Text Technologies For Data Science Assignment 1: Report

Introduction

This report is about the implementation of a basic IR system. This system is capable of parsing a given XML file and performing some basic preprocessing for the XML file. The preprocessing includes tokenisation, case folding, stop word removal and stemming (normalisation). In addition, it can take search queries including Boolean search, Phrase search, Proximity search and Ranked IR based on TFIDF. The results would be written as index.txt, results.boolean.txt and results.ranked.txt respectively.

XML File processing and Stopwords generation. (Initialisation)

The XML file was parsed into tree and root structure based on the xml.etree.ElementTree package. The function iterates through each root and get the relevant document number, document headline and the document text. Each data will be stored into the local variable (self.docnums, self.headlines, self.texts). The English stopwords are read once and was stored in a local variable(self.STOPWORDS) and the data type is a set. Because stopwords are fixed and set is efficient for searching.

Tokenisation and stemming methodology(preprocessing)

For tokenisation, it was decided that tokens would be parsed by splitting the text on each non-letter character, non-digit character which can preserve useful context when it comes to queries. Furthermore, tokens were converted to lowercase. The stemming method is porter2 stemmer. And the token will be stemmed if the token is not in the STOPWORDS set. The output is the list of cleaned terms.

Implementation of inverted index

The inverted index system was wrapped as a class called ir_tool. And the data structure of my inverted index was created in the following format

$$(Dictionary[String -> term] : [List[0] -> DocumentFrequency], [List[1] -> Dictionary[Int -> DocID] : [List -> Positions]])$$

The function of IR can be divided into three main steps. First, iterating the self.text file to get the body of the document, and then preprocess this token into list of terms. Getting the document number based on the index. Furthermore, it needs to process each single term in the list. If this term is in the IR dictionary, increment the document frequency of this term, and then check whether this term has existed in that DocID before. If existed, just adding the term position into this DocID dictionary, otherwise new a DocID dictionary and put the position into it. However, if the term is not in the IR dictionary, it needs to new a postings dictionary and append into the IR dictionary. Eventually, this IR dictionary is sorted according to the keys.

Search Functions

The search functions were wrapped as a class called Search Engine. The initialization of this Search Engine requires the ir_tool.docnums and ir_tool.pos_index. And then iterating the queries.boolean.txt file to initialize the self.queriesID which represent the id of the queries and self.queries which is the bodies of the quires. Because quires have different

kinds of style, so adding an extra function (Querydiversion) to divert the query. This function is used to check if the query contains AND, OR relationship, otherwise it is treated as a single term. If the query contains AND, OR relationship, then split this query into two separate terms. The purpose of this function is to split a complex query into two single terms. Furthermore, the terms would be processed by the Boolean search.

Boolean search (AND, OR, and NOT)

The Boolean search took a term as parameter and return a set of documents ID as the output. It firstly checks whether the term contains the NOT. When the term contains NOT, the final output will be the difference set of self.docnums and original output. Furthermore, the function based on the starting character in the term, to decide using which searches (Phrase search, Proximity search and singleTermSearch).

singleTermSearch

This function took a term and a Boolean check as the parameters, and it will return a set of documents IDs as the output. The Boolean check is used to check if the terms are preprocessed. If the term existed in the IR dictionary, it would return the set of documents IDs that have the term. Otherwise return an empty set.

Phrase search

This function took a term as parameter and return a set of documents number as the output. Because the phrase format is '(A B)', the phrase can be split into two terms (LeftTerm, RightTerm) by " ". Furthermore, LeftTerm and RightTerm are done preprocessing to get clean terms. Hence, it can be used as an input to SingleTermSearch function separately, then getting setL and setR. Because LeftTerm and RightTerm must in the same document and in order. The intersection set was used to get the overlap between setL and setR. According to the intersection set, iterate each docno to get the LeftTerm positions list, and check whether LeftTerm and RightTerm are in order. If a document contains LeftTerm and RightTerm, and LeftTerm and RightTerm are in order, adding this docno into the result. After iterating all the docs, return it as a set of documents number.

Proximity search

This function is quite similar to phrase search, except the distance between LeftTerm and RightTerm. In the Phrase search, the distance is 1. However, in the proximity search, the distance depends on #number. Based on this features, it conclude a important condition that is (indexL+distance >= indexR and indexL < indexR). The first Boolean is used to ensure distance satisfied the requirement; the second Boolean is used to ensure indexL is in front of indexR. Otherwise, it violates what we what.

Ranked IR based on TFIDF

This function is based on the formula. TF represent term frequency of this word in this document. DF represent documents frequency.

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

The function(queriesRanked) iterates through each query and assigns them a score for each document based on a scoring formula. The return is query and its relevant TFIDF scores. And based on the requirements, the top 150 documents scores would be written into a file called results.ranked.txt.

This System as a Whole

The whole system divided into two class: IR tool and search Engine. The IR tool is supposed to generate an IR system. And the search tool is supposed to do the Boolean search, Phrase search, Proximity search and Ranked IR based on TFIDF. Because this system contains such large amounts of data, and complex data structures. The design of the data structure and data IO would influence the efficiency of the system a lot. It's essential to refactor the whole system and data structures in the future. Currently, the index.txt can be written within 5s, the Boolean.txt file and queriesRanked.txt file can be written within 5s as well.

Challenges and improvements.

Implementing this system was very difficult at times and I found it easy to make errors and test results when dealing with such large volumes and creating large, complex data structures. Because IR's data structures are complex and error-prone, it requires care and patience to find errors and ensure that you get the expected values at each step. Otherwise, it is easy to fall into the trap. This made me realize the importance of regular testing at each stage of the system. Also, breakpoints help to check if the code is working as expected. In addition, it is necessary to check the function works properly and it considers all potential conditions, even fraudulent data and defective data. Another challenge is to improve the efficiency of the system. Because IR systems and search systems require a large amount of data, a reasonable and well-organized data structure will save a lot of processing time. Also, within the function, different methods will have different time complexity. There are several for loops that can be optimized to be more efficient. For example, finding the difference between two lists, it would be faster when you use the set instead of list. Also, if the lists are ordered, they can be optimized by some algorithm: quicksort. In addition, it is important to refactor the whole system to add more useful access functions to make the system readable and easy to implement.