I. Tokenization

For this part, I made tokenization before indexing in the function $preprocess\_file$. Here I used regular expression to match particular characters and substitute these characters with a space. Regular expression pattern is

$$r'[\^A-Za-z\backslash s]'$$

which means that I only remained characters in English alphabet and spaces. Here are my reasons:

1> The remains are enough to represent meanings of an article for this task. Although I lost some characters, like "-", there are few influences on the result that I will get. For example, "copper-bottomed" doesn't lose much information when transformed into "copper" and "bottomed".

2> In most cases, uses are usually lazy to type in some characters, such as "-" and" 's".

3> In this task, given the test queries, digits are not important information in retrieval. Ignoring these digits won't lose much information of an article.

4> Space is used in text splitting.

II. Stemming

After tokenization, I used $SnowballStemmer$ from $nltk.stem.snowball$ as the tool for stemming. Because $SnowballStemmer$ is multi-language, I chose English language for stemming each $word$ in the list, $textsplit$, that was generated from tokenization. By applying

$$SnowballStemmer('english').stem(word)\ for\ word\ in\ textsplit$$

I got a list called $textstemmer$ which stored all stemmers of words in tokenization list.

III. Inverted index

To generate inverted index, I need to get an index for documents showing which term appears and where the term appears. Therefore, I designed a list, $term\_position\_list$, to store these information. Every element in this list is dictionary called $term\_position\_dict$ which contains a document ID, terms that appears in this document and positions that the term appears. Each dictionary corresponds to a document by storing its docID. Every key in each dictionary is a list because one term can appear in many positions. The list structure is

$$term\_position\_list = [term\_position\_dict, term\_position\_dict, \dots]$$

and the structure of each $term\_position\_dict$ is

$$term\_position\_dict = \{docID: [docID], term: [pos1, pos2 \dots], \dots\}$$

where the first key is docID and the first value is docID in a list format.

I used dictionary to represent document because:

1> dictionary can store docIDs as keys and positions as values

2> dictionary is convenient to find a particular docID and corresponding positions while a list can't do that

They are implemented in the function $preprocess\_file$ in $indexing.py$ file. Besides that, I also get other things with the function $preprocess\_file$ for convenience afterwards. They are

1> $stword\_set$: a set to store stop-words in English

2> $term\_set$: a set to store different terms that appeared

3> $phrase\_position\_dict$: a dictionary to store bigram phrases and the docID where phrases appeared. It has such structure

$$phrase\_position\_dict = \{phrase: [docID, docID, \dots], \dots\}$$

After that, I write the inverted index into $index.txt$ based on $term\_position\_list$ and $term\_set$. For each term in $term\_set$ and for each $term\_position\_dict$ in $term\_position\_list$, if this term appeared in this docID, then I stored this docID and get values whose the key is this term in this $term\_position\_dict$. The value is exactly the position where this term appeared in this document. Here I used

$$if\ term\ in\ term\_position.keys()$$

to decide whether this term appeared in this document.

Through this method, I got the term, the document IDs where the term appeared and positions that the term appeared in a given document. After that, all I need is to write these information into a file in the given format and then the inverted index written in the $index.txt$ file.

IV. Implementing Search Module

First of all, I need to load my index from $index.txt$ file. Here I used a list called $index\_list\_term$ to store a list of dictionaries. Each dictionary, $term\_index\_dict$, corresponds to a term storing its docID and positions. The structure of the list is

$$index\_list\_term = [term\_index\_dict, term\_index\_dict, \dots]$$

and structure of dictionary is

$$term\_index\_dict = \{term: [term], docID: [pos, pos, \dots], \dots\}$$

where the first key is this term and the first value is this term in a list format. So, $index\_list\_term$ is the index that I want. I can also, in the same way, load $phrase\_index\_dict$ which is the index of phrase.

For each line in $queries.boolean.txt$ and $queries.ranked.txt$, I used regular expression pattern

$$pattern\_query\_ID\_Question = '(\backslash d+)\ (.+)'$$

to split queryID and query. Then query is processed the same way as $trec.5000.txt$ is processed.

i. Boolean Retrieval: uses $queries.boolean.txt$

Regular expression patterns are also used to distinguish different kinds of queries:

1> $pattern\_AND = '(.+)AND(.+)'$ is to decide to handle AND operator.

2> $pattern\_PHRASE = '"(.+)\ (.+)"'$ is to use phrase search.

3> $pattern\_PROX = '\#(\backslash d+)\backslash((.+),(.+)\backslash)'$ is to use proximity search

4> $pattern\_NOT = 'NOT\ (.+)'$ is to decide if there is NOT in queries

5> If it doesn't match all, it is a single word

In condition 1, I used $q1 = query.group(1)$ and $q2 = query.group(2)$ to store 2 parts of the query and I put them into a list, AND_query_list. I also make a list, $result$, to store the results that $q1$ and $q2$ might return. For each q in AND_query_list, I need to see whether q is phrase after processing q. If so, q is put into condition 2. If not, I need to see whether q has NOT. If so, q is put into

condition 4. If not, q is a single word and put into condition 5.

In searching, I used set to store document ID. Find the index $index\_list\_term$, get the dictionary, $term\_index\_dict$, of corresponding q, fetch all keys excluding the first key and add them into set. If q is a phrase, the index should be $phrase\_index\_dict$.

If the set is generated by a NOT condition, the list $result$ should append "-1" before append this set, indicating the set after "-1" should be subtracted. In other times, the list $result$ should append "1" before append this set, indicating the set after "1" should be intersected.

Apart from being used in condition 1, condition 2, 4 and 5 can also be used directly.

As for condition 3, I used $q1 = query.group(1)$ as distance, $q2 = query.group(2)$ and $q3 = query.group(3)$ as two parts of query text. Then I searched the two parts separately and got the common set of docIDs by $intersection$ of two sets. Then I went to index again and get the two terms position in the common document and compute the distance.

Finally, I got a set for each query and wrote them into $results.boolean.txt$ file in a given format.

ii. Ranked Retrieval: uses $queries.ranked.txt$

To achieve ranked retrieval, I need to start from computing tf and df.

I used a list, tf_list, to store all terms frequency in all documents. Each element in tf_list is a dictionary, tf_dict, storing a particular term's frequency in every document. The first key of every tf_dict is a particular term and the first value of every tf_dict is set as 0. The structures of tf_list and tf_dict are

$$tf\_list = [tf\_dict, tf\_dict, \dots]$$
$$tf\_dict = \{term: 0, docID: tf\_value, \dots\}$$

where tf_value can be computed by the length of value in $term\_index\_dict$

As for df, I used a dictionary, df_dict, to store df for every term. The structure of df_dict is

$$df\_dict = \{term: df\_value, term: df\_value, \dots\}$$

where df_value can be computed by the length of keys for each term in $term\_index\_dict$

Then based on the equation

$$idf = \log_{10}(\frac{N}{df})$$

where N is the number of documents, I can get idf_dict, a dictionary to store the value of idf.

Based on the equation

$$weight = (1 + \log_{10} tf) * idf$$

I can get the weight for every document when given a term. I used list $weight\_list$ to store each dictionary $weight\_dict$ storing every term's weight. The structure of $weight\_dict$ and $weight\_list$ are

$$weight\_list = [weight\_dict, weight\_dict, \dots]$$
$$weightdict = \{term: 0, docID: weight, docID: weight, \dots\}$$

where the first key is a particular term and the first value is set as 0.

Finally, after being processed, query is transformed into a list of terms and we can sum up these terms' weight in each document and set this sum as $score$ which is used to sort the order. After doing this, I can write results into $results.ranked.txt$ file.

V. Comments on System as a whole

1> My system is designed specifically for this task. In reality, boolean retrieval should contain more situations to deal with more complicated searching. So there are many limitations on my system.

2> For convenience, I use many dictionaries to store information. Maybe it's more efficient to store in vectors.

VI. What I learnt from Implementing it

1> How to process files

2> More deep understanding of concepts

VII. the challenges you faced

1>There are too many possible situations in boolean retrieval. Query can become very complicated when combining "AND","OR","NOT" and phrase and so on. It's quite hard to fully deal with such queries.

VIII. some ideas on how to improve it and scale it.

1> Consider more complicated indexes, like trigrams. Index them and maybe it is very useful.

2> When processing query, we can split it into bigrams or trigrams as well as terms. Then we can use it to compute new score and combines it with the existing score to rank documents.

3> Distinguish headline and text will generate better results.