TTDS Coursework1

Pai Peng s2154448

1. Tokenisation and Stemming

To implement tokenisation operation, we can use the Python function '*re.sub()*' to replace all punctuations in strings with spaces and the function '*lower()*' to convert uppercase characters in words to lowercase characters, and then split all the strings into individual words with the function '*split()*'. In this way, we can store the words in each document as a list for later operations. Punctuations are not helpful in information retrieval and increase computation, so we remove all of them.

After tokenisation, we remove stopwords from each document list with 'englishST.txt'. As for stemming, we use the function '*stem()*' form the package stemming.porter2 to stem each word in every document list.

2. Inverted index

Here are some steps I take to get the inverted index.

1) After tokenisation and stemming, we store the word position with a dictionary called *word_position_dict.* In the dictionary, 'key' is the number of each document and 'value' is a list of words and their positions which looks like [[1, word1], [2, word2]…].
2) Then, we use *word_position_dict* to generate special words list called *specialwords_list*. Each of words in the list corresponds to every word of inverted index.
3) Finally, we create a dictionary called *term_position_dict* which will be used to output inverted index. We use each word in *specialwords_list* as the 'key' in the dictionary, and list[document number, position number] as the 'value'. Retrieve *word_position_dict* and add the position of each word in it to the corresponding list. In the end, we get the term-position dictionary which looks like {'Term1': [[doc1, pos1], [doc2, pos2], [doc3, pos3]], 'Term2': [[doc1, pos1], [doc2, pos2], [doc3, pos3]], …}. After getting the dictionary, all we need to do is to build a function for the desired output.

3. Search Functions

To implement the search function, I designed a number of small search functions and combined them together. We use two dictionaries, one is *term_position_dict*, and another is *term_doc_pos_dict* which looks like {'Term1': {doc1: [position1, position2], doc2: [position1, position2]}, 'Term2': {doc1: [position1, position2, position3], doc2: [position1, position2]}, …}

1) *word_search()*: Query for a single word and return its location. Using *term_doc_pos_dict* to find the value of the searching word. The value is also a dictionary, and the key extracted from the value is the location returned.
2) *link_search()*: Query for a phrase and return its location. Here are two words in the

phrase. We use *term_position_dict* to find document number and position number of each word. Then, we compare the position of two words. If the two words are in the same document and the difference between the positions in the document is 1, return the document position.

3) *not_search()*: Returns a list where the search word does not exist. The function is just opposite to *word_search()*. After we get the search word list, return the list of the total document number minus the word list searched.
4) *distance_search()*: Query for two words whose distance is a constant. It is similar to *link_search()*. If the two words are in the same document and the difference between the positions in the document is the constant, return the document position.
5) *and_search()*: Query for two terms in the same document, return the list of unions. If 'NOT' is in the term, we invoke *not_search()*. If two words are in the term, we invoke *link_search()*. If only a single word is in the term, we invoke *word_search()*.
6) *or_search()*: Query for two terms, return the list of intersections. If 'NOT' is in the term, we invoke *not_search()*. If two words are in the term, we invoke *link_search()*. If only a single word is in the term, we invoke *word_search()*.
7) *search()*: It is the main search function. Determine which search function to use based on the input criteria. If 'AND' is in the query, we invoke *and_search()*. If 'OR' is in the query, we invoke *or_search()*. If '#' is in the query, we invoke *distance_search()*. If 'NOT' is in the query, we invoke *not_search()*. We decide whether to use *word_search()* or *link_search()* according to the number of words

4. Commentary and Challenges

1) As far as I am concerned, it is very difficult to implement Search Functions. This is because the input query is varied, and we need to decide which function to call based on the condition. To make the search function more flexible with different queries, I designed a number of small functions.
2) Besides, at the beginning of the rank part, I felt confused about how to translate complex mathematical formulas into codes. To solve this problem, I found a lot of useful functions to help me.
3) At the beginning of writing the program, I did not have a clear idea, so I was always troubled by the problem of how to structure the program. After that, I tried to draw a flow chart of program design, and added many details in each link, and the problem was finally solved.

5. Improvement in Implementation

1) Improvement in inverted index: When processing 1000 documents, it took 9 minutes to generate *term_position_dict*. It is too long for me as it will take about 45 minutes to process 5000 documents in the coursework1. Therefore, I was looking where I can optimize. Luckily, I found it. To generate *term_position_dict*, I first traversed *specialwords_list* and then *word_position_dict*. The time complexity in this case is O(N×M). To reduce the calculation time, I modified the code to iterate over *word_position_dict* once, adding the positions of the iterated words to *term_position_dict*, which reduced the time complexity from O(N×M) to

O(N). In the end, it only took 2 minutes to finish the indexing work.

2) Improvement in search functions: In processing Boolean characters, my algorithm is very clumsy and can only handle a single Boolean character. In future work, I will use stacks to handle incoming queries, with one stack storing terms and the other Boolean characters.