According to the requirements of this coursework, I split my implementation into mainly four parts:

1. **Pre-processing**
2. **Inverted index**
3. **Boolean search, phrase search and proximity search**
4. **Ranked IR based on TFIDF**

1. Pre-processing

(1) Tokenisation and Case Folding

Aim: convert text into tokens with no punctuations and make all text into lower case.

Method: I imported re module and used regular expression to remove all punctuations. The punctuation removal set I set is '[!@#$%^&*()_+{}|:"<>?,./;\'[\]\-=]+' which means I removed all the punctuations. Then I used str.lower() to make all text into lower case and str.split() to split text into tokens.

(2) Stopping

Aim: remove the specified English stop words which are included in a file.

Method: After reading the stop words document, convert all stop words into tokens and store them in a list. Then use the list comprehension to record all the elements that are in the original text list but not in the stop words.

(3) Stemming

Aim: Normalisation

Method: I got Porter stemmer from the 'stemming.porter2' and used list comprehension to stem every tokens which were after the stopping operation.

2. Inverted Index

(1) XML parser

Aim: Extract the docID of each document and its headline and text fields.

Implementation: The XML parser used was coded by myself. I used regular expression "<DOCNO>(\d+)</DOCNO>.*?<HEADLINE>\n(.+?)\n</HEADLINE>.*?<TEXT>\n(.*?)</TEXT>
" to extract the docID of each document and its headline and text fields. And also used regular expression to convert escape characters in XML back to individual symbols.

(2) Creates a positional inverted index

Implementation:

a) First, pre-processes the extracted headline and text fields and merge the pre-processed tokens list.
b) I created two lists. One list 'file_tokens_all' recorded all tokens and performs deduplication operations using set() to get a non-repeating set. Each element in the other list 'file_info' is a two-tuple. For the two-tuple, the first element is the docID, and the second element is the pre-processed tokens list (the above (a) step) corresponding to the docID.
c) Then, to get the following information and record them:
   - term (pre-processed)
   - list of documents where this term occured
   - for each document, list of positions where the term occurred within the document
d) To achieve the goals, for the inverted index, the structure I constructed is dictionary. The key in the dictionary is each term, and the corresponding value is a list of two-tuples. In the list, the first element is the extracted docID, and the second element is a list to record each position of the term

in the docID document.

The following is the specific implementation:

Traversed each term in the non-repeating set ‘file_tokens_all’: for each term, traversed the second element of each two-tuple in ‘file_info’ (the tokens list corresponding to a docID), and used a loop traversal to record each term’s position in the document where it exists to form a list, and then record each docid and its corresponding position list to form a two-tuple. After traversing all documents for a term, it will form a list containing many two-tuples. Then, as mentioned at the beginning, insert the term key into the dictionary, and its corresponding value is a list of 2-tuples formed.

In this way, a dictionary of inverted index can be formed. The structure is like below:

{term_a: [(docid_1, [position_1,position_2]), (docid3, [postion_2, position_6])], term_2: [(docid_2,[position2])],...}

(3) Output inverted index to a text file

At this time, I only need to traverse the dictionary, and then for each term, calculate the length of the two-tuple list (to get the document frequency), traverse each of its two-tuples (to get the docID corresponding to a specified term), and traverse the position list for each docID (to get the positions where the term occurred within the document), that is, all required information can be output to a specified text file.

3. Boolean Search, Phrase Search and Proximity Search

(1) Extract queries from file

Since there are many types of queries, I need the system to automatically recognize which type of query statement belongs to to call related functions. I used regular expressions with if else statements to identify types of query statements.

The types included:

term1 Op term2, “term1 term2” Op term3, “term1 term2” Op “term3 term4”, #8(term1, term2), etc

(2) Boolean search

In this format “Part_1 Operator Part_2”

Here ‘part_1’ or ‘part_2’ can be either term or phrase.

Operator can be ‘AND’ ‘OR’ ‘AND NOT’ and ‘OR NOT’.

Use the regular expression again in the related function to obtain ‘part_1’, operator and ‘part_2’, and I applied pre-processing to each part.

First of all, I need to get the ‘part_1_docid_list’ and ‘part_2_docid_list’ which record the docid matched by the corresponding part. When either ‘part_1’ or ‘part_2’ refers to phrase, I can get the docid list by calling Phrase Search function, and when either ‘part_1’ or ‘part_2’ refers to a single term, I can simply get the docid list from the inverted index I created using loop traversal.

Then perform an if conditional statement on the operator, as follows:

There are four forms of Operator:

a) if operator =='AND':

Use the list comprehension to traverse each element of ‘part_1_docid_list’. If the element also exists in ‘part_2_docid_list’, record the element. The final list is the final result.

b) if operator =='OR':

Add the list of ‘part_1_docid_list’ and ‘part_2_docid_list’, and then use the set() function to process the result, and finally get the final result of sorting and deduplication.

c) if operator =='AND NOT':

Use list comprehension to traverse each element of ‘part_1_docid_list’. If the element does not exist in ‘part_2_docid_list’, record the element. The final list is the final result.

d) if operator =='OR NOT':

First create a list all_docid to record the docid of each document in the XML file.

Then traverse each element in all_docid, if the element does not exist in ‘part_2_docid_list’, record the element, use the list comprehension to perform the above process, and finally get the list ‘not_part_2_docid_list’.

Then execute ‘part_1_docid_list’ OR ‘not_part_2_docid_list’ and use the OR operation in 2 to get the final result.

(3) Phrase search

First, the regular expression '(\w+)\s"(\w+)\s(\w+)"' extract the query format 5 “term1 term2” to get the serial number and term1 and term2. And I applied pre-processing to each term.

First, get the docid and the corresponding position list of term1 and term2 from the inverted index.

Then for each term, I store the information I get in the following structure for each term.

[(docid,[positon1,position2]), (....)]

Traverse the list of term1, for each tuple corresponding to term1, traverse the list of term2, for each docid document containing both term1 and term2, traverse the position list of term1 and term2, when position(term2)-position(term1)== 1, add the docid to ‘list_of_docid’.

(4) Proximity search

First, use regular expression "(\w+)\s#(\d+)\((\w+),\s?(\w+)\)" to extract the query format 6 #8(term1, term2) of Proximity search.

Get serial number, proximity number, term1 and term2 through regular expression, and I applied pre-processing to each term.

First obtain the docid and the corresponding position list for term1 and term2 from the inverted index. The structure is as belows:

[(docid,[positon1,position2]), (....)]

Traverse the list of term1, for each tuple corresponding to term1, traverse the list of term2, for each docid document containing both term1 and term2, traverse the position list of term1 and term2, when |position(term1)–position(term2)| <= proximity number, add the docid to ‘list_of_docid’.

4. Ranked IR based on TFIDF

(1) TFIDF dictionary

First create a TFIDF dictionary to store the TFIDF value of each term in a specified document where it occurs.

Since I already have an inverted index, I can extract the docid list of each term from the inverted index to calculate the IDF. And get the term frequency in a document (because the inverted index have the positions of a term in a specified document).

To record TFIDF, I created a dictionary: the key is term, and the value corresponding to the key term is another dictionary. The TFIDF dictionary structure is as follows:

{term1:{docid1:w(term1,docid1), docid2:w(term1,docid2)}, term2:{docid1:w(term2,docid1), docid2:w(term2,docid2)}....}

(2) Ranked IR

After establishing the TFIDF dictionary, apply Ranked IR.

When get a query sentence, I apply pre-processing to it to get a token list.

I first get a list of docid of each term, then add all the docids to a list, and then use the set() function to remove duplicate elements. At this time, I get a list 'doc_result'.
Next, I traverse each docid in 'doc_result'. For each docid, directly extract the TFIDF value w(term, docid) of each term for the docid document from the TFIDF dictionary, and then add up all the extracted TFIDF values to a variable 'docid_score'.
Then add (docid, docid_score) to the empty list 'docid_score_list'. After finishing the 'doc_result' traverse, I can get the list storing all the rank results of the query:
[(docid1, docid1_score), (docid2, docid2_score).. ..]

5. A brief commentary on the system as a whole and what you learned from implementing it

(1) A brief commentary

The system preprocesses documents, including: Tokenisation, Stopwords removal and Porter stemming. The text includes headline and text. The system uses the same pre-processing for the documents and all query statements.
In terms of query, the system supports boolean search, phrase search and proximity search. You only need to write all the queries into a file. The system can automatically identify different types of query statements, and finally output all the query results.
At the same time, the system supports Rank IR based on TFIDF. You can write all the terms you want to query into the query document, and the system will perform all the query results according to the results of the rank value from large to small. Ranked results file list only up to the top 150 results per query.

(2) What I learned from implementing it

a) I enhanced the python programming ability, and strengthened the ability of text recognition and acquisition through using regular expressions.
b) I have a preliminary understanding of the basic principles and basic structure of search engines, which laid a solid foundation for my further learning.

6. What challenges you faced when implementing it

(1) First of all, when pre-processing the text, the extraction of XML or TXT file content and the removal of punctuation are not very familiar for me. Later, I learned the application of regular expressions and solved the above problems.
(2) When processing the automatic query of each query statement for the queries text file, for the automatic identification of different types of query statements, at the beginning, I did not know how to implement it. Later, combining the re module with regular expressions and if-else statements I realized the recognition of different types of query statements, so that different function calls were made to different types of queries.

7. Any ideas on how to improve and scale your implementation

Certain optimizations can be made in the running time of the program, because almost all the code is written myself, and I didn't use many existing module functions, including the extraction of XML file content. There are some modules that may reduce a certain amount of running time when implementing the same function. Secondly, in the creation of my inverted index, I used nested structure, but due to the use of some loop nesting, the running time increased, perhaps by changing the created data structure and optimizing the algorithm, such as using hash algorithm could reduce running time.