IR ASSIGNMENT 1

KANTILAL PRAVINBHAI PATEL

MT23063

Kantilal23063@iiitd.ac.in

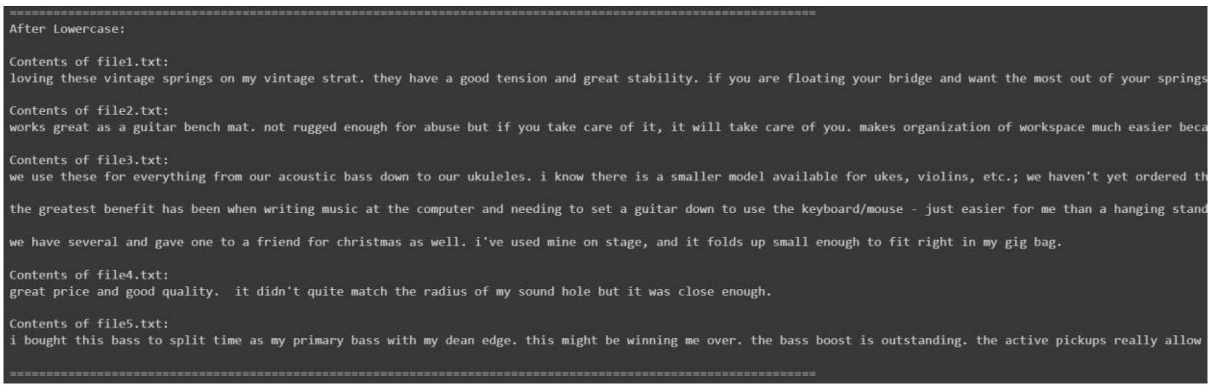
QUESTION 1: PREPROCESSING

In the first task of the Information Retrieval Assignment 1, They will be given a collection of 999 text files. The initial question requires you to preprocess these files and display sample outputs. The preprocessing involves five essential steps: converting the text to lowercase, breaking it into tokens, eliminating common stopwords, removing punctuation, and getting rid of space tokens. Once you've applied all these steps, the goal is to rewrite the content back into their respective files. To showcase progress, we will print out samples from five files, giving a clear view of the modified content.

Before performing the preprocessing steps some files contain some HTML tags so we will use the BeautifulSoup library to remove all Tags.

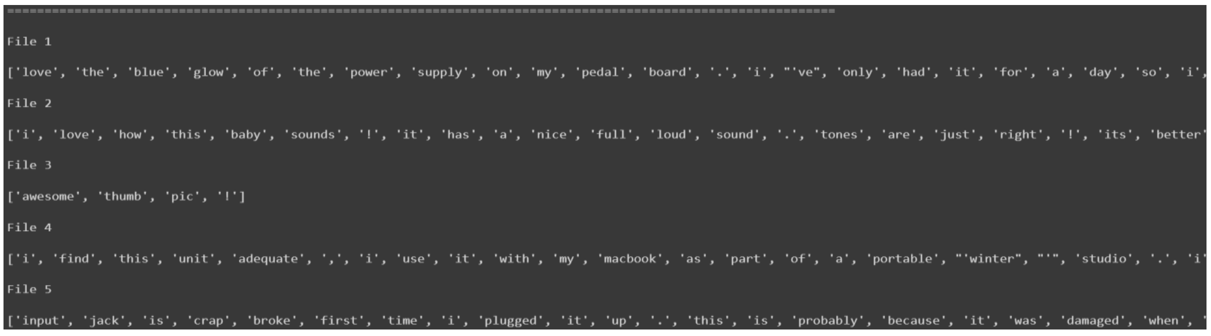
Details of all Five Preprocessing steps as below:

Lowercase the text: During this phase, we retrieve the contents of the text files individually and transform them to lowercase using the lower() method. The resulting output after this step appears as follows:



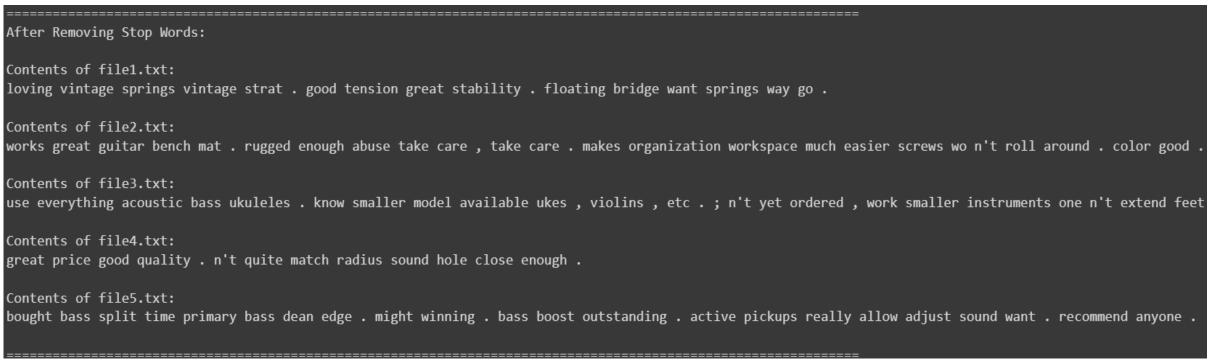
```
====================================================================================================
After Lowercase:

Contents of file1.txt:
loving these vintage springs on my vintage strat. they have a good tension and great stability. if you are floating your bridge and want the most out of your springs

Contents of file2.txt:
works great as a guitar bench mat. not rugged enough for abuse but if you take care of it, it will take care of you. makes organization of workspace much easier beca

Contents of file3.txt:
we use these for everything from our acoustic bass down to our ukuleles. i know there is a smaller model available for ukes, violins, etc.; we haven't yet ordered th

the greatest benefit has been when writing music at the computer and needing to set a guitar down to use the keyboard/mouse - just easier for me than a hanging stand

we have several and gave one to a friend for christmas as well. i've used mine on stage, and it folds up small enough to fit right in my gig bag.

Contents of file4.txt:
great price and good quality.  it didn't quite match the radius of my sound hole but it was close enough.

Contents of file5.txt:
i bought this bass to split time as my primary bass with my dean edge. this might be winning me over. the bass boost is outstanding. the active pickups really allow

====================================================================================================
```

Perform Tokenization: In this stage, we generate tokens for all the files utilizing the nltk library. To accomplish this, we employ the nltk.word_tokenize method. The resulting output after this step appears as follows:



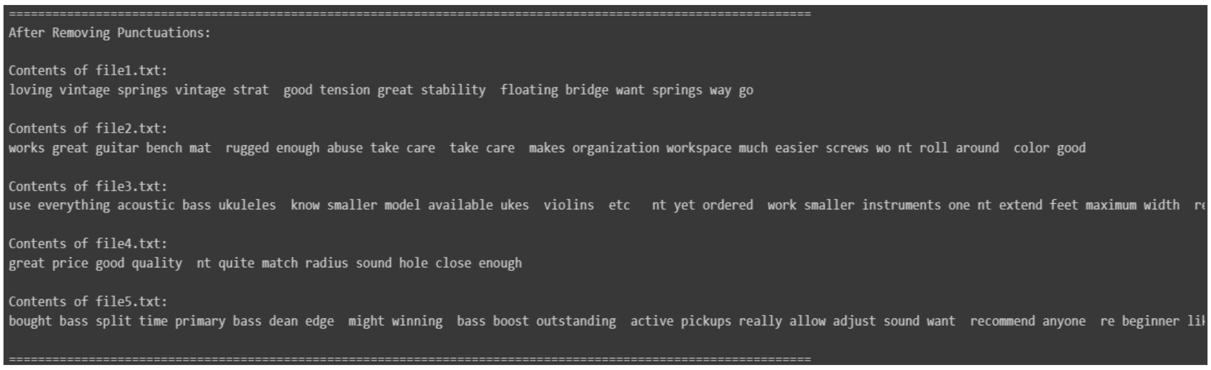
```
====================================================================================================

File 1

['love', 'the', 'blue', 'glow', 'of', 'the', 'power', 'supply', 'on', 'my', 'pedal', 'board', '.', 'i', "'ve", 'only', 'had', 'it', 'for', 'a', 'day', 'so', 'i',

File 2

['i', 'love', 'how', 'this', 'baby', 'sounds', '!', 'it', 'has', 'a', 'nice', 'full', 'loud', 'sound', '.', 'tones', 'are', 'just', 'right', '!', 'its', 'better'

File 3

['awesome', 'thumb', 'pic', '!']

File 4

['i', 'find', 'this', 'unit', 'adequate', ',', 'i', 'use', 'it', 'with', 'my', 'macbook', 'as', 'part', 'of', 'a', 'portable', "'winter", "'", 'studio', '.', 'i'

File 5

['input', 'jack', 'is', 'crap', 'broke', 'first', 'time', 'i', 'plugged', 'it', 'up', '.', 'this', 'is', 'probably', 'because', 'it', 'was', 'damaged', 'when', '
```

Remove stopwords: In this step, we aim to eliminate commonly used words, also known as stopwords. The resulting output after this step appears as follows:



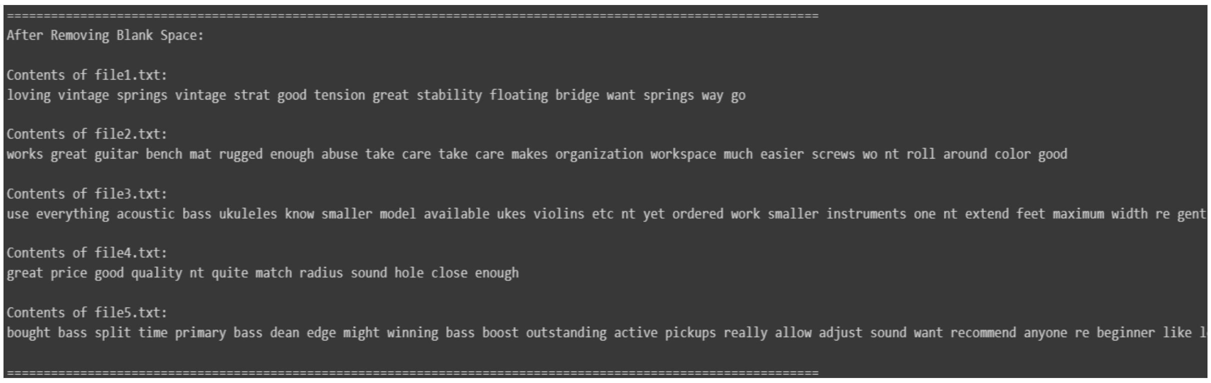
```
==========================================================================================================
After Removing Stop Words:

Contents of file1.txt:
loving vintage springs vintage strat . good tension great stability . floating bridge want springs way go .

Contents of file2.txt:
works great guitar bench mat . rugged enough abuse take care , take care . makes organization workspace much easier screws wo n't roll around . color good .

Contents of file3.txt:
use everything acoustic bass ukuleles . know smaller model available ukes , violins , etc . ; n't yet ordered , work smaller instruments one n't extend feet

Contents of file4.txt:
great price good quality . n't quite match radius sound hole close enough .

Contents of file5.txt:
bought bass split time primary bass dean edge . might winning . bass boost outstanding . active pickups really allow adjust sound want . recommend anyone .

==========================================================================================================
```

Remove punctuations: For remove punctuation we will use content.translate(str.maketrans('', '', string.punctuation)). The purpose of this code is to remove punctuation from the content string during the "remove_punctuations" preprocessing phase. It does so by utilizing a translation table that maps each punctuation character to None, effectively deleting those characters from the string. This step helps in cleaning the text and preparing it for further processing without considering punctuation marks. The resulting output after this step appears as follows:



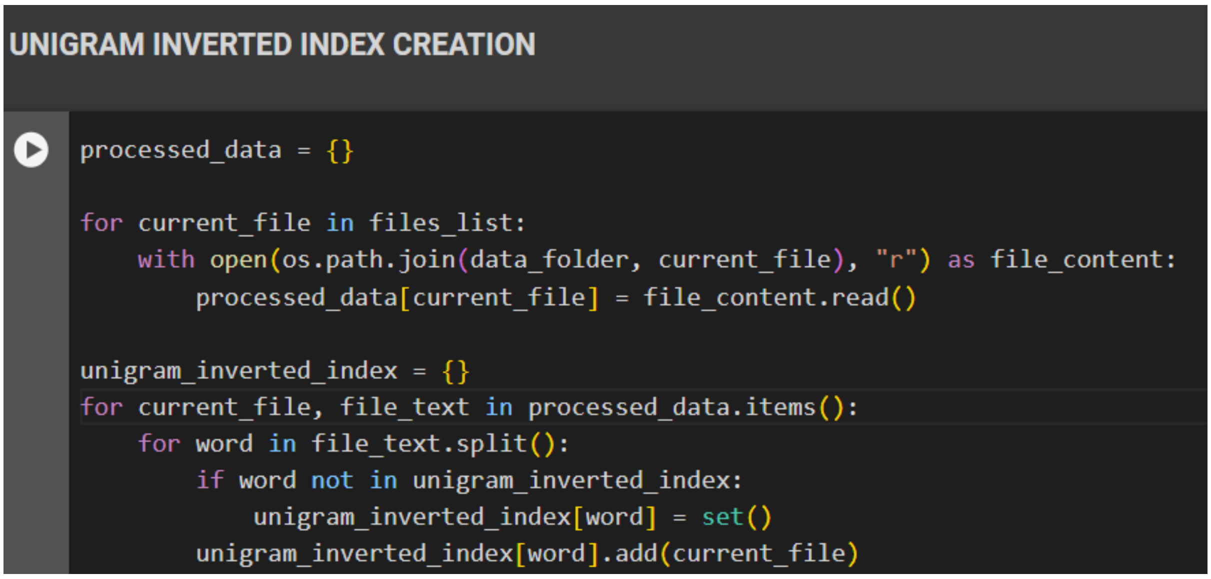
```
==========================================================================================================
After Removing Punctuations:

Contents of file1.txt:
loving vintage springs vintage strat  good tension great stability  floating bridge want springs way go

Contents of file2.txt:
works great guitar bench mat  rugged enough abuse take care  take care  makes organization workspace much easier screws wo nt roll around  color good

Contents of file3.txt:
use everything acoustic bass ukuleles  know smaller model available ukes  violins  etc   nt yet ordered  work smaller instruments one nt extend feet maximum width  re

Contents of file4.txt:
great price good quality  nt quite match radius sound hole close enough

Contents of file5.txt:
bought bass split time primary bass dean edge  might winning  bass boost outstanding  active pickups really allow adjust sound want  recommend anyone  re beginner lil

==========================================================================================================
```

Remove blank space tokens: In the "remove_blank" preprocessing phase, we streamline our text using the code `cleaned_content = re.sub(r"\s+", " ", content, flags=re.UNICODE)`. This clever bit of code, fueled by Python's `re` module, focuses on tackling consecutive whitespace characters within our content. By employing the regular expression `r"\s+"`, it identifies and targets sequences of whitespace, such as spaces or tabs. The subsequent use of `re.sub(...)` then replaces these sequences with a single space.



```
==========================================================================================================
After Removing Blank Space:

Contents of file1.txt:
loving vintage springs vintage strat good tension great stability floating bridge want springs way go

Contents of file2.txt:
works great guitar bench mat rugged enough abuse take care take care makes organization workspace much easier screws wo nt roll around color good

Contents of file3.txt:
use everything acoustic bass ukuleles know smaller model available ukes violins etc nt yet ordered work smaller instruments one nt extend feet maximum width re gent

Contents of file4.txt:
great price good quality nt quite match radius sound hole close enough

Contents of file5.txt:
bought bass split time primary bass dean edge might winning bass boost outstanding active pickups really allow adjust sound want recommend anyone re beginner like l

==========================================================================================================
```

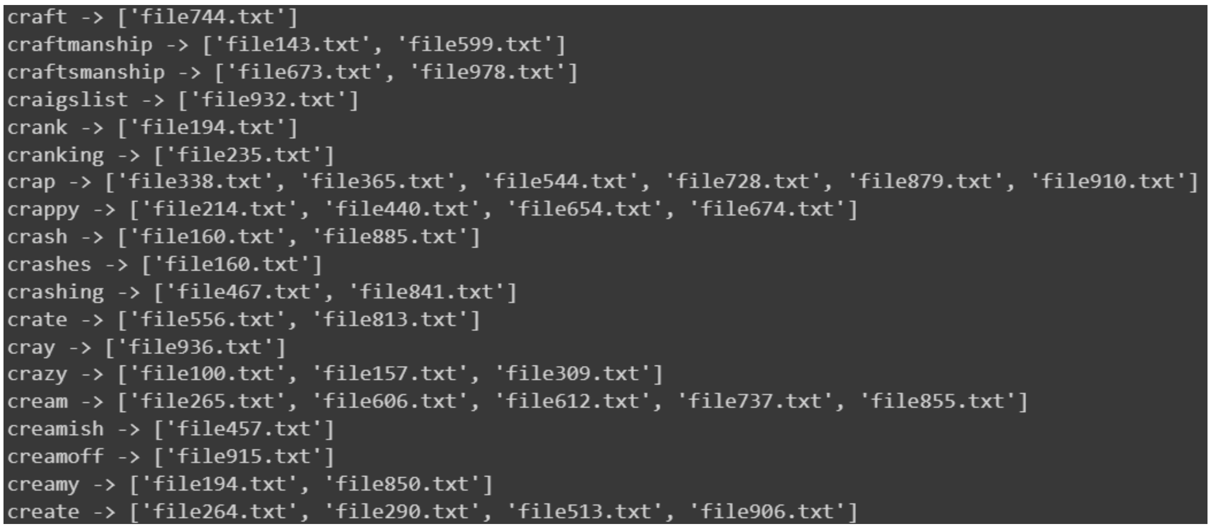
QUESTION 2: UNIGRAM INVERTED INDEX



UNIGRAM INVERTED INDEX CREATION

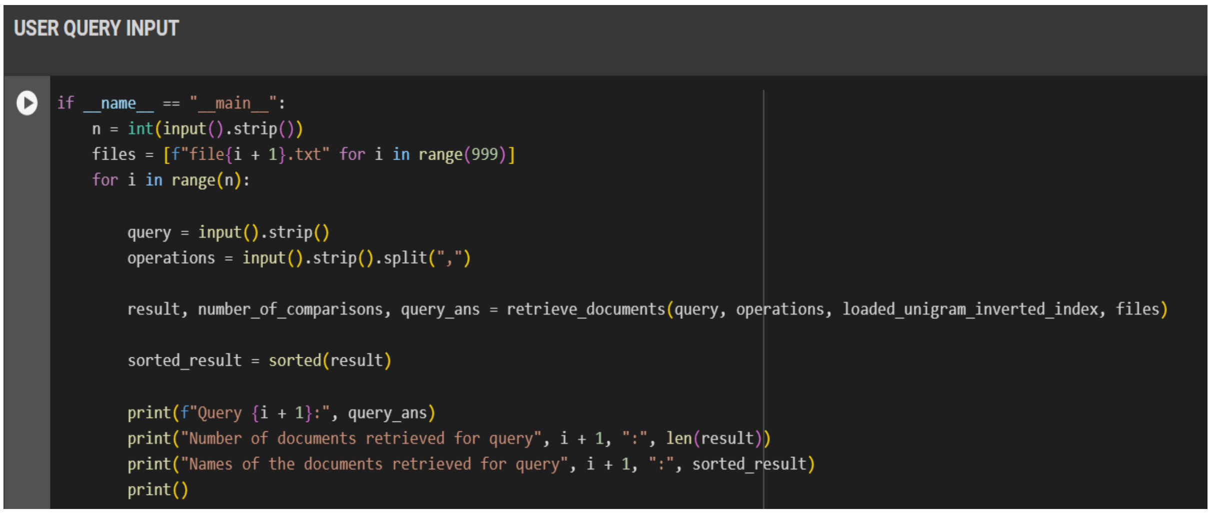
```python
processed_data = {}

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

unigram_inverted_index = {}
for current_file, file_text in processed_data.items():
    for word in file_text.split():
        if word not in unigram_inverted_index:
            unigram_inverted_index[word] = set()
        unigram_inverted_index[word].add(current_file)
```

For creating the unigram inverted index, we have already preprocessed all text files. Initially, an empty dictionary named `processed_data` is established to store the content of each file. The code then iterates through a list of file names, opening and reading the content of each file before storing it in the `processed_data` dictionary with the respective file name as the key. Subsequently, a new dictionary, `unigram_inverted_index`, is introduced to build the unigram inverted index. As the code traverses the files and their corresponding text in `processed_data`, it strategically examines each word within the text. If a word is not already present in the `unigram_inverted_index`, it is added with an associated empty set. Regardless of whether the word is new or existing, the code then appends the current file name to the set associated with that particular word. In essence, this process creates a structured index where each word is linked to a set of files in which it occurs, facilitating efficient retrieval of information about word occurrences in the given files. Created unigram_inverted_index as below:



```
craft -> ['file744.txt']
craftmanship -> ['file143.txt', 'file599.txt']
craftsmanship -> ['file673.txt', 'file978.txt']
craigslist -> ['file932.txt']
crank -> ['file194.txt']
cranking -> ['file235.txt']
crap -> ['file338.txt', 'file365.txt', 'file544.txt', 'file728.txt', 'file879.txt', 'file910.txt']
crappy -> ['file214.txt', 'file440.txt', 'file654.txt', 'file674.txt']
crash -> ['file160.txt', 'file885.txt']
crashes -> ['file160.txt']
crashing -> ['file467.txt', 'file841.txt']
crate -> ['file556.txt', 'file813.txt']
cray -> ['file936.txt']
crazy -> ['file100.txt', 'file157.txt', 'file309.txt']
cream -> ['file265.txt', 'file606.txt', 'file612.txt', 'file737.txt', 'file855.txt']
creamish -> ['file457.txt']
creamoff -> ['file915.txt']
creamy -> ['file194.txt', 'file850.txt']
create -> ['file264.txt', 'file290.txt', 'file513.txt', 'file906.txt']
```

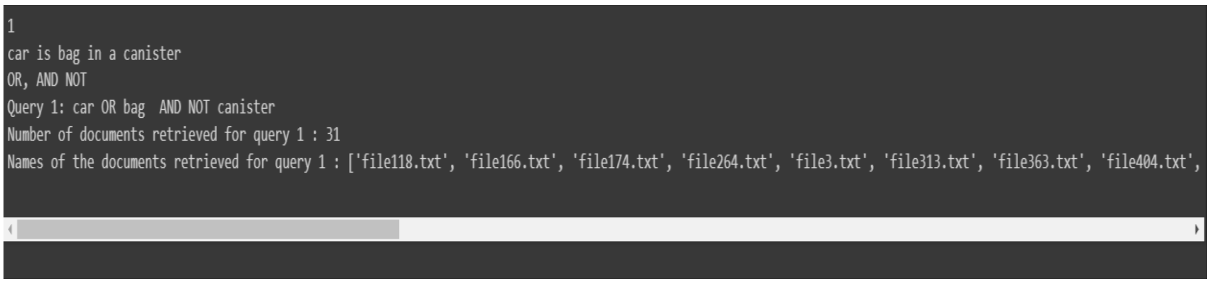
After constructing the `unigram_inverted_index`, the code serializes and stores it in a file named `unigram_inverted_index.pkl` using the pickle module. This facilitates efficient storage and retrieval. Subsequently, a crucial function is implemented to handle Boolean query operations, including AND, OR, AND NOT, and OR NOT. This functionality allows users to refine search queries, enhancing the flexibility of the information retrieval system.



USER QUERY INPUT

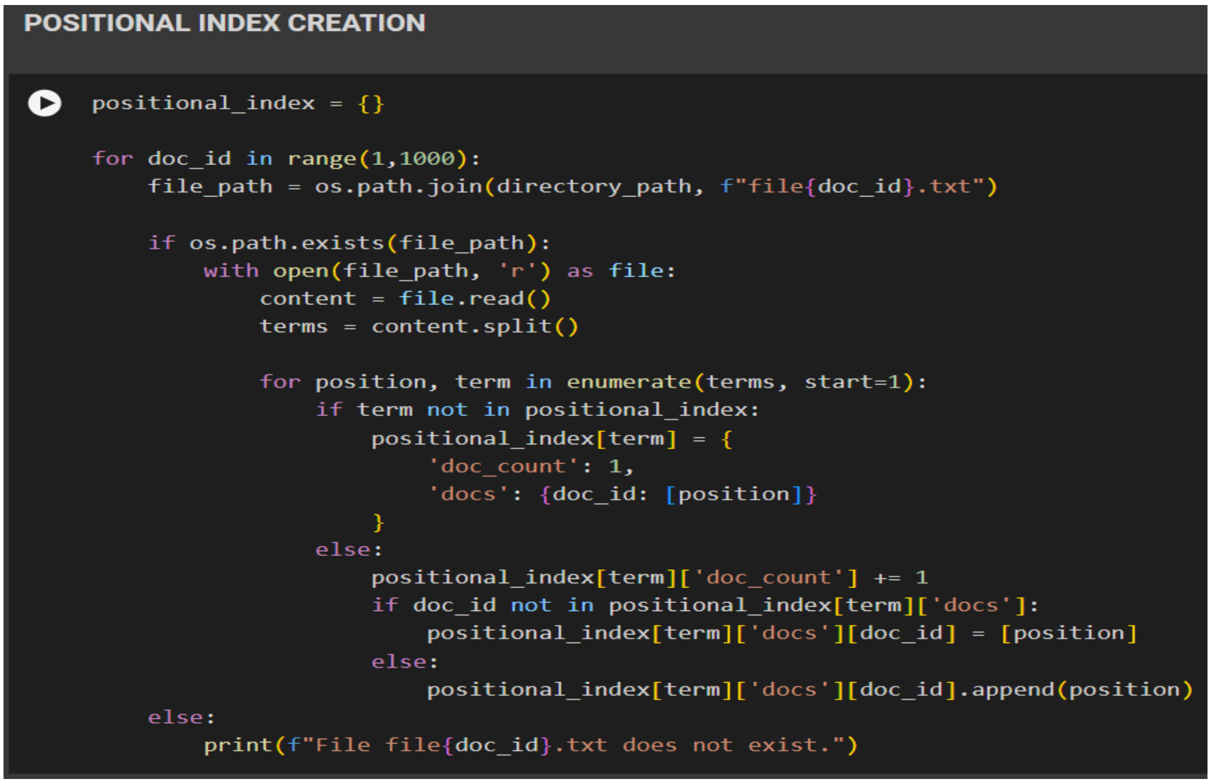
```
if __name__ == "__main__":
    n = int(input().strip())
    files = [f"file{i + 1}.txt" for i in range(999)]
    for i in range(n):

        query = input().strip()
        operations = input().strip().split(",")

        result, number_of_comparisons, query_ans = retrieve_documents(query, operations, loaded_unigram_inverted_index, files)

        sorted_result = sorted(result)

        print(f"Query {i + 1}:", query_ans)
        print("Number of documents retrieved for query", i + 1, ":", len(result))
        print("Names of the documents retrieved for query", i + 1, ":", sorted_result)
        print()
```

In the above code section, we kick things off by asking the user how many Boolean queries they'd like to make. Following that, we dive into the interactive part, where users input their query and specify the Boolean operation, they want to perform on it. Once we've processed the entire query, we share the results – the number of documents retrieved for that specific query and a neat list of all these documents. One Demo of the Whole output is below:



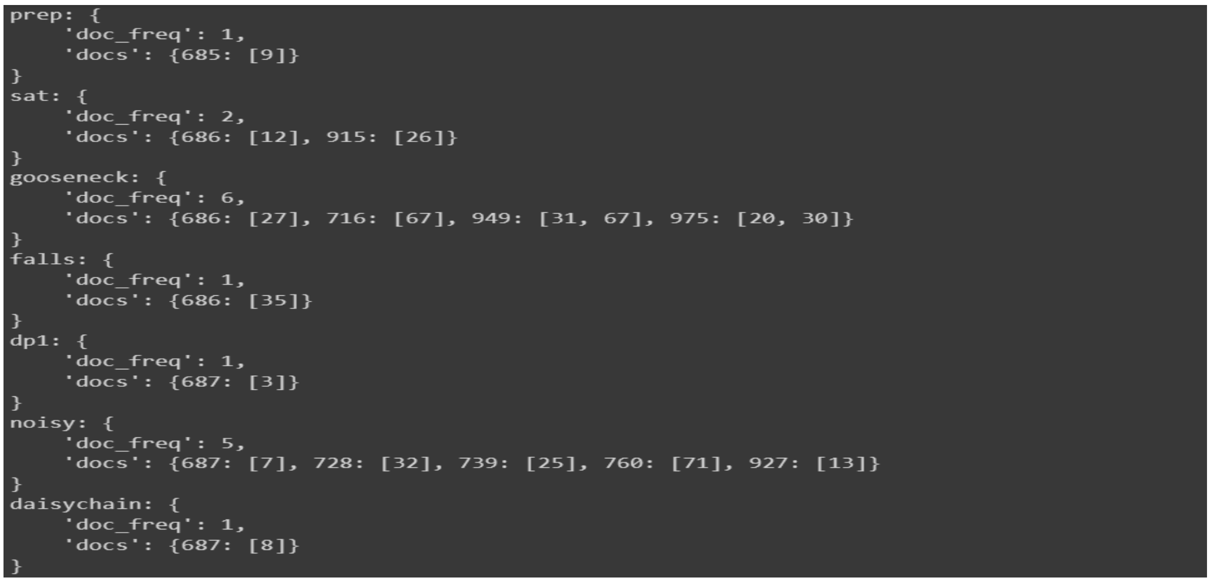
```
1
car is bag in a canister
OR, AND NOT
Query 1: car OR bag  AND NOT canister
Number of documents retrieved for query 1 : 31
Names of the documents retrieved for query 1 : ['file118.txt', 'file166.txt', 'file174.txt', 'file264.txt', 'file3.txt', 'file313.txt', 'file363.txt', 'file404.txt',
```

QUESTION 3: POSITIONAL INDEX


POSITIONAL INDEX CREATION

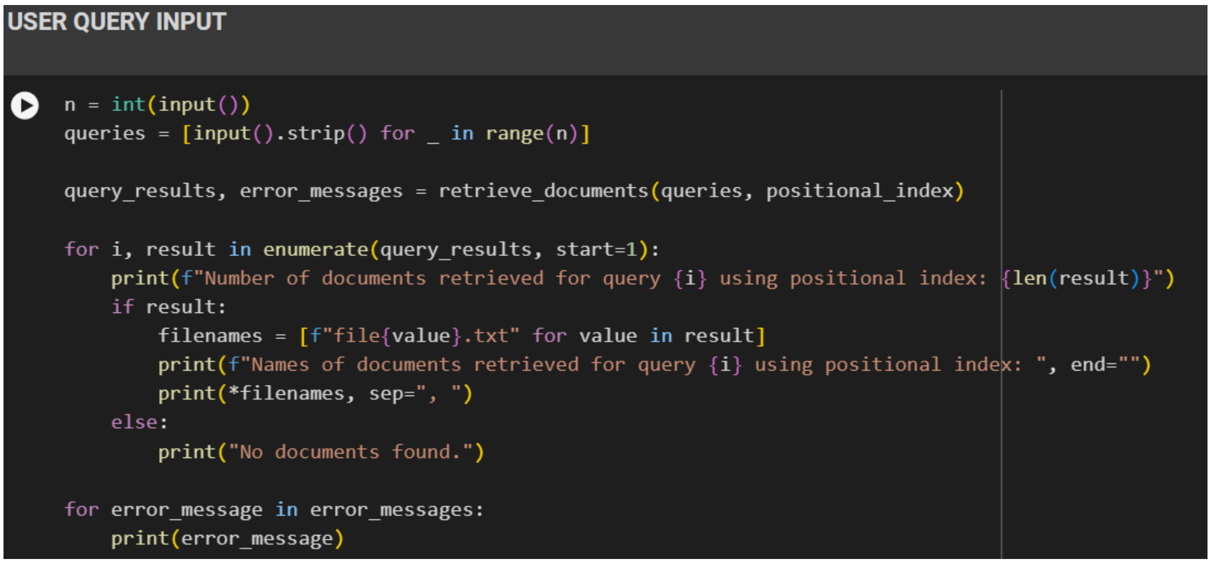
```python
positional_index = {}

for doc_id in range(1,1000):
    file_path = os.path.join(directory_path, f"file{doc_id}.txt")

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

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

For the creation of a positional index, the code processes a series of text documents (file1.txt to file999.txt) in a specified directory. It iterates through document IDs, reads each file's content, and tokenizes it into terms. The positional index is a dictionary where terms act as keys. For a new term, a corresponding entry is established with the term's document count set to 1 and a sub-dictionary ('docs') mapping document IDs to lists of term positions within each document. If the term already exists, the document count is incremented, and the current document ID is added to the 'docs' sub-dictionary with the term's position. If a document file is missing, a notification is printed. In essence, this code efficiently compiles a positional index, capturing term occurrences and positions across multiple documents for subsequent retrieval and analysis. Created positional index as below:



```
prep: {
    'doc_freq': 1,
    'docs': {685: [9]}
}
sat: {
    'doc_freq': 2,
    'docs': {686: [12], 915: [26]}
}
gooseneck: {
    'doc_freq': 6,
    'docs': {686: [27], 716: [67], 949: [31, 67], 975: [20, 30]}
}
falls: {
    'doc_freq': 1,
    'docs': {686: [35]}
}
dp1: {
    'doc_freq': 1,
    'docs': {687: [3]}
}
noisy: {
    'doc_freq': 5,
    'docs': {687: [7], 728: [32], 739: [25], 760: [71], 927: [13]}
}
daisychain: {
    'doc_freq': 1,
    'docs': {687: [8]}
}
```

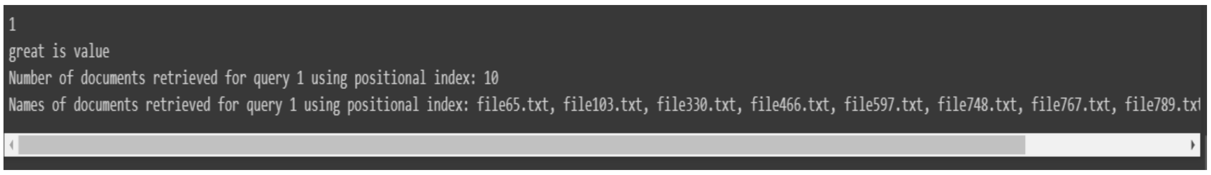
After constructing the positional index, the code serializes and stores it in a file named `positional_index.pkl` using the pickle module. This facilitates efficient storage and retrieval.



USER QUERY INPUT

```
n = int(input())
queries = [input().strip() for _ in range(n)]

query_results, error_messages = retrieve_documents(queries, positional_index)

for i, result in enumerate(query_results, start=1):
    print(f"Number of documents retrieved for query {i} using positional index: {len(result)}")
    if result:
        filenames = [f"file{value}.txt" for value in result]
        print(f"Names of documents retrieved for query {i} using positional index: ", end="")
        print(*filenames, sep=", ")
    else:
        print("No documents found.")

for error_message in error_messages:
    print(error_message)
```

This code processes a user-defined number of queries and retrieves relevant documents based on a positional index. It utilizes the 'retrieve_documents' function to obtain two lists: 'query_results' containing document IDs matching the queries and 'error_messages' capturing any encountered issues. The code then prints the number and names of retrieved documents for each query, or a "No documents found" message if none match. Overall, the script enables user interaction, query-based document retrieval, and informative output, encompassing both successful results and error messages. One Demo of the Whole output is below:



```
1
great is value
Number of documents retrieved for query 1 using positional index: 10
Names of documents retrieved for query 1 using positional index: file65.txt, file103.txt, file330.txt, file466.txt, file597.txt, file748.txt, file767.txt, file789.txt
```