Report – TTDS coursework 1

Methods used for tokenization and stemming

My tokenizer is a function that takes a string and returns a list of tokens. All the occurrences of non-letter characters in the input string have been changed into space by using Python String `maketrans()` and `translate()` methods. All the characters in the string are case folded into lower case. Then, I split the whole string into a list of tokens by using space as separator and return the list. Since the tokenizer did not have special treatment for symbols such as – or ‘ hence Hewlett-Packard will be treated as two separate tokens and Kiehl’s will also be treated as two separate tokens. However, this should not be a problem for information retrieval as the queries contained these words will be tokenised by same tokenizer as well. If we have special treatment for those symbols and the queries entered by users did not contain those symbols, it may cause some problem in information retrieval stage unless we spent more memories to index both cases.

My stemmer is a function that takes a list of tokens and returns a list of stemmed tokens. I used the PorterStemmer from the NLTK python library to reduce morphological variations of words to a common stem. For example, this list of tokens
`['connection', 'connected', 'connects', 'connecting']` will be return as `['connect', 'connect', 'connect', 'connect']`

The implementation of the inverted index

The collection that stored in xml format is parsed as a tree (by using the ElementTree from xml python library) so that my program does not need to read the file again every time it wants to update the inverted index. To include the headline of the article to the index, the document text is formed by append the text field right after the headline field. For each document, I tokenize the document text, remove stop words and stem the remaining tokens.

My inverted index is a nested python dictionary. It stores tokens as keys and inner python dictionaries as values. For each inner dictionary, document ID is stored as key and the list of positions that the tokens appeared is stored as value. The index of position is beginning with 1.

```
{ token1:
      { docNo1: [ pos1, pos2, … ] ,
        docNo2: [ pos1, pos2, … ] ,
        … } ,
  token2:
      { docNo1: [ pos1, pos2, … ] ,
        docNo2: [ pos1, pos2, … ] ,
        … } ,
  … }
```

The program will loop through all the stemmed token lists for each document. If the token is already present in the inverted index and the document ID is already present in the corresponding inner python dictionary, add the corresponding positions it appears. Else, create a new entry.

The implementation of the four search functions

1. Boolean search

   The query for Boolean search is tokenized, removed stop words, and stemmed. The list of documents that contained those stemmed tokens from query will be input to the Boolean search function. The AND Boolean search and OR Boolean search are the functions that take two list of document IDs as input and return a list of document ID. Whereas NOT Boolean search is a function that take one list of document IDs as input and return a list of document IDs. The input lists for these three functions are converted into python set and leave

the document IDs that satisfy certain conditions by using the Python Set methods. Since it is in the type of python set thus any duplicates of document IDs will be removed automatically. Then, the python set is converted into the python list before returning.

Boolean search function	Method of Python Set used in implementation	Condition to be satisfied
AND	`intersection( )`	The documents ID must appear in both input lists.
OR	`union( )`	The documents ID must appear in at least one of the input lists.
NOT	`difference( )`	All the document IDs of the collection except those appeared in the input list

2. Phrase search

   The query for phrase search is tokenized, removed stop words, and stemmed. The stemmed tokens from query are sent as input to the phrase search function. The phrase search function uses the AND Boolean search function to find out the list of document IDs that contained all the stemmed tokens. It then returns a list of document IDs where the positions of the stemmed tokens are consecutive, and the order of positions is consistent with the query.

3. Proximity search

   The query for proximity search is tokenized, removed stop words, and stemmed. The stemmed tokens from query are sent as input to the proximity search function. The proximity search function uses the AND Boolean search function to find out the list of document IDs that contained all the stemmed tokens. It then uses another function to check if the differences between the positions of stemmed tokens is equal to or less than the input distance. The proximity search function returns a list of document IDs where the positions of the stemmed tokens are close enough.

4. Ranked IR based on TFIDF

   The query for ranked IR search is tokenized, removed stop words, and stemmed. The stemmed tokens from query are used to calculate the retrieval score of the documents that contained those stemmed tokens. For each stemmed tokens, the program will loop through all the documents contained the tokens and calculate the weight of the tokens by using the corresponding term frequency (TF) of the document and the inverse document frequency (IDF). After that, the program sums over the weights of tokens for each document. The retrieval scores are rounded to four decimal places. The program then sorts a list of tuples (document ID, retrieval score) by the score from largest to smallest.

Reflection from the implementation of the system

Through implementing the system, I learned the importance of data structure. A suitable data structure consumes less memory and take advantages on regular operations. For example, the system has faster look up time by saving the document IDs in a dictionary. Instead of accessing the files every time we got a query and read through them lines by lines, applying the knowledge of text laws and having the information about the collection in a data structure improves the efficiency of the information retrieval to a very great extent. I also learned the amazing of algorithms. By having operations like binary search, the searching time can be further improved for the same operation on the same data structure.

The biggest challenge I faced is to get the data structure for the inverted list right, because it is not intuitive to come up with an answer. I searched various references in the lecture and across the web to compare different implementation.

During the implementation of the system, I chose to enumerate the parsing of the Boolean queries manually. This could be improved by writing a more robust Boolean query parser.