Abstract:

This project focuses on developing a robust web document retrieval system by leveraging Python 3.10+, Scikit-Learn 1.2+, Scrapy 2.11+, and Flask 2.2+. The system comprises three key components: a Scrapy-based Crawler for downloading web documents, a Scikit-Learn-based Indexer for constructing an inverted index, and a Flask-based Processor for handling free text queries. The abstract encapsulates the project's development summary, objectives, and potential next steps.

Development Summary:

The development of the web document retrieval system commenced with comprehensive research into existing technologies and methodologies for web crawling, indexing, and query processing. Following this, the project proceeded to the design phase, where the architecture and functionalities of each component were outlined in detail. Subsequently, the system was implemented, integrating Scrapy for crawling, Scikit-Learn for indexing, and Flask for query processing. Rigorous testing and validation procedures were conducted throughout the development process to ensure the reliability, efficiency, and accuracy of the system.

Objectives:

The overarching objectives of the web document retrieval system are multifaceted and encompass:

1. Content Crawling: Develop a sophisticated Scrapy-based crawler capable of efficiently traversing the web, downloading web documents, and adhering to specified constraints such as seed URL/domain, maximum pages, and maximum depth. By enabling seamless content acquisition, the crawler aims to provide a solid foundation for subsequent indexing and query processing tasks.
2. Effective Search Indexing: Implement a robust Scikit-Learn-based indexer tasked with constructing an inverted index using TF-IDF score/weight representation and enabling seamless search through cosine similarity calculations. By organizing and structuring the retrieved content in an efficient manner, the indexer aims to facilitate swift and accurate retrieval of relevant information in response to user queries.
3. Accurate Query Processing: Create a versatile Flask-based processor equipped with functionalities for comprehensive query validation/error-checking and retrieval of top-K ranked results. By ensuring the delivery of accurate and relevant responses to user queries across diverse contexts and scenarios, the processor aims to enhance the overall user experience and utility of the system.

Solution Outline:

The solution outlined in this project seeks to address the challenges associated with web document retrieval through the development of a comprehensive system comprising three main components: a Scrapy-based Crawler, a Scikit-Learn-based Indexer, and a Flask-based Processor.

1. Scrapy-based Crawler: The crawler is responsible for traversing the web, downloading web documents in HTML format, and storing them locally for further processing. It is initialized with a seed URL/domain, maximum pages to crawl, and maximum depth of traversal, ensuring focused and efficient content acquisition.
2. Scikit-Learn-based Indexer: The indexer constructs an inverted index using TF-IDF score/weight representation, enabling efficient search indexing. It calculates cosine similarity scores between documents, facilitating accurate and relevant retrieval of information based on user queries.
3. Flask-based Processor: The processor handles user queries in JSON format, providing functionalities for query validation/error-checking and retrieval of top-K ranked results. It ensures that users receive accurate and relevant responses to their queries, enhancing the overall user experience.

Proposed System:

The proposed web document retrieval system integrates Scrapy, Scikit-Learn, and Flask to offer a seamless user experience. The Scrapy-based Crawler efficiently acquires web content, customizable to user preferences. The Scikit-Learn-based Indexer constructs an inverted index using TF-IDF, ensuring accurate search results via cosine similarity calculations. The Flask-based Processor handles user queries, providing JSON interface, query validation, and top-K results retrieval. Together, these components address challenges in content crawling, search indexing, and query processing, empowering users to extract valuable information from the internet effortlessly.

Design:

System Capabilities:

The system offers the following features:

Web Crawling: Utilizing a Scrapy-based web crawler, the system retrieves web documents from specified URLs, following constraints like traversal depth and maximum page count.
Indexing: Employing a Scikit-Learn-based approach, the system creates an inverted index using TF-IDF scores, with added functionalities like word embeddings and FAISS integration for enhanced similarity search.
Query Processing: The Flask processor utilizes cosine similarity and TF-IDF scores to handle user queries, ensuring accuracy and offering options like spell check and query expansion.

Interactions:
Web Crawling to Indexing: Crawled web content is utilized by the indexing engine to generate an inverted index with TF-IDF scores.
Indexing to Query Processing: The indexing engine produces an inverted index to retrieve relevant documents in response to user queries.
User Interaction: Users interact with the system through the Flask-based query processor by entering queries and receiving search results.

Integration:
Two APIs facilitate search functionality, connecting with the indexing engine and query processor. Web data flows to the indexing engine, then to the query processor, which interacts with both APIs. The modular design allows for future feature enhancements, ensuring an efficient system for delivering accurate search results.

Architecture:
Software Components:
1. Web Crawler: Utilizes Scrapy to fetch web documents.
2. Indexing Engine: Utilizes Scikit-Learn for TF-IDF indexing, with optional advanced techniques such as word embeddings and FAISS.
3. Query Processor: Flask-based module responsible for managing user queries, validation, and retrieving results.
4. APIs: Consist of two distinct APIs to deliver search results using both standard and advanced indexing methods.

Interfaces:
API Interfaces: RESTful APIs providing search functionality.
Data Interface: Facilitating communication between the crawler, indexing engine, and query processor via data pipelines.
The architecture ensures smooth interaction among components, featuring well-defined interfaces for data exchange and modular implementation to support scalability and future enhancements.

Operation
Install Python and Install Linux in windows
wsl –install
Install required libraries
Pip install scrapy
Pip install sckit-learn
pip install beautifulsoup4
pip install flask
pip install requests

Instructions for running the project:
Step 1: To set up the Scrapy-based Crawler, navigate to the spiders directory in the terminal and execute the command "Scrapy crawl <file name>". This will initiate the crawler to download web documents in HTML format according to specified parameters such as seed URL/domain, maximum pages, and maximum depth.

Step 2: After the crawler has completed its task, access the index.pkl file by navigating to the access pickle folder in the terminal. Execute the provided Python script to view the contents of the index.pkl file, which stores TF-IDF scores and cosine similarity for the downloaded HTML documents.

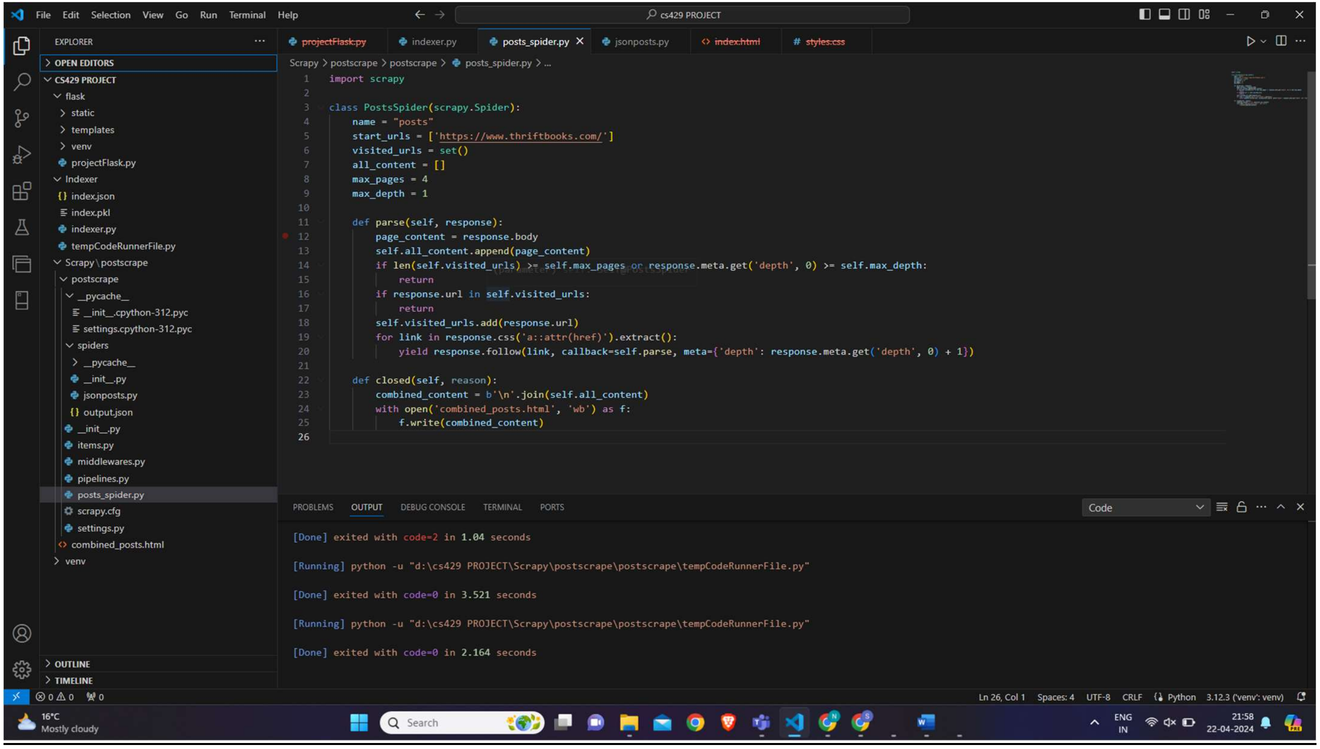
Step 3: Launch the Flask-based Processor by navigating to the Flask directory in the terminal and running the provided Python script. This processor is responsible for handling free text queries in JSON format.

Step 4: With the Flask server running, open a new terminal window. Run the command for output:
python “your file name”

Upon execution, the server will return a JSON response containing cosine similarity scores and document names for the top k results relevant to the provided query.

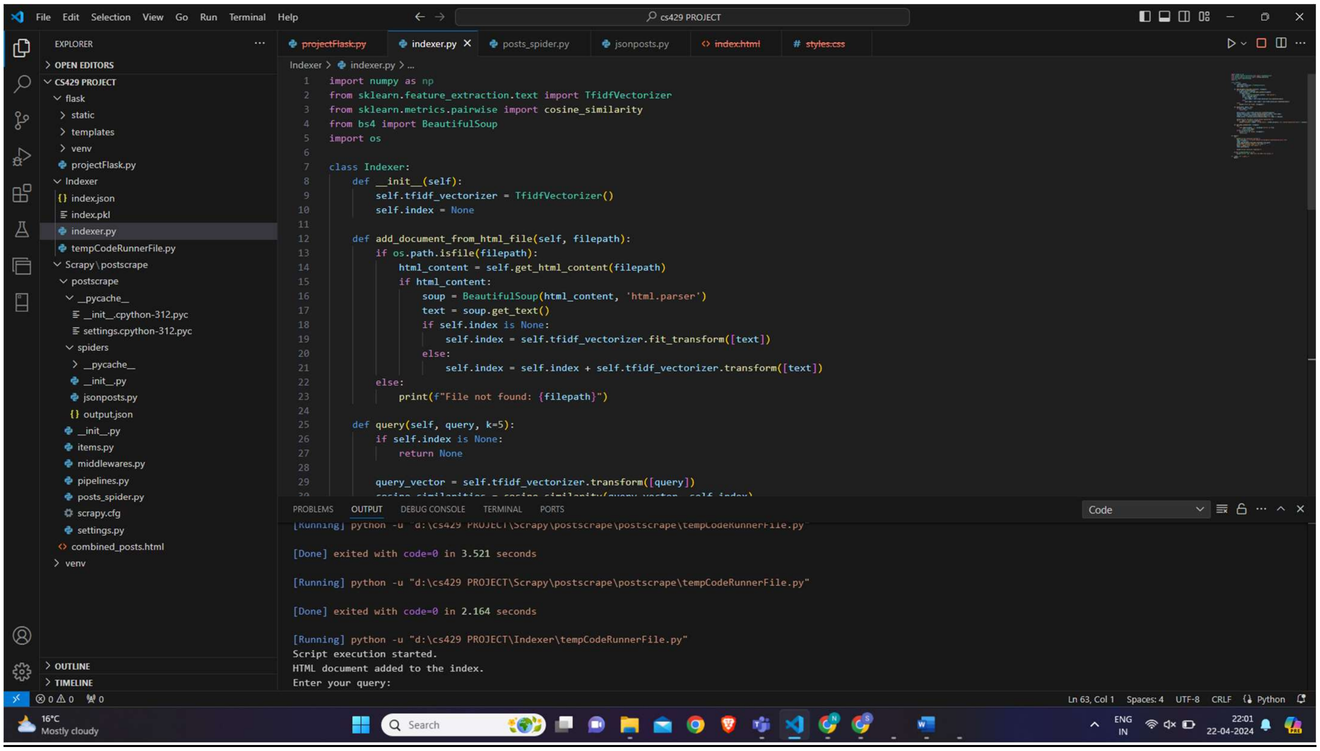
<u>Output:</u>

<u>Scrapy model:</u>



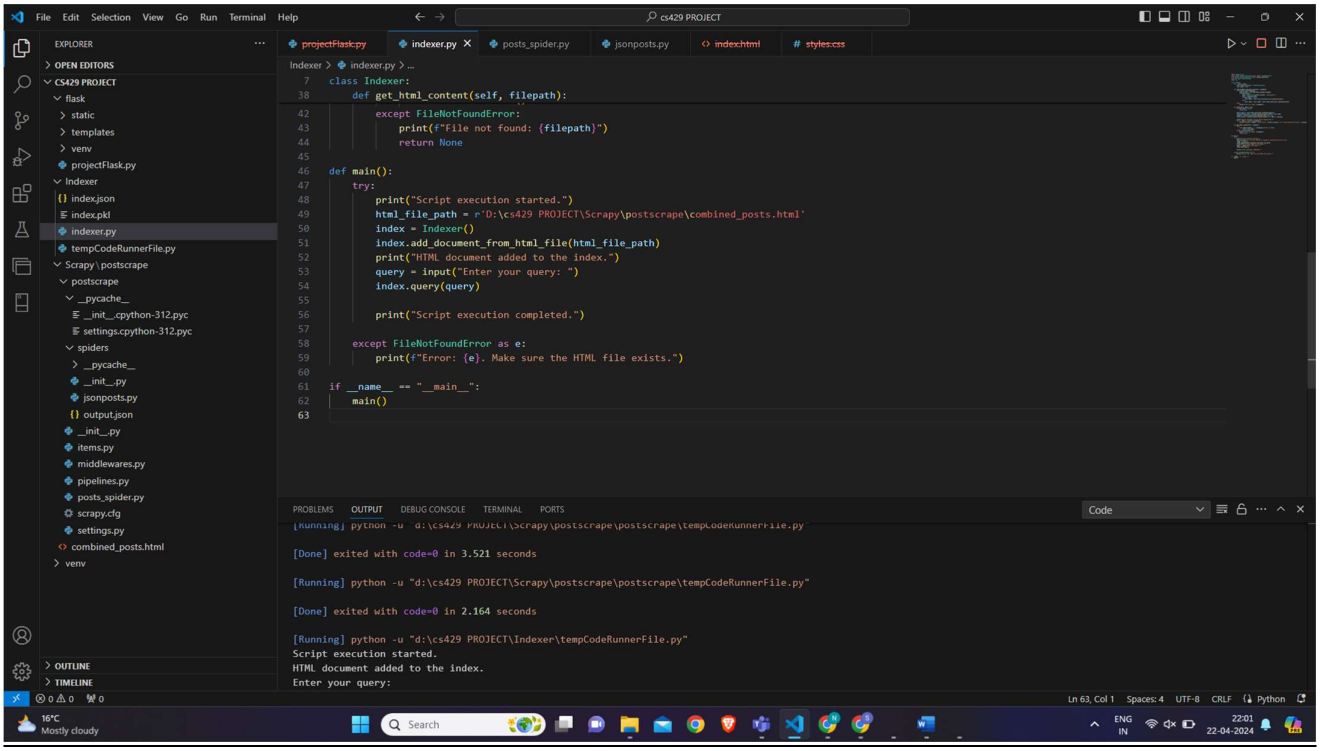
```python
import scrapy

class PostsSpider(scrapy.Spider):
    name = "posts"
    start_urls = ['https://www.thriftbooks.com/']
    visited_urls = set()
    all_content = []
    max_pages = 4
    max_depth = 1

    def parse(self, response):
        page_content = response.body
        self.all_content.append(page_content)
        if len(self.visited_urls) >= self.max_pages or response.meta.get('depth', 0) >= self.max_depth:
            return
        if response.url in self.visited_urls:
            return
        self.visited_urls.add(response.url)
        for link in response.css('a::attr(href)').extract():
            yield response.follow(link, callback=self.parse, meta={'depth': response.meta.get('depth', 0) + 1})

    def closed(self, reason):
        combined_content = b'\n'.join(self.all_content)
        with open('combined_posts.html', 'wb') as f:
            f.write(combined_content)

```

```
[Done] exited with code=2 in 1.04 seconds

[Running] python -u "d:\cs429 PROJECT\Scrapy\postscrape\postscrape\tempCodeRunnerFile.py"

[Done] exited with code=0 in 3.521 seconds

[Running] python -u "d:\cs429 PROJECT\Scrapy\postscrape\postscrape\tempCodeRunnerFile.py"

[Done] exited with code=0 in 2.164 seconds
```



```python
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from bs4 import BeautifulSoup
import os

class Indexer:
    def __init__(self):
        self.tfidf_vectorizer = TfidfVectorizer()
        self.index = None

    def add_document_from_html_file(self, filepath):
        if os.path.isfile(filepath):
            html_content = self.get_html_content(filepath)
            if html_content:
                soup = BeautifulSoup(html_content, 'html.parser')
                text = soup.get_text()
                if self.index is None:
                    self.index = self.tfidf_vectorizer.fit_transform([text])
                else:
                    self.index = self.index + self.tfidf_vectorizer.transform([text])
        else:
            print(f"File not found: {filepath}")

    def query(self, query, k=5):
        if self.index is None:
            return None

        query_vector = self.tfidf_vectorizer.transform([query])
        cosine_similarities = cosine_similarity(query_vector, self.index)
```

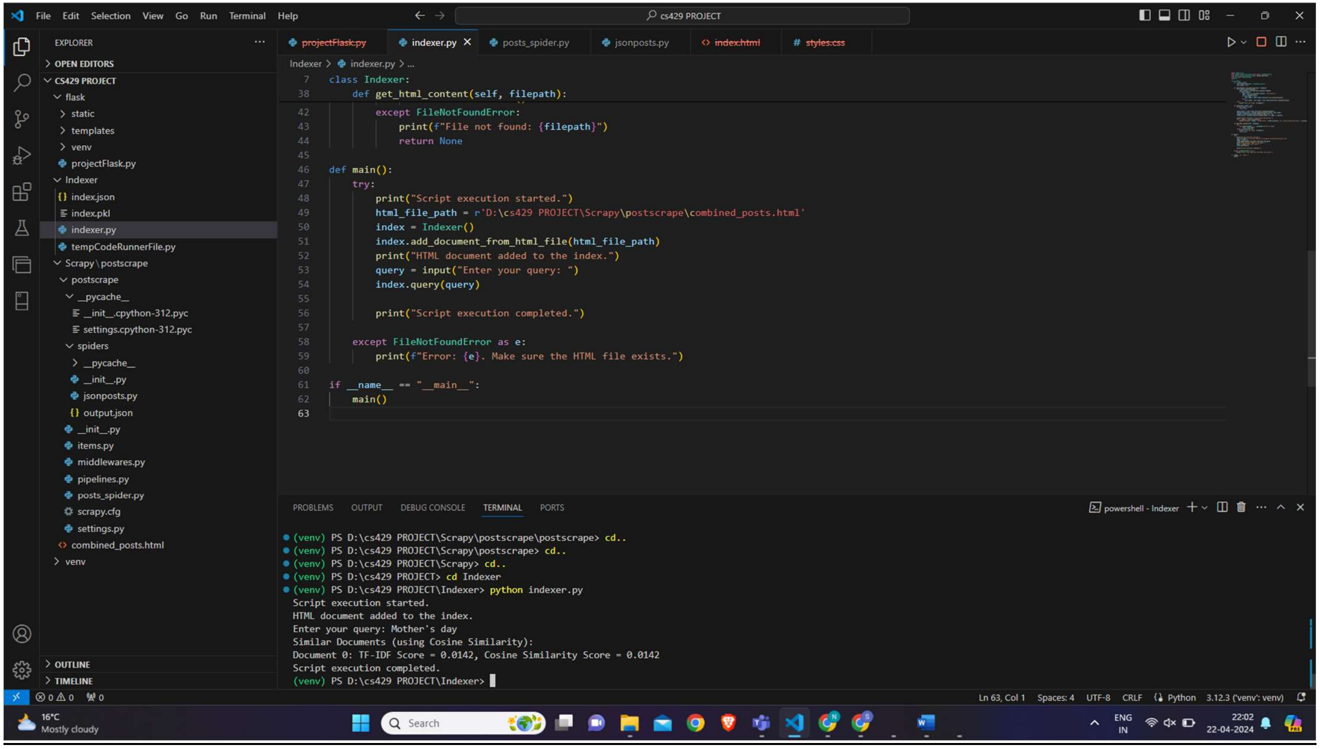
```
[Running] python -u "d:\cs429 PROJECT\Scrapy\postscrape\postscrape\tempCodeRunnerFile.py"

[Done] exited with code=0 in 3.521 seconds

[Running] python -u "d:\cs429 PROJECT\Scrapy\postscrape\postscrape\tempCodeRunnerFile.py"

[Done] exited with code=0 in 2.164 seconds

[Running] python -u "d:\cs429 PROJECT\Indexer\tempCodeRunnerFile.py"
Script execution started.
HTML document added to the index.
Enter your query:
```



```python
class Indexer:
    def get_html_content(self, filepath):
            except FileNotFoundError:
                print(f"File not found: {filepath}")
                return None

def main():
    try:
        print("Script execution started.")
        html_file_path = r'D:\cs429 PROJECT\Scrapy\postscrape\combined_posts.html'
        index = Indexer()
        index.add_document_from_html_file(html_file_path)
        print("HTML document added to the index.")
        query = input("Enter your query: ")
        index.query(query)

        print("Script execution completed.")

    except FileNotFoundError as e:
        print(f"Error: {e}. Make sure the HTML file exists.")

if __name__ == "__main__":
    main()
```

```
[Running] python -u "d:\cs429 PROJECT\Scrapy\postscrape\postscrape\tempCodeRunnerFile.py"

[Done] exited with code=0 in 3.521 seconds

[Running] python -u "d:\cs429 PROJECT\Scrapy\postscrape\postscrape\tempCodeRunnerFile.py"

[Done] exited with code=0 in 2.164 seconds

[Running] python -u "d:\cs429 PROJECT\Indexer\tempCodeRunnerFile.py"
Script execution started.
HTML document added to the index.
Enter your query:
```

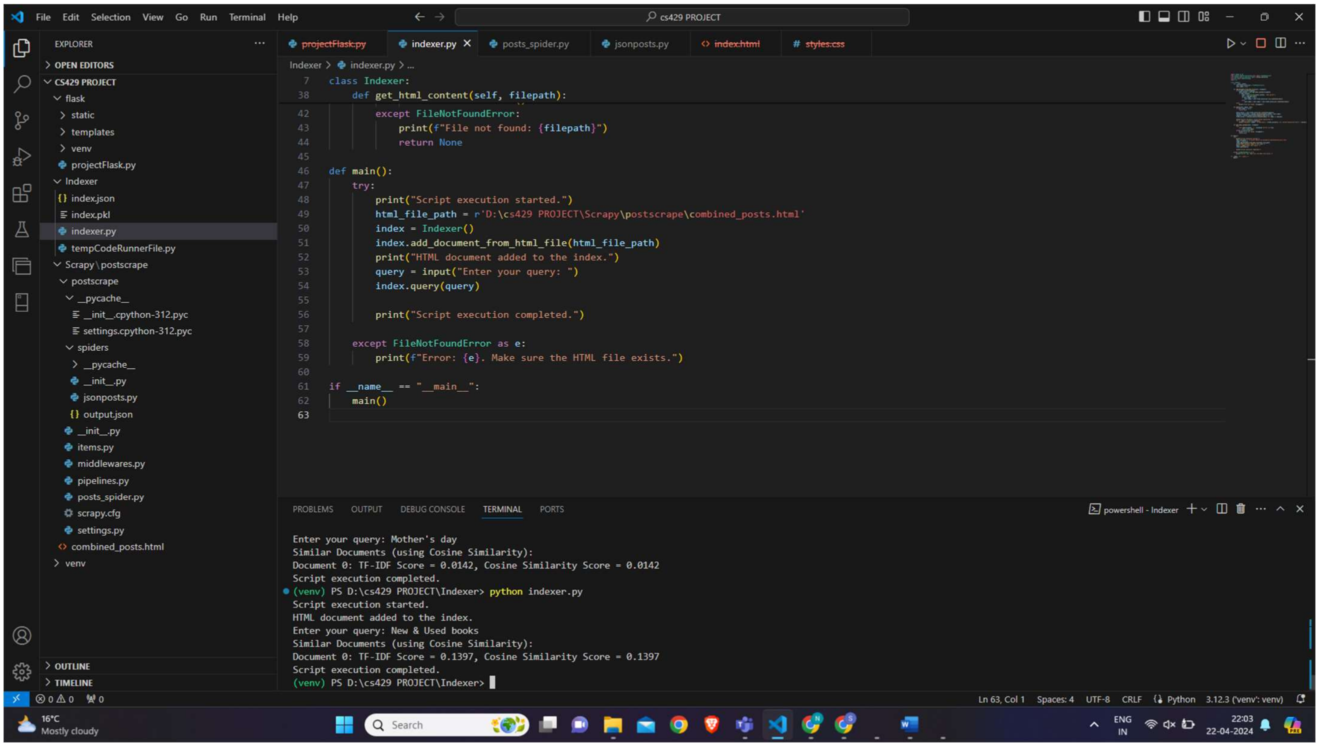
Test Case – 1 for Scikit-learn:



```python
class Indexer:
    def get_html_content(self, filepath):
            except FileNotFoundError:
                print(f"File not found: {filepath}")
                return None

def main():
    try:
        print("Script execution started.")
        html_file_path = r'D:\cs429 PROJECT\Scrapy\postscrape\combined_posts.html'
        index = Indexer()
        index.add_document_from_html_file(html_file_path)
        print("HTML document added to the index.")
        query = input("Enter your query: ")
        index.query(query)

        print("Script execution completed.")

    except FileNotFoundError as e:
        print(f"Error: {e}. Make sure the HTML file exists.")

if __name__ == "__main__":
    main()
```

```
(venv) PS D:\cs429 PROJECT\Scrapy\postscrape\postscrape> cd..
(venv) PS D:\cs429 PROJECT\Scrapy\postscrape> cd..
(venv) PS D:\cs429 PROJECT\Scrapy> cd..
(venv) PS D:\cs429 PROJECT> cd Indexer
(venv) PS D:\cs429 PROJECT\Indexer> python indexer.py
Script execution started.
HTML document added to the index.
Enter your query: Mother's day
Similar Documents (using Cosine Similarity):
Document 0: TF-IDF Score = 0.0142, Cosine Similarity Score = 0.0142
Script execution completed.
(venv) PS D:\cs429 PROJECT\Indexer>
```

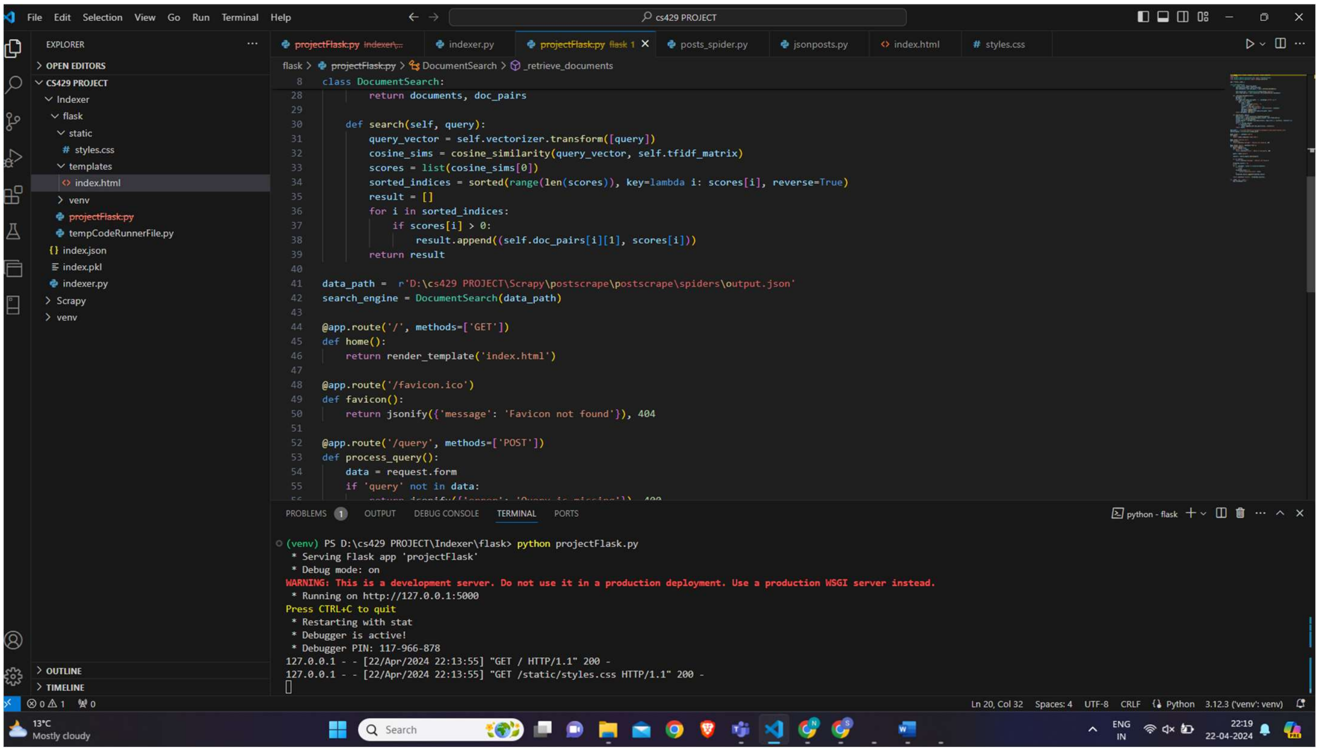
Test case 2 for Scikit-learn:



```python
class Indexer:
    def get_html_content(self, filepath):
        except FileNotFoundError:
            print(f"File not found: {filepath}")
            return None

def main():
    try:
        print("Script execution started.")
        html_file_path = r'D:\cs429 PROJECT\Scrapy\postscrape\combined_posts.html'
        index = Indexer()
        index.add_document_from_html_file(html_file_path)
        print("HTML document added to the index.")
        query = input("Enter your query: ")
        index.query(query)

        print("Script execution completed.")

    except FileNotFoundError as e:
        print(f"Error: {e}. Make sure the HTML file exists.")

if __name__ == "__main__":
    main()
```

```
Enter your query: Mother's day
Similar Documents (using Cosine Similarity):
Document 0: TF-IDF Score = 0.0142, Cosine Similarity Score = 0.0142
Script execution completed.
(venv) PS D:\cs429 PROJECT\Indexer> python indexer.py
Script execution started.
HTML document added to the index.
Enter your query: New & Used books
Similar Documents (using Cosine Similarity):
Document 0: TF-IDF Score = 0.1397, Cosine Similarity Score = 0.1397
Script execution completed.
(venv) PS D:\cs429 PROJECT\Indexer>
```

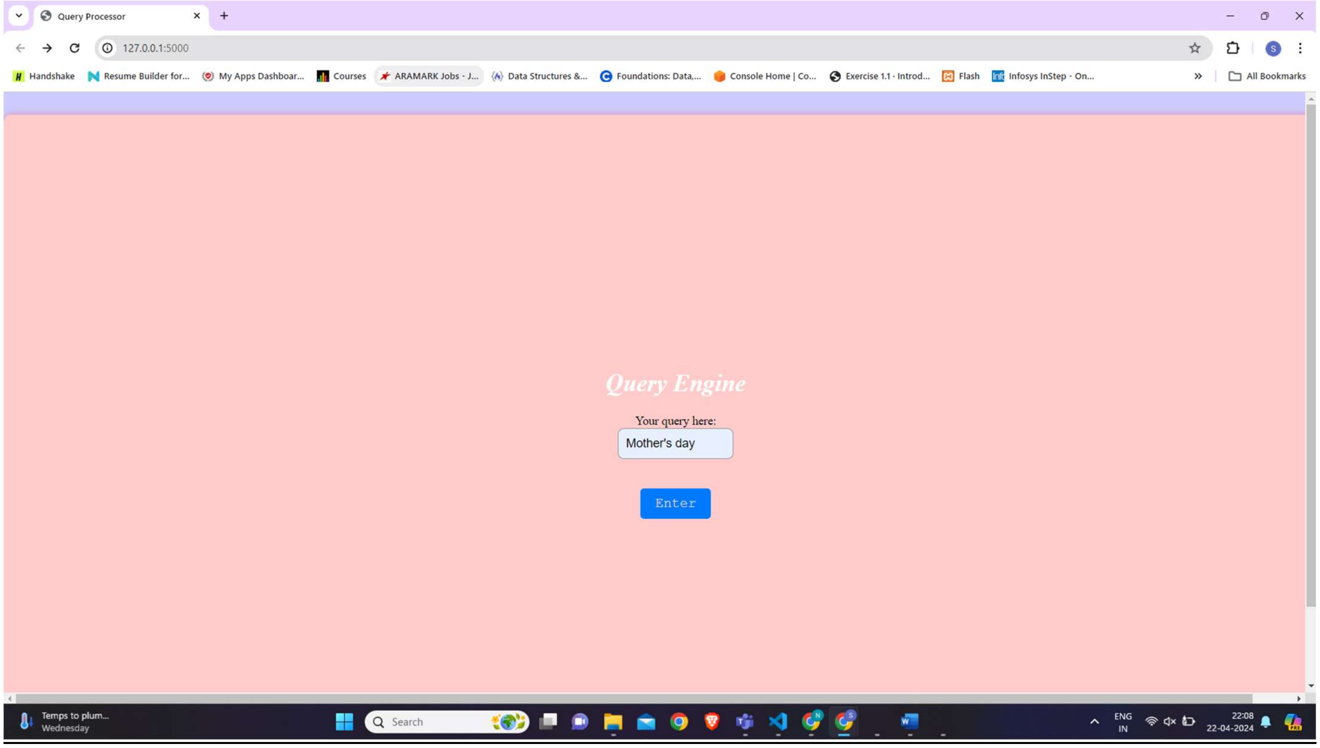
Output for top k-ranked:



```python
class DocumentSearch:
            return documents, doc_pairs

    def search(self, query):
        query_vector = self.vectorizer.transform([query])
        cosine_sims = cosine_similarity(query_vector, self.tfidf_matrix)
        scores = list(cosine_sims[0])
        sorted_indices = sorted(range(len(scores)), key=lambda i: scores[i], reverse=True)
        result = []
        for i in sorted_indices:
            if scores[i] > 0:
                result.append((self.doc_pairs[i][1], scores[i]))
        return result

data_path =  r'D:\cs429 PROJECT\Scrapy\postscrape\postscrape\spiders\output.json'
search_engine = DocumentSearch(data_path)

@app.route('/', methods=['GET'])
def home():
    return render_template('index.html')

@app.route('/favicon.ico')
def favicon():
    return jsonify({'message': 'Favicon not found'}), 404

@app.route('/query', methods=['POST'])
def process_query():
    data = request.form
    if 'query' not in data:
```

```
(venv) PS D:\cs429 PROJECT\Indexer\flask> python projectFlask.py
 * Serving Flask app 'projectFlask'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 117-966-878
127.0.0.1 - - [22/Apr/2024 22:13:55] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [22/Apr/2024 22:13:55] "GET /static/styles.css HTTP/1.1" 200 -
```

Test case – 1 for top k-ranked:

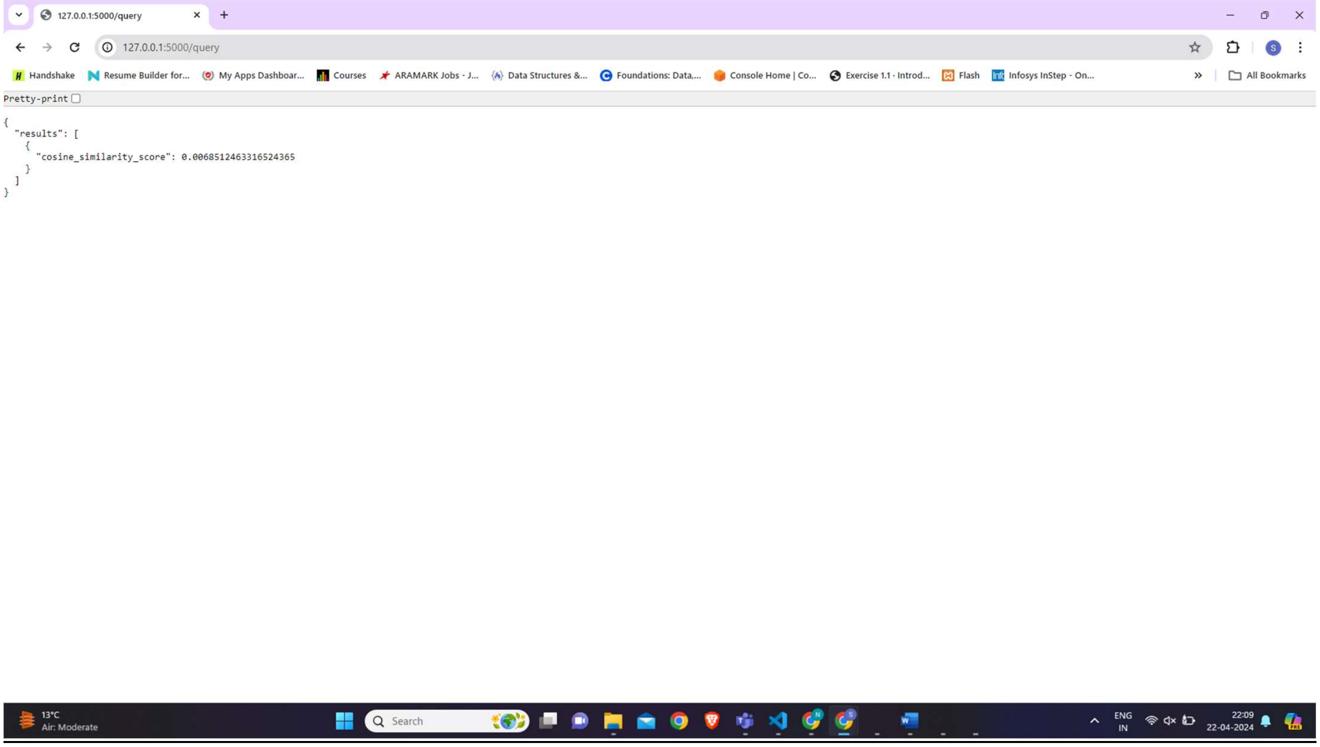


Query Engine

Your query here:

Mother's day

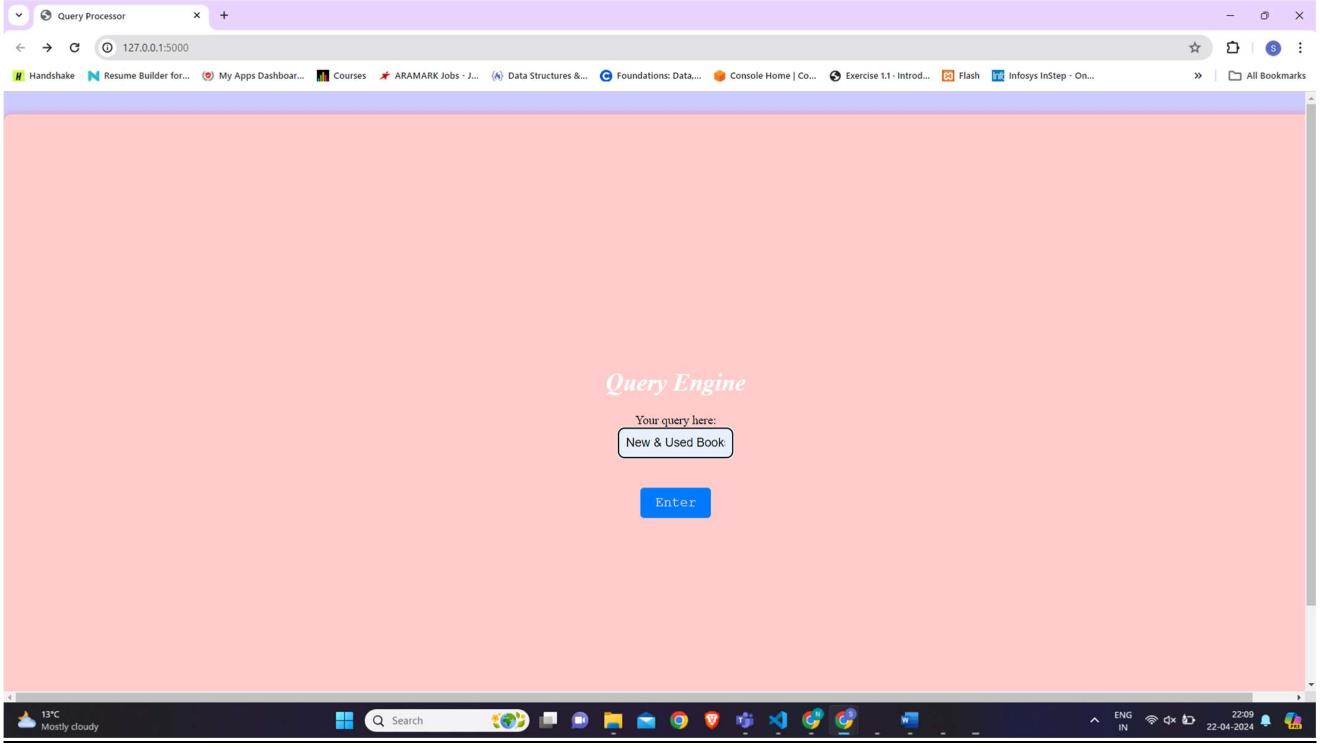
Enter



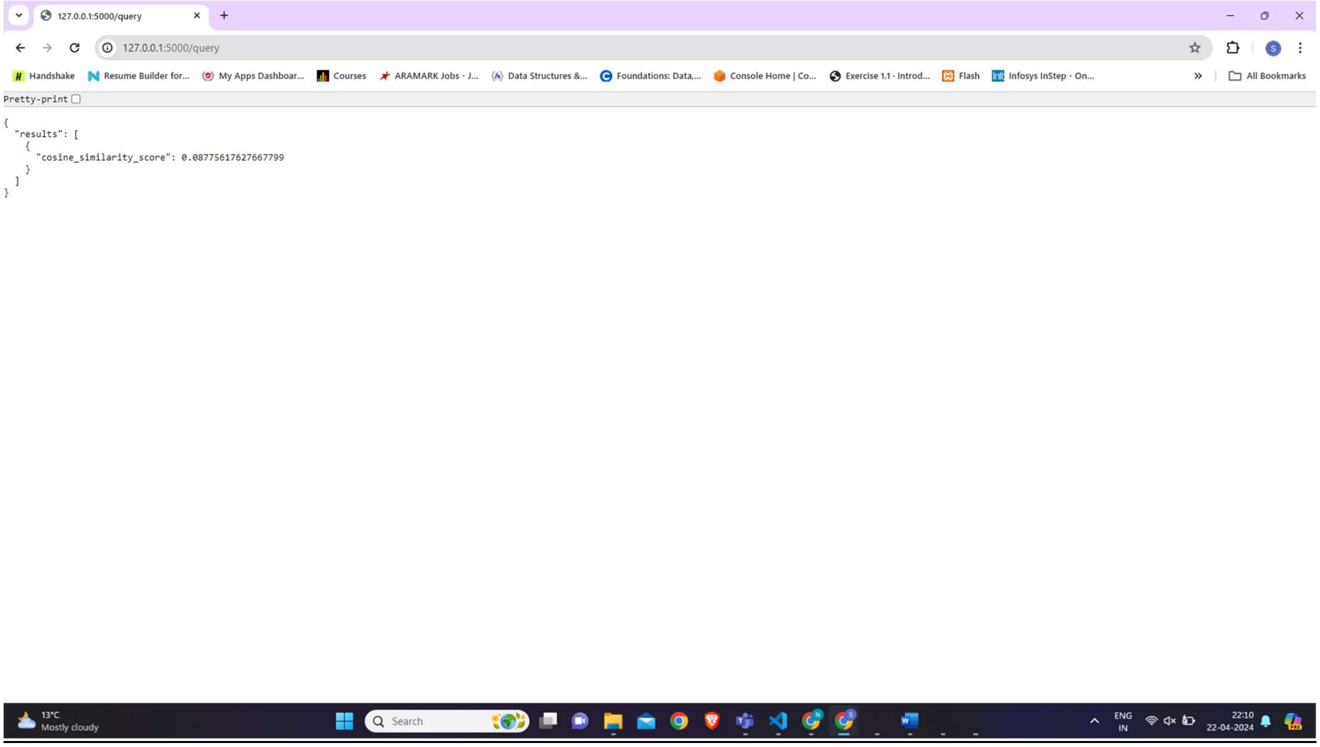
Pretty-print □

```
{
  "results": [
    {
      "cosine_similarity_score": 0.0068512463316524365
    }
  ]
}
```

Test Case – 2 for top k – ranked:



New & Used Book



```
{
  "results": [
    {
      "cosine_similarity_score": 0.08775617627667799
    }
  ]
}
```

Conclusion:

The development and implementation of the web document retrieval system have yielded significant insights and outcomes. By fulfilling the requirements of a Scrapy-based Crawler, a Scikit-Learn-based Indexer, and a Flask-based Processor, the system has demonstrated notable success in efficiently acquiring, indexing, and processing web-based content. However, several factors must be considered when evaluating the system's performance, outputs, and potential limitations.

Data Sources

Scrapy - version 2.11.1
Beautiful Soup - version 4
Scikit-learn - version 1.4.2
Flask - version 3.0.3

Bibliography:

Scrapy-based Crawler:
https://www.digitalocean.com/community/tutorials/how-to-crawl-a-web-page-with-scrapy-and-python-3
https://requests.readthedocs.io/en/latest/
Scikit-Learn-based Indexer:
https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
Co-sine similarity and Tf-Idf score:
https://github.com/williamscott701/Information-Retrieval/blob/master/2.%20TF-IDF%20Ranking%20-%20Cosine%20Similarity,%20Matching%20Score/TF-IDF.ipynb
Top k-ranked:
https://dl.acm.org/doi/10.1145/2348283.2348384
Flask-based Processor:
https://stackoverflow.com/questions/69932835/python-flask-server-to-retrieve-certain-records