Web Crawling and Keyword Analysis System: An ETL Pipeline for SEO Analysis

Submitted by:
Basma Arnaoui

Supervised by:
Dr. Karima ECHIHABI
Anas AIT AOMAR

College of Computing
Mohammed IV Polytechnic University
4th Year Engineering Student

Course: Data Management
Submission Date: December 29, 2024

Contents

1 Introduction

This report presents a comprehensive Extract, Transform, Load (ETL) pipeline designed for web crawling and keyword analysis. The system processes a dataset of 999 company websites, extracts their textual content through intelligent crawling mechanisms, and provides sophisticated analysis tools for keyword insights. The project combines parallel processing techniques with natural language processing to create a scalable and efficient SEO analysis platform.

A live demonstration of the system is available at: https://BasmaETLProject.pythonanywhere.com

2 System Architecture

2.1 Project Structure Overview

The project is organized to ensure modularity and maintainability. Below is an overview of the key directories and files:

- **backend/** – Core logic for data crawling, processing, and database management.
  - `clean_grouped_db.py` – Cleans and optimizes grouped data.
  - `crawler_utils.py` – Manages web crawling and scraping.
  - `db_utils.py` – Handles database operations and initialization.
  - `group_domains.py` – Aggregates data by domain for analysis.
  - `pipeline.py` – Coordinates the data pipeline process.
  - `text_cleaning.py` – Preprocesses scraped text data.
- **Database Files:**
  - `scraped_data.db` – Raw scraped data.
  - `cleaned_data.db` – Processed and cleaned data.
  - `grouped_data.db` – Aggregated data by domain.
  - `inaccessible_sites.db` – Logs inaccessible URLs.
- **templates/** – HTML templates for the web interface.
  - `index.html` – Frontend interface for keyword input and analysis.
- **app.py** – Flask application managing analysis requests and rendering results.
- **.gitignore** – Excludes unnecessary files from version control.

Design Rationale: This structure separates data collection, processing, and visualization. It promotes scalability, allowing different components to evolve independently.

2.2 Technologies Used

The project leverages a combination of modern technologies for efficient web crawling, data analysis, and visualization:

- **Python** – Core programming language for backend development.
- **Flask** – Lightweight web framework for building the application.
- **SQLite** – Database for storing and managing scraped data.
- **BeautifulSoup** – Extracts and parses HTML content from web pages.
- **NLTK** – Natural language processing for text tokenization and cleaning.
- **Plotly** – Generates interactive data visualizations.
- **Tailwind CSS** – Designs responsive and modern frontend interfaces.
- **Threading** – Enables concurrent web scraping for faster execution.
- **TF-IDF (TfidfVectorizer)** – Extracts important terms for keyword analysis.

2.3 Overview

The system architecture follows a modular design with distinct components handling different aspects of the ETL process. The pipeline consists of four main stages:

2.3.1 Stage 1: Data Extraction

The extraction phase reads company URLs from a CSV file and validates them by enforcing proper protocols (`http://` or `https://`). The process involves:

- **URL Validation:** URLs are corrected by testing server responses and adjusting protocols.
- **Domain Locks:** Locks are applied per domain to avoid simultaneous requests and ensure respectful crawling.
- **Parallel Crawling:** Multiple domains are scraped concurrently using a thread pool.
- **Depth and Link Limit:** Crawling is limited by depth and the number of links to prevent overloading.
- **Content Extraction:** HTML content is fetched and parsed into text using `BeautifulSoup`.
- **Error Logging:** Inaccessible URLs are logged for review, while scraped data is stored in the database.

2.3.2 Stage 2: Data Storage

The system employs multiple SQLite databases to organize and store data efficiently:

- **scraped_data.db:** Holds raw HTML content and extracted text from successfully crawled pages.
- **inaccessible_sites.db:** Logs URLs that couldn't be accessed, including error details and reasons for failure.
- **cleaned_data.db:** Contains processed text with HTML tags, special characters, and stop words removed.
- **grouped_data.db:** Aggregates and stores content by domain for streamlined analysis.

For URLs that repeatedly fail, future iterations plan to integrate Selenium to simulate browser interactions. This approach aims to bypass bot detection by mimicking real user behavior and loading JavaScript-heavy pages.

2.3.3 Stage 3: Data Transformation

The transformation phase involves cleaning and organizing the extracted data for analysis. Key steps include:

- **Text Cleaning:**
  - Remove non-alphanumeric characters and excess whitespace.
  - Convert all text to lowercase for uniformity.
  - Eliminate stop words (e.g., "the," "and") using the NLTK library.
  - Exclude common website-related terms (e.g., "www," "html," "index"," "contact", "us").
- **Data Cleaning Pipeline:**
  - Process raw text content from `scraped_data.db`.
  - Store cleaned text in a new database, `cleaned_data.db`, using a multithreaded approach for efficiency.
- **Domain Aggregation:**
  - Group cleaned text by domain using the `urlparse` library.
  - Store aggregated content in `grouped_data.db`.
- **Optimization:**
  - Remove short and irrelevant words.
  - Optimize grouped data by re-cleaning it to enhance the quality of the content for analysis.

The system is designed to handle large volumes of data efficiently, ensuring that processed text is well-structured and ready for further analysis.

2.3.4 Stage 4: Analysis and Visualization

The final stage provides interactive analysis tools through a Flask web interface, offering multiple visualization types for keyword analysis.

3 Implementation Details

3.1 Web Crawler Implementation

3.1.1 Crawler Architecture

The crawler implementation, found in crawler_utils.py, employs several sophisticated mechanisms:

Listing 1: Domain Lock Implementation

```
def acquire_domain_lock(domain):
    """Acquire a lock for a specific domain to prevent concurrent
       scraping."""
    with domain_locks_lock:
        if domain not in domain_locks:
            domain_locks[domain] = threading.Lock()
        lock = domain_locks[domain]
    lock.acquire()
```

This implementation ensures that no two threads can simultaneously crawl the same domain, preventing server overload while allowing parallel processing of different domains.

3.1.2 Protocol Handling

The system implements intelligent protocol handling to resolve HTTP/HTTPS confusion:

Listing 2: Protocol Resolution Logic

```
def force_protocol(url):
    """Ensure the URL starts with http:// or https://."""
    url = url.strip()
    if not url.startswith(('http://', 'https://')):
        url = 'http://' + url
    try:
        response = requests.get(url, timeout=5)
        if response.status_code != 200:
            url = re.sub(r'^http://', 'https://', url, count=1)
    except requests.exceptions.RequestException:
        url = re.sub(r'^http://', 'https://', url, count=1)
    return url
```

3.1.3 Parallel Processing

The crawler implements parallel processing using Python's ThreadPoolExecutor:

Listing 3: Parallel Crawling Implementation

```
def crawl_links_parallel(links, depth=1, max_links=20, max_workers
    =5):
    """Crawl and scrape multiple domains in parallel."""
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        futures = {executor.submit(crawl_and_scrape,
                   link, depth, max_links): link for link in links}
```

3.1.4 Database Structure

The system uses SQLite to manage both successful and failed web crawls. The database schema includes:

- **web_content:** Stores raw data from successful crawls.
  - **id** – Unique identifier.
  - **url** – URL of the crawled webpage.
  - **text_content** – Extracted text.
- **cleaned_data:** Contains processed text.
  - **url** – URL of the page.
  - **cleaned_text** – Cleaned and tokenized text.
- **domain_data:** Aggregates text by domain.
  - **domain** – Website domain.
  - **combined_text** – Combined text for each domain.
- **inaccessible_sites:** Tracks URLs that failed to load.
  - **id** – Unique identifier.
  - **url** – URL that could not be accessed.
  - **reason** – Reason for inaccessibility.

This structure ensures comprehensive tracking of both successful and failed crawling attempts, allowing for better monitoring and debugging.

4 Analysis System

4.1 Keyword Analysis Engine

The KeywordAnalyzer class implements sophisticated text analysis features:

Listing 4: Keyword Analysis Implementation

```
class KeywordAnalyzer:
    def analyze_keyword(self, keyword):
        results = {}
        results['domain_analysis'] = self.analyze_domains(df,
           keyword)
        results['related_terms'] = self.analyze_related_terms(df)
        results['context_analysis'] = self.analyze_context(df,
           keyword)
        results['network_analysis'] = self.analyze_network(df,
           keyword)
        results['location_analysis'] = self.analyze_location(df,
           keyword)
        return results
```

4.2 Overview of Keyword Analysis

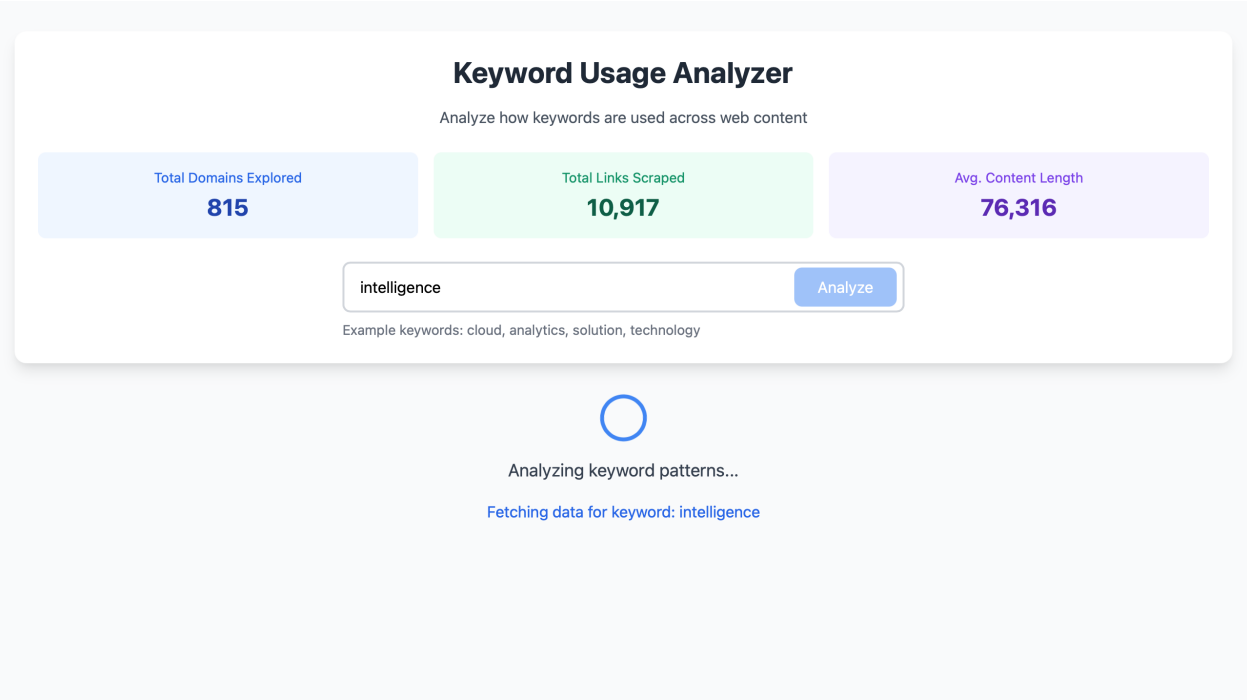

Figure 1: Keyword Usage Analyzer Dashboard: Summary of Web Crawling and Analysis Progress

The dashboard provides a snapshot of the keyword analysis process. It highlights:

- **Total Domains Explored:** The number of unique companies crawled.
- **Total Links Scraped:** The cumulative web pages extracted for analysis.
- **Average Content Length:** The typical size of the text data gathered from each page.

This interface reflects the preliminary results of web crawling, showcasing the scale and depth of data collection before deeper analysis, such as domain frequency and keyword distribution, is performed.

Note: The use of Plotly enhances interactivity, allowing users to dynamically explore keyword distribution. This visualization simplifies pattern detection and provides quick insights into domain relevance.

4.2.1 Domain Analysis

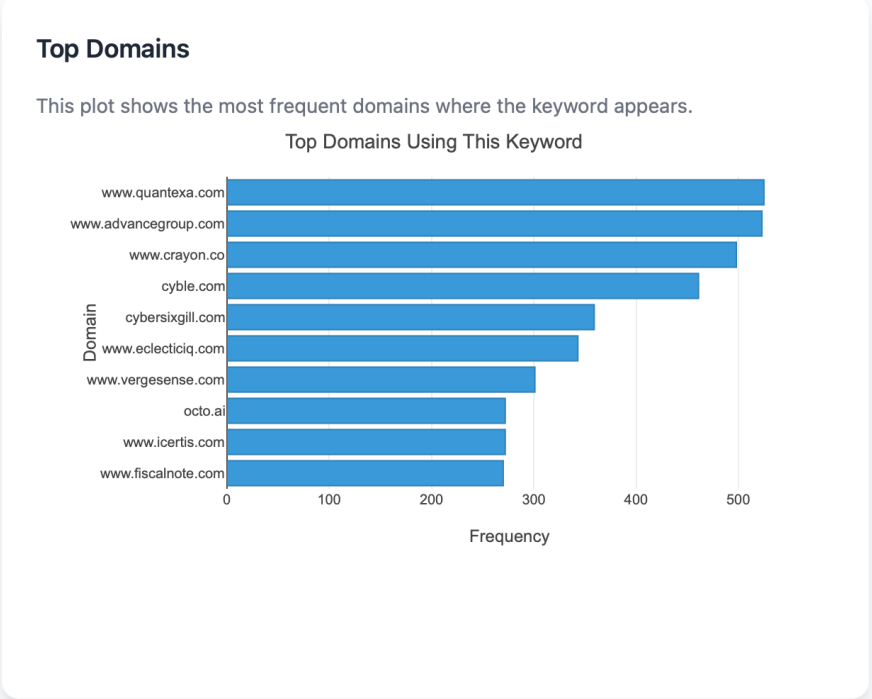

Figure 2: Visualization of Top 10 Domains by Keyword Matches

The domain analysis highlights the frequency of keyword occurrences across different websites. By aggregating keyword counts at the domain level, the system identifies which domains mention specific terms most frequently. This insight can reveal market leaders, industry trends, or competitive benchmarks.

Why It Matters: This analysis helps pinpoint influential domains and prioritize them for deeper investigation. Companies with high keyword density may serve as competitors, collaborators, or significant industry players.

Implementation Overview: The analysis:

- Counts keyword occurrences within each webpage's extracted text.
- Aggregates results by domain to compute the total keyword mentions.
- Visualizes the top 10 domains using interactive bar charts created with Plotly.

Listing 5: Domain Analysis Implementation

```
def analyze_domains(self, df, keyword):
    df['keyword_count'] = df['combined_text'].str.lower().str.count(
        r'\b' + re.escape(keyword) + r'\b'
    )
    domain_counts = df.groupby('domain')['keyword_count'].sum().
       sort_values(ascending=False).head(10)
```

4.2.2 Related Terms Analysis

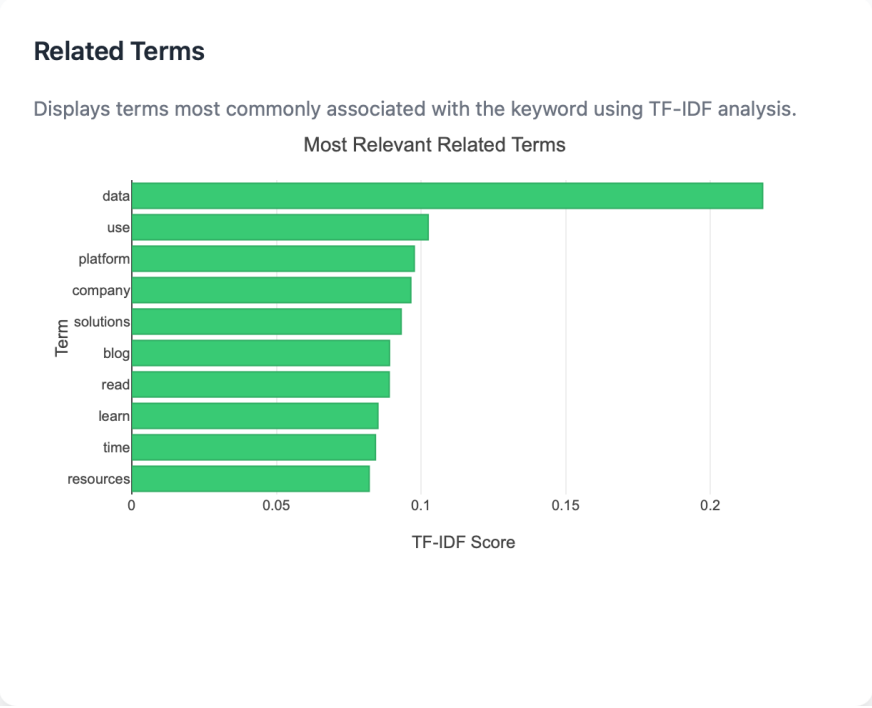

Figure 3: Visualization of Most Relevant Related Terms

The related terms analysis extracts and highlights key phrases commonly associated with the search keyword. Using TF-IDF (Term Frequency-Inverse Document Frequency), the system identifies terms that frequently appear near the keyword, filtering out irrelevant or overly common words.

Why It's Useful: This analysis helps reveal contextual patterns, showing how keywords relate to other terms across different web domains. It assists in understanding content trends and refining search strategies.

Key Process:

- Apply TF-IDF to identify the top 10 most relevant terms.
- Consider single words and two-word combinations (bigrams).
- Highlight terms that stand out in association with the keyword.

Listing 6: TF-IDF Implementation

```
def analyze_related_terms(self, df):
    vectorizer = TfidfVectorizer(
        max_features=100,
        stop_words='english',
        ngram_range=(1, 2)
    )
    tfidf_matrix = vectorizer.fit_transform(df['combined_text'])
```

TF-IDF's ability to downplay common words ensures the results focus on significant and informative terms linked to the keyword.

4.2.3 Context Analysis

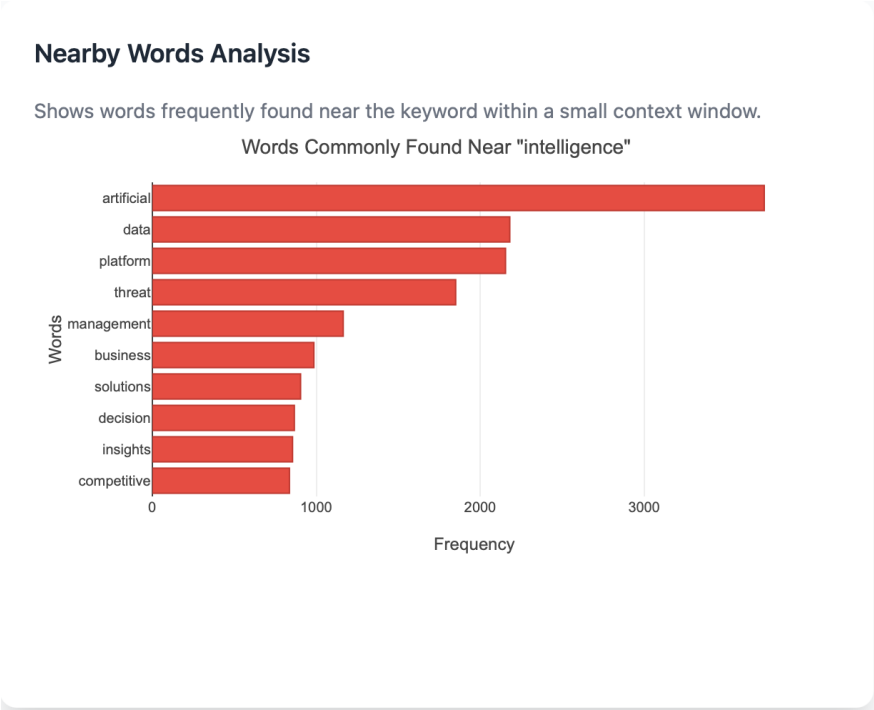

Figure 4: Visualization of Words Found Near the Keyword

The context analysis identifies words frequently appearing around the target keyword. By scanning within a five-word window, the system captures nearby terms, reflecting the contextual environment of the keyword.

Why It's Useful: This analysis uncovers the context in which keywords appear, offering insights into their associations and typical usage patterns.

Key Process:

- Tokenize text and search for keyword occurrences.
- Extract words within a five-word window around each instance.
- Visualize the top 10 most frequent neighboring words.

Listing 7: Context Analysis Implementation

```
def analyze_context(self, df, keyword):
    tokens = word_tokenize(' '.join(df['combined_text']).lower())
    window_size = 5
    for i, token in enumerate(tokens):
        if token == keyword.lower():
            context_words.extend(tokens[max(0, i-window_size): i+
               window_size+1])
```

This method reveals term relationships, shedding light on how keywords fit into broader web content.

4.2.4 Network Analysis

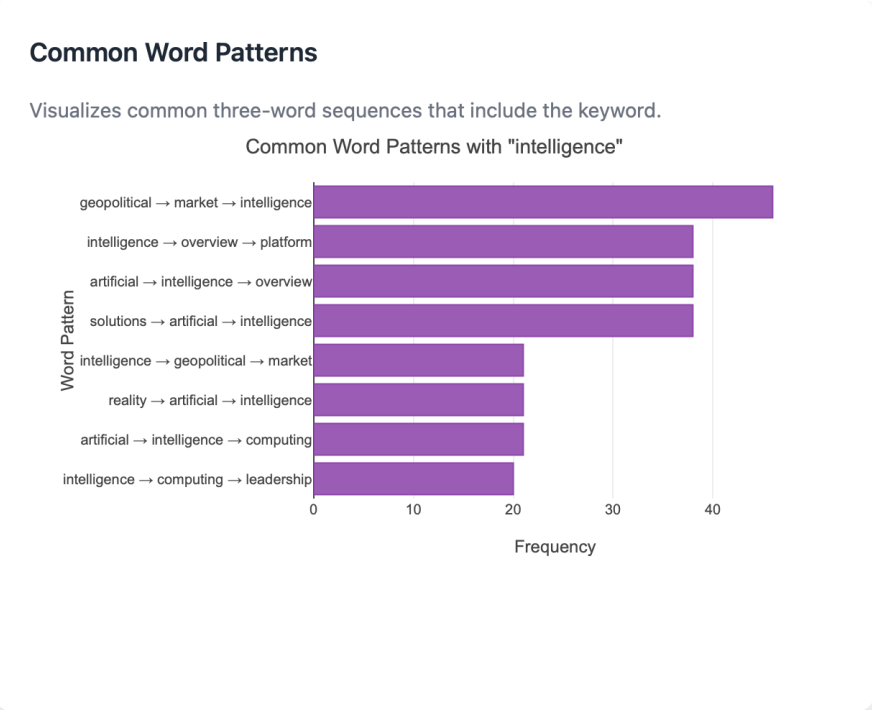

Figure 5: Visualization of Common Word Patterns (Trigrams)

The network analysis detects three-word sequences (trigrams) that include the keyword, providing insight into common phrases and usage patterns. This helps reveal how keywords are framed within larger textual structures.

Why It's Useful: Understanding frequent trigrams highlights recurring patterns and common expressions involving the keyword, offering valuable insights for content strategy and SEO.

Key Process:

- Tokenize text and extract trigrams.
- Filter trigrams that contain the keyword.
- Visualize the top 8 most frequent patterns.

Listing 8: Network Analysis Implementation

```
1 def analyze_network(self, df, keyword):
2     tokens = word_tokenize(' '.join(df['combined_text'].str.lower())
      )
3     keyword_trigrams = [tg for tg in ngrams(tokens, 3) if keyword.
      lower() in tg]
```

This analysis reveals how keywords are integrated into broader textual contexts, showcasing common word associations.

4.2.5 Location Analysis

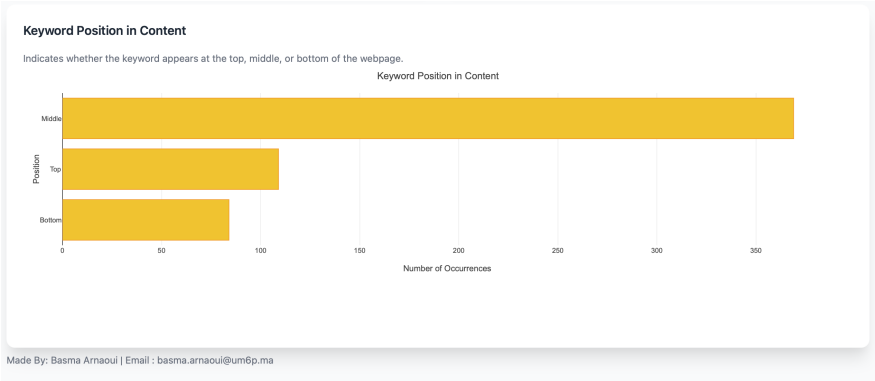

Figure 6: Visualization of Keyword Position in Content

The location analysis identifies where keywords appear within web content — categorizing occurrences at the top, middle, or bottom of the text. This provides insights into how prominently the keyword is featured.

Why It's Useful: Tracking keyword placement helps assess content structure and identify whether keywords are used for emphasis (at the top) or appear naturally throughout the text.

Key Process:

- Classify keyword occurrences by relative position (top, middle, bottom).
- Aggregate counts to visualize distribution.
- Highlight patterns that may affect SEO performance.

Listing 9: Location Analysis Implementation

```
def analyze_location(self, df, keyword):
    position_counts = df['position_category'].value_counts()
```

This analysis helps optimize keyword placement strategies for improved visibility and content relevance.

5 Web Interface

5.1 Frontend Implementation

The web interface is built using Flask and enhances user experience through:

- Real-time keyword analysis
- Interactive Plotly visualizations
- Responsive Tailwind CSS design
- Database statistics dashboard

5.2 API Implementation

The Flask application implements RESTful endpoints:

Listing 10: Flask Route Implementation

```
@app.route('/analyze', methods=['POST'])
def analyze():
    keyword = request.json.get('keyword', '').strip()
    results = analyzer.analyze_keyword(keyword)
    return jsonify(results)
```

6 Use of AI Tools

AI tools enhanced development by automating repetitive tasks and improving code quality:

- **Code Optimization:** Refined logic, reduced redundancy, and improved readability with clear comments.
- **Frontend Design:** Suggested responsive layouts using Tailwind CSS for better user experience.
- **Report Writing:** Assisted in structuring and refining documentation.

AI-driven suggestions accelerated progress, but all key decisions, architecture, and design were human-driven, reflecting my expertise and ensuring project integrity.

7 Deployment

The system is deployed on PythonAnywhere, offering:

- 24/7 accessibility
- Web-based access to analysis tools
- Testing without reliance on local machine configurations
- Simplified deployment process

8 Challenges and Solutions

8.1 Technical Challenges

Several technical challenges were encountered and resolved:

8.1.1 Concurrent Access

Challenge: Multiple threads attempting to access the same domain simultaneously. Solution: Implemented domain-level locking using threading.Lock().

8.1.2 Resource Management

Challenge: Potential for infinite crawling depth. Solution: Implemented depth limits and maximum sublink restrictions.

9 Future Improvements

To expand the system's capabilities, several enhancements are planned:

- **Selenium Integration:** Some websites block traditional crawlers by detecting automated requests. Due to time constraints, Selenium, which simulates real user behavior by interacting with JavaScript-rendered content, was not implemented. Adding Selenium will enable the system to access dynamic websites that rely heavily on client-side rendering.
- **Advanced NLP Techniques:** Implementing more sophisticated natural language processing (NLP) methods to improve text analysis accuracy.
- **Sentiment Analysis:** Extracting sentiment from web content to gain insights into tone and emotional context.

10 Conclusion

The implemented system successfully creates a comprehensive ETL pipeline for web content analysis. Through careful architecture decisions and robust implementation, it provides valuable insights for SEO analysis while maintaining efficiency and scalability.