ETL for SEO Watch : Data Pipeline and Analysis for Web Scraping

Anass Kemmoune

December 29, 2024

Supervised by :
Pr. Karima Echhabi
Pr. Anas Ait Aomar

Mohammed VI Polytechnic University - College of Computing

Contents

1 Introduction

This report describes the development of a web scraping and data analysis pipeline, which uses Python's `requests` library and `BeautifulSoup` for HTML parsing, as well as `multiprocessing` for parallel execution. The primary goal is to scrape content from multiple websites, clean the HTML text, remove common stopwords, and produce keyword statistics and visualizations (e.g., a word cloud).

2 AI Use in the Project

Artificial Intelligence tools played a supportive role in the development of this project. Specifically, ChatGPT was utilized to:

- Code the initial version of the Flask dashboard (User Interface).
- Refactor some of the Python functions and the LaTeX code of this report.

However, it is important to note that all main functionalities and core algorithms of the project were handcrafted to ensure tailored performance and adherence to project requirements.

3 Data Pipeline Architecture

3.1 Overview

The pipeline is composed of several sequential steps:

1. Reading URLs from a CSV file.
2. Crawling each website and its subpages up to a specified depth.
3. Extracting and cleaning textual data (removing HTML tags, punctuation, and stopwords).
4. Running the above operations in parallel to speed up scraping.
5. Aggregating the results and generating a word cloud and keyword frequency analysis.

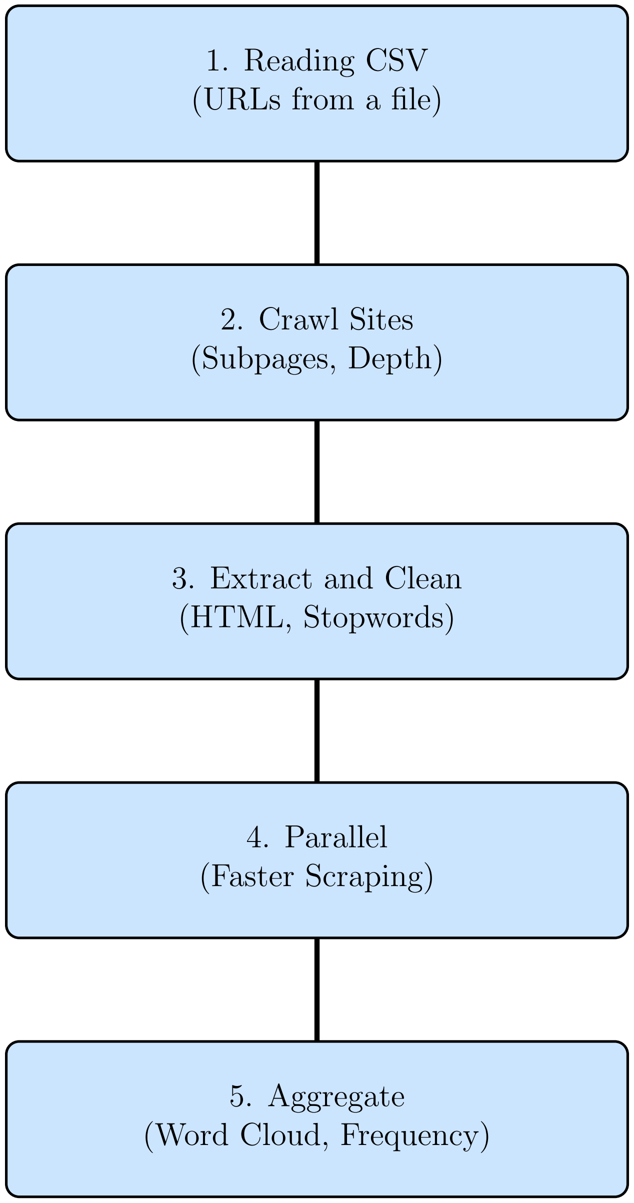

Figure 1: High-level architecture of the data pipeline.

3.2 Data Extraction

(a) Web Crawling: A tree traversal algorithm is implemented to systematically crawl and collect relevant web pages from an initial list of competitor websites. For each root URL, the crawler explores nested links up to a specified depth, ensuring that no page is visited more than once.

(b) Parallel Scraping Algorithm: To improve efficiency, a parallel scraping algorithm is designed where a dedicated process is assigned to each URL in the list. This approach ensures that:

- In each batch, no two pages from the *same* website are scraped simultaneously.
- This compliance respects the `robots.txt` guidelines by preventing excessive concurrent hits to the same domain.
- The distribution of work among processes is balanced so as to minimize idle time while respecting ethical scraping limits.

In practice, multiple URLs that belong to the same website are assigned to only one processor and will be scraped sequentially to ensure that no two sub-links are scraped concurrently.

3.3 Parallel Scraping Code Snippet

Below is a code snippet from the project illustrating how `crawl_websites` is implemented and parallelized:

```
from webCrawler import crawl_and_extract
from multiprocessing import Pool
import pandas as pd
import utils.countUtils as cu


def scrape_parallel(args):
    url, depth, max_sublinks = args
    result = crawl_and_extract(url, depth, max_sublinks)
    return result




def crawl_websites(websites, depth=1, max_sublinks=20, num_processes
   =4):
    """
    Parallel crawling function that accepts a list of URLs instead
       of reading from CSV.

    :param websites: list of URLs to crawl
    :param depth: how deep to follow sublinks
    :param max_sublinks: max number of sublinks to crawl
    :param num_processes: number of processes in the multiprocessing
        pool
    :return: list of results (each result is likely whatever
       crawl_and_extract returns)
    """


    # Prepare args for parallel processing
    args = [(url, depth, max_sublinks) for url in websites]
    # Run in parallel
    with Pool(num_processes) as pool:
        results = pool.map(scrape_parallel, args)


    # dicto = {results[i][0] : cu.list_to_count(results[i][1]) for i
        in range(len(results))}
    dicto = {"urls" :  [results[i][0] for i in range(len(results))]
         ,   "keywords" : [cu.list_to_count(results[i][1]) for i in
       range(len(results))]}
    df =  pd.DataFrame.from_dict(dicto)
    print(df)
    res = [results[i][1] for i in range(len(results))]
    df.to_csv("static/output.csv", index=False)
    print(f"Results saved to output.csv ")

    return res,df
```

3.4 Key Components

- **HTML Parsing and Cleaning:** Utilizes `BeautifulSoup` to remove unwanted tags and extract textual content, followed by Regex-based filtering and stopword removal using `NLTK`.
- **Duplicate URL Handling:** A `Link` class ensures that each URL is crawled only once, preventing repetitive work and loops.
- **Parallel Execution with Respect for Robots.txt:** The `crawl_websites` function uses `multiprocessing.Pool` to parallelize the crawling of multiple URLs, ensuring no two processes scrape the same domain simultaneously.
- **Data Aggregation and Analysis:** The scraped keywords are combined, then analyzed using frequency counts and visualized via a word cloud.

4 Data Analysis

Once the raw textual data from multiple websites has been collected and cleaned, the next steps focus on analyzing the tokenized text.

4.1 Keyword Extraction and Frequency

After cleaning and tokenization, each webpage yields a list of keywords. These keywords are aggregated into a single collection. A simple frequency analysis determines the most common words across all scraped pages.

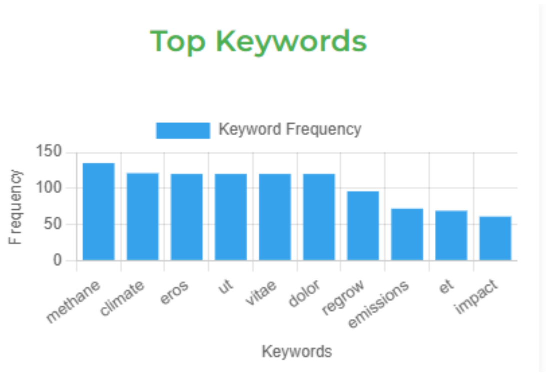

Figure 2: Keyword Extraction Screenshot 1

4.2 Word Cloud Generation

To visualize the distribution of the most frequent keywords, a word cloud is generated using the `WordCloud` library in Python. This visual representation helps identify dominant topics or frequently discussed terms across the scraped websites.

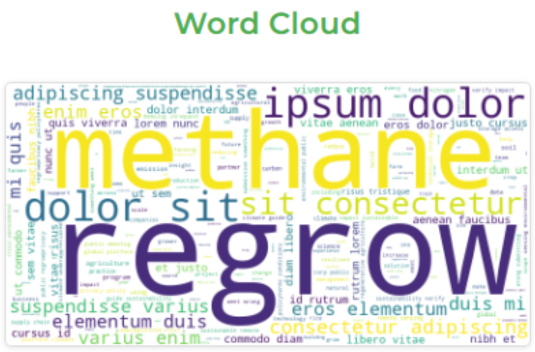

Figure 3: Word Cloud Generation Screenshot

4.3 Topic Modeling (LDA)

In addition to keyword frequency, we perform **topic modeling** using the **Latent Dirichlet Allocation (LDA)** algorithm. This technique groups the scraped text into a user-specified number of topics, each associated with a set of top words. This provides deeper insight into the thematic structure of the collected data. The `scikit-learn` library is used to vectorize the tokenized text, fit an LDA model, and extract representative words from each discovered topic.

5 Keyword Search in output.csv

After scraping finishes, results are saved into a CSV file named `output.csv` which has:

- A column `urls` indicating the original source URL.
- A column `keywords` storing a compressed dictionary of keyword counts. The dictionary is typically JSON-like, for example: {`"seo": 5, "python": 2`}.

For convenience, the keywords dictionary is **compressed** into a single string, which makes storage in CSV format straightforward. We then provide a **front-end search** where a user can type any keyword. If that keyword is found in the dictionary for a given URL, the web interface displays:

- The matching URL.
- The count of how many times that keyword appeared in that page.

This search is powered by **Papa Parse** on the client side to load the CSV, parse each row's `keywords` dictionary, and check for a user-specified keyword.

6 Challenges Faced & Solutions

6.1 HTML Parsing Complexities

Challenge: Real-world HTML often contains irregular or malformed structures, dynamic content (e.g., JavaScript), and embedded media tags (`script`, `style`, etc.).

Solution: `BeautifulSoup` is used to parse and navigate the DOM. Specific tags identified as "useless" (like `script`, `style`, `img`, etc.) are removed via the `decompose` function. This ensures that only meaningful textual content is analyzed.

6.2 Stopword Removal and Text Normalization

Challenge: Large quantities of "filler" words (e.g., "the," "an," "of") can distort keyword frequency results.

Solution: `NLTK`'s predefined English stopword list is used. Additionally, a Regex match ensures that only alphabetic tokens of length greater than one are retained. Token normalization includes converting words to lowercase and stripping whitespace.

6.3 Handling Inaccessible Websites and Errors

Challenge: Some URLs may be inaccessible (e.g., due to network errors, 404 statuses, or redirects).

Solution: A try-except block captures exceptions during the `requests.get` call. If a URL is inaccessible, it is skipped, and the crawler moves on. For HTTP to HTTPS transitions, the code attempts to correct the protocol when encountering certain status code errors (e.g., substituting `http` with `https`).

6.4 Ethical and Parallel Scraping (Respecting robots.txt)

Challenge: Running multiple processes can overload a single domain if multiple worker processes simultaneously scrape the same site. This potentially violates `robots.txt` guidelines.

Solution: The parallel algorithm staggers requests from the same domain across different batches. In practice, each process handles a different set of URLs, ensuring that no two processes scrape the same domain concurrently. This approach balances the workload while honoring rate limits and ethical scraping constraints.

7 Conclusion and Future Work

In this report, we described a web scraping pipeline that efficiently processes multiple URLs to extract textual data, remove noise, and perform keyword analysis. Key results include:

- Successful parallel crawling of multiple sites.
- Cleaned and normalized text data suitable for further natural language processing tasks.
- Visual insights into the most frequent keywords via word cloud generation.
- **Advanced Topic Modeling (LDA)** to reveal hidden themes in the collected text.
- **Convenient Keyword Search** on the CSV results, enabling quick lookups of specific terms across all URLs.

7.1 Possible Improvements

- **Advanced Text Analysis:** Incorporate sentiment analysis or named entity recognition for richer insights.
- **Improved Error Handling:** Enhance logic to handle various HTTP status codes and domain-specific constraints.
- **Scalability Enhancements:** Extend beyond simple multi-processing to distributed systems (e.g., Spark, Dask) for crawling very large numbers of URLs.
- **Dynamic robots.txt Compliance:** Integrate a dedicated `robots.txt` parser and scheduler that monitors and respects per-domain crawl-delay instructions.

References

[1] `requests` library documentation: https://docs.python-requests.org/

[2] `BeautifulSoup` documentation: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[3] `NLTK` documentation: https://www.nltk.org/

[4] `WordCloud` documentation: https://amueller.github.io/word_cloud/