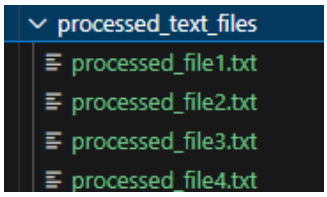
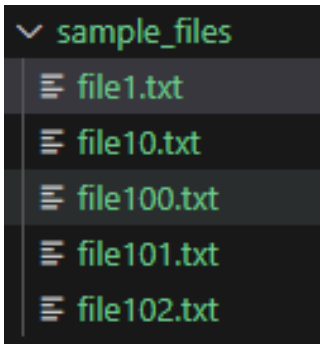
Q1.

Q1. Data Preprocessing [20 Marks]

1. Perform the following preprocessing steps on each of the text files in the dataset linked above.
    a. Lowercase the text
    b. Perform tokenization
    c. Remove stopwords
    d. Remove punctuations
    e. Remove blank space tokens
2. Print contents of 5 sample files before and after performing each operation. Remember to save each file after preprocessing to use the preprocessed file for the following tasks.

```python
# preprocess
def preprocess_file(input_path, output_path):
    with open(input_path, 'r', encoding='utf-8') as file:
        content = file.read()
    # a. Lowercase the text
    content = content.lower()
    # b. Tokenization
    tokens = word_tokenize(content)
    # c. Remove stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words]
    # d. Remove punctuations
    tokens = [token for token in tokens if token not in string.punctuation]
    temp_tokens = []
    for token in tokens:
        temp_token = ""
        for char in token:
            if char.isalnum():
                temp_token += char
        temp_tokens.append(temp_token)
    tokens = temp_tokens
    # e. Remove blank space tokens
    tokens = [token for token in tokens if token.strip()]
    # Save the preprocessed content to a new file
    with open(output_path, 'w', encoding='utf-8') as file:
        file.write('\n'.join(tokens))
    return tokens
```





- The provided Python code performs preprocessing tasks on a collection of text files. It begins by importing necessary libraries such as NLTK and string. Then, it defines a function preprocess_file() to carry out the following preprocessing steps on each text file:

1. Lowercasing the text to ensure uniformity.

2. Tokenizing the text to segment it into individual tokens.
3. Removing stopwords using NLTK's English stopwords corpus.
4. Removing punctuation marks using the string.punctuation set.
5. Removing blank space tokens.

The code iterates over each file in the input directory, applies the preprocessing steps using the preprocess_file() function, and saves the preprocessed content to new files in the output directory. Additionally, it prints the contents of five sample files before and after each preprocessing step, demonstrating the effect of each operation.

Q2.

```python
def create_inverted_index(directory):
    inverted_index = {}
    document_ids = os.listdir(directory)
    for doc_id in document_ids:
        with open(os.path.join(directory, doc_id), 'r') as file:
            terms = file.read().strip().split('\n')
            for term in terms:
                if term not in inverted_index:
                    inverted_index[term] = set()
                inverted_index[term].add(doc_id)
    return inverted_index

def save_inverted_index(index, filename):
    with open(filename, 'wb') as file:
        pickle.dump(index, file)

def load_inverted_index(filename):
    with open(filename, 'rb') as file:
        index = pickle.load(file)
    return index
```

```python
def process_query(query, operations, inverted_index):
    terms = preprocess_query(query)
    operations = [operation.strip() for operation in operations]
    print(operations)
    if len(operations) != len(terms) - 1:
        print("Error: Number of operations doesn't match the number of terms.")
        return set()  # Return an empty set to indicate an error
    result = set(inverted_index[terms[0]])

    for i in range(1, len(terms)):

        if terms[i] not in inverted_index.keys():
            inverted_index[terms[i]] = set()

        if operations[i-1] == "AND":
            result = result.intersection(set(inverted_index[terms[i]]))
        elif operations[i-1] == "OR":
            result = result.union(set(inverted_index[terms[i]]))
        elif operations[i-1] == "OR NOT":
            result = result.union(file_not(terms[i], inverted_index))
        elif operations[i-1] == "AND NOT":
            result = result.intersection(file_not(terms[i], inverted_index))
    return result

def execute_queries(queries, inverted_index):
    results = []
    for input_sequence, operations in queries:
        result = process_query(input_sequence, operations, inverted_index)
        results.append(result)
    return results
```

1. Create a unigram inverted index (from scratch):

- The create_inverted_index() function generates the inverted index based on the terms extracted from the processed text files.

2. Use Python’s pickle module to save and load the unigram inverted index:

- The save_inverted_index() function saves the inverted index to a file using Python's pickle module, and the load_inverted_index() function loads the inverted index from the saved file.

3. function process_query() processes the query and performs binary operations

4. The input is taken from the user and then pre-processed, after which binary operations are performed to give the output files

Q3.

1. Preprocessing and Tokenization: The preprocess_input function tokenizes the query, converts tokens to lowercase, removes punctuation, and eliminates stop words using NLTK's functionalities.

2. Positional Index Creation: The create_positional_index function generates a positional index, where each term maps to a dictionary of document IDs along with their corresponding positions in the documents.

3. Saving Positional Index: The save_positional_index function utilizes the pickle module to serialize and save the positional index dictionary to a binary file specified by the filename.

4. Loading Positional Index: The load_positional_index function loads the saved positional index from the binary file using pickle's load function.

5. Processing Phrase Queries: The process_phrase_query function retrieves documents containing a given phrase query by checking if the terms appear consecutively with valid positions in the positional index.

6. Document Retrieval: The get_document_names function returns a list of document names corresponding to the retrieved document IDs.

7. Reading Sample Documents: The code reads and processes text documents from a specified directory, preparing them for the creation of the positional index.

8. Creating and Saving Index: It generates the positional index from the sample documents and saves it to a file for future use.

9. User Input and Query Processing: The code prompts the user to input the number of queries and the queries themselves, processes each query against the loaded positional index, and retrieves relevant documents.

10. Output Display: Finally, the code displays the number of retrieved documents and their names for each query, adhering to the specified output format.