REPORT: ASSIGNMENT-1

Roll number: 2021228

Question-1)

Part-a:

1. **Function Definition: `lowercase(file_num)`**
    - This function takes an integer argument `file_num`.
    - It is responsible for converting the contents of a text file to lowercase using the lower() function on the text read by the function.
    - The input file is specified by the `file_num` parameter (e.g., `file1.txt`, `file2.txt`, etc.).
2. **Setting Up Paths:**
    - The script starts by determining the current directory where it is executed using `os.path.dirname(os.path.realpath(__file__))`.
    - It then constructs the path to the directory containing the text files (`text_files_directory`) relative to the script's location.
    - The input file path (`text_file_path`) and output file path (`text_file_path_out`) are created based on the `file_num`.
3. **Reading and Lowercasing Content:**
    - The script reads the content of the input text file (`text_file_path`) using `with open(text_file_path, 'r') as file`.
    - The entire content is converted to lowercase using the `.lower()` method.
    - The lowercased content is stored in the `content` variable.
4. **Creating and Writing to Output File:**
    - An output file (`text_file_path_out`) is created using `open(text_file_path_out, 'x')`.
    - The lowercased content is written to the output file using `with open(text_file_path_out, 'w') as file`.
    - The output file now contains the same content as the input file but in lowercase.
5. **Printing Content (Optional):**
    - If `file_num` is less than or equal to 5, the script reads the content from the output file (`text_file_path_out`) and prints it to the console.
6. **Main Function: `main()`**
    - The `main()` function iterates through a range of file numbers (from 1 to 999).
    - For each file number, it calls the `lowercase(file_number)` function to process the corresponding text file.

Part-b:

1. **Function Definition: `tokenize(file_num)`**
    - This function takes an integer argument `file_num`.
    - Its purpose is to tokenize the content of a text file into individual words using the Natural Language Toolkit (NLTK) library using the word_tokenize() function on the contents of the text files.
    - The input file is specified by the file_num parameter (e.g., `file1_1a.txt`, `file2_1a.txt`, etc.).
2. **Setting Up Paths:**
    - The script determines the current directory where it is executed using `os.path.dirname(os.path.realpath(__file__))`.
    - It constructs the path to the directory containing the text files (`text_files_directory`) relative to the script's location.
    - The input file path (`text_file_path`) and output file path (`text_file_path_out`) are created based on the `file_num`.
3. **Tokenization with NLTK:**
    - The `with open(text_file_path, 'r') as file` statement opens the input text file in read mode.
    - The entire content of the input file is read into the `content` variable.
    - The NLTK function `nltk.word_tokenize(content)` tokenizes the content into individual words (tokens).
    - The resulting list of words is stored in the `words` variable.
4. **Creating and Writing to Output File:**
    - An output file (`text_file_path_out`) is created using `open(text_file_path_out, 'x')`.
    - The `'x'` mode ensures that the file is created only if it does not already exist.
    - The tokenized words are joined into a single string using `" ".join(words)`.
    - The joined string is written to the output file using `with open(text_file_path_out, 'w') as file`.
5. **Optional Printing of Content:**
    - If `file_num` is less than or equal to 5, the script reads the content from the output file (`text_file_path_out`) and prints it to the console.
6. **Main Function: `main()`**
    - The `main()` function iterates through a range of file numbers (from 1 to 999).
    - For each file number, it calls the `tokenize(file_number)` function to process the corresponding text file.

Part-c:

1. **Function Definition: `remove_stopwords(file_num)`**
    - This function takes an integer argument `file_num`.
    - Its purpose is to remove common stopwords from the content of a text file using the Natural Language Toolkit (NLTK) library by going word by word through the content of the text files, and if any word lies in the stop-words list in the NLTK library, it is ignored while the rest are stored.
    - The input file is specified by the `file_num` parameter (e.g., `file1_1b.txt`, `file2_1b.txt`, etc.).
2. **Setting Up Paths:**
    - The script determines the current directory where it is executed using `os.path.dirname(os.path.realpath(__file__))`.
    - It constructs the path to the directory containing the text files (`text_files_directory`) relative to the script's location.
    - The input file path (`text_file_path`) and output file path (`text_file_path_out`) are created based on the `file_num`.
3. **Loading Stopwords with NLTK:**
    - The NLTK library provides a set of common stopwords for various languages.
    - The `stopwords.words('english')` call retrieves the list of English stopwords.
    - These stopwords include common words like "the," "and," "is," etc., which are often removed during text analysis.
4. **Tokenization and Filtering:**
    - The `with open(text_file_path, 'r') as file` statement opens the input text file in read mode.
    - The entire content of the input file is read into the `content` variable.
    - The NLTK function `nltk.word_tokenize(content)` tokenizes the content into individual words (tokens).
    - The list comprehension `[word.lower() for word in words if (word.lower() not in stop_words) and (word.lower() not in string.punctuation)]` filters out stopwords and punctuation marks.
    - The filtered words are stored in the `filtered_words` list.
5. **Creating and Writing to Output File:**
    - An output file (`text_file_path_out`) is created using `open(text_file_path_out, 'x')`.
    - The `'x'` mode ensures that the file is created only if it does not already exist.
    - The filtered words are joined into a single string using `" ".join(filtered_words)`.

    - The joined string is written to the output file using `with open(text_file_path_out, 'w') as file`.
6. **Optional Printing of Content:**
    - If `file_num` is less than or equal to 5, the script reads the content from the output file (`text_file_path_out`) and prints it to the console.
7. **Main Function: `main()`**
    - The `main()` function iterates through a range of file numbers (from 1 to 999).
    - For each file number, it calls the `remove_stopwords(file_number)` function to process the corresponding text file.

Part-d:

1. **Function Definition: `remove_punc(file_num)`**
    - This function takes an integer argument `file_num`.
    - Its purpose is to remove punctuation marks from the content of a text file.
    - The input file is specified by the `file_num` parameter (e.g., `file1_1c.txt`, `file2_1c.txt`, etc.).
2. **Setting Up Paths:**
    - The script determines the current directory where it is executed using `os.path.dirname(os.path.realpath(__file__))`.
    - It constructs the path to the directory containing the text files (`text_files_directory`) relative to the script's location.
    - The input file path (`text_file_path`) and output file path (`text_file_path_out`) are created based on the `file_num`.
3. **Removing Punctuation:**
    - The `with open(text_file_path, 'r') as file` statement opens the input text file in read mode.
    - The entire content of the input file is read into the `content` variable.
    - The `content.translate(str.maketrans('', '', string.punctuation))` line removes all punctuation marks from the content.
    - The `string.punctuation` constant contains a string of all punctuation characters (e.g., `.,!?`).
4. **Creating and Writing to Output File:**
    - An output file (`text_file_path_out`) is created using `open(text_file_path_out, 'x')`.
    - The `'x'` mode ensures that the file is created only if it does not already exist.
    - The modified content (without punctuation) is written to the output file using `with open(text_file_path_out, 'w') as file`.
5. **Optional Printing of Content:**

- If `file_num` is less than or equal to 5, the script reads the content from the output file (`text_file_path_out`) and prints it to the console.

6. **Main Function: `main()`**
   - The `main()` function iterates through a range of file numbers (from 1 to 999).
   - For each file number, it calls the `remove_punc(file_number)` function to process the corresponding text file.

Part-e:

1. **Function Definition: `tokenize(file_num)`**
   - This function takes an integer argument `file_num`.
   - Its purpose is to tokenize the content of a text file into individual words using the Natural Language Toolkit (NLTK) library.
   - The input file is specified by the `file_num` parameter (e.g., `file1_1d.txt`, `file2_1d.txt`, etc.).
2. **Setting Up Paths:**
   - The script determines the current directory where it is executed using `os.path.dirname(os.path.realpath(__file__))`.
   - It constructs the path to the directory containing the text files (`text_files_directory`) relative to the script's location.
   - The input file path (`text_file_path`) and output file path (`text_file_path_out`) are created based on the `file_num`.
3. **Tokenization with NLTK:**
   - The `with open(text_file_path, 'r') as file` statement opens the input text file in read mode.
   - The entire content of the input file is read into the `content` variable.
   - The NLTK function `nltk.word_tokenize(content)` tokenizes the content into individual words (tokens).
   - The resulting list of words is stored in the `words` variable.
4. **Filtering Empty Tokens:**
   - The list comprehension `[token for token in words if token.strip()]` filters out any empty tokens.
   - An empty token could occur due to extra whitespace or other non-visible characters.
   - By using `token.strip()`, we ensure that only non-empty tokens are included in the `filtered_words` list.
5. **Creating and Writing to Output File:**
   - An output file (`text_file_path_out`) is created using `open(text_file_path_out, 'x')`.
   - The `'x'` mode ensures that the file is created only if it does not already exist.

- The filtered words are joined into a single string using `" ".join(words)`.
- The joined string is written to the output file using `with open(text_file_path_out, 'w') as file`.

6. **Optional Printing of Content:**
   - If `file_num` is less than or equal to 5, the script reads the content from the output file (`text_file_path_out`) and prints it to the console.
7. **Main Function: `main()`**
   - The `main()` function iterates through a range of file numbers (from 1 to 999).
   - For each file number, it calls the `tokenize(file_number)` function to process the corresponding text file.

Question-2)

1. **Imports and Downloads:**
   - The code begins by importing necessary libraries: `os` for file handling, `nltk` for natural language processing tasks, `stopwords` and `string` from the NLTK library for handling stop words and punctuation, respectively.
   - It also downloads necessary NLTK resources using `nltk.download()` function.

2. Function Definitions:
   - `**process_queries(queries, operation, dict)**`: This function processes the queries provided by the user. It takes three parameters: `queries` (a list of words), `operation` (a string representing the logical operation between queries), and `dict` (the inverted index dictionary). It iterates through each query and applies the given logical operation to the corresponding sets of document numbers. It calls `evaluate_condition()` function to perform the actual evaluation.

Special case: if the size of the operations is 0, i.e. there are no boolean operators as inputs, then the function straight away returns the documents containing the 1st word of the query assuming that the query has only 1 word and no operator to follow.

   - `**evaluate_condition(final, word2, operation, dict)**`: This function evaluates the logical condition based on the given operation. It takes four parameters: `final` (the set of document numbers obtained so far), `word2` (the second word or query term), `operation` (the logical operation), and `dict` (the inverted index dictionary). Depending on the operation, it performs set operations like union, intersection, difference, etc., and returns the resulting set of document numbers.
   - `**invert_ind(file, dict, num)`:** This function builds the inverted index. It reads each text file, tokenizes it, and updates the inverted index dictionary (`dict`) with the document numbers where each word occurs.

3. Inverted Index Creation:

- The code initializes an empty dictionary `my_dict` to store the inverted index.
- It iterates through a range of file numbers (different documents).
- For each file, it constructs the file path, reads the content of the file, tokenizes it, and updates the inverted index using the `invert_ind()` function.

4. Serialization (Pickling):
- After constructing the inverted index, the code serializes it using the `pickle` module. It writes the dictionary to a file named 'index.pkl'.
- It then reads the serialized dictionary back into memory using `pickle.load()`.

5. **User Interaction & basic working:**
- The code prompts the user to enter the number of queries (N).
- It then enters a loop to process each query.
- For each query, it prompts the user to input the query string and the logical operation to be applied.
- It tokenizes the query, removes stop words and punctuation, and passes the filtered words along with the operation to `process_queries()` function for evaluation.
- After processing each query, it prints the names of documents retrieved for that query along with the count of retrieved documents.

Question-3)

1. Reading Documents:
- The code starts by iterating over the range of document numbers from 1 to 999.
- For each document, it reads the content from a file named `file{i}_1e.txt` located in the directory `../text_files`, and appends the content to a list named `lis`.

2. Positional Inverted Index Creation:
- It defines a function `cpi(docs)` (Create Positional Index) to build the positional inverted index from the list of documents.
- Inside the function, it initializes a defaultdict `pi` to store the positional index.
- It tokenizes each document using `nltk.word_tokenize()`, converts the tokens to lowercase, and iterates over each token.
- For each token, it updates the positional index with the document ID and the positions of the token within the document.
- It calculates the term document frequency (TDF) for each token and constructs a dictionary `piwf` containing the TDF and the positional index for each token.
- Finally, it returns the positional inverted index `piwf`.

3. Serializing the Positional Inverted Index:
- The code serializes the positional inverted index using the `pickle` module and writes it to a file named "pi.pkl".

4. Loading the Positional Inverted Index:

- It loads the serialized positional inverted index from the "pi.pkl" file back into memory using `pickle.load()`.

5. Retrieving Documents for Queries:

- It defines a function `retdoc(pi, query)` to retrieve documents based on a query from the positional inverted index.
- The function takes the positional inverted index `pi` and a query as input.
- It first preprocesses the query by converting it to lowercase, removing stopwords, and removing punctuation.
- It then retrieves documents containing all the terms in the query by performing set intersections on the valid document IDs for each term.
- For each valid document, it checks if the positions of the terms in the document match the positions specified in the query.
- It returns the list of document IDs satisfying the query.

6. User Interaction:

- The code prompts the user to enter the number of queries (`N`).
- It then enters a loop to process each query.
- For each query, it prompts the user to input the query string.
- It preprocesses the query by converting it to lowercase, removing stopwords, and removing punctuation.
- It calls the `retdoc()` function to retrieve documents for the query and prints the number and names of the retrieved documents.