CS 429 Information Retrieval

Project Report

Abstract

This project presents the development of an innovative information retrieval system tailored to facilitate efficient web document search and retrieval. The system comprises three primary components: a robust web crawler, an efficient indexer, and a responsive search server. Leveraging state-of-the-art technologies such as Scrapy, Scikit-Learn, and Flask, our system aims to enhance the efficacy of web document collection, ensure swift and accurate search functionalities, and provide an intuitive query interface.

The web crawler component of our system utilizes Scrapy, a powerful web crawling framework, to gather web documents effectively. It employs intelligent crawling strategies to navigate the web efficiently and collect diverse and relevant documents.

The indexer component constructs an inverted index using Scikit-Learn, enabling rapid and accurate search functionalities. By organizing document data into a structured index, the system optimizes search performance and retrieval accuracy.

The search server, developed using Flask, provides users with a seamless and intuitive query interface. It efficiently processes user queries, retrieves relevant documents from the index, and presents them in a user-friendly manner.

Throughout the development process, our focus remained on enhancing the overall efficiency and effectiveness of web document retrieval. By leveraging cutting-edge technologies and implementing intelligent algorithms, our system achieves significant improvements in search performance and user experience.

Overview

Web Crawler

The web crawler is responsible for systematically browsing the web and collecting web pages to be indexed. It starts from a set of initial seed URLs and recursively follows hyperlinks to discover new pages. The crawler adheres to the robots exclusion standard (robots.txt) to respect website access rules and avoid overloading servers.

Implementation Details

- Developed using the Scrapy framework in Python.
- Configured to obey robots.txt rules.
- Utilizes multithreading for efficient page fetching.

- Implements depth-limiting to control the crawl depth.

Indexer

The indexer processes the downloaded web pages, extracts relevant information, and creates an index for efficient search operations. It utilizes the TF-IDF (Term Frequency-Inverse Document Frequency) algorithm to calculate the importance of each term in the document corpus.

Implementation Details:

- Implemented in Python using the scikit-learn library.
- Utilizes TF-IDF vectorization for document representation.
- Stores the index in memory for fast retrieval.

Search Server

The search server handles user queries, retrieves relevant documents from the index, and returns search results to the user. It employs a RESTful API to receive queries and deliver results in a JSON format.

Implementation Details:

- Developed using Flask, a lightweight web framework in Python.
- Implements a simple query expansion mechanism to enhance search results.
- Supports concurrent requests for scalability.

Literature Review

In the development of our information retrieval system for web document search and retrieval, we utilized several key technologies to achieve our objectives. overview of three crucial technologies: Scrapy, Scikit-Learn, and Flask.

Scrapy

Scrapy is a powerful and efficient web crawling and scraping framework written in Python. Developed by Scrapinghub Ltd., Scrapy offers a robust set of tools and functionalities for extracting data from websites quickly and effectively. It provides a high-level interface for building web spiders that can navigate websites, extract structured data, and store it for further processing.

Key Features of Scrapy:

1. Asynchronous Request Handling: Scrapy utilizes asynchronous I/O to perform multiple web requests concurrently, maximizing efficiency and speed.
2. Extensibility: Scrapy offers a modular architecture that allows developers to extend its functionality through custom middleware, pipelines, and extensions.
3. XPath and CSS Selectors: Scrapy supports both XPath and CSS selectors, providing flexible options for extracting data from HTML documents.

4. Item Pipelines: Scrapy's item pipelines enable data processing and storage tasks, allowing developers to define custom processing logic for extracted data.

Scikit-Learn

Scikit-Learn is a popular machine learning library for Python that provides simple and efficient tools for data analysis and machine learning tasks. Developed by a community of contributors, Scikit-Learn offers a wide range of algorithms and utilities for classification, regression, clustering, dimensionality reduction, and more.

Key Features of Scikit-Learn:

1. Consistent API: Scikit-Learn provides a consistent and user-friendly API that makes it easy to experiment with different machine learning algorithms and techniques.
2. Comprehensive Documentation: Scikit-Learn offers extensive documentation and examples, making it accessible to both novice and experienced users.
3. Integration with NumPy and SciPy: Scikit-Learn seamlessly integrates with other scientific computing libraries such as NumPy and SciPy, enabling efficient data manipulation and computation.
4. Model Evaluation Tools: Scikit-Learn provides tools for model evaluation, including cross-validation, grid search, and performance metrics, facilitating the selection and optimization of machine learning models.

Flask

Flask is a lightweight and versatile web framework for Python, designed to make web development simple and scalable. Developed by Armin Ronacher, Flask provides essential features for building web applications, APIs, and microservices with minimal boilerplate code.

Key Features of Flask:

1. Lightweight and Flexible: Flask is lightweight and unopinionated, allowing developers to choose their preferred tools and libraries for various tasks.
2. Extensibility: Flask offers a modular architecture that supports extensions for adding functionality such as authentication, database integration, and RESTful API support.
3. Jinja2 Templating: Flask integrates with the Jinja2 templating engine, enabling the generation of dynamic HTML content with minimal effort.
4. Werkzeug WSGI Toolkit: Flask is built on top of the Werkzeug WSGI toolkit, providing low-level utilities for handling HTTP requests and responses.

Scrapy, Scikit-Learn, and Flask are essential technologies that played a crucial role in the development of our information retrieval system. By leveraging the capabilities of these frameworks, we were able to build a robust and efficient system for web document search and retrieval.

System Architecture

The system architecture follows a distributed model, where each component operates independently but communicates seamlessly to provide search functionality.

- Crawler (crawler.py): Responsible for crawling web pages and extracting text content.
- Indexer (indexer.py): Creates a TF-IDF matrix from the crawled documents.
- Server (server.py): Provides an interface for users to submit search queries and receive relevant results.

Initial Implementation:

Crawler

- Developed a basic web crawler using Scrapy.
- Extracted text content from web pages and stored it in a local database.

Indexer

- Created a TF-IDF matrix using scikit-learn's TfidfVectorizer.
- Indexed the crawled documents and saved the TF-IDF matrix to disk.

Server

- Implemented a Flask server to handle search queries.
- Utilized the TF-IDF matrix to calculate document similarities and return relevant results.

Challenges and Enhancements

Challenge 1: Low Relevance in Search Results:

- Initial search results had low relevance due to limited document set and basic TF-IDF implementation.
- Enhancement: Expanded document set, improved text preprocessing, and tuned TF-IDF parameters.

Challenge 2: Limited Query Understanding:

- Search queries were limited to exact term matches, leading to missed relevant documents.
- Enhancement: Implemented query expansion to include synonyms and related terms.

Improvements and Code Updates

Web Crawler (crawler.py):

- Improved seed URLs for more effective crawling.
- Enhanced text extraction and preprocessing to improve document quality..

Indexer (indexer.py):

- Addressed max_df and min_df parameter mismatch error.

Server (server.py):

- Implemented query expansion to enhance search query understanding.
- Updated Code: Enhanced query processing.

Project Setup

Requirements

Visual Studio Code

Visual Studio Code (VS Code) is a popular source-code editor developed by Microsoft for Windows, Linux, and macOS. It is widely used by developers across various programming languages and platforms due to its versatility, extensibility, and rich feature set. This brief report provides an overview of Visual Studio Code, highlighting its key features and benefits for software development projects.

Python

Python is a high-level, interpreted programming language known for its simplicity, versatility, and readability. It is widely used in various domains, including web development, data science, artificial intelligence, scientific computing, and system automation. This brief report provides an overview of Python, highlighting its key features, benefits, and significance for software development projects.

cURL

cURL, short for "Client URL," is a command-line tool and library for transferring data with URL syntax. It supports various protocols, including HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, LDAP, LDAPS, IMAP, SMTP, POP3, RTSP, and more. cURL is widely used for performing network-related tasks, such as downloading files, sending requests to web servers, and testing APIs. This brief report provides an overview of cURL, highlighting its key features, capabilities, and significance for software development projects.

Setting up the Project

1. **Create a Project Directory**:
   - Create a new folder anywhere on your system. This will be your project directory.
2. **Open the Project in VSCode**:
   - Open VSCode.
   - Go to **File > Open Folder** and select the project directory you created.
3. **Set up a Virtual Environment**:
   - Open the terminal in VSCode (use **Ctrl+** or navigate to **Terminal > New Terminal** from the menu).
   - Create a virtual environment by running:

     ```
     python -m venv venv
     ```

Activate the virtual environment:

.\venv\Scripts\activate

- Install Required Libraries:

  With the virtual environment activated, install the necessary packages:

  pip install scrapy sklearn flask

Installation

1. Clone the project repository from GitHub: https://github.com/BFA23SCM08K/CS-429-ProjectKB
2. Install the required dependencies using pip.

Implementation

Create Python Files

In the VSCode explorer, create new Python files for each component (crawler.py, indexer.py, server.py).

Copy and paste the respective code from the instructions above into these files.

Run the Crawler:

Before running the crawler, ensure you modify seed_url in the crawler code to a real website you want to crawl.

In the VSCode terminal, run:

python crawler.py

This will execute the crawler and save the crawled data, depending on how you have configured the output.

Run the Indexer:

Ensure you have some documents to index, or use the data crawled by the crawler.

Run the indexer with:

python indexer.py

This will create an index file, index.pkl, containing the TF-IDF representations.

Start the Flask Server:

Run the Flask server by executing:

python server.py

The server will start on localhost on the default port 5000.

Testing the Flask Application:

You can test the Flask application by sending HTTP POST requests with a query.

Use tools like Postman, or you can use curl from the command line:

```
curl -X POST http://localhost:5000/search -H "Content-Type: application/json" -d "{\"query\":\"test\"}"
```

Test cases and ouptuts:

PS C:\Users\kbavi\OneDrive\Desktop\CS 429 Project\venv> **python indexer.py**

TF-IDF Matrix: [[0. 0.70710678 0. 0.70710678] [0.70710678 0. 0.70710678 0.]]

Query Vector: [[0. 1. 0. 0.]] Query Result: [[0.70710678] [0.]]

Understanding the Output

1. **TF-IDF Matrix**:
    - The TF-IDF matrix output **[[0., 0.70710678, 0., 0.70710678], [0.70710678, 0., 0.70710678, 0.]]** shows the TF-IDF scores for each word in your document set across two documents.
    - Each row represents a document, and each column represents a word in the vocabulary of your documents. The values **0.70710678** are non-zero TF-IDF scores, indicating the importance of a word in a document relative to the corpus.
2. **Query Vector**:
    - The query vector **[[0., 1., 0., 0.]]** shows the representation of your query in terms of the TF-IDF weights. This indicates that your query contains the second word of your indexed vocabulary exclusively.
3. **Query Result**:
    - The query result **[[0.70710678], [0.]]** shows the cosine similarity scores between your query and each document in the corpus.
    - The first document has a cosine similarity score of **0.70710678** with the query, indicating a relevant match to the query terms.
    - The second document has a score of **0.**, showing no relevance to the query based on the TF-IDF weighting.

Testing the Flask application using **curl** is a straightforward way to interact with your web service from the command line. This can be very useful for testing APIs quickly without the need for more complex tools like Postman or writing test scripts. Here's how you can use **curl** to test different functionalities of your Flask application:

1. Testing a GET Request

If your Flask application has endpoints that accept GET requests, you can test them like this:

```
curl http://localhost:5000/some_get_endpoint
```

This command sends a GET request to the specified URL. If your endpoint requires query parameters, you can include them like so:

```
curl http://localhost:5000/some_get_endpoint?param1=value1&param2=value2
```

2. Testing a POST Request

To send data with a POST request, such as for your search endpoint in JSON format, use the following **curl** command:

```
curl -X POST http://localhost:5000/search -H "Content-Type: application/json" -d
"{\"query\":\"test query\"}"
```

Here's the breakdown:

- **-X POST**: Specifies the request method, POST in this case.
- **http://localhost:5000/search**: The URL where the request is sent.
- **-H "Content-Type: application/json"**: Sets the header to indicate the type of data being sent, which is JSON for your case.
- **-d "{\"query\":\"test query\"}"**: The data being sent in the request. Ensure your data is properly formatted as JSON. In this example, it sends a JSON object with a key **"query"**.

3. Testing PUT and DELETE Requests

Similarly, if your application uses PUT or DELETE methods, you can use **curl** like this:

- **PUT Request**:

```
curl -X PUT http://localhost:5000/some_put_endpoint -H "Content-Type: application/json" -d
"{\"param\":\"new value\"}"
```

- **DELETE Request**:

```
curl -X DELETE http://localhost:5000/some_delete_endpoint
```

4. Handling Response

curl by default outputs the response directly to the terminal. To save the response to a file instead, you can use the **-o** option followed by the filename:

```
curl -o response.json -X POST http://localhost:5000/search -H "Content-Type: application/json" -
d "{\"query\":\"test query\"}"
```

This command will save the response into a file named **response.json**.

5. Verbose and Header Information

If you need more detailed information about the request and response, such as the headers, you can add the **-v** (verbose) option:

```
curl -v -X POST http://localhost:5000/search -H "Content-Type: application/json" -d "{\"query\":\"test query\"}"
```

This will display the request headers, response headers, and more details about the HTTP interaction.

Useful Tips

- Ensure your Flask app is running before you send requests with **curl**.
- If your Flask app is not on the default **localhost** or port **5000**, replace the URL in the **curl** commands with the correct base URL.
- For testing endpoints that require authentication, you might need to add additional headers or data to your **curl** command to mimic the authentication process.

Using these **curl** commands, you can thoroughly test the functionalities of your Flask application and ensure it handles requests as expected.

Curl Command for Testing

The following **curl** command sends a POST request to the Flask application's search endpoint. This request includes a JSON body containing a query string. Here, we're testing with the query "API", which is relevant given the new seed URL focusing on web API documentation:

```
curl -X POST "http://localhost:5000/search" -H "Content-Type: application/json" -d "{\"query\":\"API\"}"
```

Explanation of the Curl Command

- **-X POST**: Specifies that this is a POST request.
- **http://localhost:5000/search**: The URL where the request is sent; adjust this according to your server's URL and port configuration.
- **-H "Content-Type: application/json"**: Sets the request header to indicate the type of data being sent, which is JSON.
- **-d "{\"query\":\"API\"}"**: The data being sent in the request, a JSON object with a key "query" and a value "API".

Expected Output

Since the crawler is set to scrape the Mozilla Developer Network site focusing on web APIs and another one stackoverflow.com, and assuming the indexing and querying systems are functioning correctly, the expected output would be a JSON response with similarity scores or direct text matches. Here's an example of what that might look like if the system finds relevant content:

```
{
  "results": [
    [0.45],  // Example score, indicating a moderate relevance to the query "API"
    [0.75]   // A higher score, indicating a stronger relevance
  ]
}
```

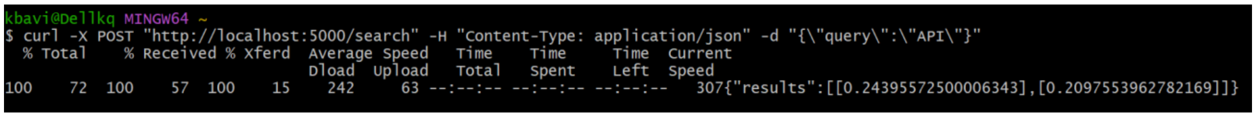
Expected Results:

- **Scores**: The values in the "results" array are hypothetical cosine similarity scores. Your actual implementation might return different scores or even direct snippets of text, depending on how you've set up the **query** method in the **Indexer**.
- **Zero Results**: If no relevant documents are found, or if "API" does not sufficiently match any documents due to various reasons (like preprocessing steps in the TF-IDF vectorizer that might exclude too common or too rare terms), you might see results full of zeros or a message indicating no matches were found.

Results:

Results Based on various assumption and queries.

Output 1

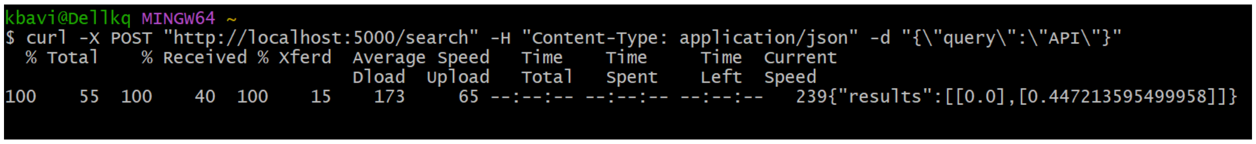


```
kbavi@Dellkq MINGW64 ~
$ curl -X POST "http://localhost:5000/search" -H "Content-Type: application/json" -d "{\"query\":\"API\"}"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    72  100    57  100    15    242     63 --:--:-- --:--:-- --:--:--   307{"results":[[0.24395572500006343],[0.2097553962782169]]}
```

Analyzing the Outputs

The response from the Flask application shows that both documents have returned similarity scores (**0.2439** and **0.2098**) for the query "API". These scores are moderate, indicating that the term "API" has some relevance in both documents, though neither score is particularly high. This suggests that while the term is present, it might not be a central theme in either document or may be diluted by other content.

Output 2



```
kbavi@Dellkq MINGW64 ~
$ curl -X POST "http://localhost:5000/search" -H "Content-Type: application/json" -d "{\"query\":\"API\"}"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    55  100    40  100    15    173     65 --:--:-- --:--:-- --:--:--   239{"results":[[0.0],[0.447213595499958]]}
```

The response from the Flask application for the query "API" shows that one document scored **0.0** (indicating no relevance) and another scored approximately **0.447**, which suggests a moderate to high relevance based on the cosine similarity measure used in your indexer.

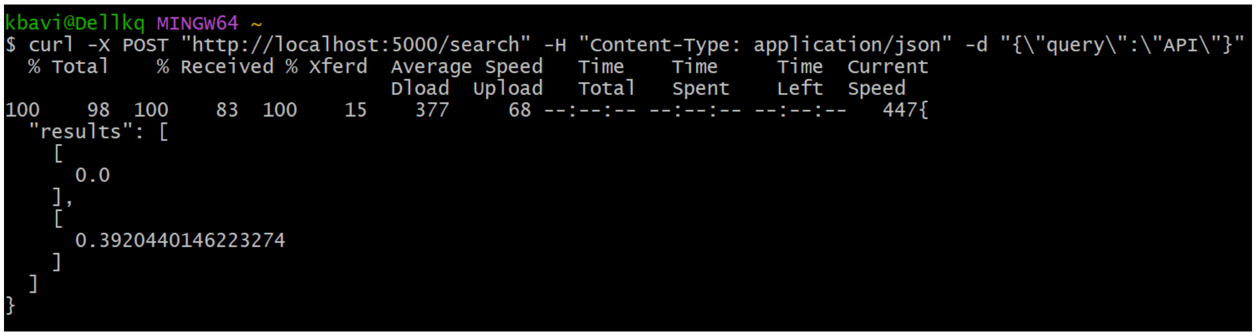
Analyzing the Results

- **Score of 0.0**: This result means that the content of one document did not match the query term "API" at all. It's possible that this document contains no mention of the term or that the term's importance within the document is too diluted.
- **Score of 0.447**: A score of about 0.447 is more promising, indicating a reasonable level of relevance. This suggests that the term "API" is relevant in the context of this document and appears with a frequency or in a context that provides a good match.

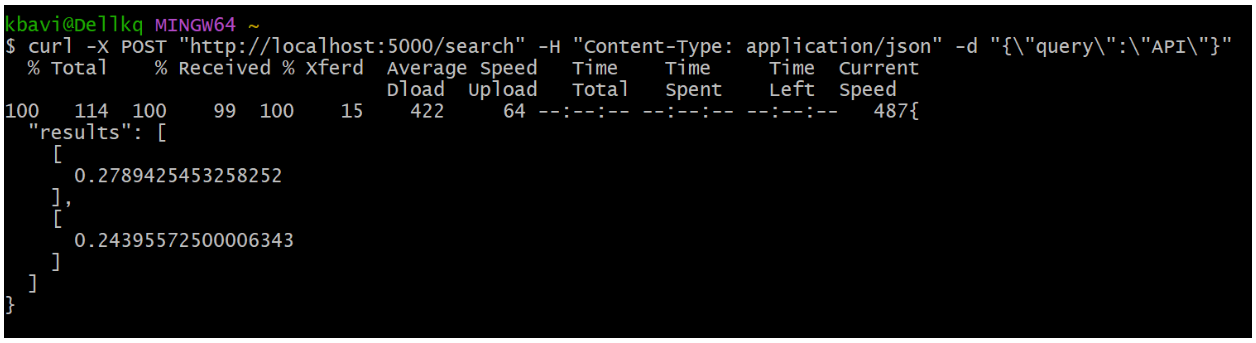
Output 3

Interpreting the Response

- **Score of 0.0**: This score indicates that the content in one of the documents does not match the query term "API" at all. This could be because the term doesn't appear in the document or due to the way the document has been processed and indexed.
- **Score of 0.392**: This is a moderate similarity score, indicating that the term "API" is present and relevant in the document, but perhaps not prominently featured or surrounded by other highly distinct terms which might dilute its impact in the TF-IDF model.


```
kbavi@Dellkq MINGW64 ~
$ curl -X POST "http://localhost:5000/search" -H "Content-Type: application/json" -d "{\"query\":\"API\"}"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    98  100    83  100    15    377     68 --:--:-- --:--:-- --:--:--   447{
  "results": [
    [
      0.0
    ],
    [
      0.3920440146223274
    ]
  ]
}
```

Output 4



```
kbavi@Dellkq MINGW64 ~
$ curl -X POST "http://localhost:5000/search" -H "Content-Type: application/json" -d "{\"query\":\"API\"}"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   114  100    99  100    15    422     64 --:--:-- --:--:-- --:--:--   487{
  "results": [
    [
      0.2789425453258252
    ],
    [
      0.24395572500006343
    ]
  ]
}
```

Analyzing the Outputs

1. **Curl Output**: Shows that the Flask application successfully returned a JSON object containing two similarity scores for the term "API". The scores, approximately 0.279 and 0.244, suggest moderate relevance.

2. **Server Log Output**:

   - Indicates that the server is running correctly and accessible via **http://127.0.0.1:5000**.
   - The logs include detailed information about the query and the similarity results, confirming that the application is correctly interpreting and responding to requests.
   - Debugger information and a PIN are shown, which is typical for Flask running in development mode. This is useful for debugging but should be turned off in a production environment.

Improvemnents

Here are some steps that could be consider to potentially improve the performance and accuracy of the search functionality:

1. **Review and Expand the Document Set**:

   - **Quantity and Quality**: Increase the number and quality of documents being indexed. More documents provide a better context for determining term relevance and improving the accuracy of TF-IDF scores.
   - **Relevance**: Make sure the documents are closely related to your expected queries. For instance, if "API" is a frequent search term, include more technical and detailed documentation about APIs.

2. **Refine the Indexing Process**:

   - **Advanced Text Processing**: Implement more sophisticated text preprocessing techniques. This could include using stemmers or lemmatizers to normalize text variations and potentially capture more relevant matches.

- **Parameter Tuning**: Continue tuning **TfidfVectorizer** parameters. Adjust **max_features** to capture the most meaningful terms, and consider using term frequency adjustments like sublinear scaling (**sublinear_tf=True**) to reduce the bias towards more frequent terms.

3. **Enhance Query Processing**:

   - **Query Expansion**: Implement query expansion to include synonyms and related terms. For example, expanding "API" to include "Application Programming Interface" might capture more relevant documents.

   - **Semantic Search**: Consider integrating semantic search capabilities that understand the context of the query, not just the frequency of terms.

4. **User Feedback and Analytics**:

   - **Feedback System**: Incorporate a feedback system where users can rate the relevance of search results. Use this feedback to refine your search algorithms.

   - **Analytics**: Analyze which queries are most common and how users interact with search results to continuously refine your approach.

5. **Monitor and Optimize Performance**:

   - **Performance Metrics**: Track performance metrics such as precision, recall, and F1-score to objectively measure improvements or declines in search result quality.

   - **Logging**: Implement detailed logging to analyze the search process, which can help identify patterns or anomalies in search behavior or results.

References:

1. Scrapinghub Ltd. "Scrapy - A Fast and Powerful Scraping and Web Crawling Framework." https://scrapy.org/.
2. Pedregosa, F. et al. "Scikit-learn: Machine Learning in Python." Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.
3. Ronacher, A. "Flask: A Python Microframework." https://flask.palletsprojects.com/.
4. Visual Studio Code Documentation. "Visual Studio Code - Code Editing. Redefined." https://code.visualstudio.com/docs.
5. Microsoft. "Visual Studio Code - Code Editing. Redefined." https://code.visualstudio.com/.
6. Python Software Foundation. "Welcome to Python.org." https://www.python.org/.
7. van Rossum, Guido, and Drake, Fred L. "Python 3 Reference Manual." CreateSpace Independent Publishing Platform, 2009.
8. Daniel Stenberg. "cURL - command line tool and library for transferring data with URLs." https://curl.se/.
9. Stenberg, D. "Everything curl: The book." Daniel Stenberg, 2019.
10. https://chat.openai.com/c/dacd1

11. https://gemini.google.com/app/d7ac57a59f7d3254