AIWR ASSIGNMENT 1 AND 2
BASIC SEARCH ENGINE IMPLEMENTATION AND RECOMMENDER SYSTEM

PRAJWAL BS (PES1UG20CS287)
PRANAV KALWAD (PES1UG20CS291)
PURVIK S NAKUL (PES1UG20CS315)
RAHUL RANGANATH (PES1UG20CS316)

Title: Search engine and recommender system.

Introduction: This assignment includes implementation of basic search engine. It enables searching through use of Inverted index, positional index and similarity metrics. It supports Boolean queries, phrase queries, wild card queries and retrieval of top ranked documents. Additionally it also includes relevance feedback and semantic matching.
The second assignment we Preprocess the data, neighborhood based or Model based collaborative filtering for recommendation , Content based recommendation , Analyze the results , Use suitable evaluation metrics.

Dataset description:

Here is the link to the dataset that has been used for the 1st assignment (Building a search engine):

https://www.kaggle.com/datasets/ashishjangra27/imdb-top-250-movies

This dataset, available on Kaggle, contains information on the top 250 movies from the Internet Movie Database (IMDb) as of October 2021. The dataset includes information such as the movie title, director, year of release, runtime, genre, and the IMDb rating, along with the number of votes and reviews for each movie.

The dataset also includes a brief plot summary for each movie, as well as the cast and crew members involved in making the movie, such as actors, writers, and producers. Additionally, the dataset contains URLs linking to the IMDb page for each movie, where users can access further information, reviews, and ratings.

This dataset can be used for various purposes, such as conducting analyses of the top-rated movies across different genres, analyzing trends in movie ratings over time, or predicting the success of future movies based on certain characteristics.

The dataset contains the following attributes:

Ranking: Ranking of the movie based on the IMDb rating.

Title: Title of the movie.

IMDb Rating: The average user rating of the movie on IMDb, on a scale of 1 to 10.

Director: The director of the movie.

Cast: The main cast of the movie.

Year: The year in which the movie was released.

Genre: The genre of the movie.

Run time (Minutes): The duration of the movie in minutes.

Certificate: The certification of the movie.

Votes: The number of votes for the movie on IMDb.

Revenue (Millions): The revenue generated by the movie in millions of dollars.

Metascore: The Metacritic rating of the movie, on a scale of 0 to 100.

Synopsis: A brief plot summary of the movie.

```python
import nltk
import numpy as np
import pandas as pd
```

```python
nltk.download("punkt")
nltk.download("stopwords")
nltk.download("wordnet")
nltk.download('omw-1.4')
from nltk.tokenize import word_tokenize
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
```

```python
from google.colab import drive
drive.mount('/content/drive')
```

TOKENIZING THE **CORPUS**

```python
corpus = df["tagline"].values
```

```python
def tokenizing_corpus(type):
    return list(map(word_tokenize, type))
```

```python
corpus_tokenized = tokenizing_corpus(corpus)
```

```python
for i in range(250):
    print(f"Doc {i}: {corpus_tokenized[i]}")
```

```
Output exceeds the size limit. Open the full output data in a text editor
Doc 0: ['Fear', 'can', 'hold', 'you', 'prisoner', '.', 'Hope', 'can', 'set', 'you', 'free', '.']
Doc 1: ['An', 'offer', 'you', 'ca', "n't", 'refuse', '.']
Doc 2: ['Why', 'So', 'Serious', '?']
```

REMOVING THE STOPWORDS

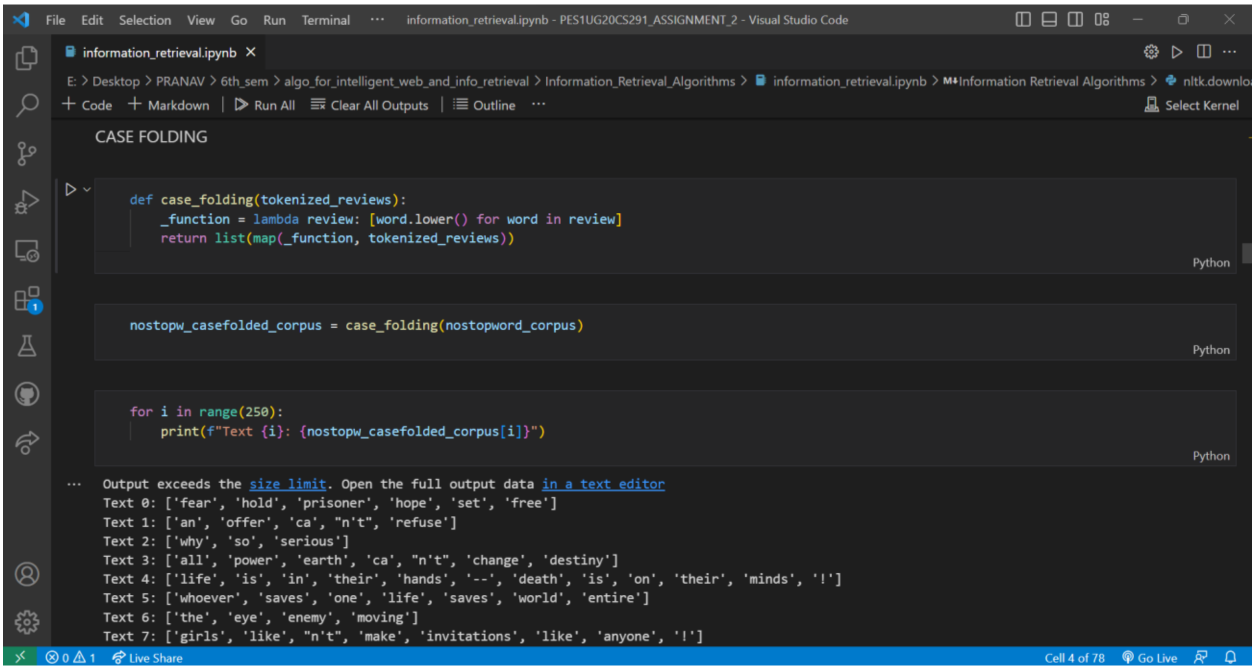
```python
verify=[',',':',';','.','#','(',')','{','[',']',"}",'/','?']
```

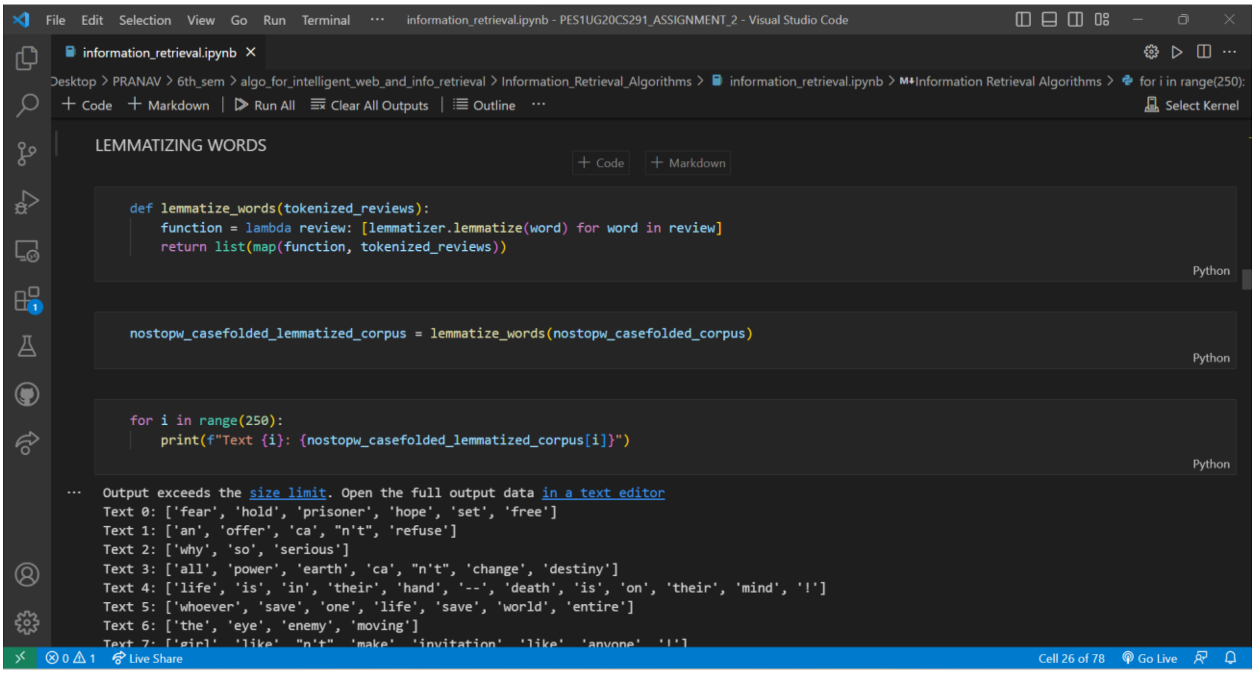
```python
def removing_stopwords(tokenized_review):
    _function = lambda review: [word for word in review if word not in stopwords and word not in verify]
    return list(map(_function, tokenized_review))
```

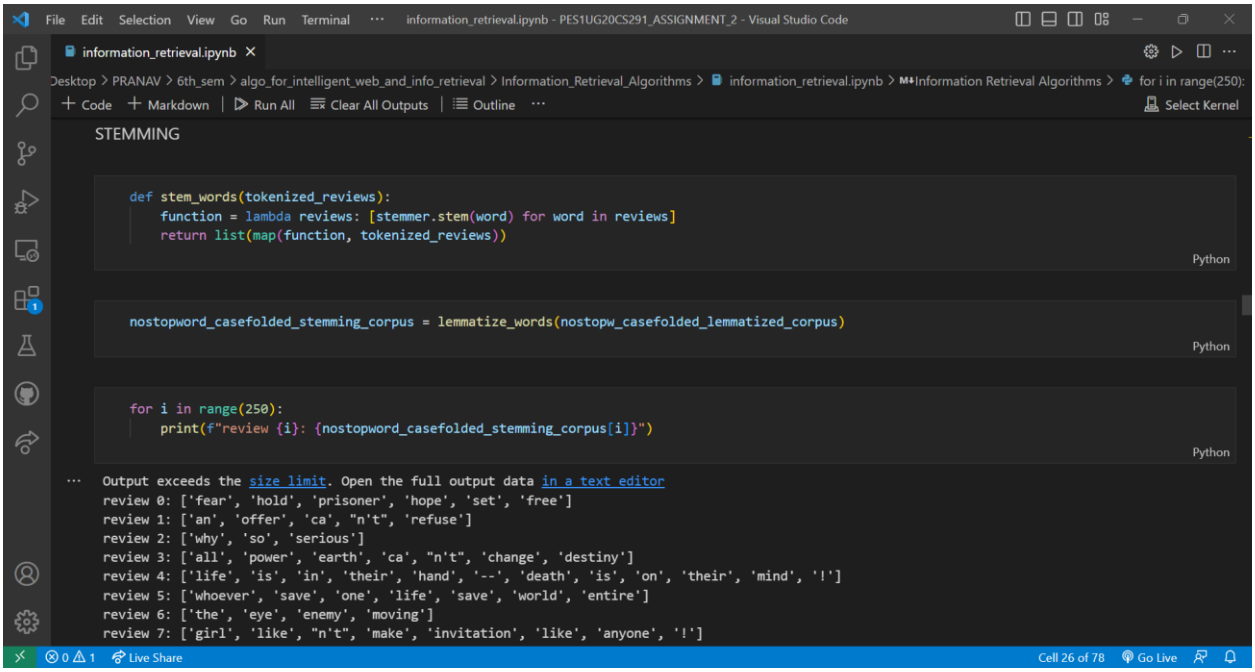
```python
nostopword_corpus = removing_stopwords(corpus_tokenized)
```

```python
for i in range(250):
    print(f"review {i}: {nostopword_corpus[i]}")
```

```
Output exceeds the size limit. Open the full output data in a text editor
review 0: ['Fear', 'hold', 'prisoner', 'Hope', 'set', 'free']
review 1: ['An', 'offer', 'ca', "n't", 'refuse']
```


File Edit Selection View Go Run Terminal ··· information_retrieval.ipynb - PES1UG20CS291_ASSIGNMENT_2 - Visual Studio Code
information_retrieval.ipynb
E: > Desktop > PRANAV > 6th_sem > algo_for_intelligent_web_and_info_retrieval > Information_Retrieval_Algorithms > information_retrieval.ipynb > M↓Information Retrieval Algorithms > nltk.downlo
+ Code + Markdown Run All Clear All Outputs Outline ···
Select Kernel
CASE FOLDING
def case_folding(tokenized_reviews):
    _function = lambda review: [word.lower() for word in review]
    return list(map(_function, tokenized_reviews))
Python
nostopw_casefolded_corpus = case_folding(nostopword_corpus)
Python
for i in range(250):
    print(f"Text {i}: {nostopw_casefolded_corpus[i]}")
Python
Output exceeds the size limit. Open the full output data in a text editor
Text 0: ['fear', 'hold', 'prisoner', 'hope', 'set', 'free']
Text 1: ['an', 'offer', 'ca', "n't", 'refuse']
Text 2: ['why', 'so', 'serious']
Text 3: ['all', 'power', 'earth', 'ca', "n't", 'change', 'destiny']
Text 4: ['life', 'is', 'in', 'their', 'hands', '--', 'death', 'is', 'on', 'their', 'minds', '!']
Text 5: ['whoever', 'saves', 'one', 'life', 'saves', 'world', 'entire']
Text 6: ['the', 'eye', 'enemy', 'moving']
Text 7: ['girls', 'like', "n't", 'make', 'invitations', 'like', 'anyone', '!']
⊗ 0 ⚠ 1 Live Share
Cell 4 of 78 Go Live


File Edit Selection View Go Run Terminal ··· information_retrieval.ipynb - PES1UG20CS291_ASSIGNMENT_2 - Visual Studio Code
information_retrieval.ipynb
Desktop > PRANAV > 6th_sem > algo_for_intelligent_web_and_info_retrieval > Information_Retrieval_Algorithms > information_retrieval.ipynb > M↓Information Retrieval Algorithms > for i in range(250):
+ Code + Markdown Run All Clear All Outputs Outline ···
Select Kernel
LEMMATIZING WORDS
+ Code + Markdown
def lemmatize_words(tokenized_reviews):
    function = lambda review: [lemmatizer.lemmatize(word) for word in review]
    return list(map(function, tokenized_reviews))
Python
nostopw_casefolded_lemmatized_corpus = lemmatize_words(nostopw_casefolded_corpus)
Python
for i in range(250):
    print(f"Text {i}: {nostopw_casefolded_lemmatized_corpus[i]}")
Python
Output exceeds the size limit. Open the full output data in a text editor
Text 0: ['fear', 'hold', 'prisoner', 'hope', 'set', 'free']
Text 1: ['an', 'offer', 'ca', "n't", 'refuse']
Text 2: ['why', 'so', 'serious']
Text 3: ['all', 'power', 'earth', 'ca', "n't", 'change', 'destiny']
Text 4: ['life', 'is', 'in', 'their', 'hand', '--', 'death', 'is', 'on', 'their', 'mind', '!']
Text 5: ['whoever', 'save', 'one', 'life', 'save', 'world', 'entire']
Text 6: ['the', 'eye', 'enemy', 'moving']
⊗ 0 ⚠ 1 Live Share
Cell 26 of 78 Go Live



Preprocessing: The following preprocessing techniques have been used:

Removing punctuation: Punctuation marks such as commas, periods, and question marks don't add much meaning to the text, and can be safely removed without losing important information.

Tokenization: Tokenization is the process of breaking up text into individual words, or tokens. This is useful because it allows us to treat each word as a separate unit, and perform operations on them independently.

Removing stopwords: Stopwords are commonly used words such as "the", "and", and "of" that don't carry much meaning on their own. Removing them helps to reduce noise in the text data and focus on the more meaningful words.

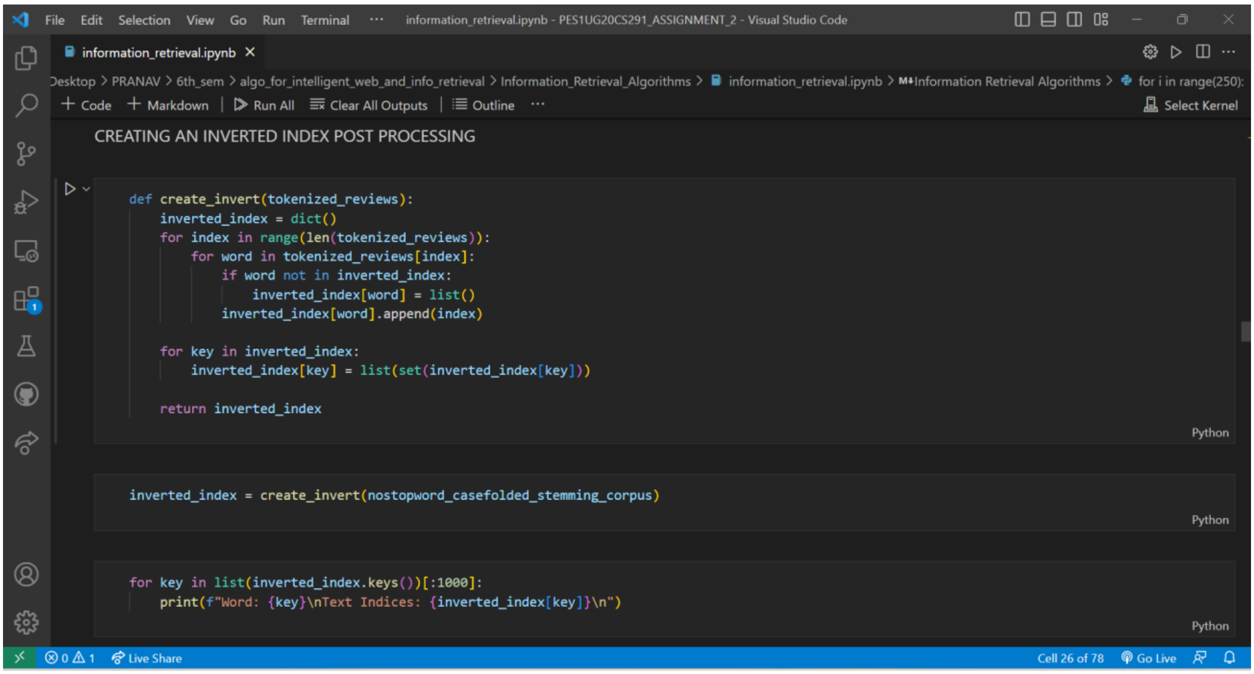
Stemming: Stemming is the process of reducing words to their root form, or stem. This is useful because it helps to reduce the number of unique words in the text data, and makes it easier to compare words that have the same root.

Inverted index generation:

We used a dict data structure to implement the inverted index. The dict creates a new set for any keythat is not already in the dictionary, which is useful for efficiently adding new postings to the inverted index.

We then loop through each file in the directory, read in the contents of the file, tokenizes the text into words, and updates the inverted index for each word by adding the current document ID to the set of postings for that word.

After processing each block of files, it saves the inverted index to a file using the pickle module, which can serialize and deserialize Python objects. Once all the blocks have been processed, the code combines the inverted indexes from each block into one final index file.



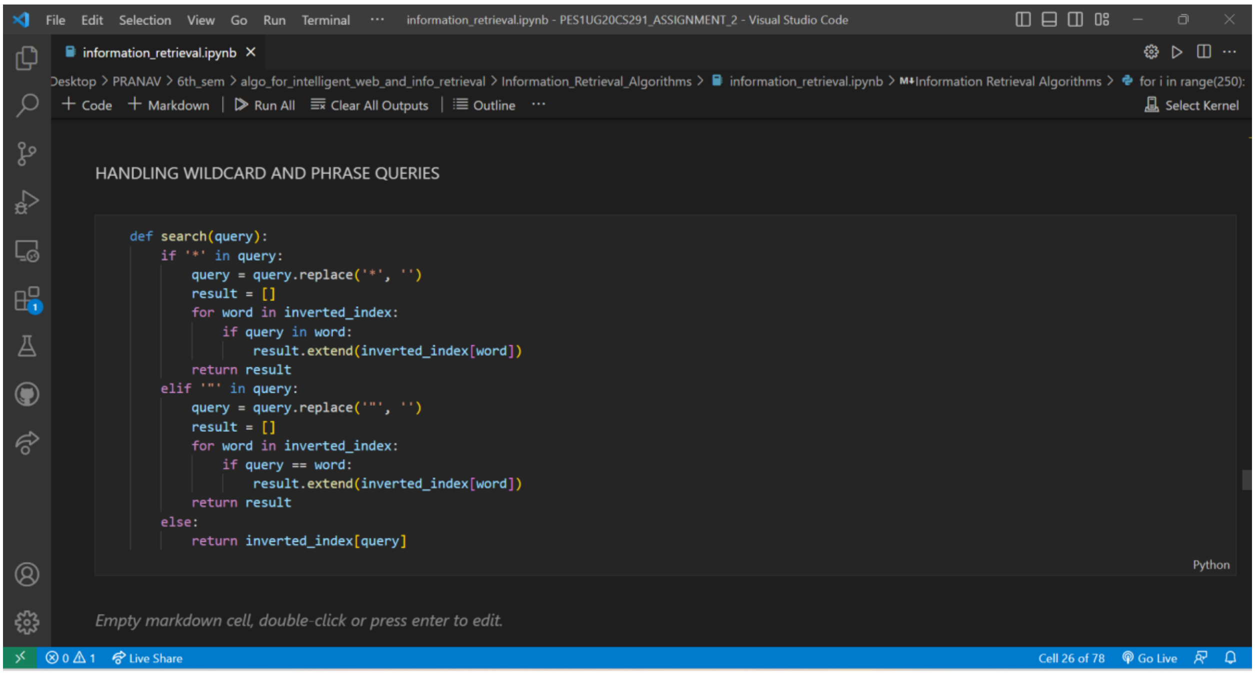
Wildcard queries:

The wildcard_query() function takes a query string as input and returns a set of matching documents. The query string can contain * and ? characters, which are treated as wildcards. The * character matches any number of characters, while the ? character matches any single character.

Inside the function, the query string is first transformed into a regular expression pattern using the replace() method of the string class. The .* and . characters are used to represent the * and ? wildcards respectively in the regular expression pattern.

Next, the regular expression pattern is compiled into a regular expression object using the re.compile() function from the Python re module. The regular expression object is used to match words in the index that match the query string.

The matching_words variable is a list of words that match the query string in the index. The list comprehension used here iterates over the keys of the final_index dictionary (which is not shown in the code snippet provided), and checks if each word matches the regular expression pattern using the regex.match() method of the regular expression object.



Finally, the matching_docs variable is a set of documents that contain any of the words in the matching_words list. The set.union() method is used to combine the document sets of all the matching words in the final_index dictionary.

Boolean queries:

Given a Boolean query, we can perform AND,OR etc by using the inverted index.

The function first parses the input query to obtain a list of tokens, and then further parses the query to obtain a list of boolean operators and operands. It then evaluates the boolean expression by iteratively popping operands from a stack, performing the appropriate operation, and pushing the result back onto the stack.

The 'AND' operator returns the intersection of two sets of document ids, 'OR' returns the union of two sets of document ids, and 'NOT' returns the complement of a set of document ids with respect to the set of all document ids.

BOOLEAN QUERIES

```python
def query_parsed(infix):

    order = {}
    order['('] = 0
    order[')'] = 0
    order['OR'] = 1
    order['NOT'] = 3
    order['AND'] = 2


    output = []
    stack = []

    for token in infix:
        if (token == '('):
            stack.append(token)

        elif (token == ')'):
            operator = stack.pop()
            while operator != '(':
                output.append(operator)
                operator = stack.pop()
```

```python
def boolean_query(query, inverted_index):
    query = query.strip()
    query_tokens = query.split()
    boolean_query = query_parsed(query_tokens)

    result_stack = list()
    for idx, token in enumerate(boolean_query):
        if token not in ["AND", "NOT", "OR"]:
            result = set(inverted_index[token])
        else:
            if token in ['AND', 'OR']:
                right_operand = result_stack.pop()
                left_operand = result_stack.pop()

                if token == 'AND':
                    operation = set.intersection
                else:
                    operation = set.union

                result = operation(left_operand, right_operand)

            else:
                operand = result_stack.pop()
                complement_document_ids = inverted_index[boolean_query[idx-1]]
                result = list()
                for word in inverted_index:
                    result.extend([_id for _id in inverted_index[word] if _id not in complement_document_ids])
                result = set(result)
```

testing with queries

```python
doc_id = boolean_query("Why AND So AND Serious", inverted_index)
print(f"Document IDs: {doc_id}")
```

Python

```
Document IDs: {2}
```

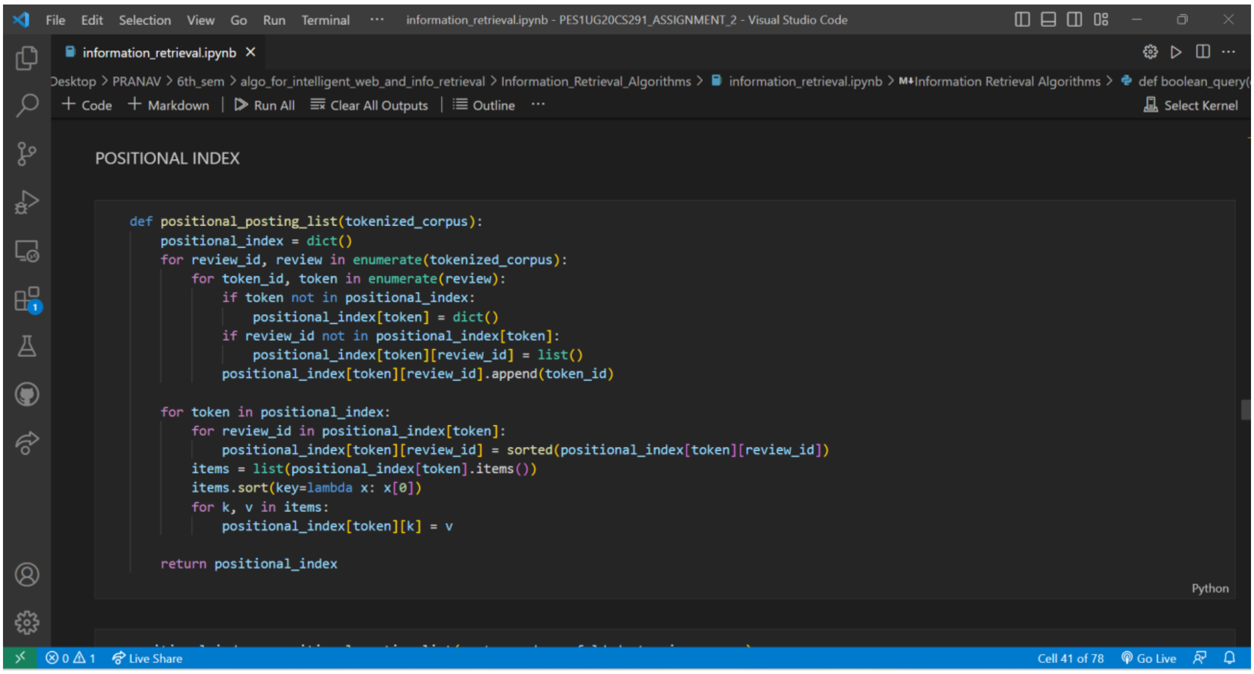
```python
doc_id_or = boolean_query("Your OR mind OR scene AND crime AND Free", inverted_index)
print(f"Document IDs: {doc_id_or}")
```

Python

```
Document IDs: {4, 21, 57, 13, 142, 15}
```

Positional index:

Code is almost same as inverted index, but in the values part of the dictionary datastructure, I am storing the doc Id and the position of the word in the documents as a tuple .



POSITIONAL INDEX

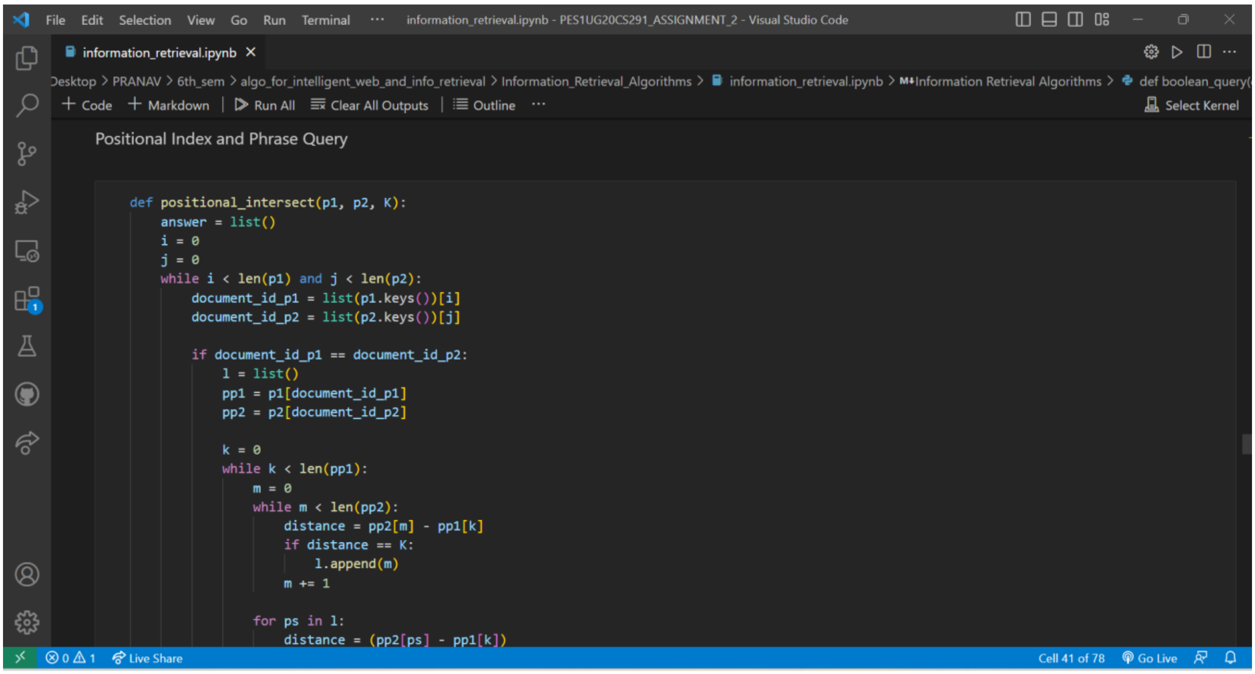
```python
def positional_posting_list(tokenized_corpus):
    positional_index = dict()
    for review_id, review in enumerate(tokenized_corpus):
        for token_id, token in enumerate(review):
            if token not in positional_index:
                positional_index[token] = dict()
            if review_id not in positional_index[token]:
                positional_index[token][review_id] = list()
            positional_index[token][review_id].append(token_id)

    for token in positional_index:
        for review_id in positional_index[token]:
            positional_index[token][review_id] = sorted(positional_index[token][review_id])
        items = list(positional_index[token].items())
        items.sort(key=lambda x: x[0])
        for k, v in items:
            positional_index[token][k] = v

    return positional_index
```

Phrase query:

Using the positional index and bi gram index, we implement phrase queries.



Positional Index and Phrase Query

```python
def positional_intersect(p1, p2, K):
    answer = list()
    i = 0
    j = 0
    while i < len(p1) and j < len(p2):
        document_id_p1 = list(p1.keys())[i]
        document_id_p2 = list(p2.keys())[j]

        if document_id_p1 == document_id_p2:
            l = list()
            pp1 = p1[document_id_p1]
            pp2 = p2[document_id_p2]

            k = 0
            while k < len(pp1):
                m = 0
                while m < len(pp2):
                    distance = pp2[m] - pp1[k]
                    if distance == K:
                        l.append(m)
                    m += 1

                for ps in l:
                    distance = (pp2[ps] - pp1[k])
```

checking if positional index works

```python
def search_positional_index(query):
    query = query.split()
    words = [query[i] for i in range(0, len(query), 2)]
    k = [int(query[i][1:]) for i in range(1, len(query), 2)]

    document_list = list()
    for i in range(0, len(words)-1):
        word1, word2 = words[i:i+2]
        p1 = positional_index[word1]
        p2 = positional_index[word2]
        result = positional_intersect(p1, p2, k[i])
        document_list.extend(result)

    return document_list
```

```python
search_positional_index("life /7 ")
```

```
[]
```

Retrieval using similarity score:

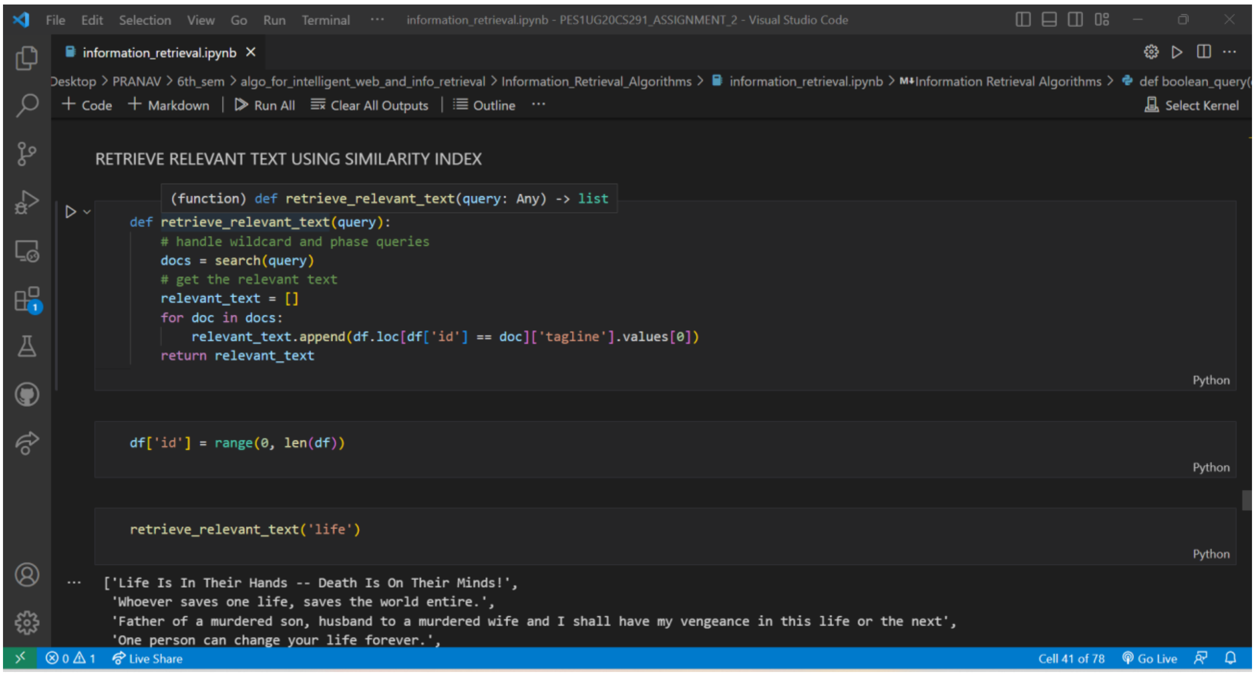
We have decided to use cosine similarity to implement this because of following reasons.

It is robust to document length: Cosine similarity takes into account the vector representation of documents, which means that it is able to compare documents of different lengths. This is important in information retrieval, where we often want to find relevant documents regardless of their length.

It ignores document size: Cosine similarity only measures the angle between two vectors and not their magnitudes. This means that it doesn't matter if one document is much larger than the other; the similarity score will only depend on the content of the documents.

It is computationally efficient: Cosine similarity can be calculated quickly and efficiently, even for large datasets. This makes it a practical method for real-world applications.

It is effective for text-based data: Cosine similarity is particularly well-suited for text-based data, as it takes into account the frequency of words in a document. This is important in information retrieval, as we want to find documents that contain the same or similar words to our query.

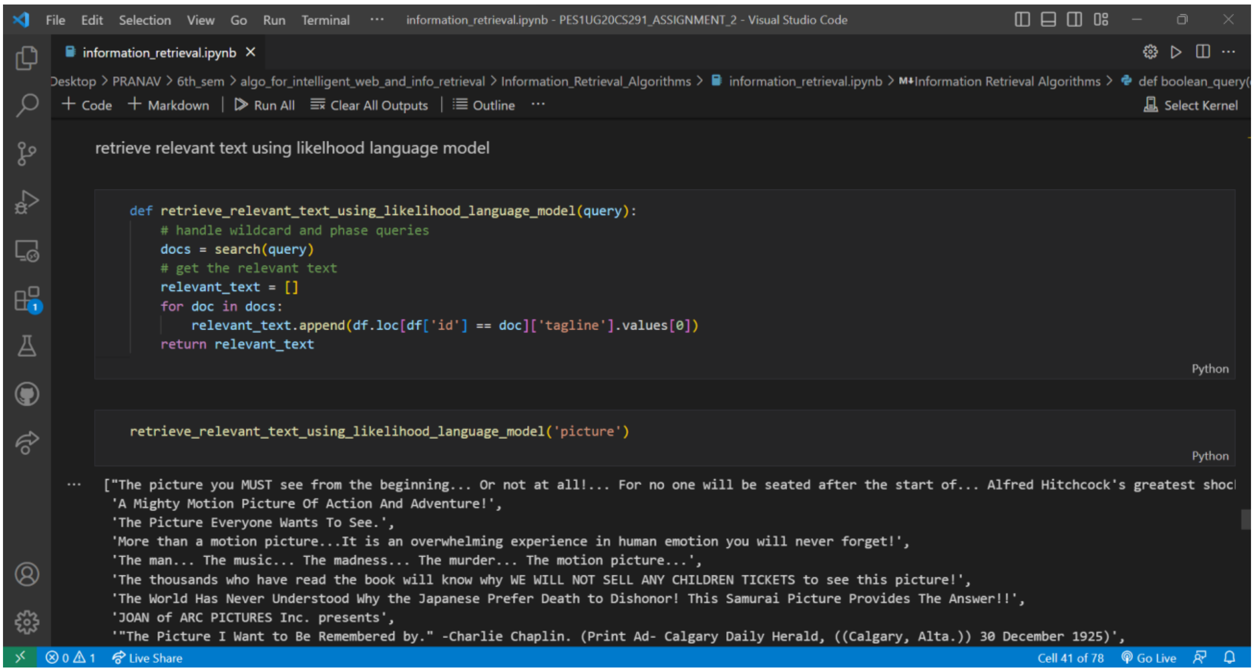


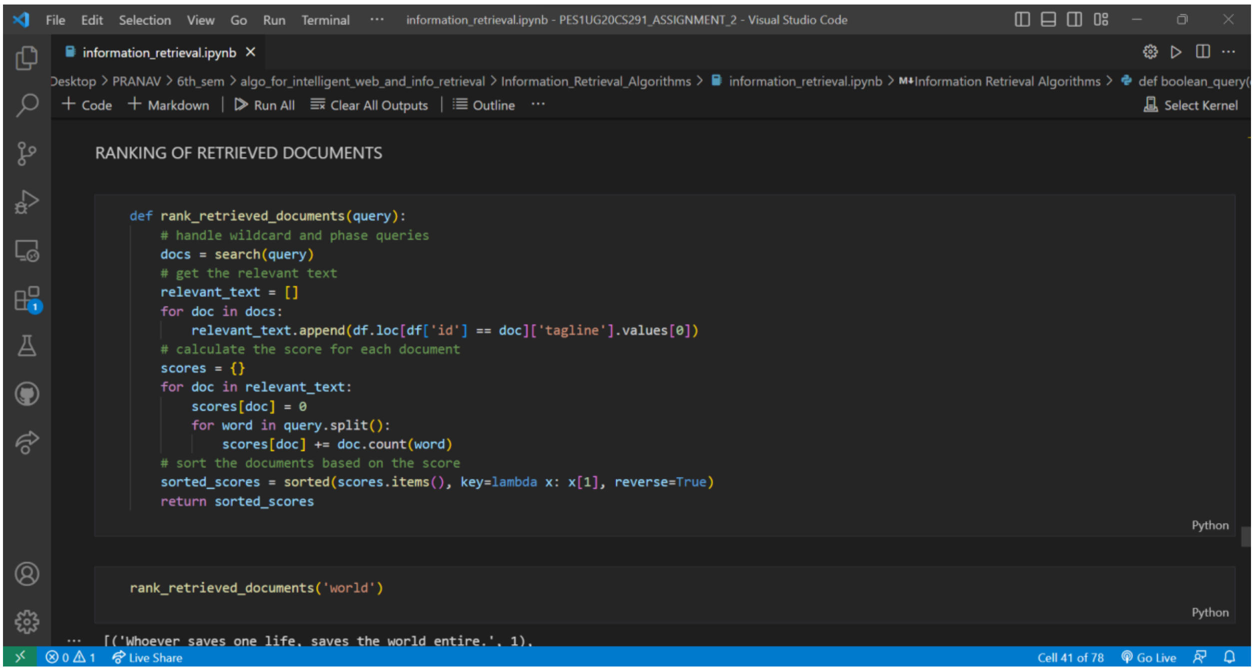
Next I am running cosine similarity to get most relevant documents to my search query.

Log- likelihood model:

Log likelihood is a commonly used measure in information retrieval to rank documents based on their relevance to a query. The log likelihood ratio (LLR) is a statistical measure that compares the likelihood of a term occurring in a document that is relevant to the query to the likelihood of the term occurring in a document that is not relevant to the query.

Log likelihood is used to retrieve top relevant documents is that it is a good indicator of the discriminatory power of a term in distinguishing relevant documents from irrelevant ones. A term that appears frequently in relevant documents and infrequently in irrelevant ones will have a high log likelihood score, indicating that it is a good indicator of relevance for the query.





Advanced search:

We have implemented relevance feedback and semantic matching in our assignment. Here is how :

Firstly we load a pre-trained English language model for semantic matching using spacy.

Next we do the same procedure as the cosine similarity between document and query.

Then we ask the user for feedback on the relevance of the top-k documents, and update the set of relevant documents accordingly.

We then performs relevance feedback by updating the query vector based on the relevant documents using a weighted sum of the original query and the average TF-IDF vector of the feedback documents.

We then performs semantic matching to expand the query based on related concepts using the NOUN part-of-speech tags of the top feedback_docs relevant documents. Specifically, the script extracts the first three nouns from each document, concatenates them with the original query, and uses the resulting string as the expanded query for the next iteration.

Print the final set of relevant documents.

Advanced search: relevance feedback, semantic matching, reranking of results, finding out query intention

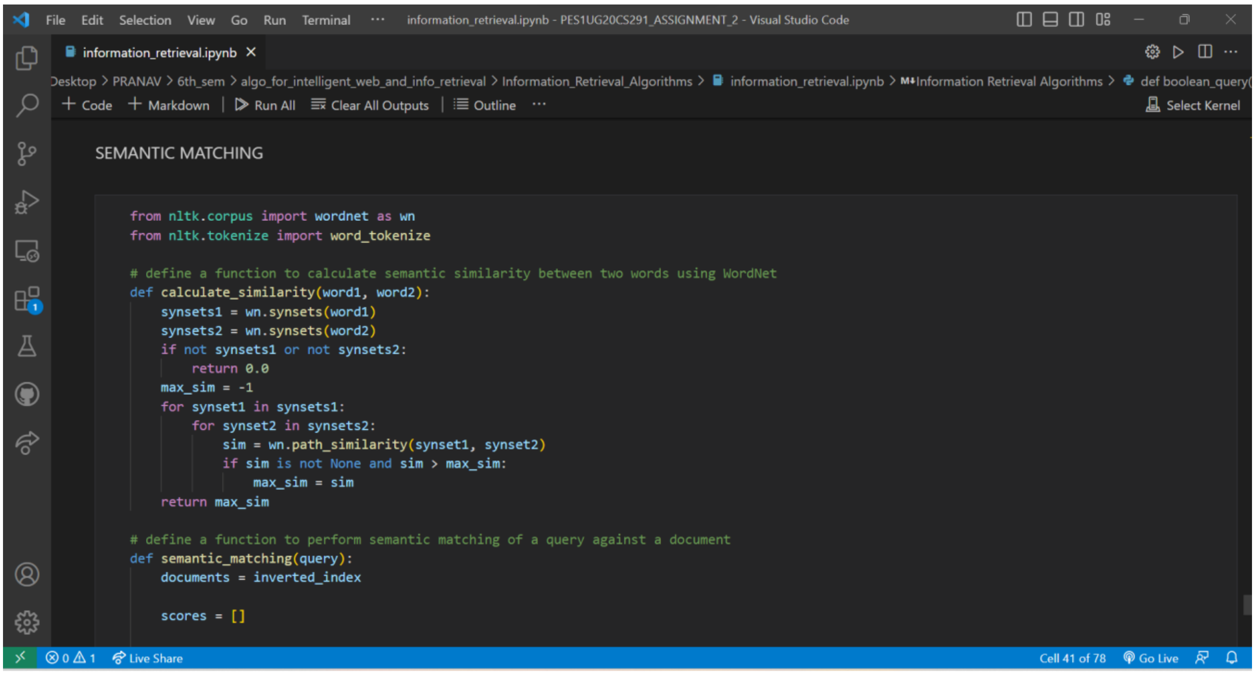
```python
def advanced_search(query):
    # get the relevant text
    relevant_text = retrieve_relevant_text(query)
    # calculate the score for each document
    scores = {}
    for doc in relevant_text:
        scores[doc] = 0
        for word in query.split():
            scores[doc] += doc.count(word)
    # sort the documents based on the score
    sorted_scores = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return sorted_scores
```

```python
advanced_search('world')
```

```
[('Whoever saves one life, saves the world entire.', 1),
 ("The greatest trick the devil ever pulled was to convince the world he didn't exist",
  1),
 ('Two unlikely people. Two different worlds come together in a story about a most unusual friendship.',
  1),
 ("All the world's a stage...", 1),
```

RERANKING RESULTS

```python
def rerank_results(query):
    index = inverted_index
    documents = search(query)
    # Create a list to store document scores
    scores = []
    # Split the query into individual terms
    query_terms = query.split()
    # Iterate over each document
    for doc_id in documents:
        # Initialize the score for this document
        score = 0
        # Iterate over each query term
        for term in query_terms:
            # If the term appears in the document
            if term in index and doc_id in index[term]:
                # Increment the score by the frequency of the term in the document
                try:
                    score += index[term][doc_id]
                except:
                    pass
        # Add the document score to the list of scores
        scores.append((doc_id, score))
    # Sort the list of scores in descending order
    scores.sort(key=lambda x: x[1], reverse=True)
    # Return the sorted list of document IDs
```



Results:

As we don’t really know the ranking of relevant documents, we will be taking the cosine similar documents as relevant and log likelihood documents as retrieved.

Conclusions:

In this assignment, we learnt how to implement all the basic requirements of a search engine.