could have been useful because the document IDs are large but in order. There may also exist other data structures rather than dictionary of dictionary which could be easily assessed and require less time and space for storing such as binary trees. Perhaps I will take time to compare them later on.

· The operation time could be further reduced. This system takes about 1 min to operate which is acceptable. But for larger scale, may nested for-loops could take much time. Perhaps better ways may be found using more sophisticate algorithms.