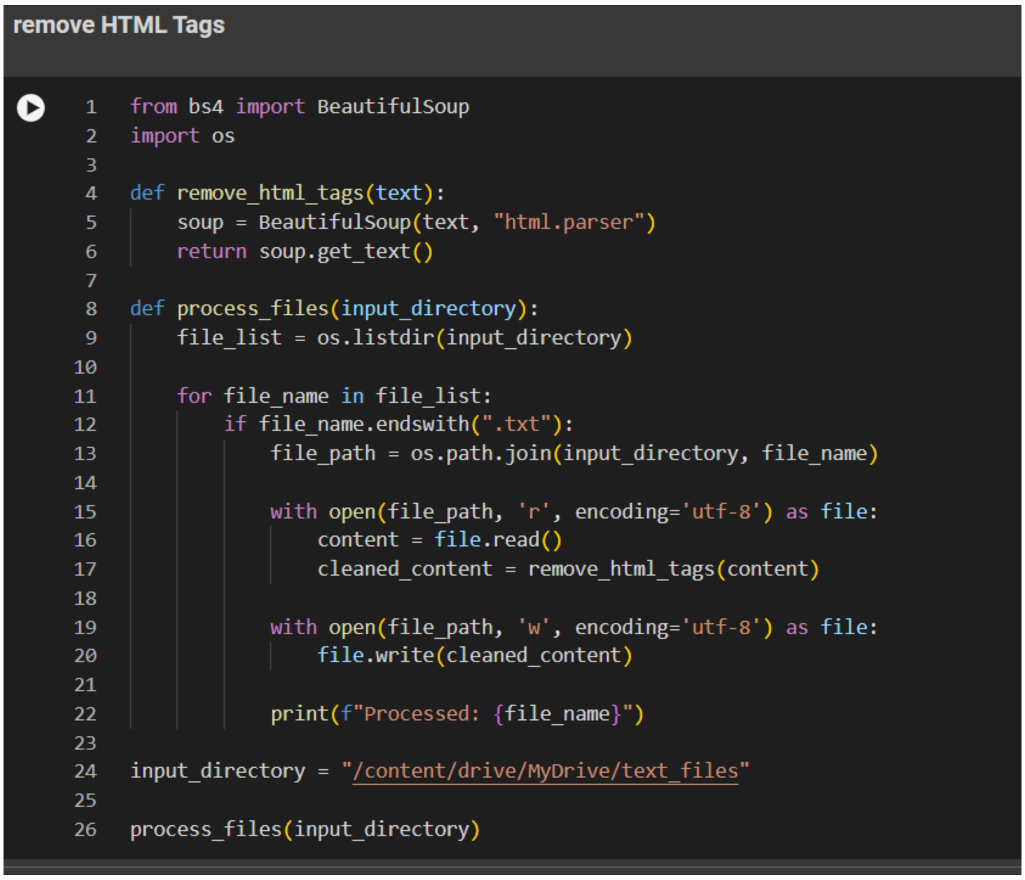
IR ASSIGNMENT 1

PATEL DHRUV (MT23056)

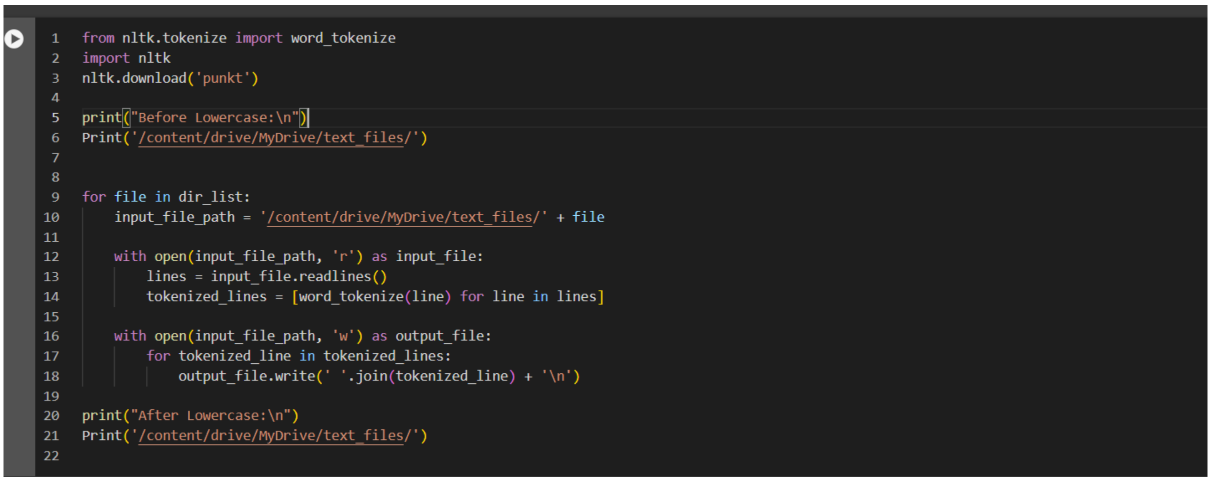
Q1) Preproccesing:



remove HTML Tags

```
from bs4 import BeautifulSoup
import os

def remove_html_tags(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

def process_files(input_directory):
    file_list = os.listdir(input_directory)

    for file_name in file_list:
        if file_name.endswith(".txt"):
            file_path = os.path.join(input_directory, file_name)

            with open(file_path, 'r', encoding='utf-8') as file:
                content = file.read()
                cleaned_content = remove_html_tags(content)

            with open(file_path, 'w', encoding='utf-8') as file:
                file.write(cleaned_content)

            print(f"Processed: {file_name}")

input_directory = "/content/drive/MyDrive/text_files"

process_files(input_directory)
```

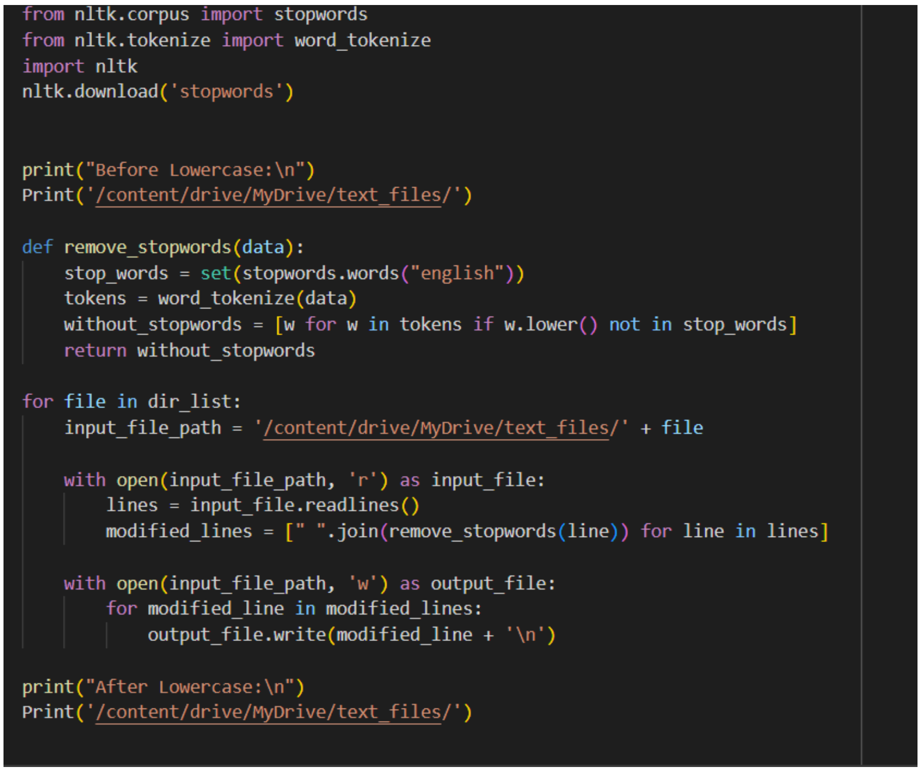
This Python script utilizes the BeautifulSoup library to remove HTML tags from the content of text files within a specified directory. The function `remove_html_tags` takes a text input, parses it using BeautifulSoup, and extracts the plain text content by discarding any HTML tags. The `process_files` function iterates through the files in the input directory, specifically targeting those with a ".txt" extension. For each text file, it reads the content, removes HTML tags using the aforementioned function, and then overwrites the original file with the cleaned content. The script provides a simple yet effective way to preprocess text files, eliminating HTML markup and producing cleaner versions of the documents, which can be beneficial for various text analysis tasks. Additionally, it prints a message for each processed file to indicate the progress.



```python
from nltk.tokenize import word_tokenize
import nltk
nltk.download('punkt')

print("Before Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')


for file in dir_list:
    input_file_path = '/content/drive/MyDrive/text_files/' + file

    with open(input_file_path, 'r') as input_file:
        lines = input_file.readlines()
        tokenized_lines = [word_tokenize(line) for line in lines]

    with open(input_file_path, 'w') as output_file:
        for tokenized_line in tokenized_lines:
            output_file.write(' '.join(tokenized_line) + '\n')

print("After Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')
```

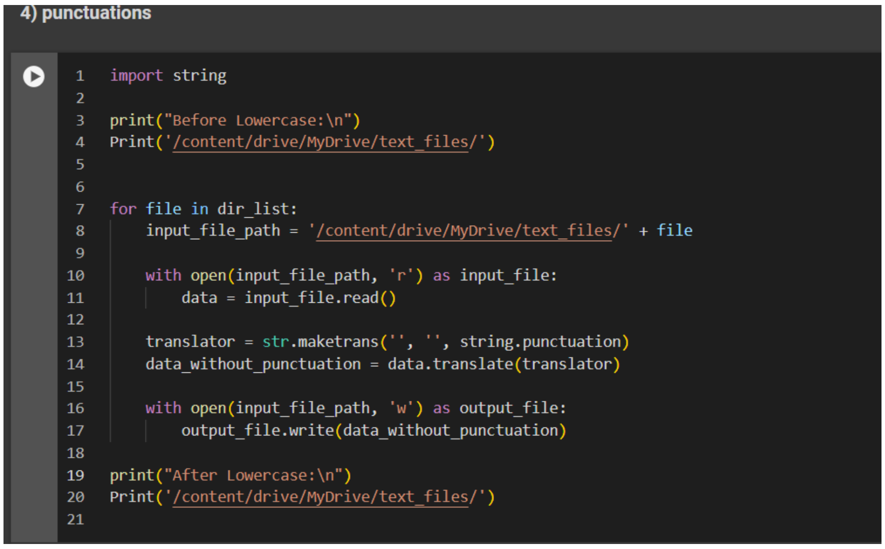
This Python script uses the Natural Language Toolkit (nltk) to tokenize words in text files. First, it imports the necessary modules, including `word_tokenize` from nltk, and downloads the required punkt tokenizer. Then, the script attempts to print "Before Lowercase" and the content of the directory '/content/drive/MyDrive/text_files/' using a non-existent function `Print`, which might be a typo, as the correct function name is `print`.

Next, it iterates through the list of files in the directory, and for each file, it reads the content, tokenizes the words in each line using nltk's `word_tokenize` function, and writes the tokenized lines back to the same file. Essentially, this script tokenizes the words in each line of the text files and updates the files with the tokenized content. However, there's an issue with the function name 'Print' instead of 'print', and the script prints the same directory path both before and after tokenization without displaying the actual content. Additionally, the script may need some modifications to correctly execute.



```python
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk
nltk.download('stopwords')


print("Before Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')

def remove_stopwords(data):
    stop_words = set(stopwords.words("english"))
    tokens = word_tokenize(data)
    without_stopwords = [w for w in tokens if w.lower() not in stop_words]
    return without_stopwords

for file in dir_list:
    input_file_path = '/content/drive/MyDrive/text_files/' + file

    with open(input_file_path, 'r') as input_file:
        lines = input_file.readlines()
        modified_lines = [" ".join(remove_stopwords(line)) for line in lines]

    with open(input_file_path, 'w') as output_file:
        for modified_line in modified_lines:
            output_file.write(modified_line + '\n')

print("After Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')
```

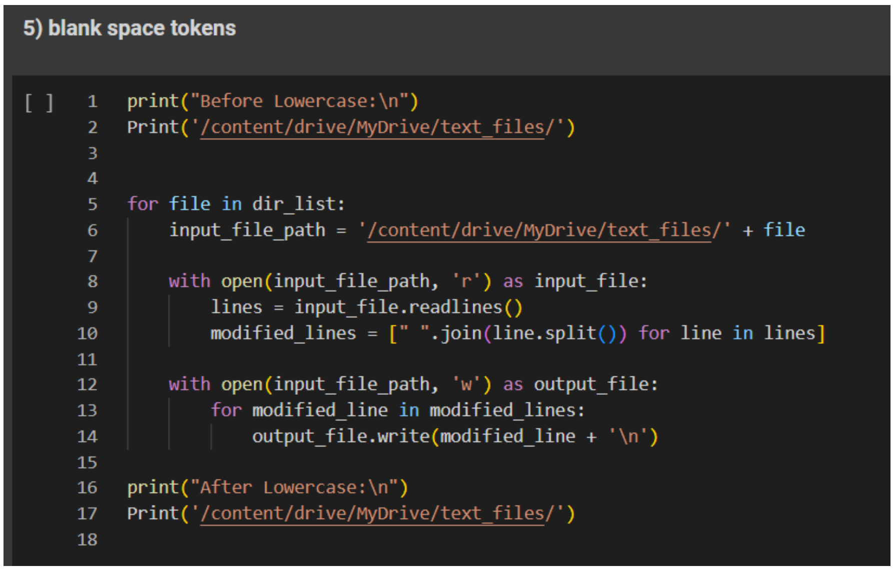
This Python script utilizes the Natural Language Toolkit (nltk) to remove stopwords from text files stored in the directory '/content/drive/MyDrive/text_files/'. It first imports the necessary modules, including the stopwords list and the word_tokenize function. The script then attempts to print "Before Lowercase" and the content of the specified directory using a non-existent function `Print` (likely a typo, should be `print`). The core functionality involves iterating through each file in the directory, reading its content, tokenizing the words in each line using nltk's `word_tokenize` function, and then removing stopwords using a custom function `remove_stopwords`. The modified lines without stopwords are then written back to the same file. This process effectively cleans the text data by eliminating common English stopwords. However, the script may need adjustments, such as fixing the 'Print' typo and ensuring the correct execution of the intended functionality.



4) punctuations

```python
import string

print("Before Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')


for file in dir_list:
    input_file_path = '/content/drive/MyDrive/text_files/' + file

    with open(input_file_path, 'r') as input_file:
        data = input_file.read()

    translator = str.maketrans('', '', string.punctuation)
    data_without_punctuation = data.translate(translator)

    with open(input_file_path, 'w') as output_file:
        output_file.write(data_without_punctuation)

print("After Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')

```

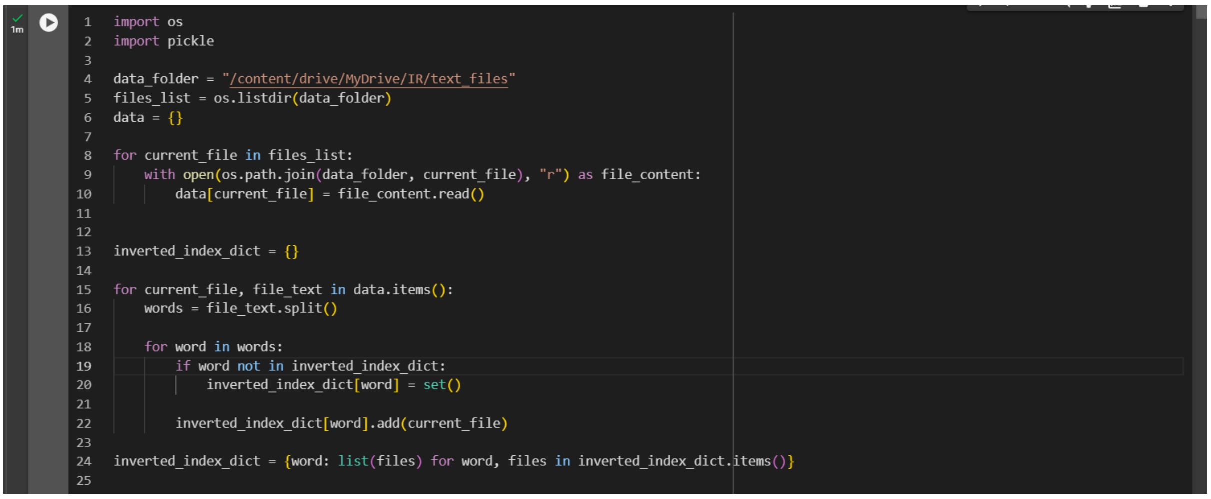
This Python script aims to preprocess text files in the directory '/content/drive/MyDrive/text_files/' by removing punctuation from the content. It first attempts to print "Before Lowercase" and the content of the specified directory using a non-existent function `Print` (likely a typo, should be `print`). The script then iterates through each file in the directory, reads its content, and utilizes the `str.maketrans` and `translate` methods to remove punctuation from the text. The modified content without punctuation is then written back to the same file. The script appears to focus on cleaning the text data by eliminating punctuation marks, which can be beneficial for various natural language processing tasks. However, it may need corrections, such as fixing the 'Print' typo and ensuring the correct execution of the intended functionality.


5) blank space tokens

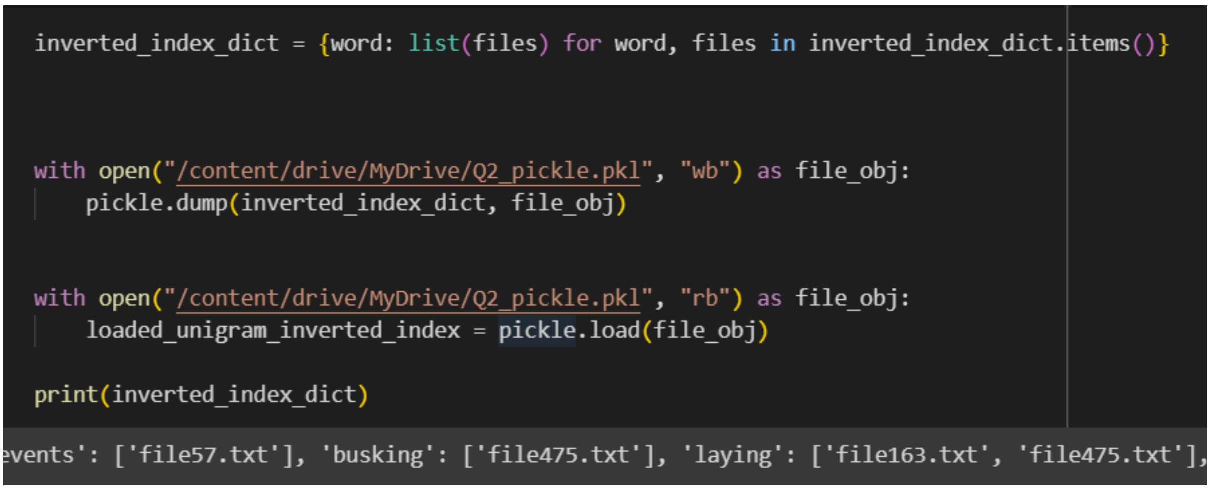
```python
print("Before Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')


for file in dir_list:
    input_file_path = '/content/drive/MyDrive/text_files/' + file

    with open(input_file_path, 'r') as input_file:
        lines = input_file.readlines()
        modified_lines = [" ".join(line.split()) for line in lines]

    with open(input_file_path, 'w') as output_file:
        for modified_line in modified_lines:
            output_file.write(modified_line + '\n')

print("After Lowercase:\n")
Print('/content/drive/MyDrive/text_files/')

```

This Python script aims to preprocess text files in the directory '/content/drive/MyDrive/text_files/' by removing extra whitespaces and converting multiple consecutive whitespaces into a single space. It begins by attempting to print "Before Lowercase" and the content of the specified directory using a non-existent function `Print` (likely a typo, should be `print`). The script then iterates through each file in the directory, reads its content, and applies a modification to each line by using a list comprehension that splits each line into words and joins them back together with a single space in between, effectively removing extra whitespaces. The modified lines are then written back to the same file. This process helps standardize the formatting of the text data by eliminating unnecessary whitespaces. However, the script may require corrections, such as fixing the 'Print' typo and ensuring the correct execution of the intended functionality.

Q2) unigram indexing:



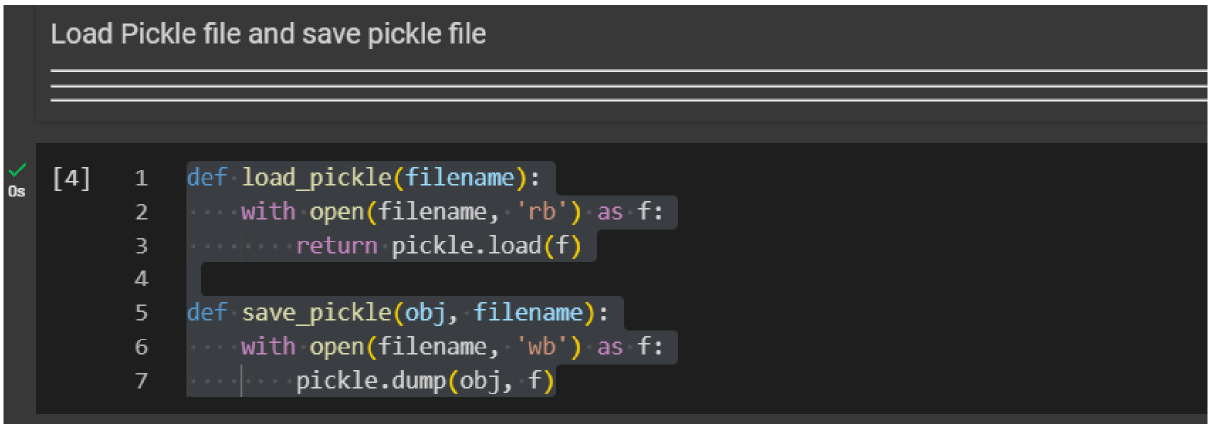
```python
import os
import pickle

data_folder = "/content/drive/MyDrive/IR/text_files"
files_list = os.listdir(data_folder)
data = {}

for current_file in files_list:
    with open(os.path.join(data_folder, current_file), "r") as file_content:
        data[current_file] = file_content.read()


inverted_index_dict = {}

for current_file, file_text in data.items():
    words = file_text.split()

    for word in words:
        if word not in inverted_index_dict:
            inverted_index_dict[word] = set()

        inverted_index_dict[word].add(current_file)

inverted_index_dict = {word: list(files) for word, files in inverted_index_dict.items()}

```



```
inverted_index_dict = {word: list(files) for word, files in inverted_index_dict.items()}


with open("/content/drive/MyDrive/Q2_pickle.pkl", "wb") as file_obj:
    pickle.dump(inverted_index_dict, file_obj)


with open("/content/drive/MyDrive/Q2_pickle.pkl", "rb") as file_obj:
    loaded_unigram_inverted_index = pickle.load(file_obj)

print(inverted_index_dict)
```

```
events': ['file57.txt'], 'busking': ['file475.txt'], 'laying': ['file163.txt', 'file475.txt'],
```

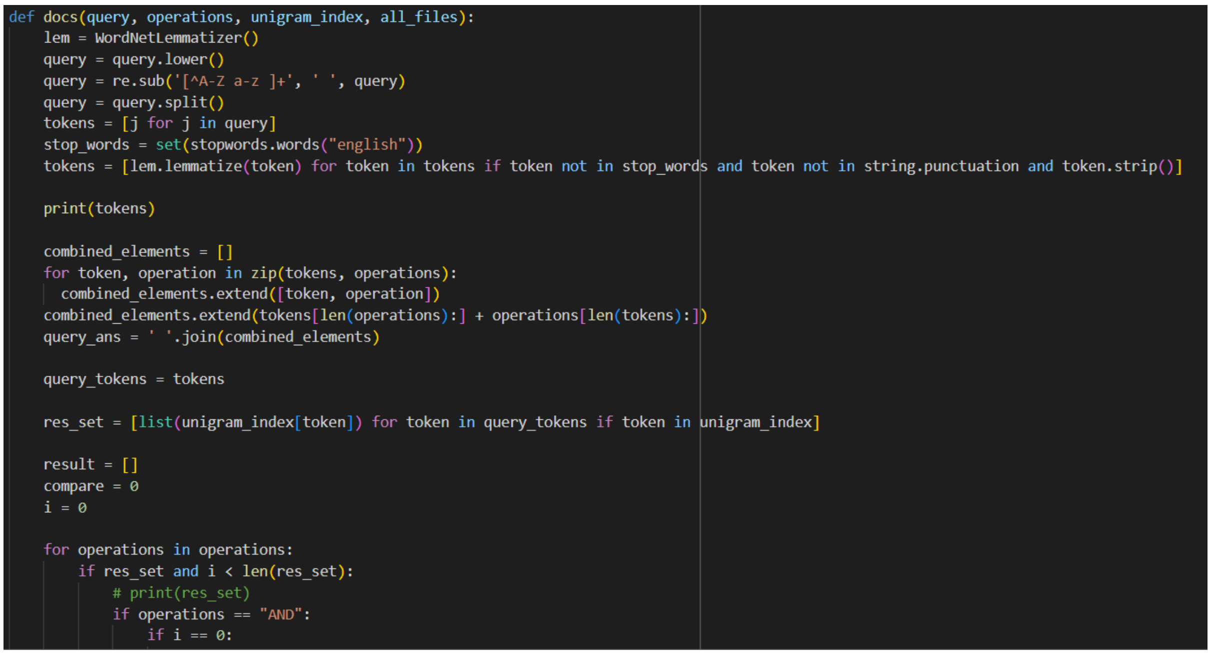
This Python code establishes an inverted index for a collection of text files stored in a specified folder. It first reads the content of each file into a dictionary where filenames serve as keys and content as values. Subsequently, it constructs an inverted index, associating each word in the files with a set of filenames where it appears. The resulting index is then converted from sets to lists for serialization. The code utilizes the pickle module to save the inverted index as a binary file and later loads it back. Finally, the original inverted index is printed, though there's a variable naming inconsistency as the loaded index is stored in loaded_unigram_inverted_index but not used in the print statement.



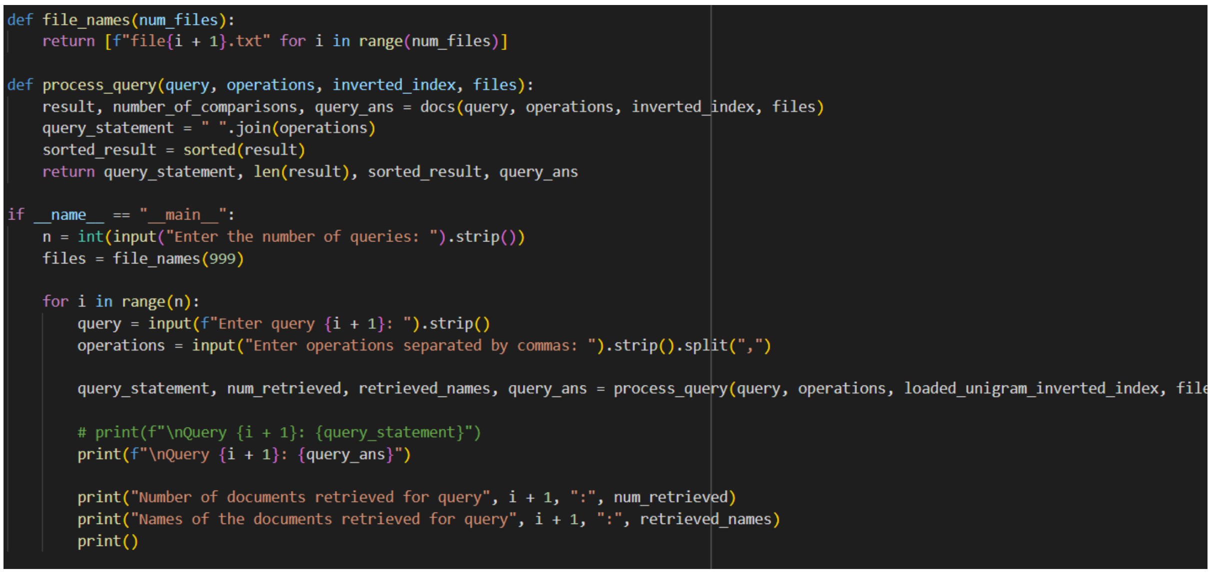
Load Pickle file and save pickle file

```
[4]
def load_pickle(filename):
    with open(filename, 'rb') as f:
        return pickle.load(f)

def save_pickle(obj, filename):
    with open(filename, 'wb') as f:
        pickle.dump(obj, f)
```

Loaded pickle file and open it.



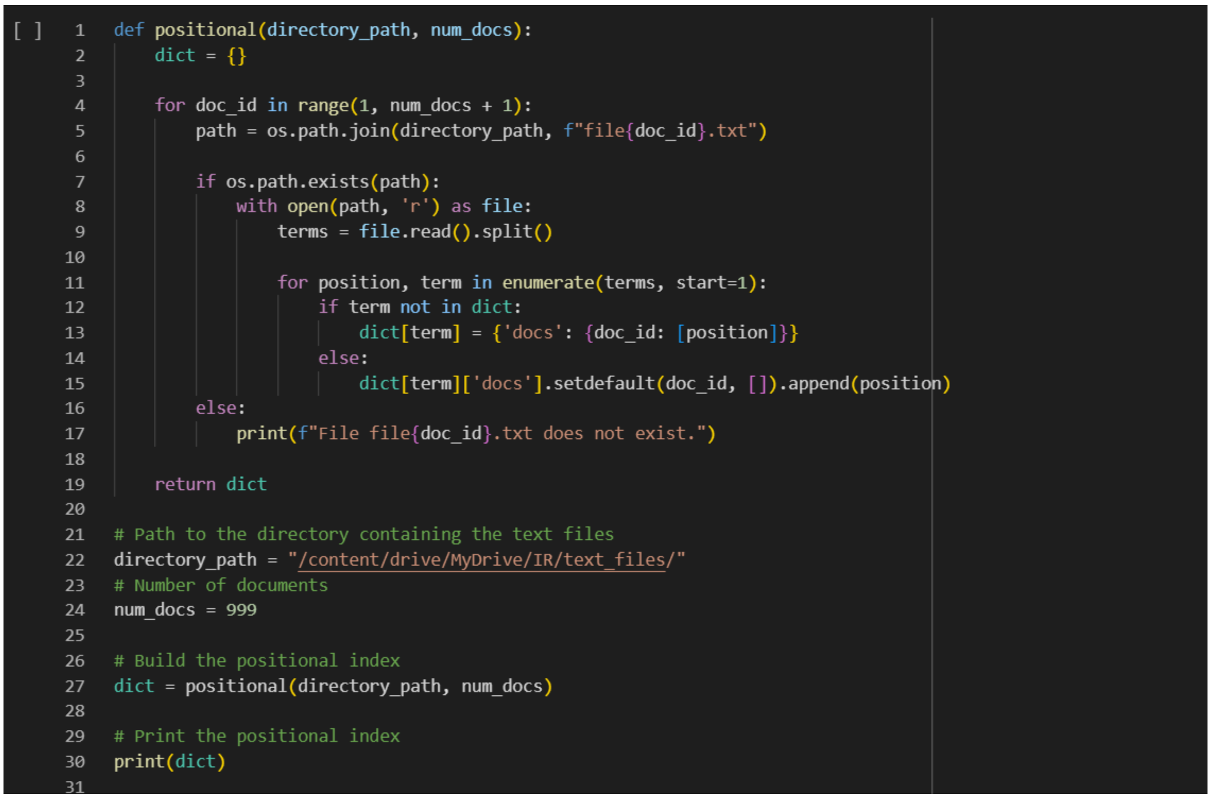
```python
def docs(query, operations, unigram_index, all_files):
    lem = WordNetLemmatizer()
    query = query.lower()
    query = re.sub('[^A-Z a-z ]+', ' ', query)
    query = query.split()
    tokens = [j for j in query]
    stop_words = set(stopwords.words("english"))
    tokens = [lem.lemmatize(token) for token in tokens if token not in stop_words and token not in string.punctuation and token.strip()]

    print(tokens)

    combined_elements = []
    for token, operation in zip(tokens, operations):
        combined_elements.extend([token, operation])
    combined_elements.extend(tokens[len(operations):] + operations[len(tokens):])
    query_ans = ' '.join(combined_elements)

    query_tokens = tokens

    res_set = [list(unigram_index[token]) for token in query_tokens if token in unigram_index]

    result = []
    compare = 0
    i = 0

    for operations in operations:
        if res_set and i < len(res_set):
            # print(res_set)
            if operations == "AND":
                if i == 0:
```



```python
def file_names(num_files):
    return [f"file{i + 1}.txt" for i in range(num_files)]

def process_query(query, operations, inverted_index, files):
    result, number_of_comparisons, query_ans = docs(query, operations, inverted_index, files)
    query_statement = " ".join(operations)
    sorted_result = sorted(result)
    return query_statement, len(result), sorted_result, query_ans

if __name__ == "__main__":
    n = int(input("Enter the number of queries: ").strip())
    files = file_names(999)

    for i in range(n):
        query = input(f"Enter query {i + 1}: ").strip()
        operations = input("Enter operations separated by commas: ").strip().split(",")

        query_statement, num_retrieved, retrieved_names, query_ans = process_query(query, operations, loaded_unigram_inverted_index, file

        # print(f"\nQuery {i + 1}: {query_statement}")
        print(f"\nQuery {i + 1}: {query_ans}")

        print("Number of documents retrieved for query", i + 1, ":", num_retrieved)
        print("Names of the documents retrieved for query", i + 1, ":", retrieved_names)
        print()
```

This Python code defines functions for basic information retrieval operations on a set of text documents. The functions include `union`, `intersect`, `complement`, `andd`, and `orr` for set operations. The `docs` function processes a user query, tokenizes and lemmatizes it, and then applies the specified operations on the inverted index of the documents. The program takes user input for the number of queries, and for each query, it prompts the user to enter a search query and corresponding operations (e.g., AND, OR) separated by commas. The `process_query` function utilizes the `docs` function to execute the query, and the results, including the query statement, the number of retrieved documents, and their names, are then printed for each query. The code demonstrates a basic information retrieval system with set-based query operations on a collection of documents.

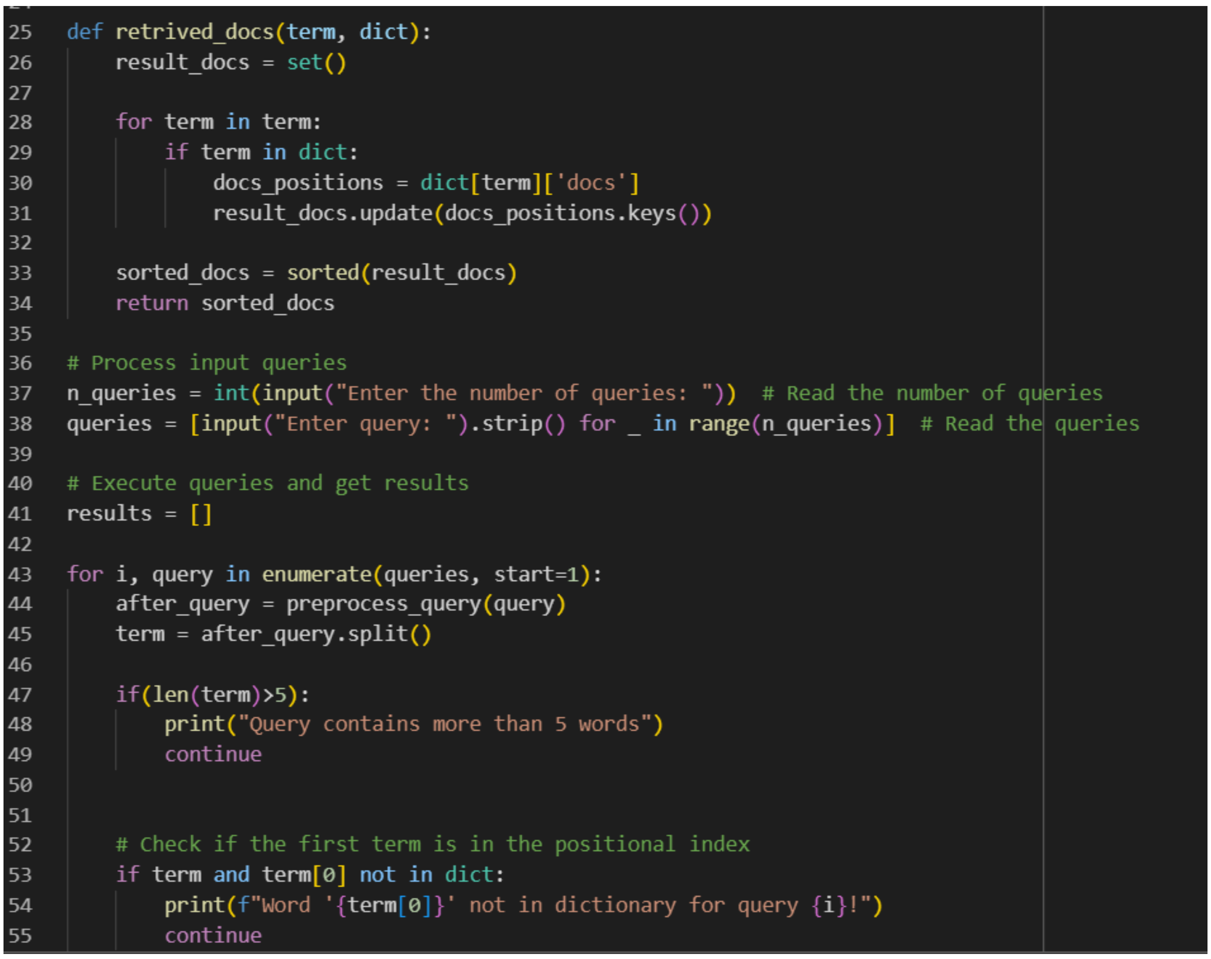
Q3) Positional Indexing



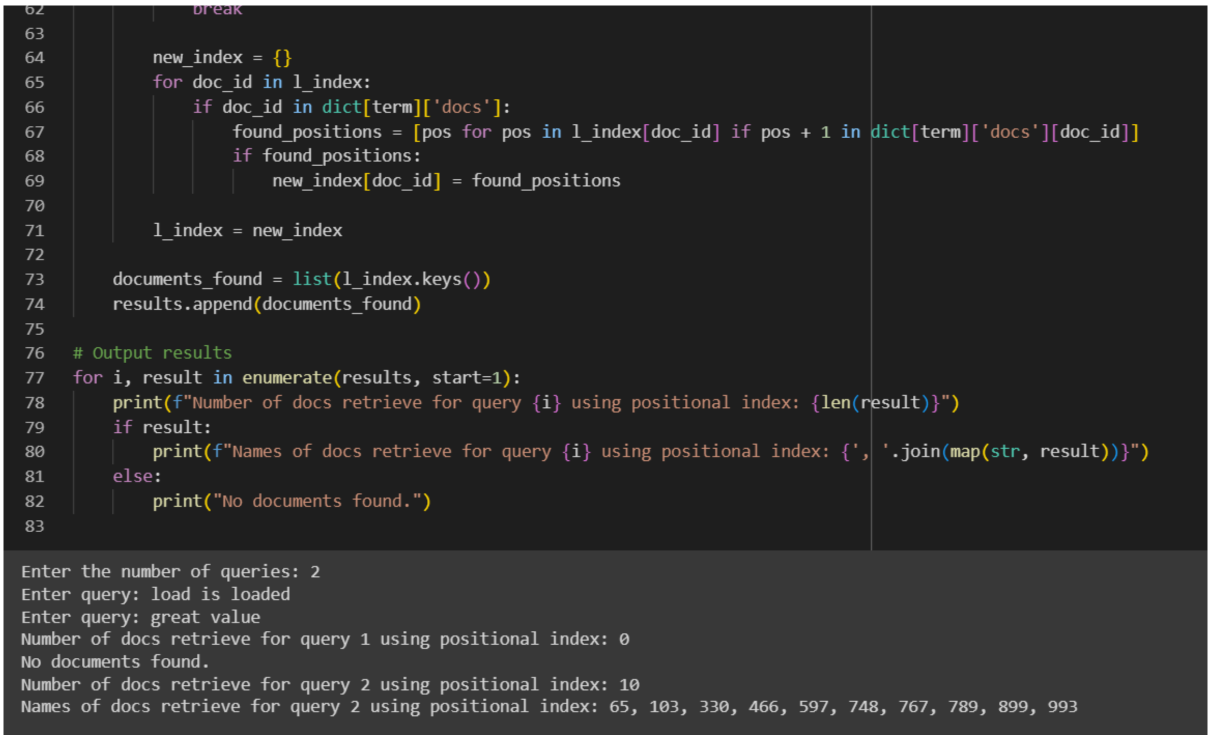
```python
def positional(directory_path, num_docs):
    dict = {}

    for doc_id in range(1, num_docs + 1):
        path = os.path.join(directory_path, f"file{doc_id}.txt")

        if os.path.exists(path):
            with open(path, 'r') as file:
                terms = file.read().split()

                for position, term in enumerate(terms, start=1):
                    if term not in dict:
                        dict[term] = {'docs': {doc_id: [position]}}
                    else:
                        dict[term]['docs'].setdefault(doc_id, []).append(position)
        else:
            print(f"File file{doc_id}.txt does not exist.")

    return dict

# Path to the directory containing the text files
directory_path = "/content/drive/MyDrive/IR/text_files/"
# Number of documents
num_docs = 999

# Build the positional index
dict = positional(directory_path, num_docs)

# Print the positional index
print(dict)

```

This Python code defines a function positional that builds a positional index for a set of text documents. The function takes two parameters: directory_path, which is the path to the directory containing the text files, and num_docs, which is the total number of documents to be processed.

The code iterates through each document (indexed from 1 to num_docs) in the specified directory. For each document, it reads the content, tokenizes it into terms, and then constructs a positional index. The positional index is stored in a dictionary (dict), where each term is a key. For each term, the corresponding value is another dictionary with the key 'docs' representing the document IDs and their associated positions within the document.If a term is encountered for the first time, a new entry is added to the main dictionary with the term as the key. If the term already exists, the document ID and position information are appended to the existing entry.



```python
def retrived_docs(term, dict):
    result_docs = set()

    for term in term:
        if term in dict:
            docs_positions = dict[term]['docs']
            result_docs.update(docs_positions.keys())

    sorted_docs = sorted(result_docs)
    return sorted_docs

# Process input queries
n_queries = int(input("Enter the number of queries: "))  # Read the number of queries
queries = [input("Enter query: ").strip() for _ in range(n_queries)]  # Read the queries

# Execute queries and get results
results = []

for i, query in enumerate(queries, start=1):
    after_query = preprocess_query(query)
    term = after_query.split()

    if(len(term)>5):
        print("Query contains more than 5 words")
        continue


    # Check if the first term is in the positional index
    if term and term[0] not in dict:
        print(f"Word '{term[0]}' not in dictionary for query {i}!")
        continue
```



```python
            break

        new_index = {}
        for doc_id in l_index:
            if doc_id in dict[term]['docs']:
                found_positions = [pos for pos in l_index[doc_id] if pos + 1 in dict[term]['docs'][doc_id]]
                if found_positions:
                    new_index[doc_id] = found_positions

        l_index = new_index

    documents_found = list(l_index.keys())
    results.append(documents_found)

# Output results
for i, result in enumerate(results, start=1):
    print(f"Number of docs retrieve for query {i} using positional index: {len(result)}")
    if result:
        print(f"Names of docs retrieve for query {i} using positional index: {', '.join(map(str, result))}")
    else:
        print("No documents found.")
```

```
Enter the number of queries: 2
Enter query: load is loaded
Enter query: great value
Number of docs retrieve for query 1 using positional index: 0
No documents found.
Number of docs retrieve for query 2 using positional index: 10
Names of docs retrieve for query 2 using positional index: 65, 103, 330, 466, 597, 748, 767, 789, 899, 993
```

This Python code defines a query processing system using a positional index built on a set of text documents. The program first includes a function preprocess_query to clean and preprocess user input queries. This involves converting the query to lowercase, removing non-alphabetic characters, tokenizing, eliminating stop words and punctuation, and lemmatizing the tokens.

Another function, retrived_docs, is designed to retrieve documents based on the processed query terms from the positional index. It iterates through each term in the query, checking if it exists in the index, and then narrowing down the search by considering the positional information of each term in relation to the previous term. The retrieved documents are stored and returned.

The main part of the program takes user input for the number of queries, processes each query using the preprocess_query function, and then utilizes the retrived_docs function to retrieve relevant documents based on the positional index. The results are printed, including the number of retrieved documents and their names.