Rufus: Intelligent Web Scraping and Data Extraction

Link to repository : https://github.com/Nishi46/Rufus.git

Rufus is an AI-powered web scraping and data extraction tool designed to intelligently crawl websites and extract relevant content based on user-defined instructions. It is built to work seamlessly within Retrieval-Augmented Generation (RAG) pipelines, focusing on extracting the most relevant sections of a webpage (e.g., FAQs, pricing, customer support) while minimizing irrelevant data.

Features

- Intelligent Crawling: Rufus crawls websites based on user-defined prompts, ensuring only relevant pages are scraped.
- AI-Powered Data Extraction: It leverages natural language processing (NLP) models to interpret user instructions and extract the most relevant content (such as product details, FAQs, or pricing).
- Customizable: Users can define specific instructions to guide the web scraping process, making it flexible for various applications.
- Error Handling: Built-in error management to ensure smooth crawling and extraction processes, with helpful feedback on failures.
- Supports Structured and Unstructured Data: Can handle both structured (tables, lists) and unstructured data (text blocks, paragraphs).

How to use:

1. clone this repo - **https://github.com/Nishi46/Rufus.git**
2. `pip3 install -r requirements.txt`
3.

```
from Rufus.agent import Rufus
 import os

 key = os.getenv('RUFUS_API_KEY')
 client = Rufus(api_key=key)

 instructions = "Find information about product features and customer FAQs."
 url = "https://example.com"

 documents = client.scrape(url, instructions)
```

```
print(documents)
```

Example Usage:

```
from Rufus.agent import Rufus
import os

key = os.getenv('RUFUS_API_KEY')  # Assuming the API key is stored in environment variables
client = Rufus(api_key=key)

instructions = "We're making a chatbot for the HR in San Francisco."
url = "https://www.sf.gov"

documents = client.scrape(url, instructions)

print(documents)
```

Example Output:

```
{
"extracted_data":
 [
[ ....
...
],
[ ...
...
],
]
}
```

Advanced Features

- AI-Powered Instruction Parsing
  Rufus uses NLP models to interpret user instructions and extract context. For example, if a user requests “Find product pricing and customer support,” Rufus will focus on these sections of the page.

- Dynamic Web Crawling
  Rufus intelligently handles nested links and dynamic page structures, only following relevant links that align with user instructions, ensuring efficient data extraction.
- Error Handling
  Rufus includes robust error handling to manage issues during web scraping, such as request failures, parsing issues, or missing data, with clear error messages to guide troubleshooting.

Core Components:

- WebCrawler: Crawls web pages and retrieves raw HTML content.
- DataExtractor: Extracts relevant data using NLP to match user instructions.
- DataFormatter: Formats extracted data into a structured JSON file.

WebCrawler (crawler.py) :

- Crawl Method: Starts from a base URL, visits pages up to a specified depth and number of pages, collecting HTML content.
- Page Request with Retry: Makes HTTP requests with retry logic for failed page loads (up to 3 retries).
- Link Parsing: Extracts internal links from HTML to discover additional pages to crawl.
- Rate Limiting: Implements a delay between requests to avoid overloading the server.
- Error Handling: Logs errors and retries page requests in case of failures, raising exceptions if all retries fail.

DataExtractor (extractor.py) :

- Loads raw data, instructions, and initializes NLP model and PhraseMatcher.
- Phrase Matcher Setup: Extracts non-stop, non-punctuation words from instructions and prepares the PhraseMatcher to filter relevant content.
- Content Extraction: Processes raw HTML, applies NLP to instructions, and filters relevant lines based on semantic similarity or phrase matching.
- Content Filtering: Filters sentences by checking exact matches with prepared phrases or similarity with instructions (using a similarity threshold).
- Error Handling: Logs and raises errors if extraction fails, using a centralized error handler.

DataFormatter (formatter.py) :

- Takes the raw data and sets a hardcoded output file path (output_data.json).
- Data Formatting: Converts the data into pretty-printed JSON format using json.dumps().
- File Writing: Writes the formatted JSON data to the output file (output_data.json).

- Error Handling: Logs and raises errors if data formatting or file writing fails using a centralized error handler.

Agent (agent.py) :

- Initializes with an optional API key, falling back to an environment variable if not provided.
- Scrape Method:
    - Takes a URL and instructions for data extraction.
    - Uses WebCrawler to fetch raw HTML data from the URL.
    - Passes the raw data to DataExtractor for extracting relevant content based on the instructions.
    - Formats the extracted data into JSON using DataFormatter.
    - Returns the formatted data.

Error Handling (error_handler.py) :

The tool integrates an error handling system through the ErrorHandler class. It logs different types of errors (e.g., crawling errors, extraction errors, formatting errors) and provides detailed logs, including stack traces.

Challenges

- Handling Complex Web Structures: Crawling nested links and dynamic content was difficult, so I implemented a depth-limited crawling strategy and handled retries for network failures to ensure comprehensive and reliable data extraction.
- Extracting Relevant Data: Filtering content based on vague instructions was tricky, but I used spaCy for NLP-based semantic similarity to match the content to user instructions, ensuring relevant data extraction.
- Network Failures: Web scraping can encounter network issues, so I added a retry mechanism to handle page fetch failures and used a logging system to capture detailed error information for troubleshooting.
- Data Formatting Consistency: Ensuring uniform data output was a challenge, so I created a DataFormatter to structure extracted data into consistent JSON format for easy consumption.
- Balancing Performance and Accuracy: Managing large datasets while maintaining accuracy was difficult, so I introduced parameters to limit crawl depth and page count, along with a similarity threshold for filtering content.
- Scalability for Larger Projects: Scaling the system for large websites was a concern, but I focused on modular design to ensure components could be improved, optimized and reused independently for future performance and scalability.