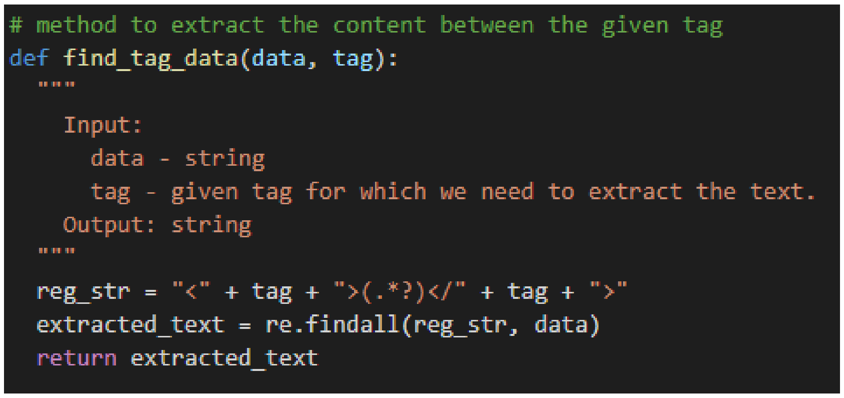
CSE508 Information Retrieval
Assignment 1

Question 1: Data Preprocessing

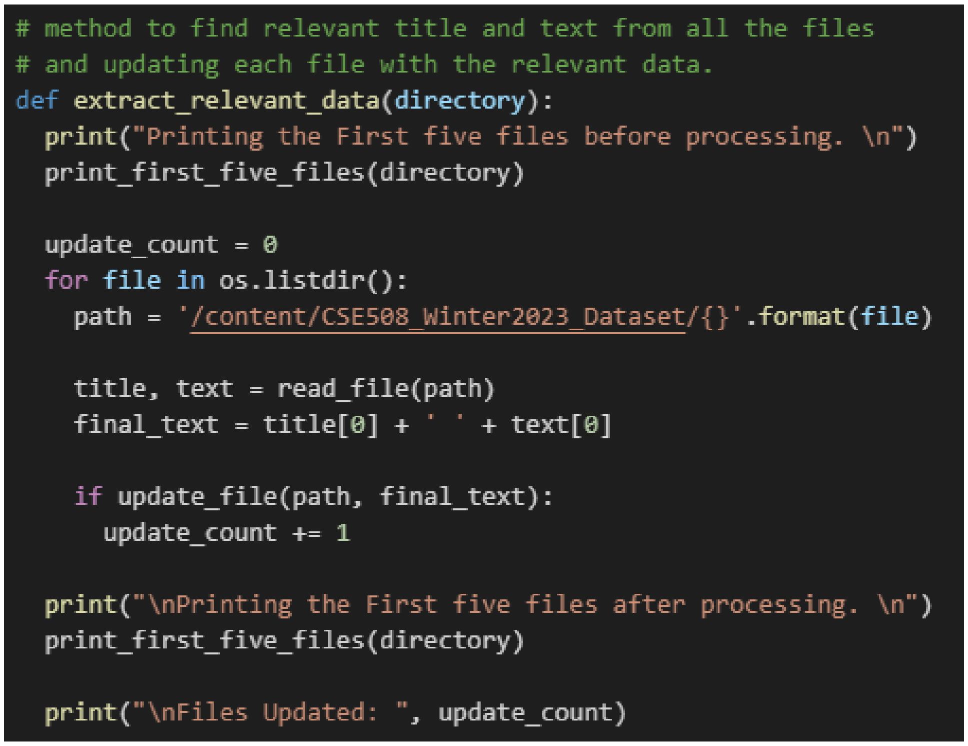
(i) Relevant Text Extraction

For this part, we have created a method that takes two parameters, the first one is content, and the second one is the tag (which we have to extract the text). This method will return the extracted content. We are using regex for this task.



```
# method to extract the content between the given tag
def find_tag_data(data, tag):
  """
    Input:
      data - string
      tag - given tag for which we need to extract the text.
    Output: string
  """
  reg_str = "<" + tag + ">(.*?)</" + tag + ">"
  extracted_text = re.findall(reg_str, data)
  return extracted_text
```

We are iterating over all the files, one by one, extracting the content between "TEXT" and "TITLE" tags, then concatenating them with a space and overriding the same file.



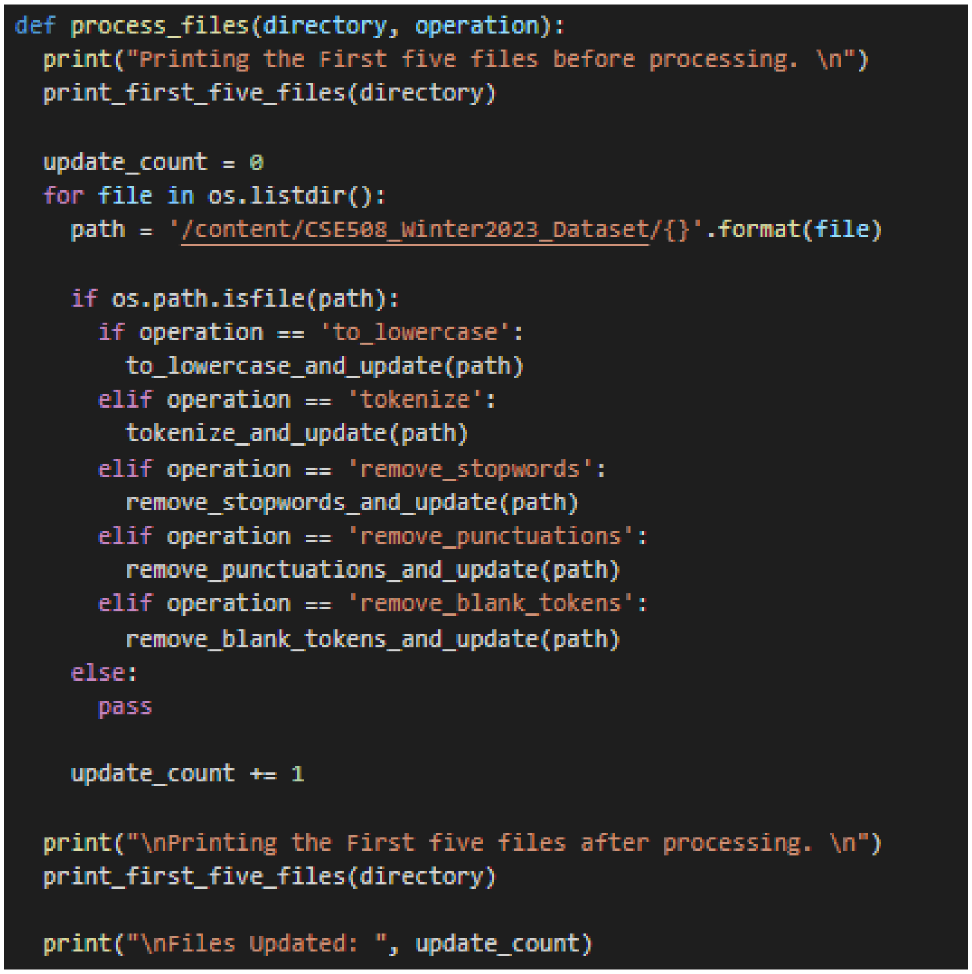
```
# method to find relevant title and text from all the files
# and updating each file with the relevant data.
def extract_relevant_data(directory):
  print("Printing the First five files before processing. \n")
  print_first_five_files(directory)

  update_count = 0
  for file in os.listdir():
    path = '/content/CSE508_Winter2023_Dataset/{}'.format(file)

    title, text = read_file(path)
    final_text = title[0] + ' ' + text[0]

    if update_file(path, final_text):
      update_count += 1

  print("\nPrinting the First five files after processing. \n")
  print_first_five_files(directory)

  print("\nFiles Updated: ", update_count)
```

(ii) Preprocessing

For this part, we have created individual modules for each of the given operations. Then a main method, which takes two parameters, the directory path, and the operation which we want to perform on the content of the directory.

For tokenization and stopwords, we have used the nltk library, and for other operations, we use regex and simple python.

In each of the methods, we send the path of the file, then the method reads the content and updates it according to its operation, then overrides the same file with the new content.



```
def process_files(directory, operation):
  print("Printing the First five files before processing. \n")
  print_first_five_files(directory)

  update_count = 0
  for file in os.listdir():
    path = '/content/CSE508_Winter2023_Dataset/{}'.format(file)

    if os.path.isfile(path):
      if operation == 'to_lowercase':
        to_lowercase_and_update(path)
      elif operation == 'tokenize':
        tokenize_and_update(path)
      elif operation == 'remove_stopwords':
        remove_stopwords_and_update(path)
      elif operation == 'remove_punctuations':
        remove_punctuations_and_update(path)
      elif operation == 'remove_blank_tokens':
        remove_blank_tokens_and_update(path)
    else:
      pass

    update_count += 1

  print("\nPrinting the First five files after processing. \n")
  print_first_five_files(directory)

  print("\nFiles Updated: ", update_count)
```

Assumption: We have the word **"wing-aileron"** after processing. It is becoming **"wingaileron"**, but it should not be changed. We are getting this result because it is taking the '-' as punctuation and removing it. So we are taking this forward and building our indexes based on this.

Question 2: Boolean Queries

Step - 1: Extract the Processed Files to load the files in the Master_Data Folder as a Destination folder. - The code utilizes the pre-processed dataset from Que-1. The files from Que-1 are retrieved by unzipping the files from the Pre-Processed Dataset. The Dataset is loaded in a new directory in the current working directory in Drive with the name as Master_Data.

Step - 2: Get the list of paths of files in file_paths_list and the name of all files in file_name_list. - To check the total number of files and maintain a list of file-paths so as to get each word from the given file and append it to the inverted index with the specific ID

Step - 3: Check the total no. of files in the given document and sort the lists so as when the lists are iterated over, the DOC-ID Postings are sorted, but as we use a SET to store all unique DOC-Postings, there is a need to sort the Postings again before executing the Query Operations.

Step - 4: Create a dictionary to add DOC-ID and DOC-NAME for each file to identify the file used for the query and then retrieve the corresponding DOC-NAME and DOC-ID for specified output format.

Step- 5: Creating a Unigram Inverted Index by iterating over a token of words and appending the DOC-ID corresponding to each word. The desired text for each file is available in the list of file_content, and hence we tokenize each file by iterating over file_content.

Step - 6: The Unigram-Inverted Index consists of all unique words as the Key and the corresponding DOC-IDs; if the word exists in a DOC, the DOC-ID is set as the Value.

- The Key-Value Pair consists of the first index of the value at '0' as the length of the DOC-IDs corresponding to the number of elements in the set.
- The next index has the set of all DOC-IDs, where- ever a particular word exists. The data structure being used to store all of the DOC-IDs as a SET will keep only unique DOC-IDs, i.e., if a given the word already exists as a KEY and the word is being read at multiple positions in the same document, the DOC-ID is not added.

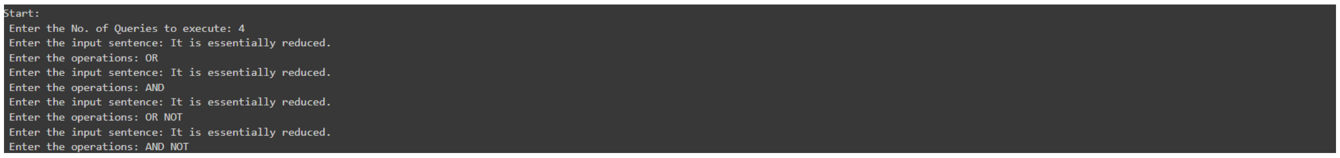
Step - 7: Create Pickle Files - Unigram-Postings.pickle is the pickle file which consists of the Unigram Inverted Index. DOC-ID-to-DOC-NAME.pickle is the pickle file which stores the DOC-ID and the corresponding DOC-NAME.

Step - 8: The helper function get_ans is used to process the sequence of operations by updating the list of values for each successive operation . A temporary list is used to store the output of the executed query on the given two lists, and the rest of the posting lists for the words extracted from the pre-processed query are processed with the consecutive result at each step. The opt_count variable is used to store and update the count of operations required to resolve the entire query.

Step - 9 :- The helper functions - pre-process() and makeQuery() are used to preprocess the input and complete the query for display. makeQuery() is used to join the operators with the pre-processed keywords and hence form the complete query as in the Output Format.

Step - 10: The begin() function allows the user to begin the program and enter the queries to get the answer and then get the final answer.

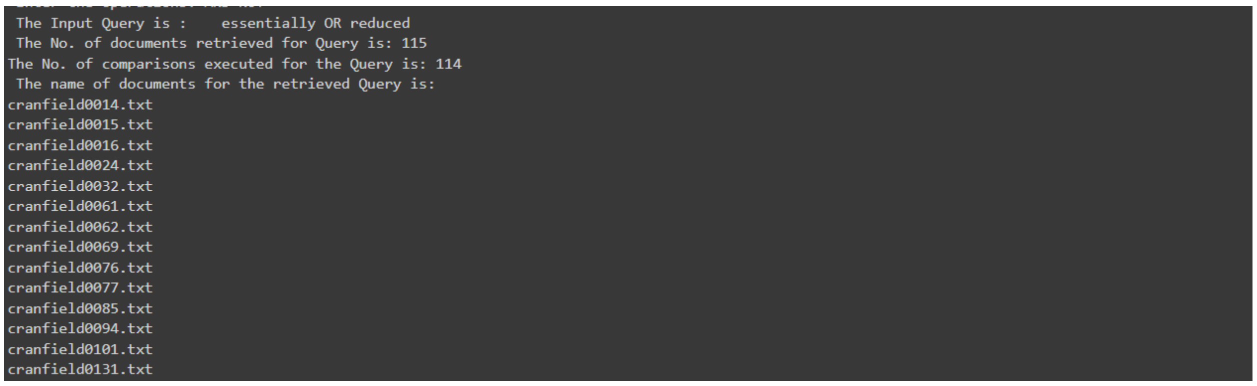
INPUT -



```
Start:
 Enter the No. of Queries to execute: 4
 Enter the input sentence: It is essentially reduced.
 Enter the operations: OR
 Enter the input sentence: It is essentially reduced.
 Enter the operations: AND
 Enter the input sentence: It is essentially reduced.
 Enter the operations: OR NOT
 Enter the input sentence: It is essentially reduced.
 Enter the operations: AND NOT
```

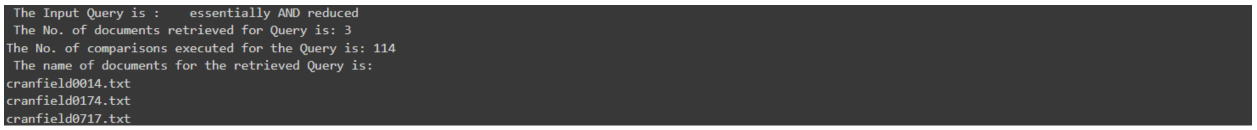
OUTPUT-

For Query -1 : -



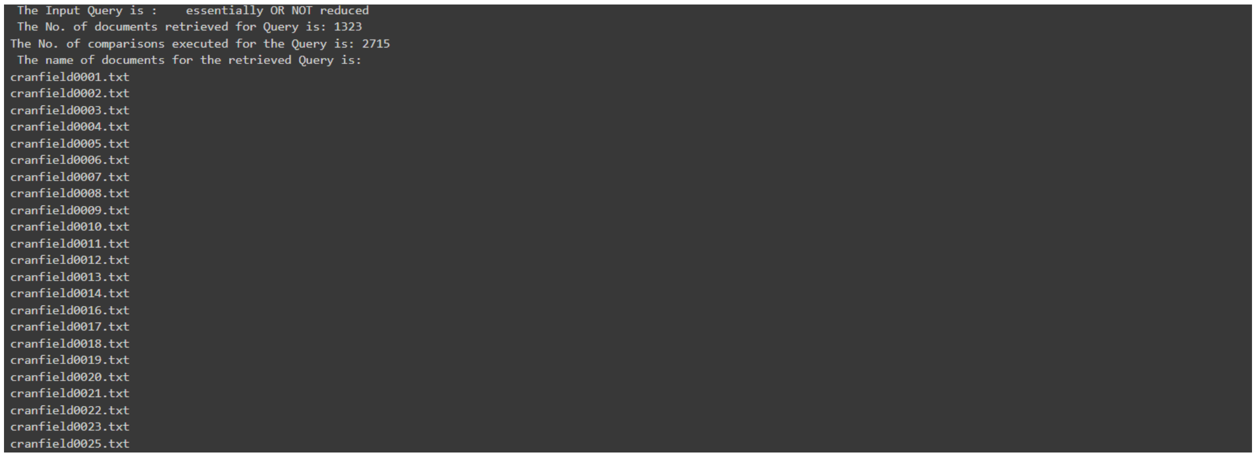
```
 The Input Query is :    essentially OR reduced
 The No. of documents retrieved for Query is: 115
The No. of comparisons executed for the Query is: 114
 The name of documents for the retrieved Query is:
cranfield0014.txt
cranfield0015.txt
cranfield0016.txt
cranfield0024.txt
cranfield0032.txt
cranfield0061.txt
cranfield0062.txt
cranfield0069.txt
cranfield0076.txt
cranfield0077.txt
cranfield0085.txt
cranfield0094.txt
cranfield0101.txt
cranfield0131.txt
```

For Query-2 : -



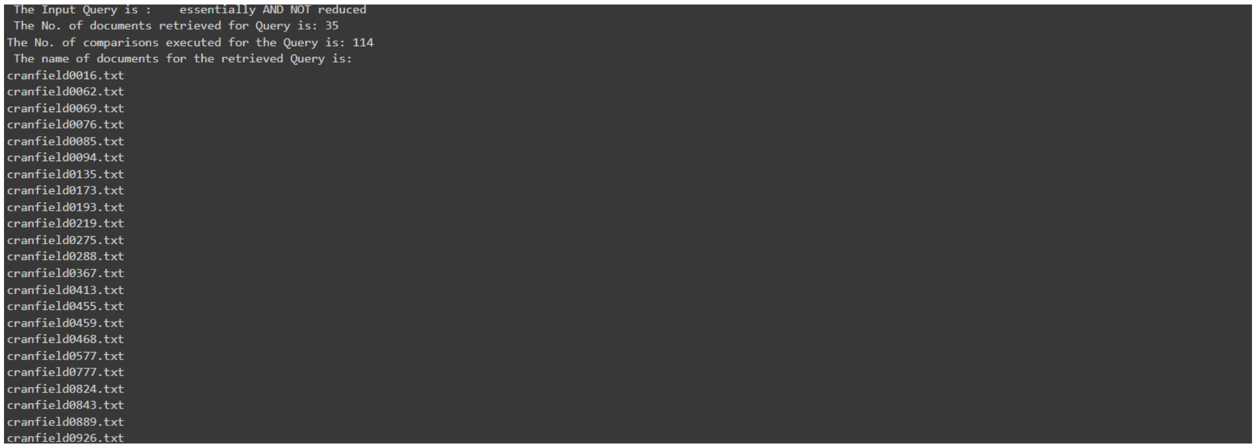
```
 The Input Query is :    essentially AND reduced
 The No. of documents retrieved for Query is: 3
The No. of comparisons executed for the Query is: 114
 The name of documents for the retrieved Query is:
cranfield0014.txt
cranfield0174.txt
cranfield0717.txt
```

For Query - 3 : -



```
 The Input Query is :    essentially OR NOT reduced
 The No. of documents retrieved for Query is: 1323
The No. of comparisons executed for the Query is: 2715
 The name of documents for the retrieved Query is:
cranfield0001.txt
cranfield0002.txt
cranfield0003.txt
cranfield0004.txt
cranfield0005.txt
cranfield0006.txt
cranfield0007.txt
cranfield0008.txt
cranfield0009.txt
cranfield0010.txt
cranfield0011.txt
cranfield0012.txt
cranfield0013.txt
cranfield0014.txt
cranfield0016.txt
cranfield0017.txt
cranfield0018.txt
cranfield0019.txt
cranfield0020.txt
cranfield0021.txt
cranfield0022.txt
cranfield0023.txt
cranfield0025.txt
```

For Query - 4 : -



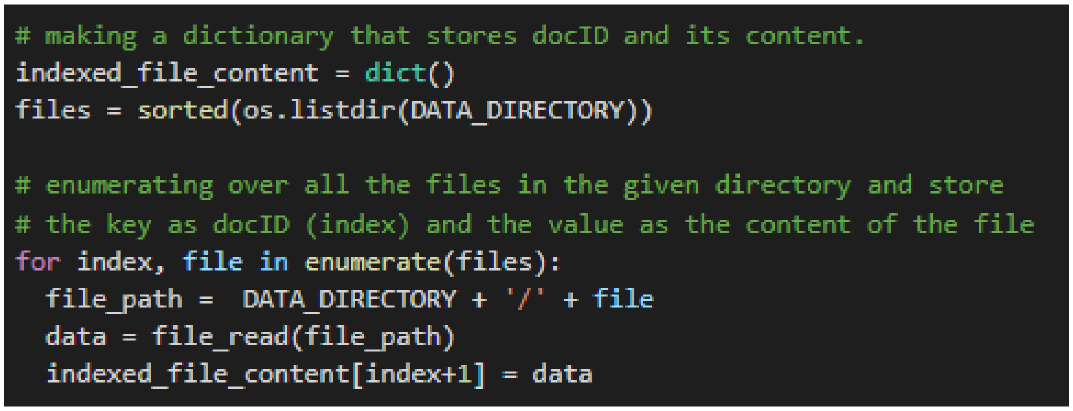
```
 The Input Query is :    essentially AND NOT reduced
 The No. of documents retrieved for Query is: 35
The No. of comparisons executed for the Query is: 114
 The name of documents for the retrieved Query is:
cranfield0016.txt
cranfield0062.txt
cranfield0069.txt
cranfield0076.txt
cranfield0085.txt
cranfield0094.txt
cranfield0135.txt
cranfield0173.txt
cranfield0193.txt
cranfield0219.txt
cranfield0275.txt
cranfield0288.txt
cranfield0367.txt
cranfield0413.txt
cranfield0455.txt
cranfield0459.txt
cranfield0468.txt
cranfield0577.txt
cranfield0777.txt
cranfield0824.txt
cranfield0843.txt
cranfield0889.txt
cranfield0926.txt
```

Question 3: Phrase Queries

We have stored the processed dataset from question 1, and used it in this question for creating bigram and positional indexes.

(i) Bigram inverted index

For this, what we have done is, firstly, we have created a python dictionary that will store the docID as key and its content as value because accessing a dictionary is much less expensive than opening each document, reading its content, and closing the file.



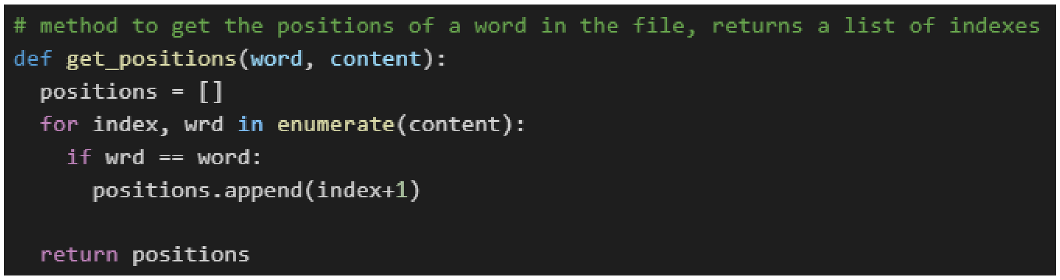
```
# making a dictionary that stores docID and its content.
indexed_file_content = dict()
files = sorted(os.listdir(DATA_DIRECTORY))

# enumerating over all the files in the given directory and store
# the key as docID (index) and the value as the content of the file
for index, file in enumerate(files):
  file_path =  DATA_DIRECTORY + '/' + file
  data = file_read(file_path)
  indexed_file_content[index+1] = data
```

After creating the dictionary, we used it to create our bigram inverted index. For storing the bigram index, we used a **dictionary** to story word pair as the key and its posting list as the value. After this, the total (key, value) pairs in the bigram inverted index is found to be **85114**.

(ii) Positional index

For this, we divided the problem into individual modules, which will perform sub-tasks, and a main function to call these utility functions to create the index.

- **get_positions()**: returns a list of indexes where the given the word exists in the content.

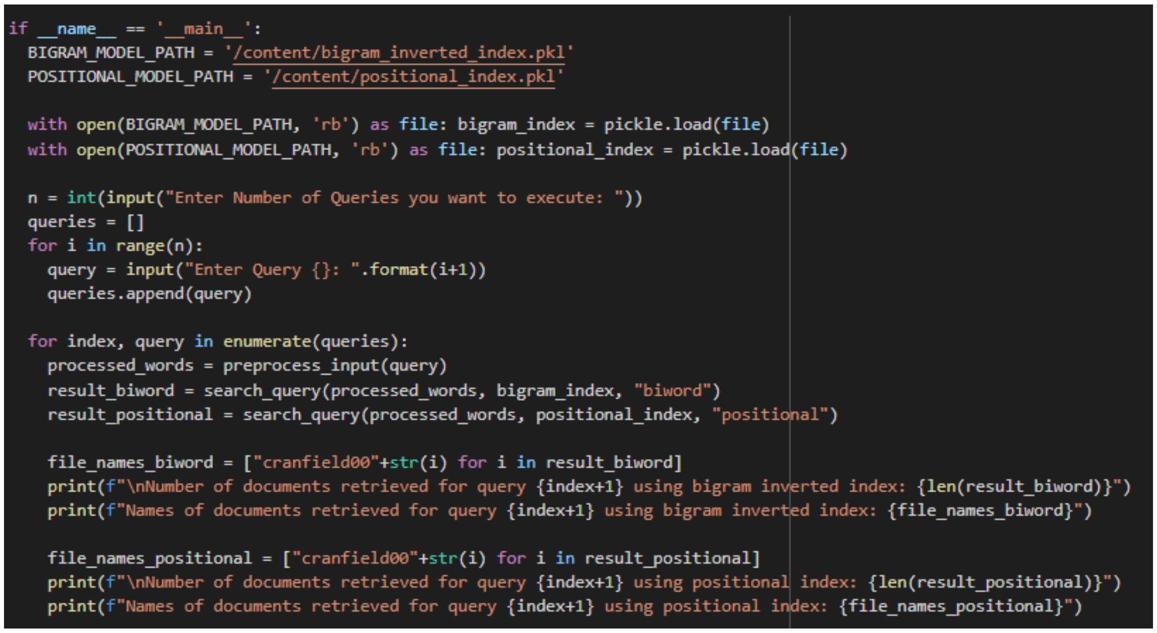


```
# method to get the positions of a word in the file, returns a list of indexes
def get_positions(word, content):
  positions = []
  for index, wrd in enumerate(content):
    if wrd == word:
      positions.append(index+1)

  return positions
```

We have used a python **dictionary** to create the positional index. After creating the positional index, we have found that ***8967*** indexes have been generated. We have created a pickle file for storing the positional_index to use the model in the future.

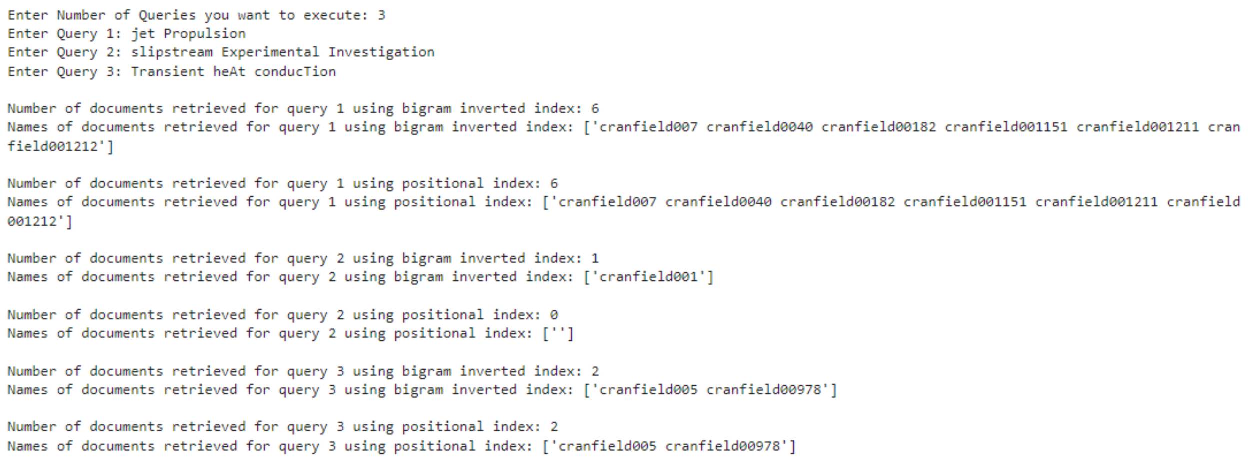
After creating both indexes, we have created a main function that will load the pickle models of these indexes and run an input search query above them to find the documents. We have created utility functions to preprocess the input query based on each index, then find the documents.

For the bigram inverted index, we are taking AND of all the postings list and returning the docIDs. And for the positional index, we first find out the common docs, then iterate over the posting list of the first word, and search for the position of the next word in their corresponding list of positions.



```python
if __name__ == '__main__':
  BIGRAM_MODEL_PATH = '/content/bigram_inverted_index.pkl'
  POSITIONAL_MODEL_PATH = '/content/positional_index.pkl'

  with open(BIGRAM_MODEL_PATH, 'rb') as file: bigram_index = pickle.load(file)
  with open(POSITIONAL_MODEL_PATH, 'rb') as file: positional_index = pickle.load(file)

  n = int(input("Enter Number of Queries you want to execute: "))
  queries = []
  for i in range(n):
    query = input("Enter Query {}: ".format(i+1))
    queries.append(query)

  for index, query in enumerate(queries):
    processed_words = preprocess_input(query)
    result_biword = search_query(processed_words, bigram_index, "biword")
    result_positional = search_query(processed_words, positional_index, "positional")

    file_names_biword = ["cranfield00"+str(i) for i in result_biword]
    print(f"\nNumber of documents retrieved for query {index+1} using bigram inverted index: {len(result_biword)}")
    print(f"Names of documents retrieved for query {index+1} using bigram inverted index: {file_names_biword}")

    file_names_positional = ["cranfield00"+str(i) for i in result_positional]
    print(f"\nNumber of documents retrieved for query {index+1} using positional index: {len(result_positional)}")
    print(f"Names of documents retrieved for query {index+1} using positional index: {file_names_positional}")
```

(iii) Compare and comment on your results using (i) and (ii).



```
Enter Number of Queries you want to execute: 3
Enter Query 1: jet Propulsion
Enter Query 2: slipstream Experimental Investigation
Enter Query 3: Transient heAt conducTion

Number of documents retrieved for query 1 using bigram inverted index: 6
Names of documents retrieved for query 1 using bigram inverted index: ['cranfield007 cranfield0040 cranfield00182 cranfield001151 cranfield001211 cran
field001212']

Number of documents retrieved for query 1 using positional index: 6
Names of documents retrieved for query 1 using positional index: ['cranfield007 cranfield0040 cranfield00182 cranfield001151 cranfield001211 cranfield
001212']

Number of documents retrieved for query 2 using bigram inverted index: 1
Names of documents retrieved for query 2 using bigram inverted index: ['cranfield001']

Number of documents retrieved for query 2 using positional index: 0
Names of documents retrieved for query 2 using positional index: ['']

Number of documents retrieved for query 3 using bigram inverted index: 2
Names of documents retrieved for query 3 using bigram inverted index: ['cranfield005 cranfield00978']

Number of documents retrieved for query 3 using positional index: 2
Names of documents retrieved for query 3 using positional index: ['cranfield005 cranfield00978']
```

For query 1: "**slipstream experimental Investigation,**" the bigram Index returns one file, but positional index returns zero files. This is because bigram indexes give ***FALSE POSITIVE*** results, which is not an issue in positional indexing. In the document **"cranfield001"**, (slipstream experimental), and (experimental Investigation) exists but not in the correct order or sequence, so it gives this file as output. This is the limitation of bigram indexing, and hence we use positional indexing.