CS 429: Information Retrieval

Name: Iqtedar Uddin
Professor: Jawahar Panchal
Date:04/22/2024
ID: A20505407

Abstract:

In this paper, we introduce a system that is created for scraping data from the web, retrieving information, and conducting searches through a web application. The system starts by exploring a starting domain with set limits on the number of pages and depth levels to gather HTML content from different webpages. The URL, title, and content of each document are extracted and saved. This data is then utilized to create an inverted index using term frequency-inverse document frequency (TF-IDF) to make searching more efficient.[1]

The system is connected to a Flask web app, where users can enter search terms and receive related documents sorted by cosine similarity. The top fifteen search results show the document title, index ID, cosine similarity score, and a link to the original URL.[2]

In the future, I plan to make the system better by looking into ways to crawl multiple sites at once or use distributed crawling to make it more efficient and scalable. I also want to implement advanced techniques like word2vec and neural search kNN to improve how we retrieve information. This includes adding features like query correction and expansion to make searching more accurate and enjoyable for users.[3]

Overview:

The system discussed in this paper aims to tackle the issue of effective web crawling, information retrieval, and querying within a web application setting. Through utilizing web scraping methods, the system gathers HTML content from a designated seed domain, emphasizing reaching a maximum number of pages and depth levels to guarantee thorough coverage.[4]

---

[1] Text generated by ChatGPT, OpenAI, April 22, 2023, https://chat.openai.com/share/91c8bb72-37df-4c04-86d3-ebca4d5468ee.
[2] OpenAI
[3] OpenAI
[4] OpenAI

When data is collected, such as document URLs, titles, and contents, it is preprocessed and indexed to create an inverted index using TF-IDF scores. This index makes it quick and easy to search for relevant documents based on user queries. [5]

The frontend of the system is developed with Flask, offering a user-friendly interface for users to interact with the search feature. Users can enter queries on the web application, starting a search that retrieves and ranks relevant documents based on cosine similarity scores. The top fifteen search results are shown to users, giving them convenient access to important information.[6]

Overall, the system provides a complete solution for gathering data from websites, finding specific information, and searching for data, with room for improvements in future versions.[7]

Design:

The system's design has several components that enable efficient web crawling, information retrieval, and querying.[8]

System Capabilities:

1. Web Scraping: The system utilizes a web spider implemented with Scrapy to crawl through a specified seed domain, collecting HTML content from web pages.
2. Data Extraction: During crawling, the system extracts relevant information from each web page, including the URL, title, and content.
3. Preprocessing: extracted content goes through preprocessing, like tokenization, stop word removal, and stemming to prepare it for indexing.
4. Indexing: The preprocessed data is indexed using a TF-IDF-based inverted index for efficient querying.
5. Querying: Users can input search queries through a Flask-based web application, causing the search and retrieval of relevant documents from the inverted index.
6. Ranking: Retrieved documents are ranked based on cosine similarity scores and pruned so most relevant results to be displayed to users.[9]

---

[5] OpenAI
[6] OpenAI
[7] OpenAI
[8] OpenAI
[9] OpenAI

Interactions:

1. Web Spider Interaction: The web spider interacts with the specified seed domain, following links and collecting HTML content from web pages.
2. Data Extraction Interaction: Extracted data is passed to the preprocessing component for tokenization, stop word removal, and stemming.
3. Indexing Interaction: Preprocessed data is indexed using the TF-IDF-based inverted index and stored for querying.
4. Querying Interaction: Users interact with the Flask web application interface to input search queries, triggering the retrieval of relevant documents from the inverted index.
5. Ranking Interaction: Retrieved documents are ranked based on cosine similarity scores, with the top results displayed to users through the web application interface.[10]

Architecture:

The system architecture is designed to support the integration of the various components involved in web crawling, information retrieval, and querying.

Software Components:

1. Web Spider: Implemented using Scrapy, the web spider component is responsible for crawling through the seed domain and collecting HTML content from web pages.
2. Preprocessing Module: This module uses NLTK to preprocess the extracted data, including tokenization, stop word removal, and stemming, to prepare it for indexing.
3. Indexing Module: The indexing module constructs the TF-IDF-based inverted index for the preprocessed data allowing for efficient querying.
4. Querying Interface: Implemented using Flask, the querying interface provides users with a user-friendly interface to input search queries and retrieve relevant documents.
5. Ranking Module: The ranking module ranks retrieved documents based on cosine similarity scores, ensuring that the most relevant results are displayed to users.[11]

Interfaces:

1. Web Spider Interface: The web spider interacts with the specified seed domain to collect HTML content from web pages.
2. Preprocessing Interface: Extracted data is passed to the preprocessing module for tokenization, stop word removal, and stemming.
3. Indexing Interface: Preprocessed data is indexed using the TF-IDF-based inverted index, which is then stored for querying.

---

[10] OpenAI
[11] OpenAI

4. Querying Interface: Users interact with the Flask-based web application interface to input search queries and retrieve relevant documents.
5. Ranking Interface: Retrieved documents are ranked based on cosine similarity scores, with the top results displayed to users through the querying interface.[12]

Operation:

The operation of the system involves a sequential execution of steps to achieve web crawling, indexing, querying, and information retrieval.

1. Web Scraping:
    - Begin by configuring the web spider with parameters such as the seed URL, maximum depth, and maximum number of pages to crawl.
    - Run the web spider to initiate the crawling process. The spider navigates through the seed domain, following links and collecting HTML content from web pages.
    - Extracted data, including the URL, title, and content of each web page, is saved to the specified directory as HTML files.
2. Indexing:
    - After web scraping is complete, preprocess the collected HTML content by tokenizing, removing stop words, and stemming the text.
    - Build an inverted index using the TF-IDF algorithm, which maps terms to the documents in which they appear and their corresponding TF-IDF scores.
    - Save the inverted index to disk for efficient querying.
3. Launching the Querying Interface:
    - Once indexing is finished, launch the Flask-based querying interface.
    - Users can input search queries through the interface, which triggers the retrieval of relevant documents from the inverted index.
    - Retrieved documents are ranked based on cosine similarity scores, and the top results are displayed to users.
4. Modifying Parameters:
    - If desired, parameters such as the seed URL, maximum depth, and maximum number of pages can be modified to target different domains or increase the scope of crawling.
    - Modify these parameters in the web spider configuration and rerun the scraping process to collect additional data.
5. Clearing Index:
    - If needed, clear the existing index to start indexing anew or with different data.

[12] OpenAI

- This step ensures that the inverted index reflects the most current data and parameters.

6. Repeatable Process:
    - The entire operation can be repeated as necessary, allowing for iterative improvements or adjustments to the crawling, indexing, and querying processes.[13]

Conclusion:

The system has demonstrated success in achieving its primary objectives of web crawling, indexing, and querying, providing users with efficient access to relevant information from crawled web pages. Key highlights of the system's performance include:

- Successful Retrieval: The system effectively retrieves and indexes web content, allowing users to query and retrieve relevant documents based on their search queries.
- Top Rankings: It returns the top 15 ranked results based on cosine similarity scores, enabling users to quickly identify the most relevant documents.
- Evaluation of Results: Users are provided with the cosine similarity scores for each retrieved document, offering insights into the quality and relevance of the search results.

However, certain challenges and limitations are observed in the system's operation:

- Error Handling: Some links may result in errors during crawling, leading to skipped documents or incomplete indexing. This can impact the comprehensiveness of the index and the accuracy of search results.
- Resource Intensive: Crawling and indexing large volumes of web content can consume significant time and storage space. Additionally, querying a large index may require considerable computational resources.
- Depth and Coverage: Limitations in the depth and coverage of crawling may result in certain words or terms being missed, particularly if they appear in deeper layers of the web hierarchy. This can affect the completeness of the index and the recall of search results.

Despite these challenges, the system represents a valuable tool for information retrieval, providing users with access to a curated collection of web content and enabling efficient search and retrieval processes. Future enhancements may focus on improving error handling, optimizing resource utilization, and expanding the coverage and depth of web crawling to enhance the system's effectiveness and usability.[14]

---

[13] OpenAI
[14] OpenAI

Data Sources:

1. Seed URL: https://en.wikipedia.org/wiki/Main_Page is where the initial scraping starts in our application. This can be changed to your own personalized seed.

2. Downloadable Dataset: For replication of test cases, the Github repository includes the inverted index, as well as the contents that were scraped, as different instances of the scraper can produce different results. This ensures you can test the query shown in the test case.[15]

Test Cases:

Framework

To set up the necessary environment and dependencies for the system, we employed a straightforward approach to ensure reproducibility and ease of use. The framework includes the following steps:

1. Environment Setup: We used Python as the primary programming language for developing the system. Python provides libraries and tools for web scraping, natural language processing, and web application development, which were essential for the project.

2. Dependency Management: We utilized the Python package manager, pip, to install the required libraries. The main dependencies for our system include Scrapy for web scraping, NLTK for natural language processing tasks such as tokenization and stemming, Flask for building the web application, and other standard libraries such as os, collections, and math.

3. Installation Instructions: To install the necessary dependencies, users can use the following command:

```
pip install scrapy nltk flask
```

[15] OpenAI

4. Python Version: The system was developed for Python 3.10+, scrapy 2.11+, and Flask 2.2+[16]

Harness

5. Scraping and Indexing:
    - Objective: Verify that the system correctly scrapes web content from the seed domain, builds the inverted index, and handles errors appropriately.
    - Steps:
        1. Set the maximum pages to 10,000 and depth to 6.
        2. Run the web scraping process.
        3. Monitor the process for any errors and record the total number of documents scraped.
        4. Build the inverted index from the scraped documents.
    - Expected Outcome: The system should successfully scrape web content, handle errors gracefully, and build the inverted index with the specified parameters.

6. Initial Application Setup:
    - Objective: Ensure that the Flask application is initialized correctly and displays the initial search interface.
    - Steps:
        1. Start the Flask application.
        2. Navigate to the application URL in a web browser.
        3. Verify that the application interface loads without errors and displays the search input field.
    - Expected Outcome: The Flask application should start without issues, and the initial interface should be displayed, ready for user interaction.

Coverage

7. Querying and Retrieval:
    - Objective: Test the system's ability to process user queries and retrieve relevant documents.
    - Steps:
        1. Input a search query into the provided field.
        2. Submit the query and observe the search results(valid or invalid).
    - Expected Outcome: The system should return the top 15 ranked results based on the cosine similarity scores for the given query.
8. Viewing Document:

[16] OpenAI

- Objective: Verify that users can view the content of a selected document from the search results.
- Steps:
    1. Click on the "View Document" link associated with a search result.
    2. Verify that the document content is displayed correctly in a new tab or window.
- Expected Outcome: Users should be able to access and view the content of the selected document from the search results seamlessly.[17]

Summary Screenshots:

1. Scraping and Indexing:

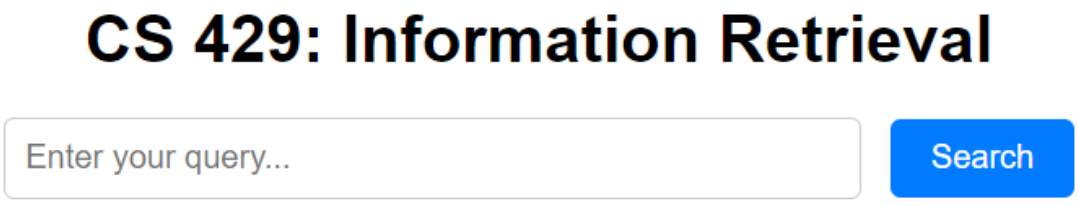
[17] OpenAI

Screenshot:

```
'downloader/response_bytes': 121015217,
'downloader/response_count': 3020,
'downloader/response_status_count/200': 2967,
'downloader/response_status_count/301': 6,
'downloader/response_status_count/302': 45,
'downloader/response_status_count/404': 2,
'dupefilter/filtered': 3979,
'elapsed_time_seconds': 173.041661,
'finish_reason': 'finished',
'finish_time': datetime.datetime(2024, 4, 22, 16, 35, 15, 263568, tzinfo=datetime.timezone.
'httpcompression/response_bytes': 666575868,
'httpcompression/response_count': 2968,
'httperror/response_ignored_count': 2,
'httperror/response_ignored_status_count/404': 2,
'log_count/DEBUG': 3408,
'log_count/ERROR': 684,
'log_count/INFO': 14,
'log_count/WARNING': 1,
'offsite/domains': 386,
'offsite/filtered': 3037,
'request_depth_max': 3,
'response_received_count': 2969,
'scheduler/dequeued': 3020,
'scheduler/dequeued/memory': 3020,
'scheduler/enqueued': 3020,
'scheduler/enqueued/memory': 3020,
'spider_exceptions/AttributeError': 4,
'spider_exceptions/FileNotFoundError': 5,
'spider_exceptions/OSError': 659,
'spider_exceptions/ValueError': 16,
```

This index has been run multiple times, to test duplication handling as well. It can correctly get url responses, but we can see some errors in scrapy that are skipped, which reduce index size.

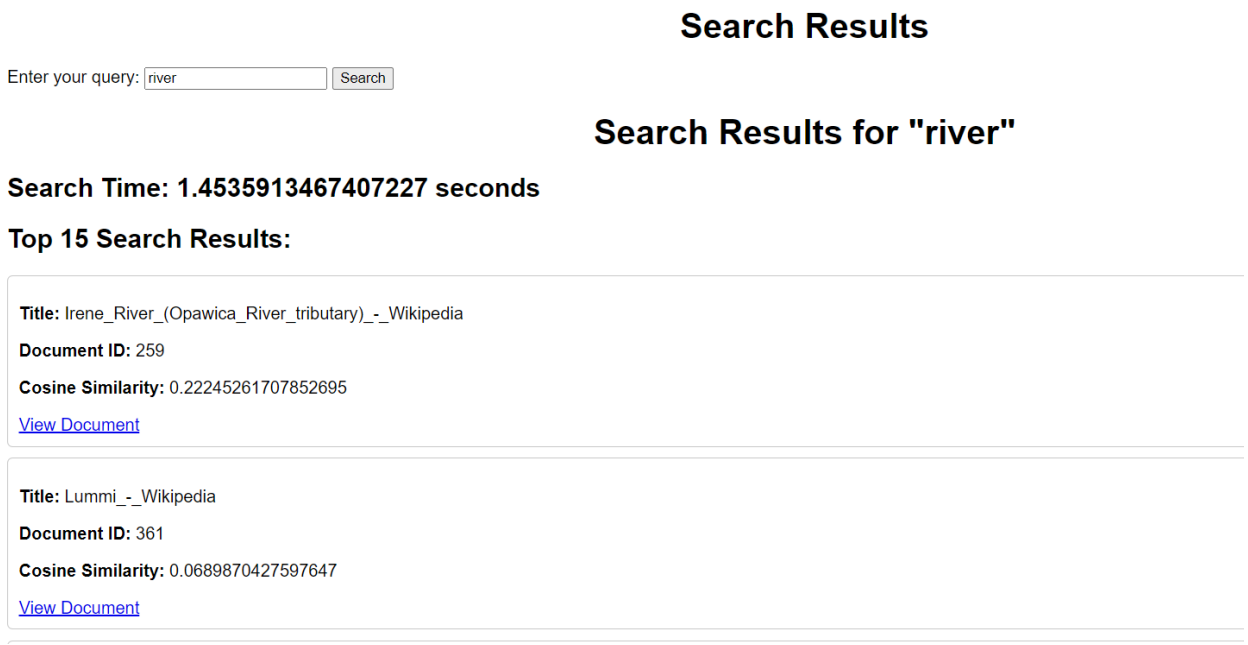
2. Initial Application Setup:



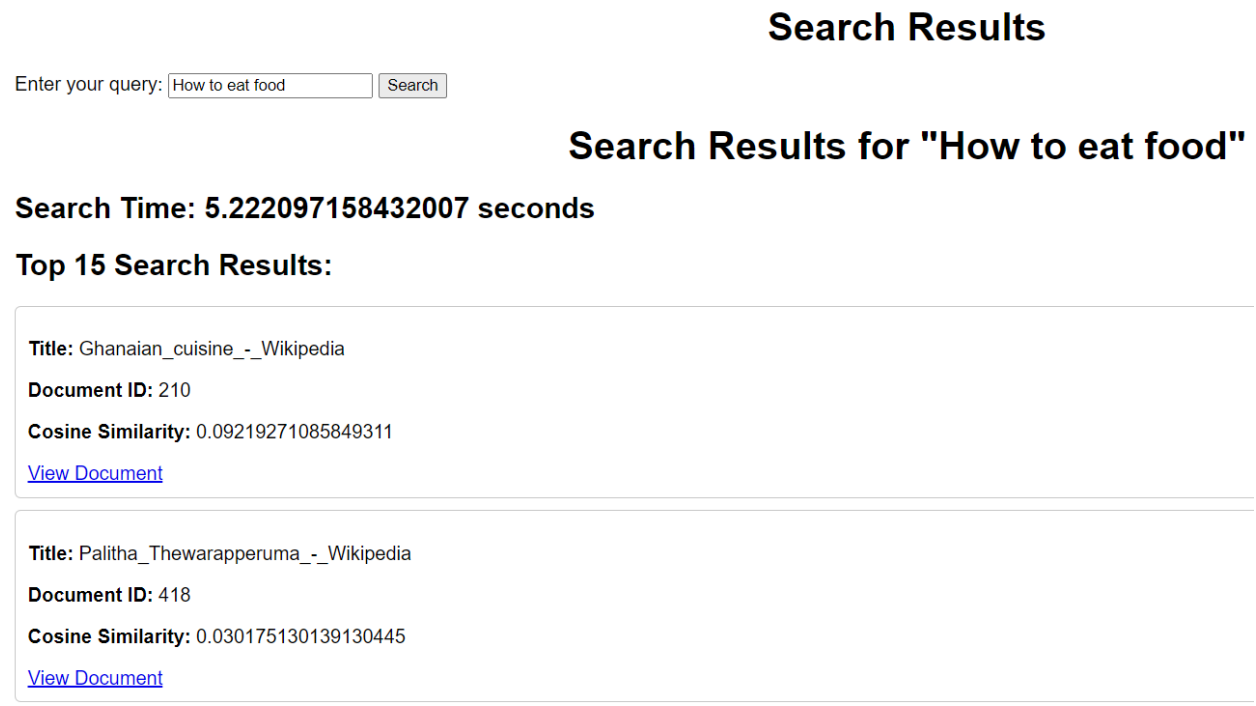
3. Querying and Retrieval:

Valid Results:

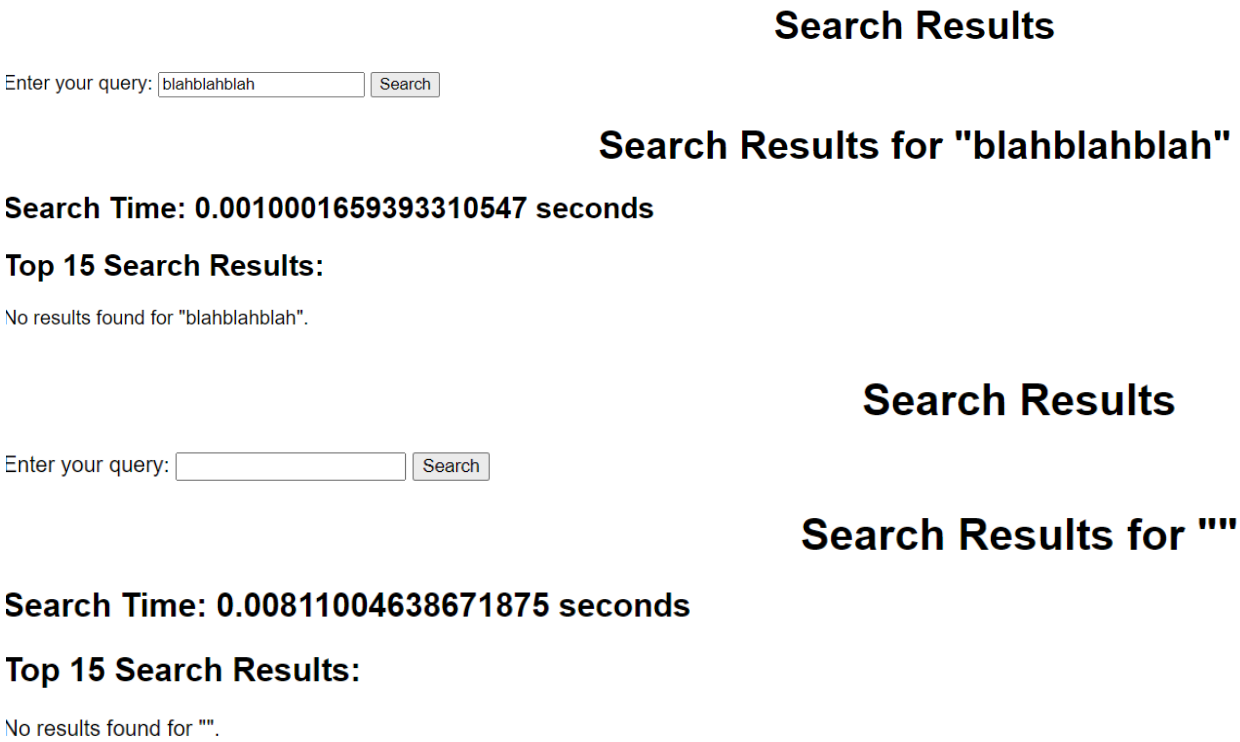
One Term:



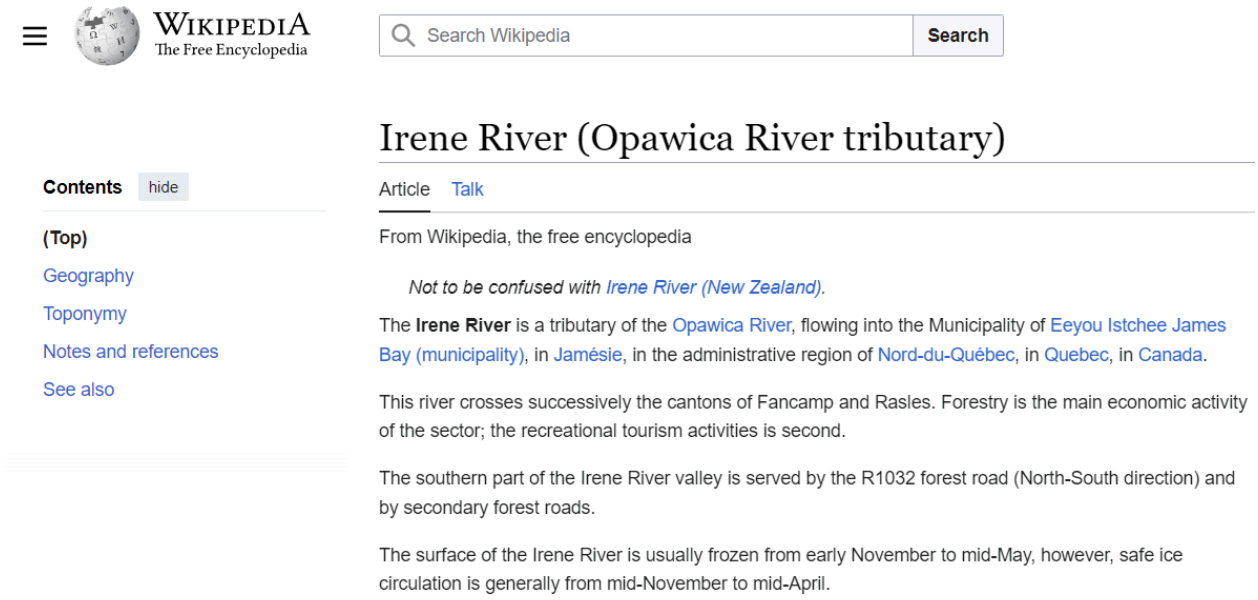
Multiple terms:



Invalid Results(Not in Query or empty):



4. Viewing Document:



Source Code:

1. GitHub Repository:
    - Link to GitHub Repository
2. Dependencies:
    - The system relies on several Python libraries, including, Math, Os, Collections Scrapy, NLTK, Flask, and pickle for serialization.
3. Code Files:
    - web_spider.py: Contains the Scrapy spider for web scraping.
    - scraper.py: Runs the web spider to scrape web content.
    - index.py: Builds the inverted index using block-based processing.
    - search.py: Implements cosine similarity searching and preprocessing for relevant document retrieval.
    - app.py: Constructs the Flask application for user interaction.
    - clear.py: Deletes the existing directory and recreates it for building a new index.
    - search.html: includes template for search screen which displays top 15 results
    - index.html: displays initial home screen for query entry

Code and Documentation Screenshots:

1. web_spider.py:[18]

```python
import os
import scrapy

class WebSpider(scrapy.Spider):
    name = "web_spider"
    allowed_domains = ['en.wikipedia.org']  # Limit crawling to Wikipedia
    start_urls = ['https://en.wikipedia.org/wiki/Main_Page']  # Starting URL
    max_pages = 10000  # Maximum number of pages to crawl
    max_depth = 6

    def parse(self, response):
        # Extract content from the current page
        content = {
            'url': response.url,
            'title': response.css('title::text').get(),
            'body': response.css('p::text').getall()+ response.css('h1::text').getall() + response.css('h2::text').getall() +
            response.css('h3::text').getall() +
            response.css('h4::text').getall() +
            response.css('h5::text').getall() +
            response.css('h6::text').getall() +
            response.css('li::text').getall() +  # List items
            response.css('.mw-parser-output > *::text').getall(),
        }

        # Save content to file
        self.save_content(content)

        # Extract links from the current page and follow them if within max depth and max pages limit
        if response.meta.get('depth', 1) < self.max_depth and self.max_pages > 0:
```

[18] OpenAI

```
        for link in response.css('a::attr(href)').getall():
            if self.max_pages <= 0:
                break
            self.max_pages -= 1
            yield response.follow(link, callback=self.parse, meta={'depth': response.meta.get('depth', 1) + 1})

def save_content(self, content):
    # Create a directory to store HTML files if it doesn't exist
    directory = 'C:/website_content'
    if not os.path.exists(directory):
        os.makedirs(directory)


    file_name = f"{content['title'].replace(' ', '_')}.html"
    text_file_name = f"{content['title'].replace(' ', '_')}.txt"
    file_path = os.path.join(directory, file_name)
    text_file_path = os.path.join(directory, text_file_name)

    # Write URL to a separate text file
    with open(text_file_path, 'w', encoding='utf-8') as f:
        f.write(f"URL: {content['url']}\n")

    # Write HTML content to file
    with open(file_path, 'w', encoding='utf-8') as f:
        f.write(f"<html><head><title>{content['title']}</title></head><body>")
        for paragraph in content['body']:
            f.write(f"<p>{paragraph}</p>")
        f.write("</body></html>")
```

Description: This is the code for the web_spider, which is where the user defines the max pages, ma depth, and seed url. This then gathers all html related content and puts it in a directory for storing the documents, as well as the original url for our search engine.

2. scraper.py:[19]

[19] OpenAI

```python
from scrapy.crawler import CrawlerProcess
from scrapy.utils.project import get_project_settings

from web_spider import WebSpider  # Adjust import statement

def main():
    # Create a CrawlerProcess instance with project settings
    process = CrawlerProcess(get_project_settings())

    # Run the spider
    process.crawl(WebSpider)
    process.start()  # the script will block here until the crawling is finished

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

Description: This is the main code to create and run the web_spider.

3. index.py:[20]

[20] OpenAI

```python
import os
import math
from collections import defaultdict
import pickle
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# Function to read HTML content from files in a directory
def read_html_files(directory):
    documents = []
    for filename in os.listdir(directory):
        if filename.endswith(".html"):
            with open(os.path.join(directory, filename), "r", encoding="utf-8") as f:
                title = filename.split(".")[0]  # Extract title from filename
                content = f.read()  # Read content from file
                documents.append((title, content))  # Append tuple of title and content
    return documents

# Function to preprocess text
def preprocess(text):
    tokens = word_tokenize(text.lower())  # Tokenization and lowercase
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word.isalnum() and word not in stop_words]  # R
    stemmer = PorterStemmer()
    tokens = [stemmer.stem(word) for word in tokens]  # Stemming
    return tokens
```

```python
def merge_blocks(block_files):
    """Merges inverted blocks from disk into a single inverted index."""
    inverted_index = defaultdict(list)
    for block_file in block_files:
        with open(block_file, "rb") as f:
            block = pickle.load(f)
            for token, postings in block.items():
                inverted_index[token].extend(postings)

    return inverted_index

# Step 4: Create a function to process documents in batches and build the inverted index
def process_batch_and_invert(documents, batch_size, output_dir):
    """Processes documents in batches, creates inverted blocks, and merges them."""
    inverted_index = defaultdict(list)
    doc_id = 0
    doc_freq = defaultdict(int)

    os.makedirs(output_dir, exist_ok=True)  # Ensure the output directory exists

    for i in range(0, len(documents), batch_size):
        batch = documents[i:i+batch_size]
        for doc_title, doc_content in batch:
            tokenized_doc = preprocess(doc_content)
            doc_freq_for_doc = {}
```

```
            for term in tokenized_doc:
                doc_freq_for_doc[term] = doc_freq_for_doc.get(term, 0) + 1

            for term, freq in doc_freq_for_doc.items():
                tf = freq / len(tokenized_doc)
                doc_freq[term] += 1

                # Calculate TF-IDF score
                tf_idf_score = tf * math.log(len(documents) / doc_freq[term])
                inverted_index[term].append((doc_id, doc_title, tf_idf_score))

            doc_id += 1

        block_file = f"{output_dir}/block_{i}.pkl"
        with open(block_file, "wb") as f:
            pickle.dump(inverted_index, f)

        inverted_index = defaultdict(list)  # Reset for the next batch

    # Merge blocks into a single inverted index
    block_files = [f"{output_dir}/block_{i}.pkl" for i in range(0, len(documents), batch_size)]
    merged_index = merge_blocks(block_files)

    # Combine TF-IDF scores with inverted index
    for term, title, scores in inverted_index.items():
        for doc_id, score in scores:
            merged_index[term].append((doc_id, title, score))

    return merged_index
```

```
# Define batch size and output directory (adjust as needed)
batch_size = 1000
output_dir = "C:/index/"
index_directory = "C:/index/"

# Read HTML content from directory
website_content_directory = "C:/website_content"
documents = read_html_files(website_content_directory)

# Create the inverted index using block-based processing
inverted_index = process_batch_and_invert(documents, batch_size, output_dir)

# Store the inverted index in the output directory
os.makedirs(output_dir, exist_ok=True)
index_file = os.path.join(output_dir, "inverted_index.pkl")
with open(index_file, "wb") as f:
    pickle.dump(inverted_index, f)
```

Description: This is the index where we load the html documents that we scraped, and preprocess them before indexing. We then use block-based processing, in case the user wants to do a high amount of scraping later on. They can adjust the block size here, but 1000 should suffice as a good benchmark. This then merges and orders terms by TF-TDF for efficient querying and saves the index as a pickle file.

4. search.py:[21]

```
import math
import pickle
from nltk import word_tokenize, PorterStemmer
from nltk.corpus import stopwords

def preprocess(text):
    tokens = word_tokenize(text.lower())  # Tokenization and lowercase
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word.isalnum() and word not in stop_words]
    stemmer = PorterStemmer()
    tokens = [stemmer.stem(word) for word in tokens]  # Stemming
    return tokens


def load_index(index_file):
    with open(index_file, "rb") as f:
        inverted_index = pickle.load(f)
    return inverted_index


def calculate_query_vector(query_terms, inverted_index):
    query_vector = {}
    total_docs = len(inverted_index)  # Total number of documents

    for term in query_terms:
        if term in inverted_index:
            idf_score = math.log(total_docs / len(inverted_index[term]))
            query_vector[term] = idf_score
```

[21] OpenAI

```
        else:
            query_vector[term] = 0  # Set IDF score to 0 if term not found in inverted index

    return query_vector

def search_query(query, inverted_index):
    # Preprocess the query
    preprocessed_query = preprocess(query)
    query_vector = calculate_query_vector(preprocessed_query, inverted_index)

    document_scores = {}  # Dictionary to store document scores
    document_titles = {}  # Dictionary to store document titles

    # Calculate dot product for each document
    for term, query_score in query_vector.items():
        if term in inverted_index:
            for doc_id, title, doc_score in inverted_index[term]:
                if doc_id not in document_scores:
                    document_scores[doc_id] = 0
                # Calculate dot product
                document_scores[doc_id] += query_score * doc_score
                document_titles[doc_id] = title

    # Calculate magnitudes of query vector
    query_magnitude = math.sqrt(sum(score ** 2 for score in query_vector.values()))

    # Normalize dot products by dividing by document vector lengths and query vector length
for doc_id in document_scores:
    # Calculate magnitude of document vector
    doc_magnitude = math.sqrt(sum(score ** 2 for term_scores in inverted_index.values() \
                                  for doc_id_, _, score in term_scores if doc_id_ == doc_id))
    # Calculate cosine similarity score
    document_scores[doc_id] /= (query_magnitude * doc_magnitude) if doc_magnitude != 0 else 1

# Sort documents by cosine similarity scores
sorted_documents = sorted(document_scores.items(), key=lambda x: x[1], reverse=True)

return [(doc_id, document_titles[doc_id], score) for doc_id, score in sorted_documents]
```

Description: This is our search query file, where we open the inverted index, preprocess a user query, and then use cosine similarity, with the query terms in order to find relevant documents. We then rank them in order from highest to lowest, so we have the top results.

5. app.py:[22]

[22] OpenAI

```python
import time

from flask import Flask, request, jsonify, render_template, send_file, redirect
import os
from search import search_query, load_index

# Initialize Flask application
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')  # Assuming you have an HTML file named index.html

@app.route('/view_document/<string:title>', methods=['GET'])
def view_document(title):
    # Retrieve the file path of the text file containing the URL
    text_file_name = f"{title}.txt"
    directory = 'C:/website_content'
    text_file_path = os.path.join(directory, text_file_name)

    # Read the URL from the text file
    with open(text_file_path, 'r', encoding='utf-8') as f:
        url = f.readline().split(": ")[1].strip()

    # Redirect the user to the URL
    return redirect(url)
```

```python
@app.route('/search', methods=['GET'])
def search():
    # Extract the search query from the request parameters
    query = request.args.get('query')

    # Load inverted index
    index_file = "C:/index/inverted_index.pkl"  # Update with your index file path
    inverted_index = load_index(index_file)

    start_time = time.time()

    # Perform search query processing
    search_results = search_query(query, inverted_index)

    end_time = time.time()

    search_time = end_time - start_time
    # Slice search_results to only include the first 15 results
    search_results = search_results[:15]

    # Render the search template with the search results
    return render_template('search.html', query=query, search_results=search_results, search_time=search_time)


if __name__ == '__main__':
    # Run the Flask application
    app.run(debug=True)
    app.run('0.0.0.0', '5000')
```

Description: This is the flask app responsible for allowing users to query the index made. This has an index for the home page, and search which sends the result to the search_query file to find relevant documents, and information about them. It also times it, to see how certain queries affect search time. If the user wants to view the document, they can click the view_document, which opens the actual url used in wikipedia. This displays the top 15, but if you change the limit, you can go higher or lower.

6. Index.html[23]

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Information Retrieval</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
        }
        h1 {
            text-align: center;
            margin-bottom: 20px;
        }
        .search-form {
            text-align: center;
            margin-bottom: 20px;
        }
        .search-input {
            width: 400px;
            padding: 10px;
            font-size: 16px;
            border-radius: 5px;
            border: 1px solid #ccc;
            margin-right: 10px;
        }
```

[23] OpenAI

```
        .search-button {
            padding: 10px 20px;
            font-size: 16px;
            border-radius: 5px;
            background-color: #007bff;
            color: #fff;
            border: none;
            cursor: pointer;
        }
        .search-results {
            margin-top: 20px;
        }
    </style>
</head>
<body>
    <h1>CS 429: Information Retrieval</h1>
    <form class="search-form" action="/search" method="GET">
        <input class="search-input" type="text" name="query" placeholder="Enter your query...">
        <button class="search-button" type="submit">Search</button>
    </form>
    <div class="search-results">
        <!-- Search results will be displayed here -->
    </div>
</body>
</html>
```

Description: This is the index, It holds the main page for the user to query.

7. Search.html[24]

[24] OpenAI

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Search Results</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
        }
        h1 {
            text-align: center;
            margin-bottom: 20px;
        }
        .search-results {
            margin-top: 20px;
        }
        .result-item {
            margin-bottom: 10px;
            padding: 10px;
            border: 1px solid #ccc;
            border-radius: 5px;
        }
        .result-link {
            color: #007bff;
            text-decoration: none;
            font-weight: bold;
        }
```

```
    </style>
</head>
<body>
    <h1>Search Results</h1>

    <form action="/search" method="GET">
        <label for="query">Enter your query:</label>
        <input type="text" id="query" name="query" value="{{ query }}">
        <button type="submit">Search</button>
    </form>

    <h1>Search Results for "{{ query }}"</h1>
    <h2>Search Time: {{ search_time }} seconds</h2>
    <h2>Top 15 Search Results:</h2>

    <div class="search-results">
        {% if search_results %}
            {% for doc_id, title, similarity in search_results %}
                <div class="result-item">
                    <p><strong>Title:</strong> {{ title }}</p>
                    <p><strong>Document ID:</strong> {{ doc_id }}</p>
                    <p><strong>Cosine Similarity:</strong> {{ similarity }}</p>
                    <a href="{{ url_for('view_document', title=title) }}" target="_blank">View Document</a>
                </div>
            {% endfor %}
        {% else %}
            <p>No results found for "{{ query }}".</p>
        {% endif %}
    </div>
</body>
</html>
```

Description: This is the search html used to display search results and to allow users to see documents. We can see a query validation where if something is not in the index, or a error case arises, a no result found is returned.

8. clear.py:[25]

[25] OpenAI

```python
import os
import shutil


def clear_and_create_directory(directory_path):
    # Check if the directory exists
    if os.path.exists(directory_path):
        # Clear the directory by removing all its contents
        shutil.rmtree(directory_path)

    # Create a new empty directory
    os.makedirs(directory_path)


# Specify the directory path
directory_path = "C:/website_content"

# Call the function to clear and create the directory
clear_and_create_directory(directory_path)

print("Cleared and created the directory:", directory_path)
```

Description: Responsible for clearing the index and creating it again, in case you move on to another domain to test it.

Bibliography

1. Text generated by ChatGPT, OpenAI, April 22, 2023, https://chat.openai.com/share/91c8bb72-37df-4c04-86d3-ebca4d5468ee.