Rufus - Intelligent Web Scraper

This repository implements an intelligent web scrapper that takes in a query value amd relevant sites, scraps the site and return the relevant content based on the query for downstream tasks such as RAG.

Usage

You can install the package using the following command:

```
pip install -e .
```

After installing the package, you can use the following code to scrape the web and get the relevant content based on the query:

```
from Rufus import client

urls = ["https://lilianweng.github.io/posts/2023-06-23-agent/"]
queries = ["What is an agent in AI?"]

client = Client(urls, queries)
for i in range(len(urls)):
    docs = client.load_page(urls[i])
    splits = client.chunk_webpage(docs)
    extracted = client.extract_content(splits)
    client.store_data(extracted)
    extracted_data.append(extracted)

results = []
for query in queries:
    result = client.vectorstore.similarity_search(query, k=1)
    results.append(result)

print(results)
```

Approach and Explanation

Note: The following scraper make use of OpenAI GPT models, so in order to use the scraper, you need to have an OpenAI API key. You can take a look at quickstart guide here

- This client is designed with large-scale queries in mind. Most of the functions implemented in the `Client` class are capable of running asynchronously, ensuring efficient handling of extensive operations. Additionally, the underlying database is optimized to process large-scale queries seamlessly.
- The web loader leverages `playwright` to load and extract content from web pages. A key advantage of using `playwright` is its ability to simulate

a real user's experience, allowing it to load dynamic content that might not be accessible through simpler web scraping tools.

- Once a web page is loaded, we employ Large Language Models (LLMs) to filter out nested URLs. The LLMs analyze the content in relation to the query, identifying and extracting the most relevant nested URLs for further processing.
- After retrieving both the nested and original web pages, we recursively split the content into smaller chunks. This step ensures that LLMs maintain context throughout the content processing, leading to more accurate and relevant outputs.
- Following this segmentation, the content is passed through the LLMs to extract the most pertinent information. This approach makes the scraper resilient to dynamic changes in web pages, enhancing its ability to capture meaningful data from complex and evolving web environments.
- The extracted content is then stored in the database for future use. Finally, we apply a similarity search algorithm to identify the most relevant content based on the user's query, ensuring precise and efficient retrieval of information.

Challenges faced

- One of the primary challenges faced was handling of nested URLs and how to make a robust and effective scraper in context of dynamic web pages.
- Another challenge was to ensure that the scraper and the LLM could handle large corpus of text data effectively and efficiently.
- The scalibility of the system was also a challenge, as the system should be able to handle large-scale queries and process them in a timely manner.

Further Improvements

- The client can be further optimized to handle URLs more asynchronously, enhancing its ability to process large-scale queries efficiently.
- Error handling mechanisms can be implemented to address issues such as broken links or inaccessible web pages, ensuring the client's robustness in handling diverse web environments.
- Database could be further optimized to handle large corpus of text data by specifying the parameters of Hierarchical Navigable Small World(HNSW) algorithm which is commonly used for nearest neighbor search.