Keyi Liu
s1703084

I.Introduction

This project aims to implement a basic Information retrieval tool (search engine) at document level, including text pre-processing, creating a positional inverted index, boolean searching, phrase searching, proximity searching and ranked IR based on TFIDF. The following includes description, challenges and criticism of my implementation.

II.Environment

This project is implemented on PyCharm 2020.2.2 (Community Edition), using the python interpreter of python 3.8 on macOS 10.15.7. Runtime version is 11.0.8+10-b944.31 x86_64 and the Virtual machine (VM) is OpenJDK 64-Bit Server VM by JetBrains s.r.o.

III.Explanation of Implementation

i. Tokenisation

This is the first step of text pre-processing. Importing the package ***string*** and ***string.punctuation*** is used to compare the text and headline in each document. If the contents of headlines and texts exist any punctuations (!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~), it should be replaced by space. Then all valid words form texts and headlines of each documents were split and stored into a list named *processing_list*. The format is [(word, doc number, position), ……].

ii. Stemming

Before stemming, all the words in *processing_list* were made into lower case in order to better match the query and all the stopping words in the list were filtered to increase searching speed. Importing the package ***stem*** from ***stemming.porter2*** is used to normalise every words by ***stem()*** in the list to reduce the vocabulary list.

iii. Positional Inverted Index

The basic idea is to turn the list to a nested dictionary named *inverted_positional_index*: in outer dictionary, key represents a unique word; in the inner dictionary, key represents the document number and the values present positions of occurrence of this word in this document. Using for loop to write each element of the *processing_list* by checking whether there is a word (outer key) existing in the outer dictionary firstly, then checking if there is a number of document (inner key) existing in the inner dictionary. The format of the nested dictionary is {word1:{documentNo.1: [position1, position2, …], document No.2: [position, …]}, word2:{….}, …..}.

iv. Boolean Query Extraction: *preprocess_Bquery(data_path, encoding='utf-8')*

To better deal with each query in searching process, extracting useful information from queries is very crucial to increase the speed of searching. The basic idea is to remove redundant punctuations such as parentheses, quotation marks but to keep the hash marks (#), then to separate rest elements in each query and store in the list named *Bquery_List*. For example, for query5 in the file is "5 #20(income, taxes)", it will be translated as ['5', '#', '20', 'income', 'taxes'] and stored in the list; for query8 in the file is "8 "Financial times" AND NOT BBC", it will be translated as ['8', 'Financial', 'times', 'AND', 'NOT', 'BBC'].

v. Phrase Search

Phrase search (***__phrase_Query__(arg)***) is the most important function in this project, it works as a fundamental method to support other searching functions. In my implementation, phrase searching includes single word searching and adjacent two words (phrase) searching. If the query in *Bquery_List* satisfy the condition that there is no '#', 'AND' and 'NOT', it will execute phrase search function. To delete the first element (query number) and normalise query before searching, it checks how many element left in the query. If there is only one element (word) left, phrase search function will get all the documents that belong to this word and return a list. Otherwise, it will get all the documents and return a set for each element (*term1* and *term2*) at first. Secondly, to find the common documents for two words by ***.intersection()***. Finally, it uses for loop to check every common document. In each document, if there exists a position of *term2* equal to one of positions of *term1* plus 1, then add the document number to a list and return it.

vi. Boolean Search

Boolean search includes two functions to satisfy the requirements, namely query searching consisting of 'AND' and query searching consisting of 'AND NOT'.

- To search query with 'AND' (***__AND__(arg)***), it deletes first element (query number) and removes 'AND' from the query at first. The terms can be single word or adjacent two words. For each term, it uses the function of ***__phrase_Query__(arg)*** to get the document list and turn it to a ***set()***. Then return the common document set of the two terms by using ***.intersection()*** and turn it to a list.
- To search query with 'AND NOT' ***(__ANDNOT__(arg))***, it deletes the first element (query number) and removes 'AND' and 'NOT' from the query at first. The first term is before 'AND' and the second term is after 'NOT'. Likewise, it uses the function of ***__phrase_Query__(arg)*** to deal with two terms and return document number list for each. Then it uses ***entire_doc.difference()*** to get the document number set without *term2* in it. Finally, it finds the common documents set for two terms by ***.intersection()*** and turn it to a list to return.

vii. Proximity Search

Proximity research ***(__Proximity__(arg))*** should be applied when there is a '#' in the query. Distance as a parameter in proximity search is in the second index. It creates two sub-lists to store two elements at first, then applies phrase research function of (***__phrase_Query__(arg))*** to get a list of relevant documents. Again, it uses ***.intersection()*** to obtain the common document numbers. For every common document, it uses for loop to check if the absolute value of any *position of term1 - position of term2* equals to or less than the distance. If it is, it stores this document number into a list.

viii. Ranked Query Extraction: *preprocess_Rquery(data_path, encoding='utf-8')*

This function aims to pre-process the ranked query in the file including removing the stopping words, removing punctuations and normalising by ***(normalise_query(arg))***. In addition, it is used to return a nested list consist of query lists. For example, the inner query list of query7 in the file is ['7', 'dow', 'jone', 'industri', 'averag', 'stock'].

ix. Ranked IR based on TFIDF

TFIDF term weighting is used in search engine to find the most relevant document (with highest scores) for a query, which means terms from the query appears more than any other files. Use parameters of term frequency and document frequency to calculate weight:

$$w_{t,d} = (1 + \log_{10} tf(t,d)) \times (\log_{10} \frac{N}{df(t)})$$

And the total score of the document is the sum of every term's weight from a query:

$$Score(q,d) = \sum_{t \in q \cap d} w_{t,d}$$

It obtains all the relevant documents for each term in the query at first. Then it uses ***set().union(*OR_doc)*** to get union documents for all the terms in a query. For each document in the union documents, it calculates the weight of each term that exists in that document. The total score is accumulated by checking through each document.

IV. What I learned

- Better understand different data structures such as list, dictionary, tuple, string and sets.
- Learned how to store the content of the file in any type of data structures.
- Learned how to transfer between different data structures and how to write different data structures into a *.txt* file.
- Gained basic knowledge on how to choose different data structure to fit different requirements.
- Improved the coding skills on how to write nested loop effectively.

V. Challenges

- Modulated functions in this project sometimes are quite useless since they are too large to reuse in different part.
- Had no idea how reduce the execution time, the execution time of this project is around 1 minute, but it is still too low to be a search engine.
- Refactoring the codes is the biggest challenge in this project.

VI. Criticism and Future work

- The implementation of this project is based on these 20 queries, so some other functions are deleted to fit these specific queries. It can't be used in different types of queries.
- More work needed to refactor the codes and to build more reusable functions.